Evaluation of Low Latency Audio Synthesis Using a Native Java – ASIO Interface

Richard D. Corbett, Kees van den Doel, and Dinesh K. Pai

Department of Computer Science, University of British Columbia
email: {rcorbett, kvdoel, pai}@cs.ubc.ca

Abstract

A real time user interface for sound synthesis should have the smallest latency achievable. With recent advances in technology the possible latencies are becoming impressive for even the standard desktop computer. This paper discusses a method of low latency sound synthesis using Java with a native interface to the audio hardware. We describe the implementation and report the results achieved for a test application with mouse interaction.

1 Introduction

With the recent increase in computing power it has become possible to accurately model sounds of the real world in virtual environments, and synthesize them in real time on desktop computers (Gaver 1993, Hahn *et al 19*95, Doel, Kry, and Pai 2001). It is also becoming increasingly possible to create responsive user interfaces with auditory feedback that respond with minimal delay. Computations can be performed fast enough to simulate real world sound events using any one of many representative models. One limiting factor, which governs the degree to which a synthesized sound can be perceived as natural, is its latency. Even the most accurate synthetic sounds are not realistic if they don't occur at the precise moment that they are needed. A good user interface should take the input from the user and as quickly as possible synthesize and output the desired sound.

Latency is often measured as the delay in sound throughput (MacMillan, Droettboom, and Fujinaga 2001). In the case of sound synthesis, where sound input is not required, only the output latency is of interest. In this paper the use of the word 'latency' refers to the interval of time required, once user input is received, before the desired signal is emitted from the sound card. For instance, in the case where the user clicks a button on a windows application, the latency is defined as the interval in time that passes between the mouse click and the emergence of the beep at the speakers of the PC.

As need is recognized for lower latency sound synthesis, more strategies are developed to help get the synthesized sounds out through the sound card as fast as possible. In recognition of the need for lower latency Microsoft developed the DirectX drivers for sound cards. More recently, recognizing the need for even lower latency, Steinberg Media Technologies AG created ASIO (Audio Streaming Input Output) (Steinberg 1999).

This paper will discuss an implementation that uses ASIO in an effort to minimize the latency of the Java Audio Synthesis System (JASS) (Doel and Pai, 2001), a previously developed cross-platform sound synthesis package written in pure Java. An application written in pure Java can run on many operating systems and is also easily deployed on the web. The downside is that current Java implementations are slower than C language programs for certain tasks such as array access. We have found that Java audio synthesis applications run 1.5 to 2 times slower than similar C coded applications. Another disadvantage of using pure Java for audio synthesis is the high latency. Java provides a means to output synthesized sound to the audio hardware through the JavaSound API, which unfortunately has very high latency of the order of 200ms (at least on the Windows platform). Therefore, using only pure Java latency is too high for a usable real time user interface for sound synthesis, even for prototyping.

Since we desired lower latency an implementation using JNI (Java Native Interface) to access the audio hardware more directly through ASIO was undertaken. JNI is an interface that gives Java the ability to use native methods (usually written in C) that lie in libraries outside the scope of the Java programming language. Native methods are used also in the Java audio synthesis toolkit Jsyn (Burk 1998) not just for the final rendering of the audio but also for computation within the unit generators for efficiency.

The development done for the purposes of this paper was completed in part to determine if JNI is fast enough, and provides enough functionality, to work with ASIO on desktop machines. The study was conducted with the interest of quantifying performance on standard desktop machines, therefore, real-time operating systems were not evaluated. In our tests the operating system specific ASIO interface is restricted to a single Java class, minimizing the impact on portability. The implementation with full source code will be made

available on the web for non-commercial use at www.cs.ubc.ca/~kvdoel/jass.

2 ASIO Interface with Java

JASS was used to create the sounds to be output through the ASIO system. Much care had to be taken to make sure that the Java classes could interface through the JNI to have a working connection with the ASIO system. We have added a class to JASS that takes care of the output of the sound buffers, which are computed in pure Java, using JNI to interface to ASIO.

An investigation was done into the PortAudio API because it is designed to create a portable audio API for most platforms. (Burk and Bencina 2001) and supports ASIO. In the end PortAudio could not be used because of difficulties in getting it to work through JNI. As will be described, great care must be taken with the treatment of threads when making function calls through JNI, and the PortAudio API, due to its user-friendly interface designed for C, is too demanding on JNI.

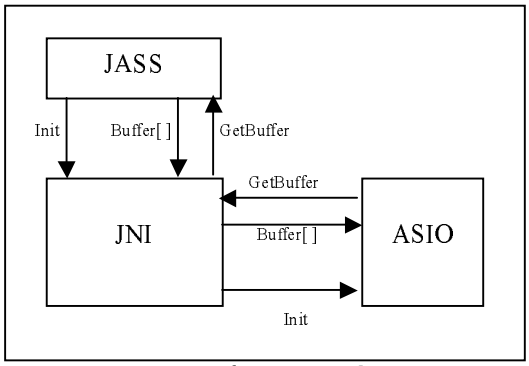

Figure 1: Overview of interaction between JASS and ASIO

Figure 1 gives a brief description of how JASS was interfaced with ASIO. ASIO is designed to have the driver call back to its host application (in our case, JASS) when it needs more data. There is a double buffer in the driver, and as buffer 1 is being output, buffer 2 must be filled with the next set of output data. Once Buffer 1 has been fully output, the callback is called again and this time the new data in buffer 2 is output while the host application is busy filling buffer 1 with the new data. So, in order to get the cycle running, an initialization call is sent to ASIO through JNI and then ASIO needs some way to call back into JASS and ask for the data to fill the buffers, as they are required.

Since this system is based on a pull mechanism, i.e., the ASIO driver pulls the data out of the host application as it needs it, JNI had to be used in a way that would allow calls to the Java Virtual Machine (JVM) from ASIO asynchronously.

2.1 Using JNI With ASIO

Timing is an important factor to be considered when interfacing ASIO with JASS. ASIO can put high demands on the computing of buffers in the host application. For instance, running at 48000hz and using an ASIO buffer size of 48 samples (the minimum buffer size on a SoundBlaster Audigy) JASS can expect a request for data once every millisecond. So, to get this interface to work for all possible buffer sizes, JNI must be able to handle calls back to JASS from ASIO at a minimum of 1000Hz. It is very important when putting high demands on JNI to consider the use of threads. Since the control of the timing is passed over to ASIO once it is initialized, JASS must fill the requests for buffers as fast as possible.

In order to pass control to ASIO, allowing it to call the host Java functions, the Java Virtual Machine must remain accessible to the ASIO callback function. Therefore, once the initialization call is received from JASS, a reference to the JVM should be retained. The JVM is used to allow native code to create its own threads and then use these threads for asynchronous calls through JNI at the demand of ASIO. If a request for data from ASIO is not returned before another call must be made to keep the ASIO buffer cycle running, unexpected race conditions may arise, as the callback will attach yet another thread to the JVM, with unpredictable behavior.

2.2 Full System Description

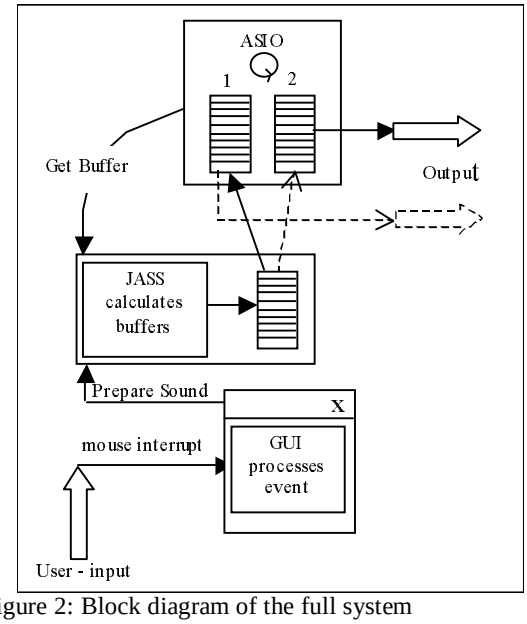

Figure 2: Block diagram of the full system

The full sound synthesis system starts from user input at the mouse and finishes with the desired sound arriving on the speakers. Figure 2 provides a visual representation starting with the user input. Once the user input has been received when the mouse is clicked on the button (from the Java.awt library) a mouse interrupt is created in the operating system. Once this mouse interrupt is interpreted a message is dispatched to the GUI. The GUI

interprets the message and tells JASS to prepare to create the buffers for the sound.

Once ASIO has begun running, it will ask for buffers, as they are needed. At this point ASIO queries JASS for a buffer, while outputting the other internal buffer. In the diagram the solid lines connected to the ASIO buffers represent what the buffers are doing during cycle 1. Once this cycle is complete, cycle 2 (dotted lines) will occur and this pattern will continue until the application is terminated.

3 Latency Measurements

In the context of this paper latency is, as stated before, the delay between the user request for sound and the detection of the requested sound at the output. Since ASIO uses a streaming double-buffered architecture the theoretical latency can be calculated (assuming no GUI or input latency) as the time it takes one partition of the double buffer to be played. For instance, if the sample rate is set to 48000hz, and each half of the double buffer holds 48 samples, then the latency could be expected to be 1ms. In fact, this is the case, once the output has begun. However, this value cannot be used in our definition of latency because it overlooks the latency due to user input, which as will be shown is quite substantial.

The latencies were measured with a simple GUI , which takes user input through a mouse click on a button. A microphone was set up next to the mouse, and the speakers are placed close to the microphone. In this way, any recording software can record the sound of the mouse click and a small period of time later the synthesized sound will appear on the output of the speakers. The latency can be found as the time interval from the mouse click to the emergence of the sound at the speakers. In Figure 3 we show a typical recording.

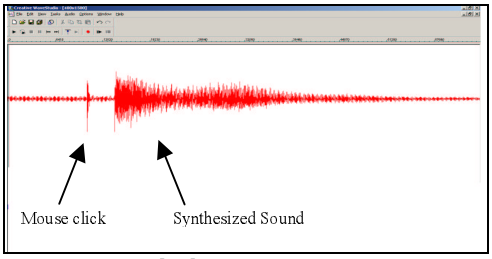

Figure 3: Sample data set

3.1 Test Results

When investigating the latency two variables were manipulated: the ASIO buffer size and the system load. Five buffer sizes were investigated at two different CPU loads. In all tests the SoundBlaster Audigy soundcard was used because it is the cheapest sound card on the market that comes with ASIO drivers. These tests were designed with the intention of providing information about standard affordable desktop performance.

The JASS system was used to synthesize a bell sound using a modal model, which models the bell as a bank of reson filters with well defined frequencies, dampings and gains, which is excited by an impulse. Choosing the number of active modes can vary the sound quality, and the computational expense. This is how the CPU load was varied.

If the buffer size is too small for a certain load the sound will break up. The correct sound is still audible, but there are very short gaps in between the correct sounds that make it sound crackly. By increasing the buffer size, the latency is increased, but the quality of sound is more dependable. A familiar example of this is the 'shock-proofing' of portable CD players. They use large buffers so that if an interruption in sound occurs (the CD skips) the user will not hear the break up in sound because there is enough data stored in the buffer to continue sound output and allow the hardware to catch up. Latency is not a large concern in CD players because they output continuous sound, and as long as the output stream is unbroken, the customer is satisfied.

3.2 Minimum buffer Sizes

We have measured the minimum buffer size such that sound does not break up under various loads obtained by varying the number of modes that were synthesized. The results are depicted in Figure 4.

# modes	% cpu load(Sys 2)	System 1	System 2
1	3	96 (2)	800 (16.7)
10	5	96 (2)	800(16.7)
100	17	96 (2)	800(16.7)
200	30	150 (3.13)	800(16.7)
300	43	200 (4.17)	1200(25)
400	57	800 (16.7)	1800(37.5)
500	70	1500 (31.3)	2200(45.8)
600	83	--	4800(100)

Figure 4: Minimum ASIO buffer sizes for different system loads as determined by synthesizing a given number of modes. In parenthesis are the calculated latencies for the given buffer sizes in milliseconds.

These tests were performed on two different machines to give an idea of variability in performance. System 1 had dual Pentium® III 750Mhz processors with 512Mb RAM running Windows 2000. System 2 had a Pentium® III 950Mhz processor with 512 Mb RAM and Windows ME. There is no CPU load data for system 1 because it has a dual processor and it was felt that the CPU load values would give inaccurate impressions of the actual computer load. This is because one processor would saturate while the other was idle, due to the fact that we restricted ASIO to using mainly one thread. As stated before, both machines used the SoundBlaster Audigy.

The test with 600 modes could not be completed on system 1. On attempt not even the maximum buffer size could produce an unbroken sound. It

appeared that since ASIO and the Java Virtual Machine are sharing the same thread, only one of the two processors was used and it was saturated at this load.

3.3 Latency Measurements

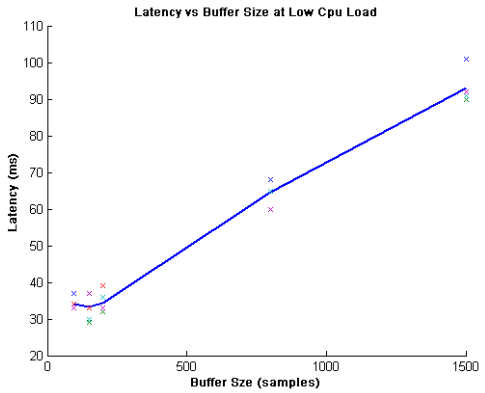

Figure 5: Latencies at low load

In Figure 5 we plot the measured latency at low CPU load as a function of ASIO buffer size. Because of the unpredictability of the CPU load in multitasking environments five separate measurements were taken. It can be seen that the latency increased linearly with buffer size as expected. It can also be seen that if linear extrapolation is used it appears that our GUI has an inherent latency of roughly 30ms. Since only 10 modes were used, our smallest buffer size of 96 samples was large enough to support the synthesis. If more computing was required, as in the next set of data, 96 samples is not a large enough buffer and too much demand is put on the buffer synthesis, resulting in the breakup of the sound.

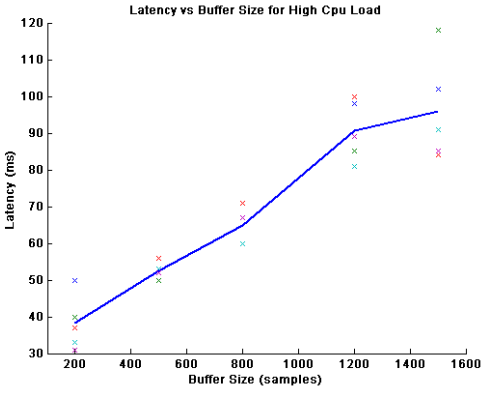

Figure 6: Latencies at high load

In Figure 6 we plot the latencies under high load. A similar trend as with the light load measurements can be observed, and the GUI latency derived from this is unchanged at the high load. However, the smaller buffer sizes (96, 150) were not big enough to allow JASS enough time to synthesize the required buffers. This is presumably due to process swapping and might be improved by running the synthesis at higher priority.

4 Conclusions

We have described an implementation of a low latency synthesis interface to ASIO from pure Java within the JASS synthesis system. It was found that JNI interface is efficient enough to satisfy the strict timing requirements for real-time audio synthesis with low latency. We have tested the latency of the system triggered by a Java GUI on a Windows system and found that the minimum latency depends strongly on the system load. By measuring the latency at several ASIO buffer sizes we were able to reconstruct the latency of the GUI.

The implementation is made available on the web for anyone to use.

Acknowledgements

This work was supported in part by grants from the UBC Peter Wall Institute for Advanced Studies and NSERC.

References

Burk, Phil. 1998. “Jsyn – A Real-time Synthesis API for Java”. *Proceedings of the International Computer Music Conference.* International Computer Music Association.

Burk, Phil and Bencina, Ross. 2001. “PortAudio - an Open Source Cross Platform Audio API”. *Proceedings of the International Computer MusicConference.* International Computer Music Association.

Gaver, W. 1993. “Synthesizing Auditory Icons”. *Proceedings of the ACM INTERCHI.pp. 228 – 235.*

Hahn, James K., J. Geigel, J. W. Lee, *et al. 1995.* “An Integrated Approach to Motion and Sound”. *Journal of Visualization and Computer Animation. 28(2): 109-123.*

MacMillan, K., M. Droettboom, and I. Fujinaga. 2001. “Audio Latency Measurements of Desktop Operating Systems”. *Proceedings of the International Computer Music Conference.* International Computer Music Association.

Steinberg. 1999. “Steinberg Audio Streaming Input Output Specification: Development Kit 2.0”. Steinberg Soft- und Hardware GmbH.

van den Doel, Kees, and Pai, Dinesh K,. 2001 “ JASS: A Java Audio Synthesis System for Programmers”. *Proceedings of the International Conference on Auditory Displays*.

van den Doel, Kees, P. G. Kry, and D. K. Pai. 2001. “FoleyAutomatic: Physically-based Sound Effects for Interactive Simulation and Animation”. *Computer Graphics (ACM SIGGRAPH 01 Conference Proceedings).*