Touhou Vorbis Compressor

Technical documentation

Nmlgc, May 2011

Contents

Disclaimer

There is no "right" way to do code. There are always multiple ways to go about creating a program, with no single one being objectively “better” or “worse”. Hence, there's also no absolute definition what is "good" or “bad” code. For every aspect of your program, you will always find some guy who thinks that this implementation is completely wrong. But only if this implementation is responsible for a certain flaw that arises when working with the program (in any way), and you could have done it better, you have a reason to call the code “bad”.

This document only supplements the source code and I'm not including any of it here. It's mainly aimed at newbies to game patching. I try to explain all the immediately related functionality which may not be clear, but I still have to expect some general background in programming.

Integer constants (e.g. byte offsets and sizes) are always noted in hexadecimal notation, prefixed with 0x.

1. Overview

The general aim of this project is to compress the lossless BGM used in the Touhou game series to a lossy format (Ogg Vorbis), and patch the games to be able to read from this file. This will save a huge percentage of disk space for people not interested in immediate data transparency or lossless background music (which I estimate to be the overwhelming majority of the fan base). Also, it will help to educate noobs and raise awareness as to why these games are so large when compared to their actual content.

To get there, we need to care about the following:

- How do we manage to add custom code to the (compiled) games?
- How do we change the original BGM files?
- What game-specific code do we need?

2. Patching framework

This patch should be implemented with four aims in mind:

1. The patch should be as transparent and invisible as possible, requiring little effort on the part of the user. It should "just compress the BGM". We don't have to keep lossless BGM reading support around.
2. This will not be the only Touhou patch. Often, there are a multitude of other patches (Translation patches, vsync patches, bug fix patches, to name but a few). Since we're replacing BGM data which is usually not affected by other patches, we should be able to cooperate with all of them.
3. Multiple Touhou games with more or less different BGM engines have to be supported with as few changes to the source code as possible.
4. (Of course, all the changes we make to a game have to be expressed as code. The patching process should not require human interaction.)

All of these aims require compiling our custom code into one or more DLLs. By splitting the functionality into more DLLs, we can physically separate game-specific code from common code used across multiple patches.

From there on, we have three methods how to implement the patch itself:

- ➢ **Binary modifying an input DLL file name in the EXE file.** We scan each game executable file for the name of the "old" DLL whose functions we are replacing (usually `kernel32.dll`) and change it to the name of our patch DLL. The patch DLL has to export all the functions of the "old" DLL in this case.
  As some other patches perform checksum verification on the game EXE, we then either have to create a new one, or overwrite the current one and make a backup of the old one and asking the user to restore it in case it's needed.
  This method is fine if this were the only patch people use, but as it certainly isn't, this will probably get really annoying.
- ➢ **External code injection (as used by the fighting game English patches).** We write a loader program which injects a DLL with our custom code into the game, using the `WriteProcessMemory` API. The game then always has to be started from this loader to get BGM. If there are multiple valid game executables, we display a list box, asking the user to select one.
  This method has shown to not work on certain Slackware-based Linux distributions, which alone is a reason for me to not use it. Also, if some hackers suddenly decide to use this method for their patch, we suddenly end up with two layers of injection and are… pretty much fucked.
- ➢ **Proxy DLL.** We "fake" one of the system DLLs referenced by the game by placing a custom DLL with the same name into the game's directory. This DLL has to export the same functions as the original, so we have to

dynamically load the "real" system DLL internally and wrap the functions we replaced. Any code in our custom DLL will then be automatically made available for the game by the Windows PE loader, without us having to do any custom loading.

It's obvious that the Proxy DLL method is the best one for this patch, being the only one that fulfills all four aims and especially excels in transparency. However, it's important to note two drawbacks on certain operating systems:

- Wine, with normal behavior, ignores custom versions of system DLLs placed in an application's directory, because they are assumed to contain Win32-specific code. Hence, our entire patch gets ignored.
  Solution: Add a native->buildin override rule for the proxy DLL to the Wine registry, either by using `winecfg` or through a registry file. This will be the only instance of manual user interaction we *can't* get rid of with this method, but since Wine users are accustomed to things like this, it shouldn't be too much of a problem. Heck, the patching tool even offers the option to directly do it.

- On Win9x, there are some issues with dynamically loading a DLL with the same filename as a DLL which is already loaded by the current process. While we can load it and get a valid handle, `FreeLibrary` will fail and the DLL will remain in memory. Subsequent calls to `LoadLibrary` then fail until the system is restarted, causing no BGM to be played when the game is started a second time.
  Solution: On the affected systems, create a copy of the system DLL prefixed with `sys_` in the game directory and load this one instead when running.
  (If anyone asks why the hell I'm even supporting those legacy OS: th06-th09 run on Win9x as well. Since old PCs usually have less disk space available, this patch may be especially interesting for those.)

So, we just intercept the DLL where the original functions are stored, and we have our patch? Well, not quite.

The Windows file functions used by most of the games are stored in `kernel32.dll`. However, this is the most important Windows system DLL and can't be intercepted, because the Windows PE loader doesn't even reload it for a new process. And even if it would, it would be a pain creating wrappers for 954 functions when we only want to patch 4.

This might look like an inherent flaw of this method at first. But as we can intercept any other DLL linked by the game and inject all the code we want, we just need the right code to re-link those functions.

2.1. Patching the import table

Dependency Walker (http://www.dependencywalker.com/) is a very useful tool to list the external DLLs and their functions an application depends on. When opening a program and listing its DLLs, the following information is shown for each imported function:

- A 16-bit "ordinal" number. Refers to the index of this function in the complete list of exported functions. Some functions (e.g. `DirectSoundCreate8`) are only linked based on their ordinal number. If we create a proxy DLL containing such functions, we have to make sure that the function gets exported with the correct ordinal value.

- A 16-bit "hint" number. *Can* be used to quickly identify a function, but that's discouraged [citation needed].

- A string representing the exported function name

- The entry point, i.e. the first byte of the function.

The upper table shows the requirements as defined by the program, the lower table shows the contents of the DLL.

Changing what happens when a certain function is called basically comes down to replacing its *Entry point*[1] with a pointer to another function. Whenever a DLL is loaded, its `DllMain` function is called with the call reason `DLL_PROCESS_ATTACH`. This is where we change the function pointers.

1 Don't let that "Not Bound" in the upper table confuse you. It just indicates that the function pointer is not set, and gets defined once a DLL is instantiated after starting the application.

First, we have to obtain pointers to the "originals" of the functions we want to patch. These are used to identify a function when patching the entry point, and, of course, to access the original code of the function. To standardize the code, the macro DLL_FUNC (defined as [DLL name w/o extension]_[function name]) is used to name a pointer to an external function. For example, the original CreateFileA from kernel32.dll is referred to as kernel32_CreateFileA, whereas our custom version simply becomes CreateFileA[2]. This document also uses this scheme.

IMAGE_IMPORT_DESCRIPTOR (All pointers are relative to the module handle)	
OriginalFirstThunk	Pointer to null-terminated list of pointers to IMAGE_IMPORT_BY_NAME structures, containing the hint number and exported function name.
FirstThunk	Pointer to a null-terminated list of function pointers for this DLL.
Name	Pointer to DLL file name

2.1.1. Finding a DLL's import descriptor

The import descriptor of a DLL serves as the central point for all the data associated with function imports. Both the OriginalFirstThunk and the FirstThunk lists are in the same order, so the same list index always refers to the same function.

A list of all import descriptors, terminated by a zeroed-out structure, is stored in the EXE file, so we now have to get to it.

PE File format (simplified for our purpose) Refer to MSDN for the full specification.	
IMAGE_DOS_HEADER	
. . .	
DWORD* e_lfanew	Pointer to IMAGE_NT_HEADERS
. . .	
IMAGE_NT_HEADERS	
DWORD Signature	
IMAGE_FILE_HEADER FileHeader	
IMAGE_OPTIONAL_HEADER OptionalHeader	
IMAGE_OPTIONAL_HEADER	
. . .	
IMAGE_DATA_DIRECTORY DataDirectory[0x10]	Constant array of "data directory" entries recognized by the Windows PE loader.
IMAGE_DATA_DIRECTORY	
DWORD VirtualAddress	
DWORD Size	

The so-called "module handle" obtained by GetModuleHandle is in fact a pointer to the base address of the memory-mapped EXE file which called the active process. With this handle, we have access to its contents.

Now, we have to walk through the PE file structures according to the table on the right. IMAGE_DOS_HEADER is the first header, stored directly at the module handle address. The VirtualAddress member of DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT] contains a relative pointer to the first entry of the import descriptor list.

Then, we traverse this list, compare the Name of the current import descriptor to the one of the requested DLL, and there we have it.

2.1.2. Modifying the function pointers

From there on, we have two ways to patch the function pointers:

- Traverse the FirstThunk list, compare the pointers to the original ones obtained from the DLL, and replace the right ones with pointers to our custom functions.
- Traverse both the OriginalFirstThunk and FirstThunk lists in parallel. Compare the function name in the IMAGE_IMPORT_BY_NAME structures to the name of the function to patch. If it matches, replace the pointer at the current position in the FirstThunk list.

2 There aren't that many cases where it really matters which function we're talking about, so that scheme is pretty convenient.

Of course, the first one performs way better, and is the one that's normally used. The second one is only used on Win9x, where the function pointers in the `FirstThunk` list are still zeroed on `DLL_PROCESS_ATTACH`. And there we go, functions patched.

All this functionality is stored in the `patchlib` convenience library, offering a high-level import table patching framework. Just fill out the structures and patch away. Of course, you're free to use this library for any project under my "This is the internet, do with it what you want, I can't stop you anyway" license. (It's still common courtesy to at least credit me somewhere visible.)

3. BGM file encoding

3.1. Bitstream chaining

Naively, we could just treat the entirety of each BGM file as one wave file and encode it. This would indeed work, but we want to go one step further.

The Ogg specification (http://www.xiph.org/ogg/doc/oggstream.html) allows multiple bitstreams (with one bitstream usually being a single "track" with metadata, format information, and sound data) to be concatenated to form a single new output stream. Decoders should handle playback of such streams transparent and gapless.

Using concatenated bitstreams has a number of advantages, especially in this case:

- Seeks to the single tracks are faster than using one single bitstream, because all the seek points lie on bitstream boundaries and can be found quicker. (Although it shouldn't be that relevant of a gain.)
- A standards-compliant media player[3] can identify each bitstream and provide direct access to it.
- Music Room Interface can, in chained bitstream mode, perform Ogg extraction of the individual themes without re-encoding (except for fades).
- Track lengths and looping offsets are directly available from the Ogg file, which will turn out to be necessary for th06 and th075.

The payoff, though, is a noticeably increased initial game loading time. `ov_open_callbacks` reads in the whole bitstream structure of the file, and since there is no indexing table, it will take a while to scan through the file. However, even on the worst machine I tested this on (AMD Duron 1.3 GHz single-core processor, 128 MB SD-RAM, Windows 98 SE), this impact was not too large.

3.2 Implementation

The layout we use consists of two bitstreams for each track, one for the intro, and another for the looping part. The patching tool uses `bgmlib` with the same info file repository as Music Room Interface to access the BGM info data, and also tags each bitstream with the corresponding track title for good measure.

Encoded files receive the extension "`.ogg`" instead of "`.dat`" or "`.wav`" to distinguish them from the originals.

As explained above, the chained bitstream layout makes `ov_open_callbacks` a fairly expensive operation, as opposed to `CreateFileA` and `mmioOpenA`, which are fairly cheap calls. All of the games open and close BGM files multiple times before actually starting to read wave data. Hence, we only close and open BGM files when the filename differs from the last opened file. And to not leak anything, we always close our handle when our DLL is detached from the process.

3 So far, only foobar2000 and Winamp are known to correctly display the encoded BGM files as having multiple bitstreams. foobar2000 shows and handles each stream as a single track in the playlist, and Winamp offers tag display for each bitstream, and always displays the tags of the currently playing bitstream in its main display. Any other software using libvorbisfile should at least handle those files indistinguishable from a normal, single bitstream file. Let's hope that this project improves multiple bitstream handling with the software that currently doesn't support it.

Since the Ogg Vorbis decoding is constant across all games, it's compiled to the external `libvorbis_decode.dll`. This is a custom build containing only the decoding engines of libogg, libvorbis and libvorbisfile. The normal, dynamic build option offered by xiph's official Visual C++ projects always includes the encoding engine as well, but this adds ~1.4 MB to the DLL which we won't use anyway.

Custom callback functions are created to wrap the C-style file handling functions around the Win32 ones. We don't want to depend more on the C runtime than necessary.

Following is an overview of the game-specific implementation details, sorted by increasing difficulty.

4. Team Shanghai Alice games, starting from th07

This is the most common instance and, fortunately, the easiest one to implement.

4.1. Original BGM engine

thbgm.dat contains the lossless BGM as a continuous stream of raw PCM data. A `0x10` byte header at the beginning identifies the format ("ZWAV") and contains two bytes which identify the game.

The PCM data is indexed by `thbgm.fmt` table inside the game data file.

Format of `thbgm.dat`

Offset	Bytes	Data
0x00	0x4	"ZWAV"
0x04	0x4	???
0x08	0x2	Game identification bytes (0x08 = fractional part, 0x09 = integer part)
0x0A	0x6	???
0x10	until EOF	Raw PCM data, indexed by thbgm.fmt

4.2. Encoding

thbgm.dat gets encoded from start to finish, according to our bitstream layout. We have to be careful though, as some games don't physically store the tracks in Music Room order. Therefore, on a track boundary, the encoder chooses the next track info structure based on the current byte. The ZWAV header gets "encoded" as a single bitstream to ensure matching sample offsets[4].

The game ID bytes from the header are preserved, and saved binary in the Vorbis comments of each bitstream. This is necessary for th07, which will reject the file otherwise, and Music Room Interface, as it uses these bytes to differentiate between the games, of which many use the same file name for the BGM. This has one major drawback, though. Non-side games always have a null byte as their first ID byte, which gets interpreted as string terminator by the reference implementation of `vorbis_comment_add_tag`. To circumvent this, we change null bytes to `0xFF` on encoding, and change them back when we read the values again.

4.3. Implementation

Everything is handled by these four functions:

- `CreateFileA`
- `ReadFile (HANDLE hFile, LPVOID lpBuffer, DWORD Read, LPDWORD lpRead, LPOVERLAPPED lpOverlapped)`
- `SetFilePointer (HANDLE hFile, LONG lDistanceToMove, PLONG lpDistanceToMoveHigh, DWORD dwMoveMethod)`
- `CloseHandle`

It should be obvious how these functions operate (in contrast to a certain different file access engine). Patching should be as easy as decoding into the `ReadFile` buffer instead of directly reading from the file.

4 This was probably a stupid decision. It adds ~5 KB unnecessary overhead, and many less standard-compliant players won't play back anything from the file, as they're stuck with 4 "null data" samples. And subtracting the 0x10 bytes from the offsets wouldn't have been critical, but oh well.

SetFilePointer always returns the new file pointer, so calling it with lDistanceToMove = 0 and dwMoveMethod = FILE_CURRENT is equivalent to a "tell" or "GetFilePointer" function.

Obviously, these functions are called for all game files, so we need a way to differentiate between "real" file handling and BGM decoding.
Since we have to change the BGM file name anyway, CreateFileA checks the filename (without the directory part, of course) for the substring "bgm", which appears in all BGM files of the supported games (Touhou main series uses thbgm.dat, th075 uses th075bgm.dat, Uwabami Breakers uses albgm.dat). The function then returns a pointer to the global OggVorbis_File object as the "file handle". Other file functions check for this handle, and then either decode the Vorbis file, or default to the normal file handling versions of those functions.

On SetFilePointer, we need to take care to convert the seek distance from byte to sample format for ov_pcm_seek, and back for the return value. And that's basically all we need.

dsound.dll is chosen as the proxy DLL, as it only has one required export (DirectSoundCreate8, linked as ordinal #11), intercepting it enables us to implement a performance hack explained below, and its name already signals a sound patch. In the end, this patch configuration results in these DLLs (indentation shows the dependencies):

- dsound.dll – Proxy DLL, containing a performance hack explained below
    - thvorbis.dll – Patches the kernel32 file functions to add Ogg Vorbis decoding
        - libvorbis_decode.dll – Ogg Vorbis decoding engine

4.4. Increasing the performance

All of these games use a 0xAC440 byte large streaming buffer for the BGM, holding exactly 4 seconds of music[5]. This buffer is completely filled when a song changes. On my machine (which is fairly low-end by today's standards), this causes a noticeable drop of about 7 frames. But we want to make a quality patch, so it'd be better to get rid of that. And since we're using dsound.dll as proxy DLL anyway, we can include a little performance hack here.

This comes down to restricting the size single buffer lock calls (IDirectSoundBuffer::Lock) to (BGMBufferSize / 4) bytes and keeping track of the byte offset ourselves[6]. Intercepting COM classes works similar to the interception of normal functions. We have to create a custom class derived from the original one, which saves a pointer to the old class to access the original functionality. And since this is COM and stupid per definition, we also have to create wrappers for every other function we don't want to intercept.

Then, we have to intercept the entire route that leads to the creation of the object we want to change. In this case, this is DirectSoundCreate8, which now always has to return a custom IDirectSound8, which in turn intercepts IDirectSound8::CreateBuffer. If the buffer size matches 0xAC440, we create an instance of our "custom" IDirectSoundBuffer object on top of the normal one, and return a pointer to our custom class.

Performance hack interception

(green = changes to the function)

DirectSoundCreate8

return (custom IDirectSound8)

↓

IDirectSound8::CreateBuffer

BufferSize == 0xAC440 ?
return (custom IDirectSoundBuffer)

↓

IDirectSoundBuffer::Lock

Do performance hack stuff

When intercepting COM objects, we have to be careful of the QueryInterface method though. This method can be used to create instances of other COM objects installed in the system registry. If the passed GUID matches the one of our intercepted class, it'd be better to return a (new) instance of our custom class instead.

5 Same size is also used for th13's Spirit World themes. Since they are recorded with half the sample rate, the buffer holds 8 seconds there.
6 This works because the game sets up buffer playback notifications using the IDirectSoundNotify8 interface, with the notification positions placed in intervals of (BGMBufferSize / 0x10). This means that a BGM stream event is fired every time (BGMBufferSize / 0x10) bytes were played back, making (BGMBufferSize / 0x8) the theoretical minimum locking threshold.

4.5. Pitfalls

- th07 shows an error message on startup, faulting a version mismatch of the BGM file.
  Solution: This seems to be the only instance where a game actually wants to verify the ZWAV header.
  We need to catch the case where the game wants to read `0x10` bytes from the BGM file and fill the buffer with a faked header constructed from the `ZWAV` Vorbis tag.
- If the BGM file can't be found and `kernel32_CreateFileA` returned `INVALID_FILE_HANDLE`, the game crashes when pausing the game. This happens with every single supported game.
  Solution: Catch this case and return NULL instead.
  This allows this patch to be used as a "no-music bug fix", so a custom build configuration was created to compile a DLL (`th_nomusic.dll`) for exactly this case, not containing any Ogg Vorbis calls. To use it with a (un-encoded, lossless) game, copy this DLL into its directory, and rename it to `dsound.dll`.
- For some reason, th13 causes BGM glitches with the performance hack after pausing and resuming the game.
  Solution: Trial and error resulted in `(BGMBufferSize / 2) - (BGMBufferSize / 8)` as a stable value. A custom version of `dsound.dll` using this value can be created by adding the preprocessor definition `SLOW`. It's used by the [slow] patch class.

If this were everything, we could just stop here and call it a day. We have a perfectly working and probably even upwards compatible patch. But unfortunately, there are other lossless games with different engines waiting to be patched.

5. The mmio engine

The standard .wav files in the `bgm` directory of th06 and the `WAVE` directory of Kioh Gyoku probably already tell that these games use an entirely different BGM engine. Instead of the normal file functions, the I/O functions from the Windows multimedia system ("mmio") are used for file access.

This one seems to be a relic from the early Win32 revolution. Nowadays, you can barely find tutorials on how to exactly use it for streaming audio, which isn't all too obvious. This left most of the figuring how this actually worked to trial and error[7]. While it claims to be "format transparent", you have to know about at least about the RIFF format anyway, and it way overcomplicates and obscures its handling. Hell, it's even marked as deprecated now.

(Fortunately ZUN realized this system is shit and changed it for th07 and every game since then.)

The functions are, in order of typical usage:

- `mmioOpenA (LPSTR szFilename, LPMMIOINFO lpmmioinfo, DWORD dwOpenFlags)`
  Opens a file. `lpmmioinfo` can be `NULL` if no custom features are requested. Internally creates a `MMIOINFO` structure for the new file, associated with the returned handle. If `dwOpenFlags` contains `MMIO_ALLOCBUF`, (which is always true in our cases), a `MMIO_DEFAULTBUFFER` (8K) byte I/O buffer is allocated for streaming.

- `mmioDescend (HMMIO hmmio, LPMMCKINFO lpck, LPMMCKINFO lpckParent, UINT wFlags)`
  How Microsoft describes this function:
  "*Descends into a chunk of a RIFF file that was opened by using the **mmioOpen** function. It can also search for a given chunk.*"
  What it actually does:
  Reads the next RIFF chunk from the current file position, fills the `lpck` structure with its parameters, and moves the file position to `lpck->dwDataOffset`.

- `mmioRead (HMMIO hmmio, char* pch, LONG cch)`
  How Microsoft describes this function:
  "*Reads a specified number of bytes from a file opened by using the **mmioOpen** function.*"

7 This also explains all the debug code. I left it there in case someone wants to research it further.

What it actually does:
Well, just that. But for some reason it's only used to read chunk data, *not* the BGM data itself.

- `mmioAscend (HMMIO hmmio, LPMMCKINFO lpck, UINT wFlags)`
  How Microsoft describes this function:
  "Ascends out of a chunk in a RIFF file descended into with the ***mmioDescend*** *function or created with the* ***mmioCreateChunk*** *function."*
  What it actually does:
  *"****mmioAscend*** *seeks to the location following the end of the chunk."* Says the **Remarks** section at the bottom of the MSDN page. Why don't they put *this* description on the top instead.
- `mmioSeek (HMMIO hmmio, LONG lOffset, int iOrigin )` – At last, an obvious function.
- `mmioGetInfo (HMMIO hmmio, LPMMIOINFO lpmmioinfo, UINT wFlags)` – Retrieves the internal `MMIOINFO` structure associated with `hmmio`.
- `mmioSetInfo (HMMIO hmmio, LPMMIOINFO lpmmioinfo, UINT wFlags)` – Sets the internal `MMIOINFO` structure associated with `hmmio` to the values in `lpmmioinfo`.
- `mmioAdvance (HMMIO hmmio, LPMMIOINFO lpmmioinfo, UINT wFlags)`
  How Microsoft describes this function:
  "*Advances the I/O buffer of a file set up for direct I/O buffer access with the* ***mmioGetInfo*** *function."*
  What it actually does:
  Reads data according size of the input buffer from the file and (instantly!) fills `lpmmioinfo-> pchBuffer`.
- `mmioClose (HMMIO hmmio, UINT wFlags)` – Closes the file.

See? Why don't they just read the file normally.

5.1. Patching

As all of those functions are contained in `winmm.dll`, it suggests itself to simply create a proxy DLL directly exporting those functions. But unfortunately, DirectSound depends on `winmm`, and this would, again, result in creating an endless number of function wrappers, even though we only want to intercept 10 functions. Therefore, we're using the same procedure as with the main Touhou games, that is, intercepting `dsound.dll` and patching the `winmm.dll` import table.

- `mmioOpenA`: The mmio stream buffer and internal info structure of the song are stored globally. We don't need more dynamic allocations than necessary.
- `mmioRead` and `mmioDescend`: It's crucial to fake all the involved RIFF chunks correctly. The games depend on their correct values and a single mistake usually breaks the BGM engine early on. Fortunately, the `WAVEFORMATEX` structure, read by the sole call to `mmioRead`, can be easily rebuilt using the data from `vorbis_info`. `mmioDescend` uses a "chunk sequence" variable (which is reset when a new file is loaded) to always return the correct chunk. `ov_pcm_total` is used to return the exact size of the uncompressed BGM.

As for the encoding, we don't have to take care of anything special here. We just normally encode every wave file according to the bitstream layout, which results in each encoded file having two bitstreams.

So much for the basic implementation, shared across both games.

6. th06

Alright, so we just intercept `mmioAdvance` to include a decoding loop, and `mmioSeek` to include seeking now?

Well, turns out this game uses seeking in a quite stupid manner. It seeks the wave file back *to the beginning of the PCM data*, and then incrementally reads data until the loop point is reached (and probably even a bit more). This is too slow

(「上海紅茶館　～ Chinese Tea」 runs for **45 seconds** before reaching the loop point, we can't decode all of this in one go without noticeably dropping frames) and hence, I don't keep it that way.

The obvious solution is to lock the decoding until the loop point was reached. But this didn't work. Some of the audio data from the looping part is also read during the process and once the specified time is reached, the streaming buffer offset is manipulated to point to exactly that point. Or something to that extent. Anyway, I couldn't get it to without audible glitches.

So I did the sanest thing in this situation, and just moved the entire decoding in front of the `IDirectSoundBuffer::Unlock` call (refer to the th07+ performance hack section for how to implement this). All other mmio functions besides the ones described above serve a dummy function, only returning the right values where necessary.

This is also where the bitstream layout pays off: `mmioSeek` is only called when the BGM needs to loop. So, if the decoding reached the end of the audio file[8], we just seek back to the beginning of the current bitstream, which is, of course, the beginning of the looping part! No need to depend on any values from the game.

This entire implementation works a bit against the modular patch build structure though, as the `OggVorbis_File` object needs to be known to DirectSound. Therefore, the build configuration for this game unifies the whole patch (minus `libvorbis_decode.dll`, of course) to a single `dsound.dll`.

In the end, this patch configuration results in these DLLs (indentation shows the dependencies):

- `dsound.dll` – Proxy and interception DLL. Contains the basic mmio implementation, dummies out the rest, and intercepts `IDirectSoundBuffer` to handle the decoding
    - `libvorbis_decode.dll` – Ogg Vorbis decoding engine

6.1. Pitfalls

- For some reason, when the game opens a new track (at least on the title screen), it reads about a second of wave data, reopens the file and only then starts the proper BGM playback.[9] Because we prevent the repeated opening/closing in `mmioOpenA`, this cuts the previously read wave data.
  Solution: Lock the decoding based on the value of the chunk sequence variable used in `mmioDescend`. When looking at the log files, the proper BGM playback always starts after the fifth call to this function. Thus, the lock variable gets set to 1 after `mmioOpen` and to 0 after the 5th call to `mmioDescend`.

- If the game is started for the first time, it calls `CreateFileA` to verify the existence of the wave BGM, and defaults to MIDI if it's not there.
  Solution: Even though this is a relatively minor annoyance, I decided to patch it either way. It's just a matter of intercepting `CreateFile`, and replacing ".wav" with ".ogg" in the filename before calling `kernel32_CreateFile`.

7. Kioh Gyoku

For some weird, illogical reason, I find this game to be oddly fascinating. I don't know, it's just so dynamic and fluid. Which is why I added it to the list of games to support. Just like th06, it uses the mmio engine as well, but its implementation here is way better. Thus, we're properly patching it at the mmio level this time.

This game is pretty annoying to debug. First, the game enforces a DirectDraw full screen video mode, which is not that easy to get rid of. Fortunately, a certain program named D3DWindower (http://www.geocities.jp/menopem/) helps pretty well in this regard and attaching a debugger onto the windowed application isn't too much of a problem.

8 The original wave files always have a few extra seconds of music at the end, though. Since they are never played, they are effectively cut by the patching tool.

9 This would also explain why all tracks have a few seconds of silence at the beginning of the wave file…

Second, the game uses DirectShow as a *secondary*[10] BGM playback engine. It's required to play back `WAVE/kog_sj.wav` (the Seihou Project jingle), which is actually a MP3 stream inside a RIFF header, but it's also used for other tracks if mmio fails. This might lead to a falsely identified "patching success" if the original BGM files are still present, so it's better to get rid of them during the development. And we better don't rely on the correct DirectShow splitters for Ogg Vorbis being installed.

Other than that, there's really not much to say. Contrary to what the MSDN page for `mmioAdvance` suggests, the first byte to write is **not** `pmmioinfo->pchNext`, but the *start of the buffer. Always.*

In the end, this patch configuration results in these DLLs (indentation shows the dependencies):

- `dsound.dll` – Proxy DLL. Does nothing on its own, but adds a dependency for…
    - `thvorbis_kog.dll` – Patches mmio to call the Ogg Vorbis functions
        - `libvorbis_decode.dll` – Ogg Vorbis decoding engine

No *Pitfalls* section here! At least it's a quality game from a technical point of view.

8. th075

This gets a bit harder, as `th075bgm.dat` not only archives Wave files complete with RIFF header, but also has the loop information in RIFF cue chunks and a header with a file archive table. We have two methods here:

- Create a valid th075 archive, complete with header
- Encode it the same way as generic Touhou games and fake the archive structure on-the-fly

th075 archive format

Offset	Bytes	Data
0x00	0x2	File count
0x02	File count * 0x6C	Encrypted header
…		Archived files

As we have to add a lot of faking code for the RIFF chunks and the loop information anyway, I chose the second one, which increases compliance of the encoded file with playing software. But really, that one could have gone either way.

Like with th06, the bitstream layout already supplies us with the loop position data. This is pretty important here, as we're going to need the track lengths quite often. Normally, we'd have to add the total length of two bitstreams to get the length of one track. But in one instance (the credits theme), the song doesn't loop, and there's only the "intro" part, so we can't simply assume that one track consists of two bitstreams. Thus, the patching tool needs to include the source file name for each track (as in, `wave\bgm\sys00_op` for the title screen theme) when encoding the BGM. This way, we can compare the source names, and get the correct lengths and offsets for every track.

th075 archive table entry

Offset	Bytes	Data
0x00	0x64	File name, null-terminated
0x64	0x04	File size
0x68	0x04	File position

The game calls no more than the basic `kernel32` file functions also used in the th07+ patch class. But `ReadFile` also takes over the reading of RIFF chunks here. We have to include a multitude of checks based on the count of bytes to read (which is always constant for each "type of data") and return the corresponding faked data. It's fairly redundant to describe all of this here, so just refer to the source code if you're really interested what happens there. By the time it was working, I already forgot how and why all of this functioned anyway.

`SetFilePointer` with `dwMoveMethod = FILE_CURRENT` is only called to move between RIFF chunks, which we aren't using anyway. As this would only cut samples, we absorb all those calls, regardless of the move distance, and default to `ov_pcm_tell`.

10 I really can't imagine any situation where one BGM engine would fail and another one would succeed.

Adding a `dsound.dll` proxy to this game turned out to be fairly problematic. This game wasn't compiled with a dependency on this DLL, and only instantiates its `IDirectSound8` object through a COM request using `CoCreateInstance`. The GUID passed to this function is installed in the system registry and linked directly to the system DLL. No matter which functions we change, interception would only be possible by either adding some registry key, or caving into all this COM bullshit.

Thus, we simply intercept the next best DLL, which turns out to be `d3d8.dll`, with `Direct3DCreate8` as its only export. Yeah, I know that doesn't make it appear like a sound patch to the user anymore, but it's better than nothing.

In the end, this patch configuration results in these DLLs (indentation shows the dependencies):

- `d3d8.dll` – Proxy DLL. Does nothing on its own, but adds a dependency for…
    - `thvorbis_th075.dll` – Patches the `kernel32` file functions to fake the th075 archive header and RIFF chunk, and add Ogg Vorbis decoding
        - `libvorbis_decode.dll` – Ogg Vorbis decoding engine

… Unfortunately, the game still crashes on random, non-reproducible occasions inside the Vorbis sources. Well, all the hacking work just to have it end like this.