7. Sound system

Within the GameKit are several classes to help you add sound and music a game. The purpose of these classes is to simplify the use of the SoundKit and the CCRMA MusicKit to the point that a novice is able to use these features without any real understanding of what's going on. Of course, a user will be able to better use these features if they have had experience with sound and music under NEXTSTEP.

The MusicKit support is currently limited to the Motorola architecture because the MusicKit from CCRMA is not yet available for the Intel architecture. Once it has become available for Intel machines, the GameKit will of course support its use. If drivers for cards such as the SoundBlaster series become available, an attempt will be made to add support for the basic synthesis capabilities provided by such cards. (This is currently difficult as the author of the GameKit does not have access to the necessary hardware for such development yet. It is supposed±and hoped±that such access will become available by the time it is needed.)

Introduction to GameKit Sound and Music

As you will notice in the following diagram, the sound and music sections of the GameKit are effectively separate. You can, if you choose, use only sound, only music, both, or neither in your game.

paste.eps ¬

On the sound side of things, the GameKit communicates directly with a SoundPlayer object. There can actually be multiple SoundPlayer objects in a game, though this is not strictly necessary. The SoundPlayer takes care of loading and preparing sounds for playback. In order to play a specific sound, an appropriate message is sent to the SoundPlayer. It will decide whether or not to play the sound depending upon the state of the game, sound playback in progress, and user preferences. Each sound effect played by the game is stored in a GKSound, which is a subclass of the SoundKit's Sound class. The GKSound adds certain features which are handy for repeated playback of multiple sounds simultaneously. The GKSoundStream is not actually a subclass of the SoundKit NXSoundStream as you might suspect. Rather, it is a *collection* of NXSoundStreams. This allows multiple sound effects to be played simultaneously. By using multiple collections of multiple streams, it is possible to have dynamic allocation of playback resources while still maintaining a large degree of control over how sounds are to be played.

The ScorePlayer can load up a MusicKit score file (.score, .playscore, or .midi) and play it back via the DSP and/or MIDI as is appropriate. The GameKit takes care of starting and stopping playback at the appropriate times for a score that is meant to be a constant background for a game. *If you want snippets of different music to trigger at key points in a game, you will have to wait for a future implementation which will allow this.* To provide music support (and synthesis), the ScorePlayer makes use of the MusicKit. *Currently, the ScorePlayer object's capabilities are very limited; this will be enhanced quite a bit in the future.*

Sound

In order to make effective use of the sound system provided by the GameKit, it is important to understand what it does. It is designed to handle problems which can crop up when multiple sounds can be triggered repeatedly and simultaneously. Once you understand what it can do, then the methods for configuring the system can be explained.

The most important thing that the sound system allows you to do is play sounds simultaneously. This overlapping makes for a much more realistic feel to a game. However, there are two problems can occur when attempting to play multiple sounds. First, the hardware has limited capabilities. Therefore, there is a limit to how many sounds can be played at once. Attempting to play too many sounds will cause the sounds to ªbreak upº and/or performance to suffer greatly. Remember that you will have to trade graphics and sound performance and find a balance between them which works well for your game; although it will make an attempt, the GameKit can't get this completely right without your tuning. The second problem is that when you play the same sound multiple times, strange audio effects can be created. If you trigger the same sound twice within about a 10-25ms time frame, the user will hear a single sound, but twice as loud as it should be. If the sounds are slightly further apart, the user will hear flanging or echo effects, but will be unable to resolve two sounds. Thus, it makes sense to only actually play *one* sound and ignore the other request to provide aural consistency. How long to wait before allowing a sound to be played again depends upon the sound to a certain degree, and therefore the game designer will probably need to tune this parameter. Both of the problems mentioned above are addressed in their own ways by the sound support objects found in the GameKit.

GKSoundPlayer

The SoundPlayer object is the object with which the GameKit and all other game objects should communicate. To trigger the playback of a particular sound, simply send a ± **playNum:** message to the SoundPlayer. That is all that is necessary to use sound in a game. All other methods and options are just extras to allow you to fine tune the playback mechanism. The SoundPlayer will

look at the user's preferences to decide whether or not the sound should be played. This means that the programmer sends a ± **playNum:** message whenever a sound should be triggered, without having to worry about checking flags to see it a sound should be played. The GameKit automatically handles all this. *As the method name suggests, currently all sounds have a number attached to them; this can be seen from looking at the sound names in PacMan, for example. In a future implementation, I would like to allow the programmer to use the sounds by name and play them back by name. This will only require minor adjustments, but is not currently a high priority, since what is here works fine¼it's just a little inconvenient. Even then, it's not so bad.*

There is another feature of the SoundPlayer which can be handy. You can have multiple ªsetsº of sounds. An example of this sort of thing would be the game Columns. In the Preferences¼ panel of Columns, the user can choose between different sounds that will be played by the game (bubble vs. breaking glass, etc.). By sending a ±**setSoundType:** message to the SoundPlayer, the current set of sounds may be switched to a new set. When asked to play sounds, the SoundPlayer always plays sounds from the current set of sounds. *Right now, all sets must be assembled when the SoundPlayer is initialized. It would be nice in the future to be able to simply ask it to add a new set of sounds. Also, it would be nice to have a ªdefaultº set that is played when a particular active set is missing one of the sounds which has been requested, thus removing the need to store the same sound in multiple sets.*

As the application initializes itself, the SoundPlayer is asked to load the sounds for the game. To find out how many different sounds and how many sets of sounds are to be loaded, the SoundPlayer queries the GameInfo object. The GameInfo object stores both these parameters. (See the discussion of the GameInfo object for more information.) Your sound files must be named Sound**X**.**Y**.snd and be placed inside the app wrapper. The **X** refers to the sound number±zero through the number of sounds you are using per set minus one±and the **Y** refers to the set in which the sound belongs. In a game with only one set of sounds, this is zero. *Again, in the future, allowing ad-hoc names would be nice. To do this, I'll probably add a ªsound.confº file or somesuch inside the app wrapper which will be a string table to correlate the sound number, set, and name.* If you need to alter the way sounds are loaded, then override the ±**loadSounds** method. This method is called from ±**appDidInit:** for the ªdefaultº SoundPlayer which is connected to the GameBrain. If you have chosen to use multiple SoundPlayer objects, the extras

need to be initialized by calling the ±**initSounds:types:streams:** *and* the ±**appDidInt:** methods, but *not* the ±**loadSounds** method. There is really no need for a game to use more than one SoundPlayer, however, and this practice is highly discouraged. *I may use a +**new** style method instead of the ±**alloc** and ±**init** style in the future to enforce this, if I decie it's worth it.*

One final feature of the SoundPlayer is that you can programmatically ask it to shut up, and can set it up to start being noisy after a set time. This feature is independent of user preferences, and user preferences override it. This means that if the user has sound off, the SoundPlayer is silent regardless of whether you asked it to shut up or not. To turn it on or off, use ±**turnOn:** and to shut up for a set time, use ±**shutUpUntil:**. If running on a machine which does not have appropriate sound hardware, such as most Intel machines, then the SoundPlayer will notice this and will abort loading the sounds and will go into a permanently silent mode which ignores all play messages.

GKSound

The GKSound is a subclass of the SoundKit's Sound object. Thus, nearly everything that you know about the Sound object will apply to this object. There are a few significant differences, however, and these need to be brought out and explained. There are some new features which have been added; support for GKSoundStream objects and a special ability to time playback. The GKSoundStream support will be explained in the next section, on GKSoundStreams, since it won't make much sense until then.

The timing of the sounds is to prevent the too±frequent playback of the sound, as explained above. Basically, you use either the ± **setPercentToPlay:** or the ± **setTimeToPlay:** method to determine how this will work. Using the percentage method, you specify how much of the sound must be played, percentage-wise, before the sound will allow itself to play again. The time method ensures that a certain amount of time has elapsed before the sound can be played again. Use one method or the other depending upon whether you want relative or absolute control over how much of the sound has played. The default for GKSounds is to allow the sounds to play again immediately. (i.e. Zero time elapses before it is enabled again.) If you want to adjust this

parameter for a sound, you can do so by overriding the GameBrain's ±**appDidInit:** method. Once the call to **super** has returned, the sounds are loaded. Simply query the SoundPlayer for the GKSound objects by sending a ±**soundNum:type:** message to it. Then set the parameter as appropriate on the GKSound which is returned using one of the two methods discussed above.

There are two other things which you should know about GKSound objects. First, when a GKSound object loads into memory, and after initialization, it will attempt to convert itself to 16-bit linear sound at 22.05kHz. If it cannot do this for any reason, then the GKSound cannot be played. (This is because of SoundKit requirements that sound data which is to be directly enqueued for playback be 16-bit linear. 22.05kHz was chosen since it provides adequate fidelity for a game and does not drain system resources anywhere nearly as much as 44.1kHz would.) This conversion can be somewhat slow and will affect a game's launch time, so it is best to use short sound effects as much as possible.

The other item of note with GKSounds is that although they are subclasses of Sound, they have lost a few of the features of the sound object. Most notably, they break the delegation (±**didPlay:** is never sent) and cannot be stopped or paused with the methods that would normally stop or pause the sound, and they won't return the proper status when the sound is playing. Thisis because the ±**play** method is completely overridden so that they use custom streams. Due to NeXT's design of the object, this is the only method I can override, even though all I want to do is remove NeXT's stream handling. These side effects should not be too serious, however, since a game probably won't need them anyway. If NeXT ever changes the Sound object to make it more subclassable (and if a NeXT engineer asks me, I'll them them what I would need to do this cleanly, hint hint) I will of course try to not break so much. In case you're curious, this overriding is definitely necessary to (1) get the sounds to play back simultaneously and (2) avoid system panics when the MusicKit is used. (Basically, in my experience, NeXT's Sound object and the MusicKit will always cause a panic under 3.0 if used simultaneously; my changes avoid this.)

GKSoundStream

A GKSoundStream controls how many sounds can be played simultaneously and attempts to queue them up in a reasonable order. By using multiple GKSoundStreams you can control ªpriorityº of a sound. Basically, the way it works is this: A GKSoundStream is really a container for multiple NXPlayStream objects. When asked to play a sound, the GKSoundStream will queue up the sound to play on one of the streams. If there is a non-playing stream, it is used. If all streams are playing sounds already, then the sound will be queued up on the stream which will finish first and will play immediately after the sound that is playing on that stream. Thus, the GKSoundStream attempts to intelligently choose which stream will play the sound. Note that a GKSoundStream cannot currently stop playback or pre-empt the playback of a sound to force a new sound to be started immediately. Once queued up, a sound will always play to completion. If all the streams are in use, then sounds will pile up and play when the respective stream gets to it. You need to figure out how many streams your game needs to use in order to avoid too many clogs. The fewer the better, because a stream requires plenty of system resources.

The GameInfo object specifies how many streams are used by the default GKSoundStream. When the SoundPlayer loads up the GKSounds, it assigns them all to play on the default GKSoundStream. In many games, this is good enough. However, in some cases it will be discovered that using multiple GKSoundStreams will make for more realistic sounds. If you want to do this, initialize a new GKSoundStream with the ±**initStreams:** message and then tell the applicable GKSound objects to play themselves on that particular GKSoundStream object with a ±**setPlayStream:** message. From that time on, the sound will use the alternate GKSoundStream object to play back.

Why use multiple streams? It will depend upon your application. A classic example is the PacMan game. There are several sounds which only get played once in a blue moon, but can be quite long. There is also the dot eating sound, which should *always* accompany the eating of dots. Now, if a lot of the intermittent sounds pile up±especially the music at the beginning of a level±then many of the dot munches will end up waiting a long time before getting played and the synchronization will be ruined. To get around this, the dots are given their own GKSoundStream to use for playback±it isn't used by any other GKSounds. Since only one dot can be eaten at a time, only one stream is allocated in the GKSoundStream. This initialization is found at the head of the ±**loadPix** in the PacManGameView.m file, if the reader wishes to see the actual code. This solves the problem; the intermittent sounds and the dots are independent. So why not use one stream

per sound? It is very wasteful of resources; typical sound effects don't play very often, so it makes sense to share streams as much as possible.

Separating sounds into groups that will work together on a particular set of streams and figuring out how many streams are actually needed in a group will typically require some experimentation. Remember you are balancing resources and overhead for accuracy. Trial and error will play a big part in this part of your design.

Limitations

The timing features used in the sound objects are estimates and therefore are subject to all sorts of weird problems. The current implementation in GKSound for when to play and in GKSoundStream for when a sound will finish are both based on estimates derived from the length of the sound data to be played. This is fine under normal circumstances, but could potentially lead to problems on a heavily loaded machine. This manifests itself as sound playback bunching up such that sounds are delayed significantly before they are played and that they overlap when they shouldn't. Imagine, for example, in PacMan stopping and waiting by a power dot and hearing five or six dot-eaten sounds while you are sitting still; they are catching up. This actually happens during development, and can be caused by an overloaded system which makes the sounds play back much later than the GKSoundStream and GKSound expect, making the estimates too early. If the streams get out of sync a little, this is no problem, because once a stream has been silent for even a brief moment, it automatically resets itself and is back in sync again. Note that this symptom can also be caused by not limiting playback times on GKSounds properly, especially when they are played on a GKSoundStream with a small number of streams allocated for playback. In this case, re-adjusting the parameters is enough to fix the problem. (So be sure to check your paramters before blaming the GameKit; usually properly set parameters will result in stellar sound performance. Play with them until you get a good sound!)

Music

This section of the GameKit makes use of the CCRMA MusicKit. If you do not have the latest MusicKit intalled on your system, you will want to obtain it as soon as possible. You can get it via anonymous ftp from the site ftp-ccrma.stanford.edu in the /pub directory. There is a binary and a source distribution available. The binary version is absolutely necessary; the source code is optional. Note that currently the MusicKit only works on Motorola hardware. If you are using Intel hardware, you are out of luck. When the MusicKit is available on Intel hardware it will be supported by the GameKit.

Using the MusicKit allows for really neat synthesized and MIDI music to be added to a game. However, this is not a cheap feature. To use the MusicKit, you will have to link against several libraries and this will cause your application to grow in size tremendously. The linking options for these libraries are, in this order: -lunitgenerators -lsynthpatches -lmusickit -ldsp. They will add around 600k to your program. You may decide that this is not worth it. If NeXT would allow third party shared libraries, then this would be a moot point, and then these huge libraries could be shared by all GameKit apps. *(NeXT, this is something you really should allow us peons to do; X, the MusicKit, and the GameKit are all great candidates for shlibs! It would save me and many others* ***Megabytes*** *of disk space and that would make me a whole lot happier!)* The GameKit distribution includes a library which has MusicKit support and a library without MusicKit support; you should be sure to link against the appropriate GameKit library.

The actual Music support is currently based entirely upon the ScorePlayer object. This object can load musical scores and play them back. *Currently, the only thing which is really supported is to load a single score and play it back continually while a game is in progress.* The PreferencesBrain takes care of loading the score file and allowing the user to change which file is loaded. The GameBrain handles the start and stop of the music. Until this object is cleaned up, this is all it can really do, and it's use is completely transparent to you as a programmer±as long as you have the object and have the appropriate entries in the Preferences Panel, you need to do nothing else. (Except be sure that there is a score file called ªDefault.scoreº inside the app wrapper for the game to use in the absence of any other score files. *In the future, more control over scores and multiple*

score handling will be provided for you. Also, there's is a significant pause while the ScorePlayer object restarts a score. This will be fixed in a future release so that a looping musical score will not make everything ªhangº momentarily when it restarts. (I know what to do, but it is a lower priority right now.)