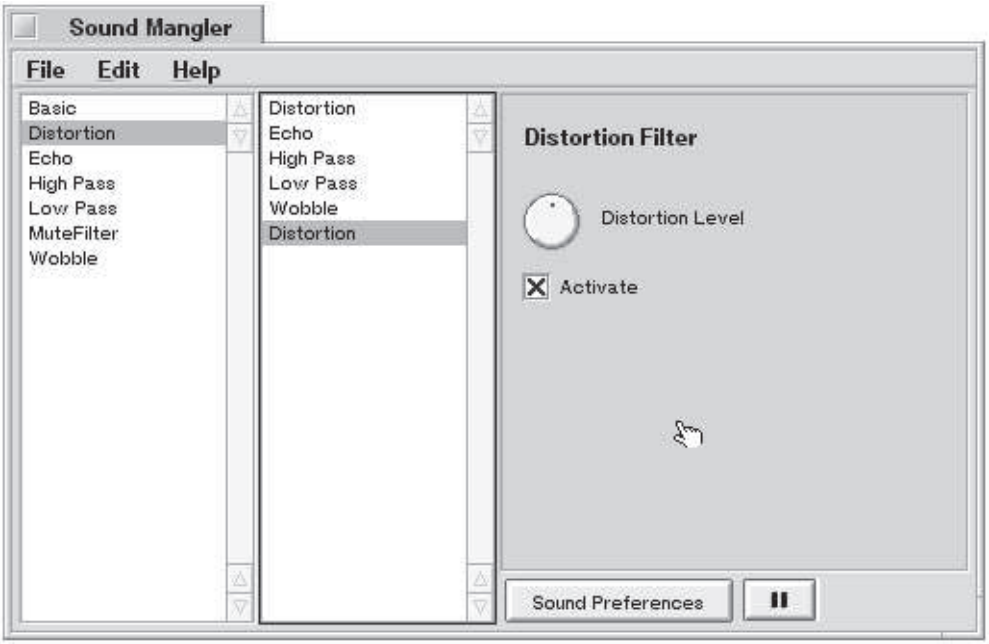
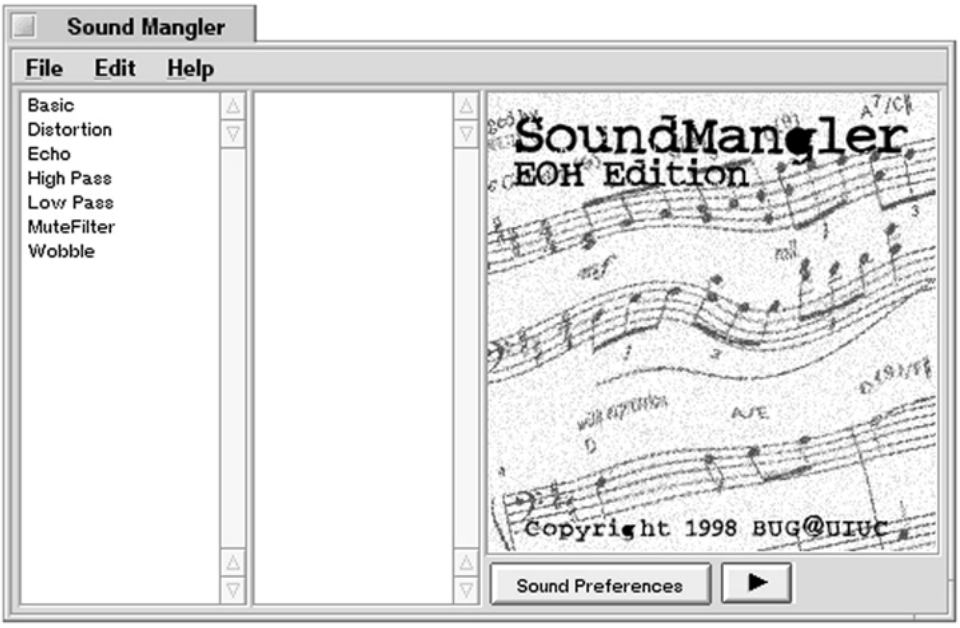
SoundMangler

a real-time audio manipulation framework for the BeOS



Engineering Open House 1998
Be Users Group—BUG@UIUC
Association for Computing Machinery
University of Illinois at Urbana-Champaign
bug@acm.uiuc.edu

SoundMangler



Project Overview

SoundMangler allows a user to manipulate an audio stream in real time using filters, which are plug-in modules of code. The audio stream is provided by the operating system, and SoundMangler hands it to each filter.

The project began in the winter of 1997 when the Be Users Group (BUG) set out to create an application that took full advantage of the BeOS, a multimedia-oriented, multiprocessor-capable operating system. They chose SoundMangler because it required the use of an object-oriented framework, shared libraries, digital signal processing, multithreading, and interprocess communication.

More specifically, SoundMangler has a list of active filter objects, each of which implements a Filter() function. The filters can change the sample in any way it likes. Some of the filters that come with SoundMangler do things like add echo to the sound, remove certain frequency ranges, and create electric guitar effects like flange and distortion.

Because the application hands the audio data to an object that another developer creates, developers are not limited to using SoundMangler to "mangle"—they can use it for any application that analyzes audio data. Some possibilities include spectrographic analysis (a spectrogram is a visual representation of the energy at certain frequencies over a period of time), voiceprinting, and voice recognition. Each of those possibilities can be combined with simpler filters like the ones described above.

The same framework that the BUG designed for audio can be modified and applied to other kinds of

data streams. Video, scientific data, and other digital signals can all be manipulated with a program like SoundMangler.

Audio Stream Manipulation

SoundMangler obtains 1024 samples of data 44 times every second from the BeOS's audio server, passes them to each of the filters, and sends the modified samples back to the audio server. During this time, the filtering function can modify or use the data in any way, provided that all of the calls to Filter() take place in less than about 1/80 of a second. Otherwise, there will be interruptions in the audio stream. Because of this low latency, SoundMangler can filter live streams of audio in real time.

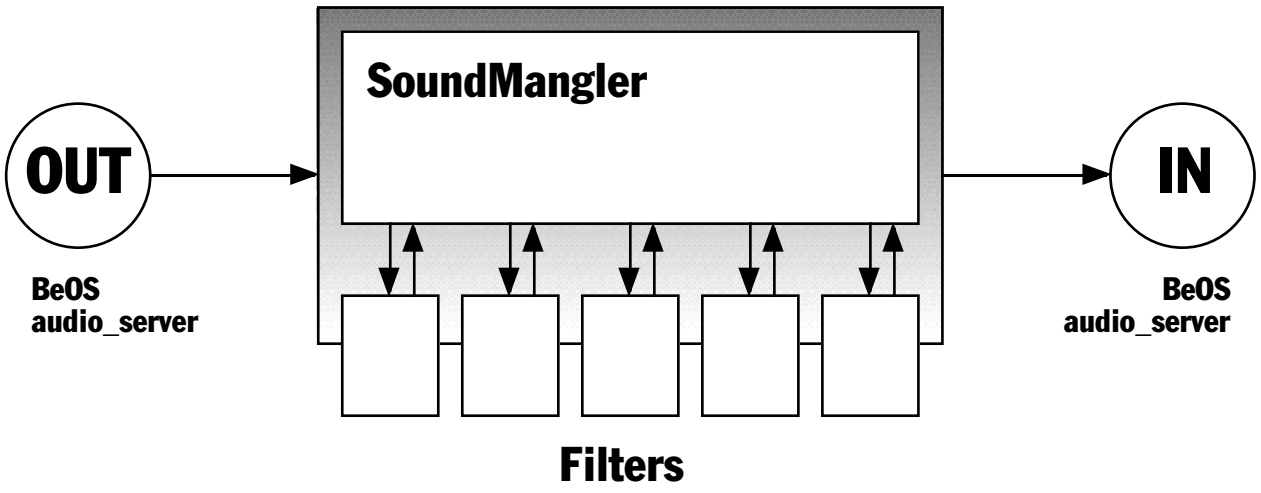


Figure 1: SoundMangler's audio processing scheme.

Typically, to write an application that manipulates an audio stream, a developer either writes one monolithic program with embedded filters, or he/she writes several independent applications that each grab the data from the OS, perform the desired modification, and give the data back to the OS. In the first case, changing a filter means recompiling the entire program—adding a filter is not easy to do. In the second case, the overhead is increased, and each application has to handle OS-specific conventions for dealing with the audio stream.

The advantage to SoundMangler's plug-in framework is that development of filters does not require any knowledge of the BeOS Media Kit (the set of APIs that involve audio and video), nor does it require modification of the program. Using a SoundMangler filter is similar to installing a Netscape plug-in.

Plug-in Architecture

A filter is a shared library that can be loaded dynamically during the execution of a program. The BeOS calls such libraries "add-ons." The filter itself consists of three pieces of code: a filter class, a creator function, and an information view creator. The creator function simply constructs a new filter and gives a pointer (or handle) to SoundMangler. The view creator constructs the informational view a user sees when he/she clicks on a filter in the list. The view is a subclass of BView, the primary

visual interface element in the BeOS.

The SoundMangler development kit provides a base class (SMFilter) that filter developers will subclass and specialize to meet their needs. SMFilter implements a default control panel view and a self-installation application. The minimum that is needed to create a new filter is to subclass SMFilter and implement the Filter() function. To create an interactive filter that has controls, a developer needs to use standard parts of the BeOS API—particularly the Interface kit, which provides views, windows, and controls.

The filter is not confined to the 240x245 view within the SoundMangler window - it can create its own window. In addition, if a filter wants to perform complex (time-intensive) operations on the data, it can use the Filter() call to save the data, and it can analyze it in another thread. This is useful for things like spectrograph creation, which could otherwise block access to the audio sample for too long.

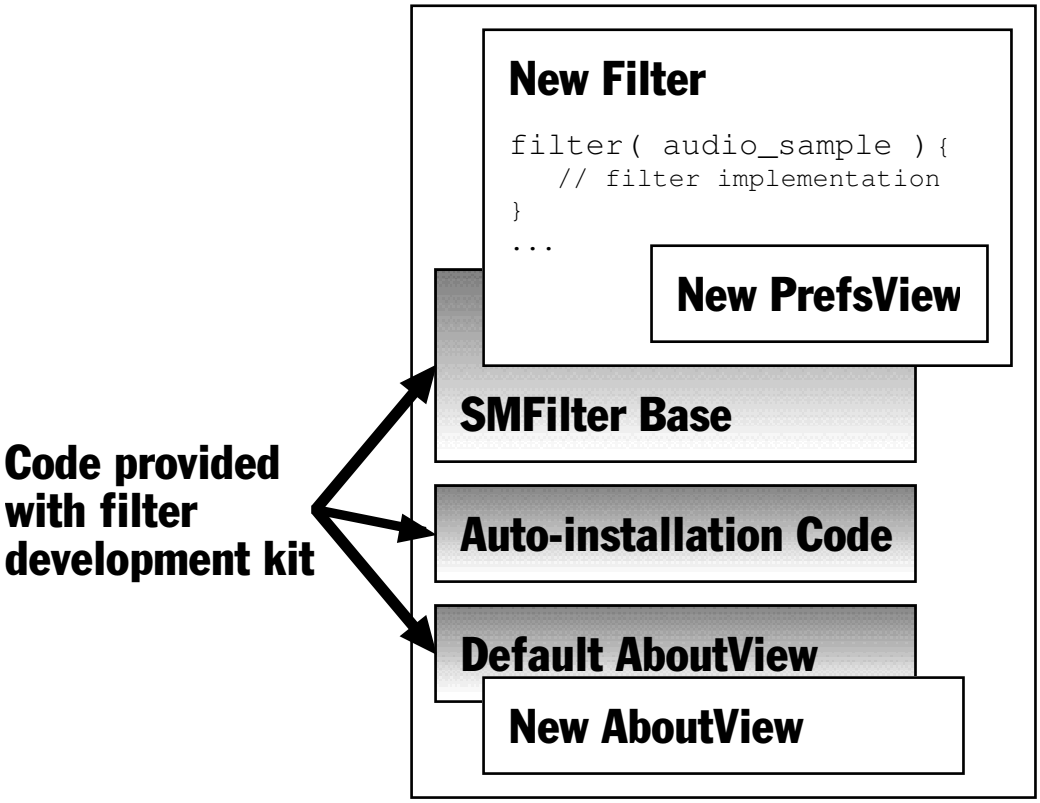


Figure 2: Creating a new filter.

Multithreading

Multithreading allows a program to perform operations in parallel. In a heavily multithreaded system like the BeOS, the performance of an application increases as more processors are added.

In thinking about a program in terms of multiple threads of execution, a developer is forced to have a

design that outlines the responsibility of each part of the program very clearly, thereby enforcing object-oriented design principles. This eliminates a lot of bugs that may be introduced in the design phase. Unfortunately, bugs in the implementation phase are often harder to find and eliminate in a multithreaded application.

SoundMangler runs with a minimum of 6 separate threads: one for the window, one for the application's message loop, and four for the sound engine. The user interface is very responsive, as is the application's main message loop, even when the sound engine is taxing the processors.

The sound engine has four threads: one that waits for messages from other parts of the application, one that gets the data from the audio server, one that handles the filtering of the audio, and one that writes the data back to the server. The BeOS has a task that runs periodically to give SoundMangler an autonomous sample of audio (usually 1024 frames). The sample is protected by a semaphore (or a lock) so that only one thread may access it at any one time. Once the sample is written, the processing thread acquires the semaphore, and it calls each Filter() function on the data. The semaphore is then released to the thread that writes the data back into the outgoing audio stream. The entire process takes place 80 times a second, and the Filter() function is called about 22,000 times a second.

Each filter also has its own thread, which allows the filter to handle its own controls (instead of relying on SoundMangler for a limited set of widgets). This also makes it possible for filters to perform non-real-time operations—those that take longer than the fraction of a second they are given to execute a Filter() call.

One of the goals of the BeOS is to bring multiprocessor machines to the desktop. SoundMangler benefits from the addition of more processors (as do most BeOS programs). With more processor time available to each thread, user interface (UI) and application messaging truly takes place simultaneously. The program will also be able to use more filters at one time when fewer tasks are competing with each other.

Simultaneous Development

SoundMangler was created by eight people, all with varying levels of experience. In one afternoon in November of 1997, the BUG came up with a rough design for an audio manipulation framework. A few weeks later in December, they sat in a room full of computers, wrote code for 18 hours, and produced modules of code that would not compile or work with each other.

After winter break, the BUG reconvened to figure out what went wrong. The biggest problem appeared to be a lack of design specifications. The modules identified as the main pieces of the application are the same modules that are implemented by the current version of SoundMangler, but, the new version was produced with strict interfaces in mind.

During the code-a-thon in December, the developers spent a lot of time asking each other to change interfaces and implementations to accommodate changes in their own code. Though each module was being developed independently, all of them ended up being interdependent.

Before any code was written for the current version of SoundMangler, the developers identified every feature they wanted the program to have and which of those features each module was

responsible for. Next, they determined exactly what a module needed to know about other modules. This led to a definition of each module's interface.

Coding to this interface was all the work that remained. After four weeks or so, the Be Users Group had a working application set. Integration was simple: they put all of the files in one project and compiled it.

Division of Labor

SoundMangler has 5 main components: the application, the window, the filter/add-on manager, the sound engine, and the filter. Each of these components has a very clear interface definition, either through a class definition or a messaging protocol.

The application “owns” the filter management module, which it controls through C++ function calls.

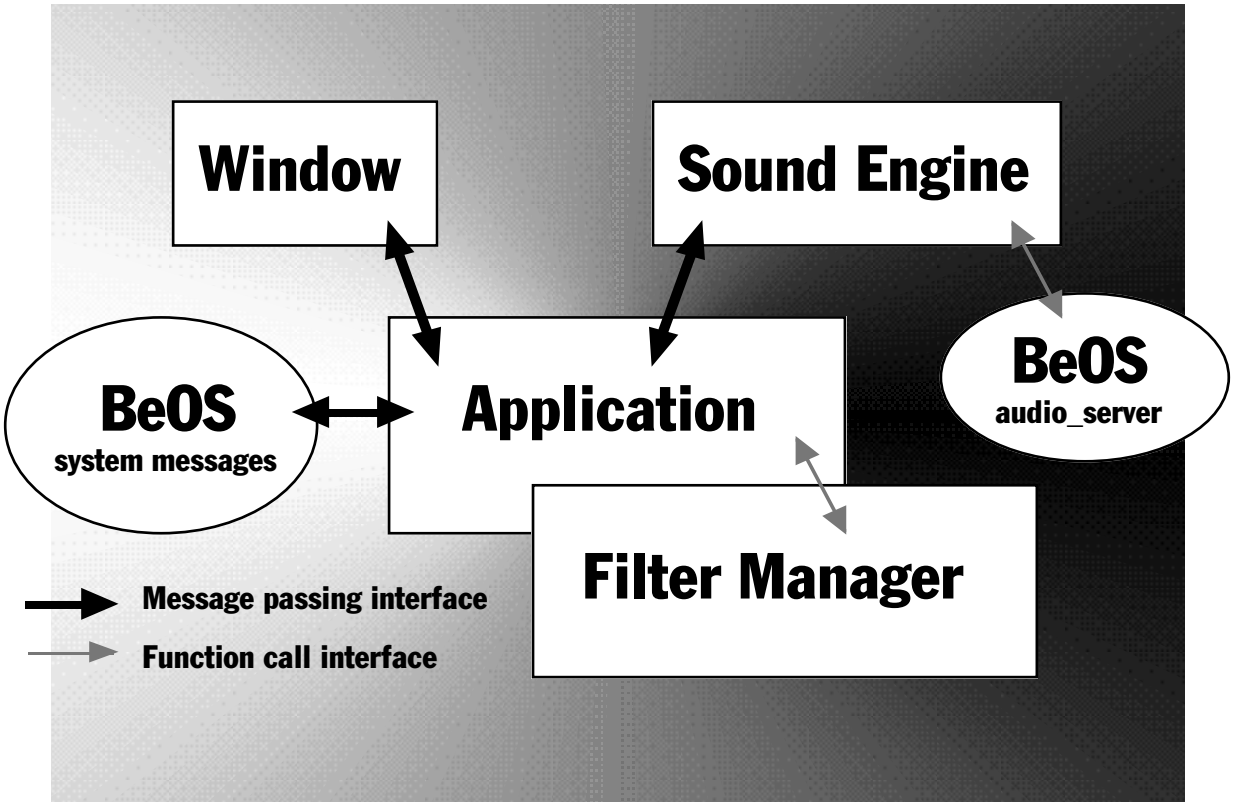


Figure 3: Communication between modules.

The window, application, sound manager, and filter all derive from BeOS library classes (including BLooper, the message loop class). To communicate with one another, they use the standard BMessage model. A BMessage is an object that passes data between objects and threads to provide a form of IPC (interprocess communication). BMessages are passed from one thread to another and handled in the order in which they are received. All of SoundMangler’s message definitions are contained in one document. Here is an example:

```
    // Message: Add Filter
    // Source: SMWindow
    // Target: SMApp
    // Data: char* addon_name, int list_position
    // Reply: SM_NEW_FILTER
    // Tells application to add filter
    SM_ADD_FILTER = '$Maf',
```

When a user drags a filter into the "active" list, the window sends that message to the application, and the application responds by creating a filter of that type (through the FilterManager):

```
void MessageReceived(BMessage * msg) {
   ...
   // A filter has been added to the active list
         case SM_ADD_FILTER: {
            // extract the parameters from the message
               int32 position;
               char *name;
               msg->FindString("addon_name", &name);
               msg->FindInt32("list_position", &position);
               // activate the filter within the FilterManager
               int fID;
               filter_manager->Activate(name, position, fID);
               // Reply to the window with the new filter's ID number
               BMessage reply(SM_NEW_FILTER);
               reply.AddInt32("filter_id", fID);
               msg->SendReply(&reply);
               break;
         }
   ...
...
```

Using Be's messaging model and the associated classes, the need for a new messaging protocol and threading mechanism was eliminated. The design also enforced the division of responsibility for tasks and ownership of data.

In creating SoundMangler, the BUG learned that the design stage of development is most important to the success of a project. With their revised design, they were able to develop separate parts of the program simultaneously, and they integrated the code modules with a minimum of effort.

Each of the developers had different degrees of knowledge of the BeOS, signal processing, and shared library/add-on management, so each developer concentrated on his strength.

The Future of SoundMangler

SoundMangler has several features that remain to be implemented. The hooks are in place for saving documents, accessing online help, and providing a more interactive user interface.

The filter framework is going to be improved by providing the default code within a shared library (libsoundmangler.so). Right now, every filter that is built contains the same code for the default functions and the self-installation program. If a change is made in the implementation of any of

those defaults, all of the filters need to be recompiled to take advantage of the new features. With the shared library, changes to libsoundmangler.so take effect in all of the filters (without a recompile).

Additionally, the default code takes up about 15K in every filter, so it takes 15K more memory to load each filter. For 20 filters to be loaded into memory, the application is wasting about 300K just to provide defaults that are probably not being used.

Using a shared library should also speed filter development by providing some commonly used controls (like those available in liblayout) and DSP algorithm implementations.

Acknowledgments

BUG@UIUC, the Be Users Group at the University of Illinois at Urbana-Champaign, is the author of SoundMangler. Jason Luther, Matt Wronkiewicz, Chip Paul, Vikram Kulkarni, Bryan Cribbs, Mike Khalili, Charlie Powe, and Jay Eychaner all participated in its creation.

This paper was part of the BUG's presentation during UIUC's 1998 Engineering Open House.

Jason Luther is the author of this paper. Please send questions or comments to jluther@uiuc.edu.

Thanks to Jason Gosnell for writing message_daemon and BeBlurb, which were invaluable in SoundMangler's development.

The BUG is a part of the student chapter of the Association for Computing Machinery at UIUC and can be reached via email at bug@acm.uiuc.edu.

SoundMangler is available at http://www.acm.uiuc.edu/bug/projects/soundmangler/.

BUG@UIUC