Independent Coursework 2

Erweiterung des Tracker-Plugins um das XM-Format

Max Linke - 558646

Inhaltsverzeichnis

Einleitung ... 3

WAV-Exporter ... 4

Auslesen von XM-Dateien ... 6

Refactoring und Tests ... 8

XM Playback ... 10

- Effekte ... 10
- Pan ... 12
- Envelopes, Autovibrato und Fadeout ... 13
- Ping Pong Loops ... 14
- Probleme mit Bresenham Sampling ... 16
- Tests ... 17

Fazit ... 18

Zeiterfassung ... 19

Abbildungsverzeichnis ... 21

Links ... 22

Einleitung

Letztes Semester implementierte ich für mein erstes IC ein Plugin, um ProTracker-Mod-Dateien in Unity abspielen zu können. Mod war das erste weitverbreitete Format für Trackermusik und damit ein guter Startpunkt, wenn auch durch die damalige Hardware in seinem Umfang begrenzt.

Das XM-Format, kurz für Extended Module, ist, wie der Name erraten lässt, eine Weiterentwicklung des Mod-Formats. XM erlaubt nativ mehr Kanäle, mehr Samples und fügt neue Effekte und Features hinzu. Die größte Neuerung sind die Instrumente, welche die Samples, die schlussendlich den Klang generieren, enthalten und einige Quality-Of-Life-Featues bieten wie zum Beispiel primitive ADSR-Kurven für Lautstärke und Pan, zuvor nur über Effekte steuerbar.

Da XM auf Mod basiert, konnten diverse Konzepte und Abläufe von der Arbeit des letzten ICs übernommen werden. Trotzdem ist XM sein eigenes Format und ohne Spezifikationen wäre es sehr schwierig gewesen, in der verfügbaren Zeit das gewünschte Ergebnis zu erzielen. Hauptsächlich verwendet wurde die inoffizielle Spezifikation des XM-Dateiformats von Vladimir Kameñar [1], doch auch die MilkyTracker-Dokumentation [2] erwies sich gelegentlich als hilfreich.

Dateien zum Testen und Vergleichen wurden selbst in OpenMPT [3] erstellt. Des Weiteren dienten Songs heruntergeladen von ModArchive [4] und der Soundtrack des 2019 veröffentlichten Spiels „Ion Fury“ [5] als Referenzmaterial.

WAV-Exporter

Bei der Entwicklung des Mod-Plugins war es wichtig, ständig den Audio-Output mit dem von etablierten Trackern zu vergleichen, um die Korrektheit der eigenen Implementierung zu verifizieren. Audacity wurde verwendet, um die Waveforms und Spektren zu vergleichen sowie beide Versionen synchron abzuspielen um eventuelle Diskrepanzen zu hören. OpenMPT, der Referenztracker, erlaubt es, Songs in verschiedene Audioformate zu exportieren, doch um den Unity-Output zu bekommen, musste der Audiostream per Stereomix aufgenommen werden, ein langwieriger Prozess.

Um dieses Problem zu beseitigen wurde eine Möglichkeit, Songs direkt aus Unity in WAV-Dateien zu schreiben, implementiert. Das Skript, das zur Laufzeit in Unity für den Audio-Output zuständig ist, ist die Klasse `TrackerMusicPlayer`. Um eigene Audiosignale zu senden, muss in einer von `MonoBehaviour` erbenden Klasse eine Methode `OnAudioFilterRead(float[] data, int channels)` implementiert werden. Wird das Skript dann einem GameObject mit entweder einer `AudioSource`- oder einer `AudioListener`-Komponente hinzugefügt, wird die Methode wiederholt vom Audio-Thread aus aufgerufen und das Float-Array kann mit eigenen Daten überschrieben werden.

Im `TrackerMusicPlayer` wird bei diesem Aufruf ein `AudioBuffer`-Objekt erstellt, welches dann an das aktuelle `PlaybackHandler`-Objekt übergeben und von diesem gefüllt wird. Ein `AudioBuffer` enthält neben dem Float-Array und der Anzahl der Kanäle zusätzlich noch die Samplerate, die verwendet werden soll, üblicherweise 48 oder 44.1 kHz. Diese Implementierung wurde bereits bei der Entwicklung des Mod-Plugins unternommen, um den Unity-Code vom Tracker-Code zu separieren und das Plugin einfach in andere Frameworks portierbar zu machen. `PlaybackHandler`-Objekte und `AudioBuffer`-Objekte lassen sich jederzeit erstellen und somit sind die zuvor genannten Bedingungen für Audio-Output zur Laufzeit in Unity irrelevant, wenn es nur darum geht, die Audiodaten zu generieren.

Ebenso für Mod implementiert war eine statische Methode, die einen temporären `PlaybackHandler` erstellt und mit diesem lediglich die einzelnen Ticks, Rows und Patterns

des aktuellen Songs durchläuft ohne Audio zu generieren, um im Vorhinein festzustellen, wie lang ein gegebener Song ist und ob dieser irgendwann in eine Wiederholschleife geht. Zum Export eines Songs wurde diese Methode leicht abgeändert, um eben doch Audio zu generieren und in einzelnen `AudioBuffer`-Objekten zu speichern. Erreicht der Song das Ende oder beginnt, sich zu wiederholen, werden die gesammelten Buffer zu einem großen Buffer kombiniert, welcher den gesamten Song enthält.

Inspiriert von der unityinternen Methode `Texture2D.EncodeToPNG` wurde eine Extension `public static byte[] EncodeWav (this AudioBuffer buffer)` geschrieben, welche für einen gegebenen AudioBuffer ein Byte-Array im RIFF WAVE Format generiert. Da keinerlei Kompression involviert ist, ist das Format sehr einfach und die Spezifikation auf Wikipedia [6] war absolut ausreichend für eine eigene Implementierung. Die einzige Hürde war es, zu erkennen, dass meine Klasse für Byte-Array-Utilities die falsche Endianess für WAV-Export hatte und per entsprechendem Refactoring das Problem zu beheben.

Zur Integration in Unity wurde eine weitere Klasse `FileExporterWizard`, erbend von `EditorWindow`, geschrieben, welche ein GUI im Unity Editor bereitstellt zum Export, sowie das Schreiben der Daten auf die Festplatte übernimmt. Fortschrittsbalken zeigen dabei an, wie weit ein gegebener Schritt des Vorgangs ist, wobei der Export üblicherweise lediglich eine Handvoll Sekunden dauert.

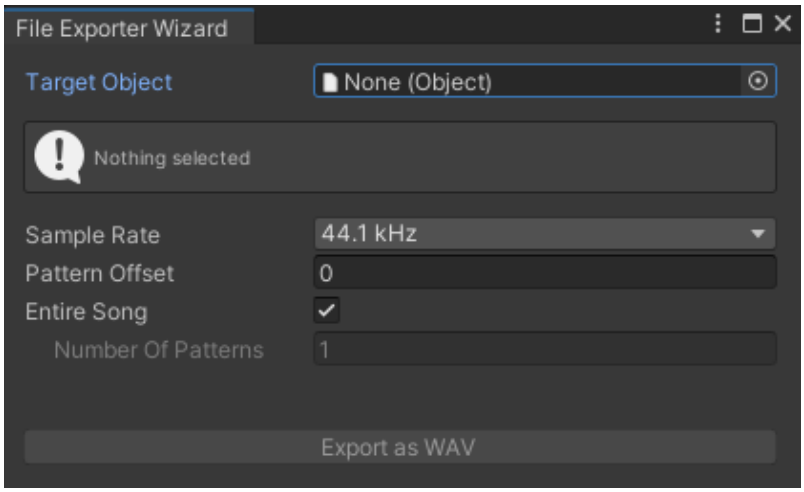

Abbildung 1: Das FileExporterWizard-Fenster ohne ausgewähltes Objekt zum Export

Auslesen von XM-Dateien

XM-Dateien, so wie Mod-Dateien müssen Byte für Byte mit genauem Wissen, welche Daten wo stehen, ausgelesen werden. Dass für den WAV-Export die Byte-Array-Utils angepasst wurden machte sich hier bezahlt, da XM im Gegensatz zu Mod Bytes in Little Endian Reihenfolge enthält.

Die einzelnen Schritte wurden, wie bei Mod, wieder über statische Methoden implementiert. Um den Auslesecode auf die dateispezifischen Aspekte zu begrenzen und Code Duplication zu vermeiden wurde eine abstrakte Base-Class `FileReader` erstellt, von der die Mod- und XM-Leseklassen erben.

```
namespace TrackerMusic {

    public abstract class FileReader {

        public abstract File ReadFile (string path);

    }

    public abstract class FileReader<TFile> : FileReader where TFile : File {

        public override File ReadFile (string path) {
            var bytes = System.IO.File.ReadAllBytes(path);
            var output = ReadBytes(bytes, out int offset);
            if(offset < bytes.Length){
                var warningsList = (IList<string>)(output.GetWarnings());
                warningsList.Add($"After reading file \"{path}\", the offset was at
{offset} (0x{offset:X}), while the file has length {bytes.Length}. So {bytes.Length -
offset} bytes were ignored!");
            }
            return output;
        }

        protected abstract TFile ReadBytes (byte[] bytes, out int offset);

    }

}
```

Eine Neuerung von XM ist, dass die einzelnen Zellen der Patterns ein einfaches Kompressionsschema enthalten, was angesichts der höheren Anzahl Kanäle durchaus einen merkbaren Effekt auf die Dateigröße hat. Neben den neuen Elementen wie den Instrumenten und ihren Hüllkurven kamen zu bereits existierenden Teilen, wie Samples,

weitere Metadaten hinzu, wie ein neuer Loop-Type, sowie drei Optionen zur Codierung der tatsächlichen Audiodaten der Samples, im Gegensatz zur einfachen 8-Bit Signed Byte Codierung von Mod.

Da die FileReference-ScriptableObjects, erstellt im ersten IC, bereits nur mit der abstrakten File-Klasse arbeiteten, lies sich über die Anzeige der generischen Metadaten eines Songs schnell testen, ob Dateien zum einen korrekt und zum anderen vollständig ausgelesen werden. Bei einigen Dateien beendete der Reader seine Arbeit vor Ende des Byte-Arrays, woraufhin sich nach etwas Nachforschung herausstellte, dass OpenMPT zusätzliche Metadaten am Dateiende anfügen kann [7], wie zum Beispiel Midi-Daten oder eine kurze Textnachricht.

Abbildung 2: Waveform von Sample 10 von "an-path.xm" visualisiert in Unity

Zur vorläufigen Verifikation der Samples und Instrument-Envelopes wurden wie bei Mod mithilfe einer Testklasse Rastergrafiken gezeichnet und per UI-Element zur Laufzeit in Unity dargestellt, um auch ohne implementierte Playback-Routine eventuelle Probleme zu erkennen.

Refactoring und Tests

Da XM eine Erweiterung von Mod ist, lag es nahe, die Gemeinsamkeiten über Vererbung zu implementieren um Code Duplication zu vermeiden. Angefangen wurde mit dem PlaybackHandler von Mod. Nach außen hin sind nur Interfaces wie `IFilePlaybackHandler` sichtbar, sodass es einfach war, die abstrakte Klasse `PlaybackHandlerBase` und die davon erbende generische Klasse, von der die tatsächlichen PlaybackHandler erben, einzubauen. Von einer handvoll Properties für die zwischen Formaten variierenden Konstanten abgesehen lies sich der gesamte Code des für Mod geschriebenen PlaybackHandlers sehr schnell in die abstrakten Klassen übertragen.

Als nächstes sollten die Channels ähnlich refactored werden, jedoch war die „schöne“ generische Lösung extrem unschön, da sich PlaybackHandler und Channels gegenseitig referenzieren und somit alle generischen Typenparameter doppelt aufgeführt werden müssten. Ein Beispiel sollte das Problem aufzeigen:

```
class Foo<T, U>
    where T : Bar<U, T>
    where U : Foo<T, U>
{ … }

class Bar<T, U>
    where T : Foo<U, T>
    where U : Bar<T, U>
{ … }

class ActualFoo : Foo<ActualBar, ActualFoo> { … }

class ActualBar : Bar<ActualFoo, ActualBar> { … }
```

Zur Referenz, der generische `PlaybackHandlerBase` hat die folgende Signatur:

```
public abstract class PlaybackHandlerBase<TFile, TPattern, TRow, TCell, TChannel>
                    : PlaybackHandlerBase, IFilePlaybackHandler
    where TFile    : File<TPattern, TRow, TCell>
    where TPattern : class, IPattern<TRow, TCell>
    where TRow     : class, IRow<TCell>
    where TCell    : ICell
    where TChannel : ChannelBase, new ()
{ … }
```

Es wäre absolut möglich (und war eine Zeit lang so implementiert), `ChannelBase` ebenfalls vollkommen generisch mit den gleichen Parametern wie der `PlaybackHandlerBase` zu machen, jedoch war die Anzahl von Typenparametern enorm, da die Channels weitere generische Parameter benutzen (z.B. Samples) und all dies in den beiden Basisklassen und davon erbenden Klassen angegeben werden musste für relativ wenig Nutzen am Ende.

Da die Channels der komplexeste Teil sind, war es wichtig, sicherzustellen, dass das Refactoring keine neuen Bugs produziert. Dafür wurden die alten Skripte in die neuen Klassen `LegacyModFilePlaybackHandler` und `LegacyModChannel` kopiert und mittels des TestFramework-Packages Tests erstellt, die den Audio-Output für diverse Mod-Songs vom neuen und alten PlaybackHandler vergleichen. Ein paar Tests schlugen tatsächlich fehl durch ein Problem, das schnell gefunden und behoben wurde.

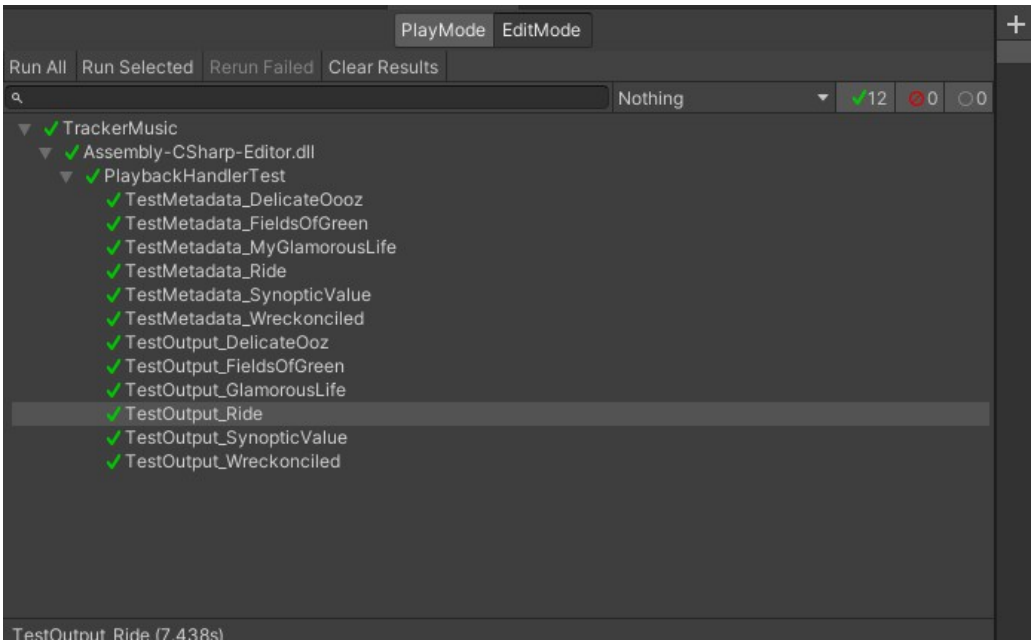

Abbildung 3: Test-Suite zum Vergleich des Legacy- und generischen Mod-PlaybackHandlers

Die Tests waren sehr nützlich, da das Refactoring der Channels wesentlich komplizierter war als das des PlaybackHandlers. Um möglichst viel Überlappung zwischen Mod und XM zu haben, mussten einige Teile des Mod-Channels umgeschrieben werden auf Arten und Weisen, bei denen es nicht direkt offensichtlich war, dass das Ergebnis das Gleiche sein würde.

XM Playback

Da der XM-PlaybackHandler so wie der neue Mod-PlaybackHandler dank Vererbung lediglich ein paar Konstanten bereitstellen muss, war der Schwerpunkt bei der Implementierung von XM die Implementierung der Kanäle, die für den Audio-Output zuständig sind.

Zum Anfang mussten die Magic Numbers gefunden werden, mit denen Samples die korrekte Tonhöhe erreichen. Bei Mod enthalten die Zellen der Patterns direkt die gewünschte Periode, den Index des abzuspielenden Samples sowie einen möglichen Effekt. Bei XM ist die Periode durch lediglich den Index einer Note ersetzt und es kommt ein weiterer Volume-Column-Effekt hinzu, sowie mehr mögliche Effekte im regulären Effekt-Feld. Glücklicherweise beschreiben einige Online-Referenzen [1] [2], wie sich aus der aktuellen Note ein Periodenwert berechnen lässt. Um damit in der korrekten Tonhöhe das gewünschte Sample abzuspielen, musste lediglich bestimmt werden, mit welcher Samplerate XM nativ umgeht, da sich nur so die Bresenham-Parameter berechnen lassen, mit denen beim Audio-Output die Audiodaten der Samples in den Output-Buffer geschrieben werden. Hierfür wurde in OpenMPT ein einsekündiges 8 kHz Sample geladen, Loop aktiviert, das Sample mit der neutralen Note C-5 ohne Finetune abgespielt und das Ergebnis in Audacity analysiert. Bei einer Samplerate von 44.1 kHz wiederholte sich das Sample nach 42.187 ausgegebenen Werten. Teilt man die 8 kHz des Samples durch das Verhältnis der Wiederholrate von 42.187/44.100 bekommt man die Referenzrate von 8363 Samples pro Sekunde. Leider war dies sowohl das Ende der klar dokumentierten Abläufe als auch der Anfang von viel Analyse der Outputs von OpenMPT in Audacity, um die Details der Implementierung von XM festzustellen.

Effekte

XM verwendet fast alle Effekte von Mod und dank des Refactorings der Effekte in die geteilte Basisklasse `ModChannelBase` musste für XM lediglich deren Applikation beim Output implementiert werden. Eine weitere Neuerung von XM-Dateien ist die Flag, die besagt, ob

Tonhöhenveränderungen („Slides“) linear oder „Amiga“-mäßig vorzunehmen sind. Letzteres bezieht sich auf die Implementierung bei Mod, wo sämtliche Slide-Offsets direkt auf der Periode vorgenommen wurden, was bei höheren Tönen (niedrigere Periode) einen größeren Einfluss hat als bei tieferen Tönen. Lineare Slides hingegen werden vor der Berechnung der Periode vorgenommen und mit der Note, aus der sich die Periode ergibt, verrechnet.

Da das Slide-Offset ein Integer ist, muss, bevor es mit dem gewünschten Wert kombiniert wird, ein Faktor involviert sein, vor allem bei linearen Slides, wo die Noten ebenfalls Integer sind. Für den Vibrato-Effekt waren diese Werte schnell gefunden. Bei linearen Slides wird das Offset durch 16 geteilt und von der Note subtrahiert, bei Amiga-Slides wird das Vibrato-Offset mit 16 multipliziert und zur Periode addiert. Die Note-Slides hingegen waren etwas komplizierter, bei Amiga-Slides beträgt der Faktor ungefähr 18.5, ein Wert ermittelt durch schrittweise Annäherung. Auch die Berechnung des Deltas für Portamento, wo zu einer gewünschten Note hingeslidet wird, ist dadurch komplizierter.

Interessanterweise funktioniert Arpeggio bei XM deutlich anders als bei Mod, was zu etwas Verwirrung führte. Arpeggio ist der Effekt, bei der die Note jeden Tick um ein eingestelltes Offset verändert wird. Effekt 047 z.B. beschreibt Arpeggio mit den Offsets 4 und 7 und emuliert somit einen Dur-Akkord in Grundstellung, da der Grundton, die große Terz (4 Halbtöne) und die reine Quinte (7 Halbtöne) nacheinander gespielt werden. Bei Mod ist die Reihenfolge bei Effekt 0xy genau diese – 0xy0xy0x..., bis der Tick zu Ende ist. XM hingegen fängt zwar mit dem Grundton an, endet aber stets mit Offset x und geht die Offsets vom Ende aus durch. Sprich bei 3 Ticks pro Row (TPR) werden die Offsets 0yx sein, bei 4 TPR 00yx, bei 5 TPR 0x0yx und so weiter.

Die neuen XM-Effekte sind zum Teil Erweiterungen der in Mod vorhandenen, zum Beispiel der Panning-Slide Effekt Pxx (Mod kann nur das Panning setzen) und extrafeine Volume Slides X1x und X2x. Es gibt allerdings auch neue, wie den Set-Global-Volume Effekt Gxx und Global-Volume-Slide Effekt Hxx, welche, wie der Name vermuten lässt, die Lautstärke des gesamten Songs langfristig verändern und nicht nur kurzfristig die des eigenen Kanals.

Die eigene Implementierung des Tremor Effekts Txy weicht von der in OpenMPT ab, wobei diese auch von der Implementierung in MilkyTracker abweicht, welches sich als ein akkurater FastTracker2-Klon beschreibt. Die Beschreibung des Effekts ist allerdings so vage und die Abweichung nur in einem Spezialfall vorhanden, sodass dies kein wichtiges Problem darstellt.

Pan

Das Mixen des Audiosignals auf die zwei Kanäle links und rechts war bei Mod sehr einfach. Interpretiert man den Pan als normierte Zahl von 0 bis 1, hört man bei 0 nur den linken Kanal bei voller Lautstärke, bei 1 nur den rechten. Bei 0.5 sind beide Kanäle gleich stark mit halber Lautstärke zu hören, da das Mixing linear verläuft.

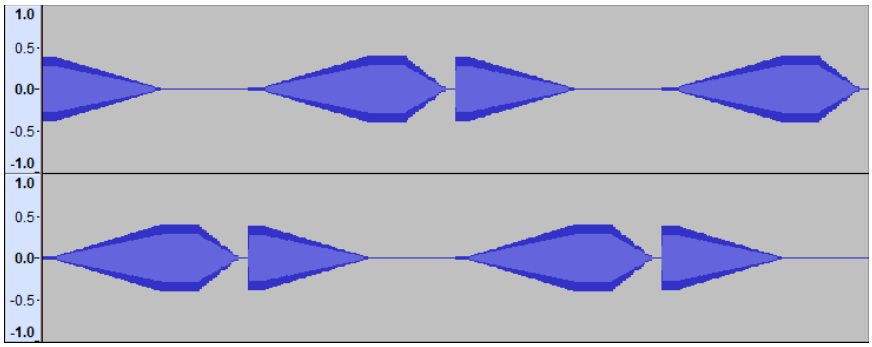

Abbildung 4: Lineares Stereo-Mixing wie in Mod (Oben der linke Kanal unten der rechte)

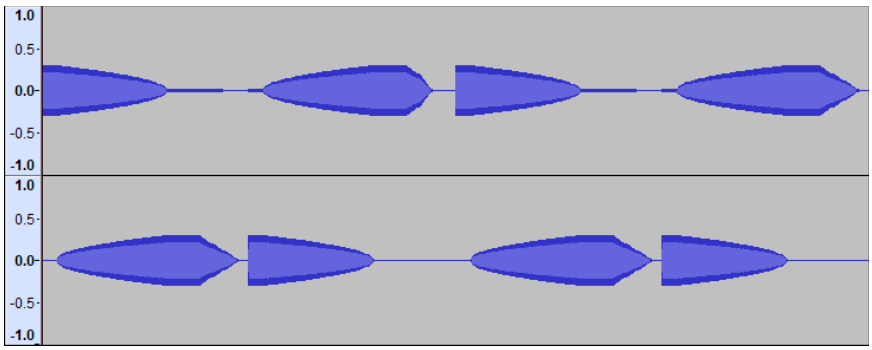

Abbildung 5: Stereo-Mixing von XM-Dateien in OpenMPT (Oben der linke Kanal unten der rechte)

Beim Testen des Panning Slide Effekts fiel auf, dass die eigene Waveform von der in OpenMPT abwich. Die korrekte Implementierung war aber sehr einfach. Statt direkt „1-pan“ für Links und „pan“ für Rechts als Multiplikator zu nehmen, wird jeweils die Quadratwurzel des linearen Wertes verwendet.

Envelopes, Autovibrato und Fadeout

Instrumente in XM machen es nicht nur möglich, je nach Note verschiedene Samples zu spielen, sie haben auch wie zuvor erwähnt die Möglichkeit, einfache Kurven für Lautstärke und Pan zu verwenden, haben einen integrierten Vibrato-Oszillator, und lassen sich mit der neuen Key-Off-Note „releasen“ und je nach Einstellung langsam ausfaden.

Die Kurven haben mindestens 2 und maximal 12 Punkte und interpolieren linear zwischen einzelnen Werten. Optional lassen sich einzelne Punkte spezifisch als Loop-Start und -Ende designieren oder als Sustain-Punkt, der gehalten wird, solange die Note nicht released ist. Was passiert, wenn sich diese Punkte überlappen, ist natürlich nicht dokumentiert und musste selbst getestet werden. Um sie an der korrekten Stelle zu samplen, wird im Channel ein Zähler jeden Tick inkrementiert, gehalten (Sustain) oder zurückgesetzt (Loop) und beim Output die Kurve an der entsprechende Stelle ausgelesen. Die Interpolation erfolgt ohne Gleitkommaarithmetik.

```
public int Sample (int position) {
    for(int i=1; i<points.Count; i++){
        if(position <= points[i].x){
            var left = points[i-1];
            var right = points[i];
            position -= left.x;
            var maxPos = right.x - left.x;
            return ((left.y * (maxPos - position)) + (right.y * position)) / maxPos;
        }
    }
    return -1;
}
```

Autovibrato macht es möglich, Vibrato (wo die Tonhöhe hin und her schwankt) für ein Instrument generell zu definieren, ohne den Vibrato-Effekt benutzen zu müssen. Die Waveforms ähneln denen der Oszillatoren, die für den Vibrato- und Tremolo-Effekt verwendet werden, haben aber eine längere Periode, zwei verschiedene Ramp-Waveforms und keinen Zufallsmodus mehr. Der Oszillator hat zwei interne Zähler. Der Erste hat eine variable Schrittweite, wiederholt sich und bestimmt, an welcher Stelle die Waveform ausgelesen wird. Der Zweite bestimmt die „Sweep Position“, mit der das Vibrato langsam eingeblendet werden kann. Das Resetten der Positionen und der Aufruf, die Zähler zu

inkrementieren führte mehrfach während der Entwicklung zu Problemen als die Reihenfolge inkorrekt war, da bei der Berechnung der Tiefe des Vibratos Division erfolgt.

```
var depth = (m_sweepPos < m_settings.sweep)
          ? (m_sweepPos * m_settings.depth) / m_settings.sweep
          : m_settings.depth;
```

Hat das Instrument einen Sweep von 0, ist dies nur ein Problem bei Division durch 0, wenn die Sweep Position negativ ist. Sie wird resettet auf -1, um nach dem ersten Inkrement bei 0 zu stehen, was allerdings nur der Fall ist, wenn der Aufruf auch erfolgt. Der Delay-Note-Effekt EDx führte kurzzeitig zu genau diesem Problem, dass der Oszillator resettet wurde, die Zähler nicht inkrementiert und beim Sampling dann Division durch 0 zu einer Exception führte.

Letztlich ist Volume Fadeout die Möglichkeit, mittels der Key-Off-Note oder des Key-Off-Effekts Kxx ein Sample faden zu lassen, ohne den Volume-Slide-Effekt. Key-Off wirkt sich zusätzlich auf die Envelopes aus, da somit der Sustain-Punkt passiert werden kann und verhindert das Inkrementieren der Sweep-Pos im Autovibrato-Oszillator.

Ping Pong Loops

Mod-Samples hatten lediglich zwei Optionen, was mit dem fortlaufenden Index, an dem sie ausgelesen werden beim Output, passiert. Zum einen konnte nichts vorgenommen werden und die Samples spielen nur bis das Ende erreicht ist und stoppen. Zum anderen konnten Start und Ende für einen Loop gesetzt werden, damit sich der Index innerhalb der spezifizierten Region wiederholt.

XM fügt eine weitere Option hinzu: Ping Pong Loops. Es werden wieder zwei Punkte gesetzt, aber statt vom Ende zum Anfang zurückzuspringen, wird das Sample daraufhin rückwärts abgespielt, dann wieder vorwärts und so weiter, stets innerhalb der spezifizierten Region. Statt einigen if-else Statements bezüglich des Loop-Types oder eines Switches wurde die abstrakte Klasse `IndexValidator` erstellt, von der die drei Klassen `SimpleIndexValidator`, `ForwardLoopingIndexValidator` sowie `PingPongLoopingIndexValidator` erben, die jeweils die Funktion `ValidateIndex` implementieren. Sie agieren als Singletons

und werden vor dem Füllen des Outputbuffers entsprechend des Loop-Types des aktuellen Samples ausgewählt. Der nicht-loopende Index Valiator ist sehr einfach gehalten:

```
public override bool ValidateIndex (
    ISample sample,
    ref int index,
    ref int advanceDirection
){
    return index < sample.length;
}
```

Der einfache Loop ist nicht viel komplexer:

```
public override bool ValidateIndex (
    ISample sample,
    ref int index,
    ref int advanceDirection
){
    if(index >= sample.loopStart){
       index = sample.loopStart + ((index - sample.loopStart) % sample.loopLength);
    }
    return true;
}
```

Der Ping-Pong-Loop hingegen schon:

```
public override bool ValidateIndex (
    ISample sample,
    ref int index,
    ref int advanceDirection
){
    if(advanceDirection > 0){
        var loopEnd = sample.loopStart + sample.loopLength;
        if(index >= loopEnd){
            var loopCount = (index - sample.loopStart) / sample.loopLength;
            index -= 2 * (loopCount / 2) * sample.loopLength;
            if(index >= loopEnd){
                advanceDirection = -1;
                index = (2 * (loopEnd - 1)) - index;
            }
        }
    }else{
        if(index < sample.loopStart){
            var loopCount = (sample.loopStart - index) / sample.loopLength;
            index += 2 * (loopCount / 2) * sample.loopLength;
            if(index < sample.loopStart){
                advanceDirection = 1;
                index = (2 * sample.loopStart) - index;
            }
        }
    }
    return true;
}
```

So wie beim einfachen Forward-Loop der Index beim Erreichen des Endes nicht einfach zum Start zurückgesetzt wird, sondern per Modulo die Menge, mit der das Ende

überschossen wurde, weitergetragen wird, ist auch beim Ping-Pong-Loop die Endbehandlung nicht trivial. Bei sehr kurzen Samples, die sehr schnell abgespielt werden, kann es durchaus passieren, dass der „gespiegelte“ Index vor dem Loopbeginn landet, sogar negativ wird. In diesem Falle müsste noch eine Spiegelung vorgenommen und wieder getestet werden, ob der Wert nun legal ist. Durch etwas Rechnerei lässt sich dies allerdings, egal wie viele Spiegelungen passieren, in einem Schritt lösen.

Probleme mit Bresenham Sampling

Um die Audiodaten der Samples mit der richtigen Geschwindigkeit in den Outputbuffer zu kopieren wurde bereits beim Mod-Player eine Variation vom Bresenham-Algorithmus verwendet, da dieser mit reiner Integer-Mathematik eine Gerade mit nicht ganzzahliger Steigung zeichnen kann. Wenn man sich zwei Achsen denkt, bei denen eine der Outputbuffer mit seiner Samplerate ist und die andere das Sample mit seiner eigenen Samplerate, kann man mit einer Gerade in diesem System beschreiben, mit welcher Rate Audiodaten vom Sample in den Outputbuffer geschrieben werden. Auf der Output-Achse legt die Gerade das Produkt der Sampleperiode mit der Samplerate des Buffers zurück, auf der anderen Seite das Produkt der Referenzperiode und Referenzrate. Die Referenzrate ist hierbei die Samplerate, mit der der Tracker ein Sample der Referenzperiode ausgibt und wurde am Anfang dieses Kapitels erwähnt.

Dies sind alles Integer und bei Mod stellte dies kein Problem dar. Bei XM jedoch sind die Zahlen größer und bei einer Outputsamplerate von 48 kHz darf die Periode nicht 44.739 überschreiten, ansonten kommt es zu Integer Overflow. Dieser Wert wird erreicht, wenn die tiefste erlaubte Note mit Index 119 (höherer Index = tieferer Ton) gespielt wird und kann durch diverse Effekte überschritten werden. Aus diesem Grund wurde die Bresenham-Arithmetik auf Long umgestellt.

Es gab allerdings noch ein weiteres Problem. Bei manchen Samples bei manchen Songs war der erste Deltawert, der zum Samplingindex addiert wurde negativ, was darauffolgend zu einer Exception führte, als der entsprechende Audiowert des Samples ausgelesen werden sollte. Der Deltawert berechnet sich aus einer Kombination der

Bresenham-Steigung und des aktuellen Bresenham-Errors. Bei sehr niedrigen Perioden (also hohen Tönen) kommt es dazu, dass der initiale Error eben einen solchen negativen Deltawert produziert, weil der Error zu groß (nicht negativ genug) ist. Die Ursache ist unbekannt, vor allem weil die Error-Berechnung nicht vom Original verändert wurde. Ein einfacher Test, ob der initiale Deltawert problematisch wäre und ein vorläufiges Advancen des Errors umgeht allerdings das Problem und es kommt weder zu Exceptions noch zu inkorrektem Sampling.

Tests

Während des Refactorings und der Implementation der XM-Effekte wurden speziell angefertigte Testdateien verwendet. Dank des WAV-Exporters gingen Vergleiche mit dem Output von OpenMPT wesentlich schneller voran als zuvor und die häufigen doppelten Tests bezüglich linearen Slides und Amiga Slides waren erträglich.

Mit allen Effekten implementiert, wurden diverse Songs, sowohl aus dem Internet heruntergeladen, als vom Ion Fury Soundtrack getestet. Neben dem zuvor genannten Bresenham-Problem kam es auch vor, dass manche Songs inkorrekt abgespielt wurden, was bei den vertrauten Songs schnell auffiel. Der Grund war die inkorrekte Methodik bezüglich von Noten und Instrumenten in Zellen. Was genau passiert, wenn eine Zelle nur eine Note oder nur ein Instrument enthält ist nicht dokumentiert. Effekte wie Portamento, der Delay-Note-Effekt und die Key-Off-Note beeinflussen ebenfalls das Ergebnis sowie die Envelopes, Autovibrato und Fadeout. Umfangreiche Testdateien zu erstellen ist dadurch schwierig, weil viele Variationen beachtet werden müssen. Die Songs hingegen hatten konkrete Beispiele bezüglich der inkorrekten Implementation und dies ermöglichte es, die tatsächlich recht einfache Routine, mit der Noten und Instrumente verarbeitet werden, zu finden und zu implementieren.

Fazit

Obwohl XM „nur“ eine Erweiterung von Mod ist, betrug die investierte Arbeitszeit in die Erweiterung des Plugins trotzdem fast die gleiche Zeit wie die ursprüngliche Implementierung von Mod. Externalitäten wie der Wav-Exporter waren schnell erledigt, der TrackerMusicPlayer, die FileReferences und mehr wurden von der Mod-Version wenig bis gar nicht verändert, da sie bereits formatunabhängig entwickelt wurden.

Das Refactoring hin zu generischen Basisklassen sowie die Verifikation, dass das Refactoring keine neuen Probleme mit sich brachte, war sehr lehrreich, da Generics unglaublich mächtig sind, aber auch gleichzeitig viel Planung erfordern, da sonst Dinge schiefgehen können. Dies ist der Grund, warum es die abstrakte Klasse `PlaybackHandlerBase` gibt, die nichts implementiert, und dann die davon erbende abstrakte generische Klasse `PlaybackHandlerBase<TFile, TPattern, TRow, TCell, Tchannel>`. Es ist nicht möglich, eine Referenz zum generischen Objekt zu haben, ohne alle Typ-Parameter zu spezifizieren, aber sehr wohl eine Referenz zur nicht generischen Version und diese dann auf einen spezifischen Typen, z.B. `XmFilePlaybackHandler`, zu casten.

Der Mangel an Dokumentation, vor allem im Vergleich zu Mod war definitiv eine Hürde, die lediglich langsam und mit vielen einzelnen Tests überwunden werden konnte. Bei Mod war klar, dass die Slide Offsets und Vibrato-Offsets direkt zur Periode addiert werden, bei XM mussten erst Multiplikatoren bestimmt werden. Doch auch Mod hatte unklar definierte Teile, wie Glissando oder Effekt EFx. Am Ende war es jedoch auch hier immens belohnend, nach Fertigstellung endlich echte XM-Songs abspielen und genießen zu können.

Zeiterfassung

Datum	Beschreibung	Zeit (Std.)
28.11.22	Anfang mit WAV-Exporter	2
29.11.22	Fertigstellung WAV-Exporter, Refactoring ByteArrayUtils wegen Endianess	2
12.04.22	Implementierung File-Export	5
06.12.22	Anfang XM, Wechsel zu Refactoring, Probleme mit zu vielen generischen Typparametern	5
08.12.22	Refactoring, Anfang XmFileReader	4
09.12.22	Fortsetzung XmFileReader, Auslesen des Headers	1
10.12.22	Fortsetzung XmFileReader, Auslesen von Patterns und Instrumenten	5
11.12.22	Fortsetzung XmFileReader, Auslesen von Instrumenten und Samples	6
25.12.22	Fortsetzung XmFileReader, Auslesen und Visualisieren von Samples	3
26.12.22	Refactoring	4
28.11.22	Refactoring, Unit Tests für Mod	4
01.01.23	Anfang XmFilePlaybackHandler, Refactoring in PlaybackHandlerBase	3
18.02.23	Fortsetzung XmFileReader, Auslesen von OpenMPT-Metadaten, Anfang XmChannel	5
19.02.23	Fortsetzung XmChannel, Refactoring von Effekten in ChannelBase	3
20.02.23	Xm Magic Numbers berechnet und getestet	1
21.02.23	Fortsetzung XmChannel, Berechnung der Periode, Envelope Samping	5
23.02.23	Envelope Samping Integer-Interpolation verifiziert per Visualisierung, Tests bzgl. überlappender Loop- und Sustain-Punkte	1
24.02.23	Tests bzgl. Instrument Fadeout und Envelopes, Vergleiche OpenMPT und MilkyTracker	1
25.02.23	Tests in OpenMPT, wann und wie Envelopes resettet werden, wie Sample Retriggering funktioniert und Volume Fadeout stattfindet	3
26.02.23	Implementierung von IndexValidator, Applikation von Instrument Envelopes	4
28.02.23	Implementierung Autovibrato	4
01.03.23	Fertigstellung Autovibrato, Mono-Exportoption für File Exporter,	1
05.03.23	Refactoring von Oscillator Effects, mehr Mod-Tests	4
06.03.23	Erstellung ModChannelBase für Code, der nicht in ChannelBase sein sollte, aber vom Mod Channel und vom Xm Channel geteilt wird	9
07.03.23	Applikation von weiteren Effekten im Xm Channel, bessere Implementierung von Effekt Memory	9
08.03.23	Glissando Effekt implementiert	1

09.03.23	Anfang Volume-Column Effects	4
11.03.23	Arpeggio und Tremor Effekt implementiert	5
12.03.23	Letzte Effekte implementiert, Probleme mit echten Songs	6
14.03.23	Unit Tests für XM erstellt, Bresenham-Problem gefixt, anhand von Songs Probleme mit einigen Effekten und vor allem Note- und Instrument-Handling Code gefunden und gefixt	11
15.03.23	OnGuiPlayer angepasst für Demo	0.5
21.03.23	Amiga-Arpeggio implementiert (vorher nur für lineare Slides), mehr Songs getestet, ein weiteres Problem gefunden und behoben	1
	Gesamtdauer	124.5

Abbildungsverzeichnis

Abbildung 1: Das FileExporterWizard-Fenster ohne ausgewähltes Objekt zum Export.....5
Abbildung 2: Waveform von Sample 10 von "an-path.xm" visualisiert in Unity.....7
Abbildung 3: Test-Suite zum Vergleich des Legacy- und generischen Mod-PlaybackHandlers.....9
Abbildung 4: Lineares Stereo-Mixing wie in Mod (Oben der linke Kanal unten der rechte).....12
Abbildung 5: Stereo-Mixing von XM-Dateien in OpenMPT (Oben der linke Kanal unten der rechte).....12

Links

[1] The Unofficial XM File Format Specification : Kameñar, Vladimir
https://archive.org/details/xm-file-format

[2] MilkyTracker/xm-form.txt at master · milkytracker/MilkyTracker
https://github.com/milkytracker/MilkyTracker/blob/master/resources/reference/xm-form.txt

[3] OpenMPT - Discover the music inside... | OpenMPT - Open ModPlug Tracker
https://openmpt.org/

[4] The Mod Archive v4.0b - A distinctive collection of modules
https://modarchive.org/

[5] Ion Fury
https://www.ionfury.com/

[6] RIFF WAVE – Wikipedia
https://de.wikipedia.org/wiki/RIFF_WAVE

[7] Development: OpenMPT Format Extensions - OpenMPT Wiki
https://wiki.openmpt.org/Development:_OpenMPT_Format_Extensions