cs780
November, 2004

Chris Nelson
chrisn1@bu.edu

Game / Music Interaction

Explorations in Symbiosis

Advances are being made in the way computer-based games utilize music. Game music has the potential to be so much more than a passive element of the background. The music can and should affect the gameplay. The gameplay can and should affect the music. The player's actions can and should influence the direction and evolution of the music. By tightly linking gameplay and music, the player becomes much more immersed in the experience, and new creative possibilities abound for the developer. By collecting and creating examples of how this interaction might take place, and by providing a hardware accelerated method for dealing with the expense of interactive audio processing, it is this author's hope to give the reader a basis for creating musically interactive games, and discovering entirely new ideas in this field.

1. History

One of the oldest examples of this symbiosis is in the 8-bit Nintendo game, Super Mario Brothers [1]. When the player is low on time to complete a level, the game doubles the speed of the music. This innovation was as effective as it was simple. By making the music sound rushed, the developers instill a great sense of urgency in the player, as they hurry to beat the clock. From the perspective of the audio library, merely adding a bit of code to allow for the doubling of a song's speed gives each and every song in the game that much more added depth and variance. As an audio library allows for more and more modifications to the music, the music becomes much more than just an element of the background.

Another example comes from a more recent game, .Hack [2]. When exploring a dungeon, the music is ambient and atmospheric. If an enemy is encountered, a percussive

track is layered on top of the first track, since combat is about to take place. Both tracks are from the same logical song, and thus in sync with one another, but the percussive track fades in and out as the current mood of the player, or perhaps the character they're controlling, goes from relaxed to intense, depending on whether combat is occurring. The important aspect of this example is that music isn't simply a single waveform. It's a combination of waveforms, which can be divided into categories of varying granularity, from broad categories like percussion, all the way down to specific instruments. Since the audio library for .Hack respected this fact, the designers were able to eliminate the auditory discontinuity of switching between one song for exploration, and another for combat. Instead, continuity was preserved, and the music flows and morphs itself to fit the scenario of the moment, without interruption.

A 3rd example is a game called Rez [3], which is, in truth, the inspiration for this paper. As the player flies through each level, the music loop evolves as each waypoint is reached. The real magic, however, is that every last sound effect is quantized to the beat of the music. A new sound effect set is loaded for each song / level, and each sound effect set is designed to be the building blocks that fit on top of the foundation that is the current level's music. Furthermore, as the player fires missiles at enemies, they explode in sync with the beat. The game instills an amount of synaesthesia in the player, as they learn to associate the visual aspect of an enemy with the audio element it produces when destroyed. Everything in this game happens in sync with the beat, and a great deal of the music is generated by the player's and enemy's actions. This bears repeating. The final musical output is a collaboration between professional musicians, game designers, the player, and enemy AI. And each gaming session generates a new version of the music.

A very different example is Konami's Dance Dance Revolution [4] series. Players stand on a specially constructed dance pad, and then must step on specific parts of it, in sync with the music, to perform a unique dance for each song. In contrast to Rez, where the game maps the player's input to the beat, the major gameplay element is the player mapping their own input to the beat of the game. The more one is in sync with the rhythm, the higher one's score. Konami has released countless other rhythm games, varying the input device to instruments, such as guitars or drums. The gameplay,

however, remains fairly constant. The success and popularity of this franchise is definitive evidence that players find the creative linking of music and gameplay to be very compelling.

A final bit of history has absolutely nothing to do with games, yet provides us with many key features for a musically interactive audio library. The art of DJing is essential to study [5], as these musicians have found techniques which are imminently applicable to video games. To solve the problem of the discontinuity that arises between two songs, DJs use beatmatching and frequency equalization to create hours of music which sounds completely continuous, as songs are carefully weaved together. Beatmatching is the process of modifying the playback speed of the next song (via pitchbending), so that each measure of the next song is as long as a measure in the song that's currently playing. Once two songs are beatmatched and synchronized, the DJ will crossfade from the current song to the next song, not just by varying the volume, but also by varying the amplitude of coarse frequency bands, usually 3 or 4. The goal, during a transition from one song to the next, is often to keep the combined volume and the amplitude of each frequency band more or less constant, to give the perception of continuity. A skillful DJ often can preserve continuity to the point where the audience doesn't know when distinct songs are beginning and ending. Equalization can also be used to emphasize or remove certain elements of a song, such as the bass line. Another electronic music technique is granular synthesis, which is the process of creating sound or modifying music by repeating the most recent ~1-100ms of audio many times per second. Pick up a CD by Venetian Snares [6] if you desire to hear examples of this effect. Less exotic modifiers such as reverb (echo) and resonance (spiking the amplitude of specific frequencies) are also valid targets for inclusion in an audio library. DJs techniques to control and modify audio have been maturing for well over 20 years, and game designers should approach their game's music with the mindset and tools of a DJ.

2. A Musically Interactive Audio Library

The history laid out so far is merely a slice of the history of game / music interaction. This slice, however, gives us motivation. In each example, a game has been

improved by a unique set of features made available in its dependent audio library. Popular audio libraries, however, provide only the most basic capabilities, and thus atrophy the musical creativity of the developers who use them. The author of this paper became so frustrated with the limitations of SDL_mixer, one of the more popular audio libraries, that he spent substantial amounts of time on the production of a musically interactive audio library, to replace SDL_mixer. Two of the author's programs, a musically linked game and DJ software, drove this library's development. The following is an overview of the features associated with this library, and planned future directions.

2.1 High Level Design

C++ was chosen for this library, because of it's ability to abstract away complexity into objects with comparatively simple interfaces. `SoundObjs` and `MusicObjs` are the two objects the programmer is exposed to. They have many methods in common, and thus should both be descended from a pure virtual audio object, in a later version of the library. `SoundObjs` are to be used for sound effects, and `MusicObjs` are to be used for music. `SoundObjs` are loaded entirely to memory in the constructor (possibly in a forked thread), but `MusicObjs` are streamed from the disk as chunks are needed. The reason for this is that sounds tend to be small, and played back several times, so we only want to load them once. Music is usually only played back once, so by distributing the decoding over the length of a song, we minimize the amount of work and memory required per frame, to deal with a song. At the moment, music is streamed from disk, but loading an mp3 or otherwise compressed song entirely into memory would decrease the delay associated with accessing the data, at the cost of increased memory use.

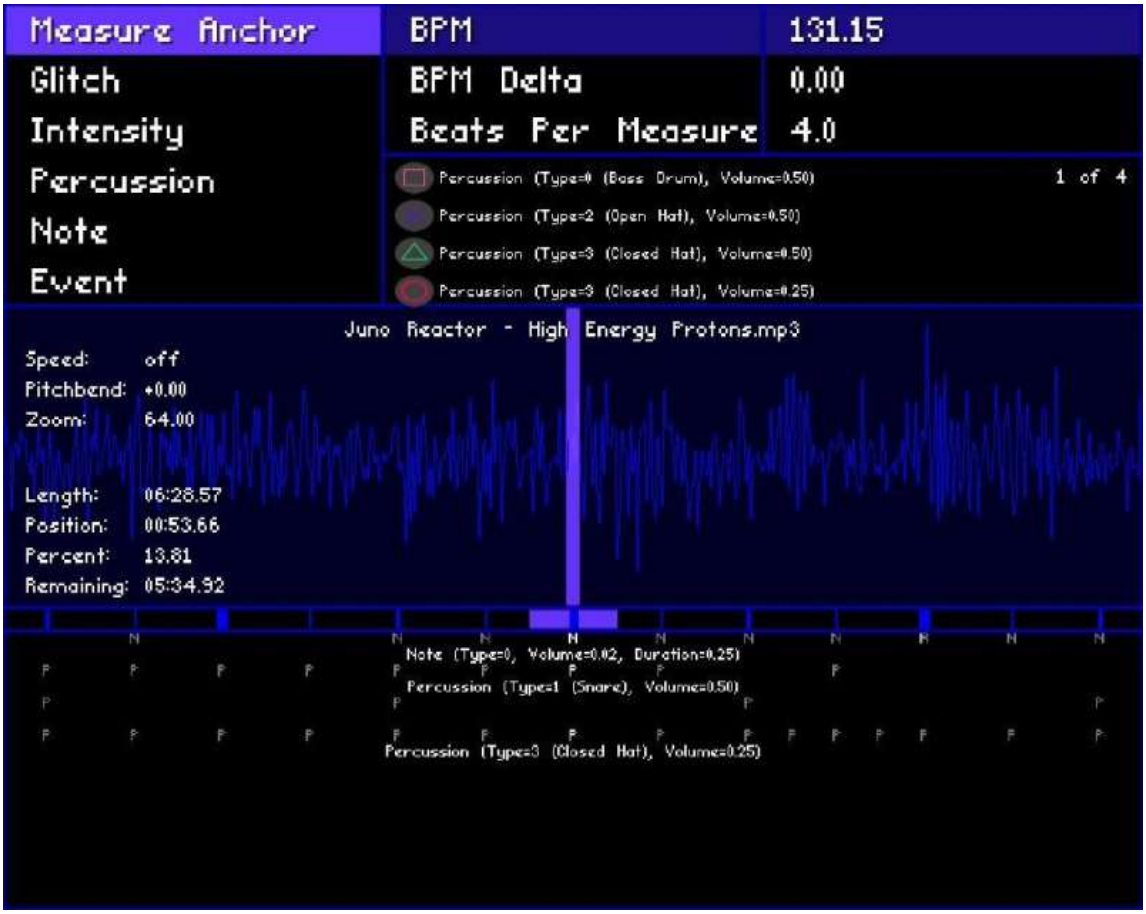
Both `SoundObjs` and `MusicObjs` share several methods. Basic functionality, such as `Play()`, `Stop()`, `Pause()`, `Resume()`, `Fade[In|Out]()`, `[Get|Set] Volume()`, and `[Get|Set]Position()`, is provided for both types of audio objects. More advanced methods are also available, and will eventually be implemented for both `SoundObjs` and `MusicObjs`. These include `SetSpeed()`, `SetSpeedInterpolationFactor()`, `SetGlitchAttributes()`, and a plethora of `MetaSync_*()` functions, which we'll explain shortly. The `SetSpeed*()`

methods deal with pitchbend, playing the audio back at a different rate, and smoothly interpolating between two different playback rates. These methods are essential in providing DJ-style control of audio. `SetGlitchAttributes()` deals with granular synthesis, and allows for varying the number of samples repeated, the speed at which they're played back, and whether the granular synthesis replaces or augments the audio currently being played. Frequency equalization is implemented, but not yet exposed through the API.

The `MetaSync_*()` functions are for use with music, primarily. They allow the designer of a game to get answers to questions about music metadata, like “Was a bass drum hit since the last frame?”, “Have we reached a new half-beat since the last frame?”, “What is the current beat of the measure?”, “What is the current BPM?”, “Is there a snare roll going on?”, and “How far into the current beat are we?”. Or, to put that briefly, they inform the program of every last aspect of the music. When a `MusicObj` is initialized, it will look for the song's MetaSync file, which holds all this data. Each frame, the audio library updates the song's MetaSync state, as music is pushed to the soundcard. Exposing the song's musical structure to the programmer allows for gameplay elements to be linked to the beat, the percussion's rhythm, or even specific instruments. Furthermore, players could drop in different music with different MetaSync data to affect the gameplay in unexpected and unplanned ways. Each song would make the game play differently, and as a result add value to both the music and the game.

2.2 Generation and Use of MetaSync Data

Generating MetaSync data can be difficult. Ideally, it could be generated algorithmically. Data must be collected, first, so that this algorithm has a basis for deciding what qualifies as a “snare drum”, or “crash symbol”, and how to differentiate a bassline from a bass drum. To collect this data, a tool was designed so a human could quickly describe a song. The resulting program is a modified version of the author's DJing software (more on this later). A 5 minute track was described completely in approximately 5 hours.



MetaSync data generation software that uses a Playstation2 controller as the input device.

The process of using the tool is quite simple. A song is loaded into a waveform (the middle row in the above image), which can be played back at various speeds. As the song plays back, it is the user's job to press buttons which will indicate the presence of a certain piece of metadata as they audibly and visibly pass under the metaphorical record needle (the vertical bar in the middle of the waveform). The timestamp of these button presses is quantized to a user-definable degree. One useful mode of playback allows a measure to be looped until the user decides they've completely described it. Specific instrument groups can be copied and pasted into and from multiple clipboards, which capitalizes on the repetitive nature of many types of music. The resulting MetaSync file format is a series of plaintext directives, each directive describing a musical element at a specific timestamp.

2.2.1 Automatic Generation of MetaSync Data

This tool, on its own, would be acceptable for use by game developers, but is more intended to gather data for an AI algorithm's learning routines, which will work as follows. By performing a Fourier Transform on a song's waveform data, in small chucks of samples, we are left with its frequency data, as it varies over time. If we map this frequency data to a grayscale image, by letting the x-axis represent time, the y-axis represent the frequency spectrum, and the intensity of each pixel represent the amplitude of a given frequency at a given time, we have now mapped the problem of music recognition to the problem image recognition, which has many well defined and tested algorithms.

Once we're in the image-domain, we can feed manually-defined MetaSync data into a neural network. If the neural net is trying to learn to identify the sound / image of a bass drum, then it will be fed vertical slices of described song images as input. If the slice has a bass drum in it, the correct output is 1. Otherwise, the correct output is 0. Repeat this process for each instrument we desire to auto-detect, and we'll have an agent that can create MetaSync data automatically. Increase the amount of human-defined music in the AI's learning set to increase its accuracy.

2.3 Frequency Querying / Modification/ Reaction

The audio library also allows a game to query and modify the amplitude of a range of frequencies, via Fast Fourier Transforms and equalization. This gives designers another set of information about the structure of the music. Aspects of the game can react to aspects of the music. As bass frequencies become increasingly prominent, the environment's lighting could become darker. The absence of midrange frequencies could make certain enemies change behavior. High frequencies, when present, could make a helpful platform appear, only to slowly fade away if the frequencies don't remain. The audio library gives developers opportunities to use the music's structure creatively, and to make their world's behavior dependent on the music, which in turn could be player-modifiable, allowing for increasingly varied gameplay. The player's actions could also

affect the music, via equalization. If you destroy a crystal, perhaps the high frequencies of the music are muted for 20 seconds, which, in turn, affects another element of the world.

2.4 Low Level Implementation

The heart of the audio library is the audio callback function. This function is called in a separate thread every time the soundcard's output buffer needs refilling. Linux allows for smaller output buffer, typically 1024 samples, providing lower audio latency. Windows typically requires a buffer size of at least 2048 samples. The author didn't have access to OSX and other Unix variants for testing. Each iteration of the audio callback function loops through every `SoundObj` and `MusicObj` that's currently playing. A nested loop then proceeds to mix data from each object's audio buffer into the soundcard's output buffer, processing the effects on a per-sample basis. A description of how some of the more experimental features work follows.

Pitchbend, the modification of a sound's playback speed (and indirectly, pitch), is controlled by the `SetSpeed*()` methods, and involves a few types of interpolation. For each audio object, a double precision floating point variable is used to keep track of the current sample that we're to copy into the output buffer. Audio libraries which don't support pitchbend would use an integer for this variable. We use a decimal because we recognize that, if we allow audio to play back at a different rate, the current sample often logically falls in between discrete samples. When this happens, we must linearly interpolate between the two neighboring samples, and copy the result into the next entry in the soundcard's output buffer. We then add the audio object's playback rate to the current sample variable, and then change the playback rate variable, if we're interpolating between two different playback rates, for the sake of continuity.

A typical playback loop looks like this:

```
for(int a=0;a<AudioObjects.size();a++)
{
     for(int s=0;s<outputBufferSamples;s++)
     {
          OutputBuffer[s]+=AudioObject[a].GetSample(s);
          AudioObject[a].NextSample();
     }
}
```

The difference between a standard audio library and a pitchbending one lies in the implementation of an audio object's `GetSample()` and `NextSample()` methods:

```
AudioObject::GetSample(double which)
{
#ifdef PITCHBEND
     float decimal=which-floor(which);
     return
     (
          (1.0-decimal)*Samples[(int)floor(which)]+
          (0.0+decimal)*Samples[(int)ceil(which)]
     );
#else
     return(Samples[(int)which]);
#endif
}
```

```
AudioObject::NextSample()
{
#ifdef PITCHBEND
     NextSample+=PlaybackRate;
#else
     NextSample++;
#endif
}
```

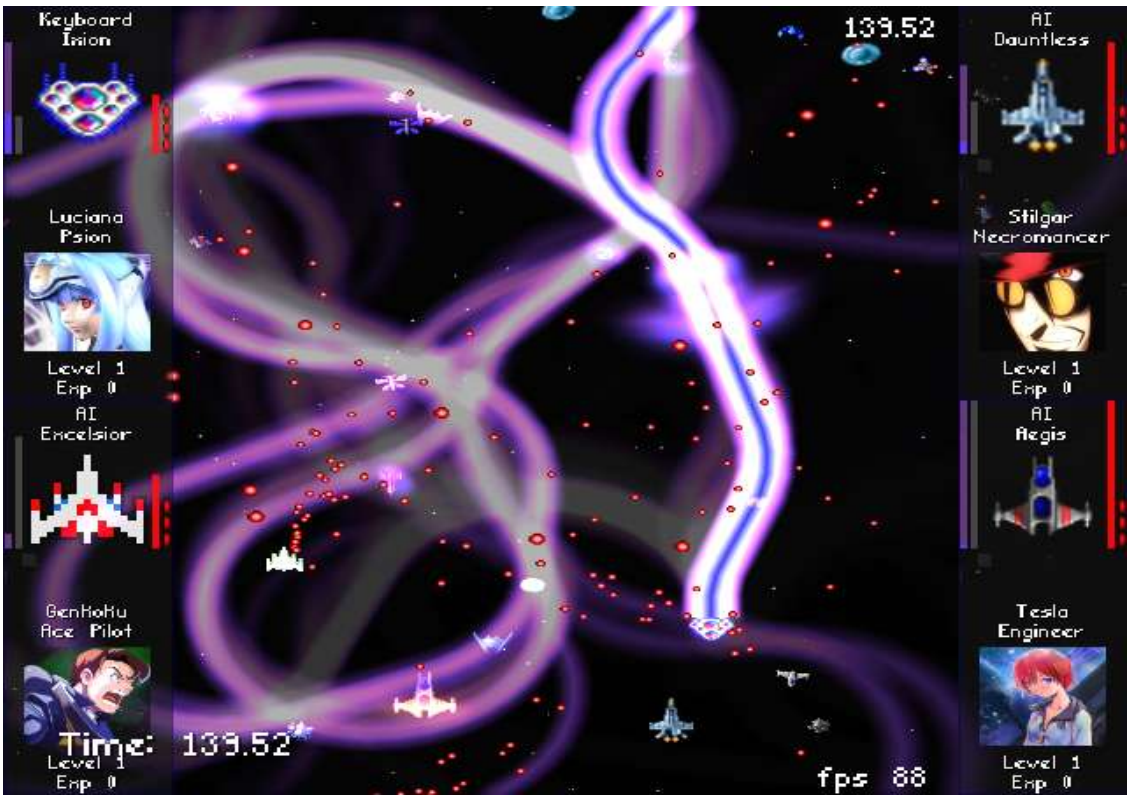
Although the goal of these functions is to modify merely the playback rate, it has the side effect of modifying the pitch as well (thus the name “pitchbend”). This is actually desirable, because it correctly simulates the behavior of a record, the medium DJs use.

MetaSync data is loaded into an audio object during its constructor. MetaSync data is sorted by increasing time, when it's generated, which will be useful. After a set of samples is copied into the soundcard's output buffer, during a loop of the audio callback function, we check to see if the next unprocessed MetaSync directive's timestamp coincides with data we just copied into the buffer. We then update the audio object's MetaSync state to respect this directive, and check for addition directives to be processed this loop. Semaphores are used to lock down MetaSync data, due to the fact that the audio callback loop exists in its own thread. From the user's perspective, the MetaSync data is updated only once per graphical frame, as the user's code typically loops on a per-frame basis.

This audio library is still in development, but it currently provides all the major features of SDL_mixer, as well as some important music / game interaction technology. Future improvements will include multi-track music, as found in .Hack, GPU acceleration (see section 4), and a re-vamped API. Once this is complete, it will be released to the public, under the GNU LGPL license, and hopefully give rise to some creative games, as a result.

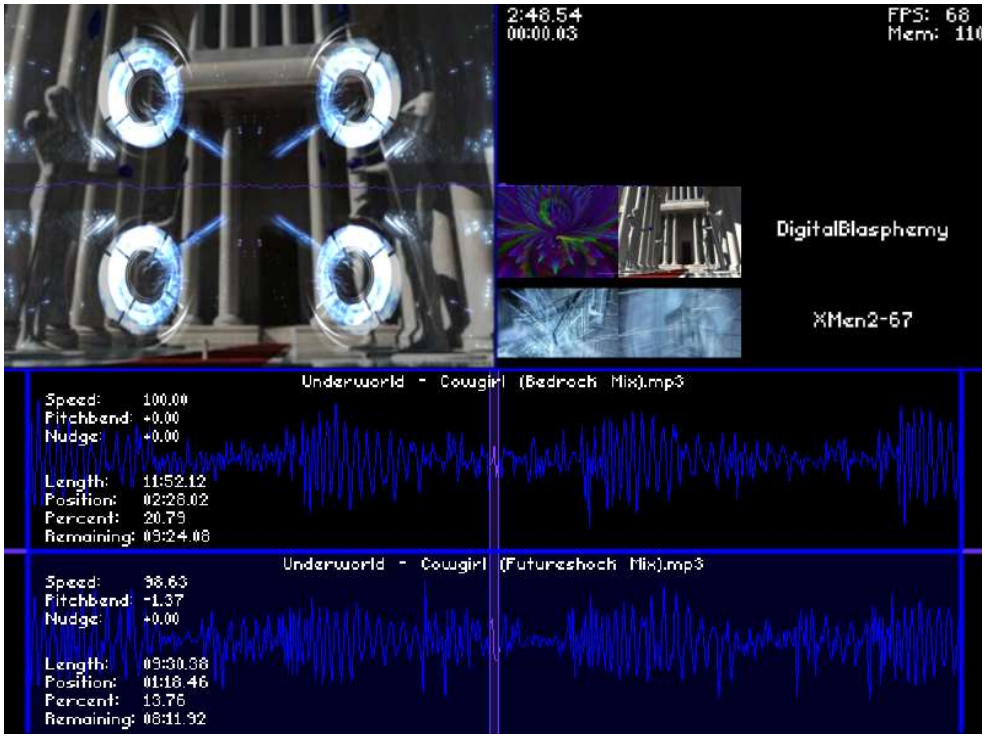
3. Example Applications

A motivation for writing the audio library was the desire to be able to write musically interactive games. One of the author's works-in-progress was chosen to be extended to use this new audio library. The game in question is a 2d spaceship shooter, similar in gameplay to Space Invaders. At the end of this experiment, enemies fired bullets when the bass drum kicked, making certain sections of the song safe, and others exceedingly dangerous. Certain weapons that the player used were triggered by certain instruments, so if the player uses the right weapon during a snare roll, more damage is output. Only a small fraction of the audio library has been used thus far, but already the game has a definite link to the music.



Red enemy bullets pulsate with the beat of the music,
as a homing laser fires in sync with the music's bass line.
None of which you can see, really, because this isn't a video.
Contact the author for a video.

Another application the author developed for use with the audio library was DJing software. The audio library provides all the functionality necessary to simulate a DJ setup, with equalization, beatmatching via pitchbend, crossfading, and scratching (via the `SetSpeed*()` functions). By combining the DJ capabilities with the MetaSync data, it's theoretically possible for a game to beatmatch two tracks, and crossfade between them, as the player progresses from one level to the next, preserving the music's continuity. This is planned for the aforementioned spaceship shooter.



Two remixes of an Underworld track are visibly beatmatched.
The high-amplitude sections of the lower two waveforms are bass drums,
which are lined up with one another, so the user knows everything is in sync.
The bottom waveform has a pitchbend of -1.37%,
so it plays back at the same speed as the top track.

4. Hardware Acceleration: Using The GPU As An APU

Certain soundcards advertise hardware audio processing capabilities, but each card has a different set of features, and there is no universal audio acceleration interface, like OpenGL for graphics. Until this changes, the acceleration provided by soundcards is of interest only to developers releasing games for fixed targets, such as consoles.

Processing and modifying audio with the CPU, on the fly, is computationally expensive. The more flexibility you give the designers by breaking a song into more and more component tracks, the more work you must do to mix them all back together. Modifiers such as equalization and pitchbend require further computation. As it turns out, however, the problem of audio processing can be mapped to the problem of graphics processing. This allows the vast majority of work to be offloaded to the graphics hardware, saving valuable CPU time, performing the calculations in a shorter amount of actual time, and potentially allowing for increased audio responsiveness and decreased latency.

Recently, many computer scientists have researched ways to harness the power of specialized graphics hardware for use in other types of computations. Although CPUs are decent at all types of computations, GPUs choose a few operations, and do them blazingly fast. Many types of audio processing can be mapped to these operations, by loading an audio signal, a chunk at a time, onto a texture. Once there, we can draw these textures to an off-screen buffer in specific ways to obtain the results of operations such as multi-track mixing, amplitude scaling, pitchbend, reverb, equalization, granular synthesis, and more.

Audio signals shall be defined as a continuous ordered set of amplitude data. 0.5 amplitude shall represent silence, and 0.0 and 1.0 shall represent the extremities of the negative and positive amplitude range, respectively. Audio buffers shall be the digital representation of audio signals, and can exist in both system memory and GPU texture memory. 16 bits per sample is the precision used to store music on audio CDs, and is

standard on all soundcards, so our audio buffers shall also use 16 bits per sample / texel. Recording studios may desire to use 24, 32, or even more precision, but 16 bits should be sufficient for most gaming purposes.

There are many possible formats in which to store audio buffers, when they're loaded into the GPU's texture memory. We shall use 32-bit `GL_LUMINANCE16` textures, as they have the desirable properly of providing just the right amount of precision per texel /sample, 16 bits. Two of these textures (or modifications of these operations to work with `GL_LUMINANCE16_ALPHA16` textures) are necessary for a stereo signal. For simplicity, we will also assume our graphics hardware has OpenGL 2.0 Shader Language (GLSL) capabilities. It's quite possible to do almost everything described herein without shaders, but they allow for the cleanest description, and are quickly becoming the industry standard, so we'll allow for their use.

4.1 Mapping Audio Operations To OpenGL

Multi-track Mixing is the simplest operation at our disposal. To add 2 audio buffers together, we use the graphics hardware's blending functionality. For each pixel in our result buffer, the following calculation is performed:

```
Result[n] = Buffer0[n] + Buffer1[n] – 0.5
```

Subtracting .5 is necessary, because we're dealing with a number system whose base value is .5, instead of the typical 0.0. We'll see many variations of this necessity, throughout the operations. Without a pixel shader, one could blend the two buffers together at 0% opacity, and then draw a black line at 50% opacity over the result.

Amplitude Scaling is another very simple operation. We merely add to silence the product of the amplitude scalar by a sample's distance from the silence amplitude, 0.5:

```
Result[n] = 0.5 + Scalar * ( Buffer[n] – 0.5 )
```

Without a pixel shader, we require a few passes. First, fill our result buffer with silence (0.5). If we're scaling down the amplitude, make a single call to glColor3f (Scalar,Scalar,Scalar), and then additively draw the source buffer into the result buffer at 0% opacity. If we're scaling up the amplitude, draw the source buffer into the result buffer at 0% opacity, subtracting 1.0 from the scalar until the scalar is in the range [0, 1.0). Then, make a call to glColor3f(Scalar,Scalar,Scalar), and draw the source buffer one last time into the result buffer. The amplitude can also vary over time. Merely add a vertex with a different glColor() value wherever you want a control point, to allow for hardware accelerated fades-ins and fade-outs.

Pitchbend relies on the stretching that occurs during texture mapping. A pixel shader is not needed for this operation. If you have an audio buffer of 1000 samples that you desire to speed up by 10%, simply draw it into a new buffer of size 900. If you wish to change the pitch/speed of a buffer over time, control vertices paired with calls to glTexCoord1f() become necessary. Negative pitchbends, which result in a reverse playback, are also possible. The following bit of pseudocode illustrates an implementation OpenGL-accelerated +8% pitchbend.

```
char* AudioBuffer;
long AudioBufferSize=LoadFileToAudioBuffer
(
     “Adam Freeland – Mind Killer.ogg”,
     AudioBuffer
);

//Load 1024 samples of the AudioBuffer to a glTexture
int glTexture;
glGenTextures(1,&glTexture);
glBindTexture(GL_TEXTURE_1D,glTexture);
glTexImage1D
(
     GL_TEXTURE_1D,
```

```
     0,                        //Level of Detail
     GL_RGBA,                  //Texture's Internal Format
     TexW,TexH,
     0,                        //No Boarder
     GL_RGBA,                  //Data Format
     GL_UNSIGNED_BYTE,         //Data Type
     AudioBuffer               //Data Location
);
glTexParameteri
(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glOrtho(0,640,0,480,0,1);
glDrawBuffer(GL_BACK);
glReadBuffer(GL_BACK);
glBegin(GL_LINES);
{
     glColor4f(1,0,0,1.0/128.0);
     glBindTexture(GL_TEXTURE_1D,glTexture);
     glTexCoord1i(0);
     glVertex2f(0,0);

     glColor4f(1,0,0,1.0/128.0);
     glTexCoord1i(SAMPLES);
     glVertex2f(0.92*SAMPLES,0);
}
glEnd();
glReadPixels(0,0,SAMPLES,1,GL_RED,GL_FLOAT,OutputBuffer);
```

Reverb, or echo, can be achieved by drawing the source buffer into the result buffer several times. The first pass is a bitwise copy, but the remaining passes are offset by an ever increasing amount, with an ever decreasing amplitude scalar.

Equalization, the scaling of the amplitude of specific frequency bands, is a slightly different operation, which is applied to 2D frequency buffer textures instead of 1D waveform buffer textures. Frequency buffers can be obtained by performing a (Fast) Fourier Transform (FFT) on a standard audio buffer. An inverse FFT will reverse the process. Once you have a frequency buffer texture, draw it to the result buffer with calls to glColor3f() for each desired control vertex. A pixel shader isn't necessary for this operation, but would be exceedingly useful if someone devised a way for it to compute the FFT of a waveform texture.

Granular Synthesis, the repetition of a small chunk of audio to produce sound at a potentially varying speed, uses the texture-repeating capabilities of the graphics card. Merely repeat a portion of the texture at the desired interval over a period of time.

Any audio library that uses GPU acceleration should allow for arbitrary combinations of these operations. Persistent audio buffers, such as sound effects, will see the greatest benefit from this acceleration, because they only need to be loaded into texture memory at the beginning of a game or level. For music, one must weigh the cost of streaming over chunks to the graphics card versus the benefit of accelerated effects processing, if we load the entire song into GPU memory. To calculate whether this is worth it, we need experimental data.

4.2 Benchmarking: CPU vs GPU

A simple benchmarking program [A] was written to test whether the GPU is faster than the CPU for audio processing, and if so, by how much. The system used in this benchmark was a 1.8Ghz Pentium 4, with a GeForce4 440 Go/AGP/SSE2. When mixing 128 raw signals together (1024 samples at a time), without any effects, the GPU gives, at

best, a 2x performance boost. Not too impressive. However, when we add a single effect, a +8% pitchbend, the results change substantially. The GPU performed 50x to 100x better, depending on whether the the audio samples reside in GPU memory (in the case of sound effects), or need to be streamed there every 1024 samples (in the case of music). Regardless, the GPU is clearly better suited to compute pitchbend.

It gets better, too. If we were to add more audio processing operations into the loop, the author believes that each cumulative operation would increase the GPU's performance advantage by an additional 50x to 100x, assuming that these operations are of comparable CPU vs GPU complexity. CPU processing time would increase linearly, while GPU processing time would remain much closer to constant, as most effects only require a slight variation in the drawing loop, which has no effect on amount of work the GPU ends up doing to perform the actual drawing. Furthermore, if we're happy with the 50x per effect speed boost, then we don't even need to keep any sounds or music loaded into the GPU's texture memory. Rather, we just use a single 1D texture (usually 1024 texels), and stream data into it as we need it processed. If we draw into an off-screen buffer, then we have almost no impact on the primary activities of the GPU.

One issue with this method is that some systems don't support calls to OpenGL functions in spawned threads, such as the audio callback loop thread. The audio callback loop thread, in this case, would have to access OpenGL somehow, possibly through a forked child process. Another issue is that we need to quickly grab the GPU about 40 times per second, and then release it and let the program pick up drawing where it left off.

Although, in the vast majority of contemporary games, audio processing is not a bottleneck to performance, future games which utilize musically interactive audio libraries could benefit greatly from acceleration.

5. Conclusions

Gaming history shows us the important effects that music can have on the gameplay experience, when it is actively linked to the world, rather than a passive

element of the background. Super Mario Brothers, .Hack, Rez, and Dance Dance Revolution all illustrate different ways music can interact with a game.

Game designers and programmers are bound by the tools they use. The most popular publicly available audio libraries don't give designers the tools they need to exercise their creativity in the field of game / music symbiosis. The primary goal of this independent study was to produce a musically interactive audio library which would give designers read / write access to the underlying structure of the music they use in their games. The sample applications demonstrate that an amount of success has been accomplished, but work remains before the audio library is suitable for release.

As audio processing becomes increasingly powerful, it also necessarily becomes increasingly computationally complex. To deal with this, we introduce the concept of using the acceleration provided by graphics hardware for audio mixing and effects. By offloading processing from the CPU to the GPU, we can perform more calculations in less time, since we can map the problem of audio processing to the strengths of modern graphics cards. Given the flexibility of modern pixel shaders, many audio effects which weren't described in this paper can likely be performed in graphics hardware by a clever programmer. Perhaps these techniques could be useful for live electronic music performances, or any other complex real-time audio task.

6. Future Work

This field is anything but saturated. At present, there's no publicly available audio library that provides an acceptable level of musical interaction. The author's audio library is neither complete nor released. The author intends to add support for music comprised of multiple tracks, revise the API, and utilize GPU acceleration before publishing his audio library.

This paper only describes a subset of the audio processing operations that can be performed by the GPU. Work needs to be done to compile a complete list of audio processing operations, decide which ones are important for games, and which ones have

no known GPU-accelerated method for computation. A framework for proving certain operations non-computable on a GLSL capable GPU might also be developed, should the implementation of an operation prove elusive.

The work presented in this paper will continue. The author will not rest until all game developers have access to tools for the production of musically interactive games.

7. Acknowledgments

This work wouldn't be possible without a number of entities. All work builds upon that which precedes it, that it may influence future endeavors. The mere existence of computer-based games is the result of countless people's efforts, the vast majority of whom go unrecognized and unknown. Thanks are due to all game developers whose creative risks advance the state of gaming as a whole. You know who you are [7]. Specifically, thanks is owed to the designers and programmers whose methods are referenced in this paper. Also, without John Carmack [8], hardware graphics acceleration wouldn't be where it is today.

A number of musicians also deserve credit. Those who compose using instruments or devices in non-standard ways are often recognized by only a few, but their innovations are what progresses music. From Lev Sergeivitch Termen's creation of one of the first electronic music instruments, the Theremin [9], to the contemporary work of Richard James [10], Aaron Funk [11], Ken Gibson [12], and Matt Anderson [13], all of whom push music in fascinating new directions, and the countless people in between, thank you for the inspiration. Adam Freeland [14] and everyone who worked on Rez's Area 5 deserve massive credit for creating one of the best levels in all of video gaming, which gave direct rise to the author's work.

Sam Lantinga's SDL library deserves credit as well. By releasing such a powerful tool under an open source license, countless programmers are empowered to become game developers. Everyone who's contributed to SDL [15], SDL_sound [16], SDL_image [17], SDL_mixer [18], and SDL_net [19] and their dependencies deserve thanks.

You would not be reading this paper, were it not for the guidance provided by my adviser, Burkhard Wuensche [20]. He allowed the freedom to explore any aspect of this field that was found interesting, provided helpful insight wherever the author's investigations took him, and helped edit the final paper.

Finally, so many thanks are due to a Kiwi who listens to my incessant babbling about computers and games, yet somehow puts up with me [21].

8. Works Cited

[1] Nintendo, Super Mario Brothers, URL: smbhq.com/smb.htm
[2] Bandai, .Hack, URL: dothack.com
[3] SonicTeam / Sega, Rez, URL: sonicteam.com/rez
[4] Konami, Dance Dance Revolution, URL: konamihwi.com/DDRPC/index2.php
[5] Sanctuary Publishing, DJ Techniques, 2003, Sloly, Frederikse.
[6] Venetian Snares, A Huge Chrome Cylinder Box Unfolding, Planet Mu.
[7] Toys For Bob, Star Control II, URL: toysforbob.com
[8] John Carmack, Wolfenstein, Doom, Quake, URL: idsoftware.com
[9] Leon Theremin, Theremin, URL: en.wikipedia.org/wiki/Theremin
[10] Richard James, Aphex Twin, Warp Records, URL: aphex-twin.com
[11] Aaron Funk, Venetian Snares, Planet Mu, URL: vsnares.com
[12] Ken Gibson, Eight Frozen Modules, Tigerbeat 6, URL: eight-frozen-modules.com
[13] Matt Anderson, Bit_Meddler, Planet Mu, URL: planetmu.com/artist01.html
[14] Adam Freeland, URL: freeland.fm
[15] SDL, URL: libsdl.org
[16] SDL_sound, URL: icculus.org/SDL_sound
[17] SDL_image, URL: libsdl.org/projects/SDL_image
[18] SDL_mixer, URL: libsdl.org/projects/SDL_mixer
[19] SDL_net, URL: libsdl.org/projects/SDL_net
[20] Burkhard Wuensche, Auckland University, URL: www.cs.auckland.ac.nz/~burkhard
[21] Seriously, thank you.

[A] A CPU vs GPU audio processing benchmark application:

```
/*
 * This program was tested on linux.
 * It isn't intended to be 100% correct.
 * Rather, it's intended to test the speed
 * at which the CPU and GPU perform
 * representative audio processing operations.
 *
 * Copyright Chris Nelson, 2004.
 */


#include <SDL.h>
#include <GL/gl.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

//<User Definable Options>

#define      SAMPLES                         1024
#define      LOOPS                           100
#define      SIMULATE_MUSIC_NOT_SOUNDFX      false

//</User Definable Options>

void  InitializeSDLGL();

float        SoundcardOutputBuffer[SAMPLES];
char         AudioChunk[128][SAMPLES];
Gluint       TextureGL[128];
```

```
int main()
{
	InitializeSDLGL();

	printf("\nProcessing 128 AudioChunks of %i 8-bit ",SAMPLES);
	printf("samples each, %i times, via CPU...\n",LOOPS);
	int BeginTicks=SDL_GetTicks();
	for(int loop=0;loop<LOOPS;loop++)
	{
		for(int chunk=0;chunk<128;chunk++)
		{
			float SampleNow=0;
			for(int sample=0;sample<SAMPLES;sample++)
			{
				float decimal=SampleNow-floor(SampleNow);
				int prev=(int)floor(SampleNow);
				int next=(int)ceil(SampleNow);
				SoundcardOutputBuffer[sample]+=
				(
					(1.0-decimal)*AudioChunk[chunk][prev]+
					(0.0+decimal)*AudioChunk[chunk][next]
				);
				SampleNow+=1.08;
				if(ceil(SampleNow)>=SAMPLES) break;
			}
		}
	}
	int TimeCPU=SDL_GetTicks()-BeginTicks;
	printf("CPU pitchbending and mixing took %ims\n\n",TimeCPU);

	printf("Loading 128 AudioChunks of %i 8-bit samples each ",SAMPLES);
	printf("to glTextures, once...\n");
	BeginTicks=SDL_GetTicks();
	glGenTextures(128,TextureGL);
	for(int chunk=0;chunk<128;chunk++)
	{
		glBindTexture(GL_TEXTURE_1D,TextureGL[chunk]);
```

```
	glTexImage1D
	(
		GL_TEXTURE_1D,
		0,			//Level of Detail=0
		GL_LUMINANCE16,		//Internal Format
		SAMPLES,
		0,			//Boarder=0
		GL_LUMINANCE,
		GL_UNSIGNED_BYTE,
		AudioChunk[chunk]
	);
	glTexParameteri(GL_TEXTURE_1D,GL_TEXTURE_WRAP_S,GL_CLAMP);
	glTexParameteri(GL_TEXTURE_1D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
	glTexParameteri(GL_TEXTURE_1D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
}
int TimeTextureLoading=SDL_GetTicks()-BeginTicks;
printf("glTexture Loading took %ims\n\n",TimeTextureLoading);

printf("Processing 128 AudioChunks of %i 8-bit ",SAMPLES);
printf("samples each, %i times, via glTextures...\n",LOOPS);
glEnable(GL_TEXTURE_1D);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,GL_ONE);
BeginTicks=SDL_GetTicks();
for(int loop=0;loop<LOOPS;loop++)
{
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	glFrustum(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);

	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	glOrtho(0,640,0,480,0,1);
	for(int chunk=0;chunk<128;chunk++)
	{
		glBindTexture(GL_TEXTURE_1D,TextureGL[chunk]);
		if(SIMULATE_MUSIC_NOT_SOUNDFX)
		{
```

```
                        glTexImage1D
                        (
                              GL_TEXTURE_1D,
                              0,                //Level of Detail=0
                              GL_LUMINANCE16,   //Internal Format
                              SAMPLES,
                              0,                //Boarder=0
                              GL_LUMINANCE,
                              GL_UNSIGNED_BYTE,
                              AudioChunk[chunk]
                        );
                  }
                  glBegin(GL_LINES);
                  {
                        glColor4f(1,0,0,1.0/128.0);
                        glTexCoord1i(0);
                        glVertex2f(0,0);

                        glColor4f(1,0,0,1.0/128.0);
                        glTexCoord1i(SAMPLES);
                        glVertex2f(0.92*SAMPLES,0);
                  }
                  glEnd();
            }
            glReadPixels
            (
                  0,0,
                  SAMPLES,1,
                  GL_RED,
                  GL_FLOAT,
                  SoundcardOutputBuffer
            );
      }
      int TimeGPU=SDL_GetTicks()-BeginTicks;
      printf("glTexture pitchbending and mixing took %ims\n\n",TimeGPU);
      printf("glTexture processing is %.2fx ",TimeCPU/(float)TimeGPU);
      printf("faster than CPU processing\n\n");
}
```

```
void InitializeSDLGL()
{
	if(SDL_Init( SDL_INIT_VIDEO | SDL_INIT_TIMER) < 0)
	{
		printf
		(
			"LGL_Init(): SDL_Init() failed... %s\n",
			SDL_GetError()
		);
		exit(-1);
	}

	SDL_WM_SetCaption("GPU-APU-Benchmark","GPU-APU-Benchmark");
	if(SDL_SetVideoMode(640,480,32,SDL_OPENGL) == NULL)
	{
		printf
		(
			"LGL_Init(): SDL_SetVideoMode() failed... %s\n",
			SDL_GetError()
		);
		exit(-1);
	}

	glDrawBuffer(GL_BACK);
	glReadBuffer(GL_BACK);
	glClearColor(0,0,0,0);
}
```