OpenSoundEdit: An Interactive Visualization and Editing Framework for Timbral Resources

Amar Chaudhary

Center for New Music and Audio Technology
Department of Electrical Engineering and Computer Science
University of California, Berkeley
amar@cs.berkeley.edu

1. Introduction

The emergence of real-time audio synthesis on desktop computer systems is providing musicians and sound designers with richer and more complex control over sounds. These sounds are specified by different representations, including time-domain waveforms, frequency-domain sinusoidal components, physical models and resonance models. *OpenSoundEdit* is a sound editing system that provides a three-dimensional user interface to visualize and edit complex sounds composed from these different sound representations.

Sound in most popular digital audio systems is represented as a time-dependent sequence of audio samples, called a *waveform*. Waveform representations of sound are similar to image representations for visual scenes, allowing transformations over sections of the array, but very little control over individual perceptual components. In graphics, geometric modeling is preferred over images when such control is required. Similarly, sound representations based on synthesis use a function, called a *sound model*, and a template, called a *timbral prototype* that together are used to produce a sound. Just as complex graphical scenes can be built by transforming geometric models, sounds in synthesizers are produced by performing transformations on timbral prototypes.

Traditional hardware music synthesizers [A85, K89] motivated the development of synthesis models based on timbral prototypes and transformations. Early synthesizers were constructed using custom-designed hardware. Today, modern computers are fast enough to implement a synthesis engine in software. Software synthesizers offer increased flexibility and extensibility compared to hardware synthesizers. A variety of model and algorithm combinations have been developed on these software instruments, including large-scale additive synthesis [FRD93], waveguide physical modeling [S96] and resonance models [BBF89].

A sound visualization and editing system is needed that provides control over different sound representations including synthesis models. This editor is a client of a *synthesis server*. A synthesis server is an application that accepts sound representations and control messages from clients and generates waveforms for audio output in real time, as shown in Figure 1.1.

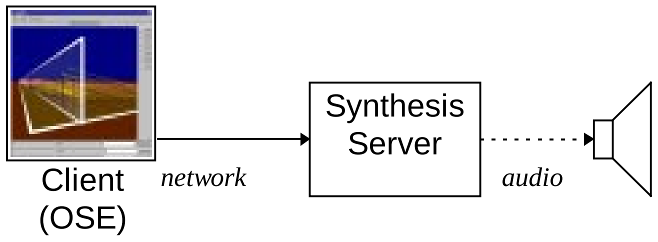
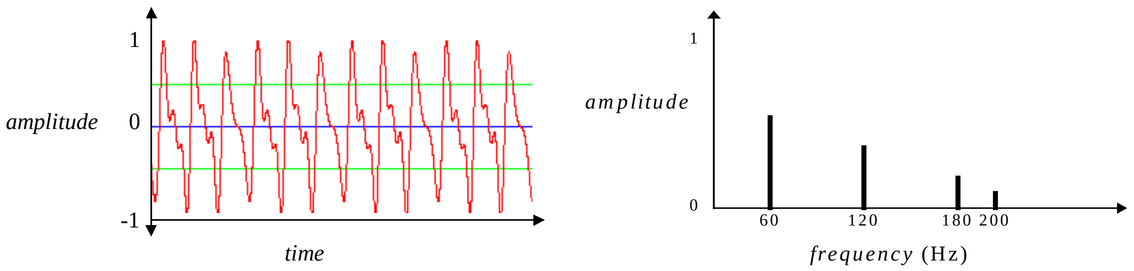


Figure 1.1. Client-server architecture

An effective client interface should have the following properties:

1. A single sound may be described using multiple sound representations. For example, an acoustic instrument may be modeled using both additive synthesis and sampling synthesis (for the attack). The interface must use a consistent visualization for the different timbral prototypes in these models.
2. A meaningful spatial and temporal orientation must be maintained when different views of the same sound or different sounds are being displayed.
3. An interface metaphor is needed to visualize the effect of an editing operation on the sound output.
4. Different timbral prototypes, sound representations, and transformations require customized interface controls.
5. Timbral prototypes, such as those used in additive synthesis, may be large datasets with thousands of data points. The user interface must use efficient methods for storing, presenting and modifying these datasets.
6. The user interface should work with different synthesis server implementations.

Traditionally, graphical user interfaces for sound editing display a sound using a single representation, such as a time-domain waveform or frequency-domain spectrum, as shown in Figures 1.2a and 1.2b, respectively. Some tools use multiple windows for 2D views of different sound representations used in synthesis models [RFL96], while others display sound representations, such as time-varying spectra, in three dimensions [P94].



a) A time-domain waveform view b) A frequency-domain spectrum view

Figure 1.2. Visualizing different sound representations.

OpenSoundEdit, or OSE, extends the ideas found in these systems with the following features:

1. Each sound representation is associated with a *view*, or a mapping of the sound onto a three-dimensional space. The *x, y* and *z* coordinates of this space are time, amplitude and frequency or channel number, respectively.
2. Multiple views are presented to the user in a single 3D display window.
3. The user can dynamically change the viewpoint in the 3D display window.
4. Underlying sound representations can be edited by direct manipulation on the 3D display. Such an edit operation changes both the sound display and the audio played.
5. In cases where the underlying model cannot be edited by a direct manipulation interface, a representation-specific edit interface is supplied.

This paper describes the design and implementation of OpenSoundEdit. The remainder of the paper is organized as follows: Section 2 describes the sound synthesis model in more detail. Section 3 illustrates the use of OSE through a series of examples. Section 4 describes the implementation of the editor, and section 5 concludes the paper.

2. An Overview of Sound Representations and Synthesis

Sounds are most commonly represented using a *waveform* signal, a function $x(t)$ describing the amplitude at time t. Sounds can be also be described using parameterized functions, called *sound models*. The process of producing a waveform representation from a sound model representation is called *synthesis*:

$$x(t) = f(t, p_1, \ldots, p_n) \qquad \textbf{(Equation 2.1)}$$

where $x(t)$ is the waveform and $p_1 \ldots p_n$ are parameters to the sound model function f. These parameters may also be time varying functions. A *timbral prototype* is the set of parameter values required to generate a given sound using a particular synthesis algorithm. It may have as few as one parameter or it might be a large dataset with hundreds of parameters that vary over time. Throughout the remainder of this paper, timbral prototypes will be referred to "prototypes."

OSE currently supports three types of synthesis: sampling, additive synthesis and resonance modeling. It can be easily extended to support other types of synthesis. The remainder of this section describes the three synthesis models supported by OSE.

The first synthesis type, *sampling synthesis*, builds an output waveform from input waveforms scaled by a supplied amplitude and frequency. It can be described using the following function:

$$x(t) = A\,s(\omega\, t) \qquad \textbf{(Equation 2.2)}$$

where s is an input waveform, and A and ω are the amplitude and frequency scaling values for s. The prototype in sampling synthesis is s. A number of editing transformations can be performed on this prototype, including time scaling, filtering, cutting or pasting a selected time range, and redrawing the amplitude function.

Although the equations presented for these synthesis models are continuous functions of time, the actual implementations in the synthesis server use discrete formulae that replace time with samples. For example, the discrete case of sampling synthesis uses the following equation:

$$x(n) = A\, s(\omega\, n / R) \qquad \textbf{(Equation 2.3)}$$

where is *n* the current sample and *R* is the *sample rate*. Throughout the remainder of this paper, only the continuous cases will be presented.

The second supported synthesis model is *additive synthesis*. In additive synthesis, sounds are modeled using sinusoid functions whose amplitude, frequency and phase change over time. Each sinusoid is described using the following equation:

$$A(t)\cos\big(\omega(t)t + \phi\ (t)\big) \qquad \textbf{(Equation 2.4)}$$

where $A(t)$, $\omega(t)$ and $\phi(t)$ represent the amplitude, frequency and phase, respectively, of the sinusoid at time *t*. The functions $A(t)$, $\omega(t)$ and $\phi(t)$ together are referred to as a *sinusoidal track*. An additive sound model is a sum of sinusoidal tracks:

$$x(t) = \sum_{i=1}^{N} A_i(t)\cos\big(\omega_i(t)\,t + \phi_i(t)\big) \qquad \textbf{(Equation 2.5)}$$

where *N* is the total number of sinusoid tracks in the model. The corresponding prototype is the collection of tracks $\{A_i(t), \omega_i(t), \phi_i(t)\}$. The amplitude, frequency and phase components of each track can be independently scaled or modified using the same operations available for the amplitude function in sampling. Additive synthesis provides a very general model for component-based editing of sounds, but this generalization comes at the expense of large representations. An accurate model of a sound may require hundreds of tracks with several hundred points in each track.

The third type of synthesis supported is *resonance modeling*. Sounds in which the frequencies of the partials remain constant and amplitudes are exponentially decaying functions are referred to as *resonances*. Resonance models can be used to describe a wide variety of sounds, including many musical instruments such as piano strings, percussion instruments and the human vocal tract. Sounds based on resonance models are represented more efficiently using a special form of the additive synthesis model:

$$x(t) = \sum_{i=1}^{N} 10^{g_i/20} e^{-\pi k_i t} \cos\big(\omega_i\, t + \phi_i\big) \qquad \textbf{(Equation 2.6)}$$

where g_i, ω_i, k_i and ϕ_i are the gain (i.e., initial amplitude, expressed in decibels), the frequency, bandwidth and phase of the *i*th resonance, respectively. The gain determines initial energy of the resonance, and the bandwidth determines the rate of decay. A smaller bandwidth means a longer decay, and a bandwidth of zero means the resonance stays at constant amplitude. A resonance prototype is a collection of N quadruples $\{\omega_i, g_i, k_i, \phi_i\}$. Resonances can be selected by amplitude, frequency or duration ranges, and reshaped by scaling any of these parameters. Since the entire temporal evolution of each decaying sinusoid is defined by a single parameter, resonances require far less data to specify and modify than additive synthesis tracks. Furthermore, changes in resonance representations can be efficiently transferred from an editor to a synthesis server.

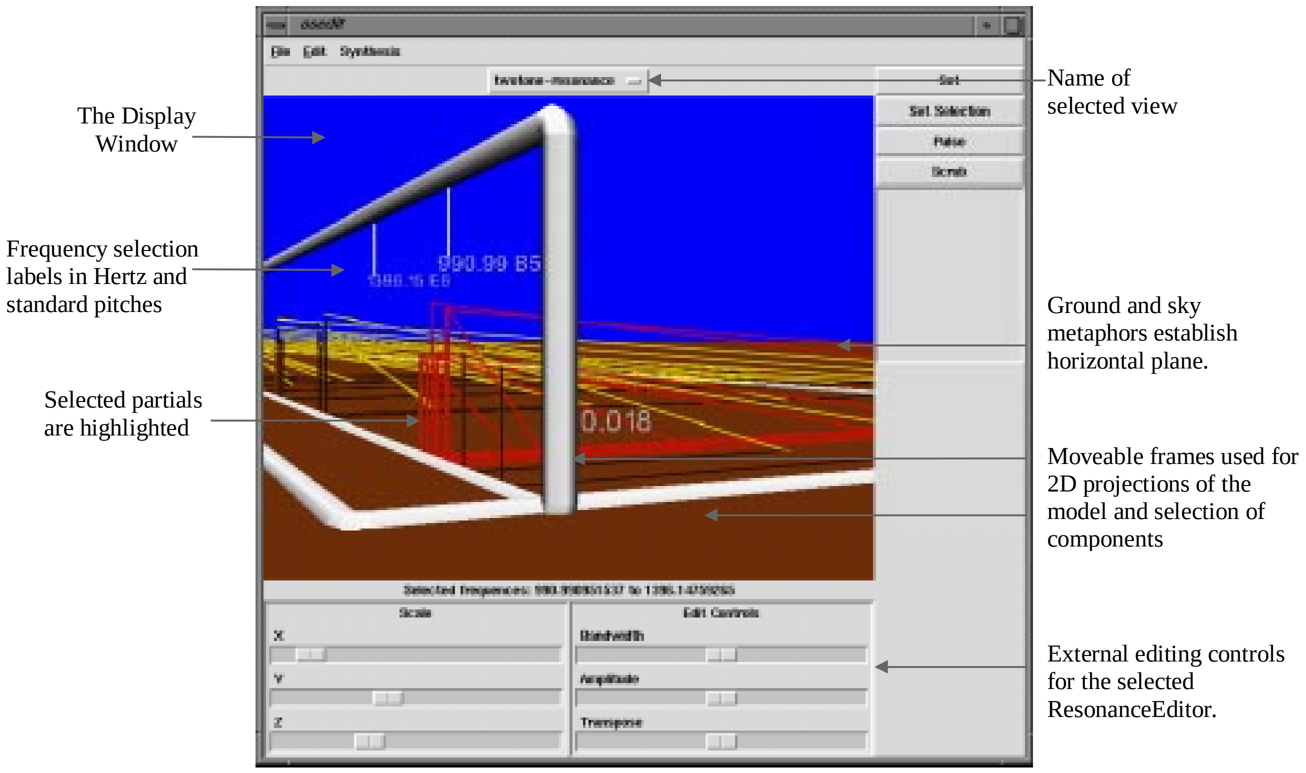
Table 2.1 summarizes the three types of synthesis used to model sounds in OSE.

Synthesis Type	**Description**	**Algorithm / Model**	**Timbral Prototype**
Sampling	Sum of waveforms	$x(t)=\sum x_i(t)$	sampled waveforms: $x_i(t)$
Additive Synthesis	Sum of sinusoid partials	$x(t)=\sum_{i=1}^{N} A_i(t)\cos(\omega_i(t)t+\phi_i(t))$	N frequency, amplitude and phase functions: $\omega_i(t)$, $A_i(t)$, $\phi_i(t)$ Each set of such functions is a *track.*
Resonance Modeling	Sum of exponentially decaying sinusoid partials (i.e., resonances).	$x(t)=\sum_{i=1}^{N} 10^{g_i/20} e^{-\pi k_i t} \cos(\omega_i t+\phi_i)$	N frequency, gain, bandwidth and phase records: $\{\omega_i, g_i, k_i, \phi_i\}$

Table 2.1. Examples of synthesis sound representations.

3. OpenSoundEdit

The OSE interface is illustrated in Figure 3.1.


osedit
File
Edit
Synthesis
twotone-resonance
Set
Set Selection
Pulse
Scrub
The Display Window
Frequency selection labels in Hertz and standard pitches
990.99 B5
1396.15 E6
Selected partials are highlighted
0.018
Name of selected view
Ground and sky metaphors establish horizontal plane.
Moveable frames used for 2D projections of the model and selection of components
External editing controls for the selected ResonanceEditor.
Scale
X
Y
Z
Edit Controls
Bandwidth
Amplitude
Transpose

Selected Frequencies: 990.95893537 to 1396.14759255

Figure 3.1. Features of the OpenSoundEdit interface. This example features a ResonanceEditor view.

In the 3D display window in Figure 3.1, the *x* and *y* coordinates represent time and amplitude, respectively. The *z* coordinate represents frequency in track or resonance views, and channel in waveform views. The *x*-*z* plane, referred to as the *ground* plane, is drawn to help orient the user. Views are placed at arbitrary locations on top of the ground plane, and can be moved in the time or frequency directions. Figure 3.2 illustrates the coordinate axis and ground plane in the display window.

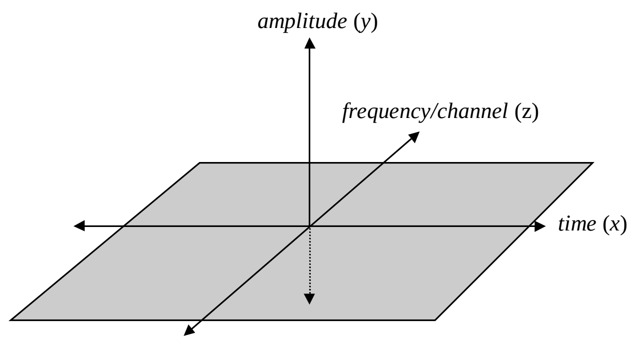
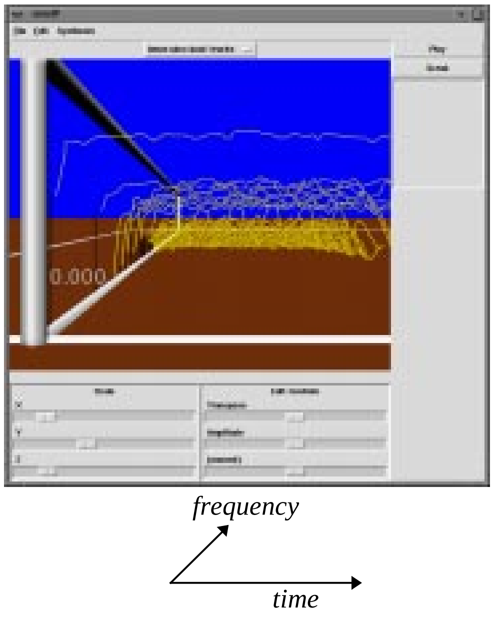
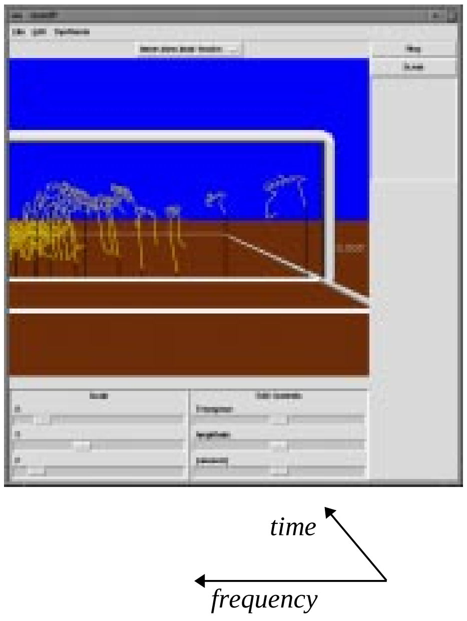


Figure 3.2. The coordinate system of the 3D display window, with the ground plane shown in solid gray.

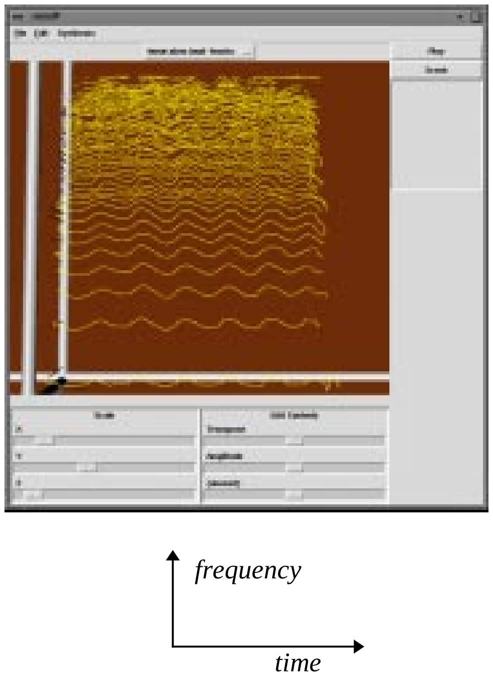
Interface controls are supplied to translate and rotate the current viewpoint, or camera position, within the display. Figure 3.3 shows a view from camera positions perpendicular to the time-amplitude, frequency-amplitude and time-frequency planes:



a) The time-amplitude plane



b) The frequency-amplitude plane



c) The time-frequency plane

Figure 3.3. Looking a view from different camera positions. Different perspectives show different characteristics of the sound. In this example, the frequency modulation is confusing in the time-amplitude and frequency-amplitude planes, but can be clearly seen in the time-frequency plane.

Common viewpoints, such as those in Figure 3.3, can be selected from buttons. However, the camera can also be translated and rotated to arbitrary orientations.

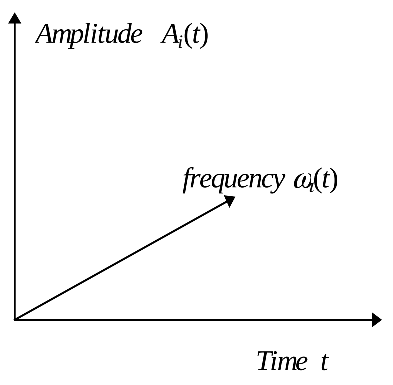
Each view contains a coordinate system drawn on the ground plane, a visualization of the data set (i.e., the yellow lines in Figure 3.1), one or more moveable selection frames (outlined with white bars), and 2D projections of the data set onto the frames (i.e., the black lines). The data set visualizations will be described in detail in the following subsections. The selection frames are used for marking a position on one of the axes. For example, the ResonanceEditor view in Figure 3.1 has selection frames in the frequency-amplitude and ground planes. The frame in the frequency-amplitude plane can be moved along the time axis, and the projected black lines will change to reflect the value of the resonance model at the selected time.

Editing operations are performed via direct manipulation in the 3D display window, or using external controls, as shown at the bottom of Figure 3.1. The behavior of these edit controls are specific to the type of view selected.

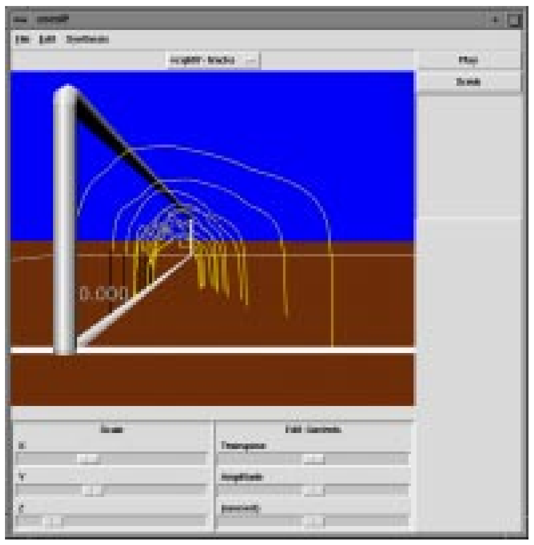
The following subsections describe the Track, Resonance and Waveform views in greater detail.

3.1 Track Editor View

The TrackEditor view is used to display and edit additive synthesis prototypes (i.e., sinusoidal tracks). Tracks are rendered as connected line segments representing the change in amplitude and frequency over time. Time is measured in sections. Frequency and amplitude are measured in Hertz and decibels, respectively, and plotted on logarithmic scales. Phase is not shown. An example of an additive synthesis representation used in the TrackEditor is illustrated in Figure 3.4.



$$x(t) = \sum_{i=1}^{N} A_i(t)\cos\left(\omega_i(t)t + \phi_i(t)\right)$$



a) Coordinate system b) An example TrackEditor,

Figure 3.4. The structure of TrackEditor view**.**

The TrackEditor includes a time-selection frame parallel to the frequency-amplitude plane. It can be moved along the time axis, as illustrated in Figure 3.5. The interpolated frequency and amplitude values of the tracks at the selected time are projected onto the frame as black lines.

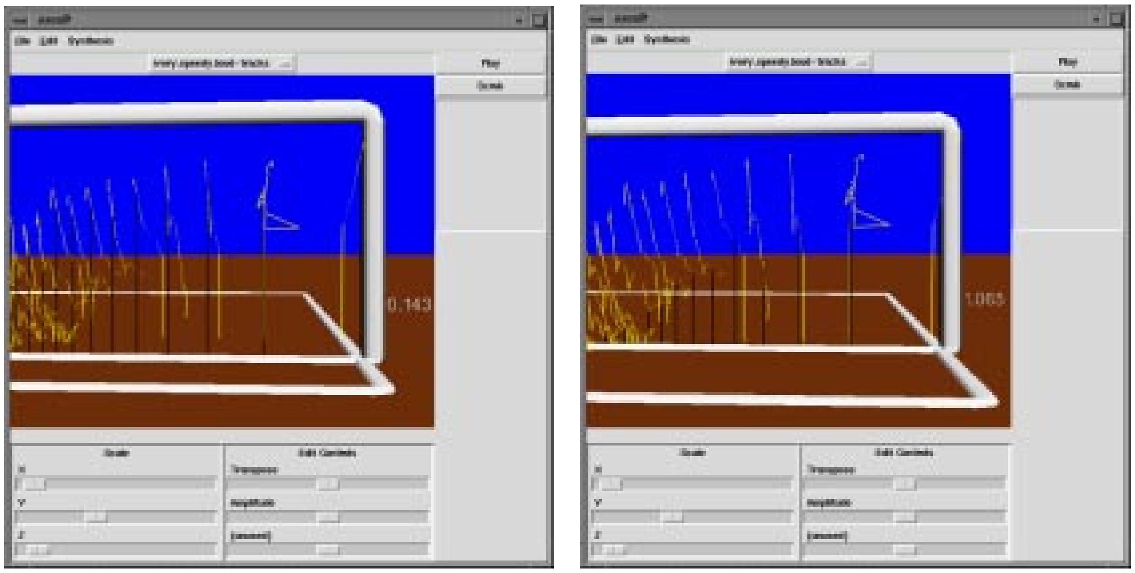


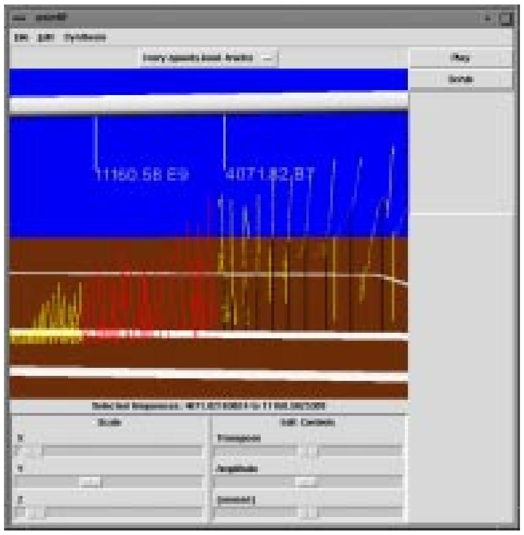
Figure 3.5. Moving the time-selection frame. The black lines projected on the frame reflect the values of the tracks at the selected time.

The time-selection frame can also be used to control time manually when the prototype is being realized on a synthesis server. This technique is known as *scrubbing* in the professional audio community. When scrubbing is used, the values of the tracks are held constant at the current time window position *T*:

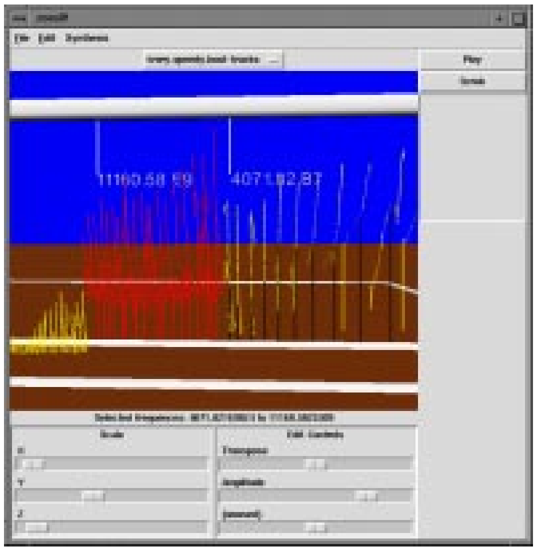
$$x(t) = \sum_{i=1}^{N} A_i(T)\cos\left(\omega_i(T)t + \phi_i(T)\right) \qquad \textbf{(Equation 3.1)}$$

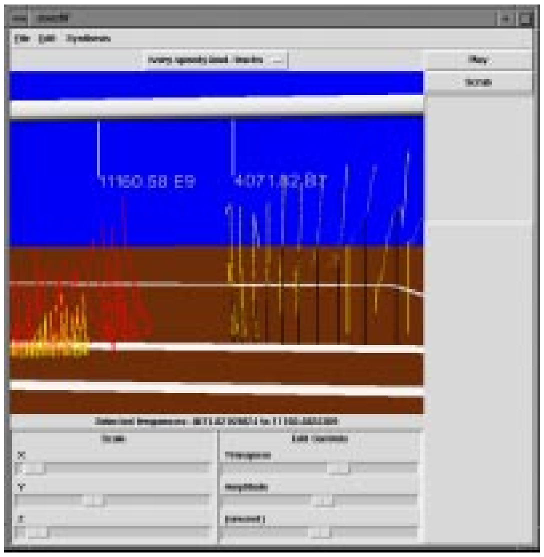
The rate and direction of the time frame motion determines the rate of change in the sound output.

Tracks can be selected by clicking on the white bar at the top of the time-selection frame and dragging the pointer along the range of frequency values to be selected, as shown in Figure 3.6a. The maximum and minimum frequencies of the selection are indicated in Hertz and standard pitch notation (i.e., degree and octave). Selected tracks can be copied into other TrackEditors using the traditional cut, copy and paste operations. Tracks can also be scaled by a constant factor along the amplitude axis or the frequency axis (i.e., transposed), as illustrated in Figures 3.6b and 3.6c.



a) The selected tracks are highlighted. The frequency range of the selection is indicated.





b) The selected tracks are amplitude-scaled

c) The selected tracks are transposed. Note that the selection may overlap unselected tracks after a transposition.

Figure 3.6. Selection and editing of tracks.

Selection can also be used to play or scrub a subset of tracks on a synthesis server instead of the entire prototype.

In addition to scaling in the amplitude and frequency dimensions, the user can reshape a track. This operation is performed by selecting and scaling individual data points within specified tracks. Only those points of the specified tracks that intersect the time window are scaled.

3.2 Resonance Editor View

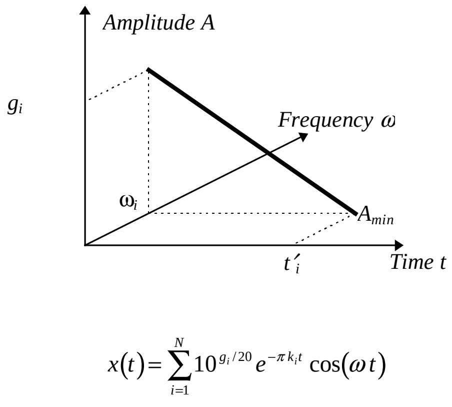
The ResonanceEditor is used to view and edit resonance timbral prototypes. Recall resonance modeling from equation 2.5:

$$x(t) = \sum_{i=1}^{N} 10^{g_i/20} e^{-\pi k_i t} \cos(\omega_i t + \phi_i) \qquad \textbf{(Equation 3.2)}$$

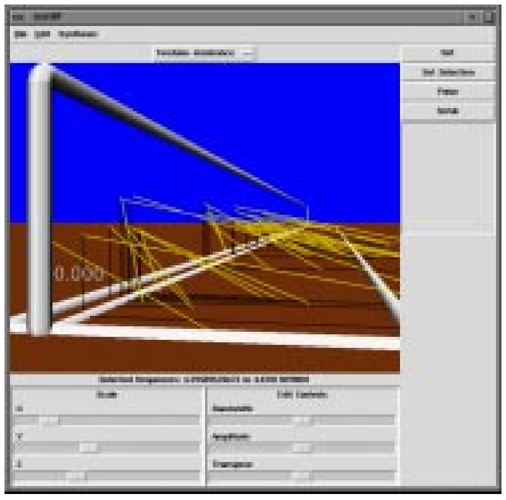
where ω_i, g_i, k_i and ϕ_i denote the frequency, amplitude, bandwidth and phase, respectively, of the *i*th resonance partial. Resonance prototypes are displayed in a coordinate system where the *x*-axis is time in seconds, the *y*-axis is amplitude in decibels, and the *z*-axis is frequency in Hertz. As with TrackEditor views, amplitude and frequency are plotted on logarithmic scales, and phase is not shown. Each component $\{\omega_i, g_i, k_i, \phi_i\}$ is represented by a line whose endpoints are $(0, g_i, \omega)$ and (t'_i, A_{min}, ω), as shown in Figure 3.7a. A_{min} is the *threshold of hearing*, or the gain at which the sound is no longer audible (usually set at -80dB) and t'_i is the time at which the energy of the resonance reaches A_{min}, calculated as follows:

$$t'_i = \frac{g_i - A_{\min}}{(\pi \log_{10} e) k_i} \qquad \textbf{(Equation 3.3)}$$

If the bandwidth k_i is zero, the resonance is represented by a horizontal line starting at $(0, g_i, \omega_i)$.



Amplitude A
g_i
Frequency ω
ω_i
A_{min}
t'_i
Time t
$x(t) = \sum_{i=1}^{N} 10^{g_i/20} e^{-\pi k_i t} \cos(\omega t)$

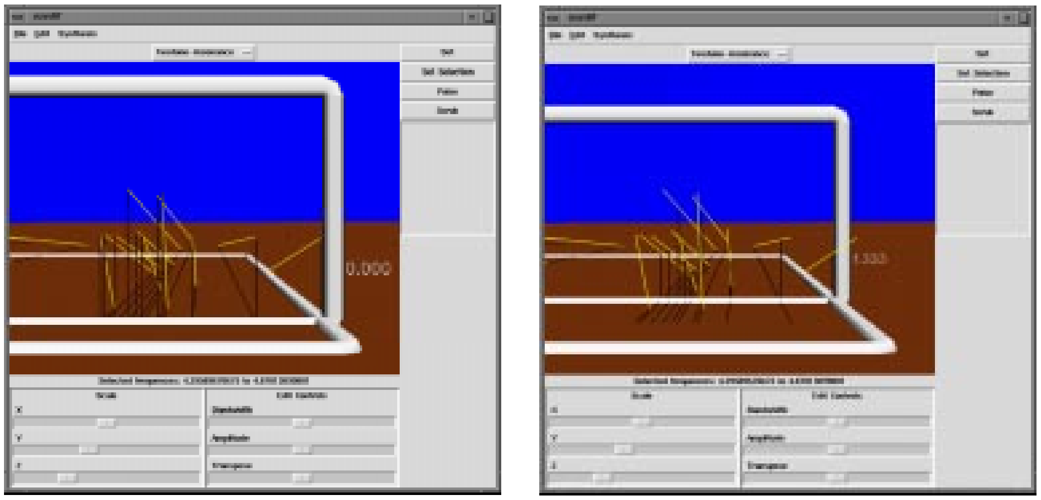


a) Visualization model

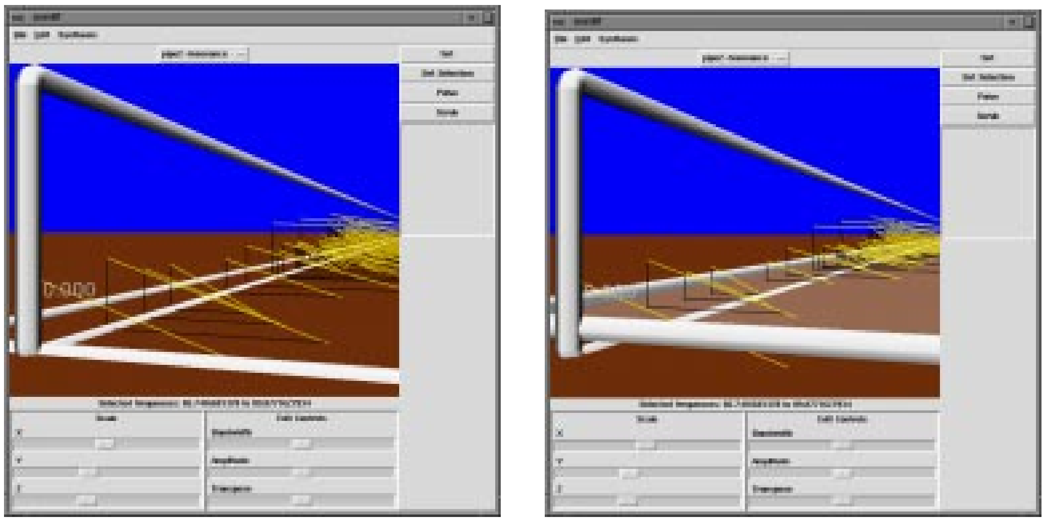
b) A ResonanceEditor

Figure 3.7 Resonance representation

The ResonanceEditor view includes moveable time-selection and amplitude-threshold frames parallel to the frequency-amplitude and time-frequency (i.e., ground) planes, respectively. As in TrackEditor views, the time-selection frame can be moved along the time axis. Lines representing the amplitude values at the selected time are projected on the frame, as shown in Figure 3.8a. The threshold frame can be moved along the amplitude axis, and changes the value of A_{min}. Lines indicating how long each resonance lasts until its energy reaches the selected energy level are drawn on the frame, as shown in Figure 3.8b. The time is calculated using equation 3.1.



a) Moving the time frame. The black lines projected on the window shrink to reflect the amplitude at the new time.



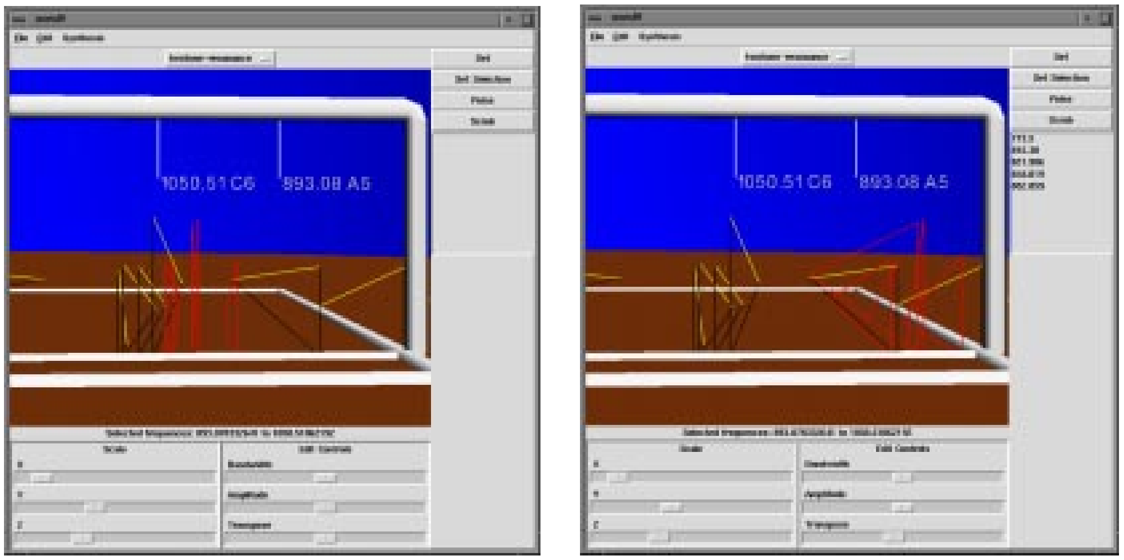
b) Moving the threshold frame up from the ground plane.

Figure 3.8. The time and threshold windows.

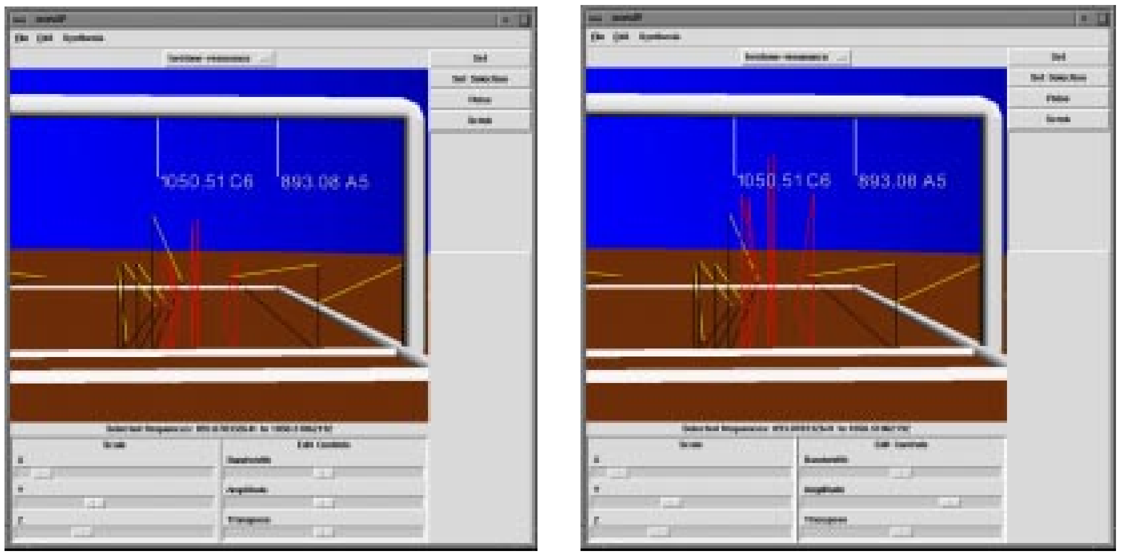
As in the TrackEditor view, the time-selection frame can be used for scrubbing. During scrubbing, the energy values of the sinusoids are held constant at time position *T*:

$$x(t) = \sum_{i=1}^{N} 10^{g_i/20} e^{\pi k_i T} \cos(\omega t) \qquad \textbf{(Equation 3.4)}$$

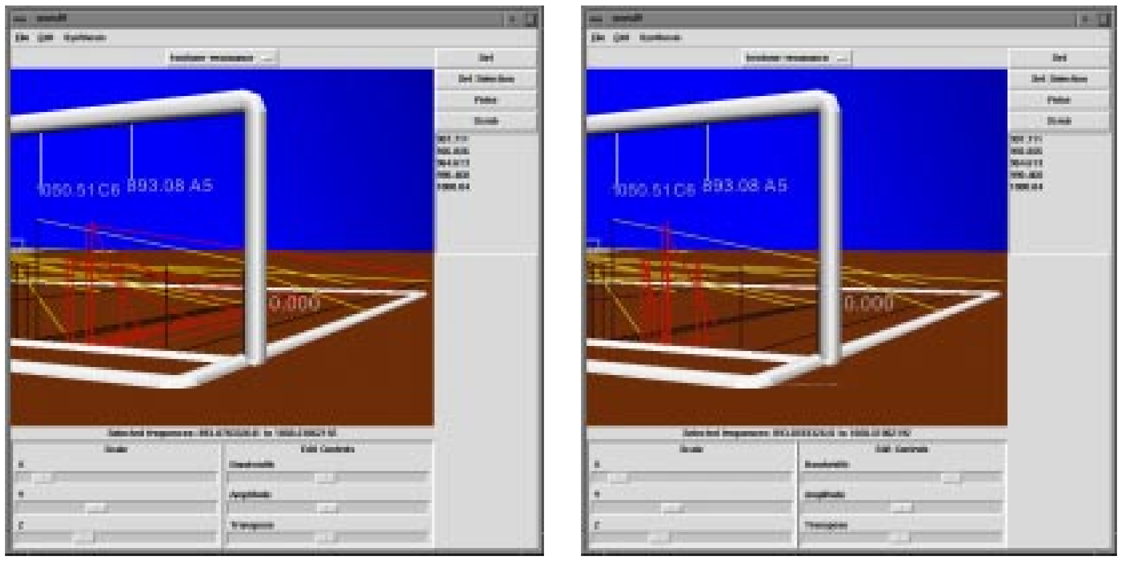
Resonances to be edited are specified by selecting a frequency range along the top bar of the time frame, as shown in Figure 3.9a. Selected resonances can be transposed, gain-scaled or time-scaled, as shown in Figures 3.9a, 3.9b and 3.9c, respectively. They can also be cut, copied and pasted into other ResonanceEditor views.



a) The selected resonances are transposed.



b) The selected resonances are amplitude-scaled.



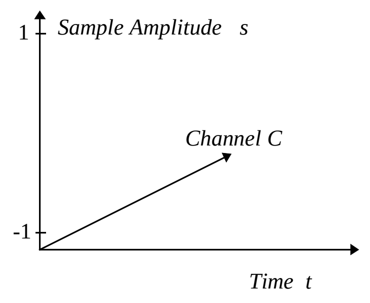
c) The selected resonances are time-scaled.

Figure 3.9. Scaling operations on selected resonances

The user can play or scrub a group of selected resonances on a synthesis server, as well as the entire prototype.

3.3 Waveform View

The Waveform view displays sampled sounds as time-domain waveforms. Waveforms have only two dimensions, time and amplitude. However, waveforms are often separated into *channels*, representing different audio outputs. For example, a stereo sound has two channels. A Waveform view contains sequences of samples separated into one or more channels, as illustrated in Figure 3.10. The *x*-axis represents time, the *y*-axis represents amplitude on a scale from -1 to 1, and the *z*-axis represents the channel as an integer value.



$$x_C(t) = A s_C(\omega t)$$

Figure 3.10. Waveform view

Waveform views do not have any editing controls. They are primarily used to view the output waveform of a synthesis server, or sampled sounds to be loaded by a server.

4. Implementation Issues

OSE is a client process that connects to a real-time synthesis server, as shown in Figure 4.1. It communicates with the server using the Open Sound Control (OSC) [WF97] protocol and uses the Sound Description Interchange Format (SDIF) [SDIF97] to read and write sound representations in files and share them with other users and applications.

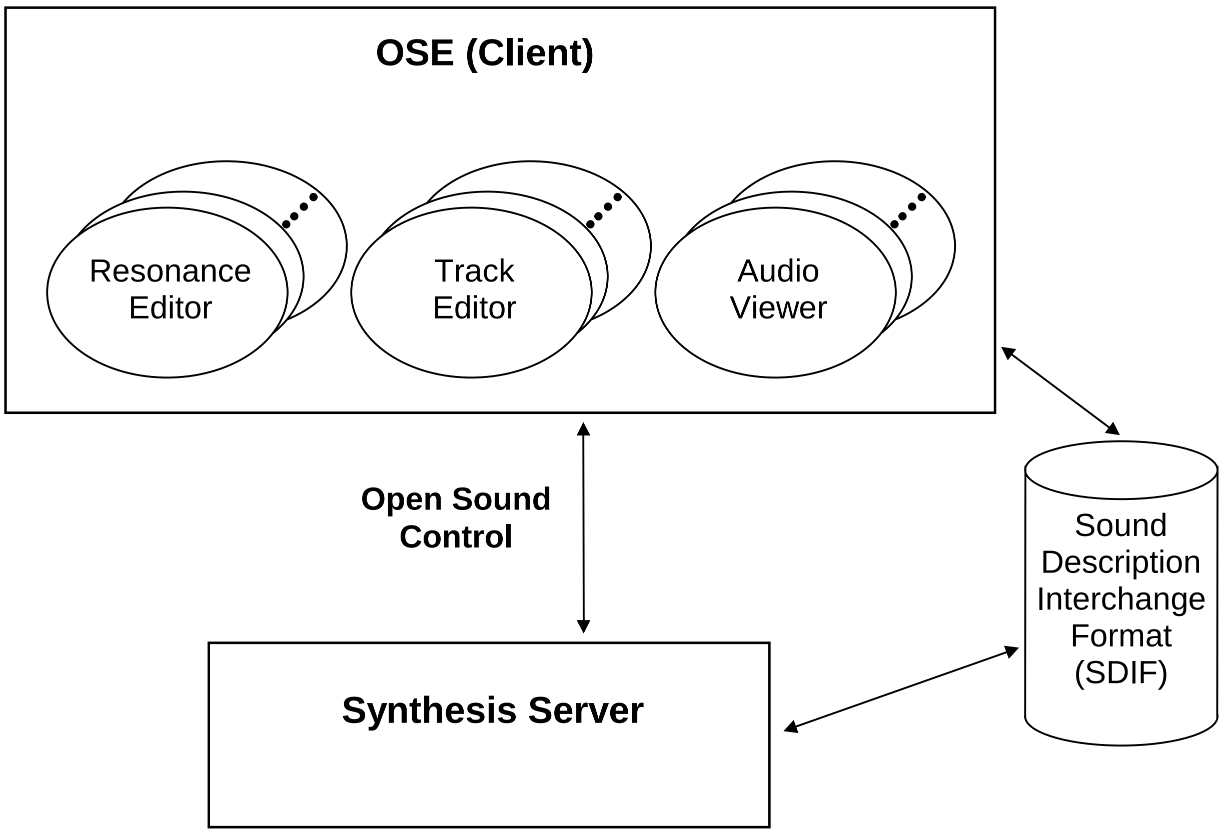


Figure 4.1. The architecture of OpenSoundEdit.

OSE is implemented using OpenGL graphics [OGL97] for displaying and interacting with 3D visualizations and a set of C++ classes for building views from primitive graphics objects. Tcl/Tk [W97] and MIT Otcl [WL95] are used to bind the user interface to views and objects for communication with the synthesis server.

The remainder of this section describes the implementation in greater detail. Section 4.1 describes the synthesis server and the OSC protocol, section 4.2 describes SDIF, and section 4.3 discusses the implementation of graphics and user-interface objects.

4.1 Synthesis Server Communication

A synthesis server is an application that accepts sound representations and control messages from clients, and generates waveforms for audio output in real time [F94]. OSE uses a synthesis server to play the sounds being edited.

The synthesis server can write samples to an audio device, disk file or network device. It can run on the same machine as OSE, or on a separate machine. If the server is on a separate machine, it can write samples to its own audio device, or send them back to the client over a network.

The following examples illustrate the Tcl commands used by OSE to open a synthesis server connection and execute various operations (e.g., load and play a sound file, and define and manipulate sound representations). Note that users do not normally enter these commands; rather, OSE executes them in response to use instructions (e.g., executing operations in buttons or as the result of direct manipulation).

OSE uses an Otcl object, called `SynthesisServer`, to access these services. A new `SynthesisServer` object is instantiated by supplying the address of the server host and the port on which the server listens. For example, the Tcl command

```
SynthesisServer myserver bach.cnmat.berkeley.edu 7005
```

creates an object `myserver` that opens a connection to a server on the host `bach.cnmat.berkeley.edu`. To load a resonance prototype into the server and play it from the beginning, OSE sends the commands:

```
myserver LoadResonance {frequency0 gain0 bandwidth0 frequency1 gain1 bandwidth1 … }
myserver PlayAt 0
```

Because additive synthesis prototypes may be very large, they are initially loaded into the server from a file. After an edit operation, only modified track elements are sent, and the server then updates its copy of the timbral prototype:

```
myserver LoadTracks filename
myserver UpdateTracks {track# {time0 freq0 amp0 time1 freq1 amp1 }
                       track# {time0 freq0 amp0 time1 freq1 amp1 }
                         … }
```

Some operations, such as scrubbing, require that the server use a “logical time system” [RS92]. The server maintains logical time pointers for traversing prototypes. Clients set these pointers, as well as change the rate at which they progress. For example, the rate at which logical time flows with respect to physical time determines the audio playback speed. With such a time system, scrubbing is implemented by setting the rate to zero and the time pointer to the location of the time frame as it is being moved.

```
myserver Rate 0
myserver Goto t
```

where *t* the current time position. If the time rate subsequently set to a non-zero value, the sound will play forward or backward from *t*.

The `SynthesisServer` object binds these high-level methods (e.g., LoadResonance, PlayAt, etc.) to messages sent to the server using the OSC protocol [WF97]. This protocol requires reliable delivery and bounded delay. The bounded delay is needed for commands that change play parameters during real-time synthesis. Reliable delivery is required for commands and data sets that are larger than one packet.

A synthesis server is implemented using an object hierarchy that represents the various transformations, it uses to shape the sound output. These transformations include scaling operations, control of time and conversion from one sound representation to another. OSC uses a hierarchical address to identify a synthesizer object and the messages being sent to it. The address is followed by the message arguments. An example message is:

$/name_1/name_2/\ldots/name_n/message \quad arguments$

The hierarchical address space is specific to each synthesis server. For example, the `LoadResonance` and `PlayAt 0` methods are converted to the following OSC messages when using *softcast* [S97], a server based on additive synthesis:

```
/voices/n/tp/resonances frequency0 gain0 bandwidth0 frequency1 gain1 bandwidth1 …
/voices/n/tm/goto 0.0
```

`tm` (i.e., the “time machine”) controls the logical time system in *softcast*, and `tp` loads and selects timbral prototypes. *Softcast* contains multiple instances of these objects which can run in parallel. Prepending `/voices/n/` to the object name selects the *n*th instance. (This name comes from hardware synthesizers, in which parallel sound-generating structures are referred to as “voices.” It is a metaphor for having multiple voices or instruments in an ensemble).

4.2 Sound Description Interchange Format

The Sound Description Interchange Format (SDIF) is a data format for storing and exchanging sound representations.

An SDIF stream is a sequence of frames arranged in time-ascending order, as illustrated in Figure 4.2. Each frame has a time stamp and a tag corresponding to one of the registered frame types listed in Table 4.1, followed by the actual sound representation for the frame.

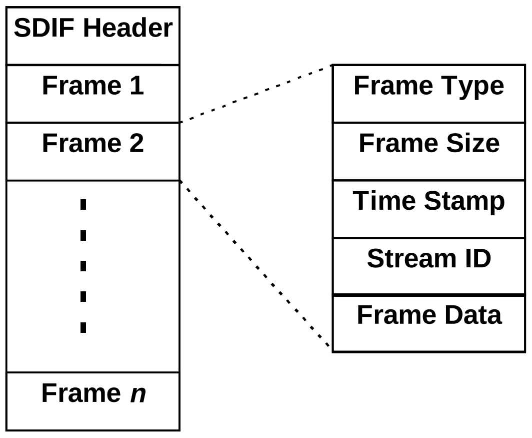


Figure 4.2. The structure of an SDIF stream

Frame Type	Type Code	Sound Representation
Sound	SND	Sampled Waveform
Tracks	TRC	Sinusoidal Tracks
Resonances	RES	Resonances
STFT Spectrum	STF	Descrete Short-Term Fourier Transform (i.e., Frequency Spectrum)
Pitch	PCH	Estimated Pitch Frequencies

Table 4.1. Registered SDIF Frame types for commonly used sound representations.

The time stamps on frames in an SDIF stream correspond to logical time within the server. A frame is used when the logical time pointer reaches the value indicated in the time stamp. Sound representations that are functions of time, such as waveforms or sinusoidal tracks, use separate frames for each time region in the representation. Resonance prototypes, which are not time-varying, use only one frame. The time stamp corresponds to the onset of the resonances in logical time. Figure 4.3a and Figure 4.3b illustrate frames for sinusoidal tracks and resonances, respectively.

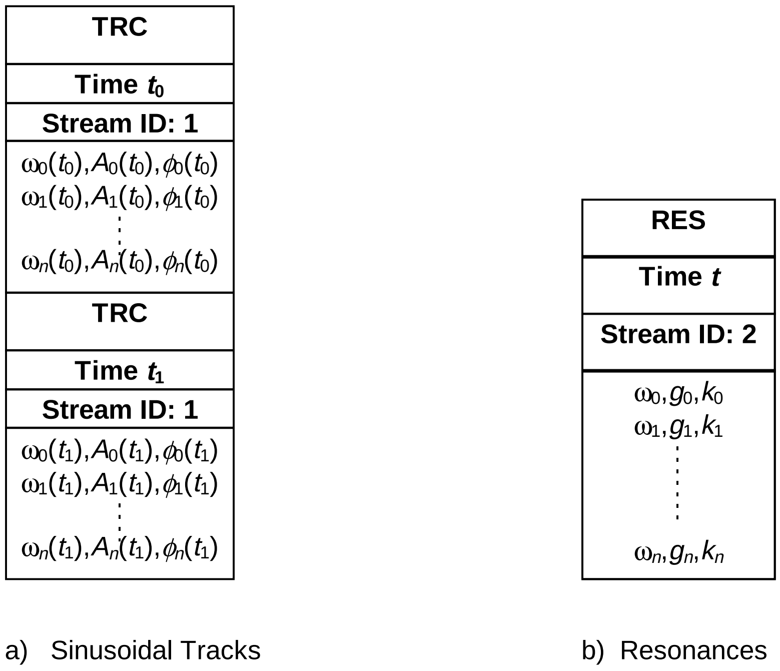

TRC
Time t_0
Stream ID: 1
$\omega_0(t_0),A_0(t_0),\phi_0(t_0)$
$\omega_1(t_0),A_1(t_0),\phi_1(t_0)$
$\omega_n(t_0),A_n(t_0),\phi_n(t_0)$
TRC
Time t_1
Stream ID: 1
$\omega_0(t_1),A_0(t_1),\phi_0(t_1)$
$\omega_1(t_1),A_1(t_1),\phi_1(t_1)$
$\omega_n(t_1),A_n(t_1),\phi_n(t_1)$
RES
Time t
Stream ID: 2
ω_0,g_0,k_0
ω_1,g_1,k_1
ω_n,g_n,k_n
a) Sinusoidal Tracks
b) Resonances

Figure 4.3. SDIF streams for different sound representations.

Because the sound representation type is determined on a frame-by-frame basis, it is possible to mix different sound representations in a single SDIF stream. Each sound representation that is part of a stream is given a unique identification number.

More details on the SDIF representation are presented elsewhere [SDIF97].

4.3 Implementation of OSE

OSE is built on a suite of portable technologies, as shown in Figure 4.4. VTK, the Visualization Toolkit [SML96], is used for modeling 3D objects. Tcl/Tk and Otcl are used to bind user input (i.e., from direct manipulation of the 3D objects or from external Tk widgets) to editing operations and OSC commands for the synthesis server.

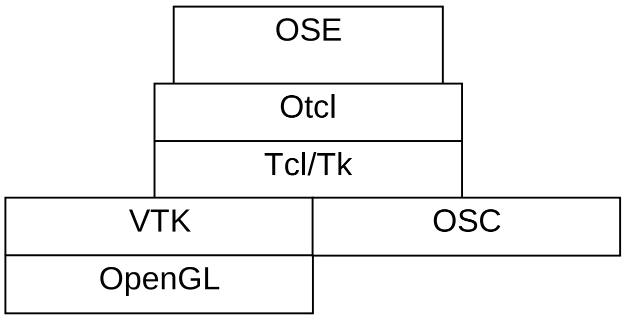
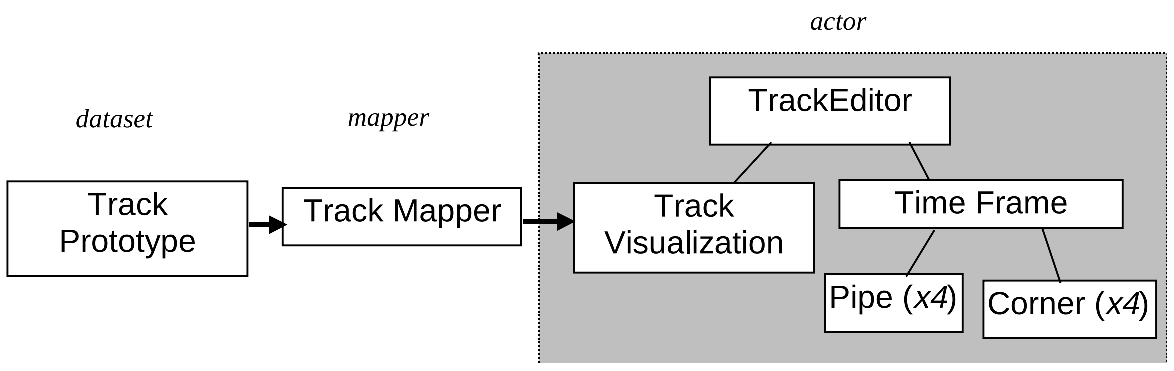
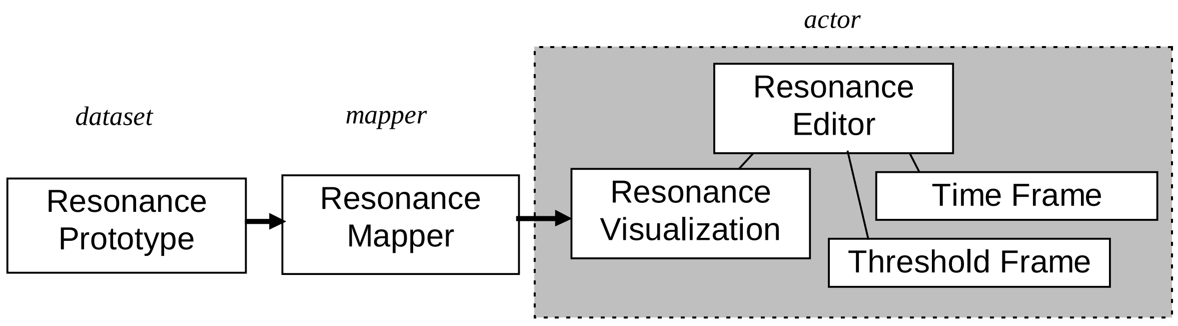


Figure 4.4. OSE implementation layers.

VTK is a portable C++ library for 3D graphics, and is particularly well suited for data visualization. VTK provides three basic abstractions: datasets, mappers and actors. Mappers convert datasets into graphics primitives. Actors are geometric modeling objects that contain other actors or graphics primitives produced by mappers. OSE uses special mappers for track and resonance representations. Each OSE view is a complex actor that contains the graphics output of a track or resonance mapper and actors representing the selection frames. The selection frames are themselves composed of primitive actors representing pipes and corners. Figure 4.5 illustrates the implementation of TrackEditor and ResonanceEditor views using VTK objects.



a) VTK TrackEditor view implementation



b) VTK ResonanceEditor view implementation. As in the track view implementation, the time and threshold frames are composed of primitive actors.

Figure 4.5. VTK implementations of OSE views using the dataset mapper and actor hierarchy abstractions.

In addition to providing abstractions, VTK also includes useful graphics optimizations, such as display lists, in which graphics primitives produced by a mapper are cached until the associated data set is modified. These optimizations are particularly important when visualizing larger prototypes.

5. Discussion

The 3D interface adopted in OSE provides the user a more expressive and intuitive view of sound representations than can be provided on a 2D display. The varying third dimension can be used to express parameters that previously required special mappings onto two dimensions [BBF89, F97, F87]. The unified display allows the user to work with sounds across different representations without separate 2D windows or the clutter of 2D overlays. Changing the viewpoint and moving views on the ground plane makes it easier for the user to locate components of a sound representation (e.g., resonance partials) that are responsible for a particular characteristic of a sound. Direct manipulation allows efficient editing of sound representations using familiar gestures, such as pulling, stretching and sweeping.

Early tests underscored the need for 3D interface models that are well known. For example, early versions of OSE displayed sounds on a completely black background and allowed the user unconstrained translation and rotation of the viewpoint. Such an interface presents the user with the metaphor of “flying through space,” which is unfamiliar and disorienting. The addition of the ground and sky metaphors provide a familiar orientation, which is further enhanced by constraining movement to be parallel to the ground plane. Furthermore, properties of familiar 3D environments can be used to provide additional useful information. For example, the black lines projected onto the time and threshold frames use a familiar “shadow” metaphor for projecting the three dimensional representations onto two dimensions.

User feedback for the current implementation has been encouraging. Experienced users of sound editing tools particularly enjoy the ResonanceEditor view. It is easier to identify and modify components of resonances, which have only one data point per partial, than with sinusoidal tracks, which may have hundreds of points for each partial. Modifying the frequency, gain and decay rate parameters of a resonance partial has an intuitive effect on the sound output. The compact representation also allows more tightly synchronized graphics and audio output when editing resonances.

Although the TrackEditor view is useful for visualizing and exploring track representations, the primitive editing capabilities for scaling individual points or entire tracks have proved inadequate. However, these primitive controls can be used as the basis for customized curve and surface deformations [HHK92, LV97]. Advanced users can add their own curve and surface methods to OSE using Tcl and VTK. Techniques suggested by users include a “pillow model,” in which data points near a modification are raised or lowered in order to maintain a constant total amplitude [W97a].

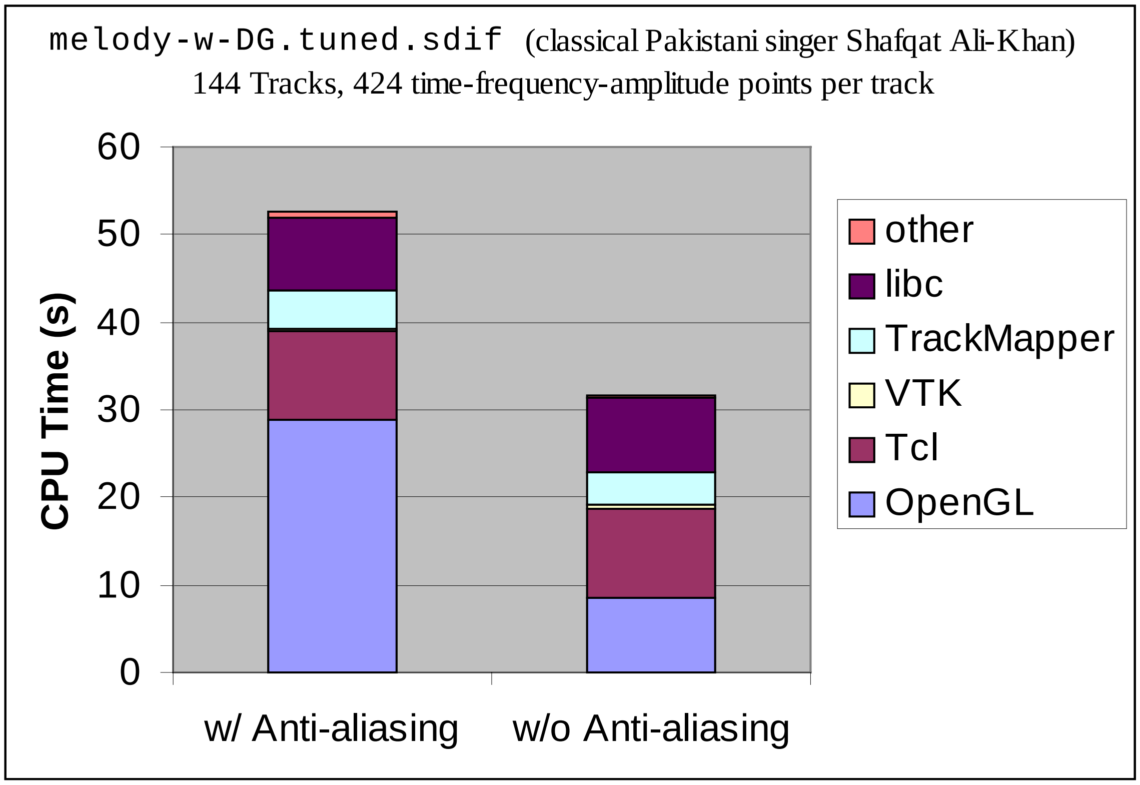
The current implementation of OSE contains approximately 1500 lines of Tcl code, plus an additional 1000 lines of C++ to implement the custom VTK mapper classes.

It was been run on an SGI O2 with Irix 6.3 and an Intel Pentium Pro 200Mhz with Windows NT Workstation 4.0. The O2 includes its own highly optimized OpenGL graphics engine, and the Intel machine uses a Matrox Millennium graphics accelerator card. Performance is adequate on both platforms when viewing small prototypes, but degrades as the size and number of views increases. In an effort to determine the primary performance bottlenecks in the current implementation, the following test was conducted on the O2 using SGI SpeedShop performance tools. A large timbral prototype was loaded and centered in the display. The time-selection frame was then scrubbed from 0 to 2 seconds in 20 steps. The animated centering operation uses display lists for redrawing the view, while scrubbing uses the mapper to draw the changing projection lines on the time-selection frame. The relative amount of time spend executing OpenGL, Tcl, VTK, the custom VTK mapper classes and standard C library routines was determined for each run of the test. The results on large track and resonance prototypes are presented in Figures 5.1a and 5.1b, respectively. It was discovered that the use of anti-aliased lines [OGL97] severely degrades performance, particularly for large resonance prototypes, so separate tests were done with anti-aliasing enabled and disabled.

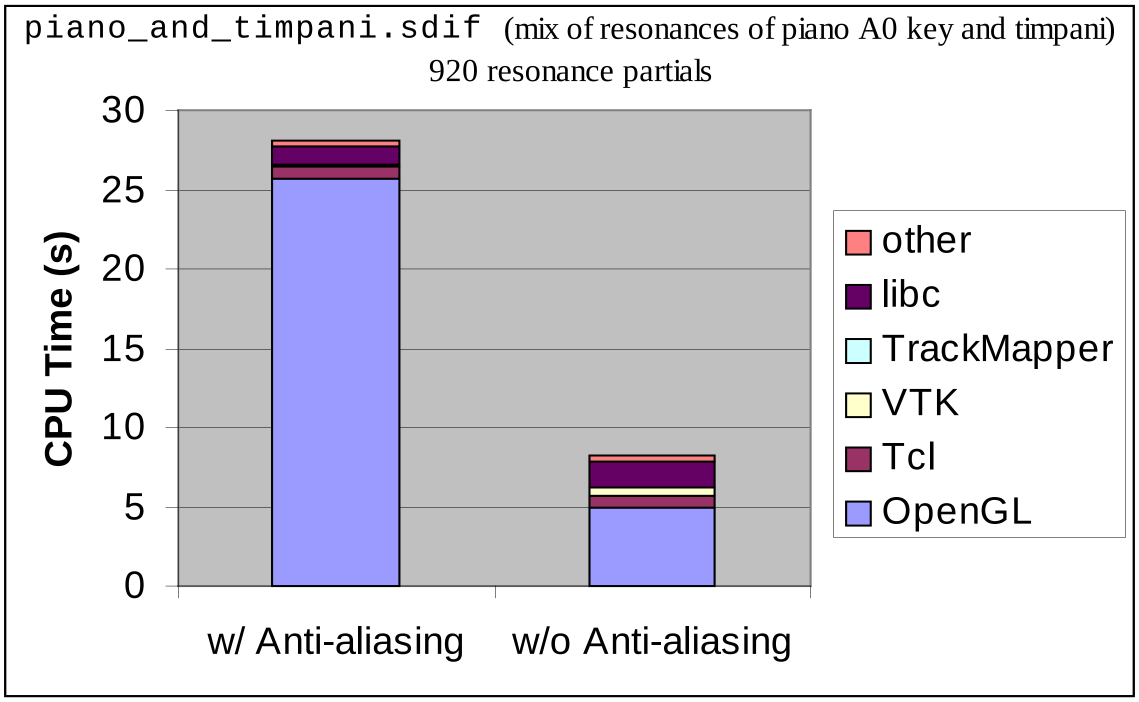
Both TrackEditor and ResonanceEditor views sepnt a significant amount of time in OpenGL system routines. The TrackEditor spend proportionally more time in Tcl and `libc` routines during initialization because the sinusoidal track prototype was much larger than the resonance prototype (over 50,000 data points compared to 920). Likewise, the VTK TrackMapper requires much more time than the ResonanceMapper to traverse its dataset during redraw operations. Graphics optimizations that could improve the mapping process include culling, in which hidden objects are not rendered, and multi-resolution methods [HG97], in which distant models are approximated using fewer data points. Aside from the custom mapper classes, use of the VTK library incurred no significant execution costs.

Acknowledgements

I gratefully acknowledge the NSF Graduate Research Fellowship Program and Silicon Graphics, Inc. for their support of this research.



a) Breakdown of execution time for a TrackEditor view.



b) Breakdown of execution time for a ResonanceEditor view.

Figure 5.1. Breakdown of execution time for loading, centering and scrubbing an OSE view.

References

[A85] J. Allen. “Computer Architectures for Digital Signal Processing.” *Proc of the IEEE*, 73(5), 1985.

[BBF89] J-P Barrière, P-F Baisnee, A. Freed. “A Digital Signal Multiprocessor and its Musical Application.” *Proceedings of the 15th International Computer Music Conference*, Ohio State University, 1989.

[FRD93] A. Freed, X. Rodet, X., P. Depalle. “Synthesis and Control of Hundreds of Sinusoidal Partials on a Desktop Computer without Custom Hardware.” *Proceedings of the International Conference on Signal Processing Applications & Technology*, Santa Clara, CA, 1993.

[F97] `http://cnmat.CNMAT.Berkeley.EDU/Research/Resonances/`
This page discusses Adrian Freed’s 2D representation of resonances in MacMix [F87], and the transition to the ResonanceEditor in OSE.

[F94] A. Freed. “Codevelopment of User Interface, Control and Digital Signal Processing with the HTM Environment.” *Proceedings of the International Conference on Signal Processing Applications & Technology,* Dallas, TX, 1994.

[F87] A. Freed. "Recording, Mixing, and Signal Processing on a Personal Computer." *Proceedings of the AES 5th International Conference on Music and Digital Technology*, 1987

[HG97] P. Heckbert, M. Garland. “Survey of Surface Simplification Algorithms.” Notes from *SIGGRAPH ’97 Course on Multiresolution Surface Modeling*, Los Angeles, CA, 1997.

[HHK92] W.M. Hsu, J.F. Hughes, H. Kaufman. “Direct Manipulation of Free-Form Deformations.” *Proceedings of SIGGRAPH ’92*, Chicago, IL, 1992.

[K89] S.J. Kaplan. “Developing a Commercial Digital Sound Synthesizer.” *The Music Machine*. MIT Press, Cambridge, MA, 1989.

[LV97] J.C. Leon, P. Veron. “Semiglobal Deformation and Correction of Free-form Surfaces Using a Mechanical Alternative.” *Visual Computer*, 13(3), Springer-Verlag, 109-126, 1997.

[OGL97] M. Woo, J. Neider, T. Davis. *OpenGL Programming Guide, 2nd ed.* Addison-Wesley, Reading, MA, 1997.

[P94] A. W. Peevers. “Real-time 3D Signal Analysis/Synthesis Tool Based on the Short Time Fourier Transform.” Masters Degree Report, University of California at Berkeley, 1994

[RFL96] X. Rodet, D. François, G. Levy. “Xspect: A New X/Motif Signal Visualisation, Analysis and Editing Program.” *Proceedings of 22nd International Computer Music Conference*, Honk Kong, 1996

[RS92] L.A. Rowe, B.C. Smith. “A Continuous Media Player.” *Proc. 3rd Int. Workshop on Network and Operating System Support for Digital Audio and Video,* San Diego, CA, 1992.

[SDIF97] Sound Description Interchange Format (SDIF), `http://www.cnmat.berkeley.edu/SDIF`

[SML96] W. Schroeder, K. Martin, B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice Hall, Upper Saddle River, NJ, 1996.

[S96] J. O. Smith, “Physical Modeling Synthesis Update.” *Computer Music Journal*, 20(2), 44-56, 1996.

[S97] The *softcast* synthesizer is documented at `http://cnmat.CNMAT.Berkeley.EDU/CAST/`

[W97] B. B. Welch. *Practical Programming in Tcl & Tk, 2nd Edition*. Prentice Hall, Upper Saddle River, NJ, 1997.

[W97a] Personal communication with David Wessel, CNMAT, 1997.

[WL95] D. Wetherall, C. J. Lindblad. “Extending Tcl for dynamic object-oriented programming.” *Proceedings of the Tcl/Tk Workshop 1995*, Toronto, Ontario, 1995.

[WF97] M. Wright, A. Freed. “OpenSound Control: A New Protocol for Communicating with Sound Synthesizers.” *Proceedings of the 23rd International Computer Music Conference*, Thessoloniki, Greece, 1997.