Interacting with Music in Video Games
Derek Dahmer <dahmerda@seas.upenn.edu>
Advised by Dr. Badler

Abstract

A musical track has been a part of nearly all commercially sold video games created in the past decade. Yet as essential as it is, the musical track is almost always relegated to the background, only affected by macro changes in the game such as the shifting of levels or substantial plot changes. A new concept emerging in the independent games genre has been making the musical component a substantially more prominent aspect of the game experience. Games such as Rez and Everyday Shooter have attempted this by tying user actions to changes in the musical track, as well as by showing certain characteristics of the active sound track in the surrounding environment. The result has been a markedly more immersive, expressive and unique experience for the player that has manifested in both game sales and critical acclaim. This project seeks to further the development of the focus on audio in two ways.

First is by expanding on the idea of having a sound track change as the result of specific user actions. In particular, the visual-audio connection of game events can be further strengthened, creating a distinct relationship for the player between potential actions and potential musical outcomes. For example, results of a game event such as having damage dealt to the player could be expressed by not only the current method of a sound cue for pain, but also by distorting the collection of other sounds that are currently playing in a negative sounding way. A distinct relationship between action and audio will serve to increase overall immersion in the game since audio manipulation has the potential to feel like more of a natural extension of gameplay than numbers on a screen or a prerecorded sound clip. In addition, the player will feel a stronger connection to game as it becomes apparent that each action choice has a controllable outcome on the previously uncontrollable sound track.

Second, this project will explore using audio as a means of giving the player additional information that will help or even be essential in non-musical gameplay decisions. In particular, states of the active entities in the game, such as players and enemies, can be represented through audio. For example, the sound emanating from a particular enemy can change when it notices and targets the player. The shift in focus to audio be achieved a number of ways. A clever means of visually obscuring the enemies would shift reliance of information to the audio cues in the
game. A major component of this project has been discovering effective ways to accomplish this goal in a side-scrolling shooter setting.

The game takes the form of a 2D side scroller, where the player is represented by a ship on screen that appears to be moving in one direction as enemies come at the ship from the other direction. This game type was chosen because it has an inherent flow and is relatively easy to implement. The simple design allows for quick changes to game mechanics which helps immensely with trying out new combinations of concepts enabled by the expanded sound manipulation capabilities. The game will be based on three fundamentals and be expanded from there. First, the game will strive for simplicity in basic movement and gameplay. Although the game will become progressively more complicated in strategy, a player should be able to pick up the controls and begin playing immediately with no explicit instructions. Second, the game should be as fluid an experience as possible, blurring the distinctions between stages to create an uninterrupted experience. Finally, as discussed before, the game will strongly focus on a two-way connection between user action and musical effects.

The goal is to create a need for the player to be conscious of the state of the audio in order to succeed in the game, the same way visual awareness is necessary in nearly every existing game today.

Related Work

A few games released in the last two years have overtly experimented with the connection between music and gameplay. Most notable of them in terms of innovativeness and popularity are Rez HD and Everyday Shooter. However, each falls short of the ideal that this project targets.

Rez HD

Rez HD is a rail shooter where the player takes the role of an evolving being flying through digital landscapes. The goal is to eliminate enemies that appear by locking on to them with the cursor and releasing to kill as large a combination of them as possible. The music centers around the development of an electronic music track by adding a distinct background beat to the overall track with each of 10 musical stages in each level. As the player progresses through the level, the overall track evolves from a simplistic initial beat to a full track of combinations of beats arranged by a professional electronica artist. Additionally, as enemies are hit, they each make a distinct melodic note that is on beat and fits in with the main track. Locking on and eliminating a few at the same time creates a more complex melodic sequence. The wireframe landscape in the background pulses with the beat, adding a visual connection between the environment and the music.
This game is a good model to work from, as it is a simple and fluid with a strong connection between the visual and audio aesthetic. However, Rez primarily uses a one-way dynamic as the changes in music do not affect any aspect of the game pertinent to the player’s goals. This holds the player back from experiencing a two-way interaction with the music. In addition, the game mechanics are static throughout the entirety of the game. Enemy organization differs, but the basic lock-on and shoot mentality stays the same. Ideally, the mechanics of the game would take on the same evolutionary characteristics as the music. Also, there is potential to add randomness to the music development rather than be focused purely on progressing through a premixed track.

**Everyday Shooter**

Everyday Shooter is a sandboxed 2D arcade-style shoot-‘em-up in which enemies are constantly appearing and moving towards the player. The goal is to stay alive for a certain amount of time each level by preventing the onslaught of enemies from shooting or colliding with you. Musically, the game is interesting because its' sound effect bank is made up entirely of guitar riffs and notes recorded by the game’s programmer. Each enemy that is defeated sets off a riff that melds with a background guitar track. A combo system is in place that allows for the player to destroy as many as 50-100 enemies in a chain at once. This kind of action creates an interesting polyphonic jumble of riffs to the overall auditory landscape. This is interesting because it adds an auditory “reward” to a player that accomplishes a large combo in addition to the gameplay awards. Visually, there are all kinds of strange, stimulating effects going on that reflect the randomness of the musical landscape. Each level has a different set of gameplay mechanics that the player must figure out in order to survive.

The interesting background music created by the players actions in the game is an interesting part of this game, especially in relation to how the game mechanics work. However, the connection between action and individual sound is weakened by the sheer number of riffs happening at any one time. This also make it nearly impossible to associate a riff with any one particular action in the game. This aspect takes away from the immersiveness and lowers the information each sound unit could potentially deliver. One major goal of this project is to create a clear association between action and a sound result so that the player can make gameplay decisions based on the kind of musical landscape they want to create.

**Technical Approach**

The code architecture created for this project is based on three major components. Graphics and gameplay mechanics are handled by the XNA 3.0 Game Studio framework, written in C#. All audio processing is controlled by a collection of
classes written in C++ that use DirectX’s XAudio2 library for audio manipulation. To bridge the managed and unmanaged code, the audio library functionality that is relevant to the core game components is exposed through a Win32 DLL. The following diagram lays out the rough chain of calls. Light blue nodes are components I have designed for this project.

XNA

Microsoft’s XNA framework is a collection of classes that provide an abstraction over the detailed control that applications written with straight DirectX or OpenGL have to maintain. Only the primary components of the graphical application are required to be implemented. For example, rather than a complex initialization function that requires the programmer to understand all the detailed settings that need to be enabled for the scene to render properly, XNA has an Initialize function that first calls a base class to take care of the specific settings. This allows the programmer to jump straight into designing the unique aspects of the application, such as initializing user-defined objects that will be used to represent objects (e.g. enemies or the player) in the application. In addition to initialization, the framework abstracts away much of the complexity of content loading, drawing, timing and asset management. The framework is designed to put the programmer as close to the creative part of the game design as possible. In terms of this project, this allows for quickly developing and changing the gameplay aspects as new ideas come along. This is a very important quality given the exploratory nature of the project. In addition, the framework is designed with encapsulation and inheritance in mind in every component. For example, the base Game class comes with Update and Draw methods that are overridden by the user’s specific game class in order to update and draw game objects. I designed my own Level class that deals with individual level content and other level-specific functionality. In order to separate level specific drawing behavior, such as drawing objects that the level creates, I made a corresponding Draw method my own level class that is called on the current level in the game’s draw method. This pattern comes up very frequently over the entire game and has led to a very maintainable game.
Another major advantage to the XNA framework is the ability to use managed code. Due to the large number of assets getting passed around, managing memory in a game is a large and tedious task. Using a managed code base both allows for greater stability and faster development, as memory management errors will not exist and time will not have to be spent preventing them. One trade-off in this approach is performance. Managed code in XNA is not nearly as efficient as the C++ code that high performance games are traditionally built upon. However, this game will have relatively few objects in play at any one time, and will not be using any GPU intensive effects. Thus this downside is negligible for the purposes of the project. Another trade-off of using C# is the need for a bridge to XAudio2, which must be written in unmanaged C++ code. This will be covered later.

XAudio2

The project uses DirectX’s XAudio2 audio library to allow for the game to perform complex manipulations with multiple sound sources and custom effects. Two other options were considered first. One was an open source library such as OpenAL. Unfortunately open source sound libraries in their current state have mainly been designed to provide the basic features that most FPS games require but have not been developed much past that point. In particular that they allow for sound cues and 3D positional audio, but not for complex mixing of sounds and effects. The second option was to use XNA’s own built-in audio capabilities based on the Cross-Platform Audio Creation Tool (XACT). This library is in a similar state as the open source offerings in that it is a great tool for the audio features most games need, but does not allow for advanced control of the audio paths.

XAudio2 is node-based system that allows for sound to be routed through an audio graph with effects being applied at each node. There are three types of nodes: source voices, submix voices and a mastering voice. A source voice references a buffer of audio data and only has an output connection. Its purpose is to push audio data into the graph when started. Source voices can be connected to a series of submix voices or go straight to the mastering voice. Submix voices combine the outputs of source or submix voices and allow effects to be processed on a collection of them just once. For example, rather than processing reverb effects on every single source voice, the source voices can all output to a submix node with a reverb effect attached to it. The reverb effect would only have to be applied to one stream of the combined audio data rather than each source voice individually. Finally, any paths that are connected to the singular mastering voice have their data passed along to the speakers. The mastering voice takes care of any final conversions or configurations. Through the audio graph layout, it is possible to design almost any kind of sequence of audio processing events. This capability is especially empowering for this project since it will allow for audio effects to be applied to very specific sets of objects in non-conventional ways while still using standard effects on the global sound.
Below is a simplistic version of an audio graph that is used in one of the game’s levels. Each source node (Enemy 1, Enemy 2 etc) sends sound through the graph, and effects are applied at each stage. Outputs can be combined and an effect applied on that output through the submixer. Also, volume levels of the sound at any stage can be accessed by the game to create real time visual effects relating to the current sound.

Another great advantage of the XAudio2 library is that it allows for custom effects. A programmer can overload a function in a provided effect base class that takes a buffer of raw audio data as input and returns the manipulated data back. This effect can then be attached to any node in the audio graph and used just like the built in effects. An offshoot of this is it that, although complicated, an effect can be made that pipes the audio data at that node in the graph off to the main program. For the purposes of this project, the audio data can be analyzed and used as a parameter in the game. For example, a background color can change from red to blue to reflect the ratio of audio data in the higher to lower frequency ranges, providing visual feedback from the sound. Exposing the audio data opens all kinds of possibilities for representing sound by non-aural means.

The sound library component is primarily centered around a collection of classes designed for this project that manages the audio graph and puts audio graph changes into higher level terms. In particular, it eliminates the need for pointers to be used as parameters or return values in public functions of the main class. This is necessary for any functions that are exposed through the DLL since C# should not and often cannot use pointers. Instead the management class designed for this project, uses integer IDs to represent wav data, source voices, and effects. The actual game audio manager class that I’ve created has been redesigned multiple
times as the limitations of each design become apparent. The current iteration has proved to be adaptable to different game situations, and easily expandable in terms of adding new effects. This has been accomplished by creating a standard interface for working with voices and their effects. The AddEffect method of a voice initializes a new effect and adds it to the chain. Then at any time after an effect is added, an UpdateEffects method can be called, passing in a struct containing data specific to the effect that needs to be updated. By managing the list of current effects in the C++ class, the interface has been simplified significantly to encompass only the core functionality a game needs to interact with the audio layer.

**C#/C++ Bridge**

The difficulty of bridging the two languages was unexpected, and it initially set me back from my proposed schedule. Bridging managed and unmanaged code in Windows only has two options for high performance applications: COM and P/Invoke (also known as DLLs.) Exporting a DLL out of C++ and into C# was by far the easier option, however there were disadvantages to the process in general. Classes cannot be imported into C#, only functions. I've worked around this by simulating a class via a member variable in the DLL that creates an instance of the sound library class, and various functions that map directly to methods on that class. One challenge in particular was finding workarounds for the passing around of pointers to objects which is very common in C++. Instead, I implemented IDs to represent game objects where those IDs can be used to look up the relevant objects via private members in the main sound library class. An area where I was helped out significantly is that XAudio2 by default runs in its own thread, so hang-ups in the graphics program do not affect sound processing.

This solution has definitely been adequate for this project’s goals. A more ideal solution would be to provide a managed wrapper for XAudio2 itself that would allow any managed language to use the same functionality that C++ can. This would remove the need of using C++ except for customized effects, simplifying the number of updates that are needed. Currently there are people working on this, but no substantial solution has been released.

**Level Examples**

**Sync Up Music**

In this level, there are two source voices playing the same looping track, but one source is delayed behind the other by 0-200ms using the Echo effect. This creates the very noticeable result of an offbeat sound. The goal of the game is to realign the tracks. The player is not given any instructions beyond moving and shooting. When
one enemy is shot, the delay gap widens making the audio sound more offbeat. When the other enemy is shot, the delay gap closes by a fixed amount. When the delay gap has been reduced to near zero, the level finishes and the player can progress.

There is no visual feedback beyond the bullet hitting the enemy to cue the player that they are any closer or farther to beating the level. If the user simply fires randomly at the two enemies then the game will go nowhere. The player must rely on listening to how their actions in the game affect the audio output and their sense of what constitutes a correct sound in order to win. There is a connection between the player shooting an enemy and the resulting audio, and the audio itself gives useful feedback as to the state of the game.

 Shoot the Distortion

In this scenario there is one large enemy that is the source of an audio track. This is portrayed through the fact that as it moves, the 3D positional audio will update. In addition, it makes intuitive sense that the main object on the screen is the source of audio. There are also a number of smaller enemies floating around. The main audio source is distorted by a distortion effect. The amount of distortion is proportional to the distance between one of the smaller enemies and the large enemy. Given this varying distortion in the audio, the player must find out which enemy is causing the distortion and shoot that one, but not any of the others.

The player must listen to determine that one of the small enemies is the cause of the varying distortion effect. The audio in the game provides key information on exactly what is a target in the game.
Locate With Frequency

In this level, there is one enemy that is visually obscured by darkness (in the graphic the blur is intended to show where the ship is, though in the game there is just black.) There is no way to know where to shoot purely from visual information. However, the Y location is tied to a pitch of the source voice emanating from the object. As the object moves to the top of the screen, the pitch increases, and as it moves to the bottom the pitch decreases.

As damage is done to the enemy, the sound volume coming from the enemy ship increases, while the same track will begin to play progressively louder from the player’s ship. This provides a two way connection with the audio, the player must listen to the change of the effect to make progress in the level, and the state of progress in the level is represented through the balance effect.
Conclusion

The overall result of this project is a simple game that showcases at a basic level the new gameplay patterns that can be used when the ability to manipulate advanced audio properties is made available. Even very simple effects, as was seen with time delay, can be used as the building blocks to create intellectually challenging levels. Most significantly, the ability to tie game runtime variables directly to audio effect parameters allows for a large number of new gameplay possibilities. In addition to showcasing examples of what this type of approach to game design is capable of, the audio code in the project has turned into a well structured library that could be adapted or expanded for other purposes or games. There is a clear process to adding custom effects in the core audio classes and making them accessible to the game that one could follow without a very broad knowledge of the code. In addition, the C# game itself is designed by the same patterns of encapsulation that is commonly used in XNA games, which makes for very intuitive and expandable code even by someone inexperienced in game development.

The most challenging part of this project was designing a suitable audio manager in C++. In contrast to the game written in C#, there were many more choices and trade-offs to writing the audio component in C++. In particular, striking a balance between making a generalized class that can be reused for other purposes versus making a very specific version for my game involved some compromises. The class was rewritten multiple times as the advantages and disadvantages of each approach became more clear. However, as by the culmination of the project I had a working class that has since been easily expandable to the growing needs of the game, the solution as it stands can be considered to be a success.

References

"Rez HD." Rez HD. <http://www.thatgamecalledrez.com/>.

The home website for the game Rez HD.


This is a paper that was original published in the Proceedings of the Audio Mostly Conference—a Conference on Sound in Games. At the time
of publishing, the author was a Lecturer for Indiana University, a PhD candidate and co-author of several Flash books. The paper discusses the improvisational and generative forms of experimental music and the theory behind it. The paper goes on to a number of case studies, two of which are music video games. In the video game case studies, the paper describes how qualities of the music are changed based on user interaction with the program to create unique musical outputs.


The home website for the music video game, Everyday Shooter.