Dex: An Improvisational Music Game

Nick Piepmeier
pieps@aya.yale.edu
Advisor: Paul Hudak
Yale University

June 12, 2007

Abstract

Most music video games currently on the market focus on music production, but many rely upon some sort of fixed musical score to drive that production. Because of this, there has yet to be a music video game that allows and encourages the user to create new, original music, as opposed to regurgitating a predetermined score. Whereas these music games can be enjoyable to play, few include the creative process into the gameplay itself. In this paper, I discuss the implementation of a game engine that pushes the player to compose original music while still controlling the possible range of input so as to allow for smooth gameplay.

1 Introduction

Music video games are generally categorizable into two different types:

1. Those which attempt to model instruments and are geared toward the skillful playing of those instruments, and
2. Those which put the player in a more abstract, traditional game environment, in which the user's actions influence the progression of the music.

Both of these cases have interesting aspects that deserve further consideration.

Games that model instruments tend to focus so extensively on the playing of music with those instruments that other game mechanics are discarded. For instance, in the popular game *Guitar Hero* by Red Octane, the guitar metaphor is taken to such an extreme that the user plays on a guitar-shaped controller while attempting to correctly respond to guitar-tab cues onscreen. While these games may result in what some call an overly simplistic game mechanic, they often result in a rich musical experience. It is important to keep this in mind, as such an environment is most conducive to creativity and the creation of original music.

Games that utilize music as an extra dimension of gameplay or to augment the existing game mechanic tend to have far less focused musical systems, and therefore cannot achieve the sort of musical depth that most instrument-focused games do. On the other hand, such games tend to prescribe a specific musical course of action far less than the instrument-focused games do. A good example of one of these games is Sega's *Rez*, in which the musical environment changes with the user's actions. This is implemented in a rather simplistic way that the user has little control over, thus limiting the effect that it can have on the game mechanic. In many cases, however, the addition of even limited music creation mechanisms to already innovative games serves to add a great deal of depth to the overall game experience, while not necessarily affecting the nature of the gameplay in a drastic way.

2 Structure

My goal was to achieve a rich music creation environment that is played like a non-music game. That is, I wanted the power and freedom of a musical instrument with the interactivity and structure of a video game. This game concept can be broken up into several submodules that communicate over TCP sockets:

- An action game
- An instrument-like music generation system
- An AI to influence and judge the creation of music
- A metaphor to link the visual and musical aspects of the game

2.1 An action game: DexClient

Much of the work on this part of the project was done in the spring of 2006 when I wrote a generic space/shooter game. The game was created with a music game in mind as the final product, and so even without sound, the animations and scripting interface were created to be heavily based on rhythm. The user interface was also designed to allow for a large number of unique inputs that would correspond to specific notes, and the use of space in the game was tailored to the fact that visual data would also be used to make assumptions about the audio state.

The updates made to this module for Dex enabled the game client to communicate with a level server and a sound client, thereby obviating the need for heavy AI or sound coding within this client. It also allowed for parts to be interchanged relatively easily and for the possibility of online multiplayer game sessions. The current game client will send user and CPU player move information to the level server and get state updates back from the server. It also maintains its own timing system so that it can keep in sync with the sound client and make the CPU player move in time with the level server's cues.

2.2 A music generation system: HasMidi

Because of Haskell's rich typing system and the relative ease of implementing linguistic structures in it, it was an easy choice as the language to use for most of the background coding in this project. The existence of Haskore, a powerful music module for Haskell that facilitates the codification, expression, and processing of music and MIDI, made Haskell the obvious first choice for both the sound client and the level server. Haskore allows for the manipulation of music in almost any way imaginable, so it was relatively simple to create a program to traffic in music. The major difficulty was getting this music to be output in real-time, since Haskore lacks such facilities.

This problem was remedied by writing a wrapper DLL for several of Windows' low-level audio system calls, then creating a Haskell module to interface with it. The module provides facilities for scheduling individual MIDI events and for real-time input of MIDI data and output of Haskore Music and MIDI data. A wrapper for Windows' PlaySound() function was also included in the module. The MIDI functions work on Windows' stream-based MIDI system, allowing for arbitrary amounts of MIDI messages to be scheduled for output while at the same time listening for MIDI input. PlaySound() has nothing to do with MIDI, so MIDI events and WAV output can happen concurrently without any problems.

HasMidi was then put to use in the Dex sound client, which listens for specially tailored messages, then processes them and passes them to the HasMidi API. This all happens more or less instantly, allowing any timing to be done in the DexAI.

2.3 An AI: DexAI

Haskore has already accomplished a great deal of what otherwise would have been a painstaking and difficult process of expressing the series of signals going between the different modules as music, and allowed for much higher-level processing of that music, but the transition between messages and music is still a nontrivial process.

The DexAI is implemented as a server that receives messages from the Dex game client (the Dex sound client only receives messages - it doesn't send them), then processes the messages and sends output after updating its internal state. It is responsible for sending all messages to

the sound client, as well as for updating the game client's state.[1] A side-effect of the DexAI's implementation as a listening server is that it is a completely reactive AI: it can only act when it gets a message from the game client. Making it act on its own would mean juggling the input it gets from its sockets and system sleep and wake-up operations - something that is best left to procedural languages such as C++ or Java.

The DexAI's internal state is comprised of a set of rules, an index into that set of rules dictating the current rules to use, some time data structures to keep track of note times, tempo information, and information on the last measure, two measures, and four measures of player input/music.

When the DexAI gets input from the game client, the input is of one of three categories: start/stop messages, key-on messages, and key-off messages.

- A start/stop message initializes the AI's internal clock for note timing[2], then the message is sent off to the sound client to tell it to get started as well. This serves to synchronize the states of all of the modules.
- Key-on messages signal to the AI that a player has initiated an attack (i.e. pressed a key), and records a partial note (including which player attacked) in the environment as well as filling in the amount of time between the end of the last note and the start of the current note as a rest in the recent input structures. The level's current rule index is modified, and the new set of rules is sent to the game client. A note-on message is then sent to the sound client that corresponds to the particular key-on event.
- Key-off messages signal to the AI that a player has finished an attack (i.e. released a key), and completes the partial note recorded in the environment for the key-on message. The resulting note is then added to the recent input structures. A note-off message is then sent to the sound client that corresponds to the particular key-off event.

If, during a key-on or key-off message one of the recent input structures overflows (i.e. a measure of the song has been played), the completed measure is sent to the level's AI functions to determine if damage should be assessed to either side, the environment's recent input structure is emptied of everything but the overflow input (i.e., that in excess of a measure), and damage messages are sent to the game client, if necessary.

The interesting parts of the DexAI are in its measure processing AI functions. Each level contains a **statistical** AI function and a (for lack of a better identifier) **linguistic** AI function. The implementations of these are left up to the creators of each level, but each takes the music played so far along with some other parameters and returns a series of strings to send back to the game client that assess damage to particular players.

The **statistical** AI functions that I've written take the measure (actually a list of `(note, playerId)` pairs) and a playerId and run several statistical processes on the measure. An example of this would be a function that analyzes the average duration of a note, then calculates the total deviation from that average over the measure. The return values of the statistical functions are then averaged, and if the resulting value is above a given threshold, damage is assessed to a player.

The **linguistic** AI function is slightly more interesting in that it takes the current level's rule list as well as the current rule index as well as the measure and player information that the statistical AI functions take. Rules are expressed as lists of lists of `(value, weight)` pairs. In a ruleset, there are rules for which notes are advisable to play, what their durations should be, and how long to wait before playing that note. Rulesets are grouped together into lists, and those lists are indexed by RuleIndex datatypes. On each key-on event, the rule indices are advanced, causing the AI to progress through the rule structure in a traceable way. The linguistic AI functions that

[1] This makes the DexAI ideal for a multiplayer online environment, as it is responsible for keeping all clients in a consistent state. It is conceivable that in the future, instead of a human player and a CPU player, there could be two human players in different locations, both connected to a single DexAI server to play a game

[2] The AI keeps track of its own time so that it can be independent of its client's times. This is helpful if there are two or more human players or one of the players decides to cheat.

I've written merely trace back through the list of notes (the measure) and the rules and sum the weights of the rules associated with the actions taken. If this value is higher than a threshold, damage is assessed.

It is important to note that the AI functions have been left open-ended and level-specific. This is to allow for the greatest variety possible among levels, and for novel ways of analyzing player input.

2.4 A visual/musical metaphor

The original version of the Dex game client had various phases that would correspond to different styles of gameplay. The first, in which the human player was supposed to dodge a number of obstacles, was intended to acquaint the user with the level/song, allowing for less of a learning curve for the subsequent phases of the game. This was roughly analogous to the pre-planned levels/songs in instrument-based music games, in that there would be no room for improvisation due to the fixed nature of the level and lack of computer opponent.

The second phase was to be more interactive, using a computer opponent that would react to the player's moves and construct a give-and-take relationship in which one side would be attacking at any given time. This is in fact very close to the current state of the DexAI and the CPU player.

The third phase would be a free-for-all, in which either player could attack at any given time, and the other would have to respond appropriately. This would be the most difficult phase of gameplay, as it would require quick reflexes and essentially prevent either side from thinking about future moves. Thus it would create a completely reactive style of gameplay very different from that of the other two phases.

The original incarnation of the Dex game client used projectiles as attacks and expression of notes, and like in a typical action game, if a projectile hit the opponent, the opponent would be damaged. It is easy to see how this is not a good way to link the visuals of the game with the music. This early visual/musical metaphor made it nearly impossible to analyze the music played and provide feedback that made sense visually, since the only effect the AI could have on the game state was on the CPU player's movements. It was clear that a less literal approach to the interface between sound and visuals had to be taken.

By doing away with the concept of projectile collisions causing damage and replacing it with more abstract visual stimuli, a more feasible connection between the aural and visual portions of the game was made. The addition of "action suggestions" that players would attempt to hit with their projectiles to the center of the playing field, and the concept of assessing damage at the end of measures allowed for a sensible (if somewhat abstract) visual/musical metaphor.

3 Future plans and improvements

Some things I'd like to implement in future versions of this game are

Less-linear levels/rules
: Implement more interesting levels and rule structures with the capability for loops and semi-ambiguous states

More reactive CPU AI
: Make the CPU AI more reactive to the human player's moves, possibly adding different personalities and styles, while reducing its dependence on the level rules

Add third phase of gameplay
: Fix the timing bugs that prevent some forms of rapid gameplay in order to allow the third phase of gameplay mentioned above to be implemented

4 Conclusion

Dex has much room for future expansion, but provides several of the key features important in a music game that allows for creative composition of original music within a video game-style rule/prompt framework. Dex allows the user to choose which notes to play when, then judges the

user's music based on a rule framework, thereby encouraging the user to figure out what musical style the level represents and then compose music accordingly. The computer player provides stimuli and feedback based on the character's actions and the level, and the game itself provides both visual and musical feedback as to the user's status. Given enough time within a level, the visual/musical metaphor allows the user to progress from playing the game based primarily upon visual cues to playing solely based on the music being produced. This causes the game to become more intuitive, and thus more conducive to the production of original and creative music the more it is played.

Thus, Dex serves as proof that it is possible to make a product that is at once an action video game and an instrument; it is possible to create games that assist the player in creating new and interesting music.