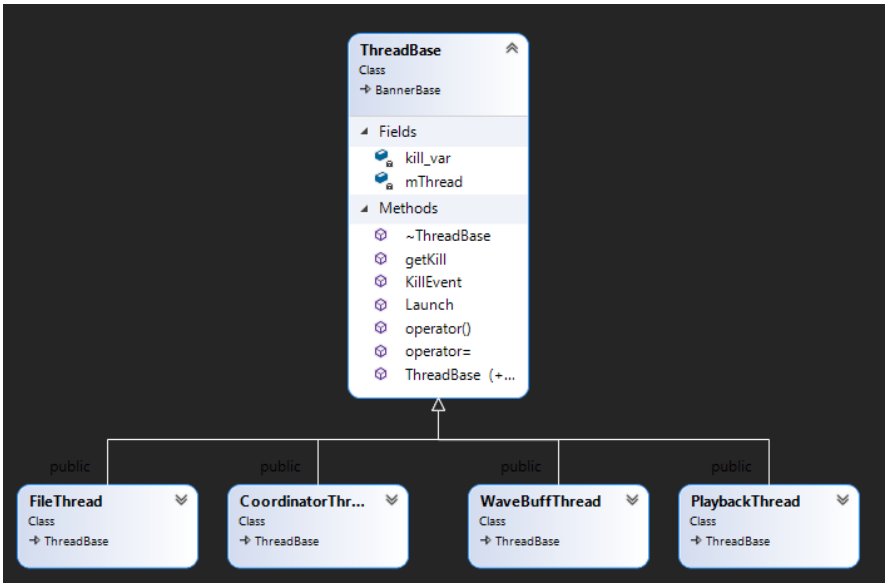
Jetson’s Playback w/ Multithreading

Joey Domino

Making this Jetsons playback system was a nightmare. I’m glad we didn’t have to keep the KillThread working. I’d rather not deal with getting all of these threads to close properly. Below I have laid out every major system that makes up the Jetsons playback system from creation, to playing the sounds, to cleaning up and closing everything out.

Main

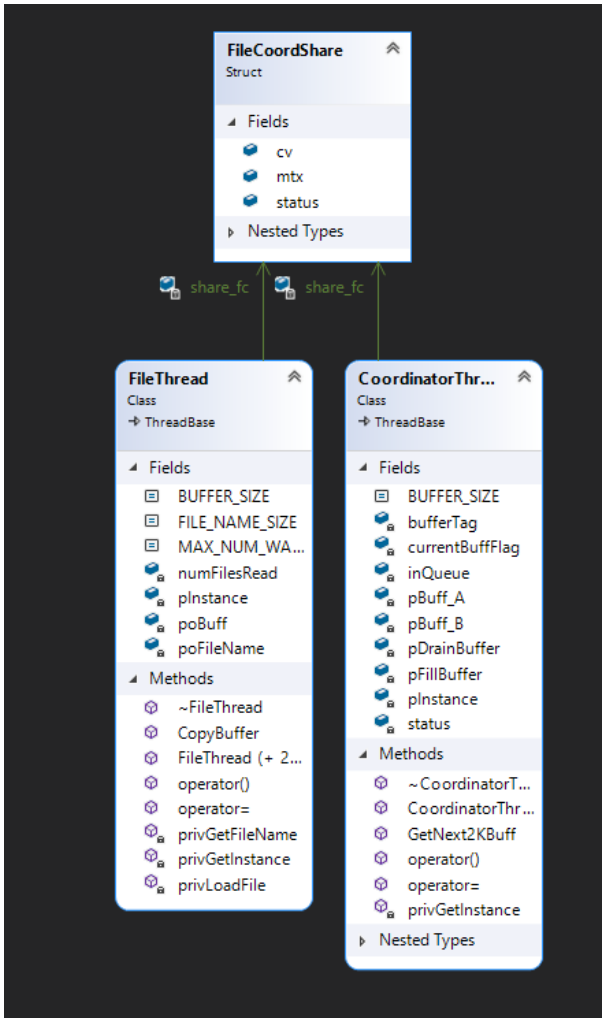
Main’s job is simple in this system. It is required to create the 3 main threads and their shared data between them. Their purposes will be covered within their sections. Once the threads are made, their respective launch() methods are called to kick off the threads proper.



Thread Base

Before diving into the threads themselves, we need to discuss their base class: ThreadBase. This thread holds all methods and data every thread will need. The base class handles creating the threads proper and joins them back to Main on destruction. There’s also a KillShare variable stored, but is not used due to the playback no longer accepting a quick on keypress option.

File Thread



FileThread is relatively simple as well compared to other threads. The file thread's job is to open the wav files in the local data folder, store the data into one of two buffers, and pass the data to the coordinator thread. This thread will continue to work until all files are done being read.

Coordinator Thread

The coordinator thread is the lynchpin of this entire operation. Without it, no thread will be able to know what to do. The coordinator thread is tasked with grabbing the buffer created by the FileThread thread. The coordinator thread stores the data to be used by the WaveFillCmd. Once WaveFillCmd has the data, the thread will drain the buffer. If both buffers are empty, the song is done and PlayBackThread is called to set its local bool to close out.

WaveFillCmd

WaveFillCmd first checks if we are killing the playback. If not, it gathers a 2k buffer from the coordinator thread to fill a buffer within WaveBuffThread. After passing the buffer, the command destroys itself. It's a smaller piece of the puzzle, but required for waveBuffThread to function.

WaveBuffThread

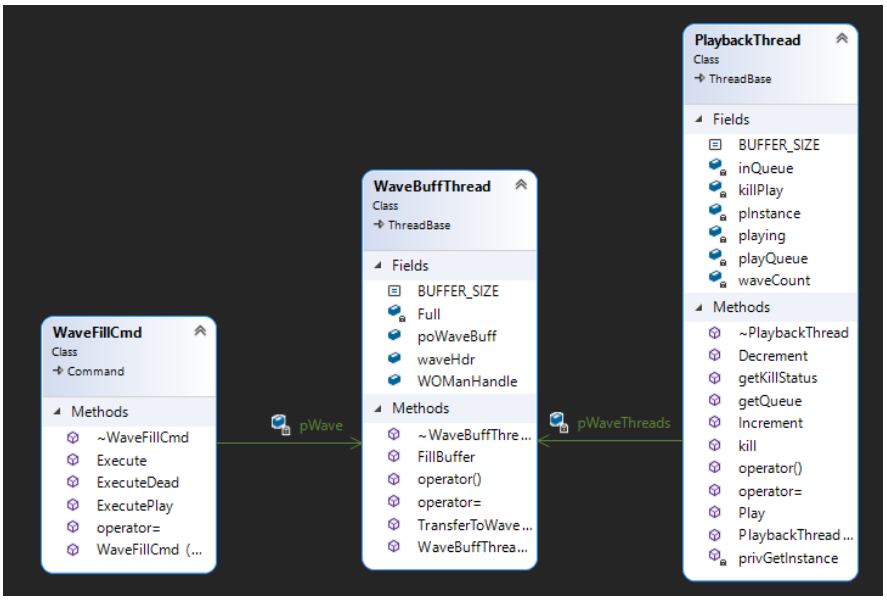
WaveBuffThread takes the buffer passed to it by the fill command and fills in WAVEHDR. This header along with a HWAVEOUT handle to be used later when playback calls the waveBuffThread to finally play.

```
void WaveFillCmd::Execute()
{
    if (PlaybackThread::getKillStatus())
    {
        PlaybackThread::Decrement();
        delete this;
    }
    else
    {
        unsigned char* pDest = CoordinatorThread::GetNext2KBuff();

        if (pDest != nullptr)
        {
            pWave->FillBuffer(pDest, 2 * 1024);
            PlaybackThread::Increment(pWave);
        }

        delete this;
    }
}
```

```
unsigned char* CoordinatorThread::GetNext2KBuff()
{
    CoordinatorThread& coordThread = CoordinatorThread::privGetInstance();

    if (coordThread.pDrainBuffer == nullptr) { return nullptr; }

    //find how much we have used out of buffer
    coordThread.bufferTag = coordThread.pDrainBuffer->GetRawBuff() + coordThread.pDrainBuffer->GetUsedSize();
    coordThread.pDrainBuffer->SetUsedSize(coordThread.pDrainBuffer->GetUsedSize() + 2048);

    if (coordThread.pDrainBuffer->GetUsedSize() >= coordThread.pDrainBuffer->GetCurrSize())
    {
        coordThread.pDrainBuffer->status = BufferStatus::EMTPY;
        Debug::out("  SWITCHED BUFFERS \n");

        if (coordThread.pDrainBuffer == coordThread.pBuff_A)
        {
            coordThread.pFillBuffer = coordThread.pBuff_A;
            coordThread.pDrainBuffer->SetUsedSize(0);
            coordThread.pDrainBuffer = coordThread.pBuff_B;
        }
        else if (coordThread.pDrainBuffer == coordThread.pBuff_B)
        {
            coordThread.pFillBuffer = coordThread.pBuff_B;
            coordThread.pDrainBuffer->SetUsedSize(0);
            coordThread.pDrainBuffer = coordThread.pBuff_A;
        }

        //if new drain is empty, the song is done
        if (coordThread.pDrainBuffer->status == BufferStatus::EMTPY)
        {
            //WRAP IT UP
            PlaybackThread::kill();
        }
    }
    return coordThread.bufferTag;
}
```

PlaybackThread

Finally, we reach the PlaybackThread. This thread gathers all the incoming wave buffer threads stored in a list and plays them in order. To do this, playback calls WaveBuffThread::TransferToWaveOut(). This method takes the above WAVEHDR and HWAVEOUT objects and converts them into an MMRESULT object that will finally play the sound.



Cleanup

Once all files have been read, a domino effect begins. The file thread stops sending info to the coordinator thread. The coordinator then passes its last buffer to the WaveFillBuffer. Once the last buffer is emptied from the coordinator thread and no others are able to be loaded, the PlaybackThread is told to close out. This sets the playback thread's local bool "killPlay" to true, telling the thread to stop looping indefinitely and to instead only loop while it is still playing

something. This calls the pWaveBuffThread to keep emptying itself as the player continues to play. This process continues until all wave buffers are emptied and there are no other sounds to play.

Eventually all the sounds will finish playing, causing the thread functors to cease processing. When the functors stop, their respective destructors are called. The destructors call join on themselves to reconnect to main. Once all threads are joined and completed, the main script will end, closing the app successfully with no leaks.

Conclusion

It may not look like much in my shortened explanation, but this project was insanely complicated. It took weeks of work to put together as many threads require constant communication with each other. While this was super interesting to work on, my lord do I hope to never work on something this insane alongside another class or project. This was WAY TOO INTENSE! I'm just glad I'm almost done. I still have so much to do. Fingers crossed I get it all in! Really want to keep this good grade rolling!