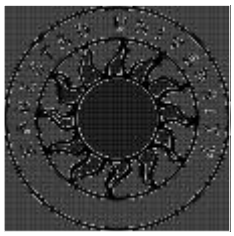

Computer Science

Johan Engdahl

An Evaluation of 3D Sound APIs

Bachelor's Project

2002:31

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Johan Engdahl

Approved, 2002-02-15

Advisor: Mari Göransson

Examiner: Stefan Lindskog

Abstract

This thesis work has been performed for Saab Bofors Dynamics as a bachelor's exam at the University of Karlstad. The thesis contains documentation and a review of a number of different sound Application Programming Interfaces (API:s). The API:s are graded according to a number of criterions, each with a different level of importance. The API that passed the criterions best has been chosen for the implementation.

A special thanks to Choong-Ho Yi, who has been tutoring this exam.

Contents

List of Figures

List of tables

1 Introduction

This project has been performed for Saab Bofors Dynamics and involves 3D-sound for training systems, simulations and visualizations of a number of different weapon systems. Saab Bofors Dynamics wish to have an Application Programming Interface (API), which can position the sound and add sound effects to the simulations and visualizations so that it represents the natural outdoors environment where the weapon systems are used. The task was to find an API suitable for this task. An API is a programming interface, which simplified can be described as similar to the programming language Java. The similarity lies in that both Java and the API contains predefined code to achieve any necessary call made by the user. This gives an API a higher level of abstraction than an ordinary programming language.

1.1 Visualization and simulation

Visualization and simulation are two concepts that will appear frequently throughout this thesis. The following part is intended to clarify the difference between the two concepts and when they are used.

1.1.1 Visualization

The concept is to visualize calculated data. That data can be generated either from real test flights or from a track simulator where the objects' (airplane, tank, missile etc) behavior is calculated and logged in a file. To visualize the track simulator one simply reads from the file and uses those coordinates to position the objects.

In a visualization, one never interacts with the course of events that a simulation allows. The possible adjustments are to change the speed of the visualization or to change the view, e.g. from the missile to the target.

1.1.2 Simulation

A simulation is a scenario where a person (Man in the loop, see below) or a weapon system (hardware in the loop, see below) interacts with each other in a simulated environment.

Man in the loop

The simulation is a game played by the person using the simulator. A simulated environment is used for training. The person faces a real scenario in a safe and inexpensive environment. Another advantage compared to real man-to-man combat is that it is easy to achieve data for evaluation (how the person handle a critical situation etc).

Hardware in the loop

The hardware is tested towards a simulated environment, which simulates indata to the hardware. The hardware (a weapon system remade for computer use) then generates its response and affects the simulated environment. Hardware in the loop is used to test Weapon system in a more simple and inexpensive way and is also a complement to the real test shootings.

1.2 Background and goals

Saab Bofors Dynamics wants to present and demonstrate their different concepts through simulations of their weapon systems. The simulations are used to test and improve weapon systems under the construction phase, as well as to market and demonstrate the finished product. The experience is enhanced when sound is added to a visualization. At the same time, the demand of having sound effects in the visualizations has increased, leading to this thesis work. It is the purpose of this project to examine what software API:s will be suitable for use at Saab Bofors Dynamics.

1.3 Requirements

This section presents the requirements deemed important for the API to fulfil. These requirements are used to determine which API to use. The requirements are listed according to the order of importance. The level of interaction is the most important requirement and is therefore listed first.

1.3.1 Level of interaction

Describes how well the API will interact to the existing code, i.e. how hard it is to add the code from the API into the existing code. Will it be unnecessary complicated to integrate an API with existing routines and structures?

1.3.2 Level of abstraction

The higher level of abstraction an API has, the more control is moved from the programmer to the API. As a developer not using any API, one would have to add support for different hardware, e.g. sound cards. If an API is used, it checks what kind of hardware the computer uses and call the functions required in order to make the sound card work properly. The developer never sees this since it is on a lower abstraction level. An API can also have integrated support for other API:s if it is on a higher abstraction level.

A higher level of abstraction renders in easier programming and faster development of the system. The higher level of abstraction also requires much less effort and is much easier to apprehend and use.

The disadvantages of a higher level of abstraction are that the control is encapsulated and moved away from the programmer to the API. An API with a high level of abstraction provides code you would have to write in a lower level API. Though this is a good thing, in the meaning of effort, it also decreases the programmers' control over the application. One will not have full control over how the functions that the API provides work.

1.3.3 Status / Stability

How long has the company who developed the API been in the business? What is its current state: is it growing or on its way back?

1.3.4 License agreement / Cost

Is it necessary to get a license and in that case, how much it will cost?

1.3.5 Complexity

What Sound Phenomena can the API handle? How difficult is the API to use?

1.3.6 Platform support

Saab Bofors Dynamics' visualizations that will utilize these API are running in the Windows environment only. Since there are no signs of this changing, the platform independendency is not important at all (in the meaning of choosing an API).

The rest of the thesis is structured as follows:

Chapter 2 is a brief introduction to a number of sound effects and how they work. Chapter 3 covers the evaluation of a number of API:s that were considered to be of interest to Saab Bofors Dynamics. Each of the different API:s has been graded on all of the requirements.

These grades are presented in a table at the end of each section, and have the possible values of: *Poor*, *Adequate* and *Excellent*. Chapter 4 gives an example on how the API can be integrated with the existing code, presented in UML-notation. Finally the 5th chapter gives a summary over the conclusions gained through this thesis.

2 Sound phenomena

This is a general introduction over a number of the sound phenomena in natural environments and how they are approximated in virtual environments. There are a huge number of sound effects in the natural environments that all affect the sound waves. The sound effects in this chapter are supported by all of the evaluated API:s. Section 2.1 is intended to give the reader an idea of the function of these sound phenomena and how they work in the virtual environment.

2.1 The Doppler effect

The equation of the Doppler effect is as follows:

$$f' = f(V \pm V_D)/(V \pm V_S)$$

In the function above: *f'* is the Doppler effect frequency, *f* is the original frequency of the sound wave, *V* is the speed of sound, V_D is the speed of the listener (detector) and V_S is the speed of the sound source. The Doppler effect principle was stated in 1842 when the Austrian Johann Doppler discovered the phenomenon. The Doppler effect concludes the way a sound wave appears to vary in frequency as the source of the wave move towards or away from the observer. The source itself never changes the frequency of the emitted wave. The reason why the wave appears to change in pitch is due to distortion. Pitch is the rate (how often) waves encounter the observer. As in Figure 2-1, one can see how the pitch increases as the source moves towards the observer and decreases if the source move away from the observer. In the computer world, all of the evaluated API:s calculate the Doppler effect automatically. The programmer simply has to add the velocity and the position of the observer respectively the sound source in the three dimensional space. [1]

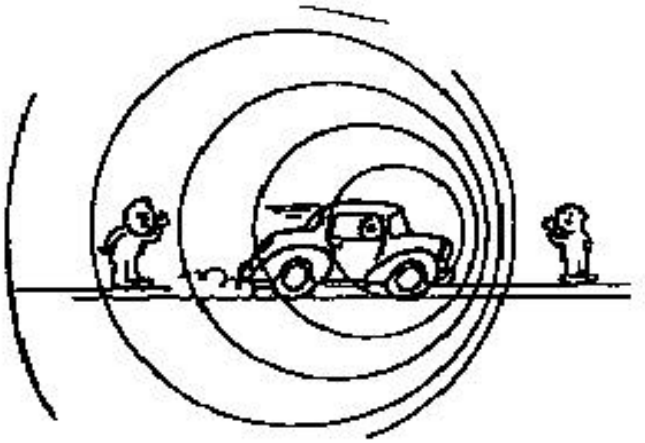

Figure 2-1: An example of the Doppler effect [2]

2.2 Reverb

The only time you hear the original sound from a sound source is when no sound reflections from the surrounding environment appear. In all other cases you will get a combination of the original sound and the reflected sound (called reverb). The amount of reverb depends on which environment the observer and sound source appear in (compare for instance a narrow isle or corridor with an open field). The ratio between the original sound and the reverbed is called the wet/dry ratio. In Figure 2-2 the wet/dry ratio is 2:1 (the amount of reflected sounds is twice the amount of the direct sound). The Rolloff factor is the rate at which reverbed sounds become attenuated with distance. To cause the sound to attenuate faster with increasing distance one simply increase the Rolloff factor parameter.

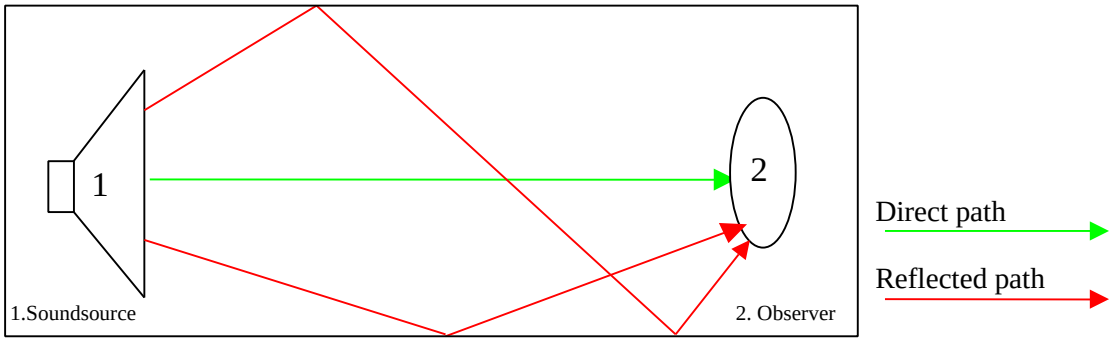

Figure 2-2: An example of the reverb effect

2.2.1 Occlusion

Occlusion is a form of reverb and occurs if there is a wall between the source and the listener i.e. no open-air path for the sound to travel through. Depending on the material and thickness of the wall, the attenuation of the sound varies (a thick wall attenuates the sound much greater than a thin).

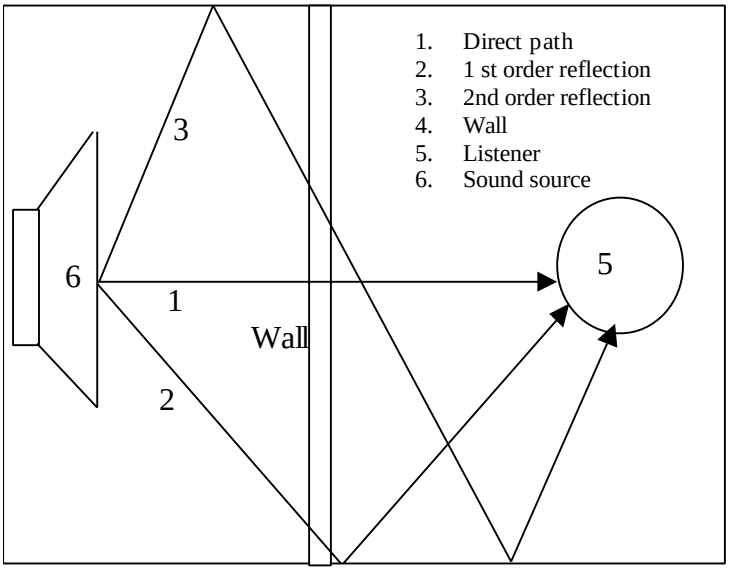

Figure 2-3: An example of occlusion

2.2.2 Obstruction

Obstruction is also a form of reverb and occurs when there is no direct path between the source and the listener. The only sound that the listener hears is the reverbed one.

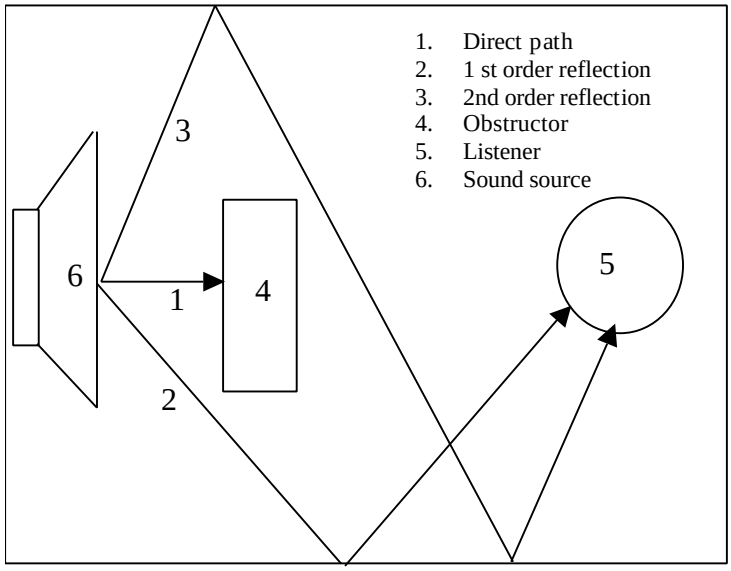

Figure 2-4 An example of obstruction

There are two ways to calculate the reverbed sound. One is described in Figure 2.4, where the obstruction and occlusion effects are calculated to create a realistic feeling. A more exact way to calculate the reflected sound is the one that Aureal uses: the wavetracing algorithm (see Wavetracing in chapter 3.1.2 for a detailed description of how the algorithm works).

2.3 Head Related Transfer Function

Small headphones that are put directly into the middle ear causes loss of the outer ears function. This will cause the illusion that the sound is coming from the inside of your head and that is no good if you want to create an authentic hearing feeling. The Head Related Transfer Functions (HRTF) was created in order to solve this problem. These Head Related Transfer Functions is achieved by measuring how long it takes for a sound to “arrive” to the left and right middle ear by placing a sound source in a number of different positions in the room. The intensity of the sound is also measured. HRTF is then used to calculate how the sound would appear in the ears and then it reaches the left respective the right one. Since every human being has a unique head shape, the sound perception is also unique. To solve this problem the API:s create an implementation to fit an average shaped head. The headshape is “created” from a number of persons head measures to make the sound as good as possible to as many people as possible. HRTF is used to calculate the direct path of the sound and does NOT handle reflected sound (reverb).

2.4 Reverberation Decay

Reverberation Decay is the effect that causes the amplitude to decrease in case of reflection.

> "Reverberation decay is the result of acoustic energy absorption by a room's surfaces and the propagation medium. Each time a sound wave bounces off a surface, it decreases in amplitude until there is negligible reflected sound. If room surfaces are acoustically "live," they absorb very little acoustic energy, reflected sounds diminish little each time they bounce, and the reverberation (the combination of all reflected sound) takes a long time to decay. If room surfaces are acoustically "dead," reverberation decays very quickly as acoustic energy is absorbed faster"[3].

3 Evaluation of the API:s

This chapter evaluates the API:s that were considered to be of interest for Saab Bofors Dynamics. The API:s chosen for this evaluation are marketleading and were considered to be the most interesting ones. Each of the sections in this chapter consists of a “History”, “General ideas” and a “Summary/Conclusions” part. The “History” section gives an idea of how the API has evolved through the years and how its current state is today. “General ideas” describes how the API works. “Summary/Conclusions” shows the advantages and disadvantages with the API and how well it lives up to Saab Bofors Dynamics´ Requirements. These requirements will be graded according to the three levels discussed in the Introduction - Requirements section. All of the API:s are evaluated against the requirements presented in section 1.3. Since it is hard to find objective information about the different API:s, the grading becomes rather subjective.

3.1 DirectSound

3.1.1 History

DirectSound3D is the part of DirectSound where one can position sound in three dimensions. When DirectSound3D was announced, the soundcard manufactures and the audio programmers were full of expectations. There would finally be a standard API to program 3D sound with hardware support towards. But a few months before the release of DirectX 3, including DirectSound3D, Microsoft announced that they would not allow any third party 3D-algoritms to accelerate DirectSound and Microsoft would provide an algorithm for this purpose. The sound card companies would not be able to create products based on their own 3D Audio technology which they, in most cases, spent years of work developing. Even though Microsoft’s intentions were good, the solution was far from optimal. Since DirectX did not support third party property sets, DirectX would need to have a 3D acceleration engine of their own in order to allow hardware acceleration. The fact that it did not have one led to a number of API:s who tried to work around this problem. Among these were Aureal 3D (A3D). A3D grew strong on the market despite the fact that DirectX added support for third party property sets in their next release, DirectX 5 (DirectX 4 was skipped).

DirectSound is a component of the DirectX Software Development Kit (DX-SDK). Since DirectX is a product developed by Microsoft and is needed to play sound, show graphics etc. in Windows, it is integrated in the Operating System (OS). Which version of DirectX that comes with Windows, depends on the version of Windows (see table 3.1 for details).

OS version	DirectX version
Windows NT 3.1	DirectX v.2.0
Windows NT 4	DirectX v.2.0
Windows 95	DirectX v.5.0
Windows 98	DirectX v.6.0
Windows Me	DirectX v.7.0
Windows 2000	DirectX v.7.0

Table 3-1: Windows and DirectX version shipped together

All the versions of the OS supports the newest version (DirectX 8.0) except for the NT – versions. They only support DirectX versions up to 3.0. The NT Operative System, for the most part, was designed for office applications and server functions and therefore not in need of DirectX-support. Due to combined demands, to have both stability (NT) and game support, Windows 2000 have been developed. Windows 2000 is built on NT-technology but has full support for the latest DirectX version.

3.1.2 General ideas

Here are the basic steps you will have to take in order to create sound using DirectSound. The wave files tend to be quite big since there's compression and playing them directly from disc wouldn't work very well. That's why there are sound buffers. As the name applies, the buffer holds the wave file data ready to play at any time. The sound buffer can be replayed and written to as it pleases you. When played, the buffer data is copied to the primary sound buffer (managed by the DirectSound interface itself). From the primary buffer the sound then finds the way to your speakers through your soundcard. DirectSound uses two different kinds of buffers: the primary and the secondary buffer. The secondary buffers are generally used to store one sound sample that will be mixed with other secondary buffers. If you want to mix a number of secondary buffers (i.e. sounds) you place these in the primary buffer.

The primary buffer is an object that mixes and plays the final audio output. This is what the primary buffer generally is used for (to mix sound from secondary buffers), but it can also be accessed directly for custom mixing or to perform other specialized activities.

All sound objects in DirectSound uses sound cones, but the angle of the cone is set to 360^{o} on those objects that has the same volume in all directions.

> "At any angle within the inner cone, the volume of the sound is just what it would be if there were no cone, after taking into account the basic volume of the buffer, the distance from the listener, the listener's orientation, and so on.
> At any angle outside the outer cone, the normal volume is attenuated by a factor set by the application."[4]

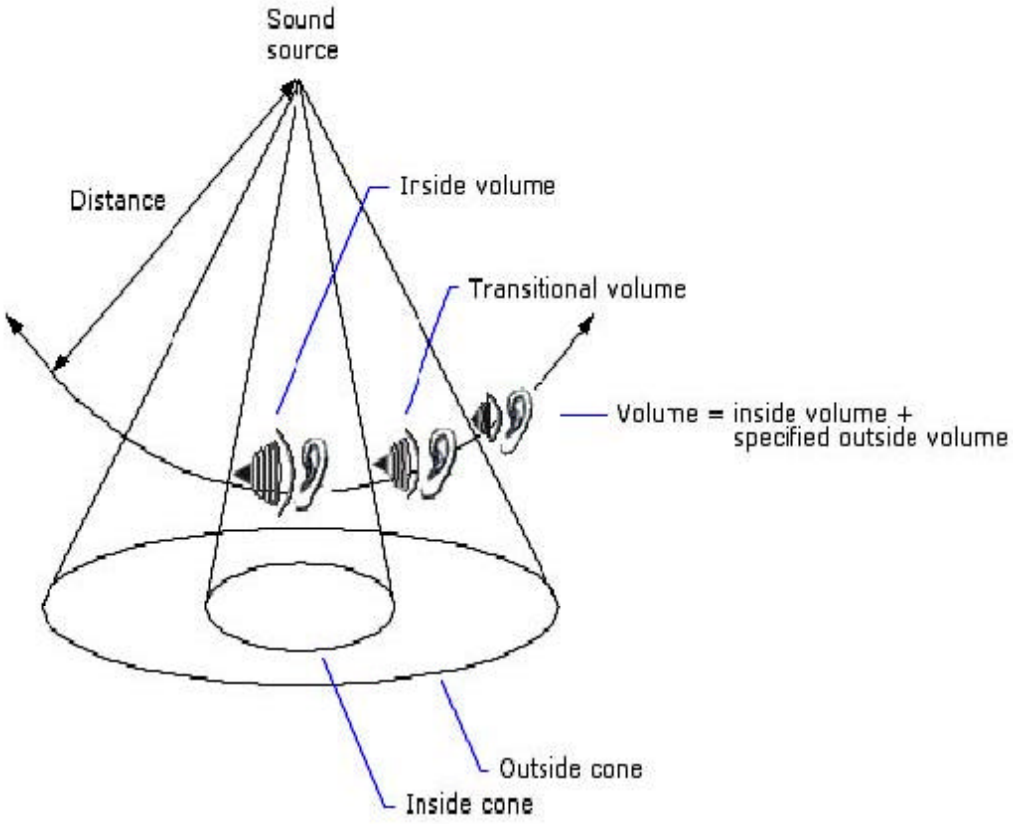

Figure 3-1: An illustration in how sound cones work [5]

DirectSound uses cones to give the sound a direction. One can use it to give a realistic sound propagation of a trumpet. Or you can use it to give a person an impression that the sound increases in volume as he/she approaches a door, by setting the inside cone at the same width as the door and the outer cone inaudible. See Figure 3-2 for an illustration of this effect.

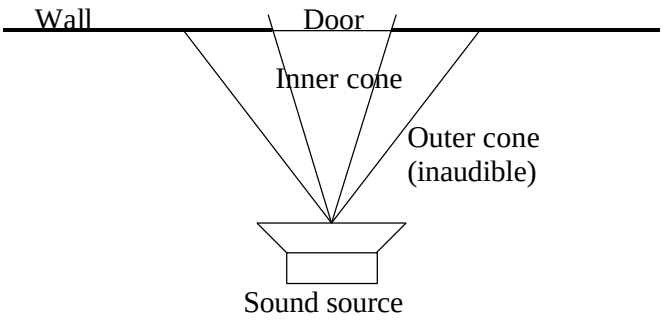

Figure 3-2:An example how to use sound cones

A sample code of an implementation using DirectSound is shown in the Appendix.

3.1.3 Summary/Conclusions

DirectSound is a complete API which has been on the market for a long time. Its founder, Microsoft, is the world leading OS manufacture. This makes the future look rather good for DirectSound. The fact that it is totally free and that there are tons of code examples on the Internet also gives it a benefit towards Figure 3-3. This is the API, which most likely is the one that will be used in Saab Bofors Dynamics simulations.

Level of interaction	Excellent
Level of abstraction	Adequate
Status / Stability	Excellent
License agreement / Cost	Excellent
Complexity	Excellent
Platform support	Poor

Figure 3-3: DirectSounds' mark on the requirements

3.2 EAX

3.2.1 History

The Environmental Audio eXtensions (EAX) became a part of DirectX in version 5.0 when Microsoft allowed Creative to add EAX as a third part property set to the DirectX API. This was a great milestone in EAX:s history since DirectX is included on every PC with the Windows platform (Table 3.1)

The 21st of September, in the year 2000, Creative bought Aureal.

3.2.2 General ideas

The EAX unified interface enables a PC without any hardware support (Creatives soundcard-circuit *emu10k1*) or the latest drivers installed to use the EAX. The EAX unified interface translates the EAX-calls to the most compatible interface on the existing PC-platform, preserving as much of the EA extension effects as possible. EAX includes two different property sets: the Listener property set and the Sound Source property set. The listener is applied to the primary buffer (also called the listener) and is used to describe the listeners surrounding environment.

The sound source property set is applied to every individual secondary buffer (the sound sources) and they describe how every source sounds in its own surrounding environment.
Here follows a simplyfied description in how the hardware support work:

> ”Once a listener is in a room with sound sources, the program does not need to do any more work—the EAX engine performs all the necessary calculations for moving sources and listener.”[6]

This allows the CPU to do other computations since the emu10k1 circuit processes all these EAX-specific calls.

EAX has an excellent reference manual that makes it easy to understand how EAX works and also how it interacts with DirectSound.

To make it easy for developers Creative have created 26 presets (Table 3-2). The presets are a number of pre-defined values. The presets each represent a specific environment.

Preset Base Environment	Volume	Decay	Time Damping
GENERIC	0.5F	1.493F	0.5F
PADDEDCELL	0.25F	0.1F	0.0F
ROOM	0.417F	0.4F	0.666F
BATHROOM	0.653F	1.499F	0.166F
LIVINGROOM	0.208F	0.478F	0.0F
STONEROOM	0.5F	2.309F	0.888F
AUDITORIUM	0.403F	4.279F	0.5F
CONCERTHALL	0.5F	3.961F	0.5F
CAVE	0.5F	2.886F	1.304F
ARENA	0.361F	7.284F	0.332F
HANGAR	0.5F	10.0F	0.3F
CARPETEDHALLWAY	0.153F	0.259F	2.0F
HALLWAY	0.361F	1.493F	0.0F
STONECORRIDOR	0.444F	2.697F	0.638F
ALLEY	0.25F	1.752F	0.776F
FOREST	0.111F	3.145F	0.472F
CITY	0.111F	2.767F	0.224F
MOUNTAINS	0.194F	7.841F	0.472F
QUARRY	1.0F	1.499F	0.5F
PLAIN	0.097F	2.767F	0.224F
PARKINGLOT	0.208F	1.652F	1.5F
SEWERPIPE	0.652F	2.886F	0.25F
UNDERWATER	1.0F	1.499F	0.0F
DRUGGED	0.875F	8.392F	1.388F
DIZZY	0.139F	17.234F	0.666F
PSYCHOTIC	0.486F	7.563F	0.806F

Table 3-2: Creative presets for different environments [7]

The developer can then tweak the reverb by changing the amount of reverb applied, the decay time and the damping. The last two basically allows the developers to simulate different material of walls/environments at a level of a whole room. At this time, the developer cannot change other settings like the size of the room, a sewer pipe etc.

A sample code of an implementation using Environmental Audio extensions (EAX) is shown in Appendix.

3.2.3 Summary/Conclusions

Since DirectSound has integrated support for EAX, there is no real reason to use it. If one want an EAX-specific effect, it is easy to make that call through DirectSound. Otherwise is this API is a great one with an excellent SDK reference manual which makes it easy to learn how EAX works and how to implement its effects.

Level of interaction	Excellent
Level of abstraction	Adequate
Status / Stability	Excellent
License agreement / Cost	Excellent
Complexity	Excellent
Platform support	Poor

Figure 3-4: EAXs' mark on the requirements

3.3 A3D (Aureal 3D)

3.3.1 History

Aureal 3D was developed in 1996. Aureal's API became very popular, since there were not any API present at that time who were able to hardware accelerate sound effects (see History –DirectX in chapter 3.1.1 for details).

3.3.2 General ideas

In A3D 1.0 there is only support for rendering rates up to 22kHz. A3D 2.0 supports up to 48 kHz, which approves the positioning accuracy as well as the overall quality. The biggest difference between A3D and the other sound API:s is that A3D uses wavetracing.

Wavetracing

The wavetracing algorithm in A3D 2.0 performs calculations on the sound and the surroundings in real-time. It calculates how the sound changes in wavelength and loudness as it reflects at walls, occludes through doors etc. The wavetracing technology is very complicated and Aureal spent several years to develop it.

3.3.3 Summary/Conclusions

Though DirectX today has integrated support for A3D, the future looks grim for A3D. Creative bought the whole company and there will probably not be any more versions of the API.

Level of interaction	Excellent
Level of abstraction	Adequate
Status / Stability	Poor
License agreement / Cost	Excellent
Complexity	Adequate
Platform support	Adequate

Figure 3-5: A3Ds' mark on the requirements

3.4 Miles Sound System

3.4.1 History

Miles' first version of its API was released as early as 1991. In the computer industry, this is a very long time and this has helped make them an established and well-respected name in the sound programming industry.

3.4.2 General ideas

Miles uses DirectSounds' driver layer to gain low-level access to the sound card hardware. Miles are very pleased with the driver layer of DirectSound, but they think that the other component that DirectSound provides the DirectSound API itself "is just plain horrible"[8]. This is based on the fact that even to accomplish simplest application, you have to write an considerable amount of code. The DirectSound 8, which Miles not supports, however is much more user friendly than the previous version.

The Miles API supports a large number of sound formats, which gives it a wide range of applications. It has a high level of abstraction (higher than DirectSound, for instance). Simplified, this means that you will not have to write as much code as lower-level API:s to achieve the same result. But a higher level of abstraction also means that you are not able to control exactly what happens with the sound (unless you learn what the automatic function calls does). Miles supports DirectSound3D software, DirectSound3D hardware, DirectSound 7 software, Creative's EAX 1 and 2, Aureal's A3D 1 and 2, fast Miles 2D, Dolby Surround,

QSound and their own RSX. And with the Miles API, you can even switch between any of these technologies at run-time. This feature allows you to take advantage of a specific API:s advantages, e.g. first you use the DS3D API for positioning, but when you switch to EAX to get the effect of being in a narrow hallway.

3.4.3 Summary/Conclusions

Miles is an excellent API with a high level of abstraction and integrated support for the most common API:s. However Miles has not added support for DirectSound 8, which was released in November 9 the year 2000. The other big disadvantage is that the license fee so high (from 4000$ and up) that the advantages of the API dose not compensate it. This is the main reason why the Miles API is not of any interest for Saab Bofors Dynamics.

Level of interaction	Excellent
Level of abstraction	Excellent
Status / Stability	Excellent
License agreement / Cost	Poor
Complexity	Excellent
Platform support	Adequate

Figure 3-6: Miles' mark on the requirements

3.5 OpenAL

3.5.1 History

OpenAL is a multi-platform API supporting Windows, the Macintosh OS, Linux, FreeBSD, OS/2 and BeOS. OpenAL is open source and can be used, free of charge, without any form of license agreement from the developers. But if you later want to sell a product with OpenAL (or any open source product) you will have to pay a small fee.

The fact that OpenAL is an open source project means that anyone is free to get involved in the project and contribute in any way possible.

There are currently two companies involved in the project: Loki and Creative.

The API is hardware independent, that is: it takes advantage of any soundcard-hardware available though you of course get a more realized on multi channel audio output cards

3.5.2 General ideas

Since it is platform independent, once you have implemented a functional version of your game or program, all you have to do is port your application to the desired OS. This instead of using an other API that the new OS supports which would demand the same amount of effort that it took to create the first application. In the equivalence in the graphics world, OpenGL, you have to add the operative system specific bindings (for instance: all the window handling) for the new OS, because OpenGL dose not handle those itself (due to the platform independendency). OpenAL include both the API core and the operative system specific bindings so all you have to do to get the sound working on another platform is the porting. An open source project has a tendency to get delayed since there often are a lot of people/companies jumping on the "circus" and after a while leaving with the code they accomplished during their stay. This delay the project cause the parts provided need to be re-programmed.

A sample code of an implementation using Environmental Audio extensions (EAX) is shown in the Appendix.

3.5.3 Summary/Conclusions

Since the importance of platform independency is not significant (see Platform support in chapter 1.3.6) OpenAL looses its greatest benefit. OpenAL is great if one want to integrate it with OpenGL, but otherwise it has no advantages compared to the other API:s.

Level of interaction	Excellent
Level of abstraction	Adequate
Status / Stability	Adequate
License agreement / Cost	Adequate
Complexity	Excellent
Platform support	Excellent

Figure 3-7: OpenALs' mark on the requirements

3.6 Comparison of the API:s

Since there is very difficult to find objective information of the API:s, the decision which API to use becomes quite hard. The decision is based on feeling, rather than on facts. The API chosen for further implementation for Saab Bofors Dynamics is Direct Sound 3D, which were considered to be the best, based on its excellent level of interaction and Microsoft's

strong position on the market. Below is a comparison table of the API:s. All the criterions have been given a different weight according to their level of importance. The grades (Excellent, Adequate and Poor) are translated into numeric grades in order to achieve a numeric total score. As shown in the tables below, both EAX and Direct Sound 3D gathers 82 points. EAX is integrated in Direct Sound (simplified: Direct Sound is both Direct Sound and EAX) and therefore Direct Sound wins the battle. Miles is not far behind, but the their high level of abstraction is not all positive (see section 1.3.2). This makes the total score of Miles somewhat misleading.

Open AL

Weight	Criterion	Grade	Grade value	Sum
10	Level of interaction	Excellent	3	30
8	Level of abstraction	Adequate	2	16
6	Status / Stability	Adequate	2	12
4	License agreement / Cost	Adequate	2	8
2	Complexity	Excellent	3	6
1	Platform support	Excellent	3	3
			Total Score	**75**

Miles

Weight	Criterion	Grade	Grade value	Sum
10	Level of interaction	Excellent	3	30
8	Level of abstraction	Excellent	3	24
6	Status / Stability	Excellent	3	18
4	License agreement / Cost	Poor	0	0
2	Complexity	Excellent	3	6
1	Platform support	Adequate	2	2
			Total Score	**80**

A3D

Weight	Criterion	Grade	Grade value	Sum
10	Level of interaction	Excellent	3	30
8	Level of abstraction	Adequate	2	16
6	Status / Stability	Poor	0	0
4	License agreement / Cost	Excellent	3	12
2	Complexity	Adequate	2	4
1	Platform support	Adequate	2	2
			Total Score	**64**

EAX

Weight	Criterion	Grade	Grade value	Sum
10	Level of interaction	Excellent	3	30
8	Level of abstraction	Adequate	2	16
6	Status / Stability	Excellent	3	18
4	License agreement / Cost	Excellent	3	12
2	Complexity	Excellent	3	6
1	Platform support	Poor	0	0
			Total Score	**82**

Direct Sound 3D

Weight	Criterion	Grade	Grade value	Sum
10	Level of interaction	Excellent	3	30
8	Level of abstraction	Adequate	2	16
6	Status / Stability	Excellent	3	18
4	License agreement / Cost	Excellent	3	12
2	Complexity	Excellent	3	6
1	Platform support	Poor	0	0
			Total Score	**82**

Table 3-3: Comparison of the API:s

4 Possible integration of the API into the existing code

This chapter gives an idea of how the visualizations and simulations are structured. It also shows how the evaluated sound APIs can be integrated in that structure.

The object hierarchy currently in use at Saab Bofors Dynamics looks approximately like Figure 4-1.

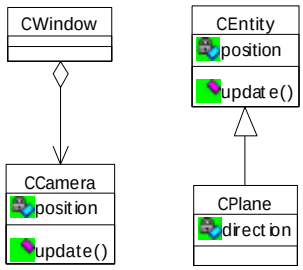

Figure 4-1: Sample simulation class hierarchy

Each window that is used in a simulation has one or more cameras (CCamera), which can be thought of as the eye through which the user sees the world. CEntity is the super class from which all actors (objects) in a simulation inherits, such as for example planes (CPlane). When trying to figure out where the implementation of sound fits in all this, it is quite easy to see that the class CCamera is a suitable candidate to also be a listener. The necessary attributes of a listener are a position and a direction in which the listener travels. The CCamera class already provides the position, but needs to be extended with a direction in order for sounds to work properly. A possible solution to this is illustrated in Figure 4-2

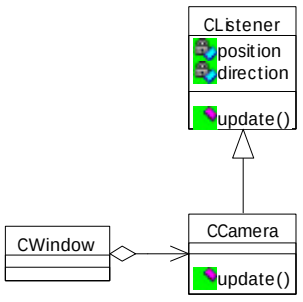

Figure 4-2: Integrating the listener

As can be seen in the figure, the position attribute is moved upwards in the hierarchy since it is not necessary to have two different variables storing the same attribute. Now the CCamera class acts as a listener and provides the user with both visual and audio parts of the visualization.

Considering the sounds sources, the first thing one might think is that each entity should be associated with a sound. This, however, is not completely correct since it is not necessary for all entities to produce sounds. Consider the example of a class CAirport, which inherits from CEntity. Certainly objects that are on the airport (such as planes) will produce sounds, but the airport itself will be silent. Thus it is better to let each suitable subclass of CEntity have their own sound, as illustrated in Figure 4-3.

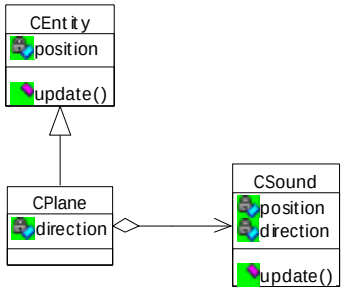

Figure 4-3: Integrating a sound source

This certainly solves the problem, but it is not the optimal solution. One very important aspect when programming real-time simulations is efficiency. Each object in the 3D world needs to be updated at least 30 times a second for the simulation to be smooth, hence we need to optimise the code. In this example we need to call the update() method on the CSound class each time update() is called on the CPlane, i.e. we get double method calls. This might seem like a very small overhead, but consider the case when we have 1000 entities with sounds in a simulation (which not is an unlikely situation). Then instead of 1000 update() calls each program loop we have 2000, which might affect the smoothness of the simulation. The same argument can be applied on the fact that both CPlane and CSound consists of the same attributes; position and direction. In a normal simulation each of these would be a vector consisting of three floats (x, y and z). This means that we have a small overhead of 24 bytes (if a float has a size of 4 bytes, which is normal for the Win32 platform) for each sound. This

certainly is a very small overhead, but in some cases it might make a difference. The scheme shown in Figure 4.4 provides a better solution.

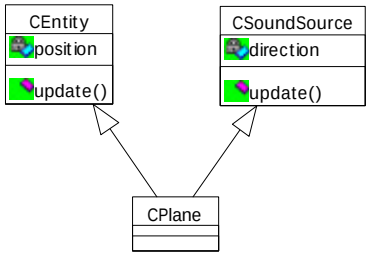

Figure 4-4: Better way of integrating a sound source

In this hierarchy we have eliminated the unnecessary method call, since both the sound and the entity parts of the plane is updated in the same call. There also are no unnecessary variables since everything is encapsulated in one single class.

5 Summary

In order to objectively decide which API is the best alternative for Saab Bofors Dynamics, one would have to work with all of the different sound API:s for an equally long time. To work with an API is the only way to see all its advantages and disadvantages. A typical developer has been working with only one API and learned how that works. Since this API is the only one known this API is used for all of the implementations. Even if the developer has experience of several API:s, he or she has most likely been working with one API more than the others. If this is the case, there is no objective way to choose which API is the best one.

It is almost exclusively the creators of the API:s themselves who describe what their own API can do. If an effect is supported in a competitors' API, but not in their own, they may not admit this. In case the API-developer uses this to advertise their own API, its competitors would usually respond that the effect is useless and that there is no need to implement support for it. Considering this, it may not be possible to obtain valuable information by reading the API-creators specification. Since there are not any totally objective persons reviewing the API:s there is only one way to get a good apprehension over which API is the best, i.e. what effects they can do, how (effective) they solve these effects, the amount of CPU-power it takes to process the effects and so on. The only solution left is to work with the all the separate API:s for the given amount of time it takes to know how the API works, including all its advantages/disadvantages compared to each other.

The second best thing may be to evaluate the API:s theoretically and then weigh the advantages and disadvantages of the different API:s towards each other. This has been done in this thesis and the API that lived up to the requirements best was **DirectSound**. See the summary of the evaluated API:s in chapter 3.6 for more details.

References

[1] Sven Roepke, *The Doppler effect*, http://www.mohawk.net/~viking/physics/doppler.html, 2001-02-13

[2] Cherie Bibo Lehman, *Doppler effect*, http://www.ccm.ecn.purdue.edu/~html/cblehman/, 2001-02-13

[3] Microsoft, DirectX Documentation (Visual C++) – Sound Cones, http://www.microsoft.com/directX

[4] Microsoft, DirectX Documentation (Visual C++) – Sound Cones, http://www.microsoft.com/directX

[5] Microsoft, DirectX Documentation (Visual C++) – Sound Cones, http://www.microsoft.com/directX

[6] Creative, *The EAX 2.0 reference manual*, Creative, 2001-02-27

[7] Mikael Hagén, *A Gamer's Guide to EAX*, http://www.3dsoundsurge.com/features/articles/EAX.html, 2001-03-05

[8] Rad Gametools, *The Miles Sound System!*, http://www.radgametools.com/miles.htm, 2001-03-01

Appendix Sample code

DirectSound

All the methods and variables presented in this example that are not commented in Swedish are specific to graphics. These are not important in the meaning of understanding in creating sound using DirectSound3D and can be ignored.

```
/************************************tgvApp.h*****************************/

ifndef __TG_V_APP_H__
#define __TG_V_APP_H__

#include "tgApplication.h"

#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>
#include "dsutil.h"


#define TG_POS_X    0
#define TG_POS_Y    0
#define TG_WIDTH    1024
#define TG_HEIGHT   768


class tgVApp : public tgApplication
{
public:
protected:
                    HDC                                     tn_hDC;
                    HGLRC                           tn_hRC;
                    DEVMODE                         tn_screenSettings;

                    BOOL                            keys[256];

                    CSoundManager               *manager;
                    DS3DBUFFER                      buffy;
                    D3DVECTOR                       pos;

private:
```

```
   public:
                                                        tgVApp(HINSTANCE, WNDPROC);
        virtual                                    ~tgVApp();

   #pragma auto_inline(on)
                       int                              init();
                       int                              execute();
                       int                              release();

                       int                              onCreate(HWND, LPCREATESTRUCT);
                       int                              onClose(HWND);
                       int                              onDestroy(HWND);
                       int                              onKeyDown(HWND hWnd, int nVirtKey,
LPARAM lKeyData);
                       int                              onKeyUp(HWND  hWnd,  int  nVirtKey,
LPARAM lKeyData);
                       int                              onLButtonDown(WPARAM  fwKeys,  WORD
xPos, WORD yPos);
                       int                              onMouseMove(WPARAM   fwKeys,   WORD
xPos, WORD yPos);
                       int                              onRButtonDown(HWND   hWnd,   WPARAM
fwKeys, WORD xPos, WORD yPos);

   #pragma auto_inline(off)
   protected:
   public:
   private:
   };


   #endif

   /************************************tgvApp.c*****************************/
   #include "stdafx.h"
   #include "tgVApp.h"
   #include <mmsystem.h>
   #include "CPosSound.h"



   tgVApp::tgVApp(HINSTANCE hInstance, WNDPROC WndProc) : tgApplication("SAAB Bofors Dynamics:
Lyyyyd!!.", hInstance, WndProc)
   {
        memset(&tn_screenSettings, 0, sizeof(DEVMODE));
        // Clear Room To Store Settings
        tn_screenSettings.dmSize             = sizeof(DEVMODE);
        // Size Of The Devmode Structure
        tn_screenSettings.dmPelsWidth = TG_WIDTH;
        // Screen Width
        tn_screenSettings.dmPelsHeight       = TG_HEIGHT;
        // Screen Height
```

```
        tn_screenSettings.dmFields           = DM_PELSWIDTH | DM_PELSHEIGHT;             //
Pixel Mode

        for (int i=0; i<256; i++) keys[i] = FALSE;
   }

   tgVApp::~tgVApp()
   {
   }

   int tgVApp::init()
   {

        if (tgApplication::init())
        {
               ////////////////////////////////TODO: CREATE STUFF
               //tn_hWnd

               manager = new CSoundManager();
               manager->Initialize(tn_hWnd, DSSCL_PRIORITY, 2, 22050, 16);// 44100


   /*
               HRESULT QueryInterface
               (
                 REFIID riid,
                 LPVOID* ppvObj
               );

               IID id;
               LPVOID* ptr;
               IDirectSound3DBuffer8::QueryInterface(id, ptr); //resultat


               DS3DBUFFER buffer;
               //GetAllParameters(&buffer);
   */

               // Ytterligare initiering om nödvändigt!


               // Ytterligare init..... blah blah blah

               ////////////////////////////////TODO: END
               return TRUE;
        }

        return FALSE;
   }

   int tgVApp::execute()
```

```
   {
        MSG msg;
   /************************************/
        int NUMBER = 0;
        CPosSound *sound;
        CPosSound *sound2;
        //Skapar en 3dListener
        LPDIRECTSOUND3DLISTENER listener;
        //LPDIRECTSOUND ljud;
        manager->Get3DListenerInterface(&listener);

        listener->SetRolloffFactor(1, DS3D_IMMEDIATE );//DEFERRRED
        listener->SetDopplerFactor(1, DS3D_IMMEDIATE );

        //skapar ett sound-object med managern
        manager->Create((CSound**)&sound,                  "E:\\Engdahl\\Lyd\\lugna\\ding.wav",
DSBCAPS_CTRL3D, DS3DALG_NO_VIRTUALIZATION);
        manager->Create((CSound**)&sound2,    "E:\\Engdahl\\Lyd\\lugna\\Utopia    -    Fel.wav",
DSBCAPS_CTRL3D, DS3DALG_NO_VIRTUALIZATION);

        while(1)
        {
                while (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))   // Process All Messages
                {
                        if (GetMessage(&msg, NULL, 0, 0))
                        {
                                TranslateMessage(&msg);
                                DispatchMessage(&msg);
                        }
                        else
                        {
                                return msg.wParam;
                        }
                }

                ////////////////////////////////TODO: UPDATE STUFF

                if(NUMBER<=20)
                {
                        sound->Play(0, 0);    //<----0 == sätter inga flaggor (ingen loop etc)



                        //Flyttar lyssnaren bort från ljudkällan
                        //listener->SetPosition((NUMBER   *   0.2f),   NUMBER*0.f,   NUMBER*0.f,
DS3D_DEFERRED );
                        //Sätter hastigheten på lyssnaren
                        //listener->SetVelocity(NUMBER*10.f,       NUMBER*0.f,       NUMBER*0.f,
DS3D_DEFERRED );
```

```
                    sound2->setPosition(0.4f  *  NUMBER,  0.f,  0.f);  //anropa  metod  för
positionen på "ljudobjektet"
                    sound->setPosition(-0.4f * NUMBER, 0.f, 0.f);
                    sound2->setVelocity(8, 0.f, 0.f);
                    Sleep(500);
                    sound2->Play(0,0);
                    ++NUMBER;
             }


             ////////////////////////////////TODO: END
        }// END while(1)-loop
   }

   int tgVApp::release()
   {
        ////////////////////////////////TODO: DELETE STUFF

        ////////////////////////////////TODO: END DELETE STUFF

        return tgApplication::release();
   }

   inline int tgVApp::onCreate(HWND hWnd, LPCREATESTRUCT lpcs)
   {
        static PIXELFORMATDESCRIPTOR pfd=
                                                {
                                                        sizeof(PIXELFORMATDESCRIPTOR),
             // Size Of This Pixel Format Descriptor
                                                        1,
                                 // Version Number (?)
                                                        PFD_DRAW_TO_WINDOW |
             // Format Must Support Window
                                                        PFD_SUPPORT_OPENGL |
             // Format Must Support OpenGL
                                                        PFD_DOUBLEBUFFER,
                    // Must Support Double Buffering
                                                        PFD_TYPE_RGBA,
                    // Request An RGBA Format
                                                        16,
                                 // Select A 16Bit Color Depth
                                                        0, 0, 0, 0, 0, 0,
                    // Color Bits Ignored (?)
                                                        0,
                                 // No Alpha Buffer
                                                        0,
                                 // Shift Bit Ignored (?)
                                                        0,
                                 // No Accumulation Buffer
                                                        0, 0, 0, 0,
                            // Accumulation Bits Ignored (?)
```

```
                                        16,
                                // 16Bit Z-Buffer (Depth Buffer)
                                        0,
                                // No Stencil Buffer
                                        0,
                                // No Auxiliary Buffer (?)
                                        PFD_MAIN_PLANE,
                        // Main Drawing Layer
                                        0,
                                // Reserved (?)
                                        0, 0, 0
                        // Layer Masks Ignored (?)
                                    };

        GLuint pixelFormat;
        tn_hDC = GetDC(hWnd);
        pixelFormat = ChoosePixelFormat(tn_hDC, &pfd);

        if (!pixelFormat)
        {
                MessageBox(0, "Failed to find a suitable pixelformat", "Init error", MB_OK |
MB_ICONERROR);
                PostQuitMessage(0);
                return 0;
        }

        if (!SetPixelFormat(tn_hDC, pixelFormat, &pfd))
        {
                MessageBox(0, "Failed to set pixelformat", "Init error", MB_OK | MB_ICONERROR);
                PostQuitMessage(0);
                return 0;
        }

        tn_hRC = wglCreateContext(tn_hDC);
        if(!tn_hRC)
        {
                MessageBox(0, "Failed to create an OpenGL rendering context", "Init error",
MB_OK | MB_ICONERROR);
                PostQuitMessage(0);
                return 0;
        }

        if (!wglMakeCurrent(tn_hDC, tn_hRC))
        {
                MessageBox(0, "Failed to activate the OpenGL rendering context", "Init error",
MB_OK | MB_ICONERROR);
                PostQuitMessage(0);
                return 0;
        }

        return 0;
```

```
}

int tgVApp::onClose(HWND)
{
     ChangeDisplaySettings(NULL, 0);
     wglMakeCurrent(tn_hDC, NULL);
     wglDeleteContext(tn_hRC);
     ReleaseDC(tn_hWnd, tn_hDC);
     PostQuitMessage(0);

     return 0;
}

inline int tgVApp::onDestroy(HWND hWnd)
{
     ChangeDisplaySettings(NULL, 0);
     wglMakeCurrent(tn_hDC, NULL);
     wglDeleteContext(tn_hRC);
     ReleaseDC(tn_hWnd, tn_hDC);
     PostQuitMessage(0);

     return 0;
}

inline int tgVApp::onKeyDown(HWND hWnd, int nVirtKey, LPARAM lKeyData)
{
     static bool fullScreen = true;
     if (27 == nVirtKey)
     {
            onDestroy(hWnd);
            PostQuitMessage(0);
     }
     else if ('F' == nVirtKey)
     {
     }
     else if ('N' == nVirtKey)
     {
     }
     else if ('M' == nVirtKey)
     {
     }
     else if ('O' == nVirtKey)
     {
            if (fullScreen)
                   ChangeDisplaySettings(&tn_screenSettings, CDS_FULLSCREEN);
            else
                   ChangeDisplaySettings(NULL, 0);

            fullScreen = !fullScreen;
     }
     else if (' ' == nVirtKey)
```

```
    {
    }
    else if (VK_SUBTRACT == nVirtKey)
    {
    }
    else if ('1' == nVirtKey)
    {

    }
    else if ('2' == nVirtKey)
    {

    }
    else if ('3' == nVirtKey)
    {
    }

    keys[nVirtKey] = TRUE;
    return 0;
}

inline int tgVApp::onKeyUp(HWND hWnd, int nVirtKey, LPARAM lKeyData)
{
    keys[nVirtKey] = FALSE;
    return 0;
}

inline int tgVApp::onMouseMove(WPARAM fwKeys, WORD xPos, WORD yPos)
{
    if (fwKeys & MK_LBUTTON)
    {
           //TODO: IF USER INPUT BY MOUSE, DO SOMETHING

           //xpos = xPos;
           //ypos = yPos;
    }
    return 0;
}

inline int tgVApp::onLButtonDown(WPARAM fwKeys, WORD xPos, WORD yPos)
{
    //TODO: IF USER INPUT BY MOUSE, DO SOMETHING

    //     xpos = xPos;
    //     ypos = yPos;
    return 0;
}

inline int tgVApp::onRButtonDown(HWND hWnd, WPARAM fwKeys, WORD xPos, WORD yPos)
{
    //TODO: IF USER INPUT BY MOUSE, DO SOMETHING
```

```
        //      xpos = xPos;
        //      ypos = yPos;
        return 0;
    }
```

Environmental Audio extensions (EAX)

An example in how EAX works.

```
    /************************    EAXPanel.h     ***********************/
    #ifndef __EAXPANELH
    #define __EAXPANELH

    #define STRICT
    #include <stdio.h>
    #include <windows.h>
    #include <commctrl.h>
    #include "eax.h"
    #include "resource.h"

    class Audio
    {
    private:
     LPDIRECTSOUND           m_lpds;
     LPDIRECTSOUNDBUFFER     m_lpPrimary;
     LPDIRECTSOUND3DLISTENER m_lpListener;
     LPDIRECTSOUNDBUFFER     m_lpdsb;
     LPDIRECTSOUND3DBUFFER   m_lpds3Db;
     LPKSPROPERTYSET         m_lpksps;
     DWORD                   m_dwSupport;

    public:
     Audio();
     ~Audio();

     BOOL Init(HWND hwnd, GUID* pGuid, BOOL bUsing3D = FALSE);
     BOOL CreatePrimary(BOOL bUsing3D);
     BOOL SetFormat(WAVEFORMATEX* pWfex);
     BOOL QuerySupport(ULONG ulQuery);
     BOOL CreatePropertySet(void);
     BOOL CreateEAX(void);
     BOOL  CreateBufferFromFile(LPSTR  szFile,  LPDIRECTSOUNDBUFFER*  lplpdsb,  BOOL  bUsing3D  =
FALSE, LPDIRECTSOUND3DBUFFER* lplpds3Db = NULL, LPKSPROPERTYSET* lplpksps = NULL);
     void ReleaseAll(void);

     BOOL SetAll(LPEAXLISTENERPROPERTIES lpData);
     BOOL SetRoom(LONG lValue);
     BOOL SetRoomHF(LONG lValue);
```

```
 BOOL SetRoomRolloff(float fValue);
 BOOL SetDecayTime(float fValue);
 BOOL SetDecayHFRatio(float fValue);
 BOOL SetReflections(LONG lValue);
 BOOL SetReflectionsDelay(float fValue);
 BOOL SetReverb(LONG lValue);
 BOOL SetReverbDelay(float fValue);
 BOOL SetEnvironment(DWORD dwValue);
 BOOL SetEnvironmentSize(float fValue);
 BOOL SetEnvironmentDiffusion(float fValue);
 BOOL SetAirAbsorption(float fValue);
 BOOL SetScaleDecayTime(BOOL bValue);
 BOOL SetClipDecayHF(BOOL bValue);
 BOOL SetScaleReflections(BOOL bValue);
 BOOL SetScaleReflectionsDelay(BOOL bValue);
 BOOL SetScaleReverb(BOOL bValue);
 BOOL SetScaleReverbDelay(BOOL bValue);
 BOOL SetFlags(DWORD dwValue);

 BOOL GetAll(LPEAXLISTENERPROPERTIES lpData);
 BOOL GetDecayTime(float* pfValue);
 BOOL GetReflections(long* plValue);
 BOOL GetReflectionsDelay(float* pfValue);
 BOOL GetReverb(long* plValue);
 BOOL GetReverbDelay(float* pfValue);

 BOOL SetListenerRolloff(float fValue);
};

class Sound
{
private:
 LPDIRECTSOUNDBUFFER     m_lpdsb;
 LPDIRECTSOUND3DBUFFER   m_lpds3Db;
 LPKSPROPERTYSET         m_lpksps;
 BOOL                    m_bPlaying;

public:
 Sound();
 ~Sound();

 void ReleaseAll(void);
 BOOL TogglePlay(void);
 BOOL PlayLooped(void);
 BOOL Stop(void);
 BOOL IsPlaying(void) {return m_bPlaying;}
 LPDIRECTSOUNDBUFFER*   GetBufferAddress(void) {return &m_lpdsb;}
 LPDIRECTSOUND3DBUFFER* Get3DBufferAddress(void) {return &m_lpds3Db;}
 LPKSPROPERTYSET*       GetPropertySetAddress(void) {return &m_lpksps;}

 BOOL SetSourceAll(LPEAXBUFFERPROPERTIES lpData);
```

```
    BOOL SetSourceDirect(LONG lValue);
    BOOL SetSourceDirectHF(LONG lValue);
    BOOL SetSourceRoom(LONG lValue);
    BOOL SetSourceRoomHF(LONG lValue);
    BOOL SetSourceRolloff(float fValue);
    BOOL SetSourceOutside(LONG lValue);
    BOOL SetSourceAbsorption(float fValue);
    BOOL SetSourceObstruction(LONG lValue);
    BOOL SetSourceObstructionLF(float fValue);
    BOOL SetSourceOcclusion(LONG lValue);
    BOOL SetSourceOcclusionLF(float fValue);
    BOOL SetSourceOcclusionRoom(float fValue);
    BOOL SetSourceAffectDirectHF(BOOL bValue);
    BOOL SetSourceAffectRoom(BOOL bValue);
    BOOL SetSourceAffectRoomHF(BOOL bValue);
    BOOL SetSourceFlags(DWORD dwValue);

    BOOL SetPositionX(float fValue);
    BOOL SetPositionY(float fValue);
    BOOL SetPositionZ(float fValue);
    BOOL SetConeOrientationX(float fValue);
    BOOL SetConeOrientationY(float fValue);
    BOOL SetConeOrientationZ(float fValue);
    BOOL SetConeInsideAngle(DWORD dwValue);
    BOOL SetConeOutsideAngle(DWORD dwValue);
    BOOL SetConeOutsideVolume(DWORD dwValue);
    BOOL SetMinDistance(float fValue);
    BOOL SetMaxDistance(float fValue);
   };


   // Functions found in Rendering.lib (or the debug version DeRendering.lib).
   BOOL InitRendering(HWND hwnd, int iMaxWidth, int iMaxHeight);
   void ReleaseRendering(void);
   void RenderReflectionsGraph(HWND hwnd, int iWidth, int iHeight, LPEAXLISTENERPROPERTIES
lpeaxlp);
   void RenderReverbGraph(HWND hwnd, int iWidth, int iHeight, LPEAXLISTENERPROPERTIES
lpeaxlp);

   #endif

   /************************    Sound.cpp   ***********************/
   #include "EAXPanel.h"


   /*
   */
   Sound::Sound()
   {
    m_lpdsb = NULL;
    m_lpds3Db = NULL;
```

```
 m_lpksps = NULL;
 m_bPlaying = FALSE;
}




/*
*/
Sound::~Sound()
{
 ReleaseAll();
}




/*

  Routine to release any aquired interfaces for this object.

*/
void Sound::ReleaseAll(void)
{
 if ( m_lpksps != NULL )
 {
  m_lpksps->Release();
  m_lpksps = NULL;
 }

 if ( m_lpds3Db != NULL )
 {
  m_lpds3Db->Release();
  m_lpds3Db = NULL;
 }

 if ( m_lpdsb != NULL )
 {
  m_lpdsb->Release();
  m_lpdsb = NULL;
 }
}




/*

  Routine to play the current buffer in loop mode.

*/
```

```
BOOL Sound::PlayLooped(void)
{
 if ( m_lpdsb != NULL  && !m_bPlaying )
 {
  if ( m_lpdsb->Play(0, 0, DSBPLAY_LOOPING) == DS_OK )
   m_bPlaying = TRUE;
 }

 return m_bPlaying;
}




/*

  Routine to stop the playing buffer.

*/
BOOL Sound::Stop(void)
{
 if ( m_lpdsb != NULL && m_bPlaying )
 {
  if ( m_lpdsb->Stop() == DS_OK )
  {
   m_lpdsb->SetCurrentPosition(0);
   m_bPlaying = FALSE;
  }
 }

 return m_bPlaying;
}




/*

  Routine used to toggle the current playing state of the buffer.
*/
BOOL Sound::TogglePlay(void)
{
 if ( IsPlaying() )
 {
  Stop();
  return m_bPlaying;
 }
 else
 {
  PlayLooped();
  return m_bPlaying;
```

```
    }
   }



   /*

     All of the following routines are simple wrappers to either setting or getting EAX buffer
     properties.

   */


   BOOL Sound::SetSourceAll(LPEAXBUFFERPROPERTIES lpData)
   {
    return                          SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_ALLPARAMETERS, NULL, 0, lpData, sizeof(EAXBUFFERPROPERTIES)));
   }


   BOOL Sound::SetSourceDirect(LONG lValue)
   {
    return                          SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_DIRECT, NULL, 0, &lValue, sizeof(LONG)));
   }


   BOOL Sound::SetSourceDirectHF(LONG lValue)
   {
    return                          SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_DIRECTHF, NULL, 0, &lValue, sizeof(LONG)));
   }


   BOOL Sound::SetSourceRoom(LONG lValue)
   {
    return                          SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_ROOM, NULL, 0, &lValue, sizeof(LONG)));
   }


   BOOL Sound::SetSourceRoomHF(LONG lValue)
   {
    return                          SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_ROOMHF, NULL, 0, &lValue, sizeof(LONG)));
   }


   BOOL Sound::SetSourceRolloff(float fValue)
   {
```

```
    return                              SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_ROOMROLLOFFFACTOR, NULL, 0, &fValue, sizeof(float)));
   }


   BOOL Sound::SetSourceOutside(LONG lValue)
   {
    return                              SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_OUTSIDEVOLUMEHF, NULL, 0, &lValue, sizeof(LONG)));
   }


   BOOL Sound::SetSourceAbsorption(float fValue)
   {
    return                              SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_AIRABSORPTIONFACTOR, NULL, 0, &fValue, sizeof(float)));
   }


   BOOL Sound::SetSourceFlags(DWORD dwValue)
   {
    return                              SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_FLAGS, NULL, 0, &dwValue, sizeof(DWORD)));
   }


   BOOL Sound::SetSourceObstruction(LONG lValue)
   {
    return                              SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_OBSTRUCTION, NULL, 0, &lValue, sizeof(LONG)));
   }


   BOOL Sound::SetSourceObstructionLF(float fValue)
   {
    return                              SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_OBSTRUCTIONLFRATIO, NULL, 0, &fValue, sizeof(float)));
   }


   BOOL Sound::SetSourceOcclusion(LONG lValue)
   {
    return                              SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_OCCLUSION, NULL, 0, &lValue, sizeof(LONG)));
   }


   BOOL Sound::SetSourceOcclusionLF(float fValue)
   {
    return                              SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_OCCLUSIONLFRATIO, NULL, 0, &fValue, sizeof(float)));
```

```
}

BOOL Sound::SetSourceOcclusionRoom(float fValue)
{
 return                          SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_OCCLUSIONROOMRATIO, NULL, 0, &fValue, sizeof(float)));
}

BOOL Sound::SetPositionX(float fValue)
{
 D3DVECTOR vec;
 if ( m_lpds3Db->GetPosition(&vec) != DS_OK ) return FALSE;
 return SUCCEEDED(m_lpds3Db->SetPosition(fValue, vec.y, vec.z, DS3D_IMMEDIATE));
}

BOOL Sound::SetPositionY(float fValue)
{
 D3DVECTOR vec;
 if ( m_lpds3Db->GetPosition(&vec) != DS_OK ) return FALSE;
 return SUCCEEDED(m_lpds3Db->SetPosition(vec.x, fValue, vec.z, DS3D_IMMEDIATE));
}

BOOL Sound::SetPositionZ(float fValue)
{
 D3DVECTOR vec;
 if ( m_lpds3Db->GetPosition(&vec) != DS_OK ) return FALSE;
 return SUCCEEDED(m_lpds3Db->SetPosition(vec.x, vec.y, fValue, DS3D_IMMEDIATE));
}

BOOL Sound::SetConeOrientationX(float fValue)
{
 D3DVECTOR vec;
 if ( m_lpds3Db->GetConeOrientation(&vec) != DS_OK ) return FALSE;
 return SUCCEEDED(m_lpds3Db->SetConeOrientation(fValue, vec.y, vec.z, DS3D_IMMEDIATE));
}

BOOL Sound::SetConeOrientationY(float fValue)
{
 D3DVECTOR vec;
 if ( m_lpds3Db->GetConeOrientation(&vec) != DS_OK ) return FALSE;
 return SUCCEEDED(m_lpds3Db->SetConeOrientation(vec.x, fValue, vec.z, DS3D_IMMEDIATE));
```

```
}


BOOL Sound::SetConeOrientationZ(float fValue)
{
 D3DVECTOR vec;
 if ( m_lpds3Db->GetConeOrientation(&vec) != DS_OK ) return FALSE;
 return SUCCEEDED(m_lpds3Db->SetConeOrientation(vec.x, vec.y, fValue, DS3D_IMMEDIATE));
}


BOOL Sound::SetConeInsideAngle(DWORD dwValue)
{
 DWORD dwIn, dwOut;
 if ( m_lpds3Db->GetConeAngles(&dwIn, &dwOut) != DS_OK ) return FALSE;
 return SUCCEEDED(m_lpds3Db->SetConeAngles(dwValue, dwOut, DS3D_IMMEDIATE));
}


BOOL Sound::SetConeOutsideAngle(DWORD dwValue)
{
 DWORD dwIn, dwOut;
 if ( m_lpds3Db->GetConeAngles(&dwIn, &dwOut) != DS_OK ) return FALSE;
 return SUCCEEDED(m_lpds3Db->SetConeAngles(dwIn, dwValue, DS3D_IMMEDIATE));
}


BOOL Sound::SetConeOutsideVolume(DWORD dwValue)
{
 return SUCCEEDED(m_lpds3Db->SetConeOutsideVolume(dwValue, DS3D_IMMEDIATE));
}


BOOL Sound::SetSourceAffectDirectHF(BOOL bValue)
{
 DWORD dwReceived, dwFlags;
 if  (  FAILED(m_lpksps->Get(DSPROPSETID_EAX_BufferProperties,  DSPROPERTY_EAXBUFFER_FLAGS,
NULL, 0, &dwFlags, sizeof(DWORD), &dwReceived)))
  return FALSE;

 dwFlags &= (0xFFFFFFFF ^ EAXBUFFERFLAGS_DIRECTHFAUTO);
 if ( bValue ) dwFlags |= EAXBUFFERFLAGS_DIRECTHFAUTO;

 return                          SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_FLAGS, NULL, 0, &dwFlags, sizeof(DWORD)));
}


BOOL Sound::SetSourceAffectRoom(BOOL bValue)
{
 DWORD dwReceived, dwFlags;
```

```
    if  (  FAILED(m_lpksps->Get(DSPROPSETID_EAX_BufferProperties,  DSPROPERTY_EAXBUFFER_FLAGS,
NULL, 0, &dwFlags, sizeof(DWORD), &dwReceived)))
     return FALSE;

    dwFlags &= (0xFFFFFFFF ^ EAXBUFFERFLAGS_ROOMAUTO);
    if ( bValue ) dwFlags |= EAXBUFFERFLAGS_ROOMAUTO;

    return                          SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_FLAGS, NULL, 0, &dwFlags, sizeof(DWORD)));
   }


   BOOL Sound::SetSourceAffectRoomHF(BOOL bValue)
   {
    DWORD dwReceived, dwFlags;
    if  (  FAILED(m_lpksps->Get(DSPROPSETID_EAX_BufferProperties,  DSPROPERTY_EAXBUFFER_FLAGS,
NULL, 0, &dwFlags, sizeof(DWORD), &dwReceived)))
     return FALSE;

    dwFlags &= (0xFFFFFFFF ^ EAXBUFFERFLAGS_ROOMHFAUTO);
    if ( bValue ) dwFlags |= EAXBUFFERFLAGS_ROOMHFAUTO;

    return                          SUCCEEDED(m_lpksps->Set(DSPROPSETID_EAX_BufferProperties,
DSPROPERTY_EAXBUFFER_FLAGS, NULL, 0, &dwFlags, sizeof(DWORD)));
   }


   BOOL Sound::SetMinDistance(float fValue)
   {
    return SUCCEEDED(m_lpds3Db->SetMinDistance(fValue, DS3D_IMMEDIATE));
   }


   BOOL Sound::SetMaxDistance(float fValue)
   {
    return SUCCEEDED(m_lpds3Db->SetMaxDistance(fValue, DS3D_IMMEDIATE));
   }
```

OpenAL

All the function calls that start with ‘gl’ is OpenGL calls. The glut…() calls are for the platform independent window handling as mentioned earlier. Through this example one can see how similar OpenGL and OpenAL is.

```
   static void display( void );
   static void keyboard( unsigned char key, int x, int y );
   static void reshape( int w, int h );
```

```
static void init( void );

static ALuint left_sid = 0;
static ALuint right_sid = 0;
static ALfloat left_pos[3] = { -4.0, 0.0, 4.0 };
static ALfloat right_pos[3] = { 4.0, 0.0, 4.0 };

static void display( void )
{
    static time_t then = 0;
    static ALboolean left = AL_FALSE;
    time_t now;

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );

    gluLookAt( 0.0, 4.0, 16.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0 );

    /* Inverse, because we invert below. */
    if( left == AL_FALSE ) {
        glColor3f( 1.0, 0.0, 0.0 );
    } else {
        glColor3f( 1.0, 1.0, 1.0 );
    }

    /* Draw radiation cones. */
    glPushMatrix( );
    glTranslatef( left_pos[0], left_pos[1], left_pos[2] );
    glRotatef( 180, 0.0, 1.0, 0.0 );
    glutWireCone( 1.0 , 2.0, 20, 20 );
    glPopMatrix( );

    if( left == AL_FALSE ) {
        glColor3f( 1.0, 1.0, 1.0 );
    } else {
        glColor3f( 1.0, 0.0, 0.0 );
    }

    glPushMatrix( );
    glTranslatef( right_pos[0], right_pos[1], right_pos[2] );
    glRotatef( 180, 0.0, 1.0, 0.0 );
    glutWireCone( 1.0 , 2.0, 20, 20 );
    glPopMatrix( );

    /* Let's draw some text. */
    glMatrixMode( GL_PROJECTION );
    glPushMatrix( );
    glLoadIdentity( );
    glOrtho( 0, 640, 0, 480, -1.0, 1.0 );
```

```
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );
    glTranslatef( 10.0, 10.0, 0.0 );
    glScalef( 0.2, 0.2, 0.2 );
    glColor3f( 1.0, 0.0, 0.0 );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, 'R' );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, 'e' );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, 'd' );
    glColor3f( 1.0, 1.0, 1.0 );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, ' ' );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, 'i' );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, 's' );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, ' ' );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, 'A' );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, 'c' );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, 't' );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, 'i' );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, 'v' );
    glutStrokeCharacter( GLUT_STROKE_ROMAN, 'e' );

    glMatrixMode( GL_PROJECTION );
    glPopMatrix( );

    now = time( NULL );

    /* Switch between left and right boom every two seconds. */
    if( now - then > 1 ) {
     then = now;

     if( left == AL_TRUE ) {
         alSourcePlay( left_sid );
         left = AL_FALSE;
     } else {
         alSourcePlay( right_sid );
         left = AL_TRUE;
     }

    }

    glutSwapBuffers( );
    glutPostRedisplay( );
}

static void keyboard( unsigned char key, int x, int y )
{

    switch( key ) {
    case 27:
     exit( 0 );
     break;
    default:
```

```
     break;
    }

}

static void reshape( int w, int h )
{
    glViewport( 0, 0, w, h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity( );
    gluPerspective( 60.0f, (float) w / (float) h, 0.1f, 1024.0f );
}

static void init( void )
{
    ALfloat zeroes[] = { 0.0f, 0.0f, 0.0f };
    ALfloat front[] = { 0.0f, 0.0f, -1.0f, 0.0f, 1.0f, 0.0f };
    ALfloat back[] = { 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f };
    ALuint sources[2];
    ALuint boom;
    void* wave;
    ALsizei size;
    ALsizei bits;
    ALsizei freq;
    ALsizei format;

    glClearColor( 0.0, 0.0, 0.0, 0.0 );
    glShadeModel( GL_SMOOTH );

    alListenerfv( AL_POSITION, zeroes );
    alListenerfv( AL_VELOCITY, zeroes );
    alListenerfv( AL_ORIENTATION, front );

    alGetError();
    alGenBuffers( 1, &boom );
    if(alGetError() != AL_NO_ERROR) {
     fprintf( stderr, "aldemo: couldn't generate samples\n" );
     exit( 1 );
    }

    alutLoadWAV( "boom.wav", &wave, &format, &size, &bits, &freq );
    alBufferData( boom, format, wave, size, freq );

    alGetError();
    alGenSources( 2, sources );

    if(alGetError() != AL_NO_ERROR) {
     fprintf( stderr, "aldemo: couldn't generate sources\n" );
     exit( 1 );
    }
```

```
    left_sid = sources[0];
    right_sid = sources[1];

    alSourcefv( left_sid, AL_POSITION, left_pos );
    alSourcefv( left_sid, AL_VELOCITY, zeroes );
    alSourcefv( left_sid, AL_ORIENTATION, back );
    alSourcei( left_sid, AL_BUFFER, boom );

    alSourcefv( right_sid, AL_POSITION, right_pos );
    alSourcefv( right_sid, AL_VELOCITY, zeroes );
    alSourcefv( right_sid, AL_ORIENTATION, back );
    alSourcei( right_sid, AL_BUFFER, boom );
}

int main( int argc, char* argv[] )
{
    /* Initialize GLUT. */
    glutInit( &argc, argv );

    glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize( 640, 480 );
    glutInitWindowPosition( 0, 0 );
    glutCreateWindow( argv[0] );

    glutReshapeFunc( reshape );
    glutDisplayFunc( display );
    glutKeyboardFunc( keyboard );

    /* Initialize ALUT. */
    alutInit( &argc, argv );

    init( );

    glutMainLoop( );

    return 0;
}
```