An Architecture for the Java VST Wrapper that supports the Implementation of Digital Sound Synthesis Algorithms in an Educational Environment

Joseph Timoney*, Robert Voigt*, Stephen Brown* and Victor Lazzarini**

**Department of Computer Science*
National University of Ireland Maynooth
email: jtimoney@cs.nuim.ie;
robert.a.voigt@nuim.ie;
sbrown@cs.nuim.ie

***Department. of Music*
National University of Ireland Maynooth
email: victor.lazzarini@nuim.ie

***Abstract* — A framework to support the development in an educational environment of real-time, digital sound synthesis algorithms is proposed. Sound synthesis algorithms are an important subset of Digital Signal Processing (DSP) and are an excellent way of teaching the application issues of many DSP concepts. Steinberg's Virtual Studio Technology (VST) is a very flexible format for creating digital sound synthesis and audio effect plugin applications. The company provides an associated C/C++ SDK, and an open source wrapper (jVSTwRapper) is available that allows the plugin code to be written in Java. However, the jVSTwRapper documentation is sparse and the examples bundled are difficult to extrapolate from, reducing its effectiveness in an educational context. This paper proposes an improved interface for the JVSTwRapper that comprises a novel generalised voicing structure. This simpler interface built on top of the existing jVSTwRapper architecture allows synthesis algorithms to be implemented more easily, thereby shifting the development emphasis from the VST implementation to the higher-level DSP & Synthesis concepts. This abstraction means that an educator can focus more thoroughly on the core DSP material while simultaneously taking advantage of the benefits that the Java language offers for students.**

***Keywords* – Digital Sound Synthesis, Audio Processing, VST SDK, JVSTwRapper, Java**

I INTRODUCTION

Digital sound synthesis is an important aspect of Digital Signal Processing (DSP) for audio. As both a research and a commercial field, it has expanded significantly in the last two decades driven by both the fall in the cost, and rise in the power, of hardware and software. Common digital sound synthesis algorithms include band-limited (or low-aliasing) oscillators, envelope generators, analogue filter models, and sample/ wavetable players [1].

Generally, a university module in this field would be taught at advanced undergraduate or at Master's level, following a foundation module in DSP. It is an excellent subject as it offers a great opportunity for students to apply what they have learnt in the foundation module to produce applications that are immediately satisfying: they can generate live sound, but are also intellectually challenging because many signal processing elements must be combined and real-time user interaction elements incorporated. In particular, the requirement for interaction demands that the implementation focus shift from the well-known Matlab [2], which is an excellent analysis and prototyping tool for DSP but is not well designed for real-time, interactive execution, to a more suitable platform. C/C++ and Java, for example, provide more application-oriented support along with libraries for GUI creation and sound processing, and facilitate more direct interfacing with the operating system and peripheral hardware. Moreover, a particularly good choice for building Digital Sound synthesis applications is the popular VST (Virtual Studio Technology) standard of Steinberg [3]. A Software Development Kit (SDK) for C/C++ is available from the company, and *plugins* (i.e. synthesis applications) designed with this can be used in many digital music production environments,

both commercial and open source. This makes VST a very rewarding environment. Additionally, the VST framework obviates a need to detail the low-level mechanisms for dealing with MIDI input or the delivery of digital audio streams to the output D/A converter in a sound card. This allows the user to conceptualize them at a higher level as inputs and outputs to the VST system itself, and focus on the DSP elements of the system.

There are two significant issues that arose in the context of adopting VST for a module in Digital sound synthesis programming that was taught at NUIM in 2011:

> Firstly, the students had programming skills in Java but not C/C++. This was easily addressed by using the open source wrapper, jVSTwRapper [4], which supports Java-based VST plugins. The wrapper itself is a native compiled binary that invokes a Java Virtual Machine (JVM) and delegates native calls to the VST plugins implemented in Java. This provides programming advantages for the students, as Java does not have the difficulties associated with C/C++ syntax, provides better compile-time and run-time checking, and also provides platform-independence.
>
> Secondly, the accompanying documentation of both the SDK and the jVSTwRapper is sparse and examples are free-form. To allow the students to concentrate on the theory and implementation of the various digital sound synthesis algorithms, rather than detailed interface mechanisms, it was necessary to provide a more abstract architecture on top of the VST framework.

The solution to this motivated the work that is presented in this paper, where a structured rewriting of the audio processing methods for the VST wrapper will be presented. This is done in such a way that is flexible, allowing for the implementation of any sound synthesis technique, and can be achieved in a smooth and uncomplicated manner. Effectively, it means that the sound generation elements can be written into various separate classes that can be used as required, giving much more freedom and much less stress to the synthesis application designer.

The paper is organised as follows. Section II gives a brief historic context to music synthesis software design, outlining the reasons for the popularity of the VST standard. Section III discusses audio processing in Java, explaining how the VST wrapper for Java is organised. Section IV details the new architectural interpretation created within the wrapper that leads to efficient VST creation. A discussion on the relationship of this solution to abstraction will be provided in Section V. Section VI provides conclusions and identifies areas for potential future work. Example code is presented in Appendix A.

II THE VST STANDARD

The practice of using computers to generate music has been around for a many years. The development of the software program “MUSIC” by Max Matthews in 1957 is acknowledged as being a major milestone [5]. As the power of the personal computer (PC) improved over the decades, more powerful applications appeared. In particular, the 1980s saw the beginnings of full music production environments for the PC, known as *Digital Audio Workstations*, albeit with technology-related limitations. With the rapid development in PC capabilities through the 1990’s, these production platforms began to include software emulations of the tools found in hardware-based studios. Such tools include sounds effect units and synthesizers. Each vendor produced and promoted their own standard for implementing and integrating these software tools, also known as *plugins* [6].

A plugin takes in a stream of audio data (in the case of an effect), or responds to MIDI note-on and note-off information (in the case of a synthesizer). It then sends audio data back to the software production environment (the host program). From the host’s viewpoint, the plugin is simply a black box that accepts inputs and produces outputs. However, the implementation of and communication with the plugin adhere to certain standards determined by the host vendor. Cubase offered the Virtual Studio technology (VST) standard, Cakewalk used DXi derived from Microsoft’s DirectX and Digidesign used the Real-Time Audio Suite (RTAS) standard [6]. Overall, the VST standard has become the most popular approach for two primary reasons:

1. VST plugins could be easily developed for both windows and Macintosh, a feature that was impossible with the DXi standard of Cakewalk.

2. Stenberg released a VST Software Development Kit (SDK) allowing any third-party developer to make their own effects and synthesizers. This helped to promote the standard as both professional and amateur developers appeared. This was unlike the more restrictive system employed by Digidesign that required developers to pay a fee to acquire their SDK, and to satisfy a number of conditions such as demonstrating a capability for object-oriented programming.

A VST plugin requires a host application to support it, originally intended to be the Cubase production environment. However, there are other VST-specific hosts that can be used to run the plugin

itself without the other elements of the production package. An example of this is “VSTHost” [7]. This facilitates efficient development as it is easier in terms of reloading a plugin during development.

The VST SDK release by Cubase is written in C/C++ [3]. It contains a complete set of methods that manage the communications between the hardware, the host, and the processing algorithms of the VST. A couple of simple audio effects examples are included with the SDK. The supporting documentation is limited, but sufficient for a reasonably experienced developer to begin to implement their own plugins in a short time frame. An open source wrapper that allows VST audio plugins to be written in Java was released in 2006 [4]. This allows the developer to write platform-independent plugins. The wrapper delegates calls to and from the host system to a Java interface so that the developer need only write a Java class that implements the wrapper interface and code the audio algorithm. The wrapper comes with a few examples of synthesizers. Limited documentation and sparse code comments mean that modifying these to build a different synthesizer is not straightforward, as there are a number of idiosyncrasies to overcome. These issues have detracted from its use in teaching. This motivated the creation of an alternate architecture, allowing the educator to focus on teaching sound synthesis algorithms in Java.

The next section briefly explains how the Cubase SDK C/C++ is structured, before the Java version is described.

III The C++ VST SDK

The VST 2.4 SDK is written as an object-oriented C++ interface used for the development of VST plugins, implementing both effects and instruments under a shared design. These C/C++ VST Plugins are compiled as platform-specific dynamic libraries *(DLLs for Windows, Shared Objects for Linux and Bundles for Mac OS).*

The core of the Plugin SDK comprises two base plugin classes: *AudioEffect* and *AudioEffectX*. The latter is derived from the former as a subclass implementing new functions and features on top of the existing VST 1.0 codebase (and deprecating some previous functionality) [3]. New VST plugins typically inherit from the abstract base class *AudioEffectX* and provide implementations of its many virtual methods. The majority of methods in *AudioEffectX* are implemented in the class definition, but some methods such as *processReplacing()* are declared as pure-virtual methods, necessitating implementation by the developer for a functioning plugin to be compiled. The *processReplacing()* method is called by the VST Host when the plugin is expected to perform some processing. The method arguments consist of two dynamically-sized, multi-dimensional arrays (in the pointer-to-pointer syntax: *TYPE** var*) for inputs and outputs and an integer specifying the size of the arrays (used when iterating over the input/output vectors of audio samples). The input and output arrays are multi-dimensional to support the use of an arbitrary number of channels on a per-plugin basis, with a typical implementation using nested loops to iterate between channels and individual samples alternately during processing.

One of the specific difficulties with this SDK is that the practice of implementing short arrays of samples (or buffers) in audio programming is often a conceptual stumbling block for new students. Another is the use of indirection by passing pointers to arrays of samples in the VST Plugin interface, adding another layer of complexity to the programming. These issues further hinder the VST standard's applicability as a learning tool. The following section explains the Java implementation of the VST that is intended to address these problems.

IV Voicing Architecture using JVSTwRapper

In JVSTwRapper, the constructor that describes the plugin being created is an extension to the wrapper’s *VSTPluginAdapter* class. The two key methods are:

- *processEvents()*, which processes MIDI events, typically by adding and removing notes from the list of active notes
- *processReplacing()*, which sums the sound data from all the active notes (represented by objects of class Voice) and returns this to VST for queuing to the audio output (see Figure 1).

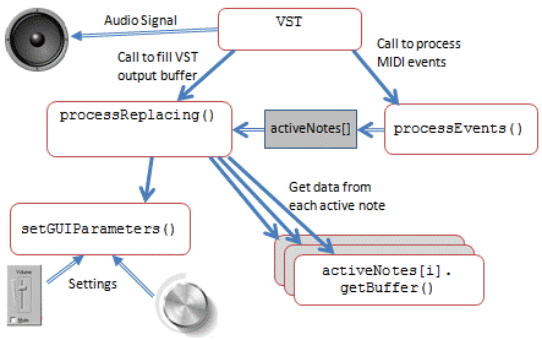

Figure 1
Call Structure for JVSTwRapper

The plugin parameter variables are defined and associated with the controls on the GUI. Parameter values are passed to the sound generation code using an array: *guiParameters*.

This is followed by a declaration of the polyphony (i.e. the number of voices for the VST synthesiser). It is best to limit this to some sensible figure that reflects the number of notes typically in a chord, such as 4, 6 or 8. This means then that as new notes are played older ones should be removed.

As shown in Figure 2, the number of voices is defined as *NUMVOICES*, the array *voicelist* contains the current Voice instances currently being used and *voiceIndex* is set to 0. This will increment as new voices are played and wrap around when the maximum number of voices is reached. Lastly, an array that flags which voices are currently active is defined.

```
public static int NUMVOICES=4;
private Voice[] voicelist;
private int voiceIndex=0;
private static double[] activeNotes
  =new double[NUMVOICES];
```

Figure 2
Parameters for the active Voices

A base class is then defined that establishes the initial values such as the number of inputs and output in addition to values for the sound parameters.

Many of the methods in the *pluginadapter*, such as *getProgram*() and *getParameterName*(), deal with the parameter values input from the GUI (using either the default slider-based simple GUI that is available with the VST host program, or a custom GUI). But the core of the VST is in two methods: *ProcessEvents()* and *ProcessReplacing()*.

ProcessEvents() deals with incoming MIDI events from the host. Most often this is note-on and note-off information, but it can also be MIDI controller information that is associated with particular sound parameters. If a note-on is received, then a new Voice is assigned to the current note, while for a note-off the associated active voice is removed from the *ActiveVoice* array.

In creating a new Voice the following method is used as shown in Figure 3. Firstly, the MIDI note value is converted to a frequency value in hertz. A new instance of the voice at that pitch and sampling rate is generated by *voicelist*[*voiceIndex*] for the current *voiceIndex*. The voice-active flag for this voice is set to true. The *voiceindex*, counting the polyphony is incremented; and if the number of voices exceeds the limit, then this wraps around.

```
private void createNewVoice(int currentNote)
{
  double pitch=midi2hz(currentNote);
  activeNotes[voiceIndex]=pitch;
  voicelist[voiceIndex]
      =new Voice(pitch,SR);
  voicelist[voiceIndex].setActive(true);
  voicelist[voiceIndex].setVoicePitch(
      pitch);
  voiceIndex++;
  if (voiceIndex >= NUMVOICES)
    voiceIndex = 0
}
```

Figure 3
The createNewVoice() method

The method *ProcessReplacing()* is the most important method of all. As in the C/C++ version it takes in two multi-dimensional arrays, inputs and outputs, along with the value *sampleFrames* that is the audio buffer size of the VST host. This buffer size determines the delay between pressing a key/ sending a MIDI note-on message from the user and hearing the sound. This is termed as the *latency*. Smaller buffer sizes result in lower latency, but can give rise to audible crackles in the audio output as the buffers are being emptied more quickly than they are filled (underflow) [8]. A larger buffer size increases latency, which, if being played for real-time output, can be unacceptable if there is a significant time gap between playing the note, and the audio output being produced.

In many example implementations of the *ProcessReplacing()* method, both in C++ and in Java, the sound synthesis code is written directly into the method. While this works, it is poor programming practice and reduces its educational value (by making it hard to separate the interfacing from the DSP algorithm). In this implementation, the method has been rewritten to allow flexible interaction with any sound synthesis algorithm that is implemented in a separate sound generating class named *Voice*, shown in Appendix A.

In this implementation of *ProcessReplacing()*, the synthesizer sound parameter values are gathered into an array. Next, there is a nested iteration, firstly over each Voice, and then for each buffer entry. Then, after ensuring that a Voice is available and active, the buffer is filled with the synthesized data. This is carried out by the *Voice.getBuffer()* method, which adds the synthesised waveform data for that Voice to the output buffer (see Figure 4).

```
public void processReplacing(
    float[][] inputs, float[][] outputs,
    int sampleFrames) {
  float[] out1 = outputs[0];
  float[] out2 = outputs[1];
  setguiParameters();
  int start=0;
  for (int index=0; index<NUMVOICES;
       index++)
    for (int i = start, j=out1.length;
         i < j; i++)
      if ((voicelist[index]!=null)&&
          (voicelist[index].isActive()))
        voicelist[index].getBuffer(
            outputs,i,guiParameters);
}
```

Figure 4
The open-architecture version of ProcessReplacing

The final class declaration in the main VST program gives a unique number to each sound parameter, *Param_ID*, and declares the variable names and types of all the parameters. Values for these variables are extracted from the GUI using a series of getter and setter methods for each one.

The class *Voice*, shown in Appendix A, is where the actual sound generation takes place. The convenience of Java is obvious here as independent

instances of this class (subclasses) can be created for each voice of polyphony. Inside the Voice class, variables associated with each parameter are declared and assigned values from *guiParameter[]*.

New instances of sound elements, such as oscillators, filters, and envelope generators, are made here as well as methods to flag the current status of the note. This is used by the envelope methods to determine where the current envelope stage should be. Once a note is played then the envelope normally goes through attack, decay and sustain, remaining in sustain until the note is off, then the envelope goes into the release mode until it is finished. The flag for the envelope stage works in conjunction with a variable *sampleIndex*, initialised to 0 when the class in instantiated, and acting as a counter as the voice plays, incrementing in tandem with each buffer element.

The *getBuffer*() method uses this sample counter, and the sequence of the sound generation elements produces an audio value for each sample. The final audio value is added to the output. Changing the sound generation technique means changing the *Voice* class and then including whichever new classes are required in the package. Thus, the details of the algorithms for the sound generating elements can be hidden inside each class. Additionally, these classes can be easily re-used with a new plugin.

V Discussion

The jVSTwRapper and the Java classes discussed in this paper work together as abstraction layers that take care of the more complex aspects of VST plugin implementation so that a student can instead focus on the algorithmic aspects of audio programming, creative synthesis, and DSP systems.

Abstraction layers are used throughout various fields of computer science as a means of hiding the details and complexities of data or programs. The aim of the abstraction approach is to reduce the number of concepts down to only those directly relevant to the current perspective or objective.

Typical abstraction layers rely heavily on the *black-box* approach of implementing functions or objects which can take input, perform processing and provide output without the user's full understanding of what is actually involved in the processing. As mentioned in Part II, the basic concept of a plugin already involves this concept, but multiple layers of abstraction can be combined to achieve a particular level of user-accessibility.

One of the main abstraction layers involved in this paper is the jVSTwRapper itself, hiding the implementation details of writing a functional VST Plugin in C++, and allowing the development of plugins through Java; a language often more familiar to undergraduate students and with simplified memory management. In addition to the approachability of Java for undergraduate students, its cross-platform functionality reduces the potential for issues arising from attempts at producing GUIs for plugins that rely on either platform-specific libraries or cross-platform libraries which students may not yet have encountered in sufficient detail [9].

VI Conclusion

The novel open-architecture synthesizer classes discussed in this paper provide a new major abstraction layer for VST plugin implementation by further removing the implementation details of the VST plugin, as illustrated in Figure 5.

This facilitates its use in an educational context as the various parts can be taught as separate Java classes, allowing specific emphasis in the more difficult areas, such as the DSP required for sound synthesis. This allows the system to be addressed as a set of abstracted parts, each be individually accessible for study. The open-architecture version of *ProcessReplacing()* and the *Voice* class are the keys to this abstraction. It is intended that the impact of this abstraction layer on the education of students will be evaluated in greater detail at a later date.

An example of a complete plugin is available at [10]. This new architecture will also be the focal point in a new textbook currently under preparation on digital sound synthesis in Java. For this purpose, a complete set of sound generation algorithms and effects will be implemented, tested, and finally made publicly available.

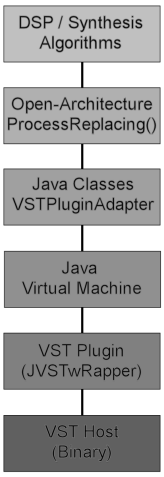

Figure 5
Java VST Abstraction Layers

References

[1] M. Russ, *Sound synthesis and sampling*, 3rd edition, Focal Press, UK, 2008.
[2] The Mathworks, *Matlab version 5.3.1*, Natick, MA, USA, 2012. http://www.mathworks.co.uk/
[3] Steinberg, *VST SDK*, 2012. http://www.steinberg.net/en/company/developer.html
[4] jVSTwRapper: Java based audio plugins, 2012.

http://jvstwrapper.sourceforge.net/
[5] T. Holmes, “Digital synthesis and computer music”, in *Electronic and experimental music: technology, music, and culture*, Taylor & Francis, UK, 2008.
[6] D. Huber, The MIDI manual: a practical guide to MIDI in the project studio, Fical Press, UK, 2008,
[7] H. Seib, *VSThost-a VST-compatible host*, 2012 http://www.hermannseib.com/english/vsthost.htm
[8] M. Walker, ‘Using VST instruments’, *Sound on Sound*, Dec. 2000.
[9] Y. Liang, *Introduction to Java programming – comprehensive version*, 7th edition, Prentice Hall, US, 2008.
[10] Java VST plugin example http://www.cs.nuim.ie/~jtimoney/ISSC2012paper/ISSC2012plugin.html
[11] V. Välimäki and A. Huovilainen, “Oscillator and Filter Algorithms for Virtual Analog Synthesis,” *Computer Music Journal*, Vol. 30, no. 2, 2006.
[12] J. Lane, D. Hoory, E. Marinez and P. Wang., “Modelling analog synthesis with DSPs,” *Computer Music Journal*, Vol. 21, no. 4, 1997.

APPENDIX A

This appendix provides sample code for an implementation of the Voice.java class. The implementation here has one oscillator based on the Differential parabolic wave method and one amplifier envelope (see [11] and [12] respectively for algorithms). These are generated by the DPWosc and ADSR classes respectively.

```
public class Voice {
  private DPWosc osc; //oscillator
  private ADSR ampEnv; //envelope
  private int sampleIndex; //sample counter
  private boolean active; //envelope flag
  private double pitch; //note pitch
  private double SR; //sampling rate in Hz
  private int status; //note flag
  private double OutputVolume; //parameter
  private double OscWaveform; //wave type
    //given by numerical value
  private String waveform; //wave type
  private double AmpA, AmpD,AmpS, AmpR;
    //parameters for ADSR values

  public Voice(double freqHz,
               float sampleRate) {
    //initial values
    pitch=freqHz;
    SR=sampleRate;
    sampleIndex=0;

    osc1=new DPWosc(); //new instance
    ampEnv=new ADSR();//new instance
    osc.init(); //initialize method
    ampEnv.init();//initialize method
    status=0;
  }
  //check if envelope active
  public boolean isActive(){
    return active;
  }
  //set envelope to active
  public void setActive(boolean activeOn){
    active=activeOn;
  }
public double getVoicePitch(){return pitch;}

  public void setVoicePitch(double noteHz){
    pitch=noteHz;
  }
  //set status flag depending on note on or
  //off and envelope stage
  public void setVoiceStatus(boolean
                             noteStatus) {
    if (noteStatus)
      status=0;
    else if (!noteStatus &&
          !(ampEnv.isFinished())) {
      ampEnv.setReleaseCounter(AmpR, SR);
      status=1;
    }
    else
      status=2;
  }
 public int getVoiceStatus(){return status;}
  //compute the output audio
  public void getBuffer(
      float[][] outputs,int i,
      double[] guiParameters) {
    float[] out1 = outputs[0];
    float[] out2 = outputs[1];
    double oscil;
    double env;
    double output;
    setGuiParameters(guiParameters); //get
    // parameter values from VST GUI
    getOscWave(); //get waveform type
    //generate oscillator
    oscil=osc.GenOsc(pitch, SR,
                     waveform1,sampleIndex);
    //generate envelope
    env=ampEnv.envGenNew(AmpA, AmpD, AmpS,
      AmpR,sampleIndex,getVoiceStatus(),SR);
    //calculate output
    output=OutputVolume*env*oscil
    //stereo output from VST
    out1[i] +=(float) output;
    out2[i] +=(float) output;
    //turn off envelope if complete
    if (ampEnv.isFinished())
      setActive(false);
    //increment the sampleIndex
    sampleIndex++;
  }
  //set parameter values from VST GUI to
  //local variables
  public void setGuiParameters(
            double[] guiParameters ) {
    OutputVolume=guiParameters[0];
    Osc1Waveform=guiParameters[1];
    AmpA=guiParameters[2];
    AmpD=guiParameters[3];
    AmpS=guiParameters[4];
    AmpR=guiParameters[5];
  }
  //convert string describing oscillator
  //waveform to a numerical value
  private void getOscWave(){
    waveform = “triangle”;
    if (OscWaveform < 0.33)
      waveform1 = "sawtooth";
    else if ((OscWaveform>=0.33) &&
             (OscWaveform<=0.67))
      waveform="square";
  }
}
```