Math 198 Final Project: illiVSS

Raeed Chowdhury

December 16, 2009

1 Project

My Math 198 project was to bring the Virtual Sound Server (VSS) back into use by the CUBE. VSS is basically a client-server application that can synthesize sound dynamically and remotely. This project had several important milestones:

- Connecting the client computer to the server computer to send messages
- Creating sound at the server computer by creating an aud file on the client computer
- Creating a good example of dynamically controlling sound from user input using an audpanel (FilterDemo)
- Creating a viable dynamically sonified version of an already existing CUBE application (Ribbons)

1.1 NOTE

Because my project was bring VSS back from the proverbial purgatory of code, much of it dealt with learning about VSS and a new syntax used in the aud files that VSS uses to create sound. Because of this, this report gives an overview of what VSS is and the basics of how VSS works. However, this document cannot possibly hope to explain everything about VSS; as such, it will reference several documents and files that have already been written to teach how to use VSS more completely. As with the learning of

any new language, these examples are very helpful. All of the files, executables, and documentation mentioned in this document can be found at http://new.math.uiuc.edu/math198/MA198-2009/chowdhu1/.

1.2 Connecting to VSS

Connecting to a remote VSS from a client requires, first of all, that VSS is running on the desired server computer. To do this, at a Windows computer, start vssDemo.exe by openning a shell prompt at the directory where it is stored and typing

start vssDemo

.
At this point, to connect to this computer, find out its IP address by typing in

ipconfig /all

in another shell prompt and write it down. At the client computer, open a shell prompt and move to the directory of the desired program, and set an environment variable SOUNDSERVER to the IP address we wrote down earlier like so

set SOUNDSERVER=IPaddress

where IPaddress is replaced by the IP address. To confirm a connection, type vssPing at the prompt where vssPing.exe is accessible. If the computers are not connected, we get an error message, and we try again by making sure the computers are on the same network, the server side does not have a firewall blocking it, we have the right IP address, etc.

1.3 Creating Sound

The second milestone in creating dynamically controlled sound through VSS is actually making a sound. There are several example audfiles and a description of how exactly to make sound in the "Guide to adding sound to your C/C++ application" link in the Reference section of the webpage mentioned above, so this document will leave the teaching of how to write audfiles to

the pages of documentation provided there (the only difference here is that we will use the program "audtest" to test aud files, instead of the mentioned program "example").
Basically, to synthesize sound, VSS uses files called aud files. These files are written in a style relatively similar to an interpretted language like Python, as that they are not necessarily compiled or assembled, but they are used as scripts by VSS to create sounds. The basic syntax of an aud file includes four main aspects:

1. Load Dynamic Libraries
2. Create sound actors
3. Create message groups
4. Add sound handles to message groups or play sound handles

Note that message groups are explained in more detail in the next section about creating dynamically controlled sound.
The creation of sound actors and sound handles are analogous to creating creating objects in object-oriented programming. A sound actor is basically a parent object from which many objects that actually make sound can be created. For example, when

h = Create FluteActor;

is called, an instance of FluteActor is created (h). A sound handle is an actual sound playing instance created out of a sound actor. For example,

n = BeginSound h;

is called, a sound handle (n) is created out of the parent actor (h) and the sound handle begins playing. Each handle has several modification functions, depending on the type of handle (complete documentation can be found on the reference website given above). A developer can also use an actor to modify all handles that stem from it at once, using similar functions (documented at the reference website). A single actor can have several handles created from it, so in truth, only one actor of each type is ever really needed, unless a developer wants to create several different sounds from one actor

and manipulate them all together, separately from another group of handles under another actor.
This is the basic idea behind creating sounds using aud files.

1.4 Dynamic Sound from an Audpanel

Dynamic control of sound in VSS is acheived through a special type of actor called the message group. In an audfile, a message group is analogous to a subroutine or function in other programming languages like Python. Bascially, when a message group is called, the contents of the message group, which are actually commands for VSS, are executed. To do this as a developer, one would create a MessageGroup actor and use the command AddMessage to add certain commands to a message group. There is also a special syntax for numerical input to the aud file: when a message in a message group requires a numerical value, we use the "*(index)" syntax to specify which element of a data vector passed to an AUDupdate() function in a program to use in the message (see presentation.aud for good examples of message group use).
At the time this report is being written, no C/C++ program has been successfully compiled and linked with vssClient.h and libsnd.a for this Math 198 project due to the use of older and incompatible libraries in this ten year old edition of VSS, but a substitute for developing the aud files was available. The Audpanel was created as a stub for developing the auditory parts (the aud files) of a program so that a developer could separate the auditory bugs from the program bugs. A detailed description of how exactly to create an Audpanel can be found on the reference website, but the essential parts are listed below:

- Set name of Audpanel
- Set number of possible sliders in every preset message group
- Set names of preset
- Set names of message group per preset
- Set default data sending rate per preset
- Set "'slider"' types (button, radio buttons, slider, square) per preset

- Set "'slider"' names
- Set values for each "'slider"' per preset
- Set mapping to use for data vector of message groups per preset

The most important part of the audpanel is the communication with the audfile it is made for; this is acheived through setting the message groups to interact correctly. Each preset in the audpanel has a specific message group. This message group can receive data vector inputs from the sliders, with the indices used in the messages corresponding to the mapping set in the audpanel. To use a button, radio button set, or a square, the name of this input device needs to be appended to the message group name with an underscore in between the original message name and the device name. One thing to note is that the mapping set in the audpanel file is only used for the sliders; since the other buttons and such use different message groups, they have a slightly different setup for the data vector indices. A very nice table for the usage of all the input devices is provided in the audpanel documentation in the reference part of the website mentioned above.

1.5 FilterDemo

The first actual dynamically sonified program created in this project was a very rudimentary equilizer. For the actual code to this part of the project, refer to presentation.ap and presentation.aud (the parts that refer to filterMsg). The basic idea behind this equilizer is to create sound and pass it through a simple set of filters to attenuate certain frequencies of the sound individually. This program used the sound generating actors MarimbaActor, NoiseActor, and ShimmerActor, as well as the processing actor FilterActor. Basically, when a button for a sound was pressed, the input to each of six filters was set to that sound, and the relative amplitude of each filter was controlled by a slider. This idea in itself is actually very simple, but the experience with message groups was invaluable to learning about how implement VSS back into the CUBE.
Taking a more in depth look at this particular program, we see that the marimba sound is basically a descending then ascending set of marimba notes, the noise sound is basically random noise that spans the frequencies the filters in this demo span, and the shimmer sound is a set of randomly walking tones and a set of harmonics of these tones. The filters are all bandpass filters

centered at the center frequency of each range of frequencies listed above each slider. The filters are meant to simulate analog filters, so they have a fairly wide transition band to represent a more "'natural"' sound. The resonance of each bandpass, which has an effect of the width of the passband, can also be set. In this case, it is set such that passbands of each bandpass do not touch in the frequency domain, i.e. there are noticable gaps in the frequency domain for the output. This was done on purpose so that when the marimba sound was played, we could hear which band the sound was passing through at a particular time.
This demo is also a good spot for anyone to begin messing around with message groups and audfiles to see what effect user input has on sound; it would be a good program with which to get familiar with message groups.

1.6 Ribbons

The last part of this project was to sonify the existing CUBE application called "Ribbons". This application is basically a three-dimensional painting program; it allows the user to draw colorful ribbons in three dimensions with the wand used to control the CUBE. However, unfortunately, at the time this document is being written, VSS has not been successfully linked with Aszgard, the platform on which the CUBE runs. Camille Goudeseune, one of the CUBE's caretakers, is currently trying to link VSS with Aszgard. Until then, though, it seems that we cannot get VSS into the CUBE.
However, I have created the audfile that would likely sonify Ribbons, using the audpanel. To access this part of the audpanel, click the Ribbons preset in the preset menu. The relevant code is also in presentation.ap and presentation.aud. The idea of this sonification of Ribbons is that while the user was drawing ribbons with the wand, the sound would appear to come from the wand tip, and the frequency of the sound would be proportional to the speed of the wand. In the audpanel, this is represented by the frequency slider in the middle and the square/slider combination on the right. The frequency slider controls the frequency of one of the four sounds that can be chosen on the left. (NOTE: Marimba only works while the frequency is changing, and Mandolin doesn't seem to work). The square/slider combination controls the position of the sound, with the square controlling the pan and the elevation and the slider controlling the distance of the sound. (NOTE: the square only has a range from 0 to 1 in each direction, so the pan left and the negative elevation cannot be set with this particular audpanel). This square/slider

combination can actually be replaced by the SetXYZ command in the aud file, which would be easier to work with in the CUBE, but for working with the audpanel for the demonstration, we needed to use each of the three commands, SetPan, SetElev, and SetDistance.

1.7 Adding Sound to a C/C++ application

Although the audpanel provides a good demonstration, we would like to eventually put these aud files to use in a C/C++ program, once everything can be compiled and linked properly. Fortunately, once a developer has an aud file working with an audpanel, it is relatively simple to make this aud file work with a C/C++ program. The basic steps are:

1. Include "vssClient.h" in your program
2. Link with libsnd.a
3. Before any intensive processing, call *BeginSoundServer()*
4. Initialize aud file by calling *AUDinit("filename.aud")*
5. Whenever we need to update the aud file, call *AUDupdateSimpleFloats("msgName",dataVectorSize,data[0],data[1],...)*
6. At the end, call *EndSoundServer()*

To clarify, AUDupdateSimpleFloats(...) has the parameters:

- msgName–name of message group to send
- dataVectorSize–size of data vector that message group needs
- data[0]-data[n]–each of the data vector numbers, in order

Using this simple scheme and an already developed aud file, it *should* be simple enough to add sound to an application.

1.8 Future

The most desirable future for this project is to get VSS back into the CUBE. Of course, there are a small number of steps required before this project can get to that point.

- Find a C/C++ platform to compile VSS
- Create a basic C/C++ program that creates dynamic sound using an already tuned audfile (like one developed during this project, or using an audpanel)
- Help or wait for Camille to compile and link VSS with Aszgard
- Add aud file compatibility to an already existing CUBE application, like Ribbons. Use an aud file that has already been developed in this project or one developed using an audpanel.

If this project reaches this point, it will have been a success, in that it set out to fullfill its goal of bringing VSS back to the CUBE. The next step would be to make it a permanent part of creating CUBE apps, so that future Math 198 students would be able to use it to sonify all of their CUBE applications.