Easily Adding Sound Output to Interfaces

Brad A. Myers and **Kenneth A. Strickland**

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
bam@cs.cmu.edu
http://www.cs.cmu.edu/~bam

ABSTRACT

Adding sound output to interfaces is a very difficult task with today's toolkits, even though there are many situations in which it would be useful and effective. We have designed an architecture that makes it very easy to add sound output to an interface. Any interaction behavior, animation, or command can be augmented with sounds to occur at the beginning or end, or for the duration. Parameters of the sound, such as the speed or volume can be easily tied to properties of the data using constraints. Two different sound objects are currently supplied: one for playing recorded sounds, and the other for text-to-speech. The text-to-speech sound object can be used to quickly build various kinds of screen readers. Easy-to-use mechanisms give the programmer complete control over interrupting and pre-empting when multiple sounds are played at the same time. Because sound output can be added to an existing application with as little as a single extra line of code, we expect that this new mechanism will make it easy for researchers and developers to investigate the use of sound in a wide variety of applications.

Keywords: sound, auditory output, earcons, text-to-speech, toolkits, multimodal interfaces, Amulet, Andalusite.

INTRODUCTION

In the introduction to the 1989 special issue of *Human-Computer Interaction* on non-speech audio [5], Buxton claims that many of the logistical problems in exploring the use of audio signals have been overcome. Indeed, today's PC's and Macintoshes contain sophisticated support for sound to support multimedia games and the world-wide-web. However, one significant barrier still remains: the programming interface to sound output is still quite awkward and low-level, and requires the programmer to deal with issues of timing, buffering, and multi-processing. In particular, existing toolkits do not provide any support for integrating audio output with direct manipulation user interfaces, and there has been little research on architectural mechanisms which would make this integration easier.

We have developed a system that provides high-level support to make it very easy to integrate sounds into direct manipulation user interfaces. Sounds can be easily synchronized with graphical and user actions, such as animations, clicking the mouse buttons, and dragging objects. The timing of the sounds, the temporal scheduling of sounds to control overlapping, interrupting and sequencing, and the coordination of foreground and background sounds are all easily handled, often with a single line of code.

There are many potential benefits for using sound in interfaces [2, 5, 7]. Expert game players do better when sound is turned on, showing that it provides strategically critical information [5]. Sound provides an extra channel that can be used to help present complex information [3]. It provides complementary information to the graphics to provide feedback for what you are doing, for notification, for beeps on errors, or for awareness of what the system and other people are doing. Sound also enables monitoring of background processes while performing a foreground task. For example, an experiment showed that when sound is used in addition to graphics to show the mode of a palette, users make fewer errors [4]. In multi-user applications, sound provides very helpful feedback about what the other users are doing, and can help with awareness [8]. We rely on sound for information in our everyday lives [7]. Even with computers, the whine of the disk drive tells us whether

they are operating as expected. As a final motivation, sound is crucial for people with visual difficulties [6].

Our new sound system is called Andalusite, which is a kind of yellow-green gemstone. Andalusite stands for **A**mulet's **N**ew **D**evelopment **A**ugments the **L**ook-and-feel **U**sing **S**ounds **I**ncluding **T**ext-to-speech and **E**ffects. Important contributions of Andalusite include:

- A design for a high-level, extensible object-oriented interface to sounds, where properties of the sounds can be computed using constraints.
- A design for a machine-independent specification of which sounds should be interrupted and pre-empted by any newly played sound, that provides machine-independence by playing something reasonable no matter how many channels of sound are supported by the hardware.
- An architecture for connecting sounds to interactive behaviors and animations, that makes it particularly easy to augment actions of a direct manipulation user interface with sounds.

Amulet [13] is a C++ toolkit that runs on X/11, Windows 95, Windows NT, and the Macintosh. One of the important goals of Amulet is to enable sophisticated features to be provided to end users without requiring much coding by designers. By providing better modularity for the software for the user interface, Amulet achieves increased reuse and decreased code size, and makes it easier for researchers and developers to create applications. For example, in Amulet, the interactive behavior of objects can be defined completely independently from their graphical look by attaching "*Interactor*" objects to the graphics. *Command objects* [12] encapsulate the complete information about operations and support undo. Animations for objects can be added with a single line of code by attaching an *animation constraint* [14] to various properties. The animation constraint detects changes to the value of the slot to which it is attached, and causes the slot to instead take on a series of values interpolated between the original and new values. We have followed this philosophy in our new support for *sound output*. The goal is to make simple sound output *extremely* easy for a programmer to add to an interface. This is achieved by allowing the sounds to be defined independently from the actions that start and stop the sound, and to provide the high-level mechanisms that synchronize sounds with other interface actions.

The Andalusite research is specifically directed at augmenting graphical user interfaces with sound output. We are not addressing sound *input* such as speech recognition, although that would be an interesting addition to the Amulet repertoire. This paper presents an overview of the related work, and then the low-level and high level interfaces to the Andalusite sound system, along with some example applications.

RELATED WORK

Most research work on auditory user interfaces has concentrated on investigating new uses of sound and on how sounds can be constructed from components. There has been very little prior work on how to integrate sounds with other modalities.

Conversational VoiceNotes [15] addressed sound-only user interfaces, such as for telephones. It uses a context-free grammar for specifying the output which matches the grammar needed for parsing the speech input. This was needed to handle the complex auditory messages composed of generated speech and recorded sounds. Grammars do not match well with direct manipulation user interfaces, however.

Mercator [6] aims to allow blind users to work with graphical interfaces. The first version identified problems with X/11 and the Xt toolkit for integrating sound, which were fixed in later versions of X. An important goal of Andalusite is to make Mercator-like interfaces significantly easier to build.

ENO [1] concentrated on describing the various acoustic properties of sounds, to make it easier to generate them, as opposed to our system which concentrates on making it easy to *integrate* the sounds with the rest of the interface, and to control the timing and coordination of sounds with animations and interactions. A sound object with ENO's features would be an excellent addition to our system.

The Earcons [2] work first investigated the different parameters of sounds that can be meaningfully interpreted by people, and then applied these to create different sounds that represented different features of an interface. Using Earcons in palettes was shown to reduce users' errors [4]. Andalusite aims to make interfaces using Earcons easier to build.

ScriptX [10] is a programming language designed specifically for multimedia. It has built-in primitives like "clocks" for controlling animations and for playing sounds. However, to coordinate animations and sounds together requires the programmer to construct a complex hierarchy of clock objects, and to write a set of callback methods for the clocks. The Andalusite mechanism requires much less work from the programmers.

Macromedia's Director has sophisticated support for sound, both in its interactive score editor and in the Lingo language. There are features for starting, stopping and looping sounds, triggered by various events. However, the designer must still deal with how to synchronize and sequence sounds, and there is no built-in support for interrupting, pre-empting or computing parameters of sounds based on the properties of other objects.

OVERVIEW

The Andalusite architecture supports any kind of sound output: speech output, either recorded or text-to-speech, music, and non-speech audio cues: beeps, buzzes, and other sound effects. Since it is built into the Amulet toolkit, the interface to programmers is machine-independent and runs on X/11, Windows and Macintosh.[1] Code written using auditory output can be written once and will run on multiple platforms.

Currently there are two kinds of sounds supported: playing of recorded sounds, and text-to-speech. The playing of recorded sounds can be modified by speed and loudness. The text-to-speech uses a platform specific library, such as PlainTalk on the Macintosh, and is parameterized by what string to say, as well as by the loudness, voice and other parameters. In the future, we plan to add more sophisticated control for generating sounds dynamically, similar to the techniques provided by ENO [1].

Multiple channels of audio are supported, up to the limit available on the hardware. The Macintosh, though theoretically able to handle an arbitrary number of sound channels, tends to make the best use of CPU load and memory with four channels or less, so Andalusite defaults to four channels of stereo support on Macintosh. Similar considerations apply to Windows machines using DirectX 5.0. Since Unix machines generally have no hardware support for sound, we expect them only to have one channel.

When a new sound is requested, it normally plays in addition to other sounds already playing. To prevent this, the programmer can explicitly stop specific sounds or all sounds, or else specify that the new sound should *interrupt* a previous sound. When there are no more tracks available and a new sound is started, a pre-empting scheme is used to decide which sound should stop playing. Normally, the pre-empted sound is resumed when the newer sound is finished. The following sections describe the interface to sounds in more detail.

PROGRAMMER INTERFACE

There are two interfaces to the Andalusite sound system, as shown in Figure 1. The low level interface hides the details of the machine-specific sound system and provides basic capabilities to load and play sounds. It also provides a sophisticated system for dealing with interrupting and pre-empting sounds, to make it easier for programmers to deal with scheduling. The high-level interface supports synchronizing sounds with behaviors and animations.

[1] At the time of this writing, only the Macintosh implementation of Andalusite is fully debugged. The Windows version is almost complete, and the Unix version has not been started yet.

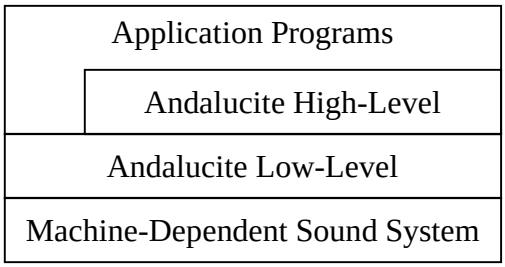

Figure 1. The Andalusite architecture for supporting sound.

Low-Level Interface

Sound Objects

The low-level part of the Andalusite interface provides Sound Objects which contain a number of parameters. Amulet uses a prototype-instance object system [13] in which parameters of objects are represented as instance variables which can be local or inherited. The advantage to the programmer is that any parameters that are not relevant or needed can be easily left at their default values. Therefore, although many aspects of the sounds can be controlled, the simplest (and most typical) interface is to ignore most or all of the parameters.

The `Am_Load_Sound` routine takes a filename describing a sound file and returns a sound object. Another version of `Am_Load_Sound` takes a resource ID, and is specifically for the Macintosh where sounds are stored in the resource fork of an application. A similar function, `Am_Load_Text_To_Speech_ Sound` returns a sound object which will take an ASCII string and read it aloud.

Playing Sounds

Once the programmer has a sound object, it can be played explicitly using `Am_Play_Sound`. Other routines are `Am_Stop_Sound` to stop playing a particular sound, and `Am_Stop_All_Sounds`. The simplest way to play a sound is therefore just to add the single line of code to the program:

```
Am_Play_Sound(Am_Load_Sound("sndfile.wav"));
```

The `Am_Play_Sound` routine returns immediately after starting the sound. `Am_Play_Sound_And_Wait` will wait for the sound to finish, but is rarely appropriate. Usually, it is better for the rest of the interface to not be delayed by the sound playing.

Basic Parameters of Sounds

Most objects in Amulet are controlled primarily through their slots, without the use of methods. Similarly, the sound object contains a variety of slots which control its behavior. This allows a *declarative specification* of the behavior of the sound objects, without the need to write new methods. The specific slots of a sound object depend on its type. Figure 2 summarizes the slots of sound objects.

Slot	Default Value	Comments
`Volume`	1.0	
`Balance`	0.0	Negative numbers cause it to be louder on the left.
`Speed`	1.0	Controls speed of playback.
`Repeat_Count`	1	Number of times to play sound
`Next_Sound`	null	Link for sequences of sounds
`Interrupt_List`	(self)	Will interrupt (stop) any sounds on this list if they are playing when this sound starts
`Pre_empt_If_Needed_List`	(self)	Will pre-empt any sounds on this list if there are no free channels.
`Currently_Playing`	false	Read-only; `true` when this sound is playing
`Text`	`""`	String for Text_To_Speech sound to read
`Voice`	1	Which voice style to use for Text_To_Speech sound

Figure 2: The slots of sound objects. Most programmers will not need to set these parameters, and can just use the default values.

All sound objects have a `Volume` slot, as well as a `Balance` slot to control stereo playback. `Balance` ranges from –1.0 to 1.0 with 0.0 being the default, centered sound. The `Speed` slot controls the playback speed for recorded sounds.

The `Currently_Playing` slot is a read-only slot which can be queried to see if the sound is still playing. This slot is seldom needed by programmers since the scheduling mechanisms are usually sufficient.

The `Text_To_Speech` sound object adds an extra slot called `Text` for the string to read. Normally this slot is computed using a constraint, as described below. The `Voice` slot can be used to control which of a small set of pre-defined voices the speech is read with.

Sequencing

To control the sequence of sounds, each sound can have a `Repeat_Count` to determine how many times the sound plays. The default is one time. A special "`Infinity`" value means to play repeatedly until explicitly stopped.

A sound object can be linked to another sound object using the `Next_Sound` slot, and the next sound will play when this sound is finished. If the sound has a `Repeat_Count`, then the next sound is played after all the repetitions. The `Am_Stop_Sound` command has an extra optional parameter that controls whether the next sound should be played when a sound is stopped. This is useful for sounds which have a repeat count of `Infinity` to control whether to just stop (the default) or whether to go on to the sound in the `Next_Sound` slot.

We plan to have more sophisticated mechanisms for complex sequences, as described below in the Future Work section.

Computing Parameters with Constraints

In Amulet, any slot of an object instead of containing a regular value like an integer or a string, can contain a *constraint*, which is an expression that calculates the value [13]. Constraints are automatically re-evaluated whenever anything changes that the expression depends on. Constraints can be arbitrary C++ code, and a large library of pre-defined constraints is available. This makes it very easy to have the values of objects depend on aspects of the application's data.

As an example of how constraints are very useful for sound objects, here is a constraint to compute the pitch of a sound (which would be controlled by the `Speed` slot) based on the size of an object, as described in SonicFinder [7]:

```
// define a constraint to get the speed from the size
float speed_from_size_constraint(self) {
    Am_Object ref_obj = Get_Obj_Over(self);
    int size = ref_obj.Get(Size);
    if (size > 1000) return 0.5; //slower
    else if (size > 500) return 1.0;
    else return 2.0; //faster
}
// put this constraint into the Speed slot
my_sound.Set(Speed, speed_from_size_constraint);
```

Whenever a different object is selected (so `Get_Obj_Over` returns a different value), or if the object changes `Size`, then the constraint will be re-evaluated, and the `Speed` of the sound will change.

Amulet introduced the idea of an *animation constraint* [14] which detects changes to the value of the slot to which it is attached. When the value is set, the animation constraint restores the original value, and causes the slot to take on a series of values interpolated between the original and new values. Animation constraints were created to animate the position and color slots of graphical objects, but because they are a general mechanism, they can be used for any slot of any type. For example, an animation constraint can be put into the `Volume` slot of a sound to cause the sound to fade in and out. If the volume was 1 and the slot was set with 0, then the animation constraint would cause it to fade from 1 to 0 over a period of time picked by the programmer:

```
my_sound.Set(Volume, 1.0);
my_sound.Set(Repeat_Count, Infinity);
anim = Animator.Create();
anim.Set(Duration, 500); // milleseconds
my_sound.Set(Volume, anim);
Am_Play_Sound(my_sound); //starts playing
// instead of jumping to 0, will fade from 1 to 0 over ½ second
my_sound.Set(Volume, 0.0);
```

Constraints could also be used in the `Repeat_Count` and `Next_Sound` slots to compute a sequence of sounds.

Interrupting

The Macintosh and Windows platforms have hardware that supports playing multiple sounds at the same time. When a new sound starts playing, sometimes the programmer might want to make sure that specific other sounds immediately stop. For example, if a screen reader sound object is reading a string from the screen, and the user moves the cursor to a different string, the reader should be interrupted and start reading the new string instead of the old one. In other situations, the new sound should play along with the old sound. For example, if the old sound is a background song and the new sound is a foreground explosion, you would want them both to be playing, if possible.

By default, if a sound is re-started while it is already playing, then it is stopped first. If a different sound starts while a sound is playing, then by default they both play in parallel. This seems like the most likely general-purpose behavior for sounds in direct manipulation interfaces and games.

When the programmer wants more control, the `Interrupt_List` slot of the sound object can be set with a list of other sound objects. Then, whenever a sound starts playing, all of the sound objects in its `Interrupt_List` are stopped. The default value of this slot is a list containing just the sound object itself, which achieves the default behavior where a sound always interrupts itself. If the programmer wanted multiple copies of the *same* sound to be playable at once (for example so that multiple explosions could be heard at the same time), then the `Interrupt_List` can simply be set to the empty list,. Then, no sounds would be interrupted when the new sound starts playing.

Adding a specific sound to the `Interrupt_List` might be useful when the programmer wants to make sure some sounds are not played together. For example, a "dying" sound for a character might be specified to interrupt that character's "walking" sound. Putting the special value `Interrupts_Everything` into the `Interrupt_List` signals that this sound should cause all other sounds to stop.

Pre-empting

Another important issue is what to do when there are not enough channels to play all the desired sounds at the same time. We want to provide machine-independence in a convenient way for the programmer. Therefore, we do not want the programmer to have to inquire about the number of channels and adjust the program code accordingly for different platforms. Instead, we provide a declarative specification that can be used to control what sounds will be pre-empted if necessary.

The `Pre_empt_If_Needed_List` can contain a list of sound objects that will be pre-empted if the current sound needs to play and there are not any available channels. The sound objects in this list are tested in order to see if they are playing. This represents a priority scheme, so that the first item on the list would get pre-empted first. As an example, an explosion sound might specify that it pre-empts the background music sound object. Any sounds that are pre-empted are pushed onto a pending queue, and are resumed when the new sound is finished. This will allow the background music to be continuously playing when there is no other sound pre-empting it. If there are no channels free, and no sound on the `Pre_empt_If_Needed_ List` is currently playing, then Andalusite pre-empts the oldest sound that is currently playing and plays the new sound instead. As a special feature, the programmer can add a special value, called `Dont_Pre_empt`, to the end of the `Pre_empt_If_ Needed_List` which overrides this behavior, and instead skips playing the *new* sound. For example, if the new sound is just for feedback and is not important, it might be marked so as not to pre-empt any other sounds by putting the `Dont_Pre_empt` value as the only value in the `Pre_empt_If_Needed_List`.

To provide additional convenience for the programmer, the objects in the `Pre_empt_If_Needed_List` and the `Interrupt_List` can be the *prototypes* for a set of sounds. In Amulet's prototype-instance object system, any object can serve as a prototype from which to create a set of instances. If the programmer makes instances of any of the objects in the `Pre_empt_If_Needed_List` or the `Interrupt_List`, then those instances will be treated as if they were in the lists. This is a handy way to concisely specify many useful behaviors. For example, if the programmer has various explosion sounds, and wants to make sure that only one explosion is played at a time, then all the sound objects for the explosions could be created from a prototype *explosion* sound object, and then that prototype could be set into all the sounds' `Interrupt_List`.

As a more complex example, suppose that an application has background music, sounds from the actions of various other users, and a foreground sound from one particular user's actions. The programmer might want to make sure that if there is only one channel, the foreground sound plays, if there are two channels, the foreground and one other user's sound plays, and if there are three or more channels, that the foreground and background sounds play, and as many of the other users' sounds as will fit. This can be specified by making prototypes for each type of sound (`foreground`, `background` and `other_users`), and then creating all the sounds as instances of those prototypes. The values for the `Pre_empt_If_Needed_Lists` in the prototypes would be as follows:

Prototype	Value in slot `Pre_empt_If_Needed_List`
`foreground:`	`foreground, background, other_users`
`background:`	*(left as the default value)*
`other_users`	`other_users, background, Dont_Pre_empt`

High-Level Interface

Although the Andalusite low-level sound interface provides a lot of power and flexibility, the interesting contributions are in the high-level capabilities that support synchronizing the sounds with animations and interactive behaviors.

The typical way that sounds are used in games and systems such as SonicFinder [7] is that a sound is played in response to the user's action with the mouse and keyboard. For example, clicking on an object might start an animation and a sound, or moving the mouse might trigger sounds based on where the mouse is located.

To facilitate specifying these kinds of behaviors, Andalusite allows a sound object to be attached to the beginning, duration, or end of any animation or interaction. Animations are supported by animator constraint objects (introduced above) which are can be attached to graphical objects to cause them to be animated. Interactive behaviors are implemented by attaching "Interactor" objects to graphics. For example, a `Move_Grow_Interactor` will move or grow an object with the mouse. The Interactor begins operating when the mouse button is pressed on the object, moves the object while the button is held down, and then stops when the button is released.[2]

We extended the Animation objects and Interactor objects to support sounds by adding three new slots: `Sound_At_Start`, `Interim_Sound`, and `Sound_At_Stop`. If a sound object is put into the `Sound_At_Start` slot of an Animation or Interactor, then whenever it starts, the sound is played. If a sound is put into the `Interim_Sound` slot, then it is played while the Interactor or Animation is operating. The interim sound starts at the end of the start sound (if any) by being linked using the first empty `Next_Sound` slot of the start sound chain (that is, if the start sound has a next sound, then the interim sound is put as the next sound of the last sound in the list). Note that it would not work to start playing the interim sound at the first mouse movement after the start sound, since the mouse might not move at all in which case the interim sound would not start, or conversely the mouse might move immediately, and the interim sound would interrupt or play in parallel with the start sound. Therefore, the `Next_Sound` chain is used to schedule the interim sound for whenever the start sound completes.

If there is no start sound, then the interim sound is started immediately when the animation or Interactor starts. When the animation or Interactor is finished, then the interim sound and start sound (if any) are stopped, and the `Sound_At_Stop` sound is played.

There are various options for how the `Interim_Sound` might be played:

- **Repeated continuously:** Normally, the `Interim_Sound` will have a `Repeat_Count` of `Infinity`, so that the sound will play continuously throughout the interaction. The Animation or Interactor object will then explicitly stop the sound when the behavior is finished.
- **Restarted at each interim event:** While an Interactor is running, its "`Interim_Do`" method is called repeatedly. If the `Interim_Sound` has a `Repeat_Count` that is *not* `Infinity`, then each time the `Interim_Do` method is called, the sound is checked to see if it is playing. If the sound is already playing and it has itself in the `Interrupt_List`, then the sound is started over each time the Interim_Do method is called. This would be useful, for example, for an Interactor in a menu that made a sound as the mouse moved from item to item. If the user moves quickly, you would still want as much of the sound to play as possible for each menu item, but you would want to hear each sound begin. Similarly, for an Interactor serving as a screen reader, it should interrupt the previous reading of any items as the mouse moves over a new item. Another use for this would be to make a clicking sound with each key stroke in a Text-Edit-Interactor.
- **Playing continuously while there are interim events:** If the sound is already playing and it does *not* have itself in the `Interrupt_List`, then the new sound is simply not started, and the old sound is allowed to continue. This is useful for situations where you want the sound to play continuously while the mouse is actively moving around, but to stop when the mouse is still. For example, if you want a truck sound to play continuously when the mouse is being used to move a graphical object, then the truck sound can have its `Repeat_Count` as 1 (the default) and its `Interrupt_List` as empty. Then, while the mouse was moving, the truck sound would play, but if the mouse stopped, then the sound would stop also. When the mouse started moving again, the sound would resume. If the `Interrupt_List` was the default, which is a list containing the sound object itself, then the sound would start over each time the mouse moved, which might sound odd.

[2] Interactors have many parameters for controlling the behavior such as the specific buttons that start and stop, gridding, the form of feedback, the maximum and minimum sizes, etc. [13].

Conveniently, in a Move-Grow Interactor, the `Interim_Do` method is called for every incremental mouse movement while the object is being modified. In a Choice-Interactor for selecting objects, which is used for menus and for the screen reader, the Interim_Do method is only called whenever the mouse moves to a different object. This makes it easy to have a sound each time the mouse cursor moves to a different item. Text-Edit-Interactors call the Interim_Do method on every keystroke.

These options for the interim sound can also be useful for animations. In an animation, the Interim_Do method is called every so many clock ticks, with the interval specified as a parameter of the animation [14]. This provides an easy way to play a sound at a regular interval, and also supports synchronizing sounds with the interim increments of an animation. For example, if a graphical analog clock hand was animated every second, then a "tick" sound could be set as the `Interim_Sound` for the animation and it would be called each time the hand moved. As a special feature, the special animator that bounces objects plays its `Sound_At_Stop` when the object bounces (even though the animation does not stop). Figure 3 shows an example.

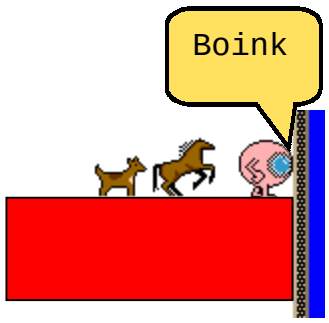

Figure 3. When the animation on the eye icon hits the wall, it plays the "boink" sound which is in its `Sound_At_Stop` slot.

Finally, a sound can be attached to a *command object*. Rather than using a "call-back procedure" as in other toolkits, Amulet allocates a "command object" and calls its "Do" method [14]. Amulet's commands also provide slots and methods to handle undo, selective undo and repeat, and enabling and disabling the command (graying it out). Command objects promote re-use because commands for such high-level behaviors as move-object, create-object, change-property, become-selected, cut, copy, paste, duplicate, quit, to-top and bottom, group and ungroup, undo and redo, and drag-and-drop are supplied in a library and can often be used by applications *without change*. For Andalusite, command objects were augmented to have a `Sound_At_Stop` slot. It contains a sound, it is played when the command's "Do" method is called. For example, the built in Cut command might be augmented with a scissors sound, and then whenever Cut is invoked, either through a menu or accelerator key, the sound will be played.

Synchronizing Animations To Sounds

The previous mechanisms handle the normal case where sounds should be synchronized with interactive behaviors and animations in an interface. The other common case is for an animation to be synchronized to the *duration* of a sound. For example, a small loop of pictures might be cycled during a short song. To make this very easy to specify, the `Animation_At_Start` slot of a sound can contain an animation object which is started when the sound starts playing. The `Interim_Animation` slot can contain an animation to run for the duration of the sound, and the `Animation_At_End` slot can contain an animation to play after the sound is over. As a simple example, if `icon_anim` is an animation constraint object attached to an icon, then the following will cause the icon to animate as long as the sound plays:

```
music = Am_Load_Sound("music.wav");
music.Set(Interim_Animation, icon_anim);
```

EXAMPLE APPLICATIONS

Providing these high-level mechanisms to connect sounds with graphical user interfaces makes a wide variety of uses of sounds very easy to implement, often with a single line of code. For example, the following code attaches the scissors sound to the Cut command:

```
CutCmd.Set(Sound_At_Stop,Am_Load_Sound("scis.snd");
```

In a typical game, there is a background sound, which is started when the game starts and is played in an infinite loop. This can be easily specified as:

```
Am_Object background =
   Am_Load_Sound("background.wav");
background.Set(Repeat_Count, Infinity);
Am_Play_Sound(background);
```

Suppose there is a picture that when animated should play a song. This could be specified as:

```
Am_Object dancer =
   Am_Load_Bitmap("Dancing_rabbit.gif")
Am_Object dancer_animation =
   Am_Animation.Create();
Am_Object music =
   Am_Load_Sound("dancing_music.wav");
music.Set(Repeat_Count, Infinity);
//play the song during the animation
dancer_animation.Set(Interim_Sound, music);
//make the image slot of the dancer be animated
dancer.Set(Image, dancer_animation);
```

As another example, in Bröderbund's KidPix and SonicFinder [7] dragging objects makes a scratching noise, and there is a noise like a screech of brakes when the movement stops. SonicFinder has the additional property that the pitch of the sound depends on the size of the object. This can be easily specified with a constraint:

```
Am_Object mover =
    Am_Move_Grow_Interactor.Create();
Am_Object drag_sound =
    Am_Load_Sound("dragging.wav");
drag_sound.Set(Interrupt_List, NULL);
mover.Set(Interim_Sound, drag_sound);
Am_Object screech_sound =
    Am_Load_Sound("screech.wav");
screech_sound.Set(Speed,
    get_pitch_from_obj_formula);
mover.Set(Sound_At_Stop, screech_sound);
```

In Windows 95, sounds can be associated with particular events, like menus opening, or the user selecting a menu item. This is a special-purpose mechanism that only works for the system's menus. Also, there is apparently no event when the user moves from one menu item to another, so no sounds can be associated with this. In contrast, sounds can be added to any widget in any application using Andalusite since all the behaviors in all widgets are implemented using Interactors. The various sound slots of the Interactors used to implement the widgets can simply be set with the desired sounds. Sounds can be associated with the start and end of behaviors, as in Windows, but also with the interim changes as the mouse moves around. For example, the following causes a click each time the mouse moves to a different menu item:

```
inter = Am_Menu.Get(Interactor);
inter.Set(Interim_Sound,Am_Load_Sound("click.snd"));
```

The generated sounds can be based on "semantic" properties of the data and interface, and not just the graphical presentation, as required by Mercator [6] since all the properties are represented as slots of the data objects, and constraints can be written to compute the properties of the sounds based on the properties of the data objects.

We created a mouse-based "screen reader" Interactor with just a few lines of code (see Figure 4). It is a `Choice_Interactor` that is specified to be always running. It selects any object in any window anywhere on the screen. We also specified that it does not consume any events, but just processes the events and then passes them on to other Interactors (see the Amulet manual [11] for a full description of the parameters and capabilities of Interactors). Thus, the screen reader sees every object the mouse moves over, but allows all the usual behaviors to also operate. The `Interim_Sound` of the screen reader Interactor was set with a `Text_To_Speech` sound object with the `Repeat_Count` left at its default value of one, and its `Interrupt_List` left at its default value of itself, so the sound would be started over each time the Interactor moved to a new object. We put a constraint into the `Text` slot of the sound object that retrieves the name of the object that the mouse is over. If the object is a string, it reads the string. If it is a menu item, it reads the menu item. For graphical objects, a short description is generated using a very simple algorithm based on one property and the type (e.g., "blue rectangle"). A more sophisticated algorithm for generating descriptions, such as used in ENO [1] would be very appropriate and easy to incorporate. If we wanted a screen reader that used the TAB key to move from one object to the next reading each label, then the same sound object and constraint could be used, but a `One_Shot_Interactor` triggered on TAB would be used instead of a `Choice_Interactor`, and its action would be to move the focus to the next object to be read.

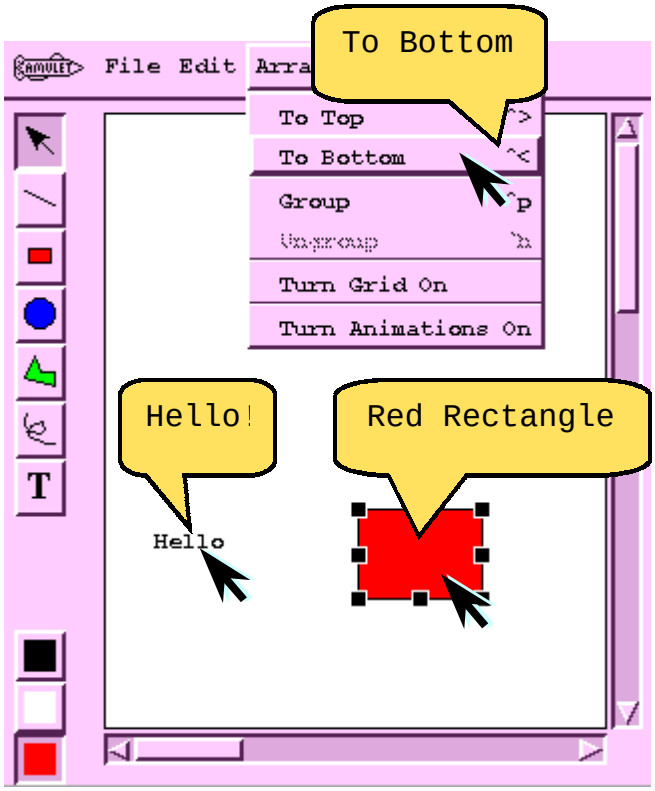

Figure 4. A simulation of what the screen reader Interactor would say as the cursor is moved over various objects. The screen reader is just a `Choice_Interactor` with a `Text_To_Speech` sound object as its `Interim_Sound`, and a constraint that calculates the appropriate words to say.

As a small additional feature, a constraint might be put into the `Text_To_Speech` sound object's `Voice` slot, to choose a different voice if it was reading a menu item that was grayed out. Other properties of the interface could similarly be signaled by changes in the voice parameters.

IMPLEMENTATION

The Andalusite sound system was added to Amulet without significant changes to the low-level Amulet architecture. Amulet already had a built-in facility for animations [14] that deals with time-based phenomenon. The animations can operate in the background driven by timers. The sound system uses the same low-level timer mechanism. Each timer can have a different time-out that determines when it wants the next tick. The way the timers are implemented is that Amulet's main event loop was modified to include a

scheduler, which uses a priority queue choose which timer's time-out is coming next. The main event loop blocks waiting for input events, but when timers are pending, this block is modified to have a time-out of the time until the next timer needs a tick. Whenever the main event loop unblocks, it checks to see an input event arrived, and if any of the timers need a tick. When a timer needs a tick, a call-back method in the timer is called. Timers can be marked as repeated or once-only, and if repeated, then the timer is put back into the scheduling queue.

The Andalusite sound system uses a special timer to handle the scheduling of all the sounds. It is currently scheduled to wake up every 50 milliseconds when there are sounds to be played. This interval was picked to be long enough not to unduly affect the performance, but short enough not to cause noticeable gaps in the sounds.

One important problem with playing sound files using today's window managers is that the programmer is responsible for keeping the buffers full. You cannot just ship the entire sound file to the hardware, especially for long sounds (which might be many thousands of bytes). Instead, the program needs to send a small amount, and then schedule a process to send the next buffer-full at the right time. Andalusite handles this for the programmer using its timer call-back. Every time it wakes up, the timer checks all currently playing sounds to see if they need their buffers refilled. The timer also checks to see if any sounds have finished, and if so, it decrements the `Repeat_Count` of that sound. If the count is zero, then the timer causes the `Next_Sound` to play, if any. If there is no next sound, then the timer checks the pending queue for other sounds to resume playing.

The sound objects are started, stopped, paused, and resumed using type-specific methods of the sound objects:

- `Play_Sound_Method`, to start the sound playing, based on the values of the slots of the sound object.
- `Stop_Sound_Method`, to stop the sound immediately, and clean up any resources.
- `Pause_Sound_Method`, to cause the sound to pause in a way that can be resumed at the current point.
- `Resume_Sound_Method`, to cause the sound to pick up where it was paused.

We have implemented these methods for the two kinds of sound objects Andalusite currently supports: playing recorded sounds and text-to-speech. New kinds of sounds can be easily integrated with the general sound mechanism by just writing these four methods for the new kind of sound.

STATUS AND FUTURE WORK

The Andalusite sound system is working in the Macintosh version of Amulet, and mostly implemented for the PC version. We have added sounds to some existing Amulet games and applications.

For the future, we plan to finish the implementations on the PC and Unix. We want to expand the number and types of applications that use sound. We particularly want to explore using sounds to "visualize" data, along the lines of Bly [3]. We will also be creating new kinds of sound objects like playing MIDI files, and more capabilities for composing sounds from the components like pitch, rhythm, timbre, register, dynamics, etc. We would like to create a variety of applications that use "Earcons" [2] and provide a "clip-sound" library that can be used by applications. We would also like to connect the Andalusite sound system with other new features in Amulet, such as the multi-user support [9], to investigate the use of sounds to support collaboration, as recommended by Gaver [8].

Another interesting future direction is to support more complex compositions of sounds, for example into bars and measures, which would be necessary to create longer and more sophisticated musical pieces.

CONCLUSIONS

The Andalusite sound mechanism in Amulet makes it very easy to add sounds to graphical and direct manipulation user interfaces. Andalusite also makes it easy to create games that use sounds. The architecture provides a machine-independent interface for sounds where most aspects of synchronization, timing, and scheduling can be declaratively specified. The result is that sounds can be easily combined and attached to existing objects, animations and behaviors. Properties of sounds can be computed with constraints based on values of data objects, or they can be calculated using animation constraints to achieve time-based variations of the parameters. We hope that by providing this significantly simpler interface to the programmer, that there will be much more exploration of how sounds can be used, so sounds in interfaces will be much more popular and effective in a wide variety of future interfaces.

ACKNOWLEDGMENTS

For help with this paper, we would like to thank Rob Miller, Rich McDaniel and Brad Vander Zanden.

This research was partially sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

1. Beaudouin-Lafon, M. and Gaver, W.W. "ENO: Synthesizing Structured Sound Spaces," in *Proceedings UIST'94: ACM SIGGRAPH Symposium on User Interface Software and Technology*. 1994. Marina del Rey, CA: pp. 49-57.

2. Blattner, M., Sumikawa, D.A., and Greenberg, R.M., "Earcons and Icons: Their Structure and Common Design Principles." *Human-Computer Interaction*, 1989. **4**(1): pp. 11-44.

3. Bly, S. "Presenting Information In Sound," in *Proceedings Human Factors in Computer Systems*. 1982. Gaithersburg, MD: pp. 371-375.

4. Brewster, S.A. "Using Earcons to Improve the Usability of Tool Palettes," in *Adjunct Proceedings SIGCHI'98: Conference Summary: Human Factors in Computing Systems*. 1998. Los Angeles, CA: pp. 297-298.

5. Buxton, W., "Introduction to This Special Issue on Nonspeech Audio." *Human-Computer Interaction*, 1989. **4**(1): pp. 1-9.

6. Edwards, W.K. and Mynatt, E.D. "An Architecture for Transforming Graphical Interfaces," in *Proceedings UIST'94: ACM SIGGRAPH Symposium on User Interface Software and Technology*. 1994. Marina del Rey, CA: pp. 39-47.

7. Gaver, W.W., "The SonicFinder: An Interface That Uses Auditory Icons." *Human-Computer Interaction*, 1989. **4**(1): pp. 67-94.

8. Gaver, W.W., Smith, R.B., and O'Shea, T. "Effective Sounds in Complex Systems: The ARKOLA Simulation," in *Proceedings SIGCHI'91: Human Factors in Computing Systems*. 1991. New Orleans, LA: pp. 85-90.

9. Huebner, J. and Myers, B.A. "Easily Programminable Shared Objects For Peer-To-Peer Distributed Applications," in *Submitted for Publication*. 1998.

10. Kaleida Labs, I., *ScriptX Architecture and Components Guide, Version 1.0*. 1994, Mountain View, CA:

11. Myers, B.A., *et al.*, *The Amulet V3.0 Reference Manual*. Carnegie Mellon University Computer Science Department, CMU-CS-95-166-R2, 1997,

12. Myers, B.A. and Kosbie, D. "Reusable Hierarchical Command Objects," in *Proceedings CHI'96: Human Factors in Computing Systems*. 1996. Vancouver, BC, Canada: pp. 260-267.

13. Myers, B.A., *et al.*, "The Amulet Environment: New Models for Effective User Interface Software Development." *IEEE Transactions on Software Engineering*, 1997. **23**(6): pp. 347-365.

14. Myers, B.A., *et al.* "Easily Adding Animations to Interfaces Using Constraints," in *Proceedings UIST'96: ACM SIGGRAPH Symposium on User Interface Software and Technology*. 1996. Seattle, WA: pp. 119-128. http://www.cs.cmu.edu/~amulet.

15. Stifelman, L. "A Tool to Support Speech and Non-Speech Audio Feedback Generation in Audio Interfaces," in *Proceedings UIST'95: Eighth Annual Symposium on User Interface Software and Technology*. 1995. pp. 171-179.