**The Sound Pixel Project**

**Will Starms, Kurt Ehlers, and Alex Spiva**

Department of Computer Science,

University of Missouri – Columbia, Columbia, Mo 65211

July 25, 2012

**Abstract**

Traditional surround sound speaker systems are based on arranging two to eight speakers around a listener. To create an apparent sense of location of certain sounds, the same stream is played at different volumes in each speaker. This project aims at creating a large array of speakers, each capable of playing a unique audio stream, creating a genuine sense of sound location. To send audio to specific speakers, or clients, in the array, they must first be connected on a central server; this is done through broadcasting the server address information across a network. The audio files are then sent through a mixing program designated to control the audio flow between speakers while also preventing clipping, or the distortion heard when mixed audio surpasses the maximum threshold. The clients then connect, the server pushes configuration data, and audio playback begins. A custom frame was built to hold the prototype speaker array. This frame was required to be lightweight, easily assembled and disassembled, sturdy, and easily modified. All of these steps have been completed, and currently the project is working on testing and implementation of these features.

**Introduction**

The Sound Pixel Project aims to create a new way to experience audio. The goal of this project is to create an array of individual network-addressed speakers managed by a central streaming server that creates a directional sound experience. If you look at TV, for example, it is a system of light emitting pixels that function independently but come together to form a unique picture. This project applies that concept to audio by using independently functioning speakers that forms a “sound image.” In order to accomplish the goals of this project three aspects are addressed: constructing a frame to mount the speakers, mixing audio between multiple channels, and networking communication between the server and the speakers.

**Methods**

The design characteristics for the frame are to be transportable, sturdy, and easily constructed. To this point, we settled on 80/20 Inc.’s Industrial Erector Set components for their dependability, flexibility, and durability to be primary building material. The 80/20 components are made of aluminum alloy composite that utilize T-slot framing that allows ease for reconfiguration and does not require welding. The material is also available in 4’ segments allowing the frame to be enlarged and reduced in size for future projects. To connect the beam segments a three-way support is used to provide extra reinforcement at each joint. Currently the frame is 4’x4’ but is capable of 16’x12’ but further expansion requires extra leg supports. Cable wiring and cable ties are used to create vertical suspensions to mount the speakers. Currently the speakers in use are stereo formatted connected via a 3.5 mm jack and are powered with 12 volt adapters. To play audio over the speakers, a network consisting of a server, a client, and a properly configured audio subsystem is required.

The server manages client operations and configuration as well as aggregation and delivery of audio. On initialization, the server notifies all listening clients of its location and the port to connect to. Once one of these “beacon” packets is received, the clients extract the port information, merge it with the address data received from the packet header, and connect to the server where configuration begins. Clients send their ID number (which is tied to their location in the array), the server checks that it is valid, and responds appropriately. Once the ID has been validated, the server pushes key audio information such as the rate, the block size, and the endianness of the samples. The clients then spawns needed threads, prepare the audio subsystem, and initialize buffers and other data structures. The server does the same and also loads the individual audio files into memory for fast access. From here, both client and server have two threads that loop until the server completes transmission and sends the shutdown signal to the client. In the server, the processor thread takes the audio data for each client and packages it into packet-sized chunks in one page of the main buffer, and once that has been done for all clients, moves to the next buffer page, using mutual exclusion locks to keep the threads in sync and under control. The streamer takes these packets from a page, sends them off to the correct client, and moves to the next page. In the client, the streamer receives a packet, files it in the main buffer, and moves to the next page. Once the playback buffer has enough room, the processor feeds the packet to the audio subsystem, and moves to the next page. Once the server has finished sending all files, the processor calculates the amount of playback time remaining, sleeps, and returns, which activates the shutdown system.

The beacon system uses standard UDP broadcasting to reach the clients for initialization. While this should be fine for most implementations, broadcasting is not ipv6 compatible and would need to be replaced with multicasting. In the early versions of the server, the clients used two streams, one TCP and one UDP. The TCP stream was used for configuration, and the UDP stream was used for audio. This system was replaced with a single TCP stream for two reasons, to reduce the number of total connections, and to utilize flow control. The first few server iterations sent a single audio sample per packet that unintentionally resulted in 80% of each packet being overhead and immensely reduced transmission speed. This speed loss, however, was an unknown boon at the moment, as the client could very easily keep up with the stream and process it which, at the time, was just writing to file. Once actual audio playback was implemented, the stream was incapable of delivering audio fast enough to actually play it back. The speed issue was finally identified, and the packet size was increased to 1024 bytes, but since the stream was being done over UDP, once all the buffers were filled in the client, additional packets were just dropped, which resulted in three second bursts of audio before skipping ahead. Once the stream was converted to TCP, flow control fixed the buffering issues and as a side effect eliminated the issue of dropped packets completely. When a connection is established and configuration completes, the stream is only used for audio data and any control data is sent as an out-of-band byte code.

In early versions of the client, SDL (Simple DirectMedia Layer) was used for audio playback as it has a very simple API. SDL, however, was not capable of individually addressing the left and right channels of the speakers. This was briefly patched by considering each client as a set of speakers and sending interleaved audio (a type of stream that alternates left and right channel samples), but this was scrapped as it was overly-complicated, SDL had too much overhead for playback, and SDL did not provide any low level hardware control. ALSA (Advanced Linux Sound Architecture) was then investigated, as it is low level and very powerful. ALSA, unlike SDL, is capable of individual addressing once an external configuration profile is setup. In this profile, there are two main components, the route device and the mixer device. The route device acts as a playback device and simply routes the given input stream to another stream on another device. The mixer device is used to compile all the rerouted input streams into one stream sent to the soundcard. For example, to access the fifth speaker in the array, instead of having to work with complicated interleaving, the client sends the audio to the route device associated with the fifth speaker, and it gets sent to the mixer as input on the fifth channel.

For future projects, an audio mixing program was developed to enable audio files to “flow” between sound emitters. To begin working on mixing multiple music tracks a default track setting had to be established. In general, digital music has two main components, sampling rate and bit depth. Bit depth describes the number of bits of information recorded for each sample while sampling rate refers to the number of samples per second recorded. The project aimed to focus on music quality so all the tracks were a signed 16-bit depth, meaning the sample’s value ranged from -32,768 to 32767. Also, each track had a sampling rate of 44100 samples per second, again to ensure quality. With a base established all the tracks are then converted into RAW format giving the individual tracks no header, which requires less ciphering through the file. When processing through the files simultaneously, the general idea was to add the samples together to create a unified sound. But when both files add up past the bit depth’s threshold the result is called clipping giving a distorted sound. In order to prevent clipping an algorithm has been put in place to find the max added value and divide the rest of the samples by it. This sets the max added value to the highest sample allowed before clipping and decreases the rest by the same ratio.

**Conclusion**

The Sound Pixel Project successfully plays audio tracks through individually addressed speakers. Currently, each sound emitter must be loaded with an individual audio file before being sent by the server. Meaning, all audio mixing must be completed before any audio plays. Presently this project is at a basic state but it creates a platform to be built upon with future projects. New undergraduate studies could start implementing real-time mixing where audio can move between sound emitters while playing. Applications can be developed to show a layout of the speaker grid with the capabilities of “dragging” music to the particular sound emitter. Further into the future plans when the grid is much larger, Xbox Kinects can be used to track the motion of a person walking by the grid allowing the sound to follow the listener.