JASS: A JAVA AUDIO SYNTHESIS SYSTEM FOR PROGRAMMERS

Kees van den Doel and Dinesh K. Pai

Department of Computer Science
University of British Columbia
Vancouver, Canada
{kvdoel, pai}@cs.ubc.ca

ABSTRACT

We describe a unit generator based audio synthesis programming environment written in pure Java. The environment is based on a foundation structure consisting of a small number of Java interfaces and abstract classes, and a potentially unlimited number of unit generators, which are created by extending the abstract classes and implementing a single method. Filter-graphs, sometimes called "patches", are created by linking together unit generators in arbitrary complex graph structures. Patches can be rendered in real-time with special unit generators that communicate with the audio hardware, which we have implemented using the JavaSound API.

1. INTRODUCTION

Several software applications for digital audio synthesis are presently available. These applications have varying degrees of user extensibility and customizability. They also differ in price from free to very expensive, and may require specialized hardware or a specific operating system. The target application of these systems varies too, but all systems that we are aware of are primarily focussed on the synthesis of music.

In our current research [1, 2, 3, 4, 5, 6, 7] we are investigating models of audio-synthesis suitable for sound-effects, sometimes called "Foley sounds", in interactive environments with real-time user interactions such as computer games, simulations, and immersive environments. All the features we wanted for an audio synthesis environment for these applications could not be found in any single existing environment, and we therefore developed an environment specifically for these kind of sounds, which we have called "JASS" which stands for "Java Audio Synthesis System", but hopefully not for "Just Another Software Synth".

The features of JASS, besides the obvious one of being capable of implementing arbitrary synthesis algorithms are:

- Platform independence; obtained by using pure Java.
- Ease of deployment in web documents.
- Simplicity; obtained by omitting support for musically oriented features such as envelopes, MIDI, etc.
- Extensibility; obtained through careful object oriented design.
- Run-time control through asynchronous method calls.
- Dynamic creation of "patches" at run-time without audio breakup.
- Efficiency; achieved by vectorizing all processing elements.
- Real-time synthesis.
- Free; which we achieve by writing it ourselves and giving it away.
- Low latency; obtained by using small buffers.

The JASS toolkit which is available for download from our website [8] consists of several software layers, organized in Java packages:

The `engine` package provides Java interfaces and abstract classes which can be extended to create unit generators (UG's). There is no strict distinction between a patch and a UG, and we shall just reserve the name "patch" for a UG which contains other UG's. Whenever the distinction is important we shall call UG's that do not contain other UG's "atomic". UG's are connected into filter-graphs, or "patches", which are also used in computer music [9]. These filter graphs are also equivalent to the "timbre trees" introduced by Takala and Hahn [10]. The fundamental interfaces are `Source` and `Sink` which encapsulate the notion of interconnected filter elements. This is a common design, also used for example in the Java Media Framework, which is intended for more general applications dealing with different media types and is quite complex. The abstract classes `Out`, `In`, and `InOut` implement respectively `Source`, `Sink`, and both. These abstract classes implement all the plumbing code necessary for UG's to communicate and be interconnected into graph structures and leave just a single method, `computeBuffer()` unimplemented. This method defines the actual audio processing to be done in the UG. The UG's provide only audio-buffers, and have no inherent rendering capability. The actual rendering is done with the classes from the `render` package, but could be implemented independently if so desired.

The `generator` package contains instantiable classes which extend the abstract classes in the `engine` package. These classes are the basic UG's. We have implemented basic audio processing blocks such as wave-tables, filters, audio file readers, resonance banks, pitch-shifters, and others as needed. They are very easy to author.

We provide a `render` package which contains a `SoundPlayer` UG to render a patch to the audio hardware through JavaSound, low level utility classes for converting between different audio data formats, and an off-line renderer which produces audio files. A `Controller` class is provided which allows the creation of simple graphical user interfaces with sliders and buttons to experiment with algorithms in real-time.

To show how easy it is to extend the abstract classes on-the-fly, here is some code to generate a sawtooth signal with a frequency of 415 Hz (perhaps useful as a virtual tuning fork) and send it to the audio hardware in real-time:

```
float srate = 44100;
float freq = 415;
int bufferSize = (int)(srate/freq);
new SourcePlayer(bufferSize,0,srate,
  new Out(bufferSize) {
    public void computeBuffer() {
      for(int i=0;i<getBufferSize();i++)
        buf[i] = i;
    }
  }
).start();
```

An unnamed `SourcePlayer` UG is created. Its last argument, which defines which `Source` to play, is an anonymous extension of an `Out` class, which is an abstract class from the `engine` package. The anonymous derived class implements the `computeBuffer()` function which fills each buffer with a period of a sawtooth. The `start()` call on the `SourcePlayer` starts a thread which pulls audio-buffers out of the `Out` UG and sends them to the audio hardware using JavaSound.

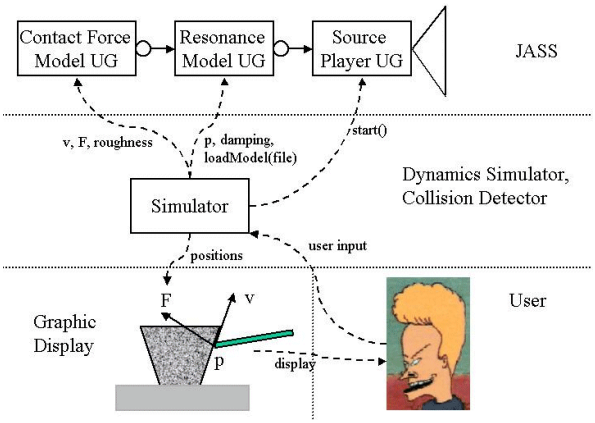

Figure 1: Example of a usage of JASS in a real-time simulation environment. The simulator make asynchronous calls on the UG's, which are not necessarily atomic, while a JASS synthesis thread renders audio.

In Fig. 1 we have indicated how an algorithm authored in JASS can be deployed in a simulation environment with real-time sound. We show a virtual pot which can be touched by a probe under user control. The simulator keeps track of position and velocities and performs collision detection and dynamics simulation. It interacts with JASS by starting the synthesis thread on the `SourcePlayer`, and then makes asynchronous calls to the UG's that implement the specific algorithms used for audio modeling of the pot and the contact. These calls occur at simulation time steps, and are made in a separate thread. At the same time, the simulator communicates to the graphics display which ensures the synchronization of audio and graphics.

JASS has similarities to Perry Cook's C++ Synthesis Toolkit [11], the main differences are that (1) JASS uses vectorized UG's which make it much more efficient, (2) JASS has no support for event control like MIDI and SKINI, (3) JASS is written in pure Java.

Another related synthesis package is Burk's JSyn [12] environment, which is a UG based Java API using native methods implemented in C, which is available for Macintosh and Windows. It is a commercial product, targeted primarily towards musical applications. No source code is available for the native C implementations, and the system is not user extendible. It includes a sophisticated event scheduler and can be deployed on the web via a browser plug-in. Because JASS is written in pure Java, synthesis algorithms authored with it can be deployed in a web page without any special plug-in on JavaSound enabled browsers such as Netscape 6.

SynthBuilder [13] is a UG based synthesis environment developed at CCRMA and consists of a scripting language describing patches, a real-time synthesis engine and a sophisticated graphical interface to design patches.

JMax [14] is a UG based synthesis environment developed at IRCAM and consists of a graphical patch design environment which is easy to use for non-programmers. The system can be extended by writing custom UG's in C++. THe system runs on Linux only.

CSound [15] provides a sophisticated synthesis language for musical instrument synthesis and has a large community of users. It is however not easy to use programmatically.

Our notation for Java interfaces, classes and inheritance relations is illustrated in Fig. 2.

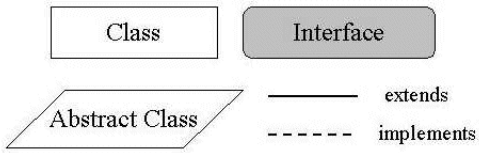

Figure 2: Notation for Java objects. In class inheritance relations the convention is that the object on the right extends the object on the left.

2. FOUNDATION

The `engine` package encapsulates the notion of interconnected unit generators. A UG in JASS is a processing element which can receive audio inputs and has at most one audio output.

Interfaces for Unit Generators

The `Source` interface, depicted in Fig. 3, encapsulates the notion

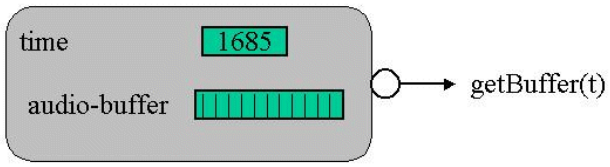

Figure 3: Source interface with temporal state which maintains an audio buffer. Methods to set and get the time and buffer are not indicated here.

of a processing element with a temporal state, which maintains an audio-buffer, which we consider to be an array of `float`. It is defined as follows:

```
public interface Source {
  float[] getBuffer(long t)
        throws BufferNotAvailableException;
  long getTime();
  void setTime(long t);
  int getBufferSize();
  void setBufferSize(int bufferSize);
  void clearBuffer();
}
```

The most important method is `getBuffer(long t)` (indicated by a circle), which requests an audio-buffer at a specific time. Depending on the implementation and the relation of `t` to the time state such a request may or may not be granted. The exception thrown when the request fails is specific for the `engine` package.

The `getBuffer()` method is intended to be called by objects implementing the `Sink` interface, depicted in Fig. 4. The methods defined in the `Sink` interface encapsulate the behavior of an object which is connected to a number of `Sources`. It is defined as follows:

```
public interface Sink {
  Object addSource(Source s)
        throws SinkIsFullException;
  void removeSource(Source s);
  Source [] getSources();
}
```

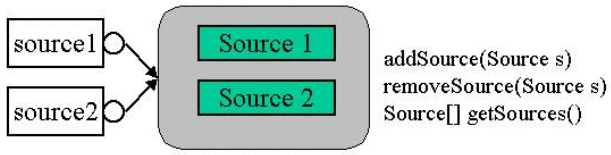

Figure 4: Sink interface encapsulating an object which contains sources, which are indicated by incoming arrows.

The method `addSource()` connects a `Source` to the `Sink`. An `Object` is returned which is intended to be used by the caller to interact with the connection. For example, if the `Source` is a contact force on a three dimensional solid body, which is the `Sink`, the `Object` returned could contain methods to set the spatial location of the contact. If the `Source` can not be added, for example because the implementor only supports one input as is the case for a filter UG, an exception is thrown.

The method `getSources()` returns an array of `Sources`. The intention is that the implementing UG will call each `Source` in order and request audio buffers from it. The ordering of the `Sources` is important in general. A `Source` can be connected to an arbitrary number of `Sinks`.

Abstract Classes for Unit Generators

Three abstract classes are defined which implement some or all of the interfaces defined above. They provide templates for UG's which produce audio buffers, consume it, or both. We made some specific implementation choices, and it is possible to author different UG template implementations, and use them within existing applications as long as they implement the `Source` and/or `Sink` interfaces.

These abstract UG's are called `Out`, `In`, and `InOut`. The class hierarchy is depicted in Fig. 5.

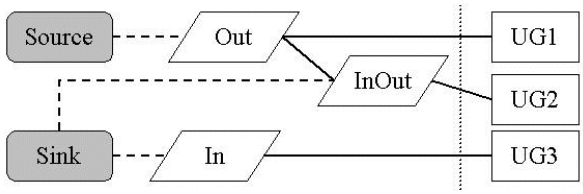

Figure 5: Class hierarchy of the engine package. Classes to the right of the vertical dotted line represent instantiable classes from other packages extending the abstract classes.

The `Out` class, which implements `Source`, represents a UG which produces audio buffers, such as a wav file, a noise generator, or a sine wave generator, etc. It contains member variables to set the time and contains an audio-buffer of `float`, see Fig. 6. Time

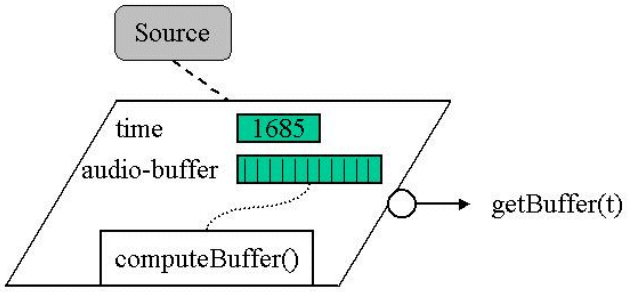

Figure 6: Abstract class Out, representing an object capable of producing audio buffers. The output in represented by the circle. It leaves a single method, `computeBuffer()`, unimplemented.

is defined as an integer which counts the number of buffers of size `bufferSize` that are processed. At a sampling rate of f_S and a buffersize of N this corresponds to time slices of size N/f_S. The most important `Source` member function implemented by `Out` is `getBuffer`, which is implemented as follows:

```
public synchronized float[] getBuffer(long t)
        throws BufferNotAvailableException {
  if(t == getTime()+1) {
    setTime(t);
    computeBuffer();
  } else if(t != getTime()) {
    throw new BufferNotAvailableException();
  }
  return buf;
}
```

It makes use of the member variable

```
protected float[] buf;
```

which is the audio-buffer at the current time. If the time `t` equals the current time, the presently held audio-buffer is returned. Alternatively, if the requested buffer lies one time-step in the future, the UG will increment time, compute the next buffer by calling `computeBuffer()`, and return it. The method `computeBuffer()` must be implemented by instantiable classes extending `Out`.

The `InOut` class represents the most general UG which has audio inputs and a single output. It implements both interfaces by extending `Out` and providing an implementation of the methods in the `Sink` interface. (Because Java does not support multiple inheritance, `InOut` can not derive from both `In` and `Out`.) The methods `addSource()` and `removeSource()` are synchronized. It maintains a list of input `Sources`, which have an associated audio-buffer cache as indicated in Fig. 7. The method `getBuffer()` inherited from `Out` is overridden by inserting a call to `callSources()` between `setTime` and `computeBuffer`:

```
...
setTime(t);
callSources();
computeBuffer();
..
```

This method calls `getBuffer(currentTime)` on every attached `Source` in order, and caches the results in the associated buffers as indicated in Fig. 7. If the filter-graph contains a

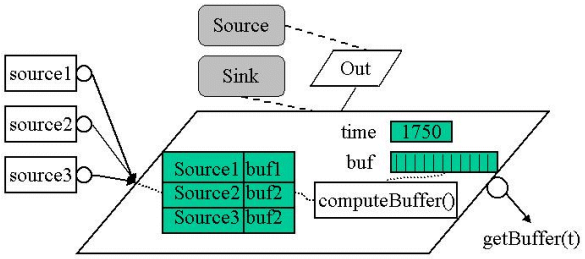

Figure 7: Abstract class InOut, representing an object capable of getting and producing audio buffers. The unimplemented method `computeBuffer()` uses the cached input buffers to compute the output buffer.

closed loop it is possible that this eventually results in another (recursive) call to `getBuffer()` on the same `InOut` object. Because time has been incremented before buffers are requested from the external `Sources`, such a call will come with a time argument equal to the current time and will therefore just return the cached buffer, which is now actually "stale", thereby preventing any further recursion. This means that a closed loop in a patch acts as a delay line, with the delay equal to the buffer-size. Because a connected graph can never have UG's out of synch by more than a single time-step (at least in this implementation of the `Source` and `Sink` interfaces), the throwing of the exception `BufferNotAvailableException` indicates a programming error.

When implementing `computeBuffer()`, it is important to realize that `getBuffer()` returns a reference to the audio-buffer for efficiency, and does not copy it. If closed loops occur one of the cached `Source` buffers may in fact be the same buffer as the one that is being computed, which may cause unexpected bugs when this is not realized.

The `In` class, depicted in Fig. 8, provides an implementation of the `Sink` interface and is derived from the Java class `Thread`. It is marked abstract so it can not be instantiated, though it does not in fact contain any unimplemented methods. This class will be

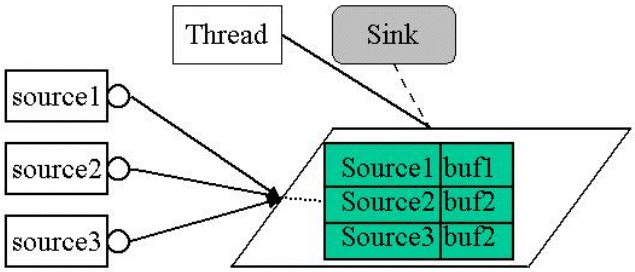

Figure 8: Abstract class In, representing an object capable of getting audio buffers. It extends the Java class `Thread` to enable the creation of synthesis threads which "pull" audio-buffers out of `Sources`.

subclassed by UG's for rendering audio produced by `Sources` for example.

JASS patches are usually deployed in a multi-threaded environment. For example, in the application depicted in Fig. 1, the simulator thread will make calls to the UG's, while the audio render thread runs simultaneously. For this reason `getBuffer()` is implemented as a `synchronized` method. In order to avoid race conditions any UG methods that change the state of the UG (for example, a filter UG may have the filter coefficients changed) should be declared `synchronized`. This ensures that the state of the UG does not change while the UG is processing a call from `getBuffer()`, which would result in unpredictable behavior. Similarly, the methods `addSource()` and `removeSource()` are also declared `synchronized`, to allow a patch to be "rewired" from a different thread, without disturbing audio processing done as a result of `getBuffer()`.

3. CONCLUSIONS

The current distribution of the JASS system contains a set of unit generators for reading and playing audio files, at varying speeds and volumes, mixers, various filters, and UG's for rendering and capturing audio. On the JASS website [8] the full documentation of all implemented JASS UG's can be found and read online. The set is not intended to be comprehensive and users of the system are expected to write their own UG's.

The website also contains an extensive set of demos such as a Karplus-Strong plucked string algorithm, a reverberation algorithm, a granular synthesis patch, and many others. These demos can be heard online in a Java 2 enabled browser (for example Netscape 6).

The algorithms authored with JASS have an inherent latency determined by the buffer-size used by the filter-graphs, which can be as low as one sample in principle. Smaller buffers require more processing overhead, though. For our applications we have found that using a buffer-size of 1 sample results in a slowdown by a factor 4, compared to using a buffer-size of 100. At a sampling rate of 44100Hz, a buffer-size of 100 translates in a latency of about 2 ms, which is excellent for most purposes.

Unfortunately current JavaSound implementations require large buffers for real-time synthesis on all platforms we are aware of. On Windows 98 and Linux we found that the lowest latency we could achieve without breakup of sound was 140ms, which is

three times higher than the streaming latency of the DirectSound API. We anticipate this situation will improve when JavaSound matures.

The JASS system has been used primarily for our research in contact sound generation, but we hope that it will find more widespread usage.

4. REFERENCES

[1] K. van den Doel and D. K. Pai, "Synthesis of Shape Dependent Sounds with Physical Modeling," in *Proceedings of the International Conference on Auditory Displays 1996, Palo Alto*, 1996.

[2] K. van den Doel and D. K. Pai, "The sounds of physical shapes," *Presence*, vol. 7, no. 4, pp. 382–395, 1998.

[3] K. van den Doel, *Sound Synthesis for Virtual Reality and Computer Games*, Ph.D. thesis, University of British Columbia, 1998.

[4] Roberta L. Klatzky, Dinesh K. Pai, and Eric P. Krotkov, "Perception of material from contact sounds," *Presence*, vol. 9, no. 4, pp. 399–410, 2000.

[5] D. K. Pai and J. L. Richmond, "Robotic measurement and modeling of contact sounds," in *Proceedings of the International Conference on Auditory Display 2000, Atlanta*, 2000.

[6] D. K. Pai, K. van den Doel, D. L. James, J. Lang, J. E. Lloyd, J. L. Richmond, and S. H. Yau, "Scanning physical interaction behavior of 3D objects," in *Computer Graphics (ACM SIGGRAPH 01 Conference Proceedings)*, 2001.

[7] Kees van den Doel, Paul G. Kry, and Dinesh K. Pai, "FoleyAutomatic: Physically-based Sound Effects for Interactive Simulation and Animation," in *Computer Graphics (ACM SIGGRAPH 01 Conference Proceedings)*, 2001.

[8] "http://www.cs.ubc.ca/~kvdoel/jass," 2001.

[9] M. V. Mathews, *The Technology of Computer Music*, MIT Press, Cambridge, 1969.

[10] T. Takala and J. Hahn, "Sound rendering," *Proc. SIGGRAPH 92, ACM Computer Graphics*, vol. 26, no. 2, pp. 211–220, 1992.

[11] Perry R. Cook, "Synthesis Toolkit in C++," in *SIGGRAPH*, 1996.

[12] P. Burk, "JSyn: Real-time Synthesis API for Java," in *Proceedings of the International Computer Music Conference*, San Francisco, 1998.

[13] N. Porcaro, P. Scandalis, J. O. Smith, D. A. Jaffe, and T. Stilson, "SynthBuilder–a graphical real-time synthesis, processing and performance system," in *Proceedings of the International Computer Music Conference*, Banff, 1995, pp. 61–62.

[14] François Déchelle, Norbert Schnell, and Riccardo Borghesi, "The JMax Environment: An Overview of New Features," in *Proceedings of the International Computer Music Conference*, Berlin, 2001.

[15] B. L. Vercoe, "Extended Csound," in *Proceedings of the International Computer Music Conference*, Hong Kong, 1996.