Review of HIT Sound System Design Documents

JDM, 21-Jan-99.

I reviewed the documents from Paul W. delivered Tuesday, 19 January, 1999.

1. Goals

I was reading the documents with a dual focus:

1. Understanding the ideas for a common sound programming layer, at a higher level than the GZSoundSys module that all internal products have been using for a while.
2. Understanding how the proposed technology would fit into *The Sims* project, as a software technology, and as a production process.

2. Overview

I think I get it: if we think about playing "tracks" where we have been so far playing .wav files, the programming is simplified on our end, while more power and flexibility is provided to the sound programmer and composers.

Here are some important assumptions I made about who does what, ideally:

- The game programmers call out "tracks" to be played, effective immediately (although a track can begin with a delay).
- The sound programmer (together with the composers?) writes "HIT-code" to implement the specific tracks. A simple track can play a .wav file to completion, or in a loop.
- The composers work with the sound programmer to generate samples to be used in the programmed tracks.
- The sound programmer (and composers) can tweak timing without troubling the game programmers or animators.
- I couldn't figure out where the production staff or game designers come in.

I read a few different documents. I was more comfortable with the ones that referred to "patches" as the primitive sound element than the one which talked about "notes" with velocity and duration. I guess that's like a sampling synthesizer versus a MIDI device or something.

3. Compelling Examples

Paul includes a few track examples based on some situations in *SimCity* and *The Sims*. I've added some.

3.1 Shower

In the shower example, he provides two tracks which correspond to "start shower running" and "stop shower running" which quite elegantly provide an "ease in" (or faucet squeak), water loop, and "ease out" (or faucet squeak) which interrupts the water loop, when stop_shower_running is invoked.

Programming on our side would be simpler (just call out "start" and "stop" without the spoof that Patrick pulled) and the effect can be as simple as ours is now (or simpler) or made more complex (with ease in, with a randomness in the loop, or whatever). Game programming wouldn't be impacted.

3.2 Car Pool

The last design discussion I heard about the carpool honk went along these lines:

- The first time the car honks, it should be an unlocalized (i.e., loud) sound, thereafter the honks should be localized, and every 15 game minutes
- When we're running in fast mode, we might want to have the honks less frequent and/or maybe just a single "honk."

The system, as I read it, would allow us to invoke just two tracks: car_pool_arrives and car_pool_leaves, and Paul could program all of this design and more, as well as triggering the departure "vroom."

(Some of this assumes that we've decided to pass the sound system the game-speed, which it could subsequently test and make decisions.)

If we ever decided we wanted to synchronize the horn honking with some animation of the car (we probably won't), we'd just add a new track call-out car_pool_honk_event and Paul could change the periodic scheme to an event driven scheme without Patrick's involvement at all.

3.3 Random Choices and Smart Lists

In a few places, we've decided to play random samples (including VOX for conversations). This complicates the trees, and they get even more complicated when we try to avoid repetitiveness. The HIT system introduces a concept of "smart lists" which are designed for random selection, with control of repetition (by specifying how many choices must occur before a specific item in the list is to repeat).

It's a good place for such a common sound operation, especially when game scripting languages are probably not particularly elegant doing the same.

We went to some trouble to support automatic variations of VOX, which are randomized, and also sensitive to the gender and age of the speaker. Now, we are trying to figure out how much work it will be to do the same for SFX. This could all be done inside HIT, relieving the game programmers. But it must be noted that part of our solution includes specification in our game database of the number of these variations, which is passed along to the piece of code which has to make the random selection. I don't see how this would be organized using HIT.

3.4 Phone Dialing

We could call out a track to start phone dialing, and another when each button is pushed. As I infer, we could even pass an argument 0..9 to select a phone pad key.

Then, any of these implementations would be transparent to us:

- The start_dialing track just plays seven tones, with delays tweaked to match our animation, and the button_push tracks are ignored
- The button_push tracks are used to play tones corresponding to the phone numbers we script and pass down.
- The button_push arguments are ignored, and Paul runs a random algorithm in HIT to pick the phone numbers.
- Paul's home phone number is dialed every time, and nobody notices.

You have to believe that this flexibility is a "good thing" to assign this some value.

4. Integration with *The Sims*

This section might also be meaningful for other games.

The API for the system is not really described in the current documents. I'm making a lot of assumptions here. The key assumptions are:

1. The game calls out tracks to be played, ultimately by passing an ID which identifies a "track resource."
2. Some kind of symbol table (the .ini file?) maps strings or #define symbols to these track ID's.

4.1 Edith

Edith (the visual in-game scripting language of *The Sims*) defines two sound "primitives," one to play a sound, and one to kill all sounds associated with a simulation object (such as a shower) or a person.

Edith defines its own "symbol tables" which map text choices in pull-down lists to ID's in the editor, and maps back to .wav file names at run time.

We would presumably rework the play_sound primitive to provide the Edith programmer with a pull-down list of track names, and pass along a track ID to HIT.

Be advised: we're not planning on changing our symbol table scheme or adding to our resource management; I'll speak about these aspects of integrating HIT again later.

The Sims sound middleware keeps tracks of which object each (Edith-driven) sound is associated with; we would need to find a way to use HIT sound groups for this functionality; HIT might need to support our Object ID numbers (how big?) as its group ID's.

The play_sound primitive in Edith could be enhanced to pass standard Edith arguments to HIT, so we could specify the phone keypad button, the gender of the speaker, or whatever else we would find useful. We might also define subgroups of sounds associated with an object, and provide ways to turn them off selectively using the kill_sound primitive.

4.2 C++ Binding

Of course, not all of our sound call-outs are from Edith scripts ("trees"). We play and control sounds from the GUI, from the locomotion engine (footsteps), and (maybe) when we go to Pause Mode and Resume, and (maybe) when objects are created and deleted (some of this might be done by Edith trees).

The C++ programmer needs a way to call out tracks symbolically; it will be nice to unify this scheme with the one Edith uses. I'm not clear on the details.

4.3 Scheduling

As I understand it, the sound engine in SC3K ran on its own thread. *The Sims* schedules time-slices for all of its components off of the main thread.

I can see how a track player would be easiest to run off of its own thread, and if you think of the audio device as "blocking I/O" than my Nazi rules for threads would still permit it ("maximum number of threads equals one per blocking device, plus one").

It might be that in order to support streaming sounds (we want) then we'll need a background thread <u>in any case</u>. I'm not sure. If so, then we're going to thread-land in any case, and I won't mind making HIT's life easier.

I'll be concerned to keep it to one thread, and to keep an eye on the "precision" that thread has to run at. I hope that we don't get sucked into supporting more precision than we need, just because HIT is used for music stitching ("Freshness?") in other games.

Of course, if HIT is running on a thread, then we'll need to talk to it through messages, and be careful if it calls us back to gather data or get help handling resources or posting errors.

4.4 Application Services

Rather than build HIT (or any other "core technology") on specific low-level services, we suggest (somewhat strongly) that such services be provided through application-specific facilities.

This is quite easy to deal with: HIT defines an abstract base class with an interface it needs to evaluate symbols, acquire resources, post errors, call debugging hooks, and ask questions; and each game implements a concrete subclass to implement those methods. This can be kept simple

Then it boils down to a question of the sizes and interpretation of the arguments, and to knitting the whole schmear together (specifically, symbol definitions and resource compilation) in the production process.

4.5 Game-Specific Processing

One question I'm not clear on is what Paul's vision is for writing a "game specific layer" on top of HITS. Clearly, the specific tracks authored for sound effects are game specific and can take game-specific arguments, and I believe that tracks can be defined and called out just to pass game variables (like game speed) into "globals" for other HIT tracks to query.

But I'm not clear on how he'd do things like he did in SC3K: going out and looking at the game layers and calling various game modules and functions. Which schedule does all that C++ code run on (how is it called?) and what does it do with its answers?

For example, we have a "spatially localized (positional) sound attenuation and panning" capability that is definitely game specific. Someday, maybe we want to be 3D spatial. Who knows, someday our franchise or others will want to us the latest environmental processing (sounds modified by their "position" behind walls, by the echo properties of their chambers, etc.).

Where does this model live, and how does HIT get at it?

4.6 Integration w/ Production Process

I'd like to see a walkthrough of a simple example: the life of a sound effect:

- Designed (by game designers with sound designers)
- Called out by programmer (probably before it's implemented, and we'll need the game to "work fine," yet have a reliable way to make sure we're not forgetting to implement anything we're calling out)
- Sound effect developed in test harness (in case the build with the new callout isn't available to the sound guys yet)
- Sounds (tracks, .wav assets) integrated with the game, picked up (and compiled) in the nightly build
- Reviewed for timing and overall effect
- Tweaked

Specifically, which files (".ini" equivalents, resources, Edith objects, track scripts, .wav assets) are created, when and by whom, checked into where, listed in which databases and/or spreadsheets, modified during the tweaking process, kept in sync with specification and tracking, and finally blessed.

5. Integration with Freshness

I don't see any indication of how HIT is expected to play with Freshness.

As I understand it, Freshness (for Music) requires a lot of precision, and I don't know what other functionality or complexity it requires. If it lives as a "layer above HIT," then it probably wants to communicate through one and only one message and queue scheme.

Bottom line: if I don't need Freshness, I don't want to pay for it in the low level via complexity, schedule, performance burden, or "architecture."

6. Core Technology

HIT has potential to be a strategic and powerful core technology. I have some experience thinking about that. I think it requires some specific things, and short-changing it in the name of expedience (or anything else) makes for wasted time and money and personal investment.

6.1 Documentation

- The design document will be read by every customer hereafter. Terse but clear is OK.
- The "Users' Guide" or API description will need to be clear and kept up to date. Terse but clear is better, especially since the more words, the less likely it's kept up to date.
- *All* documentation should have a copyright/confidentiality notice in the footer, page numbers (and section numbers if people are going to review it), and some sort of reliable version number or modification date so people aren't reviewing or relying on the wrong version.

6.2 Milestone Plan

This has to be done well, worked out before the TDR, and the project measured against it.

My suggestions for HIT development milestones (which I'm certain Paul can improve drastically) would be:

- First subset of functionality (well-defined) and a test harness
- Reorganization of code, if necessary, to support integration with *The Sims* (such as the introduction and use of an Application Services interface).
- Integration with *The Sims*
- Additional functionality needed by *The Sims* or beyond that.