Getting Started

The purpose of this document is to help GAM150 students incorporate FMOD Core into their projects. It assumes that you will be using the FMOD files provided in the “FMOD Walkthrough and Demo” zip file. To download the API yourself, see the end of this document.

Documentation

The FMOD documentation can be found on their website, https://www.fmod.com/. Follow these steps to find the API documentation on the website:

1. Under the “Learn” menu on fmod.com select “Documentation”.
2. Click on the “Learn More” link under “FMOD Engine”.
3. In the right menu area, click on either “3. Core API Guide” or “7. Core API Reference” to expand those sections.

Copy FMOD Files

You will need to add some FMOD files to your project and commit them to your SVN repository. It is recommended to copy the “FMOD” folder from the “FMOD Walkthrough and Demo” into your project as-is, but you can rearrange the files however you would like.

There are two sets of files, the headers and the library files (.dll files and .lib files). In the “FMOD” folder, the headers are in the “inc” folder and the library files are in the “lib” folder.

Eventually, the .dll files will need to exist in the same folder as the .exe compiled for your project, specifically the “Debug” and/or “Release” folders. A number of methods exist for doing so, including:

- Add a post-build step that “xcopy”’s the required .dll from the FMOD folder (recommended)
- Commit copies of the .dll’s within the “Debug” and “Release” folders to your SVN

Visual Studio Integration

Visual Studio needs to be told how to find the FMOD files and, in the case of the library (.lib) file, which library file to include. When adding the library files, both .lib and .dll, it is important to note that there are two possible options:

- fmod_vc.lib, fmod.dll
- fmodL_vc.lib, fmodL.dll

The “L” option enables output to a log file, for the purposes of debugging problems. Using this option can cause noticeable delays during file loading and is best avoided except when initially implementing FMOD support or when trying to track down audio issues.

The first step is to instruct Visual Studio on where to find the FMOD files.

1. In the Solution Explorer right-click on your project (the second entry in the view tree).
2. Select "Properties".
3. On this page you will see a "Configuration" drop-down. You will need to repeat steps 4 - 10 for both the "Debug" and "Release" configurations. Alternatively, you may select “All Configurations” to set both configurations simultaneously.
4. Select "Configuration Properties/VC++ Directories".
5. To the right of "Include Directories" click within the text field and open the drop-down which appears.
6. Select "<Edit...>".
7. Add the path to the included files (e.g. “.\FMOD” or “.\FMOD\inc”).
8. To the right of "Library Directories" click within the text field and open the drop-down which appears.
9. Select "<Edit...>".
10. Add the path to the included files (e.g. “.\FMOD” or “.\FMOD\lib”).

The next step is to instruct Visual Studio to include the FMOD library during the link step.

1. In the Solution Explorer right-click on your project (the second entry in the view tree).
2. Select "Properties".
3. On this page you will see a "Configuration" drop-down. You will need to repeat steps 4 - 7 for both the "Debug" and "Release" configurations. Alternatively, you may select “All Configurations” to set both configurations simultaneously.
4. Select "Configuration Properties/Linker/Input".
5. To the right of "Additional Dependencies" click within the text field and open the drop-down which appears.
6. Select "<Edit...>"
7. Add the name of the .lib to the list.

Compiler Warning

One of the FMOD header files will cause a compiler warning. You should disable this specific warning by either adding it to the disabled warnings field in your project settings, or by adding the following line at the top of the file where you are including the FMOD headers.

```
#pragma warning ( disable : 4201 )
```

Basic Implementation

The “FMOD_Walkthrough” project demonstrates the basics of how to initialize and update FMOD, and play a single sound, as well as the Visual Studio settings. Other examples can be found in FMOD’s documentation. The following sections explain the code in the demo project.

Initialization

The FMOD initialization code should be called only once, when the game engine first starts. There are two steps: first the system is created, and then it is initialized.

When initializing the system, the second parameter controls the number of sounds that can play simultaneously. This number can be raised if necessary: it's usually not a good idea to play too many sounds at once, so only increase the limit if you know it will be needed.

It is good practice to always check the error code returned from FMOD. The ERRCHECK function in the demo program shows how to get a text string from the error code.

```
FMOD_SYSTEM* soundSystem;

result = FMOD_System_Create(&soundSystem, FMOD_VERSION);
ERRCHECK(result);

result = FMOD_System_Init(soundSystem, 32, FMOD_INIT_NORMAL, 0);
ERRCHECK(result);
```

Update

FMOD requires periodic updates. If you do not call FMOD's update function every game loop, it will not function correctly.

```
result = FMOD_System_Update(soundSystem);
ERRCHECK(result);
```

Loading and Playing a Sound

Sounds are loaded and stored in an FMOD_SOUND object. Whenever a sound is played, it uses an FMOD_CHANNEL object. This channel can be stored if you need to change settings on the sound as it's playing, or can be ignored if it is a fire-and-forget sound effect by passing in NULL instead of the address of a variable.

```
FMOD_SOUND *sound = 0;
FMOD_CHANNEL *channel = 0;

result = FMOD_System_CreateSound(soundSystem, "sample_beep.wav",
  FMOD_LOOP_NORMAL | FMOD_2D, 0, &sound);
ERRCHECK(result);

result = FMOD_System_PlaySound(soundSystem, sound, 0, false, &channel);
ERRCHECK(result);
```

Shutdown

As your game exits, it is a good idea to release all allocated resources. Make sure you do this order: sounds should be released first, then the system is closed, then it is released.

```
// Release all sounds that have been created
result = FMOD_Sound_Release(sound);
ERRCHECK(result);

// Close and Release the FMOD system
result = FMOD_System_Close(soundSystem);
```

```
ERRCHECK(result);
result = FMOD_System_Release(soundSystem);
ERRCHECK(result);
```

Additional Concepts

Music vs. Sound Effects

Using the FMOD Core API it is possible to implement music and sounds effects using the same functions. However, there are several factors to consider when differentiating between these two categories of audio assets. The examples below are "typical" for GAM150 projects but should not be considered true in all possible cases, most notably professional AAA titles.

- Typically, music tracks are longer and require more memory than sound effects.
- Typically, only one or two music tracks will be playing at any given time, while many sound effects may be playing simultaneously.
- Typically, music tracks loop while most sound effects are "one shot". This leads to the decision of whether to use `CreateSound()` or `CreateStream()` before playing an audio resource.
- `CreateSound()` loads the specified audio resource into memory. This is ideal for sound effects, given that they are typically smaller and more numerous than music tracks.
- `CreateStream()` opens the specified audio resource and prepares it for streaming off of a mass storage device (i.e. hard drive). This is ideal for long music tracks which, due to their larger size, should not be loaded into memory. Additionally, only one or two are playing at any given time, so contention for access to the mass storage device should pose no problems.

Within an Alpha Engine project, one approach would be to create a single, reusable Sound object for playing streaming music tracks and an array of Sound objects for preloading sound effects. One would call `CreateStream()` using the reusable Sound object each time a new music track needed to be played. The array of Sound objects could be initialized just once, at the beginning of the game, using `CreateSound()`. The later assumes that your game has just one fixed set of sound effects that remain unchanged during gameplay. A more complex solution will be required if sound effects need to be loaded dynamically during gameplay.

Looping

When looping music and sound effects it is usually necessary to maintain a pointer to the Channel object being used to play the audio asset. Having access to this Channel object allows you to stop, pause or otherwise change the properties of the looping asset.

Typically, there is a 1-to-1 relationship between Sound and Channel objects when working with simple music tracks. However, this is often not the case when using sound effects. As a result, looping sound effects may require a bit more work than "one shot" sound effects. Note that the Channel objects used for "one shot" sound effects can be considered as "fire and forget". The Channel object allocated for the sound effect will be reclaimed by FMOD's update function once the sound effect is no longer playing.

Creating A Sound Module

The sound module should have some sort of data structure (for example a linked list or an array) to hold all the active sounds that are in your game. In addition to that, the FMOD structure, "`FMOD_SYSTEM`" should also be saved as a static variable in the module because this would only be initialized once at the beginning of the game.

While creating a sound module it's usually helpful to have a struct for each 'sound' that would hold both FMOD structs and your own data about the sound. The following shows an example of what structure can be used:

```
typedef struct Sound
{
  const char    *soundPath;     // file path
  const char    *name;          // sound name to be used a future reference
  FMOD_SOUND    *fmodSound;     // FMOD struct managing FMOD stuff
  FMOD_CHANNEL  *channel;       // another FMOD struct managing FMOD stuff
  SoundType     soundType;      // whether a song is effect or a background sound
}Sound;
```

This would essentially be the struct that you would maintain in a data structure to access and play particular sounds in your engine.

Other Tips:

- Make sure you release each sound only once (if your game crashes on exit because of FMOD this is the most likely problem).
- Do not call `FMOD_System_CreateSound` or `FMOD_System_CreateStream` twice on a sound (you'll end up playing the sound the number of times you call that function)
- For background sounds, create the sounds in the gamestate load functions and release them in the gamestate unload function.
- For gameobjects that have sound effects, create the sounds in the gameobject's initialize and release them in the gameobject's shutdown.
- On exiting your game, make sure ALL the sounds you created are released using the "`FMOD_Sound_Release`" function and after that use the functions "`FMOD_System_Close`" and "`FMOD_System_Release`" to shut down the FMOD system.
- Use `FMOD_RESULT` as a way to check for possible errors in the FMOD system. Every FMOD function returns `FMOD_OK` if it was successful in doing its task. If an error has occurred, use `FMOD_ErrorString` to convert `FMOD_RESULT` to a `char *` to print it out. It can be used like this:

```
printf("(FMOD): %s", FMOD_ErrorString(ErrorResult)); // ErrorResult is an FMOD_RESULT
```

Downloading the API

The FMOD sound library may be downloaded from http://www.fmod.com/. Users must create an account and log in before downloading files.

- Go to http://www.fmod.com/.
- Click on the "Download" link.
- Click on the "FMOD Engine" section.
- Click on the "Download" link next to "Windows".
- Download and install the FMOD Studio API.
    - Typically, the installed files may be found in the following location:
        - "C:\Program Files (x86)\FMOD SoundSystem\FMOD Studio API Windows"

The FMOD documentation can be found inside the "doc" folder in the FMOD Studio API installation.

The library files to include with your Visual Studio project can be found in the following locations:

- Include files
    - Relative path: ".\api\core\inc"
    - Required files: *.h
- Library files
    - Relative path: ".\api\core\lib\x86"
    - Required files: fmod_vc.lib, fmodL_vc.lib
- .DLL files
    - Relative path: ".\api\core\lib\x86"
    - Required files: fmod.dll, fmodL.dll