THE UNIVERSITY OF WESTERN AUSTRALIA

FERRARI ON A STICK: A SYSTEM FOR EMULATING ENGINE SOUNDS

Third Year Project

Chris Hellsten, 10422637
School of Mechanical Engineering
University of Western Australia

Supervisor:
Thomas Braunl
School of Electrical & Computer Engineering
University of Western Australia

Semester 1, 2009

Contents

List of Figures

Executive Summary

Recent advancements in battery storage capacity have led to the electric car moving its way away from the impractical and towards the mainstream. Electrical vehicles release no emissions and they run almost silently which may lead to less noise-polluted cities in the future. This silence does, however, pose a safety risk in today's world to both pedestrians and other road users.

This project focuses on the design and implementation of a system to emulate the sound of a petrol-based motor to be used in electric vehicles. The goal of the system is to alert nearby pedestrians and drivers of the presence of the car and to add an aesthetic perk to the electric vehicle by offering a range of realistic and impressive artificial engine sounds to the car. The software for this project has been completed and a Lotus Elise, Ferrari f350 and Aston Martin DB9 have had their engines recorded and those sound schemes are running on the software.

1 Introduction

1.1 Background

1.1.1 Electric Cars

Electric cars are not a new idea. An electric powered carriage was recorded to have been invented by Robert Anderson no later than 1839 and a working three-wheeled electric vehicle was invented by Gustave Trouve in 1881 (Westbrook 2001). This was four years prior to the first gasoline powered automobile invented by Karl Benz in 1885.

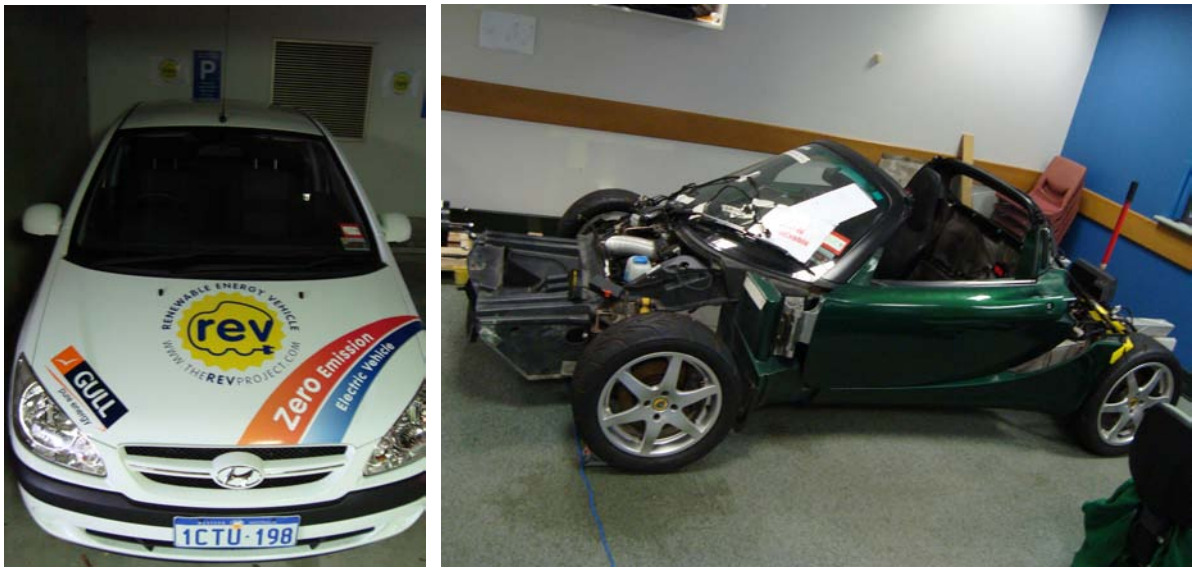
Electric cars were in full production by 1890 to 1912 and outsold gas powered cars due to their simplicity. It was not until the 1920's when the average commute distance was significantly higher and oil was becoming mass produced that gasoline powered engines began to dominate the market. Electric vehicles could not provide a sufficient range for public convenience due to batteries being unable to store enough energy. By the 1930's the electric vehicle industry had all but disappeared.

Due to recent advancements in battery storage technology and the public push towards sustainable clean development, zero-emission electric vehicles have come back into the limelight.

1.1.2 REV Team

The Renewable Energy Vehicle (REV) team at the University of Western Australia (UWA) has converted a Hyundai Getz to full-electric drive and is in the process of converting a Lotus Elise in a similar manner. The Getz was the team's first attempt to produce a renewable energy vehicle and the results were very promising. The car can be charged from any ordinary household 240V power socket. A max speed of over 100km/h can be reached and the car costs approximately 1/3 that of its petrol twin per kilometre when charged from the grid. At UWA, solar panels charge the car making it a completely zero-emission vehicle. (Figure 1a shows the electric Hyundai Getz.)

The feature of the team's work this year is the Elise, a much more ambitious project. The main goal of which is to match (or better) the performance of the petrol-driven sports car in many aspects while maintaining zero-emissions. (Figure 1b shows the Lotus Elise with combustion engine removed.)



(a) Hyundai Getz, Converted to Electric Drive

(b) Lotus Elise, Converting to Electric Drive

Figure 1: The REV projects

1.2 The Problem

The Getz runs very silently with pedestrians sometimes only a metre away and not being aware that the car is running. This poses a safety risk when driving around car parks or other populated areas such as through the university. Pedestrians use the sound of cars to make themselves aware of traffic. When a car cannot be heard pedestrians may stop paying attention to their surroundings. Additionally, if electric cars move into mainstream production they will pose a particularly high risk to blind and sight-impaired pedestrians.

The silent car is not only a problem for pedestrians however. Some test drivers of the Getz have commented that the lack of engine noise can be disconcerting. The Getz has a manual transmission and it is often useful to hear the engine to gauge when to shift gears (especially given that the tachometer in the Getz is not currently functioning.)

The problem may also affect other road users, although generally they rely less on sound for awareness given the background noise of their own car and other road vehicles.

1.3 Industry Standards

To add an aesthetic appeal to their product and to overcome the dangers of a silent car some commercial manufacturers of electric-cars have implemented or intend to implement an engine-sound emulation system.

This system alerts pedestrians that the car is running and nearby. It also assists the driver's awareness of the engine's current RPM. As an added bonus, with artificially generated engine-sounds, the driver has far more control over the sound. The volume can be adjusted and even the engine sound can be switched from a small four cylinder engine to a V8 or, in the case of the Tesla Roadster even a formula one.

1.4 The Aim

The plan for this project is to implement such a system and run it in both the Getz and the Lotus. The system software will run on the in-car Eyebot (see Section 1.5.) The audio data will be stored on a USB stick, loaded and processed through the Eyebot and amplified by speakers situated on the outer frame of the car. Figure 2 demonstrates how the system interacts with components of the car to emulate engine sounds.

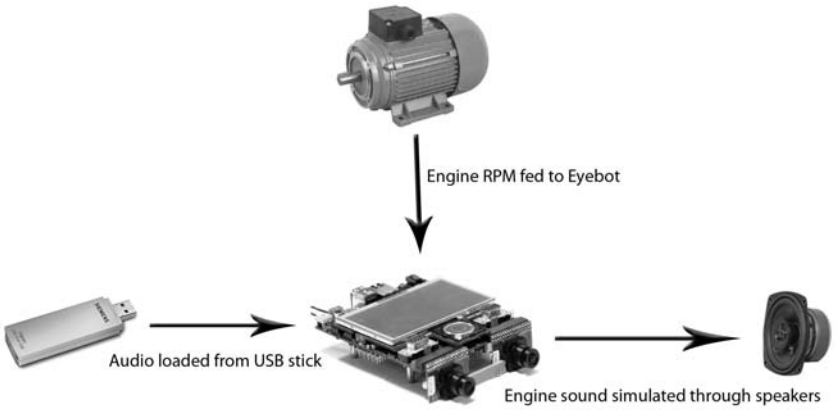


Figure 2: A simple system schematic

One of the key design goals is to have a simple user-interface making use of the LCD touch screen on the Eyebot and allowing the driver to adjust volume and sound schemes. A realistic sounding engine emulation is preferable for aesthetic appeal but any audible sound will fulfill the safety requirements.

To achieve this realistic engine sound, the audio pitch must scale with the current RPM of the electric motor so as not to confuse the driver. Achieving this smooth and realistic transition from RPM x to RPM y is the challenge of this project.

1.5 The Eyebot

The Eyebot (Figure 3) is a controller for mobile robots used throughout the Electrical and Mechatronic Engineering department at UWA. Each REV electric car will have an Eyebot mounted in the dashboard. Equipped with an LCD display and touch screen controls it serves as a powerful and multi-functional in-car computer.

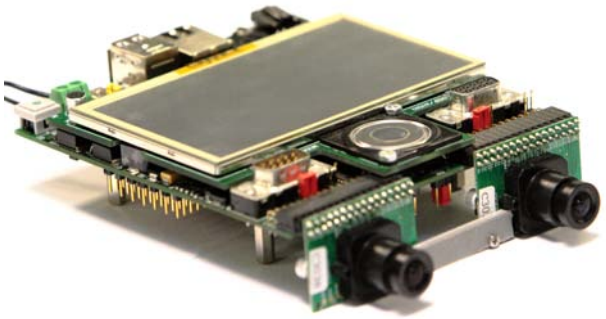


Figure 3: The Eyebot M6, installed in the Hyundai Getz

The Eyebot uses the RoBIOS operating system (O/S) which is based off the Linux O/S. The relevant features of the Eyebot include: an ARM-9 400Mhz 32-bit processor, 64 Mb ram, digital audio output and USB support (Braunl 2009).

2 General Design

2.1 Overview

The project can be effectively broken down into four parts; recording the audio, processing the audio, writing the software and the selecting, wiring and fixing of the hardware. Each of these will be discussed individually.

2.1.1 Audio Recording

The recording of the audio is the simplest process. The engine of the subject car is recorded whilst running at various RPMs so that the sound can be reproduced by the electric vehicle. Background noise should be kept minimal and the recording should be conducted outside to minimise reverberation.

A high-quality directional condenser microphone was set up approximately 50cm behind the exhaust of the car and the audio was recorded with an audio software suite running on a laptop. The driver was told to hold the car's engine at constant RPM for at least 3 seconds. This was done at various RPM intervals from 1500 RPM up to (in the case of the Ferrari) over 9000 RPM.

In some high performance cars the gas pedal may be very sensitive and difficult to keep at a constant RPM. If this is the case then audio processing becomes significantly more difficult. After these recordings had been taken the driver was asked to slowly rev from idle to red-line. These recordings are not very useful in emulating the engine but they serve as a good benchmark with which to compare the smooth transition of the emulated audio.

(a) Ferrari f350

(b) Aston Martin DB9

Figure 4: Cars used to record additional sound-schemes

The Lotus Elise was recorded before the petrol motor was pulled out by the REV team. This will serve as the default sound-scheme for the car. Figure 4 shows a Ferrari f350 and Aston Martin DB9 which were both later recorded. These sound-schemes will hopefully provide extra entertainment for the driver.

2.1.2 Audio Processing

The audio processing is a major part of this project and will be discussed in detail in its own section. Audio processing is required because it is not plausible to record the engine at every RPM or at every rate of change of RPM and yet in emulation we must produce a sound that exactly matches the state of the electric engine. To achieve this audio at every possible RPM must be produced from a set of recorded samples at every 500 RPM increments. This interpolation between the recorded samples is the main task in audio processing but conversion to Eyebot-compatible audio files is also required. Various methods to perform this interpolation are discussed later in Section 3.

2.1.3 Software Design

The task for the software is to load the processed audio files from USB stick and play them as required. To do this, the software will receive input from the electric-motor controller giving information about the current RPM.

The software should run semi-transparently on the Eyebot allowing the user to control other applications while the sounds are running but allow simple adjustments to sound scheme and / or volume via LCD GUI when prompted. Some of the features of the software include:

- Prediction based pre-emptive loading and discarding of relevant samples.
- Software controlled audio mixing and fading allowing for theoretically unlimited multi-channel audio.
- A wav parsing module that allows on-the-fly format conversion and forward compatibility for 16-bit audio.
- Scalable memory and CPU usage to balance performance with system resource usage.
- Low level double-buffered audio with variable buffer-size

More detailed information on software design will be discussed in Section 4.

2.1.4 Hardware Selection and Design

This task involves selecting speakers, amplifier and sub-woofer for each car, fixing them to the car and wiring them. This task will roll over into next semester when the Lotus is closer to completion. Some key aspects in selection of hardware involve:

- Weather proof speakers, as they will be fixed outside the car
- Should run on the 12V car battery
- Maximum volume should be at least 90dB (at 1 metre from the speaker) to compete with real car audio levels
- Minimising drag and weight so as to not throw off axle alignment or detrimentally affect on-road performance.

This will be discussed further in the future work Section 5.3.

2.2 Design Constraints

The system is to run on the Eyebot and must share resources with other programs such as GPS navigation, data-logging and user MP3 audio. For this reason there are constraints on how much resources the software can use so that the program does not detrimentally affect the performance of other parallel running applications.

2.2.1 Software Constraints

The system needs to have scalable memory usage so that it can run without impairing other programs whilst maintaining the flexibility to improve performance for later versions of the Eyebot. Likewise, the processing should be scalable. This would be in the form of updates per second so that response time and smooth transitions can be balanced against processor usage for different models.

2.2.2 Audio Quality Constraints

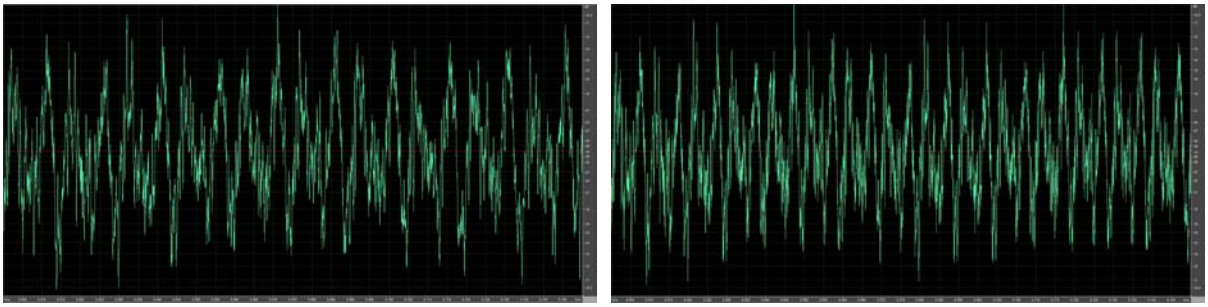
The eyebot documentation specifies that the maximum audio quality supported by the RoBIOS O/S is 8-bit sound files at 32.8kHz sampling frequency. To put this in perspective, CD quality is 16-bit at 44.1kHz. 16-bit audio support was added to the project for use with future Eyebot upgrades as this results in a vast improvement in sound quality. Adding forward compatibility for higher sampling frequencies is a simple and beneficial task.

3 Audio Processing

The purpose of audio processing is to increase the resolution between recorded samples. That is, we decrease the RPM increment between samples by producing interpolated samples to fill the gaps. Producing samples with 50 RPM increments is sufficient to produce a smooth sounding result. Several methods for this interpolation were investigated. This report covers three of those methods; frequency shifting, Fourier interpolation and wave synthesis.

3.1 Frequency Shifting

Frequency shifting is a simple method for interpolating the audio files. Adobe Audition, a sophisticated audio editor, was used to shift the frequency of one or more of the original samples by varying factors to produce a smooth spectrum of samples. Figure 5 shows the transformation of a waveform with the frequency shift effect.



(a) Original Lotus sound wave (b) Lotus sound wave after shifting

Figure 5: The effect of frequency shifting on a Lotus engine sound wave

There are two different ways that this can be done:

1. A suitable sample from the middle of the RPM range can be chosen as the sole source sample, and many frequency shifts applied to it to produce the full range of audio samples.
2. Each source sample can be used to interpolate up to the next sample.

Both methods benefit from the fact that they are very easy albeit a bit time consuming to accomplish. Figure 6 shows the difference between the two methods.

The driving rationale for the first method is that each sample blends into the next sample smoothly since they are all based off the same original source sample. In contrast, the second

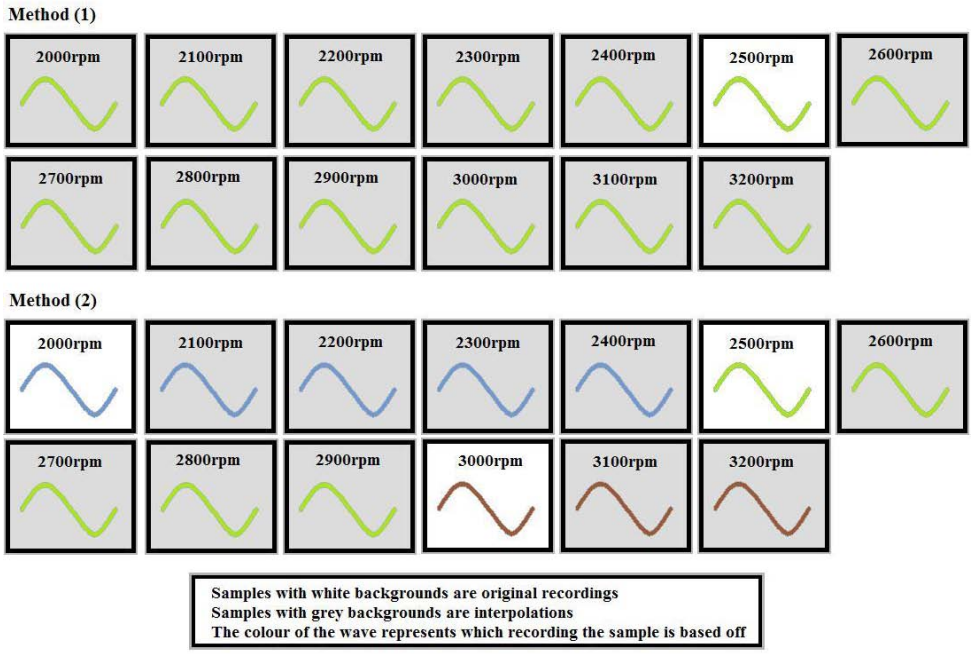


Figure 6: Methods for generating intermediary RPM sounds using frequency shifts

method causes a noticeable jump when the interpolated sample of X suddenly changes to the original recording of Y (see Figure ref, a noticeable jump is at the 2400rpm interpolated sample moving to the 2500rpm original recording in method (2).)

This occurs because the frequency of the recorded samples does not increase perfectly linearly with the RPM of the engine. In fact, the engine sound changes dramatically from low RPM to high RPM beyond just a simple frequency shift. This fact causes problems for both methods. For method (2), as mentioned above, this causes a non-smooth transition through the RPM ranges and for method (1) this causes the interpolated samples to sound very different to the actual engine at the extreme boundaries of interpolation. The end result for method (1) is a very one-dimensional sounding car.

Frequency shifting was eventually decided upon as the best method and a compromise between methods (1) and (2) was used. This is discussed in more detail in Section 3.4.

3.2 Fourier Interpolation

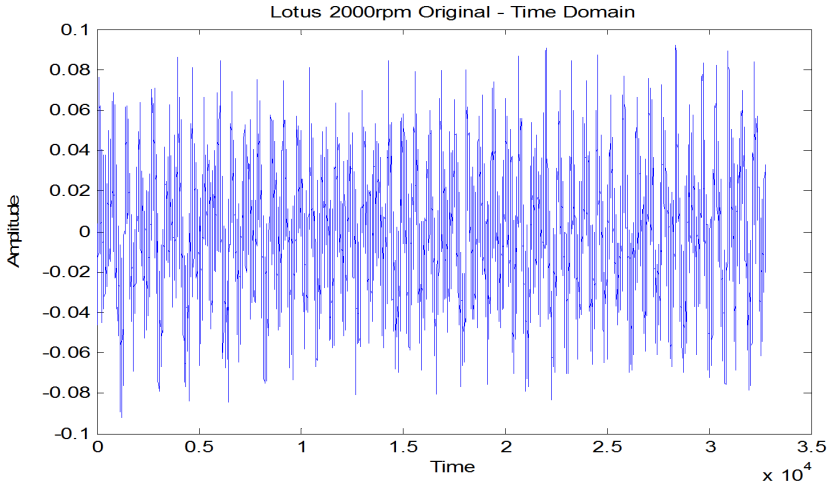
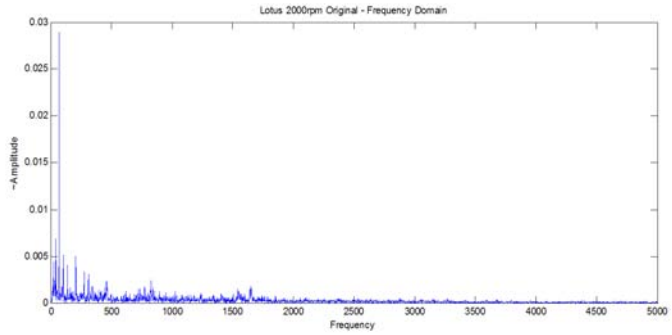


Figure 7: 2000 RPM sound wave in the time domain

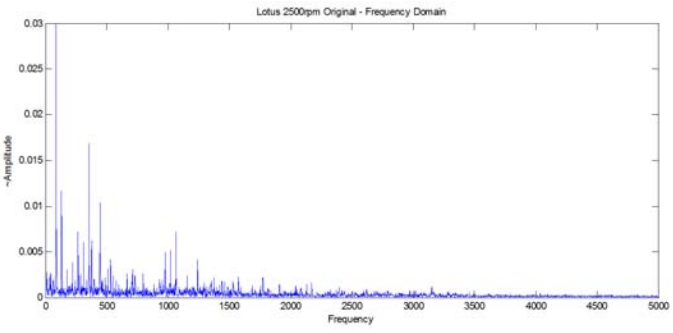
The Fourier interpolation method takes advantage of the full range of recorded samples whilst maintaining a smooth transition between interpolated samples. To perform the wave analysis MATLAB was used. Each recorded sample is imported in to MATLAB (Figure 7 shows an exmaple wave) as a waveform with respect to time and a Fourier transform is applied to switch to the frequency domain. Once in the frequency domain, data sets are created as linear interpolations between two recorded samples, producing the Fourier transform of a wave somewhere between the two recorded samples. Figure 8 shows two Fourier transforms of recorded samples; the third (2300rpm) is interpolated between these.

The next step is to apply an inverse Fourier transform to the newly created datasets, converting them back to the time domain (a standard wave-form) and this wave can be written to disk in wav format. This process was entirely automated with the use of MATLAB M-files which allows for rapid audio-processing.

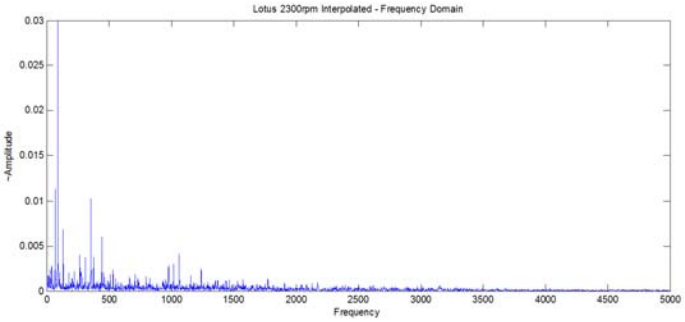
Unfortunately the end result is not realistic. The act of linearly interpolating between the Fourier of two samples simply has the same effect as playing each sample at fractional volume at the same time. Instead of increasing the dominant frequency slowly, the dominant frequency of the sample X fades away and the dominant frequency of sample Y strengthens as we interpolate towards Y and it sounds like there are two very obvious discrete signals playing. A pulsing can also be heard as the two similar dominant frequencies produce a beat effect. This method was



(a) 2000 RPM Original Sample



(b) 2500 RPM Original Sample



(c) 2300 RPM Interpolated Sample

Figure 8: Interpolating between two Fourier transforms

the first to be investigated and was quickly discarded.

With both methods described so far there is a common problem of discontinuous waveforms. That is, if the RPM of the engine changes from one instant to the next, the software will turn off the currently playing sample, and play the next sample corresponding to the new RPM. Since there is no way to know the actual amplitude of the waveform of a real sound at

any given moment, when switching from one sample to the next it is impossible to ensure the resulting wave transition is continuous. Speakers cannot play a discontinuous waveform, and the result is a popping or clicking sound similar to the clicking of an old vinyl record being played but occurring many more times per second. Figure 9 demonstrates a possible discontinuous transition between two waveforms. The wave synthesis method is designed to remove these discontinuities.

Figure 9: Example of discontinuity in wave transition

3.3 Wave Synthesis

This method involves extracting key frequency and phase information from recorded samples and using that information to completely synthesize the sound of the engine. While this sounds at first like a fruitless endeavour, it solves the discontinuity problem that arises with other methods and can produce surprisingly realistic audio.

As with Fourier interpolation MATLAB is used to analyse the waveforms and the entire process can be automated with the M-files that were created for this project.

For this method, only one sample is necessary for analysis and the first step is the Fourier transform. The dominant frequency is determined (as the highest peak in the Fourier transform) and its amplitude and phase angle are noted (this is all done automatically by the M-file.) **Note:** the Fourier transform produces a complex number plotted against a frequency. The magnitude of this complex number is the strength of the corresponding frequency in the original signal (amplitude) and the argument $\arctan(imaginary/real)$ represents the phase angle of the wave.

A user-specified number of points will then be read from the graph, below and above the dominant frequency and these points are saved as the tuple [frequency, amplitude, phase angle.] The frequency must be converted to angular frequency. The waveform can then be approximated from this list of tuples using the formula of wave superposition:

$$y = \sum_{t=0}^{T} \sum_{i=0}^{n} A_i \times \cos(F_i \times t + \pi)$$

*Where **T** is the number of samples in the signal and **n** is the number of tuples used in synthesis*

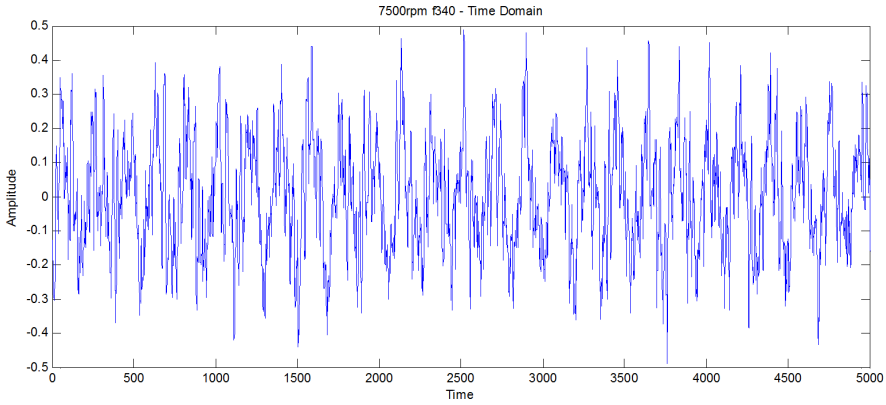


Figure 10: The original waveform for Ferrari at 7500rpm

The more points that are read (n), the more accurate the re-creation of the sample but the higher the computation time. See Figures 11 and 12 below which demonstrates the approximations for different values of n as generated from the original waveform in Figure 10.

Any other RPM sample can be approximated with this method by scaling the frequency value in each tuple. If we had just generated n tuples to describe the waveform of an RPM x and we needed to generate the waveform for an RPM y then each tuple would become:

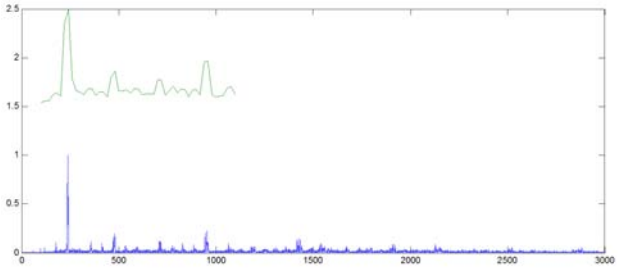
$$\text{tuple}_\text{i} = [\omega_i \times \frac{y}{x}, amplitude_i, \phi_i]$$

There are two ways to continue from here:

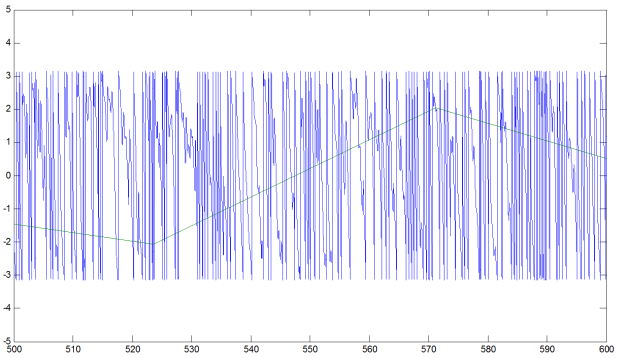
1. Produce the full range of samples now and export them as wav files to be used by the Eyebot. Ending with a similar set of wav files to the frequency shift method.

2. Simply feed the tuples to the eyebot to reproduce the samples on-the-fly.

The second method benefits from almost zero relative memory usage, since no sound files need be loaded. On the other hand the processor load increases dramatically as each sample must be calculated on the fly. The higher the quality (number of tuples) the more CPU time that will be consumed. To produce a very good quality sample, approximately 500 tuples were used and this took a 2Ghz CPU 60 seconds to compute a 2 second sample which is clearly 30x too slow to be performed on the fly. The expense here stems from the many cosine computations being calculated each loop. Using a pre-computed cosine lookup table the speed can be

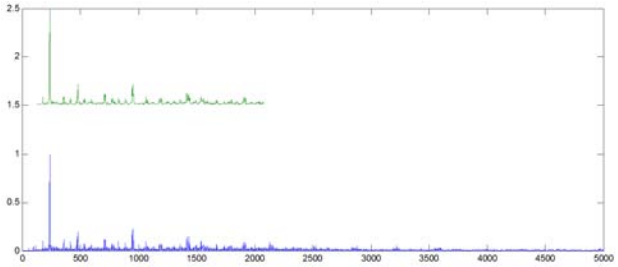


(a) Amplitude of the frequencies

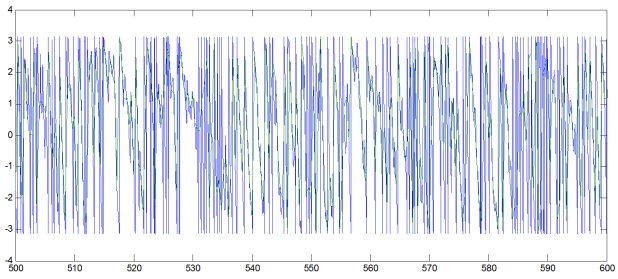


(b) Phase angel of the frequencies

Figure 11: Wave synthesis using a low number of sample points



(a) Amplitude of the frequencies



(b) Phase angel of the frequencies

Figure 12: Wave synthesis using a high number of sample points

increased by over 100x and 500 tuples may be overkill but it is still a high CPU load for the Eyebot and this would surely cause other programs to perform less effectively.

The benefit that both of these methods have over other methods is that the final wave-form is perfect. The exact amplitude can be calculated at any given time and hence the transition between any two samples can be seamless by matching phase angles from one frequency to the next. Additionally, a non-constant RPM recording can be used in analysis while still producing a constant waveform output. This method was discarded because the CPU load may well be too high for this version of the Eyebot. In the future it will be a path worth investigating deeper.

3.4 Technique Used

The final technique used was the frequency shift with two modifications to overcome the discontinuity and the one-dimensional sound problems.

Every sample was edited to fade in to maximum volume from zero volume over span of approximately 0.05 seconds. This effect forces the amplitude of every sample to start at zero and we need each sample to end on a zero amplitude as well to avoid the popping sound. It is impossible to know when a sound will be cut off by the software so editing the samples and forcing a fadeout is not useful. The fadeout effect has to be handled programmatically by the software. The end result is quite pleasing to the ear. The fade-in and fade-out is inaudible and the popping sounds disappear entirely.

To add more depth to the sounds, 3 samples are used instead of one. The first sample is at idle and it represents the low rumbling of the car. The second sample is at approximately 3000 RPM and it represents the grunt of the car. The final sample is at very high RPM (approximately 7000) and it represents the high pitch twang of the car. Each sample is used individually to produce a full spectrum of RPM using the frequency shift method from (1500 RPM to 9000 RPM.)

Each RPM now has 3 samples (rumble, grunt and twang) and all of these samples are mixed together to produce the final sample for that RPM using MATLAB. Each sample is mixed with a different scaling volume depending on the dominance of the original RPM throughout the interpolated sample. At 1500 RPM the rumble sample is mixed at 100% volume and both other samples are mixed at 0% volume (since the 1500 RPM should sound exactly as it was recorded.) At 2000 RPM the rumble would be mixed at approximately 70% volume, the grunt mixed at 35% volume and the twang remains at 0% volume until we pass the grunt RPM. The result is

a set of samples that blend smoothly into one another and increase correctly in pitch while still honouring the original car's audio profile at very low and very high RPM.

4 Software Design

4.1 Overview

The software for this project was written in C for the RoBIOS O/S. The task of the software is to load samples from USB stick and play them via the Eyebot's digital audio output. To achieve the desired multi-channelling, fade effects and seamless sample loading the audio needed to be controlled at a very low level.

4.2 Double Buffered Audio

At the base level the audio is controlled with a double buffer. A buffer is simply a contiguous block of memory that stores raw audio data. One buffer is locked and currently being played by the Eyebot while the other buffer is being loaded with audio data by the software. Once the Eyebot has finished playing the first buffer the second buffer becomes locked and the Eyebot begins playing from the second buffer. The first buffer becomes unlocked and the process repeats. The process of switching from one buffer to the next is called a buffer flip. If a large buffer size is used, the CPU load decreases and the chance of skipping decreases also. On the other hand, a large buffer adds a longer latency to the sound. That is, new samples will take longer to play and the effect is that the engine sounds are slightly delayed behind the actual RPM of the engine.

4.3 Project Breakdown

The project consists of six source and six header files:

es_engine_sounds.c / .h

This source file contains the program entry point, the main operational loop and a handful of utility functions used throughout the program. The corresponding header contains all useful definitions that a programmer might like to change to tweak the performance such as audio buffer size, number of parallel audio channels, update rate, cache size etc.

es_fileio.c / .h

The file I/O source provides a set of simplified functions for accessing files in binary form. These routines are called whenever a sample needs to be loaded from disk (often many times per second) or if the user attempts to change the audio scheme.

es_gearbox.c / .h

The gearbox source interfaces with the car's controllers and updates current RPM. It is also designed to retrieve other information about the car such as speed and throttle position though it is unclear if this data will be available to the eyebot at this stage. One of the gearbox's main tasks is to handle the simulated gears functionality, see Section 4.5.

es_soundtable.c / .h

The sound table is a data structure that stores an array of audio samples in memory. It interfaces with the gearbox code to determine which samples to load and which to discard. The main operational loop will instruct the sound table to play a new sample many times per second and it must choose the most appropriate sample that is loaded to play.

es_audio.c / .h

The audio module handles the parsing of wav files. It retrieves data from the File I/O and converts the binary data into a neat sample structure which is passed to the sound table for storage. The module is also responsible for handling multi-channels, fade effects, the mixing of these channels and the handling and flipping of the audio buffers.

es_gui.c / .h

This source file handles the GUI and user input that is displayed while the program is running.

4.4 Sample Path

The samples move through a series of phases from USB stick to audio output. Initially the audio files exist only on disk. If the sound table requests that an audio file be loaded this is handled by the File I/O and audio-wav-parsing modules and the sample is now stored in a cache within the sound table.

If the sound table chooses to play that sample it is copied to the audio module where it is loaded into the best available channel. A channel is simply an active sound. The more channels

there are the more flexible the program is but the slower the mixing phase. Once the sound is in a channel it will be loaded into a buffer after the next *buffer flip* occurs. Figure 13 shows the path of a sample through the system.

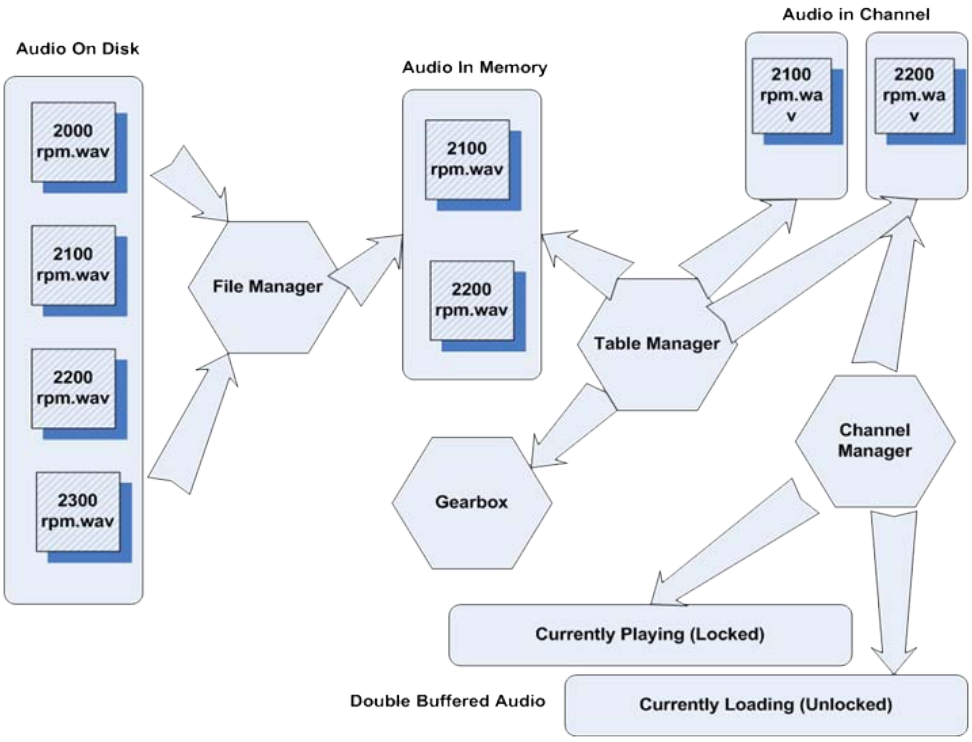


Figure 13: Sample state diagram

After a *buffer flip* occurs, the next stream of audio data is loaded into the unlocked buffer. This data is the result of wave superposition of all active channels. The volume and fade effects are handled at this mixing level. Once in the buffer, the audio will be played directly when the next *buffer flip* occurs. In most cases the buffer length is many times shorter than the length of the sample to be played, so the channel remembers how much of the audio has been played so far and updates the position in the sample. The rest of the sample will be loaded into future buffers unless the channel is overridden with a new sample.

4.5 Simulated Gears

The electric motor used in the Lotus Elise has constant torque from 0 to 10,000 RPM and redlines at 14,000 RPM. This complicates the mimicking of an internal combustion engine. To

allow the electric engine to play audio at 14,000 RPM, the RPM would need to be scaled down to the RPM range of an internal combustion engine. This would then have the side-effect of the engine sounding like it is accelerating slower than it is.

As an alternative, the software has functionality to simulate gears entirely. If simulated gears are being used, then the audio being played is no longer related to the RPM of the engine, but to the speed of the car and the gear ratio of the simulated gear. It should be noted that this simulated gear is not related to the actual gear of the car, it is simply a software state.

The software automatically shifts up and down simulated gears based on a similar algorithm used in automatic transmission vehicles. The changing of gears is a function of simulated RPM and throttle position. This may be misleading to the driver who can hear the engine pseudo-changing gears without any action on their behalf (nor any physical change in torque, RPM or engine disengagement.) It does, however, provide for a more interesting and suitable sound profile especially in populated areas where the car may never need to change gears.

Without simulated gears, if the driver is cruising at 9000 RPM, the engine sounds would be playing at max volume for the duration of the cruise which is unnecessary and annoying for everyone involved.

4.6 Wav File Format

A module for parsing WAV data in binary form has been developed for the project. The software is only compatible with the WAV file format using Microsoft's RIFF specification. RIFF documents exist as a series of descriptor chunks (Microsoft/IBM 1992). The only chunks handled in this project are the header and data chunks which are the only two chunks used in most WAV files. WAV files to be used with the project must be 8 or 16-bit depth and any sampling frequency.

5 Future Work

5.1 Other Possibilities

There are a couple of other methodologies that were investigated in less detail that could produce better results. These methods involve using the recordings of the RPM sweeps rather than the constant RPM samples. For this report the following two methods are referred to as sweep cutting and sweep strolling.

5.1.1 Sweep Cutting

If the recorded sweeps were slow enough, then they could be split into separate constant RPM files directly at a high enough resolution to avoid any audio interpolation at all. Recording these very slow sweeps can be troublesome though given that the car is not moving so the airflow through the radiator drops off significantly and the engine can overheat. For this reason it was not possible for this project to find someone with a suitable sports car that would allow this sort of treatment.

This would however provide a perfect sound profile for any given car and this method is often used for generating the audio in high budget car-racing video games.

5.1.2 Sweep Strolling

The idea behind sweep strolling is that the software uses the sweep samples directly rather than a different sample for each RPM. This technique was not investigated deeply as it would require an entirely different software architecture to perform correctly.

The recordings must be a perfect linearly increasing RPM sweep from idle to the maximum RPM and similarly a second recording in reverse from maximum RPM back to idle. Both recordings should be the same length (they increase and decrease at the same rate.)

The software could then move to the correct position in either of the audio files using a simple linear interpolation between minimum and maximum RPMs to match any RPM that the engine is running at. The strategy is then to play from the correct position in the upwards moving audio. If the RPM of the audio exceeds that of the engine by a certain threshold after a given period of time, the software switches to the decreasing sample and the audio RPM tends back towards the engine RPM.

Conversely if the engine RPM is increasing faster than the rate that the audio RPM was recorded, the position can be skipped forward to restore alignment. If the driver is holding a steady RPM the audio will be strolling up and down over that RPM by a small user-definable window. If the driver is accelerating or decelerating then the software balances direction switches with audio skipping to keep the audio RPM in-line with the engine RPM.

Very little audio processing would be required for this method and only two samples would need to be loaded into memory. This would bypass the need for a sound table entirely as all sounds could be loaded at the same time.

5.2 Improvements to Recording Process

As discussed earlier, the samples for this project were recorded in mono-channel (one microphone behind the exhaust) due to limited resources. In commercial projects it is common to record the engine sounds from multiple positions to create at least a dual-channel (stereo) sample (Hill 2002). This second microphone is often placed next to the engine or inside the car and provides a richer sounding result.

Samples should also, ideally, be recorded while the car is both loaded and unloaded as the sound in each case is very different. Loading the cars onto a dynamometer for the recording session was not possible for this project. All samples were recorded with the car in neutral. The thermal stress on the motor is further exacerbated when running recording sessions on the dynamometer for long periods of time.

5.3 Hardware

For the system to function correctly, the final stage is the selection and setup of the audio-playing hardware; including speakers and amplifier. At this stage it is unclear what the limitations for this hardware are in terms of power and weight. Another important consideration will be where to attach the speakers. If attached simply to the front of the car, it may be possible to maintain some of the advantages of being silent when not driving towards you and only warn those that the car is moving towards.

6 Conclusion

The final result is an artificial sound that is unmistakably an internal combustion engine. The crudities of the recording and processing phases do, however, drop the impressiveness of the original cars. Mostly this is due to the volume of the playback. Reproducing audio at the volume that a Ferrari produces it at is simply not viable and the driving range of the car would be affected by this wasteful use of energy.

Future work on the audio processing may produce far more stunning results. As long as constant RPM audio files are continued to be used as the input, the scalable and robust software will continue to sound better and better and should be useful for years to come.

References

Bar, M (2001), *Linux File Systems*, McGraw-Hill.

Braunl, T (2009), 'Eyebot m6 documentation', Available from: `http://robotics.ee.uwa.edu.au/eyeM6/doc/` [21 May 2009].

Hill, G (2002), 'Capturing engine sounds for games', Available from: `http://www.gamasutra.com/features/20021030/hill_02.htm` [31 March 2009].

Microsoft/IBM (1992), *Microsoft Multimedia Standards Update 1.0.97*, Microsoft.

University of Michigan (2005), *MATLAB: The Language of Technical Computing*, Mathworks Inc.

Westbrook, M (2001), *The Electric Car: Development and Future of Battery, Hybrid and Fuel-Cell Cars*, The Institution of Engineering and Technology.

Zolzer, U (1997), *Digital Audio Signal Processing*, John Wiley and Sons.