Project Title:	**i-Treasures**: Intangible Treasures – Capturing the Intangible Cultural Heritage and Learning the Rare Know-How of Living Human Treasures
Contract No:	FP7-ICT-2011-9-600676
Instrument:	Large Scale Integrated Project (IP)
Thematic Priority:	ICT for access to cultural resources
Start of project:	1 February 2013
Duration:	48 months

D5.6

Final Version of the Text to Song Synthesis Module

Due date of deliverable:	30 November 2015
Actual submission date:	08 February 2016
Version:	1st version of D5.6
Main Authors:	Marius Cotescu (ACAPELA)

Project funded by the European Community under the 7th Framework Programme for Research and Technological Development.

Project ref. number	ICT-600676
Project title	i-Treasures - Intangible Treasures – Capturing the Intangible Cultural Heritage and Learning the Rare Know-How of Living Human Treasures

Deliverable title	Final Version of Text to Song Synthesis Module
Deliverable number	D5.6
Deliverable version	Version 1
Previous version(s)	
Contractual date of delivery	30 November 2015
Actual date of delivery	8 February 2016
Deliverable filename	D5.6
Nature of deliverable	P = Prototype
Dissemination level	PU = Public
Number of pages	27
Workpackage	5
Partner responsible	ACAPELA
Author(s)	Marius Cotescu
Editor	Marius Cotescu
EC Project Officer	Alina Senn

Abstract	This document reports on the activities that have been undertaken in the context of T5.3. More specifically, we report on the activities undertaken to finalize the Text-to-Song Module of the Integrated Platform. To this extent, we have built a Byzantine singing voice, and developed a novel pitch contour generation algorithm. We have also made efforts to find industrial partners to promote the developed technologies.
Keywords	Singing synthesis, Singing Corpus, Sardinian Corpus, Rare singing, Text-to-Speech

Signatures

Written by	**Responsibility- Company**	**Date**
Marius Cotescu	ACAPELA	12/01/2016
Verified by		
Athanasios Manitsaris	UOM	31/01/2016
Stephane Dupont	UMONS	31/01/2016
Sotiris Manitsaris	ARMINES	31/01/2016
Approved by		
Nikos Grammaliis	Coordinator (CERTH)	8/2/2016
Kosmas Dimitropoulos	Quality Manager (CERTH)	8/2/2016

Table of Contents

1. Executive summary

This document presents the development of the final version of the Text to Song (TTS) Synthesis module. The work was concentrated around building the resources for Byzantine singing and improving the techniques in the synthesis engine. In addition to the first iteration of the TTS module, we have developed a novel pitch generation model that uses natural transitions between notes.

We have also focused on opening the system to other researchers by providing an API for handling new languages, as well as pitch and phoneme duration. In order to help other researchers in their efforts to study the rare singing cases we have focused on, we have also chosen to release the corpora free for non-commercial and research use.

We have had contact with MuseScore, a company providing open-source score editing software, and are discussing the integration of our singing synthesis module in their products.

2. Introduction

As part of WP5, task T5.3 is meant to develop a Text to Song Synthesis module for rare singing styles that can support educational tasks. As the final deliverable of the task, this document summarizes the work and results of T5.3.

The general structure and first implementation of the Text-to-Song module were presented in D5.3. The software reads a musical score (in MusicXML format), and generates the according song, using information stored in a synthesis corpus. The module only supported Canto a Tenore, and had a number of drawbacks related to the quality of the output.

Concerning software and algorithm development, we have focused on implementing new pitch generation algorithms that are capable of producing accurate and expressive renditions of melodies. We have also worked on creating the resources for the Byzantine singing style. We have recorded a singing corpus in the Byzantine style, and segmented its phonetic and musical content. An additional musical segmentation has been produced for the Canto a Tenore corpus as well.

One of the reasons for implementing our own singing synthesis module was the fact that the currently available singing synthesis tools are not open to development of new languages, let alone core techniques as vocal tract synthesis or pitch generation. In order to address the problem, we have decided to provide an API on which researchers can develop new languages and pitch generation modules.

The document is structured into five sections. The first two are the Executive Summary, and the present Introduction. The next section describes the resources that have been built, mainly the Byzantine corpus and the musical segmentation. The fourth section describes the new pitch generation techniques that we have implemented, as well as the changes to the vocoder and the introduction of polyphonic score support. The fifth section describes the development tools, while the final section touches on the possible collaboration with MuseScore. We provide two Appendices, containing the manuals for the singing synthesis module, and its API, respectively.

3. Singing Corpora

Modern speech and singing synthesis systems rely on corpora to get the information that is going to be used for synthesis. A synthesis corpus needs to contain renditions of representative sounds and gestures that allow a synthesis system to either learn (in the case of parametric synthesis) or have available (in the case of unit selection systems) the required sounds. In D5.3 we have described the English and “Canto a Tenore” corpora. This section will describe the content and recording conditions of the Byzantine singing corpus.

3.1 Byzantine Corpus

Acapela already offers Greek voices in its synthesis products, as opposed to the “Canto a Tenore” sub-use-case, we did not need to do an in-depth linguistic study of the language. However, as older religious texts are involved in Byzantine singing, we did need to check the behavior of the NLP module. We have thus downloaded chants and liturgical texts from a site dedicated to conserving the Greek Orthodox and Byzantine religious texts [*1*]. We have then selected the most unstable transcriptions, and we asked the Byzantine experts to check them. Their input was used to fine-tune the NLP.

One of the difficulties in creating a good unit-selection synthesis corpus resides in being able to cover the phonetic and prosodic space of a language so that the available units fit as many of the situations that might appear during real use. In the case of singing, we have decided to approach the phonetic and prosodic aspects independently. We have thus designed the corpus in two parts. The first part covers as much of the phonetic space as possible, while a second one is focused on covering musical content.

The phonetic content of the corpus follows the well-established rules of building a speech synthesis corpus; so we were able to design it before the recordings. The musical part of the corpus was designed to serve a unit selection system based in musical information. The aim is to cover all possible note transitions over syllables that cover the entire phonetic space of the language. This is obviously a daunting and immense task. When adding the fact that the transitions need to be musically expressive, thus they would need to be sampled from songs that the singer is at least familiar with, thus restricting the usable texts, the task becomes even harder. For Byzantine music, we have to add the problem that it is hard to obtain musical scores in a digital format. However, we will present a number of simplifications that bring the problem in the realm of possible.

The first thing we have to consider is that we are only interested in the pitch values. We plan to transplant them to synthetic vocal tract parameters to generate the requested songs. It is expected that a singer is going to produce a very similar pitch contour independent of the sustained vowel he produces. For unvoiced sounds, the simplification is obvious. There is still the case of voiced consonants, and the fact that different classes of unvoiced sounds affect their voiced neighbors in different manners. Table 3-1 shows the phonetic classes of the Greek language and their representatives.

Regarding the musical space, we have adopted the following simplifications. First, any non-flat transitions can be stretched to match any required transition. Second, upward and downward transitions are considered equivalent. We can transform an upward transition into a downward transition by taking the derivative of the pitch contour, multiplying it by -1, and then integrating again to obtain the opposite transition.

Phonetic class	Phonemes
ORALVOWEL	a, e, i, o, u, aI, OI, aU, eI, IU, oU
FRICATIVE	f, v, s, z, D, T, C, x, G, S, Z
GLIDE	j, w
LIQUID	r, l, L
NASAL	n, m, J, N, M
UNVOICEDPLOSIVE	p, t, k, ts, ps, ks, tS
VOICEDPLOSIVE	b, d, g, c, gj, dz, dZ

Table 3-1. Greek phonetic classes.

Considering the limited volume of Byzantine chants and the natural redundancy of religious songs, they were not suitable to efficiently completely cover neither the phonetic, nor the musical space. We did use them as much as possible, though, in order to ensure the authenticity of the recordings. The rest of the material came from liturgical texts. In order to ensure precise pronunciation for all phonemes, we have also doubled the phonetic content using isolated words. The five resulting sets are presented in Table 3-2.

Subset	**Items**	**Contents**
grg_diphone_chants	84	Phonetically rich chants
grg_diphone_chants_increment	203	Phonetically rich liturgical texts
grg_diphone_rich	559	Phonetically rich isolated words
grg_syll_chants	213	Rhythmical coverage using Byzantine chants
grg_syll_chants_increment	511	Rhythmical coverage using liturgical texts

Table 3-2. Structure of the Byzantine corpus.

Once the corpus content was selected, we have asked the Byzantine experts to validate it, from a musical view. The corpus was recorded in Thessaloniki, Greece, between the 25th and 29th of May 2015. The phonetic segmentation of the corpus was performed using the automatic techniques described in D5.3.

3.2 Musical Segmentation

The rhythmical content in the corpus has to be complemented by information about the note realisations. Unfortunately, due to the lack of Byzantine musical scores in a digital format, we were not able to plan and script the musical script as well. In addition, we also did not have any prior information about the recorded melodies. From this point of view, both the Byzantine and the “Canto a Tenore” cases were equivalent. In the following section we will describe a method that automatically transcribes the melodic line of a recording.

3.2.1 Tuning

The first step is to tune the recording. We assume the relative jumps in a recording are accurate, but a recording may be out of tune. This is especially true for the

“Canto a Tenore” sub-use-case where the singing tradition is more informal and lacks the constraints of accompanying musical instruments. For simplicity, whenever we will refer to the pitch value, we will be referring to the base logarithm of the fundamental frequency. This makes all the processing much easier as the distance spanned by one octave is equal to 1.

In order to tune the recording, we quantize the pitch curve to a resolution of one semitone (1/12). By subtracting the mean quantization error, we get a pitch curve that follows the same melody, but is perfectly tuned to the scale.

3.2.2 Note detection

Starting from the tuned pitch we can search for the note sequence that will match the pitch contour. In order to keep a realistic tempo and resolution, we make the following assumptions. The length of a note cannot be less than the shortest syllable in the corpus. Each syllable is sung over at least one note. The note detection process is a series of note boundary movements, splits, and merges, as described in the following algorithm.

1. Create one note for each syllable in the recording, and assign it the median value of the quantized pitch
2. For each note, if the pitch error is greater than half a semitone and the duration is longer than the shortest syllable in the corpus, split it in half.
3. For each note boundary, look for the local maxima of the derivative in its vicinity. Move the boundary to the maximum that minimizes the error for both notes.
4. For each pair of adjacent notes, merge notes with the same height, if they are not spanning different syllables.
5. If the refinement threshold has not been reached, go back to step 2.

Once the correct note sequence has been inferred, the musical segmentation is saved to a file, along with the underlying phonetic content of each note.

3.3 Corpora Distribution

In order to encourage the future research into the studied sub-use-cases and to increase the visibility and lasting effects of the project, we have decided to make the rare singing corpora and resources ("Canto a Tenore" and Byzantine singing) freely available for non-commercial use. The corpora have been uploaded on the project website, and can be openly downloaded and used by anybody that is interested.

4. Synthesis Engine

A first iteration of the singing synthesis engine was implemented as part of deliverable D5.3, and a detailed description of the techniques can be found in Section 5 of the document. Two major problems of the first system were its lack of melodic expressivity and the strong degradation introduced by pitch modifications. Figure 4-1 shows the general structure of the Text to Song Synthesis module.

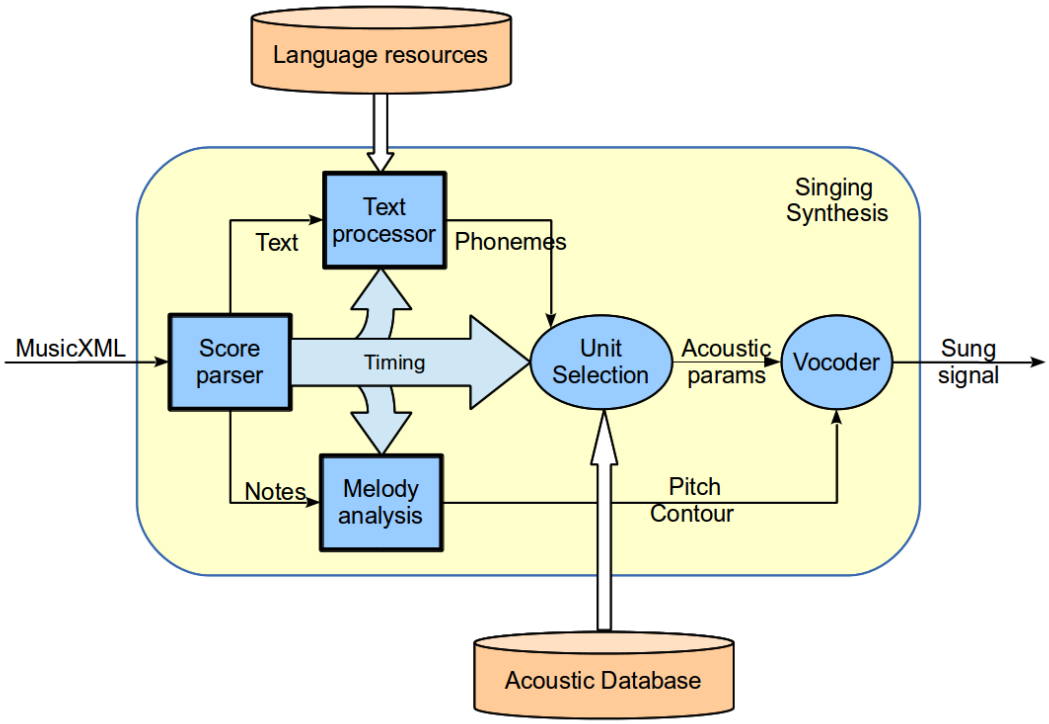

Figure 4-1. Internal structure of the Text to Song Synthesis module.

Another problem, which was raised in the reviewers' comments, is the need to implement a synthesis system from scratch, rather than use one of the systems readily available on the market. The decision was based on the fact that the currently available systems (Yamaha's Vocaloid [*2*], Orpheus [*3*], Symphonic Choir [*4*]), are closed to outside development. That means that it is impossible for third parties to use them for research purposes. In contrast we want our system to be open to researchers and developers. We are thus offering a compiled library and an API so that the F0 generation and duration control can be replaced by new code.

This section presents three major improvements that were implemented since the last iteration. The first one is a novel way of using chunks of natural pitch contours to create new melodic lines. Next, we will present the automatic support for in-score voice selection, and polyphonic output. And last we will discuss the integration of the STRAIGHT [*5*] vocoder with the synthesis engine.

4.1 Pitch Contour Algorithms

Pitch and duration are one of the most important and basic means of conveying musical expression. They are at the core of the interpretation of a musical piece. It not only needs to be exact in terms of note height and duration, but also pleasant, and emotional. The sub-use cases studied in the project have the particularity that

they are not very well formalised, often not having a score at all. Even when they do, a lot of the performance details are decided upon by the singer.

There are many pitch generation algorithms available. Most of them deploy a parametric approach, in which the contour is generated from scratch using the note and phonetic information only (see [*6*], [*7*], [*8*]). Another possible route is the unit-selection approach [*9*].

In order to capture the singer's knowledge and skill, we have decided to deploy non-parametric pitch generation techniques. This approach has the advantage of capturing the details and expertise of the recorded singers and ensuring it is present in the synthetic renditions.

4.1.1 Verse Selection

"Canto a Tenore" is a traditional singing style that doesn't use any scores for the transmission or encoding of the melodies. This makes computer-based generation of convincing songs of the style quite difficult. The style is bound by a few strict rules, though. One of them is that all verses are 11 syllables long. Furthermore, they use predefined melodic lines for the verses, and introduce small local variations to express emotions or decorate the melody.

This led us to try and copy the pitch contour of recorded verses over to the synthesized verses. In order to classify each of the recorded verses according to the melodic lines which were provided by the experts, we had to implement a simple musical segmentation algorithm. In order to achieve this, we assign a pitch value to each syllable, corresponding to the mean F0. The melody is considered to be the distance (in semitones) from each note in the verse, and to the first note. We assign the verse to the class whose melody matches best.

At synthesis time, then we select one of the contours available in the class required by the specified melody, and we search for the realisation that best matches the phonetic content of the required verse. The rhythm and pitch contours are then transplanted to the new synthetic vocal tract and source parameters.

The approach has two major advantages in that it can impose not only natural pitch movements to the synthetic signal, but also a natural rhythm. When the underlying phonetic content matches with the one that has been requested, the results are very good. The biggest draw-back is caused by the very large size of the selection unit. The phonetic variation, even after splitting the phones into classes is still huge, making complete coverage impossible. Furthermore, we haven't planned for this approach, and thus the coverage is not optimal. It is thus very easy for the system to go out of its domain, where quality suffers greatly dew to the mismatch in underlying phonetic structure.

4.1.2 Di-Note Selection

In the previous implementation of the singing synthesis engine, we have used the classic Fujisaki model of F0 control [*10*] to generate a smooth F0 contour from the pitch commands encoded in the musical score. Even though the model is able to generate satisfactory pitch contours for popular singing, it was failing to create convincing contours for the "Canto a Tenore" sub-use-case due to the lack of appropriate scores. Even in Byzantine singing, where the scores can be quite rich and exact, using real realisations of the note transitions is very useful.

In order to address the problem, we have decided to use a unit selection approach to the problem. Previous attempts have been made [*9*], but we propose a more detailed and refined version, introducing phonetic data into the selection process. We have

implemented a unit selection algorithm in which the smallest unit is the “dinote”. A dinote is defined as the transition between two notes. It stretches from the middle of the first note to the middle of the following one.

The synthesis algorithm starts by parsing the score to generate note descriptors. In addition to the musical information, the descriptors also include information about the phonemes that the note is spanning. A complete list of the di-note parameters is presented in Table 4-1.

Parameter	Description
$F_0[n]$	The pitch value for frame ***n***
$\Delta F_0[n]$	The value of the pitch derivative for frame ***n***
S_k^{seg}	The start frame of note ***k*** in segment ***seg***
E_k^{seg}	The end frame of note ***k*** in segment ***seg***
$\overline{F_0}^{(seg,k)} = \frac{\sum_{i=S_k^{seg}}^{E_k^{seg}} F_0[i]}{E_k^{seg} - S_k^{seg}}$	The mean pitch value of note ***k*** in segment ***seg***
$\overline{\epsilon_{F_0}}^{(seg,k)} = \frac{\sum_{i=S_k^{seg}}^{E_k^{seg}} \left\| F_0[i] - \overline{F_0}^{(seg,k)} \right\|}{E_k^{seg} - S_k^{seg}}$	The mean pitch value of note ***k*** in segment ***seg***
$\overline{\Delta F_0}^{(seg,k)} = \frac{\sum_{i=S_k^{seg}}^{E_k^{seg}} \Delta F_0[i]}{E_k^{seg} - S_k^{seg}}$	The mean of the pitch derivative of note ***k*** in segment ***seg***
${N_{ph}}^s$	Number of phonemes spanned by segment ***s***.
$P^s = \{{p_1}^s, \dots, {p_{N_{ph}}}^s\}$	The list of phonemes spanned by segment ***s***.
$F^{ph} = [{f_1}^{ph}, \dots, {f_m}^{ph}]$, ${f_i}^{ph} \in \{0,1\}$	Phonetic features vector of phoneme ***ph.***
$D^s = [{d_1}^s, \dots, {d_{N_{ph}}}^s]$	The durations of phonemes spanned by segment ***s***.

Table 4-1. Segment parameters and measures.

At synthesis time, the algorithm first retrieves all the candidates that are available in the database for each dinote required by the score. The next step is to prune the number of candidates by thresholding the target score. The target score is defined by equation (1) as the weighed sum of the components in Table 4-2.

$$C_T(c,t) = \sum_{s \in \{D,M,P,T\}} W_s \cdot d_s^T(c,t) \qquad (1)$$

Target cost component	Description
$d_M^T(c,t) = \sum_{k=1}^{2} \left\| \overline{F_0}^{(c,k)} - \overline{F_0}^{(t,k)} \right\|$	The absolute musical distance: the difference (in semitones) between the mean F0 of the candidate ***c*** and the required segment ***s***

$d_T^T(c,t) = \left\|\left\|\overline{F_0}^{(t,k)} - \overline{F_0}^{(t,k)}\right\| - \left\|\overline{F_0}^{(c,1)} - \overline{F_0}^{(c,2)}\right\|\right\|$	The transition distance: the difference (in semitones) between the required transition and the available ones. It's worth mentioning here that if the candidate does not follow required direction (upward, downward, or flat), it's dropped by default.
$d_P^T(c,t) = \sum_{i=1}^{{N_{ph}}^c} \left\|{F^{p_i}}^c - {F^{p_i}}^t\right\|^2$	The phonetic distance: the candidates are required to span the same number of phonemes as the target unit. If one of the phonemes of the candidate or target is a vowel, and it's not mirrored in the target, or candidate, respectively, the candidate is dropped. If the eliminating criteria have been passed, the distance between the candidate and the target is considered to be the total count of phonetic class mismatches.
$d_D^T(c,t) = \sum_{i=1}^{{N_{ph}}^c} \left\|{d_i}^c - {d_i}^t\right\|$	The duration distance: the difference between the durations of the requested notes and the realisations in the candidate.

Table 4-2. Target cost components.

After the number of candidates has been pruned, and each of them has been assigned a target cost, dynamic programming is applied to search for the minimum cost path through the units. The cost of a path is given by the sum of the target cost of each unit in the path and the concatenation cost between each two successive units in the path.

$$C_P = \sum_{i=1}^{N} C_T(c_i, t_i) + \sum_{i=1}^{N-1} C_C(c_i, c_{i+1}) \quad (2)$$

The target score C_C is defined by equation (3) as the weighed sum of the components in Table 4-3.

$$C_C(c_i, c_{i+1}) = \sum_{s \in \{D,E,M,P\}} W_s \cdot d_s^C(c_i, c_{i+1}) \quad (3)$$

Concatenation cost component	Description
$d_M^C(c_i, c_{i+1}) = \left\|\overline{F_0}^{(c_i,2)} - \overline{F_0}^{(c_{i+1},1)}\right\|$	The musical distance: the difference (in semitones) between the second note of the first dinote, and the first note of the following dinote.
$d_D^C(c_i, c_{i+1}) = \left\|\overline{\Delta F_0}^{(c_i,2)} - \overline{\Delta F_0}^{(c_{i+1},1)}\right\|$	The dynamic mismatch: defined as the difference in the mean derivative of the pitch over the two touching notes.
$d_E^C(c_i, c_{i+1}) = \left\|\overline{\epsilon_{F_0}}^{(c_i,2)} - \overline{\epsilon_{F_0}}^{(c_{i+1},1)}\right\|$	The pitch error: defined as the difference between the mean pitch error over two touching notes
$d_P^C(c_i, c_{i+1}) = \left\|F^{{p_{N_{ph}}}^{c_i}{}^{c_i}} - F^{{p_1}^{c_{i+1}}}\right\|^2$	The phonetic distance: the distance between the last phoneme in the first

	dinote and the first phoneme in the following dinote.

Table 4-3. Concatenation cost components.

Once the optimum sequence of pitch contours has been chosen, the contours are stretched to match the durations of the underlying phonemes, and then blended. The informal results show a significant increase in the naturalness of Byzantine and "Canto a Tenore" synthetic samples. We could not apply the technique to the English voice, as the corpus has been recorded using flat pitch.

4.2 Polyphonic Support

In order to make the use of the Text-to-Song module as easy as possible for choir and polyphonic scores, we have proposed two simple solutions. The first implies that the composer knows the available voices and names the parts accordingly.

If the score is prepared with a more general public in mind, or the score was not prepared for TTS altogether, a voice casting system is put in place using the configuration file. Thus the user can specify which part gets assigned to which voice. In addition, the user can also control the volume of each voice. The full description of the configuration field is described in Appendix 1.

4.3 Vocoder

In D5.3 we have presented the DSM vocoder [*11*], which we have used successfully for speech applications, and we have also proposed for the singing system. We have obtained good results with it, but some problems were reported for the "Canto a Tenore" style. In order to make the system more flexible for other developers and to take advantage of the state of the art STRAIGHT [*5*] vocoder we have implemented and integrated a spectral vocoder.

5. Development Tools

As the pitch contour and duration are very important aspects in the expressivity of singing, we have decided to create a platform on which researchers and developers can build their own solutions. Furthermore, we have decided to allow developers to connect their own NLP modules, to allow for easy integration of new languages. The decision is also inspired by our experience with the currently available software, which is completely closed, and doesn't allow third party development. We believe that this decision will both increase the exposure that the project is going to get, as well as encourage further research into the rare singing cases that we have studied.

5.1 Pitch Generation Tools

We have decided to provide two mechanisms to the interested developers and researchers. The first is an online mechanism, based on an API and a interchangeable shared library. The definition of the API and instructions on how to use the mechanism are presented in Appendix 2.

In addition, we also offer a simpler offline mechanism, where the user can dump the vocoder parameters, change the pitch contour using their own algorithm, then generate the waveform using their own pitch contour. This approach has the advantage that the developers don't need to integrate everything into a C++ library, and they can run quick experiments using this mechanism.

5.2 Custom NLP

One of the biggest problems we have confronted in using existing Singing Synthesis Software for the rare singing cases was complete inflexibility to adding new languages. In order to allow researchers to study any singing style using our software, we have decided to implement a simple API that would allow them to plug in their own NLP solutions into the singing synthesis system. The structure of the API is described in Appendix 2.

6. Industry interest

In addition to the research activities described in the previous sections, Acapela has had contact with MuseScore, a software company that provides an open-source editor. They were interested in providing a cloud-based Text-to-Song solution for their paying users. We have provided them with samples of the current capabilities of our technology, and are currently waiting for them to finish their internal and market evaluation.

Regardless of their answer, we are looking into developing a plugin that would allow MuseScore users to synthesize songs for Byzantine and “Canto a Tenore” singing styles free of charge.

7. Appendix 1: Software Manual

The Text-to-Song software is distributed as an archive containing Linux binary files and the required linguistic and acoustic databases. Additional databases might be added after installation to support new languages or voices.

7.1 Software Installation

The software is distributed as a Linux compiled binary, and can be downloaded from Acapela's website, after obtaining due permission.

To install the software, simply extract the archive content to the path of your choice.

7.2 Hardware and Software Requirements

7.2.1 Hardware Requirements

The software has been designed to run on the following minimal configuration

SVS Technology Requirements	
Processor	Intel i3
Memory	Depending on the duration of the song, and the vocoder type might go up to ~5 MB / s
Free Disk Space (per voice)	May vary between ~400 MB for backing voices to ~6 GB for full voices

7.2.2 Required Libraries

The binary requires that ***libxml*** version 2.9.1, or later, is installed on the system.

7.2.3 Supported Operating Systems

The software is currently only available for Linux.

7.3 Running the program

The software parses a musical score to extract the text and corresponding notes. In order to retrieve the phonetic information from the text, the software relies on information provided by the ***NLP database***. The desired signal is then reconstructed using the mapping between the phonetic symbols and the acoustic features found in the acoustic database. The mapping is read from the ***segmentation file***.

7.3.1 The Musical Score

The software expects to receive a musical score in MusicXML format. There are a number of commercially available score editors that can create MusicXML files (e.g. Sibelius and Finale), as well as a couple of open-source editors (MuseScore).

The input MusicXML file, should respect the following rules concerning the vocal part:

- All notes should be assigned to a syllable
- There should be no punctuation inside the lyrics

- The breaks will be clearly marked in the score by a pause of appropriate length

7.3.2 The Acoustic Database

The acoustic database consists of a number of files, organized in a relevant folder structure, containing extracted acoustical features, and an index file. The index file contains information about the structure of database, as well as the phonetic segmentation of the recordings.

7.3.3 The Configuration File

The configuration file contains the essential information needed by the system. The description of the variables is given in the table below. You can also find a configuration file example in the distribution folder. Please note that all the resource locations are required; there is no default value for them.

Resource Locations		
Variable	**Default Value**	**Description**
NLP_PATH	n/a	The absolute path to the NLP database
NLP_INI	n/a	The path to the NPL .ini file
SEG_FILE	n/a	Path to the segmentation file for the desired voice
MSEG_FILE	n/a	Path to the musical segmentation file for the desired voice
ACOUSTIC_STATS	n/a	File providing statistics of the acoustic data
SYNTHESIS_STATS	n/a	File providing statistics of the acoustic data
LANGUAGE	n/a	Language of the lyrics
Acoustic Data		
Variable	**Default Value**	**Description**
FPERIOD	5	The sampling period of the features (ms)
SAMPRATE	48000	The sampling rate of the original signal (Hz)
LAR_ORD	21	Order of the spectral features used for the concatenation cost
FFTLEN	1024	Number of samples used to compute the Discrete Fourier Transform
C_WEIGHT	10	Weight applied to the concatenation cost
T_WEIGHT	1	Weight applied to the target cost
SPEC_ORD	51	Order of the spectral features used for actual synthesis

7.3.4 Synthesizing Songs

Once all the resources are ready, the waveform can be generated by calling the shell script ***generate.sh***.

./generate.sh –v tenore –b student_1

The simple command listed above uses the ***tenore.conf*** configuration file to generate the song contained by the score contained in ***tts_scores/student_1.xml***. The waveform is saved in ***tts_scores/student_1.wav***. A more detailed description of the command line options of ***generate.sh*** is presented below.

-b b	song basename	[]
-d d	working folder	[tts_scores]
-f f	Octave offset	[0]
-v v	Database name	[tenore]
-t t	Score tempo	[120]
-p	Polyphonic mode	[true]
-P P	PHN segmentation file for forced duration	[]
-R R	Wavefile corresponding to the forced duration segmentation	[]
-c	Clean temporary files	[true]
-C	Compress waveform in MP3 format	[true]
-V	Verbose mode	[false]

Table 7-1. Detailed options for ***generate.sh***.

8. Appendix 2: NLP and Pitch Generation API

One of the reasons why we have decided to implement our own system for this task is the lack of an open singing synthesis system on the market. Even if there are a few available commercial systems, they are closed to development, both in regard to adding new languages, as well as developing new and more transparent signal processing and pitch generation techniques. In order to respond to the need of other researchers for a flexible and open development platform, and to prolong the sustainability and exposure of our results, we have decided to offer an API that would allow researchers and developers to use our techniques for any language, as well as implement their own pitch generation and duration handling algorithms. This section provides a description of the data structures, classes and functions contained by the API.

8.1 Creating Your Own Custom F0 Generator

In order to be able to implement custom pitch generation code, we provide a simple C++ API. The user will need to write a new class, called ***CustomF0***, based on the ***F0*** class.

In order to be able to use a custom pitch generator, you will need to specify it in the configuration file, by setting the ***F0_MODEL*** property to **CUSTOM**. When in custom mode, the program will call the ***getCustomF0*** function to retrieve an instance of the ***F0*** class. The new code will have to implement two abstract functions: ***buildF0***, and ***getF0***. These functions will be called by the program to build the internal representation of the pitch contour, and to retrieve a copy of it, respectively.

The program expects to find the compiled code for the custom pitch generation in a shared library called *customF0.so*.

8.2 Creating Your Own NLP Module

Just like in the case of the custom pitch generator, the developer needs to we provide an abstract base class, called ***NLProcessor***, which can be derived to implement the desired behaviour. The program will call the ***getNLP*** function to retrieve an instance of the ***NLProcessor*** class. It will then call the class' ***parsePhrase*** method to retrieve the phonetic transcription of the phrase as a list of ***Phoneme*** objects. The software expects to find the compiled code for the custom NLP in a shared library called *customNLP.so*.

In order to trigger the use of the use of a custom NLP module, the ***LANGUAGE*** variable in the configuration file has to be set to a value that is not handled by the provided binary. The easiest way to ensure this is to prefix the language code.

8.3 The API

8.3.1 config_t structure

This is an internal structure that is populated with all the entries found when parsing the configuration file that are in a correct ***PARAM = PARAM_VALUE*** format, even if they are not supported by the original application. This is very useful if you need to provide additional parameters to your custom code, that are not handled by the original application.

Member name	Type	Description
params	char**	Pointer to the list of parameter names
values	char**	Pointer to the list of values (matching with the parameter entries)
n	int	The number of parameter / value pairs

8.3.2 phrase_t structure

This is the application internal properties structure. It contains the data returned by the MusicXML parser.

Member name	Type	Description
thePhrase	char*	String containing the phrase text, as recomposed from the score.
myWords	word_t**	List of the words in the phrase
nbrOfWords	int	the number of words in the phrase
measures	measure_t**	List of the measures spanned by the phrase
nbrOfMeasures	int	Number of measures spanned by the phrase
fWord	word_t*	Pointer to the first word in the phrase
cWord	word_t*	Pointer to the current word in the phrase (used for iteration)

8.3.3 word_t structure

This is the structure managing the words in a phrase.

Member name	Type	Description
phrase	phrase_t*	Pointer to the parent phrase
theWord	char*	String containing the word, as reconstructed from the score
mySyllables	syllable_t**	Pointer to a list of pointers to the syllables in the
nbrOfSyllables	int	The number of syllables in the word (according to the score)
pWord	word_t*	Pointer to the previous word in the phrase
nWord	word_t*	Pointer to the next word in the phrase
fSyll	syllable_t*	Pointer to the first syllable in the word
cSyll	syllable_t*	Pointer to the current syllable in the word (used for iteration)

8.3.4 syllable_t structure

This is the structure managing the syllables in a word.

Member name	Type	Description
word	word_t*	Pointer to the parent word
theSyllable	char*	String containing the syllable, as read from the score
notes	note_t**	Pointer to a list of notes spanned by the syllable
nbr_notes	int	The number of notes spanned by the syllable
pSyll	syllable_t *	Pointer to the previous syllable in the word
nSyll	syllable_t *	Pointer to the next syllable in the word

8.3.5 note_t structure

This is the structure managing the notes.

Member name	Type	Description
frequency	int	Note frequency (in Hz)
logFreq	double	Note frequency in log2 scale
rank	int	The note position in the scale (12 positions per scale)

8.3.6 F0 class

Constructors	Type	Description
F0(properties_t* properties)	n/a	The default constructor. Parameters: ***properties*** – pointer to application internal properties structure
F0(F0 &orig)	n/a	Copy constructor
Public virtual functions	**Type**	**Description**
buildF0(phrase_t *phrase, Phoneme **phones, int nPhones)	int	Parses the musical and phonetic data to build the pitch contour. Optionally, it can change the duration of the phonemes. Parameters: ***phrase*** – pointer to a *phrase_t* structure, containing the orthographic and musical data ***phones*** – a list of pointers to *Phoneme* objects, representing the phonetic data. (see **Error! Reference source not found.**) ***nPhones*** – the number of items in ***phones***. Return value:

		0 – if no error occurred ***Non-zero*** – for any error
getF0()	double *	Returns a copy of the internal representation of the pitch contour.
Public functions	Type	Description

Table 8-1. Structure of the *F0* base class.

8.3.7 Phoneme class

Used to store and manipulate the phonetic data. It is used by NLProcessor to return the phonetic content of a phrase.

Constructors	Type	Description
Phoneme(char *name)	n/a	The default constructor. Parameters: ***name*** – the name of the phoneme
Phoneme(Phoneme &orig)	n/a	Copy constructor
Public virtual functions	Type	Description
distance(Phoneme *other)	double	Computes a distance to another phoneme. The distance should be minimum (preferably 0) if the phonemes are identical. Parameters: ***other*** – pointer o Phoneme object containing the other phoneme's data Return value: ***0*** – if the phonemes are identical ***> 0*** – otherwise
isVowel()	int	Function specifying if the object is a vowel
isSilence()	int	Function specifying if the object is a silence
Public functions	Type	Description

8.3.8 NLProcessor class

The NLProcessor class parses the lyrics and produces the relevant phonetic transcription.

Constructors	Type	Description
NLProcessor(config_t *conf)	n/a	The default constructor. Parameters:

		conf – pointer to config_t structure, containing the application parameters
Public virtual functions	Type	Description
parsePhrase(phrase_t *phrase)	void	Parses a phrase and extracts its phonetic content, which is stored internally. Parameters: ***other*** – pointer o Phoneme object containing the other phoneme's data Return value: ***0*** – if the phonemes are identical ***> 0*** – otherwise
reset()	void	Resets the internal structures of the processor.
getPhonemes(int *nPhones)	Phoneme *	Returns a copy of the internal list of phonemes created by parsing a phrase Parameters:
getNPhonemes()	int	Returns the number of phonemes resulting after parsing the last phrase.
getPhoneme(int idx)	Phoneme *	Returns a pointer to a copy of the *idx*-th Phoneme of the phrase
Public functions	Type	Description

8.3.9 getCustomF0

This is the interface between the custom F0 generation model and the application. It is called by the application to obtain a pointer to an instance of class F0. The users will need to define it according to the following prototype.

Function Prototype		
F0 * getCustomF0(phrase_t *phrase, config_t *conf)		
Parameter	Type	Description
phrase	phrase_t*	pointer to a *phrase_t* structure, containing the orthographic and musical data
conf	config_t *	Pointer to a *config_t* structure containing the application configuration.
Return value	Type	Description
	F0 *	Pointer to an instance of F0 base class.

8.3.10 getCustomNLP

This is the interface between the custom NLP module and the application. It is called by the application to obtain a pointer to an instance of class NLProcessor. The users will need to define it according to the following prototype.

Function Prototype		
F0 * getCustomNLP(phrase_t *phrase, config_t *conf)		
Parameter	Type	Description
phrase	phrase_t*	pointer to a *phrase_t* structure, containing the orthographic and musical data
conf	config_t *	Pointer to a *config_t* structure containing the application configuration.
Return value	Type	Description
	NLProcessor *	Pointer to an instance of *NLProcessor* base class.

9. References

[1] (2015, Mar.) glt.xyz. [Online]. http://glt.xyz/

[2] Yamaha. Description of Vocaloid3. [Online]. http://www.vocaloid.com/en

[3] (2016, Jan.) Orpheus music. [Online]. http://www.orpheus-music.org/v3/index.php

[4] (2016, Jan.) Symphonic Choirs. [Online]. http://www.soundsonline.com/Symphonic-Choirs

[5] H. Kawahara, et al., "Tandem-STRAIGHT: A temporally stable power spectral representation for periodic signals and applications to interference-free spectrum, F0, and aperiodicity estimation," in *Proc. of ICASSP 2008*, Las Vegas, 2008, pp. 3933-3936.

[6] T. Nose, M. Kanemoto, T. Koriyama, and T. Kobayashi, "HMM-based expressive singing voice synthesis with singing style control and robust pitch modeling," *Computer Speech & Language*, vol. 34, no. 1, pp. 308-322, Nov. 2015.

[7] L. Ardaillon, G. Degottex, and A. Roebel, "A multi-layer F0 model for singing voice synthesis using a B-spline representation with intuitive controls," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[8] S. W. Lee, M. Dong, and L. Haizhou, "A study of F0 modelling and generation with lyrics and shape characterization for singing voice synthesis," in *Chinese Spoken Language Processing (ISCSLP), 2012 8th International Symposium on.*, 2012.

[9] M. Umbert, J. Bonada, and M. Blaauw, "Generating singing voice expression contours based on unit selection," in *Proc. SMAC*, 2013.

[10] H. Fujisaki, S. Ohno, and W. Gu, "Physiological and physical mechanisms for fundamental frequency control in some tone languages and a command-response model for generation of their F0 contours," in *International Symposium on Tonal Aspects of Languages: With Emphasis on Tone Languages*.

[11] T. Drugman and T. Dutoit, "The deterministic plus stochastic model of the residual signal and its applications," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 20, no. 3, pp. 968--981, 2012.