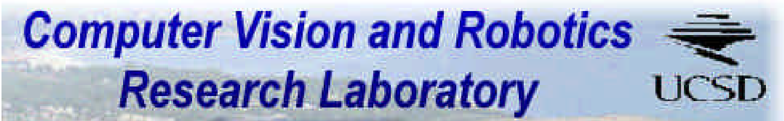

The ATON Project

Center for Research in Electronic Art Technology
University of California, Santa Barbara
Computer Vision and Robotics Research Laboratory
University of California, San Diego
CalTrans Test-Bed Center For Interoperability

ATON Report 2001.06.3: The Design of the CREATE/ATON Auralizer

Stephen Travis Pope and Brent Lehman

stp@create.ucsb.edu

CREATE, UCSB, June, 2001

Contents

Preface 1
Introduction 2
The Basic Scenario 2
Auralizer Internals 3
Extensions to the Auralizer 5
The Eventual Design 7

Preface

The CREATE/ATON Auralizer is a real-time program for spatialising sound, i.e., taking a monophonic sound (a stored sample or a stream) and “placing” it in a virtual acoustic space that is derived from a geometrical model. This report describes the design and implementation of auralizer, beginning with the stripped and simplified auralizer that served as the initial design prototype, and discusses several of the extensions undertaken within the ATON project.

Introduction

The ATON auralizer is a multi-channel sound playback program that takes one or more localized sound sources and places them in a 3-D surround-sound space using any of a variety of psychoacoustical cues. The spatial sound cues it can generate include the following:

- distance/loudness fading and low-pass filter cue,
- inter-aural amplitude differences (inter-speaker panning or IAAD),
- inter-aural time delays (IATD),
- local/global reverb mixing,
- front-back and azimuth-related spatial filtering, and
- acoustical ray-tracing for early impulse responses.

The simplified version we are using at present assumes a listener seated at the origin of the spatial coordinate system (with fixed ears 0.3 m apart), and static sound sources.

The auralizer process has four main components:

- a real-time multi-channel digital mixer, reverberator, and filter;
- a geometry mapper for placing sound sources within a mix space;
- a sample loader and playback scheduler; and
- a score file reader and playback driver.

The **mixer** reads a list of active sound sources and their spatial descriptions. It sums their sample buffers (with optional spatial filtering) and creates 1-to-n channel output. This can be further processed by a local/global configurable reverberator.

The **geometry mapper** computes the values for the chosen set of spatial cues (LR panning, IATD, reverb characteristics and/or spatial filter coefficients) for given sound source and listener's ear positions. For a fixed listener and fixed sources, the geometry calculation is done only at the time of sample triggering.

The **sample loader** reads sound files into the mixer's memory, and can be called at runtime to act as a playback scheduler for loaded samples. This will be replaced with a network sample streaming component in the future.

The simple **score file reader** is used for debugging and can sequence sounds at different points in space. It will be replaced by a CORBA interface that can be called from external processes such as VR systems or spatial sound playback sequencers.

Typically, you have at least two separate threads: (a) the input thread that accepts input commands and schedules samples for playback (possibly recomputing the geometry on any source or listener movement), and (b) the output thread that prepares output sample buffers (calling the mixer components) in response to regularly-scheduled requests from the DAC output driver.

The Basic Scenario

The Figure below shows the basic components of the auralizer schematically. The input side is split up into two blocks: the sample loading and scheduled sample playback, both of which read the score file and either register sounds (creating data structures in sample stor-

age), or create active source records and calling the geometry computation to trigger sample playback. The input and geometry components manipulate the records in the “active sources” list. The output loop is triggered by the regular call-backs from the DAC driver. (These come in at the rate of between 4 and 40 Hz, depending on sample rate and buffer size.) The output loop calls the mixer to prepare the next output buffer.

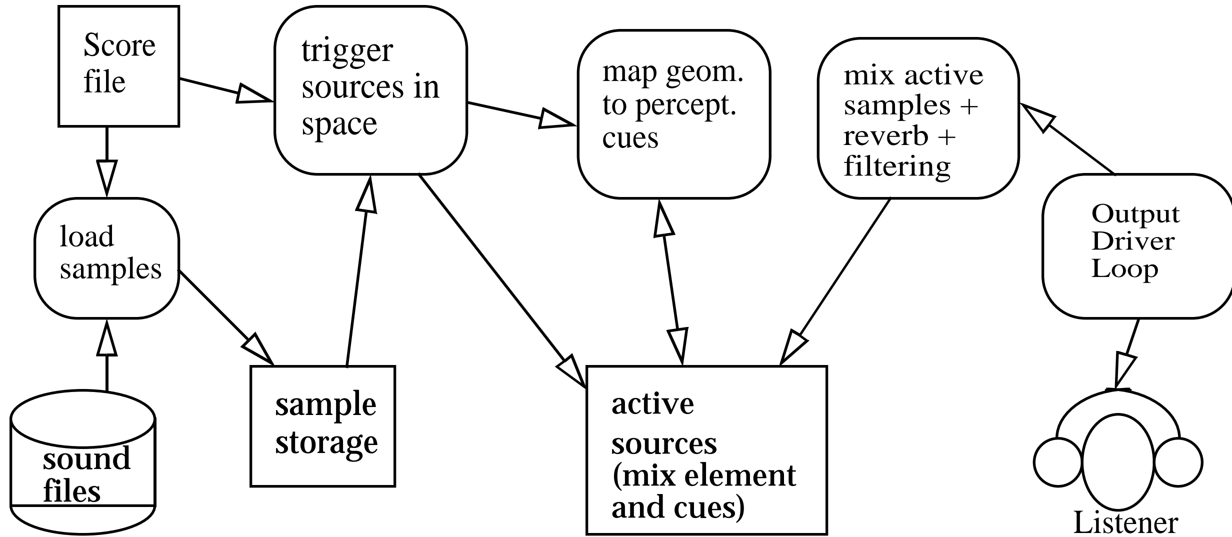

Figure: ATON Auralizer Block Diagram

Auralizer Internals

The following sections will step through the major functions of the auralizer in turn.

Input Loop

The current version reads a score file that has 2 sections:

1) Register samples given a file name and “index”; the score reader function reads lines of the form:

```
s  file_name  snd_index
```

and calls the sampler function

```
add_sound(char *sound_name,  u_int index)
```

This creates a sample record structure and reads in the samples of the given (assumed monophonic) sound file. The C declaration of this structure looks like:

```
// Sample cell data structure

typedef struct {
    char *   file_name;     /* the sample's sound file name */
    short *  samples;       /* pointer to sample storage */
    u_int    length;        /* number of samples */
    short *  f_samples;     /* pointer to low-pass filtered samples */
    u_int    index;         /* client's sound ID (MIDI key) */
} APE_sample_struct;
```

2) Play samples given an index, amplitude, and 3D position in space; the score reader reads event lines of the form

```
e  delta_time  snd_index  amplitude  posX  posY  posZ
```

and calls the function

```
play_sound(u_int index,  float amplitude,  point_t *position)
```

This will take the sample record and create a mix element structure for the source. (There can be several copies of the same sample playing at once.) The mix element record's declaration is,

```
// Mix element header structure.

typedef struct {
    APE_sample_struct *snd;    /* the sample structure for the mix element */
    float       amp;           /* relative ampl. of the signal (0.0 - 1.0) */
    float       scaled_amp;    /* scaled (pos., dist., etc.) amplitude */
    point_t     position;      /* absolute position of the sound source */
    float       LR;            /* left/right stereo balance (0.0 - 1.0) */
    float       rev_ratio;     /* ratio of direct to reverberated signal */
    u_int       l_delay;       /* left-channel delay for distance cue */
    u_int       r_delay;       /* right-channel delay for distance cue */
    float       filter_ratio;  /* front/back filter ratio */
} APE_mix_struct;
```

The play function will then call the geometry calculation function

```
map_position(APE_mix_struct *mix, point_t *position);
```

Geometry Functions

The map_position function takes a source position and computes the mix_struct's parameters according to the psychoacoustical cues, which might include the following:

- distance/loudness fading (scaled_amp) and low-pass filter cue,
- inter-aural amplitude differences (LR, inter-speaker panning or IAAD),
- inter-aural time delays (IATD, l_delay, r_delay),
- local/global reverb mixing (rev_ratio),
- front-back and azimuth-related spatial filtering (filter_ratio), and possibly
- acoustical ray-tracing for early impulse responses (not shown).

These properties can be computed once at the start of the sample, if we assume that the source doesn't move during the sample. To provide for moving sources or a mobile listener, we need to call this mapping function on any source or listener movement (or any change in the characteristics of the room). This will probably run as a separate process in a future version.

Output Loop

The output thread gets regular calls from the DAC driver (either by call-backs, asynchronous-IO, or blocking writes) to prepare the next output buffer. The output loop periodically calls the mixer's function

```
prepare_buffer(u_int bufnum)
```

Mixer

When it gets a call to compute an output sample buffer, the mixer reads through all of the active sounds and adds their current samples to the output. It makes the call

```
sum_buffer(u_int bufnum, u_int length, u_int out_buf, u_int offset)
```

which constitutes the inner loop of the mixer. This can do simple distance fading (scale amplitude with distance), stereo panning (IAAD), apply IATD delays, and chose which reverberator and filter steps are to be carried out. For each active sound, it also decrements the count of remaining samples so it can turn off the sample when it's done. When it's done, the mixer returns the ready output buffer to the DAC driver. (The next time it's called, there may be a different set of active sounds.)

A fancier mixer could do per-source early reflections (reverb) and complex spatial filtering (potentially of every source) in the inner loop.

Performance

The version of the auralizer described above was developed in C and tested on 300-500 MHz Pentium-class computers running the Linux operating system. Operating at a sample rate of 44.1 kHz with stereophonic output and playing back several overlapping sound sources, it rarely required more than 10% of the CPU for its processing.

Extensions to the Auralizer

Given this initial "stripped" design, there are several dimensions along which one could extend the auralizer for greater flexibility, better-quality rendering, or to support more sources or listeners; these include:

- better spatial processing, e.g., computed early reflections for "physical model" reverberators, linear spatial filters for front/back and height cues, or even full HRTF processing for user modeling
- user tracking;
- source movement;
- streaming source input;
- distributed clients

We will discuss several of these areas below, and present the enhancements we made within the ATON project.

Physical Model Reverberator

The initial "default" reverberator has a fixed delay pattern, with only the local/global reverb ratio being set based on the source/listener geometry. The first revision was to use a basic 2-D room model and simple acoustical ray tracing to derive the first few real reflections

of each sound source. These early reflections are then simulated in the sound using a multi-tap delay line for each source/ear pair (i.e., it's compute-expensive). The late reverb is still computed by a global reverberator.

After subjective listening experiments with this enhanced version, users generally agree that the localization of the sound sources is improved, but we have not yet undertaken a formal localization test to measure this.

HRTF-like Processing

There are several sources for reduced versions of the head-related transfer function (HRTF) data set. The main problems with HRTFs are (1) that the data set is quite large (since it's a continuous 4-D function that needs to be sampled at a high spatial frequency to be 100% effective), and (2) that individual listener differences are quite significant (i.e., because of our physiological differences, one person's HRTF will not sound very convincing to another). Nevertheless, there are some insights that can be gained from studying HRTF data sets, and several reduced versions have been derived from the full data set.

For our experiment, we chose a simple set of 24 FIR filters that represent different directions on the horizontal plane (i.e., points on a circle going around the listener's head). We integrated this with the physical model reverberator, and the 3rd-generation auralizer was agreed to sound better than either of its predecessors. We hope to continue experimenting with spatial filters in a future version.

Tracking User Movement

The current auralizer assumes a static listener and static sources, the first extension beyond would be to track the listener's head position and update the geometry of all sources on any listener motion. This implies, of course, that we either update the geometry at a high rate, or that we can interpolate between settings of the filters, mixer, and reverberator. It also implies a much larger compute requirement (depending on the performance of the geometry engine).

Moving Sources

To allow for continuous source movement, we need an API for source "clients" to register motion commands, and an interpolation method for geometrical data. (See the performance-related comments above.)

Streaming Sources

The auralizer we have implemented for ATON uses memory-based sample playback for sound production. We should look into streaming sound sources, whereby a separate buffer layer caches input sound, and can call the source's thread when it needs more input (possibly using the smallest possible buffers). This streaming could be done via standard CORBA, CORBA A/V, standard UDP sockets, of over FireWire. We have developed a simple CORBA sound file player to test whether this is reasonable.

Distributed Clients

This whole architecture is intended to support complex systems where a single acoustical space can have several independent processes triggering or generating sounds, and several users with different kinds of auralizers listening in any particular world.

The Eventual Design

In the future, we would like to evolve the current design into a scalable CORBA-distributed auralizer that can support many sources and many channels of output. We have discussed several other extensions, such as running the entire system in the frequency domain, so that all reverberation and filtering is combined into a single convolution problem that can be done using a CORBA “FFT/IFFT server.”

The Figure below shows some of the extensions we envision, focussing on the CORBA IDL that will replace our current procedure call interfaces, and the addition of source and listener motion tracking.

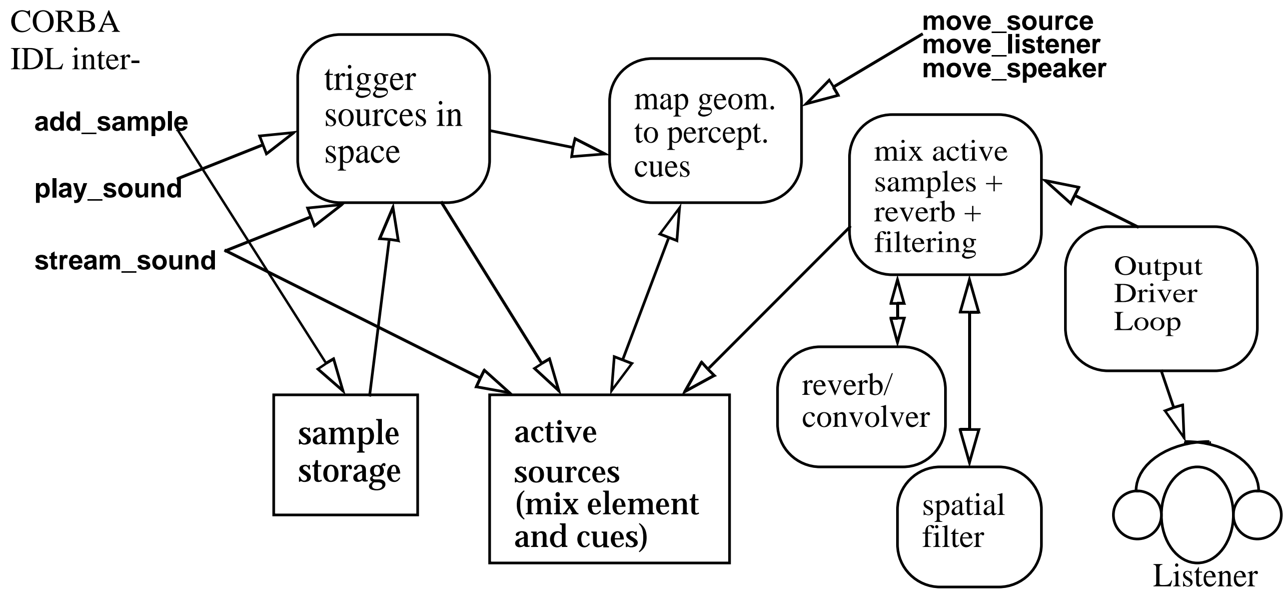

Figure: A Future Distributed Auralizer