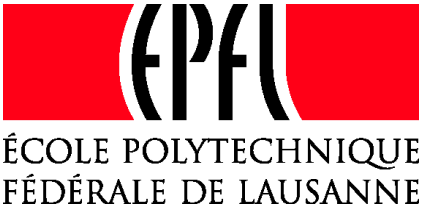


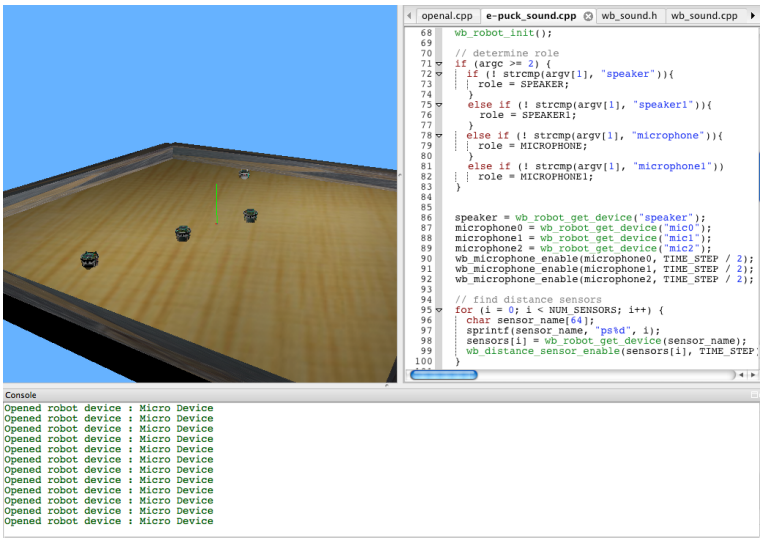
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

SS 2012, IC-IN, No. CYBERBOTICS-DISAL-MP17

Start: 20.02.2012
Finish: 17.08.2012

Development of Microphone and Speaker models in Webots using OpenAL

Christophe Schild



Professor:	Alcherio Martinoli
Assistants:	Olivier Michel, Amanda Prorok

This page is intentionally left blank.

Contents

Abstract

This report is about presenting the work I did to improve Webots with the possibility to run realistic sound simulations. It was done in the scope of my Master Thesis as an internship within Cyberbotics. The idea was to use the OpenAL library to model microphones and speakers in Webots, following several specific prerequisites such as permitting multi-robots simulations and having sound playback. My work is not included into the core of Webots but as a sound plugin instead, which is, in its current state, capable of fulfilling most of these requirements with a few exceptions.

Chapter 1 Introduction

In this chapter, I will first introduce the context surrounding my project and its goals. I will then describe the tools I had to work with and detail what I had to implement in order to reach these goals. There will be some feedback on the results I obtained followed by the possibilities that this project creates for further development.

1.1 Webots

Webots is a mobile robotic simulation software created by Cyberbotics and is vastly used in the academic domain. It allows one to create simulations mimicking the behavior of various robots. It includes support for many devices such as cameras, motors and numerous additional sensors. The whole point of this program is to have hyper realistic simulations so we can be certain enough that the robots would act the same way in real life as it would in Webots. It means that simulation control must answer to some very strict criteria that I will detail more later **(See Section 1.3)**. Some of the fields of application Webots is used for include swarm intelligence, artificial life and evolutionary robotics, self-reconfiguring modular robotics and experimental environment for computer vision.

1.2 Microphone, speakers and existing plugin

Many real robots are equipped with microphones, speakers or both and are able to react to sound stimuli to, for instance, interpret vocal commands and act accordingly or locate a sound source and look at the speaker when spoken to. These are only examples but the possibilities are vast and it would be useful to be able to test these features. Unfortunately, Webots is currently lacking the potential to simulate sound systems. It is then needed to create models for microphones and speakers in Webots that would be compliant with physical properties of sound. As a matter of fact, Webots is not entirely

deprived of such capabilities, as there is an existing plugin called 'swis2d' that allows some sound related operations that I will detail later. This plugin does not handle real sound but only buffers of data. It is capable of transmitting these data from a speaker-equipped robot to a microphone-equipped robot if there exists a path between those two entities. It also partially handles occlusion and reflection and factors aspects such as distance related delays and attenuation of sound. The plugin interacts with Webots via two functions :

```
wb_speaker_emit_sample(speaker, SAMPLE, sizeof(SAMPLE));
```

and

```
wb_microphone_get_sample_data(microphone);
```

These functions can be called by the user to send data from the speaker and recuperate them via the microphones. See **Figure 3.1** on the next page to have an example of this plugin in fonction which gives an insight into what it looks like. The white line represents the path used by the sound to travel from the speaker to the microphone and the yellow lines represent the bounding boxes of the objects of the virtual world.

As for the drawbacks, this plugin does not handle real sound (only transfer of data) meaning that it would be impossible to have sound playback. Yet, this function represents a really crucial aspect as it allows to get instant feedback on what is happening in the simulation. It also only works in two dimensions which prevents a lot of configurations to be tested such as having robots on different horizontal planes. These issues need to be addressed and will be discussed in the next section.

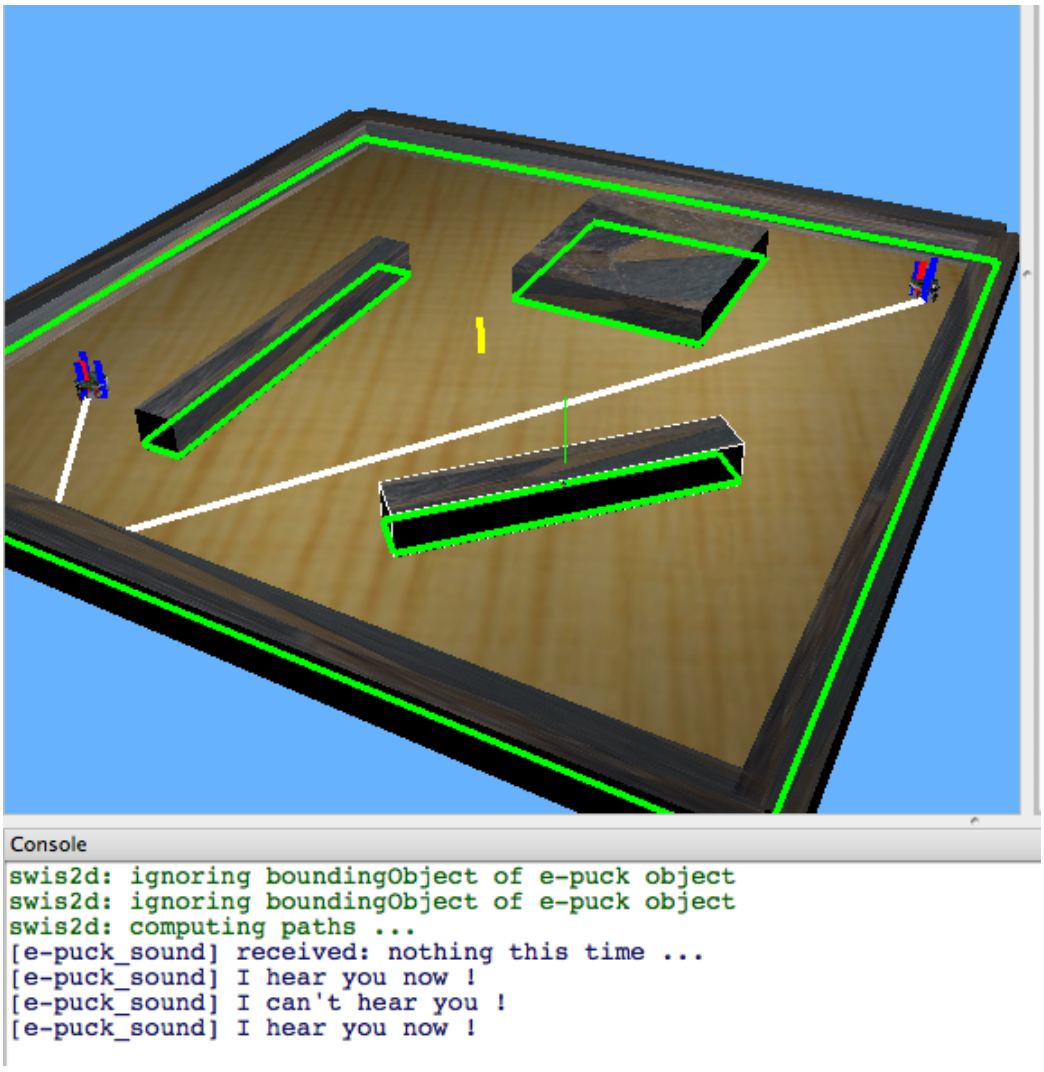


Figure 1.1: The swis2d plugin during a simulation

1.3 Project goals

My project consists in creating a new sound plugin, based on the structure of swis2d, that would use the OpenAL library to handle sound. I will detail this later but it is important to note that using this particular library was entirely part of the assignment. Here is a list of some of the criteria that need to be met :

- Realism of sound : it means it has to take into account sound velocity, distance related attenuation, Doppler's effect and so on.
- Comprehensive for the user : the user needs to be able to create interactions between robots with simple commands.
- Sound playback : as already mentioned, the user needs feedback when he observes the simulation and this is the way to obtain it.
- Multiple robots handling : it is crucial that a simulation can run with as many microphones or robots as wished.
- Running modes : Webots being a simulation program, we should be able to run simulation in real time or virtual time to get the results from long runs in a quicker way.
- Cross-platform : Webots can run on several platforms and so should this plugin.

This list is not exhaustive but gives an idea of what are going to be the main focuses of my work.

Chapter 2 Starting point

This project was not about creating an entire new library capable of accomplishing complex tasks but about adapting an existing one to reach certain goals in Webots. In this section, I will review what I started my project with. I used the OpenAL library and the swis2d plugin of Webots as a base for my work so I am going to describe shortly how they work and how they should be used.

2.1 OpenAL

OpenAL (Open Audio Library) is a software interface to audio hardware. One can use this library to produce high-quality three-dimensional sound from different sources mixed together. The major advantages of this library reside in the fact that it is cross-platform and works natively in three dimensions which complies well with the objectives of this project, even though it was initially designed for gaming applications.

2.1.1 Objects structure

In this section, I will discuss how the OpenAL objects are structured and related to each other. These explanations will give a basic overview of the functionnalities and limitations of the library in its standard released state and give a feel of what needs to be adressed in the scope of this project.

There are three fundamental types of objects in OpenAL : listeners, sources and buffers. Listeners and sources represent the two ends of a communication system with the source emitting sound that will eventually arrive to the listener (if it's reachable). The buffers will be filled with audio data to send and then linked with some sources. In other words, the buffers contain the data that is going to be played in the simulation by the sources for the listeners.

The two other types of objects present in OpenAL are the devices and the contexts. The devices represent an hardware or a virtual device. For

instance, if we want sound playback in a simulation, we will have to create a device using a backend controlling the sound card of our machine. We could also use software backends instead that perform some useful operations as we will see later. As for contexts, they represent the virtual spaces into which we will carry out operations since every AL operation is applied on a given context, not a device.

Let us take a look at how they are arranged with the following diagram :

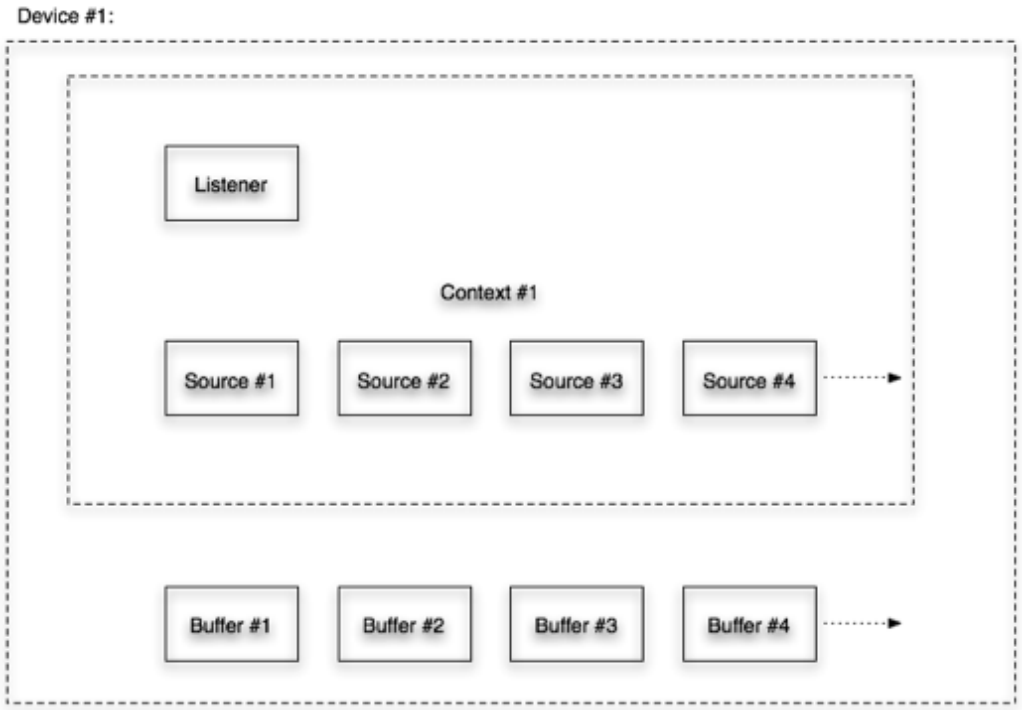


Figure 2.1: OpenAL objects and their relation to each other.[1]

This figure enables us to see the library's limitations. Webots' users should indeed be allowed to run a simulation with multiple robots at the same time. This means we need multiple listeners, each linked to a different microphone in the simulation. Yet, this diagram directly shows we will need multiple contexts in order to achieve this goal since only one listener is allowed per context. It is also apparent that we can create multiple contexts for a given device but they are not natively threaded which means that we can only execute operations on one context at any given point in time. If the context is switched while there are still operations pending in the other one, they will just be ignored because it no longer is the current context. This is

problematic since we need a perfectly parallel execution over all the robots. Nevertheless, the devices are automatically created in different threads, allowing us to work in parallel. This means that we will need multiple devices to run multiple listeners which is currently not possible in OpenAL. This incapacity to run multi-listeners applications is, moreover, OpenAL's major flaw.

The other default appearing in the diagram is the fact that sources are not shared amongst devices. This obliges us to create each and every sound source for every device, even though many of them are just several occurrences of the same sound data. This is not a problem that prevents the program from running correctly but it makes it inefficient and cannot be addressed right now since it would require to modify the whole architecture of the library based on the fact that there is one listener per context.

Let us note that buffers may be shared amongst contexts but it is pointless since we do not use multiple contexts on the same device.

Executing a basic program using OpenAL is a very trivial task and I will just describe what a basic program looks like to give one an idea. First, a device needs to be created. Then, a context is also created and linked with the device. After that, we can create sources and buffers in the scope of this context and attach one to another. The source just needs to be played and if certain parameters were correctly set (such as sources' and listeners' speeds and positions), sound should be heard in a realistic way taking into account every information we previously set.

This gives a general idea of what OpenAL is capable of and how to use it.

2.2 Swis2d plugin

As already mentioned, swis2d is the current plugin of Webots regarding sound simulation. We have already seen its features but I will now briefly discuss what I am going to use it for.

The idea was to first implement sound simulation into the existing plugin since its structure already existed and I would not need to touch Webots' source code. This would have been some sort of draft for the next version, fully included into Webots but as a matter of fact, I did not have time to achieve this and my code still lies in the plugin.

2.2.1 Plugin structure

Webots possesses an interface for sound plugins which did not give me much room for function structures or execution flow. Most functions need to be incorporated into the plugin, even if left blank. As an example, let us look at some of them :

```
void webots_sound_add_microphone(MicrophoneRef mic, double aperture,
double sensitivity);

void *webots_sound_microphone_receive_sample(MicrophoneRef mic,
double lastUpdate, int &size);

void webots_sound_step(double s);

void webots_sound_draw();
```

Some of these functions are empty (like 'draw') because I do not need them but they have to be present anyway. As it is shown, some arguments are passed to these functions but I cannot choose alternatives for them which will be troublesome for some aspects but more on that later. **(See Section 2.2.2)**

As for how the plugin works, after initializing some parameters and all the microphones and speakers of the created world, Webots will constantly call the `step` function until the end of the simulation. While it goes on, the only other functions that are going to work are `speaker_emit_sample` and `microphone_receive_sample`. These functions are called by the controller through Webots so they are controlled by the user (they also represent the only form of direction that the user has upon the simulation). It is also important to note that the latter is the only one having to return values when called by Webots. This is then through this function that data will be exchanged between these two elements.

The previous explanations should give an idea of how swis2d runs. Even tough I am basing my plugin on its body, they actually do not have much in common so most of its code is rendered useless for this project.

2.2.2 Limitations

As for the drawbacks, there are two main issues with the structure of the plugin.

As already mentioned, the first one concerns the lack of control I have

on the type of parameters passed to the plugin functions. The easiest way to exemplify this problem may be to provide a concrete example. Let us imagine that a user just wants to load a sound and play it from a robot's speakers in his simulation. He is going to send a buffer containing the sound data to Webots that will make it advance to the plugin via the following call :

```
webots_sound_speaker_emit_sample(SpeakerRef ref, const void *sample,
int rSize);
```

This is great as this function is automatically called when we need it. Yet, when we try to attach the sound data to an OpenAL buffer to be later played, we need to invoke this function:

```
alBufferData(buffer, format, data, size, frequency);
```

This will set the sound parameters of this particular sound content in OpenAL and if we take a closer look, we can notice that we have access to the `data` and `size` fields as they were passed as arguments in the function (as sample and rSize) but we have no idea about the format and frequency we are supposed to use. These parameters are contained in the headers of any .wav file and are easily retrieved but the plugin cannot get them. In this case, I sort of cheated by associating certain files with certain parameters since I only worked with three different sound files for my tests and I could then know these parameters in advance. However, this is something that would need to get fixed if it were to remain as a plugin since we want to be able to work with any .wav file. This was only one example but it can be extended to other functions as well.

The second issue is about what gets called when. We saw in the previous example that it was convenient for this case but this does not work like that all the time. I can take as an example the `cleanup` function that is called at the very beginning of the simulation. This probably worked well for the different objects that needed to be cleaned in swis2d but I have sockets in listening mode that just cannot left to be cleaned that way because it will not act the same way if the simulation is reverted with or without being run beforehand. **(See Section 3.3.6)**

2.3 Related Work

When starting with a new project, it is important to take a look at the state of the art in this particular domain to have an idea about what has already been achieved and acquire a possible basis for one's own work. After having made some research, it becomes evident that not much work has been done in this field which is not surprising given the specificity of this project.

People working on this subject tend to take one of the two following course of actions. The first option is to use OpenAL but only with a single-listener type of simulation. This can be used, for instance, to improve the realism of 3-D sound simulation as discussed by Jean-Marc Jot[2][3]. Most of these discussions are based on multi-sources applications of ultra realistic simulations with special effects and such but with only one listener. At the other end of the spectrum lies the second solution. M. Wozniewski, Z. Settel and J.R. Cooperstock[4] explore the multi-listeners problem, presenting a solution for it. The key aspect of this work is that it is mainly motivated by the fact that OpenAL does not handle this kind of situations and they try to fix it by proposing an alternative to OpenAL.

Even though there is a certain lack of documentation about this issue, I did not actually worked on a totally new field since it appeared that a certain project (called 'The Replicator'[5]) did exist and had a lot of similarities with what we intended to do. This is a European research project hosted by Almende. It is about investigating principles of adaptation and evolution for multi-robot organisms. These robots are supposed to be able to interact with the world so they developed a subproject called OpenAL-sim to extend OpenAL-soft (open source release of OpenAL) with multiple listeners and distance dependent audio delays. Those are two features that I needed in my project so taking a look at their work (their code is public) was a good start and served as a base for my project.

Chapter 3 Implementation

In this chapter, I will describe the important steps I went through during the project as well as the critical functionalities that I had to implement to reach my goals. I will start by describing how the general actors of this application act according to one another and then I will dive deeper into the details of what I had to do for each of them. There are two major actors in the scope of this project, the OpenAL library and Webots' plugin. I often had to go back working on one and then the other because the completion of some tasks sometimes induced the realization that some element needed to be modified or some features were missing. For more clarity, I did not base this chapter on the progression of the project but ordered the subsections according to the components instead. It means that some subsections will sometimes be cited but will only be explained later through the remaining sections of this chapter.

The two following sections are then about OpenAL and the swis2d plugin since those are the place where most of my coding went. The section about OpenAL will explain what features were missing and had to be added and the section about the plugin will be about the adaptation of this library into the plugin. I will also briefly describe some functions I added to help a Webot's user that would want to use these features into his simulations without having to worry too much about sound files and so on.

3.1 General Workflow

This section is about explaining the relations between each component of the execution flow **(See Figure 3.1)**. As a Webots' user, the only one you care about is the controller since this is the one you have to write yourself. As a remainder, the controller in Webots is a piece of code that will dictate the behavior of the robots in your simulation. This is crucial because we do not want to impose the user to have to handle OpenAL himself. This has to be completely invisible to the user and he should not even know what library he is using. The only tasks that you can perform from the controller is then

to interact with Webots to play some sounds from the robots if they possess speakers or get the sound data that you would hear if you were the robot in the simulation (if the robot has at least one microphone of course).

Then you have Webots itself which acts as a relay between the controller and the plugin since these two cannot interact directly with each other. It will also make all the necessary calls to initiate the plugin with the elements present in the simulation world.

The last one is the plugin that will perform much of the work. All the OpenAL commands and objects will be handled from here and it is responsible to return to Webots the correct output for each microphone when the controller decides to request it.

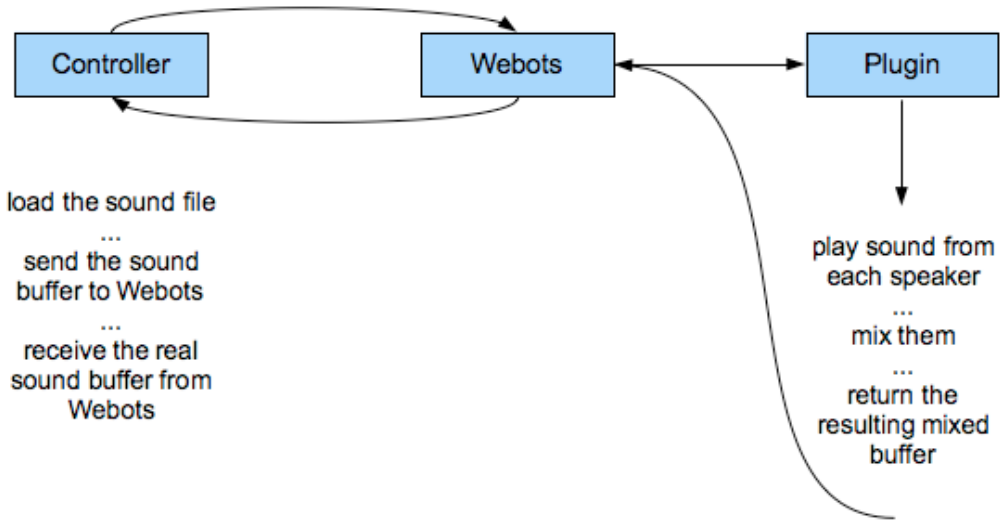


Figure 3.1: General workflow of the application

This figure is a graphical representation of what I explained beforehand. It is kept as simple as possible to give an easy-to-get idea of what is going on.

3.2 Extensions to OpenAL-soft

In this section I will describe the changes I brought to OpenAL but I will first relate why I ended up using OpenAL-soft instead of OpenAL and why it took me a long time to start making progress.

3.2.1 From OpenAL to OpenAL-soft

When I began this project, I did not know which version of OpenAL I should use so when I started working on my laptop (running on Mac OS X), I used the version shipped with the operation system like I would use OpenGL. I played around a little bit with it and realized that it was not as permissive as I thought it would be and that it would need modifications to help me reach my goals. It became obvious that I would have to replace OpenAL by its open-source counterpart, OpenAL-soft. This is also when we found out about the Almende project **(See Section 2.3)**.

As already mentioned, they had found solutions to implement several features that we would also need in OpenAL-soft. I spent a lot of time analyzing what they did and trying to repeat it but, even though they claimed to have found a cross-platform solution, I just could not get it to work entirely on Mac OS X. I had a version that I could use to create interactions between multiple robots but I could not get the sound playback which is a strong requirement of the project. The reason is that the default backend, the one that is acting as an interface between the application and the computer's hardware was not included for this platform. They made their development on Linux and could use the backend for ALSA (Advanced Linux Sound Architecture) or OSS (Open Sound System). I was then stuck between OpenAL that cannot be modified but works on Mac OS X and OpenAL-soft that does the exact opposite. The solution proposed by Almende was to use another library, PulseAudio which is supposed to be used as a sound server to make the link between OpenAL-soft and the hardware of the machine to replace the missing backend. I never managed to get it working like it should have and at this point, time was passing by and this solution did not seem very promising. I could have switched my system on Linux to use their version but this project is supposed to be cross-platform anyway so it was important to get it to work on my laptop's platform.

This is when we realized the version 1.14 of OpenAL-soft just came out (the Almende project was based on the version 1.11) bringing amongst its new features a backend for CoreAudio (the interface to manipulate sound on Mac OS X). The source code between those releases changed quite a bit,

making most of my modifications obsolete but I finally had a functioning library to work with.

3.2.2 Multiple listeners

As already discussed, one of the project's requirements is to offer the possibility to simulate multiple robots with possibly multiple microphones at the same time. OpenAL-soft does not allow to create several instances of the same type of backends for hardware backends (you cannot create multiple devices that would try to use the computer's speakers) and does not handle correctly multiple software backends. When an application is run, the configuration file `.alsoftrc` is parsed to get some parameters. This file looks like that :

```
drivers=core, wave

[wave]
file=/Users/christophe/Documents/Sound/output

[micro]
file=/Users/christophe/Documents/Sockets/mysocket
```

The first line lists the backends that will be available to the application. In this case I am restraining the choice of backends to CoreAudio and the Wave File Writer backend. The latter is used to write everything its listener perceives in a .wav file that we can play later instead of playing it live on the speakers of the computer. Then you have parameters for the wave and the micro backends. The first one is simply the file in which the sound is going to be recorded (it becomes output.wav). The second one is not important right now.**(See Section 3.2.4)**

It is not possible to create more than one device using CoreAudio but this is absolutely not problematic in our case since we do not need this (one playback device is enough). However, it is possible to create multiple backends using wave but they will not be handled correctly. They will all act as one entity and will then all produce the same result (in this case, they will all write in the file given in the configuration file). I had to tweak some details, like setting the different devices to write in separate files given their creation order in addition of removing some restrictions and it was good to go.

In the scope of this project, the types of backend needed are known so I decided to make a design choice by imposing what devices are created in

which order. In a simulation, we will always need one device for the playback sound (CoreAudio in my case) and then, as many devices as there are microphones to handle them. If you take a look back at my configuration file example, it would mean that the first device created in the simulation is always going to use CoreAudio and all the following ones are going to use the Wave File Write backend. All of them will be created using the following unique function call:

```
ALCdevice *myDevice = alcOpenDevice(NULL);
```

This would prevent people from doing mistakes when trying to create illegal devices in the plugin.

3.2.3 Multiple types of playback backends

OpenAL-soft possesses two categories of backends : playback and capture. I already explained earlier what a playback backend is used for and for the capture backend, it attempts to use a real microphone to input sound into the simulation. This last one is never going to be used into this project because we do not need this feature.

The problem is that only one type of backend per category can be created. The parser attributes the first playback backend it finds in the configuration file to the type of backend that will be created throughout the whole simulation. The same applies for the capture. Even though the Wave File Writer has the purpose of capturing sound, it acts as a playback device that just redirects resulting sound into a file instead of an hardware device like CoreAudio would do with the sound card. To circumvent this prevention (that did not exist in the older versions of OpenAL-soft), I created a third category with the same kind of proprieties as the playback backend and modified the parsing of the configuration file so this category gets a driver attributed to it. With those two modifications done, we now have a system close to what they managed to develop in the Almende project but working on Mac OS X.

3.2.4 New microphone backend

Throughout the two last subsections, I described the need to have multiple devices running on the Wave File Writer backend. As a remainder, this backend is created and then acts in a totally independent way. A new thread takes care of constantly watching the current state of the sounds being played and stores its resulting mixed perceived sound in a file. If we take a look back at **figure 3.1**, we can notice that the whole way back from the plugin to the

controller of the sound data would then be missing. We do not know what the user would like to do with the sound buffer Webots returns him. Maybe he would want to store it as a .wav file like we did until then or maybe he would just want to analyse the sound data or even make his robots follow sound sources. In any case, the sound needs to come back to the controller as raw data for the convenience of the user.

The device being created in a separate thread (which is, let us not forget, needed to insure parallelism), the data cannot just be requested from the plugin to the device. The following figure depicts the behavior of those backends :

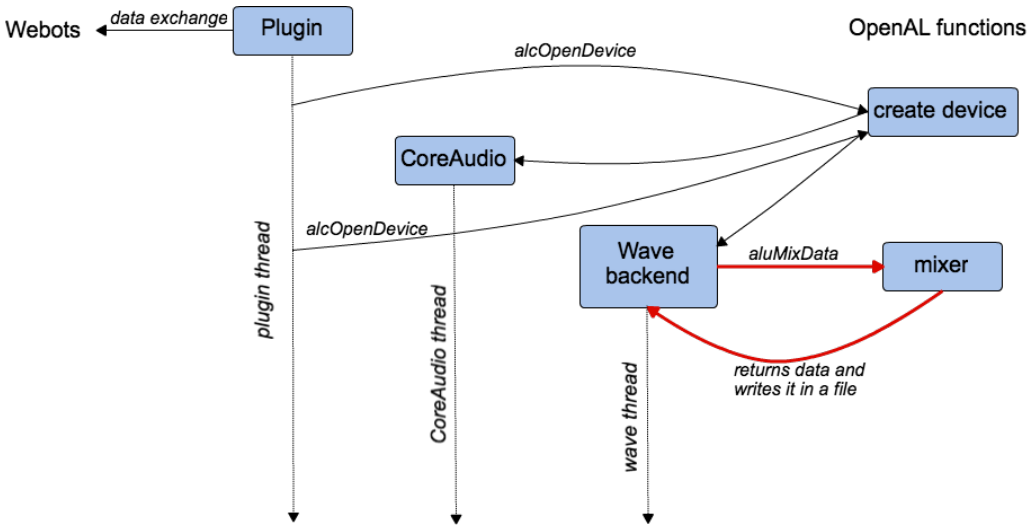


Figure 3.2: General flow of the devices with the Wave File Writer backend

The wide red arrows represent function calls that will be made in the loop that is going to run for the remainder of the execution. I did not include this loop for the CoreAudio backend because we are not really interested in its functioning since it already provides what we need at this level but the principle is the same.

In addition to the fact that our backend does not need to write in a file (it is at least not something that we want to see happening automatically), we need a backend that is able to communicate back to the plugin so it can provide the controller with data through Webots. This is what I achieved by creating a brand new backend that I just called "Microphone" since it

represents its envisaged use. This backend acts quite similarly as the Wave File Writer backend does but instead of being written in a file, the data is sent to the plugin through sockets. I will add further details about what problem arrose from this concerning parallelism and synchronization a little bit later. **(See Section 3.3.3)**

The only action that needs to be taken to use this new backend is replace the first line of the configuration file by :

```
drivers=core,micro
```

The plugin now possesses data to pass to Webots that we have to pass further to the correct controller (there is one process running a separate version of the controller for each robot present in the simulation). The plugin workflow looks like the following figure now :

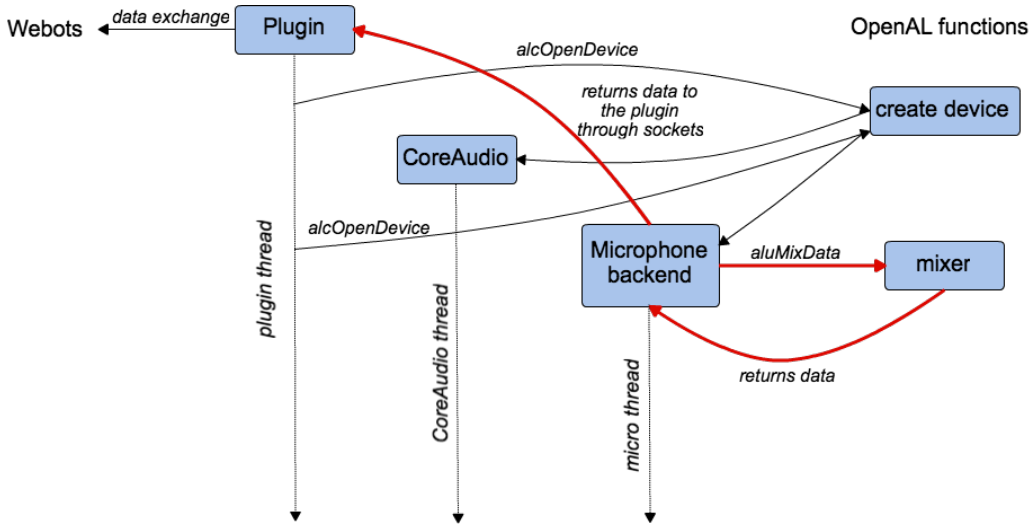


Figure 3.3: General flow of the devices using the Microphone backend

3.2.5 Mono and Stereo differentiation

Robots are often equipped with microphones that can either work in mono or in stereo (a stereo microphone is basically two mono microphones set together to work as one). I will start by describing how mono and stereo modes work in OpenAL then explain how to change from one to the other.

First of all, it is important to note that OpenAL is capable of handling sound in mono, stereo, quad, 5.1, 6.1 and 7.1. However, I am only going to discuss mono and stereo but it should be kept in mind that it is very similar for the other modes. A robot equipped with a sole mono microphone can only hear 'plain' sound. There will be no sense of direction whatsoever and the only aspect that would change between different configurations of speaker/listener positions is the volume level of sound. To have a sense of three dimensional space, the robot would need at least two mono microphones or one stereo device. The difference of phases and intensities between both microphones can be interpreted to make a 3-D map of the incoming sound. The brain basically does the same with the input from both ears, just in a much more complicated and precise way[8].

In OpenAL, a listener is not thought as a microphone, that is just why I use them for in Webots. It represents more often the position of a character being played in a video game which would indeed need to be in 3-D. It means that OpenAL does not care too much about what mode is used and act as if the listener would have two 'ears' by default. To create this illusion (the listener is described as a single point in space), OpenAL put two 'ears' at the listener's position but oriented differently like this :

```
SpeakerAngle[0] = F_PI/180.0f * -90.0f;
SpeakerAngle[1] = F_PI/180.0f * 90.0f;
```

These two angles represent both 'ears' of the listener and are then adapted in correspondance to the angle with the speaker (in other words, where the speaker lies according to the two directions pointed by the 'ears'). The result will be a single buffer containing interleaved data from both sides. If the chosen format is `AL_FORMAT_STEREO8`, the buffer will contain 8 bytes from the left side, followed by 8 bytes from the right side and so on.

As great as it is to simulate stereo for the user, we might want to use only mono microphones. My simulations, for instance, were solely run with e-puck robots. These robots are equipped with mono microphones and one would like to be able to test them as they are in real life. This parameter only concerns microphones devices since we are more than happy to have a

playback backend working natively in 3-D. The thing is, it is not possible to give a parameter in the simulation to decide whether to use mono or stereo and the only way I found to achieve this is to add the following line in the `.alsoftrc` file :

```
channels=mono
```

This way, the devices will be created with a mono listener. This command does not affect the playback backend so the simulation is exactly how we want it to be, with the playback in stereo and the microphones in mono. The only drawback is that it is not possible to make a simulation with robots equipped with both mono and stereo microphones, or at least not with this solution.

It is also maybe worthy to mention that sound sources are natively omnidirectional but this is something that can be set within OpenAL so if the user wants to specify special aperture or directionality, it is possible for him to do so.

3.3 Extensions to the swis2d plugin

In this section, I will describe the changes I brought to the plugin so it is better suited for our needs and how it basically works.

3.3.1 Initialization

As a remainder, most of the functions are automatically called by Webots when needed and the user does not have much control on that. When the simulation world is set (when you load the world or press the 'revert' button), multiple functions will be called such as :

```
webots_sound_init()

webots_sound_add_speaker(...)

webots_sound_add_microphone(...)

webots_sound_add_obstacle(...)
```

There is a serie of functions similar to the last one that exist to initiate all the obstacles of the simulation and their respective bounding boxes **(See figure 1.1**. All of these functions are left blank as I do not need to bother

about obstacles and lines of sight.

The `init` function, as its name indicates, initializes some simulation parameters and is also responsible for the creation of the playback device. This is the device that will use CoreAudio so the user can hear what sounds are being played in the simulation and also from where the sound comes. When setting the listener, one should give him a position, a velocity and an orientation. This listener being, as I call it, the 'world' listener (because it is merely an observer of the world surrounding him), is simply placed in the middle of the scene on the ground level. This way, the user can have a nice impression of where the sound speakers are located according the this specific point, which position is easy to visualize in the scene. The initial idea was to use the camera's position instead of some specific location so the user can navigate through the scene and act like if he was part of the simulation himself. However, the plugin has no way to query the current position of the camera making this feat impossible. This could go in the list of aspects to improve but it is not that problematic since it will be easily fixed when this sound process will not lie in a separate plugin anymore. As for the velocity, it is of course null and the orientation uses the default orientation of OpenAL which is {(0, 0, -1), (0, 1, 0)}.

The two other functions I mentioned are `add_speaker` and `add_microphone`. These functions will respectively create `SoundSource` and `SoundReceiver` objects (more on this on the next subsection) and not do much else. We cannot create the microphone devices in here, which is kind of counterintuitive, because once created, devices will immediately start working. It means that these devices would start recording sound before the simulation is launched. In the second function, the sockets are initialized and start waiting on a client.

3.3.2 Sound objects

In the interface of the Webots' sound plugin, two structures are defined : `SpeakerRef` and `MicrophoneRef`. These objects contain informations related to the speaker or microphone they represent such as their object reference or their 2-D positions. They are used in swis2d as the main objects manipulated into the simulation. In the new implementation, I do not have much interest into 2-D coordinates positions so I directly prompt Webots to obtain their positions with the following function :

```
webots_sound_get_translation(ObjectRef);
```

I still use `SoundObjects`, however, to keep track of their Webots' references

since I need them to get the positions. In other words, I am linking the objects interacting between Webots and the plugin `(SoundReceiver)` with OpenAL objects `(ALCdevice)` by creating one instance of each and using them jointly depending on the operation that has to be performed. One has to be careful to not confuse the positions of robots with the positions of devices. Even though they are very close, the difference of position between two microphones on the same robot is what makes it possible to localize sound sources.

Additionally, I also use `SoundReceiver` objects to handle sockets as I will discuss in the next subsection.

3.3.3 Inter-thread communications

When devices are created in the plugin, they are launched in a separate thread. They will follow action according to their own code and will no longer interact with the main application thread. Their main loop will be activated, constantly trying to mix sound sources, until they receive a signal to shut down. This means there is no way to communicate back with the plugin. As explained earlier **(See Section 3.2.4)**, the Wave File Writer backend used to instantly write to a file anything the mixer would output. The Microphone backend was then created to overcome these communication issues. The easiest way was to keep the same structure of operations. The new backend would still keep the same mechanisms, looping and mixing sounds currently played, but instead of writing its output to a file, the data would be sent through a socket to the corresponding `SoundReceiver` object in the plugin.

Even though I am using them as pipes (I only send data in one direction), I am using 'Unix Domain Sockets' because they seemed to fit my needs well enough. These sockets work with a server and a client. The server gets binded to a socket (a file on the hard disk) and waits for some client to connect. As already explained, this part happens on the initialization functions of the plugin. As a client, the protocol is similar. It first binds to a socket and then tries to connect to a server on this socket. If the connection is accepted by the server, the exchanges can begin. Following is an example of how the microphone backend tries to connect to a socket :

```
device->szDeviceName = strdup(deviceName);
ALCint cntdev = alcCountDevicesWithSameName(device);
char numbered_socketname[1024];
sprintf(numbered_socketname, "%s%i", socketname, cntdev);
int len;
```

```
struct sockaddr_un remote;
if ((data->s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
   perror("socket");
   exit(1);
}
printf("Trying to connect... \n");

remote.sun_family = AF_UNIX;
strcpy(remote.sun_path, numbered_socketname);
len = strlen(remote.sun_path) + sizeof(remote.sun_family) + 1;
if (connect(data->s, (struct sockaddr *)&remote, len) == -1) {
   perror("connect");
   exit(1);
   }
printf("Connected. \n");
```

The first four lines show how the backends knows which socket he has to connect to and the rest is just a standard connection protocol. It will create as many socket files (mysocket0, mysocket1 and so on) as there are microphones running in the simulation.

There are two additional important aspects worth noting. The first one is that all this communication process is not needed for the CoreAudio backend since it does not need to leave any trace of the content of the buffer it used during the simulation. The second one is a problem of synchronization. If the simulation is running too slowly, the backends will indeed entirely fill their buffer since they keep on mixing all the time. At some point, an error will occur or we will just lose data because the buffer could not have sustained such an important flow of bytes. The opposite would not pose any threat for the well-being of the application, the backend just could not fail because of sending too few data.

The solution was then to change the mode in which the client operates and use the socket in a 'blocking' way. It means that the backend will try to send data and then wait for the server to receive it before proceeding any further. This approach might create trouble if it was so slow that it could not have time to process the sound from a fixed window of time during that same amount of time. This is not the case so we do not have to worry about that. On the opposite side, the server is non-blocking and just reads as much as he can before passing it to Webots and eventually the controller.

Next is a figure depicting the flow of operations in the new plugin to summarize what has been said in this section :

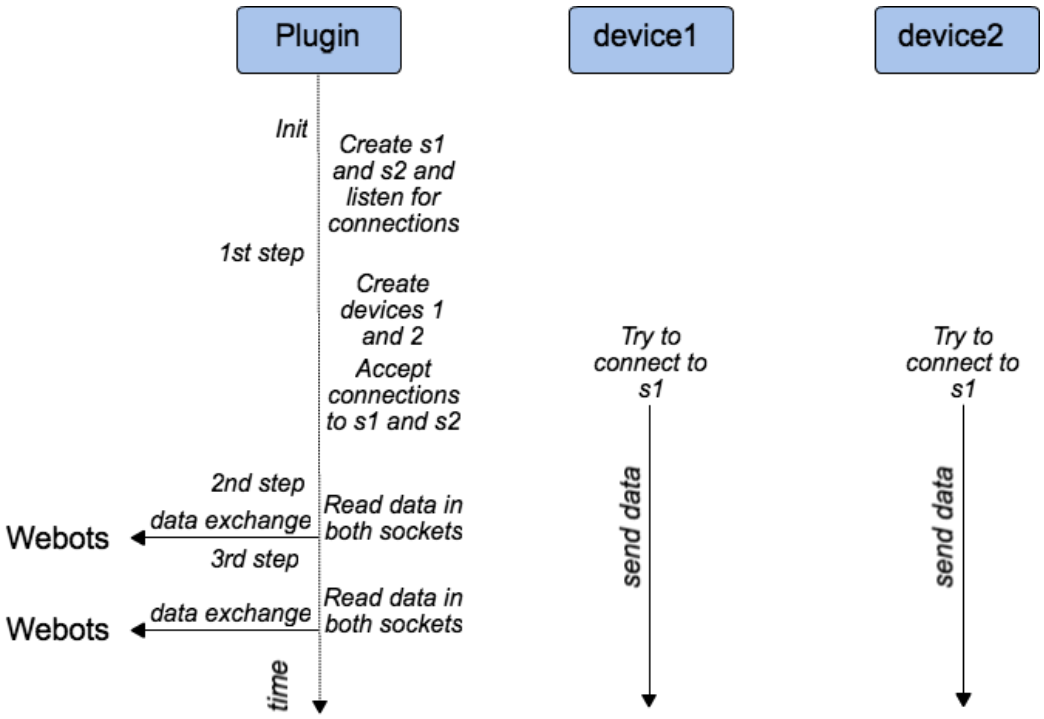


Figure 3.4: Communication triangle between Webots, the plugin and the backends

The three-way handshake initiated in the first step is clearly exposed followed by the data path. What is meant by 'data exchange' is that each microphone will receive the corresponding data in the controller.

3.3.4 Emitting and receiving samples

We have seen that to emit samples, the controller must call the function

```
wb_sound_load(...)
```

that will, in its turn, call the function

```
webots_sound_speaker_emit_sample(...)
```

in the plugin. This function is responsible for creating the buffers and sources that will be needed for sound playing. As already mentioned, one source and

one buffer need to be generated for each device present in the simulation. Once this task done, its parameters have to be set. Sources have way more possible parameters than the listeners as we can set its pitch, its gain or even decide if we want the sound to loop indefinitely (for ambiance sounds for instance). Then the sources just have to be launched and all the devices should receive the same sound in their own context. It is important to be careful to create the sources in the correct context corresponding to the correct SoundObject corresponding to the correct microphone. It is easy to perform operations in the wrong context and end up with no sound at all.

This is also where I cheated by setting formats and frequencies by hand because I know in advance what sounds will be prompted to be played. As for the receiving part, this is implemented in the following function :

`webots_sound_microphone_receive_sample`

It will first find the `SoundReceiver` object corresponding to the correct microphone. It will then read everything that there is to read in its associated socket. Let us note that the write and read calls on the socket always treat chunk of data of 8192 bytes. This is the default update size that was used in the Wave File Writer backend and I kept it that way because I did not feel the need to change that. This function will then maybe read 0, 8192, 16384 or even more bytes in one iteration but it is always going to be a multiple of 8192. This buffer will then be returned to the controller where the user can use it as it pleases him.

As I am writing the report, I am realizing that it was probably not the best approach as the user has to call this function often enough if he does not want to see the backend freezing. Remember that the backend works in blocking mode so if this function is never called, the socket will never be emptied and the backend will stop working until the plugin reads data. I chose this implementation before I started calling this function every step anyway but the socket emptying could be done in the `step` function instead and when the user wants the sound buffer, just one call would suffice to get all the data up to this point. The only drawback would be that the buffers could reach their maximum size and crash the program if, once again, the user never calls the fonction to retrieve sound.

3.3.5 Run cycles

The step function is called at each iteration of the main program. During each step, the positions and speeds of every microphones and speakers are updated. This means that the minimal time between two such updates is the

time between two iterations. The only way to be more precise by updating it at a faster rate is to reduce the `TIME_STEP` variable in the controller.

There are then three cases. On the first step, all the operations that could not be carried out in the initialization, for already detailed reasons, will be made. It includes the creation of the microphone devices and all the settings related to them like listeners parameters or the final step of the three-way handshake between the server and the client on a certain socket.

All the following steps will act as described at the beginning of this subsection, it just contains updates.

Finally, on the two hundredth step (this number was picked at random, I could have taken five hundred or five thousands), I am ending the simulation. The fact is that pressing the pause button does not induce any calls in the plugin so there is no way for the user to tell it to stop and it will run until the simulation is reverted. I decided to set a duration for the simulation, setting a maximum number of steps so it does not continue to consume CPU and memory ressources for no reason if the user forgets to revert the simulation (I managed several times to fill my hard drive entirely before having errors because I did not have enough space to write anymore). This is, of course, easily modifiable and depends on personal preferences. When this final step is reached, the plugin will then call another `cleanup` function that will take care of what has to be done but more on that in the next section.

3.3.6 Deinitialization

From the preceding subsections, it is clear that I am using two cleanup fonctions. One is called when the maximum iteration number is reached and the other one is automatically called when the simulation is reverted. They both act according to whether the simulation was run or not because this is where the difference lies. For instance, the devices being created in the first step, the OpenAL objects do not need to be taken care of if the simulation is not run because they will not be created in the first place.

Deinitialization in itself is pretty basic and would not require a subsection if it was not for the sockets. It is indeed tricky because sockets cannot be closed in a generic manner depending on the state of the simulation (unstarted, running or finished). In fact, the socket cannot be closed if there are still data that were written but are still waiting on the server to be read. It means that the device has to be notified to stop sending new data before it is actually closed. How I attained this goal might seem a little sketchy at first but is, in fact, very practical. In OpenAL, the `ALCdevice` structure contains a `BackendFuncs` field. This field is another structure containing pointers to device related functions. Instead of messing up with the library, having to

create and include additional headers and so on to be able to call a function of the backend from the plugin, I included my function into this `BackendFuncs` structure. As the pointers are the same for every type of backend (core, alsa, wave, micro and so on), this structure contains the references to every possible function a backend could have. My backend being of the 'playback backend' type, it does not have functions that it would have if it were a 'capture backend' like `OpenCapture(...)`. It means that there were functions pointers ready to use so when I call `alcStopMicro(...)` in the plugin to tell them to stop writing in the sockets, it actually calls the capture opening function of my microphone device which should in theory not exist but is actually filled with the code purposed to interrupt data sendings.

When this is done, and before closing the socket for good, the plugin is flushing the sockets by reading everything that remains in them. This data will be lost but it is not really important as it is a really small portion of data (most of the time there is nothing to read but in rare cases, it represents 8192 bytes) that happens at the very end of the simulation. The difference is like if I chose to run the simulation for 199 steps instead of 200.

3.3.7 Real time and simulated time

I have already discussed the importance of being able to start the application in simulated time. When running simulations, the user is often more interested in the results than in what goes on during the simulation itself. He then needs to be able to run as fast as his computer will allow him instead of waiting on the program to take as much time as would a real life scenario. Webots has an option that allows this and we want to be able to use it.

The backends are set on a real time model since OpenAL is usually used for real-time applications (gaming mostly). It means that they will only treat small portions of data at a time and then wait until enough time has passed so it can read some more data. This is actually easier to go from real time to virtual time since these restrictions just have to be removed (while being careful to mix data in a way that makes sense). The sockets, being blocking in one way, are responsible for the synchronization with Webots. The backends are going to wait on the plugin to catch up and the rhythm of steps will then be declared by the speed at which the plugin manages to read the data from all the microphones in addition of performing all the operations described in this chapter.

3.4 User related functions

As already mentioned several times, we want to make the user's life as easy as possible. I discussed already how he should be able to send and receive buffers of data but I did not talk about how this user would get the buffer to start with. When reading a sound file like a .wav, the headers of the file need to be read first so we can get crucial informations about its format before finally extracting the actual sound data. The following figure depicts the structure of a .wav file.

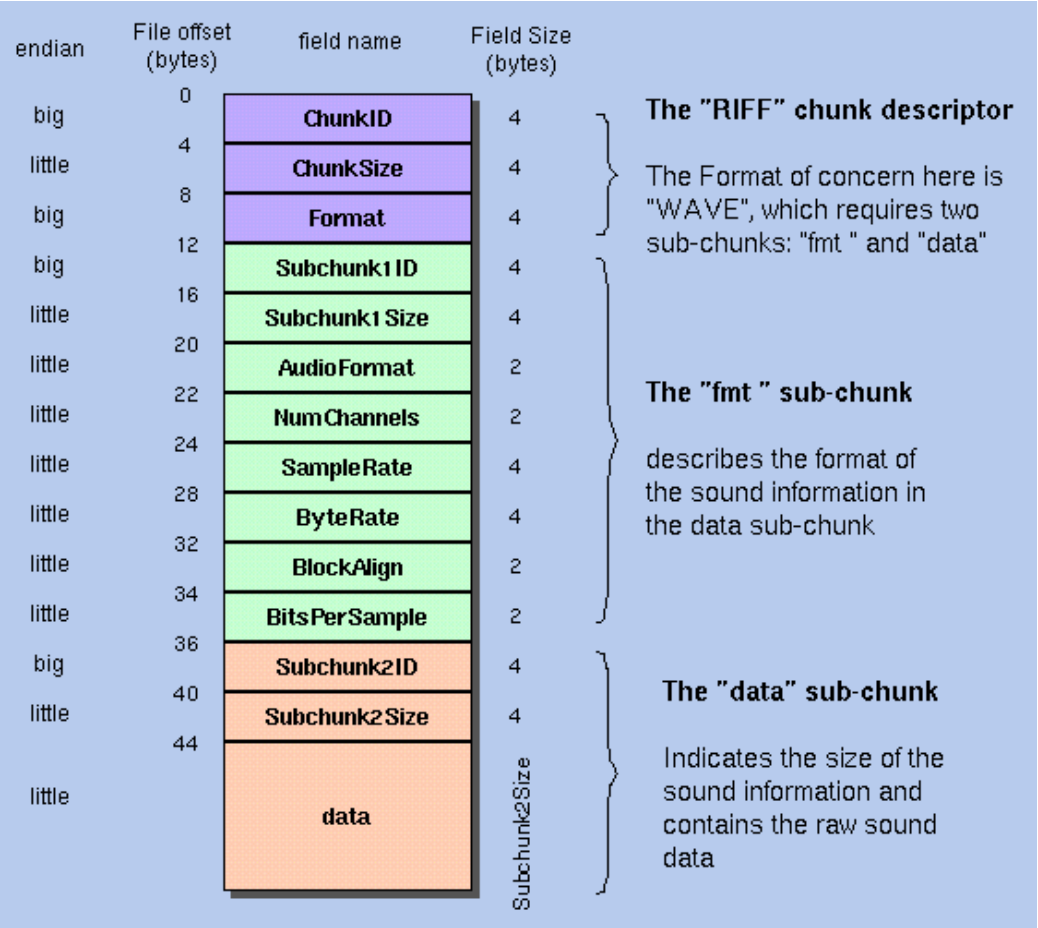


Figure 3.5: Canonical wave format.[7]

To help the user, I am providing him two functions. The first one is used to load a sound file so he does not need to worry about correctly extracting

important values himself. One just needs to make the following call :

```
unsigned char* data = wb_sound_load(loadFileName, fileSize);
```

to have the sound buffer as `data` and its size stored in the `fileSize` variable. This function will be used every time a user wants to play a sound in the simulation.

The second one is not a mandatory function. It is the opposite from the first one, it allows the user to convert a data buffer into a playable sound file. This is not something that is completely part of the assignment since we do not know what the user would want to do with its samples and it is up to him to code his functions handling this buffer. Nevertheless, it may be useful to some users and it also turned out to be useful for debugging purposes. It is nice to be able to hear what the mixed sources would sound like from the robot's perspective. This is achieved by calling the following function :

```
wb_sound_save(saveFileName, dataBuffer, dataSize);
```

3.5 General comments

Now that everything has been covered, I will summarize how this simulation process works with the help of the following figure depicting the thread map of the whole application.

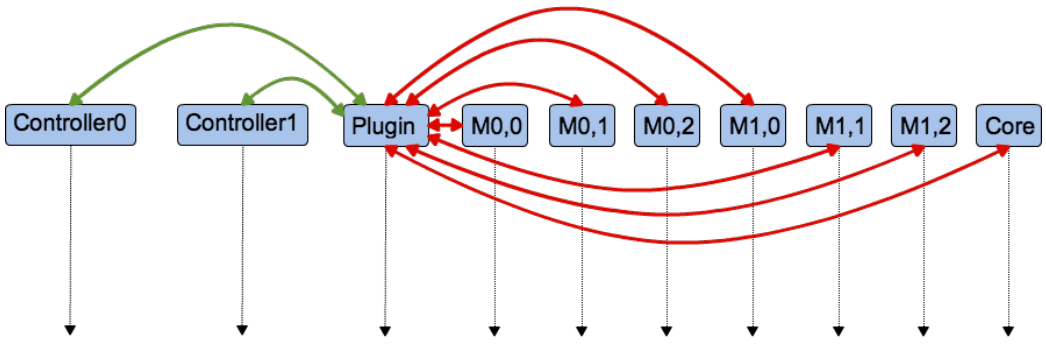


Figure 3.6: Global thread map.

This figure is an example of a simulation containing two robots, each possessing three microphones (I always tested with robots having no more

and no less because I am using the e-puck robot model and it has three). On the left side of the plugin are the controllers' threads. Even though they use the same controller, they each run the same code in their own thread (they even have their own processus to be exact but this does not matter). On the right side are the backend threads. 'Mx,y' means it represents the yth microphone of the xth robot. Since there are two robots that each have three microphones, there are six such threads. The one on the extreme right is the CoreAudio thread, responsible for the playback sound.

The arrows represent interactions occurring between two actors. When the arrow is green, it means that there is an exchange based on an operation initiated by the user by calling one of the following function :

```
wb_speaker_emit_sample(...)
```

```
wb_microphone_get_sample_data(...)
```

and when the arrow is red, it means that the operations are automatic and cannot be called or prevented on demand.

Looking at this figure, it looks apparent that the plugin is doing a lot of work, transferring data from the controller to the simulation and vice versa.

Chapter 4 Results

In this chapter, I am going to present my results. In the scope of this project, there are not many types of results that we can get. When a simulation is started, it is possible to listen to the sound playback and then listen to the .wav files that were created by each microphone if that is what the user decided to do. It is very easy to notice if the different sounds are coming from the right direction, occur in the correct order or if the sound seems to come from further or closer but to prove that the simulation runs exactly like it should in real life, one would have to create a simulation world and its real life counterpart and they would need to be exactly alike. It is then possible to compare the resulting buffers in both situations and estimate how close the simulation behave compared to reality. This would take an enormous amount of time so I am only going to present results that could be inferred in a reasonable amount of time.

4.1 Performance

I initially intended to launch tests with different numbers of microphones to see how much the application would slow down with an increasing number of devices. Then, after a few tests, I realized that it did not make much sense because even one lone microphone should use the CPU at its maximum (with simulated time). All microphones are performing exactly the same operations and the plugin has to work as much for each one of them so it would seem logical for the decrease in speed to be linear. My few tests consisted in a simple scene with three, six or nine microphones and a set number of steps of one thousand. The results confirm the theory since it took an average of about four, seven and eleven seconds respectively to complete the simulation.

The second aspect we can take a look at concerns the simulated time and how worthy it is. This will be extremely dependent on the machine since simulated time will use CPU's at their maximum potential. Then, the numer of CPU's and their speed can make the results vary a lot. It can also

be very different between two executions of the same simulation depending on what else runs on the computer. For this test, I tried to shut down every useless process to avoid such disparities but it is still a bit random. I tested the same scene with three, six, nine and twelve microphones (I am always testing with multiples of three since adding a robot in the simulation adds three microphones) in both real and virtual time. The results can be found in the following figure :

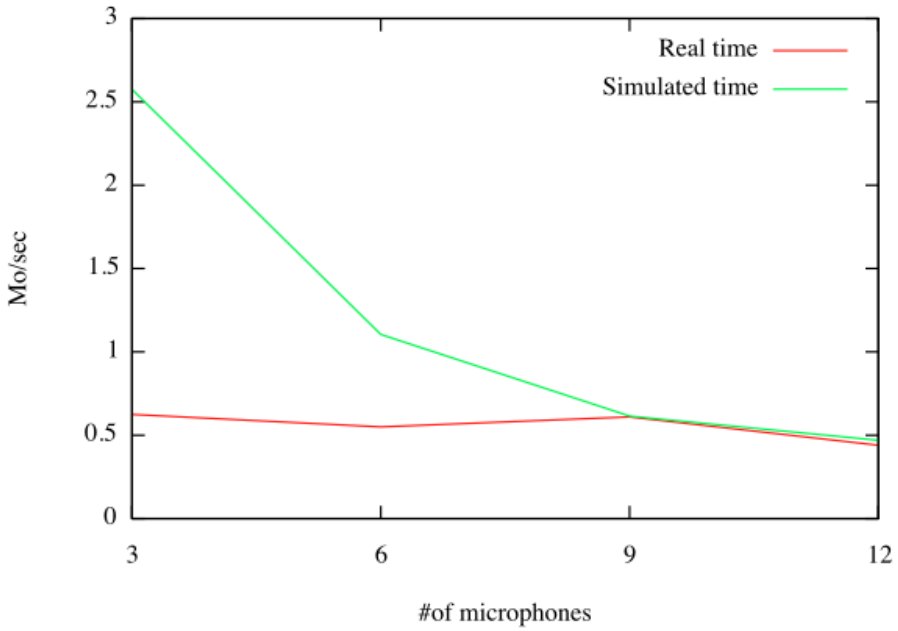


Figure 4.1: Real time vs simulated time.

The 'y' axis represents the ratio of `Mo` per second of sound data that is recorded by the microphones (in average). With only three microphones, using the simulated time represents a true upgrade but its advantage over real time fades quickly when we add additional microphones. For the real time trials, it does not change too much which is logical because when there are not many microphones, the devices will often sleep until enough time has passed to compute more data. We can see that with twelve microphones, it becomes really slow and the simulation's speed even went under the real time speed on occasions which means the calculations take too much time to keep up with real time. This is annoying as the number of robots in the simulation quickly becomes limited. As a comparison, I tried running the same kind of

scene but outside of Webots and with the Wave File Writer backends (which writes to a file instead of communicating through pipes) and the results were way more impressive. For instance, with three microphones, ten secondes of simulation allowed to simulate about six minutes of sound which is about sixty `Mo` of data or a ratio of six. It means that having to read through the sockets and communications between the plugin and the controller via Webots is pretty consuming. It should not be forgotten that when you have twelve microphone threads sending data through their respective socket, you have only one thread, the plugin, to read them all. It seems logical that it quickly becomes saturated.

In conclusion, simulated time is not that good, especially when many microphones are present but can offer a nice speed boost when only a few microphones are used.

Chapter 5 Further work

At the time of writing of this report, it appears that some additional elements should be integrated to my work. Some of them were not initially part of my project and others are aspects that I did not have time to investigate enough. I am currently working on implementing some of these features but for the others, they will need to be carried on on the future.

5.1 Collision sound

This first aspect is very crucial. It relates to the sounds that robots or other bodies would provoke in the simulation. When a robot walks, the collision between the foot of the robot and the floor will emit some noise. This sound will be depending on the matter of both surfaces (a robot will obviously make less noise walking on grass than walking on a floor made of metal) and the speed of the collision. Webots has a node called 'ContactProperties' that contains the fields to set some parameters about these contacts. Here is how this node looks like :

```
ContactProperties {
   field SFString material1 "default" # some material
   field SFString material2 "default" # another material
   field SFFloat coulombFriction 1 # Coulomb friction coefficient
(-1 stands for infinity)
   field SFFloat bounce 0.5 # range between 0 and 1
   field SFFloat bounceVelocity 0.01 # (m/s)
   field SFFloat forceDependentSlip 0 # force dependent slip (see
ODE)
}
```

We could add a field containing the name of a sound file corresponding to an impact and play it each time such a collision occurs. This is actually not possible to achieve in the plugin since it does not have any form of control over the collisions. It would have to be modified so Webots calls a fonction

from the plugin each time a collision occurs so sound could be played through OpenAL. As a remainder, I could not modify Webots' source code as I had to use swis2d's structure.

This is really important since the real life trial would be 'noised' by these kind of sounds which differs from what actually happens in the current state of the plugin.

5.2 Cross-platform adaptation

As Webots is currently running on Windows, Mac OS X and Linux, it is then important to have the possibility to use this plugin on all those platforms. It was even one of the strong motivation to use OpenAL. Even though I did not make any test on other platforms because it was not a priority, it should not create too much trouble since my principal problem on Mac OS X got fixed when the CoreAudio backend was added to the library. Backends for Linux and Windows already exist (ALSA / OSS, WinMM) and should be fully functional. My plugin can be tested on different machines by making a simple exchange in the `.alsoftrc` file. If one wants, for instance, to test it on Linux, the following line :

```
drivers=core,micro
```

should be replaced by this one :

```
drivers=alsa,micro
```

I am not going to assume it would work until tested but it should not require an enormous amount of work.

5.3 Sound propagation modelization

When sound travels between two points, not only is it attenuated by the distance but it is also delayed depending on the environment. For instance, sound travels at the speed of about 343.2 m/s in air (depending on the temperature) and about 1484 m/s in water[6]. OpenAL handles distance related sound attenuation but is unfortunately not capable of delaying the sound according to the same criterion. Robots often use difference of phases between two microphones to localize sound sources[8]. If this difference is not accounted for, robots would just receive sound on all of their microphones at

the same time and it would be rendered impossible for them to locate sound sources. Thus, this point is critical and needs to be resolved. I could not do it in time but I am currently working on it.

Distance between the speaker and the listener can easily be calculated in Webots. The time the sound would take to reach the microphone can then be calculated using the speed of sound. Using the frequencies, format and so on that I am using in the plugin, the number of bits needed to create one second of real sound can be deduced. The last step is to shift the output of sound sources by this amount (calculated independently for each source) by adding a corresponding number of 0's before the sound begins to play and the goal should be attained.

5.4 Playback in simulated time

We have already seen that I did implement virtual time but there was one aspect that I did not mention and is missing from my implementation. As a remainder, my backend used by the playback device is CoreAudio. This backend is working with hardware and cannot be modified to work in virtual time since its purpose is to play live sound. What we can do, however, is to accelerate the sound sources linked to this device according to the current speed of the simulation. This way, we will have the illusion that the sound is synched with the speed of the simulation as it will sound accelerated. We just need to keep track of the time between two steps of the simulation and set the sound's frequency accordingly. It is important to note that once the frequency is set, it cannot be modified and is going to play for its entirety at the same pace. If the computer starts running slowly at some point in the middle, the sound would not reflect the speed of the simulation anymore but this is only an indicator of this speed anyway so we do not really need it to be ultra precise. I am also currently working on this problem.

5.5 Occlusion and reflection

One great feature of swis2 is missing in OpenAL : its handling of occlusion and reflection. It means that if you put obstacles in the virtual scene, OpenAL would just ignore them and play the sound through them anyway and would then not use surroundings to bounce back and circumvent these obstacles. This is quite a setback and would require quite some work to be implemented. In fact, even if the path finding algorithm of swis2d were adapted for three dimensional space, the question of how the sound would

bounce from object to object would still need to be answered.

Luckily, OpenAL has an extension to help us face these challenges. The 'Effects' extension is designed to provide a generic framework for adding advanced effects. One can create different effects by adding filters corresponding to wanted effects. It works pretty much in the same manner as the sources and buffers. Effects are created and linked with filters that are, in turn, linked to the sources. The two figures on the following page picture the change in OpenAL's architecture with or without the extension.

Amongst the available effects, we can name reverberation, chorus, echo, distortion and many more.

We could use these effects to simulate occlusions and reflections but we would still need to run a complex algorithm to decide how to parametrize accurately these effects to correspond with the scene. For instance, you can imagine that sound would partially go through a wooden door but not a massive wall made of concrete.

This was originally not part of the project but would have been developed if time had allowed it.

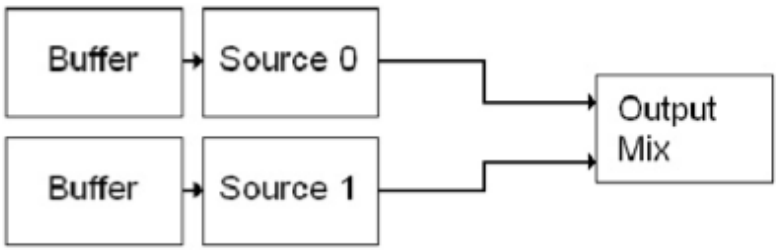


Figure 5.1: Basic OpenAL architecture.

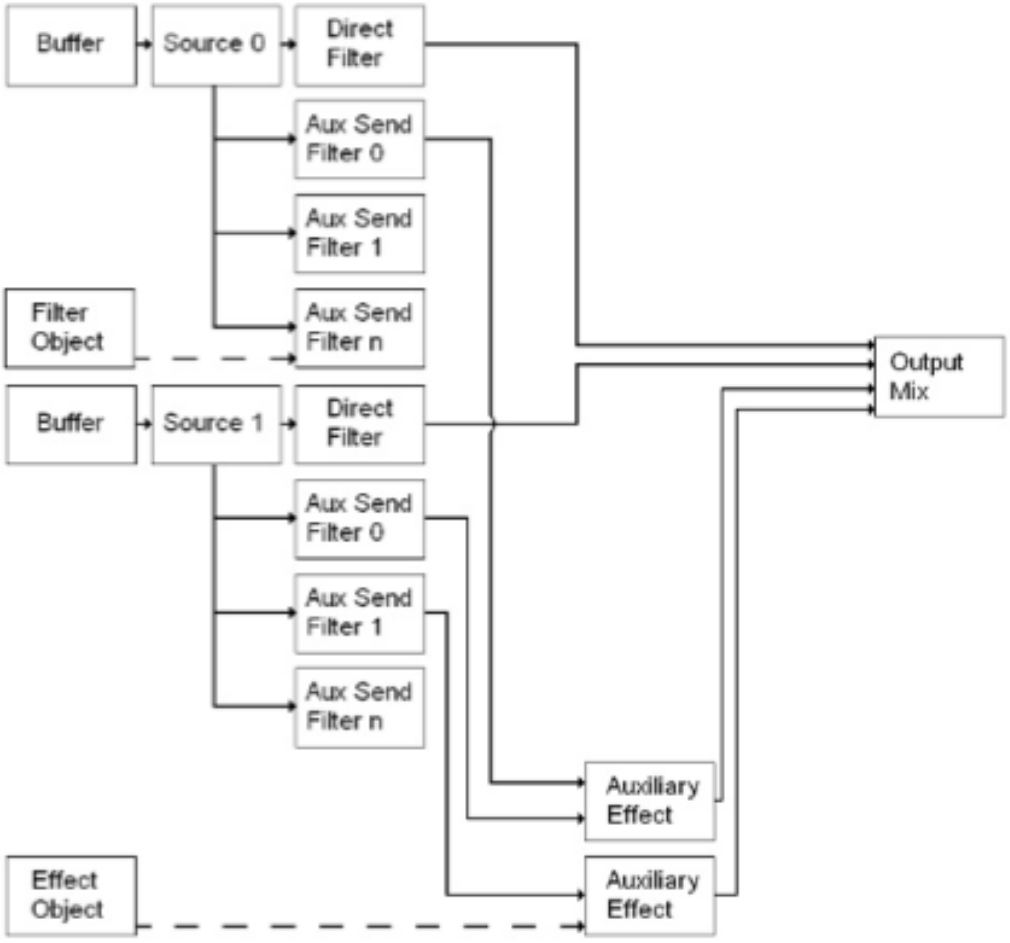


Figure 5.2: OpenAL with 'Effects' extension.

Chapter 6 Conclusion

At the end of a project, it is important to take a look back at the initial objectives to see what was accomplished. I am going to take them in order and make a point with the current situation.

- Realism of sound : most aspects are taken into account, the only missing one being the distance related delay.
- Comprehensive for the user : it is really easy for a user to use this plugin so this is acquired.
- Sound playback : this part is also fully functional.
- Multiple robots handling : this part is also complete. If I had to work more on this part, I would try to improve the performances. As seen earlier, it seems hard to make it work correctly with more than twelve microphones so I would try to reduce calculations' costs to improve this number.
- Running modes : my plugin is capable to run in simulated time but the adaptation of frequency for the playback device would need to be implemented. In its current state, an accelerated simulation having several sound played close in time would sound a bit messy.
- Cross-platform : this point has absolutely not been taken care of but this is very important nonetheless and should be done.

A bit of work could also be done on my code, which is probably not as optimized as it could be since it was not the priority but it could improve both the simulated time results and increase the number of microphones we could use at the same time.

In conclusion, there is now a nice base to work upon that includes most of the important features but since there are still some missing elements, this project cannot really be classed as fully finished for the moment.

Acknowledgments

I would like to thank Olivier Michel and Cyberbotics for giving me the opportunity to work on a subject that has such concrete possibilities. It is a strong source of motivation to know that your work will probably be used by other people in one form or another. I would also like to thank Pr. Alcherio Martinoli for accepting to be my supervisor and Amanda Prorok for all the help.

Bibliography

[1] Daniel Peacock, Creative Technology Ltd (June 2007). 'OpenAL's programming guide : OpenAL versions 1.0 and 1.1'.

[2] Jean-Marc Jot (undated). 'Efficient models for reverberation and distance rendering in computer music and virtual audio reality'

[3] Jean-Marc Jot (10th International Conference on Digitial Audio Effects, Bordeaux, France, September 10-15, 2007). 'Efficient description and rendering of complex interactive acoustic scenes'.

[4] Mike Wozniewski, Zack Settel & Jeremy R. Cooperstock (Proceedings of the 13th International Conference on Auditory Display, Montreal, Canada, June 26-29, 2007). 'User-specific audio rendering and stterable sound for distributed virtual environments'

[5] Anne van Rossum (as in March 2012).
'http://www.almende.com/replicator'

[6] Unknown writer (as in August 10, 2012).
'http://en.wikipedia.org/wiki/Speed_of_sound'

[7] Scott Wilson (Jan 20, 2003),
'https://ccrma.stanford.edu/courses/422/projects/WaveFormat/'

[8] Jens Blauert (1997), 'Special Hearing - Revised Edition : The Psychophysics of Human Sound Localization' 137-177.