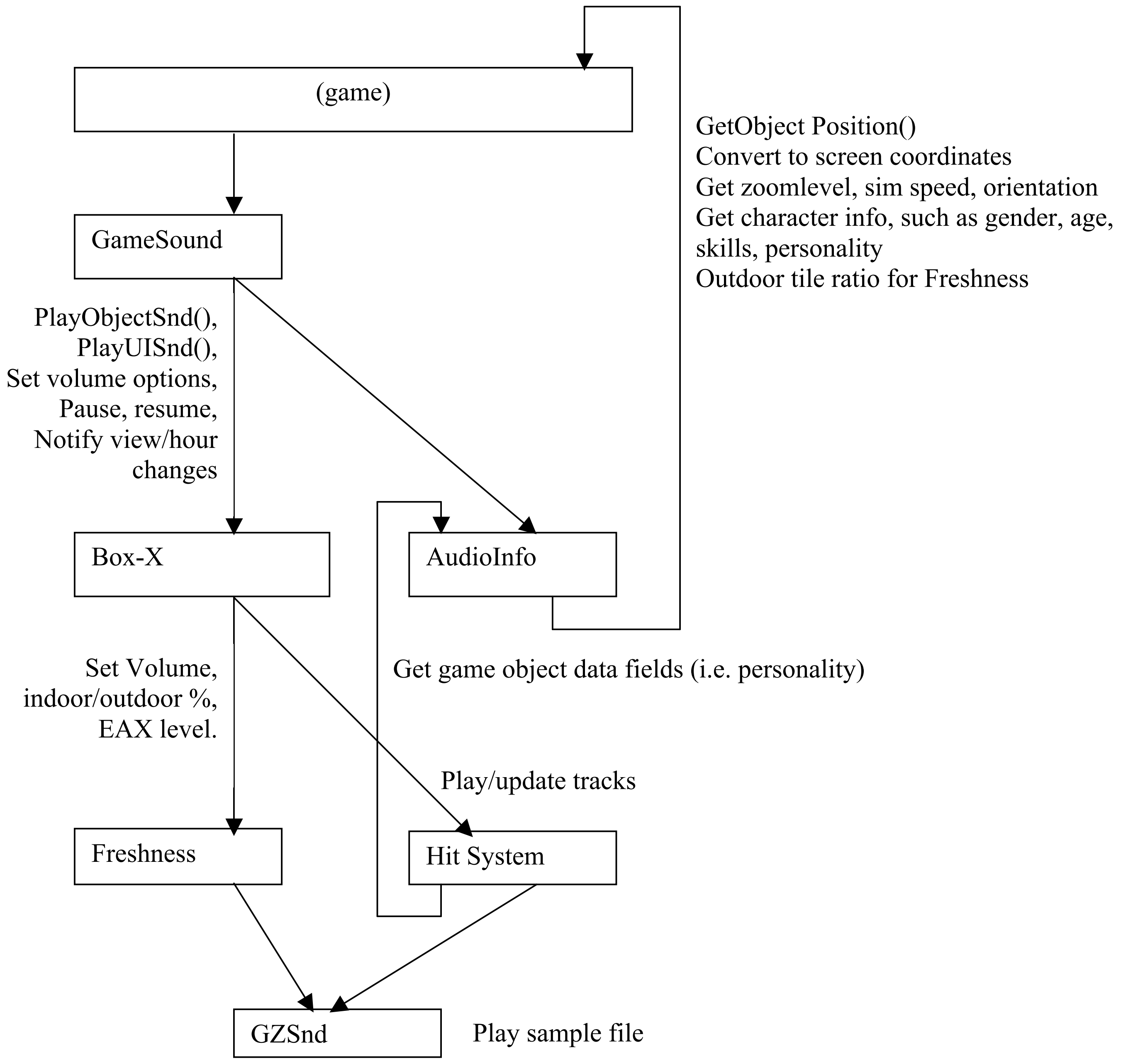
CTG Audio Technical Document

Components


(game)
GetObject Position()
Convert to screen coordinates
Get zoomlevel, sim speed, orientation
Get character info, such as gender, age, skills, personality
Outdoor tile ratio for Freshness
GameSound
PlayObjectSnd(), PlayUISnd(), Set volume options, Pause, resume, Notify view/hour changes
Box-X
AudioInfo
Set Volume, indoor/outdoor %, EAX level.
Get game object data fields (i.e. personality)
Play/update tracks
Freshness
Hit System
GZSnd
Play sample file

GameSound and cSoundPlayer

Triggering sounds

GameSound lets the game trigger sound events. Sound events are triggered by name in order to ease the transition between the old and new systems. Don't worry about the overhead – it's trivial compared to what it takes to play a sample. Furthermore, sounds are hardly ever triggered, at least as far as the computer is concerned.

Triggering a UI sound is easy:
```
PlayUISnd("UI_Nhood_bdoze_demolish");.
```

Object sounds require a bit more information, so they have separate calls
```
PlayObjectSnd("vox_braggee_admire", GetID());
```

There is also a legacy version of PlayObjectSnd that gets the sound name from the `SoundInfo` structure. Don't use it anymore, okay?

Killing sounds

All sounds for a game object are killed together. If an object is playing 2 sounds there is no way to kill one without the other:
```
void cSoundPlayer::QuietBySourceID(Int sourceID);
```

You can also kill all sounds with
```
void cSoundPlayer::QuietAll();
```

View and Hour Changes

```
void cSoundPlayer::NotifyViewChange();
void cSoundPlayer::NotifyHourChange();
```

NotifyViewChange() tells the audio system when the view changes. It's called whenever the user rotates, zooms, scrolls, or changes screens.

Hour change notifications are used for the ambience. These update Freshness and the night loop in case the game is sitting idle.

Data Files

See the HIT system documentation for descriptions of file formats.

.HOT files – contain track and event definitions.

.HIT files – contain binary track data

.SYM files – contain symbol tables that allow late binding of track data.

.HDB files – contain debug information. These files are optional. They contain information that matches track data addresses with source file line numbers. If the .hdb files are provided then any track that crashes will output an error message file and line numbers that

Sound files are of types .WAV, .XA, and .MP3.

Track types

Tracks are typed according to how they use these their arguments. Look in the [track] section of the .HIT files to see how these types are used.

- kArgsNormal – There is no automatic use of arguments. The arguments can be interpreted by the track or not at all. Stings, UI sounds, and tracks that control other tracks fall under this category.
- kArgsVolPan – The first 2 arguments are volume and pan. Only receivers (see below) use this track type.
- kArgsIdVolPan – The first 3 arguments are object id, volume and pan. All object sounds are of this type.

Buffer management

Box-X allows only 16 tracks to play simultaneously. In practice this limit is rarely reached. Tracks are assigned priorities. When a track is started it checks to see if there are too many tracks already playing. If there are, Box-X either kills an existing track, or refuses to let the current track play.

This functionality is encapsulated in `cBoxX::CheckPriority`.

Only 1 copy of a track is allowed to play at once (see "Game objects and tracks"). This obviously helps to limit the number of tracks playing at once.

Objects and tracks – the instance map

This section describes the functional relationship between game objects and tracks.

Only 1 copy of a track is allowed to play at once. If 2 people are "saying" the same track, only one of them will be heard. This was done in order to prevent the echoing and flanging effects associated with allowing multiple copies of the same sound.

The instance map represents the many-to-one mapping between tracks and the objects that are currently playing them (many objects to one track).

The following Box-X functions implement the instance map.

- `bool IsInInstanceMap( Sint32 lSndobId, Sint32 lInstanceId );`
- `void AddToInstanceMap( Sint32 lSndobId, Sint32 lInstanceId );`
- `void RemoveFromInstanceMap( Sint32 lSndobId, Sint32 lInstanceId );`

If there are a man and a woman "saying" the same track, you will only hear one and not the other. This artifact (i.e., bug) was left alone, because it had the desirable effects of limiting buffer use, improving conversations (it stopped people from talking over each other). It also provided a natural way of limiting clutter that would otherwise had to have been implemented at the application level.

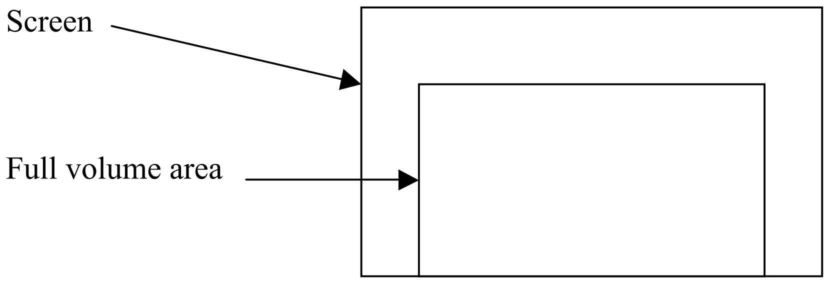
Computing volume and pan

The station computes a weighted average volume and pan position across objects playing that station. Objects closer to the center of the screen are given more influence over pan position. The result is that if there are TV's to the left and right of the screen the sound will emanate from the center. If you scroll toward the right, then pan position will move to the right as the TV on the left moves off the screen and the TV to the right takes over. Try it and you'll see what I mean.

View changes and Pan Positions

When the view changes (by navigating, panning, rotating, etc) the pan positions are updated.

When you zoom out the volumes are attenuated. All objects except stereos fade to silence as you scroll them off of the screen. There is a "full volume" area, inside which all objects are at full volume.



The objects *begin* to fade out when they leave the full volume area. They become silent a certain distance off the screen.

Stereos are always audible no matter how far off screen.

Receivers – Radios and TV Stations

Radios and TV's tuned to the same station must sound play the same sounds at the same time. If each radio had its own track then 2 radios tuned to the same station would play different music.

Here is some unfortunate terminology. A receiver is actually the station. Sorry 'bout that. Let's just call them cReceiver objects "stations".

Each station has a track. When a radio or TV turns on a track it is added to the stations list of objects that are turned on. We use this list to compute volume and pan.

Initialization

Receivers are given directories at init time. All sound files in the directory are put into a hit list. All subdirectories are put into their own hit lists, including the directory that is up one level (this is where you put commercials – see below). All of these hit lists are put into another hit list. The track then repeatedly picks a sound out of each list and plays it.

Controlling repetition and variety

HIT lists are used to all randomness without repetition. Some stations require a bit more structure however, especially TV stations. For example, in an action station we alternate dialog and chase scenes. We do this by putting the dialog and chase scenes in their own subdirectories of the action station directory.

Commercials

Commercials go in the directory that contains the station directory. This means that all stations in the same directory share the same commercials. Since radio and TV stations are in different directories, they have different commercials. Yay.

Music Modes

We use receivers for the front-end music as well, but only for the hit list and drop-in capability. There are hit lists for buy, build and neighborhood mode. We completely ignore the station objects. A specially designed track uses the hit lists created by the buy, build, and neighborhood stations.

Buy and Build music don't cut each other off. If you switch between these modes the current song will finish and then the next song will be chosen from the list corresponding to buy or build modes (whichever mode you are in). Also, if you jump out of these modes and back in the music will fade back up instead of starting a new song.

Radio stations provide the music in "live" mode. Radio music is streamed from hard disk. The user can add their own music files by dropping them into a radio station's data directory.

List of music modes

- Load – plays a load loop. It should not skip or contain static. It should not overlap any other music.
- Neighborhood – Plays music, ambience (daytime sounds), and river loop. It's always daytime in this screen.
- Live – no music (except stereo object)
- Buy, build – music
- Options – no music
- Web page publishing – no music.
- Edit family – like neighborhood but without the stream or ambience
- Credits – same as family

Options

- FX – controls all sound effects, including TV
- Music - controls all music except the load loop, including radios and stings
- VOX – controls all voice

It's a common mistake for a sound to be controlled by the wrong option because this option is entered manually.

MP3 Support

GZSnd provides MP3 support through DirectShow, which is part of DirectX Media 6.0. Since 6.0 doesn't support MP3 properly we include a Beta version of Windows Media Player. The advantage of DirectShow is that it takes care of some knotty licensing issues and allows hardware acceleration. The latter of these can prove to be a disadvantage because the 3rd party support for this new functionality is quite buggy.

Take a look at GZSnd.cpp and GZSndSys.cpp for the implementation.

Building the graph and playing the file

To build the graph, we create a GraphBuilder object and call `cGraphBuilder::RenderFile` with the file to be played. This is the simplest way to build the graph, and hopefully the most robust.

To play the file we get a MediaControl interface from the GraphBuilder and "run" the graph.

We control volume and pan through the IAMDirectSound interface, which we get by calling QueryInterface on all of the filters in the graph until we find one that will return the interface.

Notifications

DirectShow sends messages when the files reach the end. These messages unfortunately need to be passed all the way from the app, through Box-X, and to GZSnd.

Shutdown

We ran into some tricky thread problems in the shutdown code. If we shut down a file and deleted it just as it ended there could be a message still coming down the pipe. To avoid this race condition we send a message to ourselves after stopping the file. This ensures that the message we send will always come after any messages being sent by DirectShow. Once we receive our own message we know it is safe to shut down the GZSnd.

Ambience

The ambience consists of a Freshness score and a night loop.

The Freshness score contains birds, dogs, cars driving by, planes, etc.

The night loop (crickets) plays from 6 in the evening until 6 in the morning. Take a look at `cBoxX::UpdateNiteLoop()` if you don't believe me.

Freshness score

EAX

EAX is applied to the Freshness score when the active character walks in a small room.

3D sound support

Only the Freshness score is in 3D. The rest of the sounds are 2D. Stereo pan position (left/right balance) is set according to their position on the screen.

Installed Image

Disk Budgets:

Hard Disk - 100 MB

CD - 300 MB

Data Locations:

The installer copies all data to the hard disk except for music. The directory structure on the hard disk shadows that of the CD. The only difference is that the

The music player will supports shortcuts. This means the user doesn't have to copy their MP3's to hard disk.

Music is in mp3 format, streamed from hard disk or CD. Streaming from hard disk allows the user to add their own mp3 files.

Plug-in tracks and events

Box-X assembles a list of data directories. It loads all of the data in these directories.

`GameSound::AddDownloadDirectory` adds a directory of data files. If called before Box-X is initialized then the directory is added to a list that is processed at init time. Otherwise the directory is loaded immediately. Load order is last-in, so you can overwrite old tracks and events.

Debugging and Cheats

Cheat window

Pressing ctrl-alt-c opens the cheat window. You can trigger sound events from this window using the *soundevent* cheat, or its abbreviation *sev*.

Example:
```
soundevent ui_moneyback
```
or
```
sev ui_moneyback
```

Most sound events trigger sounds that emanate from a character on the screen. Since the cheat has no character, we use the character being controlled by the user. Note that the character *must take a step before the cheat will work for that character*. There is no other easy way for the debugging code to know which character the user is controlling.

Example:
```
sev flamingo_approve_vox
```

Debug output cheats

The following Box-X events control debug information that is written to the trace window in Dev Studio:

- kDebugEventsOn – start echoing event names as they are triggered
- kDebugEventsOff – stop
- kDebugSamplesOn – start echoing sample filenames as they are played

- kDebugSamplesOff - stop
- kDebugTracksOn – start echoing track names as they are started
- kDebugTracksOff – stop

You can trigger these cheats like this:

You can also trigger these cheats from the debug window like so:

Appendix – Track Examples

Track functionality

This section describes the track functionality most fundamental to the audio design and provides some examples of their use.

Hit lists

HitLists are lists of sounds. The tracks picked randomly from the lists. The hit lists were set to limit repetition in the voice tracks. This allowed the random selections without having the same sample repeated twice in a row.

Branch on object info

Tracks can get information from the objects and branch on it. AudioInfo.h contains a list of available fields.

Randomization

Tracks were programmed to wait random amounts of time. This added a natural feel. For example, Sims would occasionally go silent for random period while reading. Some tracks were random probabilities of playing. For example, a person with low cooking skill would occasionally say "ouch" while chopping.

Example 1 – Random chance of playing

This track plays a sound only if a random number between 0 and 1000 is less than 100.

```
tv_repair_deathwarning
                        ; 10% chance of playing
                        loadl v1 #1000
                        loadb v2 0
                        rand v1 v2 v1
                        loadl v2 #100
                        cmp v1 v2
                        iflt @_tv_repair_deathwarning
                        end

_tv_repair_deathwarning
                        loadl patch #3731
                        note_on v1
                        wait_samp
                        end
```

Example 2 – Branch on gender

The track `generic_fm_smartlists` takes 3 hitlist id's as arguments.

```
comeseeme_like_vox
                        loadl v1 #1028
                        loadl v2 #1029
                        loadl v3 #1030
                        jump @generic_fm_smartlists
                        end
```

Example 3 – Piano

Repeatedly choose a hit list based on skill level. Piece a piece for that skill level and play it. If the skill level changes, the next piece will be chosen according to the new skill level.

```
PlayPiano
                  ; get creativity
                  loadb v7 5                         ;v7 = field ID (5 = 
creativity)
                  getsrcdatafield v7 arg1 v7    ;arg1 = source ID
                  ; divide by 10 to get piano skill level
                  loadb v6 10
                  div v7 v6
                  ; add hitlist if of base level to get hitlist for this level
                  loadl v6 #1210
                  add v7 v6
                  ; set hitlist
                  smart_setlist v7
                  ; choose patch
                  smart_choose v7
                  set patch v7
                  ; play song
                  note_on v7
                  wait_samp

                  loop
```

`Example 4 – A complicated vox track`

A person can be neat or playful or normal. This was enough complexity to make it very difficult to test and get right.

```
med_cab_brush_teeth_vox
                  ; branch on neatness
                  loadb v7 20                   ;v7 = field ID (20 = neatness)
                  getsrcdatafield v7 arg1 v7    ;arg1 = source ID
                  loadb v6 33
                  cmp v7 v6
                  iflt @med_cab_brush_teeth_vox_sloppy

                  ; branch on playfulness
                  loadb v7 18                         ;v7 = field ID (18 =
playfulness)
                  getsrcdatafield v7 arg1 v7    ;arg1 = source ID
                  loadb v6 60
                  cmp v7 v6
                  ifgt @med_cab_brush_teeth_vox_playful

                  ; neither sloppy nor playful
                  loadl v1 #684
                  loadl v2 #685
                  loadl v3 #686
                  jump @generic_fm_smartlists
                  end

med_cab_brush_teeth_vox_playful
                  loadl v1 #681
                  loadl v2 #682
                  loadl v3 #683
                  jump @generic_fm_smartlists
                  end

med_cab_brush_teeth_vox_sloppy
                  loadl v1 #678
                  loadl v2 #679
                  loadl v3 #0      ;680
                  jump @generic_fm_smartlists
                  end
```

Example 5 – baby crying

The baby winds up and cries. The baby winds up a random amount of times and then screams over and over.

```
vox_baby_cry_windup
                              ; Kill this actor's vocals
                              seqgroup_kill Instance

                              set_loop
                              ; Get smartlist
                              loadl v1 #92
                              smart_setlist v1
                              smart_choose v1
                              set patch v1
                              note_on v1
                              wait_samp

                              ; 80% chance of looping here, 20% chance of
following thru to scream
                              loadl v1 #1000
                              loadb v2 0
                              rand v1 v2 v1
                              loadl v2 #200
                              cmp v1 v2
                              iflt @_vox_baby_cry_windup_scream   ;scream
                              loop
      ;don't scream

_vox_baby_cry_windup_scream
                              loadl patch #1235
                              note_on v1
                              wait_samp

                              loop
                              end
```