Independent Coursework 1

Erstellung eines Plugins zum Abspielen von ProTracker-Module-Dateien in der Unity Engine

Max Linke – 558646

Inhaltsverzeichnis

Motivation

Historische Entwicklung

Audio, sowohl in Form von Soundeffekten als auch Musik, ist ein Kernaspekt von Videospielen. Anfangs bestand die Klangkulisse vornehmlich aus computergenerierten Tönen mit Schwingungsformen, die einfach zu generieren waren, zum Beispiel Rechteck- und Sägezahnschwingungen. Diese haben im Vergleich zu Sinusschwingungen eine Vielzahl von Obertönen, die dieser Art von Musik, heute "Chiptunes" genannt, ihren charakteristischen Klang verleihen. Generiert wurden die Klänge zur Laufzeit, indem ähnlich zu einer Spieldose die Hardwareoszillatoren im Takt Befehle bekamen bezüglich Frequenz, Lautstärke und mehr. Mit dem Wechsel in die 16-Bit-Ära kamen auch vermehrt komplexere Synthesizer, zum Beispiel mit Frequenzmodulation, sowie Sampler zum Einsatz. Und während mit Frequenzmodulation und genügend Oszillatoren die meisten akustischen Instrumente sich zumindest erkennbar nachbilden lassen, können Sampler echte aufgenommene Klänge wieder abspielen, zum Beispiel für noch realistischere Instrumente, oder um Sounds zu produzieren, die stärkere Synthesizer erfordern, als auf der abspielenden Hardware vorhanden.

ProTracker, veröffentlicht im Jahr 1990, ist ein Trackerprogramm entwickelt für die Amiga-Computer, ausgerüstet mit dem Paula-Soundchip, welcher in der Lage war, gleichzeitig vier 8-Bit Samples abzuspielen. Der Code basierte auf vorherigen Trackern, welche allerdings nie den Erfolg von ProTracker erreichten. Das ProTracker-Module-Format, kurz Mod, etablierte Konventionen, welche auch in späteren Formaten noch verwendet werden würden.

Eigene Erfahrung

Mein erster Kontakt mit Trackern war das freie Programm FamiTracker [1], mit dem Benutzer NES-Kompatible Chiptunes erstellen können. Ohne virtuelle Erweiterungschips zu aktivieren gibt es nur vier Oszillatorenkanäle und einen Sample-Kanal. Es können „Instrumente“ erstellt werden, welche mittels einfacher Hüllkurven für Lautstärke, Tonhöhe und weiterem individualisierbar sind. Das Interface, die Effekte und auch die FTM-Dateien (FamiTracker Module) sind stark an ProTracker-Mod angelehnt. Als ich später das

Videospiel „Deus Ex“ spielte und beim Erforschen der Dateien den Musikordner fand, lernte ich, dass mit Trackern mehr als nur Chiptunes möglich sind.

Eine praktische Fertigkeit, die ich durch die Benutzung von Trackern erlernte, ist der Umgang mit kleinen Hexadezimalzahlen. Die Zeilen sind fortlaufend nummeriert, wobei 00h, 10h, 20h und 30h klar Grundschläge im Takt beschreiben, wenn man in 4/4 komponiert. Des weiteren sind Effektparameter hexadezimal anzugeben. Der Effekt 07C zum Beispiel beschreibt ein Power-Chord Arpeggio, die führende 0 steht für einen Arpeggio-Effekt, die folgende 7 besagt, dass die zweite Note um 7 Halbtöne nach oben transponiert werden soll und das abschließende C, dass die dritte Note 12 Halbtöne – eine Oktave – transponiert wird.

Gegenwärtige Relevanz und Zielsetzung

Der Vorteil von Trackermodulen gegenüber den heutzutage weitverbreiteten Audiocodecs ist hauptsächlich die geringe Dateigröße, was in den 90ern noch relevant war, heute allerdings weniger. Für Spiele hingegen erlaubt die dynamische Erzeugung der Klänge programmatischen Eingriff zur Laufzeit, beispielsweise um Audiospuren ein- und auszublenden ohne Synchronisationsprobleme, oder die Geschwindigkeit anzupassen, ohne die Tonhöhe zu beeinflussen. Wie zuvor bezüglich „Deus Ex“ erwähnt, benutzte die erste Unreal Engine für die Musik noch ein Trackerformat, allerdings wesentlich weiter entwickelt als das vergleichsweise primitive ProTracker-Module.

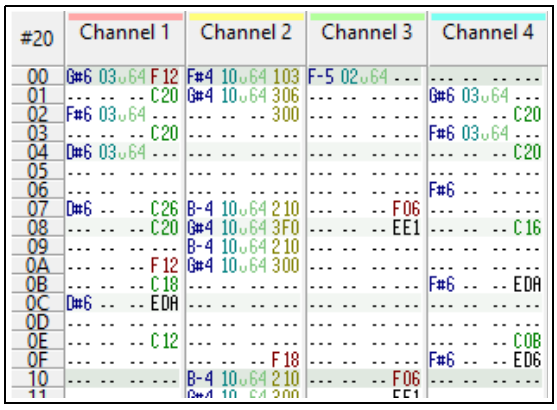
Im Audio-System der Unity Engine stehen sogenannte AudioClips im Mittelpunkt. AudioClips sind die Abstraktion von importierten Audiodateien wie wav, mp3 und ogg, um es Entwicklern zu erleichtern, Sounds in ihre Anwendungen zu integrieren. Unity ist in der Lage, ProTracker-MOD-Dateien, sowie zusätzlich FastTracker-XM, ScreamTracker-S3M und ImpulseTracker-IT zu importieren, verwandelt diese allerdings in einfache Stereo-Audioclips, wobei der Zugriff auf die darunterliegenden Daten verlorengeht.

Meine Ziel war es, ein Plugin zu entwickeln, welches Mod-Dateien nicht nur korrekt abspielen kann, sondern auch zur Laufzeit Zugriff und Modifikationen erlaubt. Zusätzlich sollte der Code so strukturiert sein, dass zukünftig weitere Formate unterstützt werden könnten.

Begriffserklärung

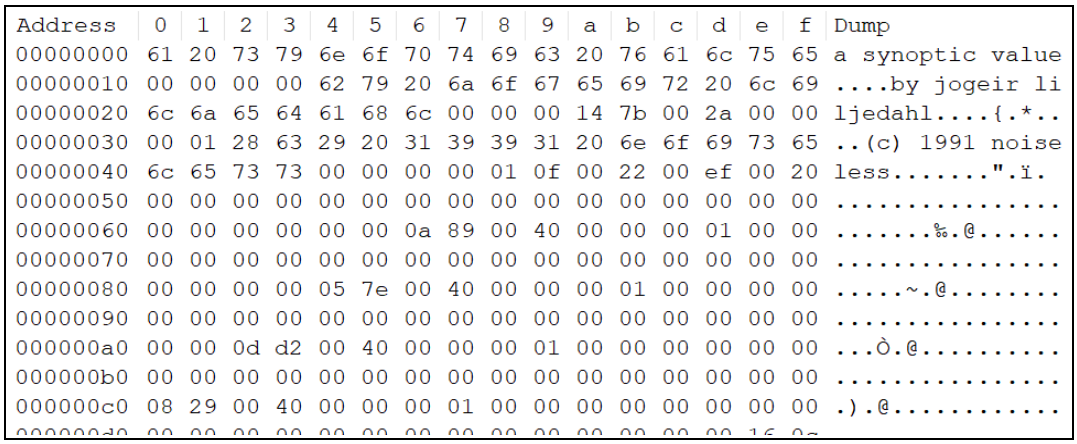
Viele der Begriffe, die im Bericht Erwähnung finden werden, sind direkt aus dem Englischen übernommen. Der Einfachheit halber, und um potentiellen Fehlübersetzungen vorzubeugen, folgt nun eine kurze Auflistung und Erklärung der Begriffe.

Waveform	Die Form einer Welle, in der Regel bezüglich der Grundschwingungen.
Square Wave	Rechteckschwingung.
Sawtooth/Saw Wave	Sägezahnschwingung.
Sine Wave	Sinusschwingung.
Sample	Entweder ein Soundclip, welcher im Song mit verschiedenen Frequenzen abgespielt als Instrument dient oder ein diskreter Zeitpunkt in einem Audio-Buffer (z.B. der Output-Buffer mit seiner Samplerate, oder die Audiodaten eines Samples).
Channel	Klangerzeuger, welcher mittels des aktuellen Samples (Soundclip) und der gewünschten Frequenz Audiodaten in den Output-Buffer schreibt.
Pattern	Eine "Seite" an Informationen, unterteilt in Rows, welche nach und nach ausgelesen und abgespielt werden und Columns, welche die einzelnen Kanäle bilden.
Row	Eine Zeile in einem Pattern, enthält pro Channel Informationen über die aktuelle Note, das aktuelle Sample und diverse Effekte.
Tick	Unterteilung von Rows beim Abspielen, Effekte werden i. d. R. pro Tick ausgeführt.
(Pattern-)Sequence	Die definierte Reihenfolge von Patterns, welche allerdings durch Effekte gebrochen werden kann.
OpenMPT [2]	Freies Trackerprogramm, mit enorm vielen unterstützten Formaten und einem hohen Standard für Genauigkeit.
MilkyTracker [3]	Freies Trackerprogramm, weniger umfangreich als OpenMPT.
Audacity [4]	Freie Software zum Aufnehmen und Analysieren von Audio.



Ausschnitt eines Screenshots eines Patterns in OpenMPT

Auslesen von Mod-Dateien



Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	61	20	73	79	6e	6f	70	74	69	63	20	76	61	6c	75	65	a synoptic value
00000010	00	00	00	00	62	79	20	6a	6f	67	65	69	72	20	6c	69	....by jogeir li
00000020	6c	6a	65	64	61	68	6c	00	00	00	14	7b	00	2a	00	00	ljedahl....{.*..
00000030	00	01	28	63	29	20	31	39	39	31	20	6e	6f	69	73	65	..(c) 1991 noise
00000040	6c	65	73	73	00	00	00	00	01	0f	00	22	00	ef	00	20	less.......".ï.
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000060	00	00	00	00	00	00	0a	89	00	40	00	00	00	01	00	00	.......‰.@......
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000080	00	00	00	00	05	7e	00	40	00	00	00	01	00	00	00	00	.....~.@........
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000a0	00	00	0d	d2	00	40	00	00	00	01	00	00	00	00	00	00	...Ò.@..........
000000b0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000c0	08	29	00	40	00	00	00	01	00	00	00	00	00	00	00	00	.).@............

Screenshot der Hex-Ansicht einer Mod-Datei (a_synoptic_value.mod von Jogeir Liljedahl) in Notepad++

Mod-Dateien beinhalten binäre Daten. Um sie auszulesen muss ihre Struktur bekannt sein. Glücklicherweise gibt es für das Format frei verfügbare Informationen bezüglich des Aufbaus. Drei Websites ([5], [6], [7]) waren außerordentlich hilfreich, um Sinn aus den Daten zu machen, auch wenn die Spezifikation anderorts ebenso auffindbar ist.

Die statische Klasse `ModFileReader` wurde erstellt und eine Methode `ReadFile`, welche als string den Pfad zu einer Mod-Datei bekommt und dessen Daten in ein C#-Objekt verwandelt, implementiert. Der erste Schritt hierbei ist es, mittels `System.IO.File.ReadAllBytes` die Daten in ein Byte-Array zu schreiben. Für die vielen Operationen auf diesem Byte-Array wurde zusätzlich eine statische Klasse `ByteArrayExtensions` angelegt, welche statische Extension-Methods bereitstellt.

```
public static string ReadText (this byte[] bytes, ref int offset, int length) {
    var sb = new System.Text.StringBuilder();
    for(int i=0; i<length; i++){
        var b = bytes[i + offset];
        if(b != 0){
            sb.Append((char)b);
        }else{
            break;
        }
    }
    offset += length;
    return sb.ToString();
}
```

Extension Methods sind ein Feature von C#, womit Objekten weitere Methoden bereitgestellt werden können, ohne diese in der Klasse implementieren zu müssen.

Funktional sind es statische Methoden, welche das angezielte Objekt als Parameter erhalten, jedoch lesen sie sich besser, wenn sie dem Objektnamen folgen und der statische Klassenname nicht erwähnt werden muss.

```
var foo = System.Text.Encoding.ASCII.GetBytes("Hello, world!");
var extensionText = foo.ReadText(ref 0, foo.length);
var staticCallText = ByteArrayExtensions.ReadText(foo, ref 0, foo.length);
```

Im obigen Beispiel beinhalten `extensionText` und `staticCallText` den gleichen string, nur dass bei ersterem die Extension aufgerufen wurde, während beim letzteren der Aufruf ganz normal statisch erfolgte. `ref`-Parameter sind ein weiteres C#-Feature, wodurch auch primitive Datentypen als Parameter in einer Methode verändert werden können und diese Veränderung zur ursprünglichen Variable gelangt.

Wie in der Hex-Ansicht der obigen Abbildung sichtbar, beginnt eine Mod-Datei mit dem Titel als Klartext. Der Titel hat allerdings ein Zeichenlimit von 20, daraufhin folgt der Sample Description Record, welcher die Metadaten der Samples enthält. Auch im Screenshot sieht man eine übliche Praxis bei Mod-Dateien: Die Samplenamen werden verwendet um zusätzliche Information zu liefern, im obigen Fall der Name des Komponisten, Jahr und Künstlername. Zusätzlich zum Namen gehören noch die Länge des Samples, der Finetune, die Lautstärke, Loopbeginn und -länge in den Record. In den meisten Fällen werden es 31 Samples sein, doch für maximale Kompatibilität mit noch älteren Mod-Dateien mit nur 15 Samples wird vorher nach einer Reihe von magischen Zeichenfolgen weiter in der Datei gesucht, die eindeutig auf 31 Samples verweist. In der Regel wird die Zeichenfolge „M.K." sein, doch andere Möglichkeiten existieren, welche zum Beispiel auf eine erhöhte Anzahl von Kanälen als den normalen vier hinweisen.

Es folgt die Länge der Pattern-Sequence, die Restart-Position sowie die Pattern-Sequence selbst. Auch wenn alle diese Werte Bytes sind, werden sie vom Reader als Ints ausgegeben, um spätere Verwendung zu erleichtern. Nach der Pattern-Sequence folgen entweder die zuvor erwähnten magischen Zeichen, oder die Patterns beginnen sofort.

Die Daten für die Patterns sind alle Rows in Folge, wobei pro Row pro Kanal vier Bytes die Daten für einen Kanal in dieser Row beschreiben. Da die Patterns und die Samples den größten Teil der Informationen ausmachen, wurde hier aufgepasst, nicht mehr Speicher zu belegen, als nötig. In den vier Bytes befinden sich 8 Bits, welche ein Sample identifizieren, 12 Bits für die Periode, mit der das Sample gespielt werden soll und 12 Bits für den Effekt. Als Kompromiss zwischen Effizienz in Speicherverbrauch und Effizienz beim Auslesen wird der Sampleindex, dessen 8 Bits nicht kontinuierlich in den vier Bytes sind, in

einem Byte gespeichert, wobei die Periode und der Effekt jeweils in einen 16-Bit Short gespeichert werden. Diese Daten werden zusammen in einem `ChannelData`-Objekt gruppiert, wobei Rows aus Arrays von ChannelDatas bestehen und Patterns als Arrays von Rows. Als zusätzliche Optimierung wurde überlegt, ob inhaltlich identische ChannelDatas nicht tatsächlich das gleiche Objekt, oder null falls leer, sein sollten. Eine kurze Rechnung zeigt allerdings auf, dass diese Optimierung für moderne Systeme komplett unnötig ist.

> *Hat ein Song die maximale Anzahl an Patterns (128) und vier Kanäle, so ergibt sich bei 64 Rows pro Pattern eine Anzahl von insgesamt 128 * 64 * 4 = 32.768 ChannelData-Objekten. Bei einer Größe von 5 Bytes (1 Byte Sample, je 2 Bytes für Periode und Effekt) ergibt dies gerade einmal 160kB, also nicht der Erwähnung wert.*

Als letztes folgen die tatsächlichen Audiodaten der Samples, encodiert als Signed Bytes. Dies fiel erst auf, als bereits mit der Implementierung vom Abspielen begonnen wurde. Die Sinus-Samples, die zum Test verwendet wurden, klangen nicht richtig. Als die Samples visuell auf den Bildschirm gezeichnet wurden, war allerdings sofort klar, wo das Problem lag. Um die gelesenen Bytes in Sbytes zu verwandeln ist ein unchecked cast vonnöten. Letztlich ist auch hier etwas bewusste Ineffizienz eingebaut, da die 8-Bit Samples beim Auslesen in 32-Bit Float-Arrays mit Werten von -1 bis 1 gespeichert werden, da diese später beim Output in Unity ohne Umwandlung direkt verwendet werden können.

Da das Byte-Offset des rohen Byte-Arrays der Mod-Datei in jeder Stufe des Auslesens per ref-Parameter modifiziert wird, kann am Ende einfach getestet werden, ob die gesamten Daten der gelesenen Datei auch verarbeitet wurden. Ist die Datei kürzer als erwartet, wird an irgendeiner Stelle ein Index außerhalb des Arrays auftauchen und zu einer Exception führen. Ist das Offset am Ende geringer als die Länge des Arrays, wird eine Warnung ausgegeben, da die Datei zwar ohne Probleme gelesen wurde, aber scheinbar nicht vollständig.

Der Output der `ReadFile`-Funktion ist ein `ModFile`-Objekt, welches die gelesenen Datenstrukturen am Ende enthält. Um von Anfang an Erweiterbarkeit in das Plugin einzubauen, ist die Klasse `ModFile` eine Erweiterung der abstrakten Klasse `File`, welche Getter für viele in allen Module-Dateien vorhandenen Informationen erzwingt, sowie statische Methoden zum Lesen mit automatischen Format-Checks bereitstellt.

Abspielen der gelesenen Dateien

Audiogenerierung

Wie zuvor erwähnt, werden Audiodateien, egal ob Musik oder Soundeffekte, von Unity in AudioClips umgewandelt, um diese über eine AudioSource-Komponente zur Laufzeit abspielen zu können. Um Mod-Dateien ohne AudioClip-Konversion abzuspielen wurden mehrere Möglichkeiten unter Betracht gezogen.

Der direkteste Ansatz war es, zur Laufzeit aus den gelesenen Samples AudioClips zu kreieren, für jeden Kanal eine AudioSource zu instanziieren und dann über den Song hinweg mit wechselnden Samples die AudioClips auszuwechseln, mit der unityinternen DSP-Clock den Abspielstart exakt zu definieren und über das Pitch-Parameter der AudioSource festzulegen, mit welcher Tonhöhe das Sample spielen sollte. So einfach dies im Konzept auch klingt, die Umsetzung hätte sich schwierig gestaltet und das Ergebnis hätte vier AudioSources (wenn nicht sogar mehr) zur gleichen Zeit verwendet. Aus Performancegründen limitiert Unity die Anzahl gleichzeitig spielender AudioSources, was zur Folge haben könnte, dass in einer Anwendung, die so ihre Musik abspielt, zeitweise Kanäle ausfallen, wenn wichtigere Soundeffekte spielen und das Limit überschreiten.

Eine weitere Idee war es, das Audio komplett außerhalb von der Unity Engine zu generieren und einige freie C#-Libraries existieren für diesen Zweck. Doch auch dies schien übermäßig kompliziert, brachte Dependencies mit sich und wäre komplett außerhalb von Unity gewesen, sodass es für potentielle Nutzer des Plugins schwerer wäre, die Musik ins Spiel zu integrieren.

Glücklicherweise gibt es in Unity tatsächlich Wege, generatives Audio über AudioSources abzuspielen. Der erste ist es, per Skript einen AudioClip zu erstellen, der keine festen Sampledaten hat, sondern diese über Callbacks streamt. Der einzige Nachteil hierbei ist es, dass diese gestreamten Daten im Clip gespeichert werden, sodass bei einem loopenden Song konstant der Speicherverbrauch ansteigen würde.

> *Wieder eine kurze Rechnung zeigt, dass 30 Minuten Stereo-Audio mit einer Samplerate von 48 kHz in einem Float-Array mit einer Länge von 30 * 60 * 48000 * 2 = 172.800.000 resultieren würden. Bei 32 Bit pro Float wären dies über 5 GB, keinesfalls akzeptabel.*

Die verwendete Methode war es, einen eigenen Audio-Filter zu implementieren. In Unity ist dies extrem einfach - eine vom Typ `MonoBehaviour` erbende Klasse muss lediglich eine Methode `OnAudioFilterRead` definieren. Eine Instanz dieser Klasse, als Komponente am selben GameObject wie entweder eine AudioSource oder ein AudioListener platziert, agiert dann als ein Audio-Filter. Des weiteren wird eine solche Instanz bei einer AudioSource, wenn kein AudioClip gespielt wird, wie ein AudioClip behandelt und liefert somit die Daten, die durch die AudioSource im Unity AudioSystem gespielt werden.

Player und PlaybackHandler

Zum Abspielen wurde die Klasse `TrackerMusicPlayer` erstellt. Sie erbt von der ebenso erstellten Klasse `AudioGenerator`, welche einige grundlegende Funktionalitäten implementiert, die der Player benötigt. Dies sind eine Methode um sicherzustellen, dass eine AudioSource-Komponente am gleichen GameObject zu finden ist und im Skript referenziert ist. Zusätzlich ist sie dafür zuständig, die Samplerate, mit der Unity agiert, zu cachen. Der TrackerMusicPlayer hingegen implementiert Methoden und Properties zum Abspielen, Stoppen und Pausieren sowie weitere trackerspezifische Features.

Vorerst ist es wichtig zu erwähnen, dass die Spiellogik in Unity, mit den bekannten Start- und Update-Methoden auf *einem* Thread läuft. Modifikationen oder gar Zugriff auf Szenenobjekte und diverse andere Systeme von außerhalb dieses Threads führen zu Exceptions. Der Aufruf von `OnAudioFilterRead` erfolgt vom *separaten* Audio-Thread, was zu genau solchen Problemen führen kann. Daher wird auch die Output-Samplerate gecached, da sie nicht vom Audio-Thread aus abgefragt werden kann.

Zur Bugprävention und besseren Organisation wurde die tatsächliche Logik, die das in `OnAudioFilterRead` gelieferte Buffer-Array mit Audiodaten füllt, in eine weitere Klasse ausgelagert, den `ModFilePlaybackHandler`. Mit zukünftiger Erweiterung im Hinterkopf implementieren PlaybackHandler das `IPlaybackHandler`-Interface und werden von den Files geliefert mittels `File.CreatePlaybackHandler`. Wird beim `TrackerMusicPlayer` die Methode `Play` aufgerufen, wird ein solcher `PlaybackHandler` erstellt und als `currentPlaybackHandler` gespeichert und die AudioSource gestartet. Von diesem Punkt an wird über den separaten Audio-Thread `OnAudioFilterRead` im `TrackerMusicPlayer` aufgerufen, ungefähr 20 Mal pro Sekunde mit jeweils einem 4096-Elementigen Float-Array als zu füllender Buffer. Der `currentPlaybackhandler` wird daraufhin angewiesen, diesen Buffer zu füllen. Dieser Aufruf befindet sich in einem try-catch-Block, was bei der Entwicklung unglaublich hilfreich war, da Exceptions nicht nur über die Stelle im Code identifiziert werden konnten, sondern auch

an welcher Stelle im Song sie passierten, indem die gefangene Exception durch die relevanten Informationen augmentiert wird, bevor sie geloggt wird.

Die erste große Hürde im PlaybackHandler war es, das Timing der Rows und Ticks korrekt zu implementieren. Mod-Files haben standardmäßig ein Tempo von 125 und eine Tickrate von 6 Ticks pro Row. Die Frage war, wie viele Samples im Output-Buffer verarbeitet werden müssen, bevor der nächste Tick ansteht. In Weblink [5] findet sich am Ende eine Erklärung, wie sich die Anzahl der Rows (dort Divisions) pro Minute berechnen lässt.

$$\frac{Rows}{Minute} = \frac{24 * Beats/Minute}{Ticks/Row}$$

Beats/Minute steht hier für das Tempo bei den standardmäßig eingestellten sechs Ticks pro Row. Bei vier Rows pro Beat kommt somit der Faktor von 24 zustande.

$$\begin{aligned}
\frac{Beats}{Minute} &= Tempo \\
\frac{Rows}{Second} &= \frac{24 * Tempo}{60 * Ticks/Row} \\
\frac{Ticks}{Second} &= \frac{Ticks}{Row} * \frac{Rows}{Second} \\
&= \frac{Ticks}{Row} * \frac{24 * Tempo}{60 * Ticks/Row} \\
&= \frac{24 * Tempo}{60} \\
\frac{Samples}{Tick} &= \frac{Samples}{Second} * \frac{Seconds}{Tick} \\
\frac{Samples}{Second} &= Samplerate \\
\frac{Samples}{Tick} &= Samplerate * \frac{60}{24 * Tempo} \\
&= Samplerate * \frac{5}{2 * Tempo}
\end{aligned}$$

Da die Samplerate normalerweise bei 48 kHz liegt, kann ohne Probleme mit Integern gerechnet werden, selbst wenn das Tempo beim Höchstwert von 255 liegt. Aus exakten

470.5882353... Samples pro Tick werden dadurch 470, was keinerlei Probleme beim Abspielen hervorruft.

```
var samplesRemaining = buffer.data.Length / buffer.channelCount;
int i=0;
while(samplesRemaining > 0){
    if(m_samplesUntilNextTick <= 0){
        NextTick();
        m_samplesUntilNextTick = (5 * buffer.sampleRate) / (2 * m_tempo);
    }
    var runLength = System.Math.Min(samplesRemaining, m_samplesUntilNextTick);
    for(int j=0; j<modFile.channelCount; j++){
        m_channels[j].WriteToBufferRegion(
            buffer,
            new IndexRange(i, i + runLength * buffer.channelCount),
            volumeMultiplier
        );
    }
    m_samplesUntilNextTick -= runLength;
    samplesRemaining -= runLength;
    i += runLength * buffer.channelCount;
}
```

Wenn im TrackerMusicPlayer `OnAudioFilterRead` aufgerufen wird, wird dies weitergeleitet an den PlaybackHandler mit der Methode `FillAudioBuffer`. Dort wird der obige Code ausgeführt, hier leicht vereinfacht dargestellt. `NextTick` inkrementiert den Tick-Counter und ruft, falls dieser die aktuellen Ticks per Row überschreitet, `NextRow` auf. `NextRow` inkrementiert den Row-Counter und ruft, falls dieser über die Anzahl Rows im aktuellen Pattern hinausgeht, `NextPattern` auf. Ist das Ende der Sequence erreicht und die Restart-Position des Songs ungültig, wird eine spezielle `JustFinishedException` geworfen, welche in `FillAudioBuffer` gefangen wird und das Abspielen beendet. Dies vereinfacht die Methoden, da die Next-Methoden somit keinen return-Type brauchen und nicht den return der jeweils aufgerufenen Methoden überprüfen müssen, um gegebenenfalls ebenso zu returnen.

Um zu verifizieren, dass das Timing korrekt ist, wurde eine Mod-Datei erstellt, welche im Takt kurze Klicks spielte. In Unity wurden der Einfachheit halber die Klick-Samples direkt in den Audio-Buffer geschrieben, ohne die Frequenzjustierungen für den korrekten Klang, da es nur um das Timing ging. Sowohl der Output von OpenMPT, wo die Datei erstellt wurde, als auch Unity, wurden mit Audacity aufgenommen und verglichen. Da die Peaks der Klicks die gleichen Abstände hatten, konnte davon ausgegangen werden, dass das Timing in Unity korrekt implementiert war.

Channels

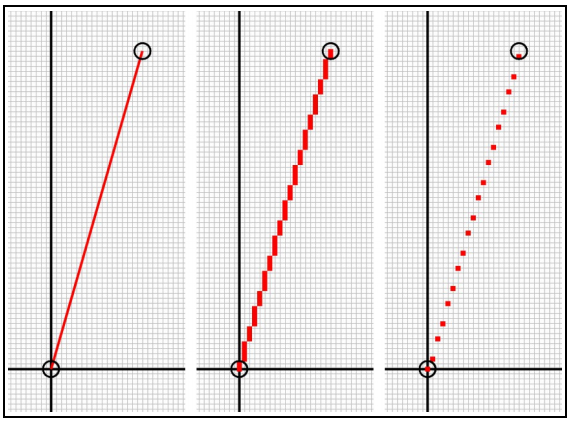
Wie im obigen Code-Snippet sichtbar, delegiert der PlaybackHandler das Schreiben der Audiodaten an seine `Channel`-Objekte mit der Methode `WriteToBufferRegion`. Um nicht nach jedem in den Output-Buffer geschriebenen Sample zu testen, ob ein neuer Tick begonnen werden muss, wird die `runLength` berechnet und daraus ein `IndexRange`-Struct als Parameter übergeben, welches einen Start- und einen End-Index bezeichnet, über deren Reichweite die Channels ihre Daten schreiben dürfen.

Sampling

Um die Samples (Soundclips) abzuspielen, müssen ihre Samples (diskrete Audiowerte) in den Output-Buffer geschrieben werden. Nimmt man ein 1:1-Verhältnis wird das Ergebnis bei einer Output-Samplerate von 48kHz sechs mal zu schnell vorübergehen und somit etwas über zweieinhalb Oktaven zu hoch sein. Der Grund hierfür ist, dass die Samples in einer Mod-Datei nur eine Samplerate von 8kHz haben. Darauf kommt noch hinzu, dass jedes abgespielte Sample mit einem Periodenwert, quasi einer Note, kommt. Dividiert man die Taktfrequenz der Amiga-CPU mit diesem Wert bekommt man die Frequenz, mit der das Sample ausgegeben wird. Die zuvor erwähnten Websites geben variierende Frequenzwerte für die Note C2 (Periode 428), so geben [5] und [7] zum Beispiel eine Frequenz von 8287 Hz für PAL-Amigas, [6] hingegen gibt 8363 Hz an ohne eine Region zu spezifizieren. Eigene Nachforschungen mit OpenMPT ergaben eine Frequenz von 8272 Hz, was als die Grundlage für Berechnungen im Channel benutzt wurde.

Das Problem, die Sample-Samples auf Output-Samples zu mappen, kann mit dem Zeichnen einer Geraden gleichgesetzt werden. Angefangen bei (0,0) ist die x-Achse der Output-Buffer und die y-Achse das Audio-Sample. Jede Einheit repräsentiert hierbei einen diskreten Samplewert. Aufgrund der höheren Output-Samplerate von 48 kHz wird die resultierende Gerade fast immer eine Steigung < 1 haben. Das Rechnen mit Gleitkommazahlen wurde kurz in Betracht gezogen, wich jedoch einer reinen Integer-Lösung: Dem Bresenham-Algorithmus [8]. Hat eine Gerade eine Steigung, die sich als ein Bruch ausdrücken lässt, so kann man mithilfe des Bresenham-Algorithmus diese Gerade nur mit Integerberechnungen zeichnen. Für die Verwendung in den Channels war jedoch eine kleine Modifikation vonnöten. Es ist zwar unwahrscheinlich, dass die Steigung größer ist als 1, aber nicht unmöglich. Ist die Steigung <= 1, wird pro Schritt der x-Wert inkrementiert, im Audio-Fall der Output-Sample-Index, während der y-Wert, der Sample-Sample-Index, nur inkrementiert wird, wenn der spezielle Error-Wert größer ist als 0. Bei

einer Steigung > 1 ist dies genau umgekehrt, standardmäßig wird y inkrementiert und x nur wenn der Error es indiziert. Da allerdings nur die Samples des Output-Buffers relevant sind, war eine Modifikation, welche auch bei einer Steigung > 1 jeden Schritt die x-Achse inkrementiert erwünscht.



Vergleich: Verbindung zwischen Punkten (L), normale Bresenham-Linie (M) und modifizierte Lösung (R)

Die naive Lösung ist es, die normale y-inkrementierende Version zu nehmen und eben nur dann zu reagieren, wenn x inkrementiert wird. Da der Wert, der pro y-Inkrement auf den Error addiert wird, allerdings konstant ist, lässt sich berechnen, wie viele y-Inkremente passieren werden, bevor er größer als 0 wird.

```
var dx = x1 - x0;
var dy = y1 - y0;
var x = x0;
var y = y0;
if(dy <= dx){                   // steigung <= 1
    var d = (2 * dy) - dx;
    while(x <= x1){
        // do something with x and y
        if(d > 0){
            y++;
            d -= 2 * dx;
        }
        d += 2 * dy;
        x++;
    }
}else if(dx > 0){               // ∞ > steigung > 1
    var dx2 = 2 * dx;
    var dy2 = 2 * dy;
    var d = dx2 - dy;
    while(x <= x1){
        // do something with x and y
        var i = 1 + ((-d + dx2) / dx2);
        y += i;
        x++;
        d += (i * dx2) - dy2;
    }
}
```

Für den Einsatz im Channel mussten nur `dx` und `dy`, sowie die Offsets bzw. Startpunkte angepasst werden.

```
static void GetBresenhamParams (int sampleRate, int period, out int dx, out int dy) {
    dx = sampleRate * period;
    dy = AMIGA_SAMPLES_PER_SECOND * AMIGA_C2_PERIOD;
}
```

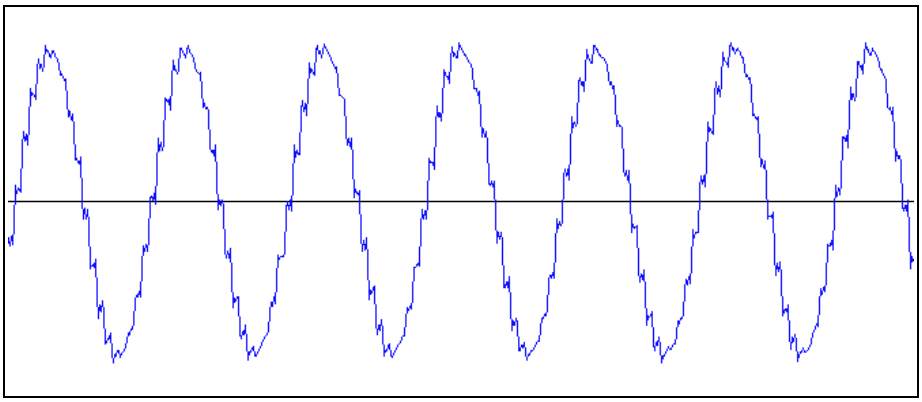
Die relevanten Variablen im Channel sind das Feld `m_t`, der aktuelle Index im Audio-Array des aktuellen Samples, was mit dem aktuellen y-Wert gleichgesetzt werden kann, sowie die Variable `i`, die der vom PlaybackHandler überlieferten IndexRange entnommen wird und für den x-Wert steht. Dementsprechend werden die out-Parameter von `GetBresenhamParams` von `dx` auf `di` und `dy` auf `dt` gemappt.

```
var i = indexRange.start;
GetBresenhamParams(outputBuffer.sampleRate, outputPeriod, out var di, out var dt);
var di2 = di * 2;
var dt2 = dt * 2;
if(dt <= di){
    while(i < indexRange.end){
        WriteCurrentSample();
        if(m_d > 0){
            if(!TryAdvanceT(1)) break;
            m_d -= di2;
        }
        m_d += dt2;
        i += outputBuffer.channelCount;
    }
}else{
    while(i < indexRange.end){
        WriteCurrentSample();
        var x = 1 + ((-m_d + di2) / di2);
        if(!TryAdvanceT(x)) break;
        m_d += (x * di2) - dt2;
        i += outputBuffer.channelCount;
    }
}
```

`TryAdvanceT` addiert den als Parameter überlieferten Wert auf das Feld `m_t` und gibt `false` zurück, falls `m_t` die Länge des Audio-Arrays des Samples überschreitet und das Sample nicht loopt. `WriteCurrentSample` liest den Audiowert aus dem Audio-Array des Samples an der Stelle `m_t` und schreibt diesen in den Output-Buffer bei `i` und `i+1` respektiv für den linken und rechten Output-Kanal, multipliziert mit dem Pan-Wert des Kanals, für Stereo.

Hierbei muss erwähnt werden, dass die Samples mit konstanter Interpolation abgespielt werden. Das Äquivalent in der Bildverarbeitung wäre Nearest-Neighbor-

Interpolation. Das Ergebnis ist ein hellerer Klang mit mehr Obertönen. Zwischen dem Schreiben der Werte in den Output-Buffer und dem Output über Windows scheint das Signal allerdings noch ein weiteres Mal verarbeitet zu werden, da die Waveform, wenn in Audacity betrachtet, nicht die konstante Interpolation aufweist.



Abgespieltes Sinus-Sample, aufgenommen mit Audacity

Eine lineare Sample-Interpolation zu implementieren würde womöglich helfen und ist in nahezu allen Tracker-Programmen als Option implementiert, wurde aber für dieses Projekt außen vor gelassen, da der Amiga ebenso mit konstanter Interpolation arbeitete. Eine mögliche Lösung, um bei Bresenham-ähnlichem Code zu bleiben, wäre Xiaolin Wu's Methode zum Zeichnen von anti-aliased Linien.

Finetuning

Mit dem Sampling-Code implementiert war es möglich, nicht nur das Timing der Ticks und Rows zu verifizieren, sondern auch die korrekte Tonhöhe der abgespielten Samples. Leider erwies sich letztere als falsch, was nach einiger Verzweiflung auf das noch nicht implementierte Finetuning zurückzuführen war. Wie in der Erklärung des ModFileReader erwähnt, haben Mod-Samples einen Namen, eine Lautstärke, einen Finetune-Wert sowie einen Loop-Start und eine Loop-Länge. Der Finetune-Wert gibt dabei an, um wie viele Achtel eines Halbtons die Periode, mit der das Sample gespielt werden soll, verändert werden muss. Ein Halbton beschreibt hierbei ein Verhältnis der zwölften Wurzel von Zwei. Zwölf Halbtöne nacheinander ergeben demnach ein Verhältnis von Zwei, was einer Oktave entspricht. Da der Finetune immer zwischen -8 und 7 liegt, lässt sich somit ein Sample stimmen, um mit anderen Samples zu harmonisieren. Für größere Veränderungen muss bei der Komposition das Sample transponiert gespielt werden.

Finetuning ist eine der wenigen Operationen, die tatsächlich mit Gleitkommaarithmetik gelöst wurden. Für maximale Präzision wurden double-Werte verwendet.

```
static readonly double RATIO_HALFTONE = System.Math.Pow(2d, 1d/12d);
static readonly double RATIO_FINETUNE = System.Math.Pow(RATIO_HALFTONE, 1d/8d);

static int ApplyTune (int input, double ratio, int steps) {
    return (int)System.Math.Round(input * System.Math.Pow(ratio, -steps));
}
```

Die ursprüngliche Implementierung von Finetune ist laut Website [5] mittels verschiedener Tables mit justierten Periodenwerten gelöst, eine Auflistung der Werte findet sich auf Seite [7]. Die Werte, die mittels Gleitkommaarithmetik berechnet werden, weichen teilweise von diesen Werten um ±1, noch seltener ±2 ab, jedoch klingt es nicht unstimmig.

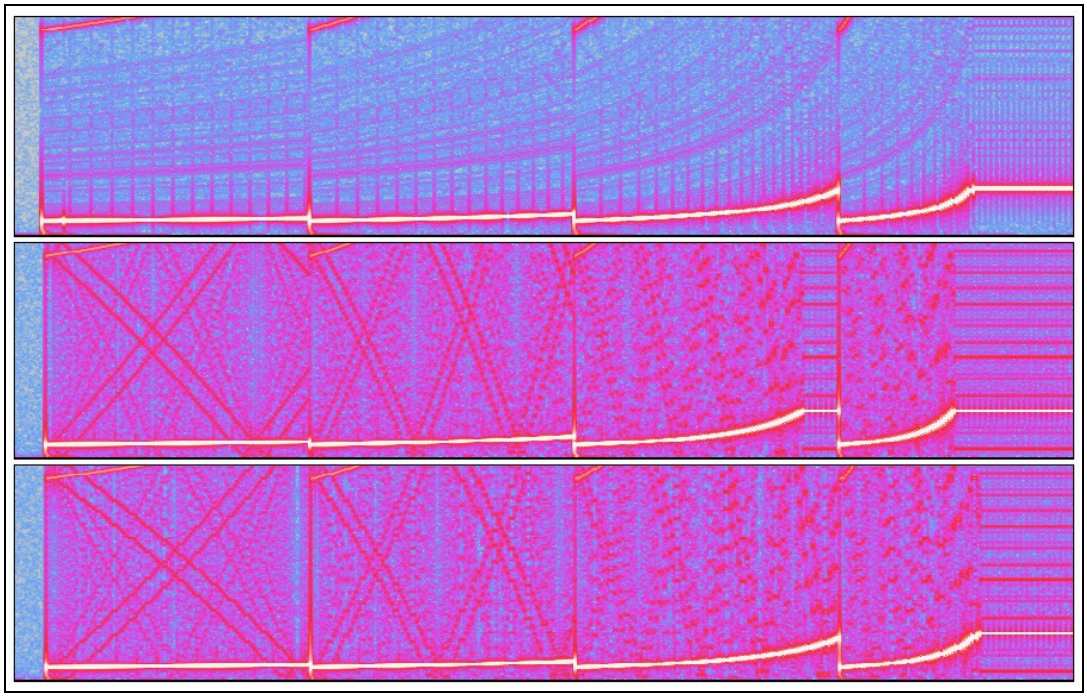
Effekte

Während das Finetuning schnell und einfach implementiert war, dauerte die Implementierung der vielen Effekte den größten Teil der Zeit des ICs. In jeder Row kann ein Effekt spezifiziert werden. Die meisten haben das Format ID-X-Y, wobei jeder Bestandteil eine Zahl von 0 bis 16 bzw. 0h bis Fh (hexadezimal) ist. Die ID zeigt an, welcher Effekt auszuführen ist und X und Y sind dessen Parameter. Effekt-ID Eh hingegen benutzt Parameter X als Identifikator für 16 weitere Effekte. Somit galt es, Effekte 0xy bis Dxy, E0x bis EFx und Fxy, insgesamt 31 an der Zahl, zu implementieren und verifizieren.

Wie zuvor Erwähnt werden Effekte pro Tick ausgeführt, somit liegt ihr Code in der Methode `OnNewTick` im Channel, mit dem Tick-Index als Parameter. Die ersten, die implementiert wurden, waren der Volume-Effekt Cxy, der Tempo-/Tickrate-Effekt Fxy sowie der Pattern Break und Position Jump Effekt Dxy und Bxy jeweils. Der Volume-Effekt ist auf den Channel, in dem er gelesen wird, begrenzt, die anderen drei werden an den PlaybackHandler weitergeleitet, da dieser für Tempo und Tickrate zuständig ist, sowie die Progression der Patterns, welche die letzteren zwei Effekte beeinflussen. Beim Vergleich mit OpenMPT fiel hierbei ein Bug in OpenMPT bezüglich Pattern Breaks auf. Der Pattern Break Effekt sorgt dafür, dass nach der aktuellen Row die Sequence advanced wird. Die x- und y-Parameter des Effekts stehen dabei für den Index der Row, in dem das neue Pattern begonnen werden soll. Aus einem unbekannten Grund setzt OpenMPT die beiden

Parameter auf 00, wenn eine Mod-Datei geladen wird, aber nur, wenn der Pattern Break der einzige im Pattern ist. MilkyTracker wies dieses Problem nicht auf.

Die nächsten implementierten Effekte waren die Slides (1xy und 2xy), bei denen die Periode pro Tick, abhängig vom Effektparameter, justiert wird. Als Referenzsample wurde eine einfache Sinusschwingung verwendet und mithilfe von Audacity's Spektrogramm ließ sich die Implementierung überprüfen.



Spektrogramm eines Sinus-Samples mit Slide-Effekt verschiedener Stärken.
Oben: OpenMPT, Mitte: Eigene Lösung, Unten: Eigene Lösung nach Fix.

Was beim Spektrogramm sofort ins Auge fällt ist die enorme Menge an Obertönen in meiner Implementierung, sichtbar durch das Pink, wo bei OpenMPT eher Blau ist. Diese sind bei Sinus-Samples äußerst auffällig. Davon abgesehen fällt allerdings auf, dass in der Mitte die helle Linie im dritten und vierten Abschnitt früher ein konstantes Level erreicht. Dies kommt daher, dass es Effekte gibt, die nur beim 0-ten Tick ausgeführt werden, wie die zuerst implementierten, und dann die Effekte, die auf allen Ticks > 0 ausgeführt werden, wie zum Beispiel Slides. Dies fiel nicht auf, bis Vibrato (4xy) und Tremolo (7xy) implementiert wurden und die unterschiedlichen Oszillationsraten nicht nur im Spektrogramm auffielen, sondern auch eindeutig hörbar waren.

Die beiden Oszillatoreffekte gehören zu den komplizierteren, da sie nicht besonders detailliert beschrieben sind. Es wurde viel in OpenMPT getestet und mit Audacity analysiert um herauszufinden, ob Oszillatoren weiter oszillieren, wenn sie nicht verwendet werden, und wie ihre Waveforms tatsächlich aussehen. Allerdings stellte es sich aus heraus, dass die

Sinusschwingung die bei weitem am meisten benutzte ist und MilkyTracker beispielsweise gar nicht erst die Saw Wave, Square Wave und Random Wave implementiert hat. Nichtsdestotrotz sind alle Formen in der eigenen Lösung verfügbar, auch wenn die Random Wave anders als in OpenMPT sich nicht alle 64 Samples wiederholt, sondern kontinuierlich neue Werte liefert.

Ein weiterer seltener Effekt, der trotzdem implementiert wurde, ist Effekt E3x, welcher für den Portamento-Effekt 3xy Glissando aktiviert oder deaktiviert. Glissando bedeutet, dass die Periode nicht glatt bewegt wird, sondern zum nächsten Halbton hin gerundet wird. Dokumentation, wie dies zu erreichen ist, gab es nicht. In OpenMPT's Effektbeschreibung ist notiert „*This effect is not widely supported and behaves quirky in OpenMPT.*", Seite [6] beschreibt den Effekt mit „Screw up 3xx. *(please clarify: how does this actually work?)*". Ein wenig Nachforschung im Internet brachte ein GitHub-Repository mit einem Mod-Player [9], geschrieben in nim. Um Glissando zu implementieren, bleibt das normale Portamento erhalten, nur wird bevor Daten in den Output-Buffer geschrieben werden, die Periode temporär auf eben die nächste verfügbare Periode gerundet. Im Gegensatz zur online gefundenen Variante wird die gerundete Periode allerdings nicht aus einer Tabelle gelesen, sondern wieder per Gleitkommaarithmetik berechnet.

```
static int GetNearestHalftonePeriod (int inputPeriod, int finetune) {
    var basePeriod = ApplyTune(AMIGA_C2_PERIOD, RATIO_FINETUNE, finetune);
    var periodRatio = (double)basePeriod / (double)inputPeriod;
    var rawHalfSteps = System.Math.Log(periodRatio, RATIO_HALFTONE);
    return ApplyTune(basePeriod, RATIO_HALFTONE, (int)System.Math.Round(rawHalfSteps));
}
```

Extrem verwirrend war, dass bei meinen Tests allerdings Glissando-Slides stets unharmonisch klangen. Dies lag daran, dass nach dem Runden wieder der aktuelle Finetune angewandt werden musste. Bevor dies klar war, wurden einige Stunden lang Alternativen zum obigen Code getestet, inklusive einer Tabelle für maximale Genauigkeit.

Die meisten anderen Effekte waren im Vergleich zu den vorigen wesentlich einfacher zu implementieren, da die diversen Feinheiten von Non-Zero-Ticks, den Amiga-Period-Limits und Finetune dann bekannt waren. Die letzten zwei Effekte Loop Pattern (E6x) und Delay Pattern (EEx) brachten allerdings noch einmal Komplikationen mit sich. Und auch hier hatte OpenMPT wieder Probleme, wo MilkyTracker zur Verifikation benutzt wurde, da diese Probleme eindeutig Bugs waren.

Letzten Endes sind zwei Effekte trotzdem unimplementiert geblieben. Der erste ist Effekt E0x, ein tatsächlich recht häufig vorkommender Effekt, welcher sich allerdings an die

Amiga-Hardware richtet und dort einen Filter aktiviert oder deaktiviert, je nach Parameter. Da die Hardware im eigenen Player nicht vorhanden ist und die Implementierung eines Software-Filters zusätzlich zum Player als unnötig kompliziert angesehen wurde, blieb dieser Effekt aus. Der andere Effekt ist Effekt EFx, bekannt als „Invert Loop“ oder „Funkrepeat“. Beide Begriffe bezeichnen komplett verschiedene und extrem schlecht dokumentierte Verhalten. Dazu kommt, dass bei eigenen Nachforschungen nur zwei Songs identifiziert werden konnten, die diesen Effekt benutzen und dass OpenMPT's Implementierung scheinbar auch problembehaftet ist, während MilkyTracker ihn komplett ignoriert. Daher wurde auch hier der Entschluss gefasst, den Effekt unimplementiert zu lassen.

Verifikation mit Songs

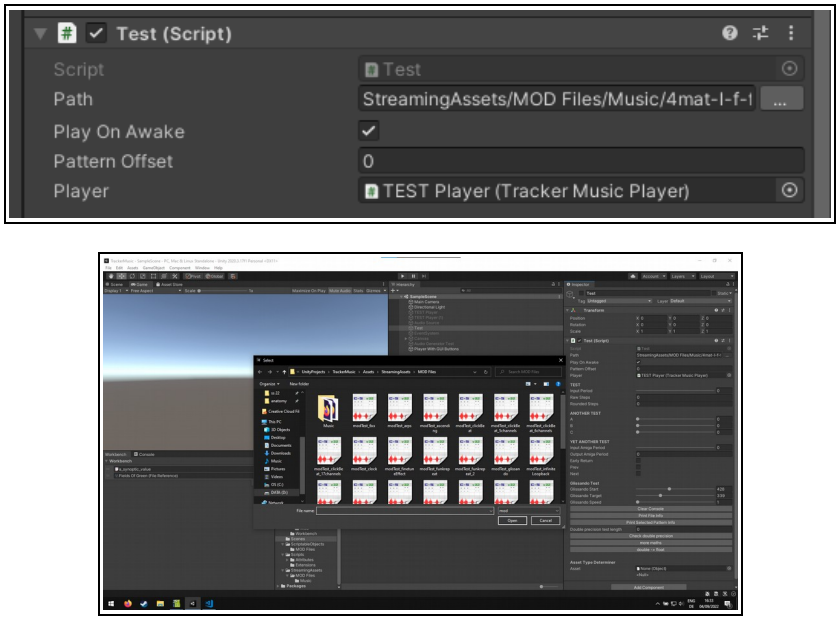
Mit allen (relevanten) Effekten implementiert war der Player funktional fertig. Damit war es an der Zeit, von selbst erstellten Mod-Dateien mit Sinus-Samples zu echten Songs zu gehen und zu testen, ob alles funktioniert. Nachdem direkt beim ersten Song NullReferenceExceptions geworfen wurden, weil dieser Periods benutzte, bevor überhaupt ein Sample gesetzt war, und andere kleine Bugs behoben wurden, zum Beispiel, dass das Sample-Volume bis zu dem Zeitpunkt hin nie verwendet worden war, konnte mit A/B-Tests begonnen werden. Hierfür wurde erst der Song in OpenMPT gespielt und mit Audacity aufgenommen, anschließend der selbe Song im eigenen Player in Unity abgespielt und ebenso auf einem neuen Track aufgenommen. Die beiden Aufnahmen wurden dann synchronisiert und parallel abgespielt, wobei entweder ein Song auf dem linken und der andere auf dem rechten Ohr gespielt wurde, oder mittels der Solo-Funktion zwischen beiden gewechselt wurde. Das Ergebnis war, dass bei den meisten Songs absolut kein Unterschied hörbar war. Wenn Samples verwendet werden, die keine reinen Sinusschwingungen beinhalten, sind die durch die konstante Interpolation hervorgerufenen Obertöne wesentlich weniger auffällig und tragen sogar zum Klang des Songs positiv bei. Dies fällt auf, wenn man mit MilkyTracker vergleicht, wo stets lineare Interpolation vorgenommen wird, wodurch Songs merklich weicher und dumpfer klingen.

Einer der letzten Bugs war es, dass bei einem Song die Slides zu früh terminierten, was extrem dissonant klang. Der Grund war, dass in meiner Implementierung Slides stets die Amiga-Limits beachteten, mit 113 als der Minimumwert und 856 als der Maximumwert für eine Periode. Die Lösung war es, den Song im Vorhinein zu Scannen und falls Perioden außerhalb dieser Limits gefunden werden, die Limits bei Slides zu deaktivieren.

Usability und Editor Scripting

Der PlaybackHandler, die Channels und generell alle Klassen außer dem TrackerMusicPlayer sind unabhängig von der Unity-API implementiert. So wurde zum Beispiel für mathematische Funktionen `System.Math` statt `UnityEngine.Mathf` verwendet, auch wenn dies teilweise ein wenig umständlicher war. Dies hat den Vorteil, dass es möglich ist, das Abspielen von Mod-Dateien auch in andere C#- oder .NET-Anwendungen zu übertragen, ohne großen Aufwand. Die Klassen, die geschrieben wurden, um den Player in ein Unity Plugin zu verwandeln, sind hingegen nicht übertragbar, da sie, wenig überraschend, auf die Unity-API angewiesen sind.

Da der Player und PlaybackHandler eine Instanz der Klasse ModFile bekommen, welche wiederum vom ModFileReader erstellt wird, indem dieser eine Datei auf der Festplatte liest, war ein Weg vonnöten, Mod-Dateien einfach und effizient zum Abspielen auswählen zu können. Während die Logik des Abspielens noch in der Implementierungsphase war, lieferte eine Test-Klasse beim Starten des Play-Modes in Unity das ModFile an den TrackerMusicPlayer. Um für die häufigen Wechsel der zu testenden Dateien nicht ständig im Code einen string ändern zu müssen, oder von Hand Pfade in im Inspector sichtbare Felder schreiben oder einfügen zu müssen, wurde die Attributsklasse `OpenFilePanelAttribute` geschrieben. Zusammen mit einem `PropertyDrawer`, der Unity darin instruiert, wie das Feld im Inspector zu zeichnen ist, ließen sich Pfade dann einfach über einen File-Browser selektieren.



Oben: Feld mit Namen "Path" im Unity-Inspector mit Custom Property Drawer für den "..."-Button
Unten: Der File-Browser, der sich öffnet, wenn der "..."-Button geklickt wird.

Für ein Plugin musste allerdings eine bessere Lösung her, vorzugsweise eine, die ähnlich zu den Unity-nativen AudioClips funktioniert. Wird eine Audiodatei in Unity importiert, so generiert Unity für diese eine Meta-Datei und speichert sie als ein AudioClip-Asset in der AssetDatabase, welche im Content Browser dargestellt ist. Vom Content Browser aus, oder über ein Kontextmenü, verfügbar bei AudioClip-Feldern, kann dann das AudioClip-Asset einfach dort referenziert werden. Selbst, wenn die Audiodatei innerhalb des Projekts umbenannt wird, oder an einen anderen Ort verschoben wird, bleiben Referenzen weiterhin gültig.

File References

Tatsächlich erstellt Unity für *alle* Dateien, die ins Projekt importiert werden, Assets, welche referenziert werden können. Problematisch ist nur, dass von wenigen Dateitypen abgesehen, die meisten als `UnityEditor.DefaultAsset` importiert werden. Der UnityEditor-Namespace ist nur im Editor gültig und wird nicht für Builds kompiliert, somit war es nicht möglich, Mod-Dateien direkt zu referenzieren. Glücklicherweise erlaubt es Unity, von der Klasse `ScriptableObject` zu erben, womit eigene Klassen als Assets im Projekt existieren können. So wurde die Klasse `FileReference` erschaffen, welche dazu dient, innerhalb der Unity Engine Trackerdateireferenzen als Assets auszudrücken. Für Builds muss hierbei der Pfad zur Datei gespeichert werden, da wie zuvor erwähnt, das korrespondierende *Asset* nicht im Build existieren wird, die *Datei* aber unter den richtigen Bedingungen sehr wohl. Für die Bequemlichkeit, die Trackerdateien während der Entwicklung im Editor bewegen und umbenennen zu können, wird trotzdem eine Referenz zu diesen gespeichert. Mithilfe von plattformabhängiger Kompilierung ist dies reibungslos möglich.

```
#if UNITY_EDITOR
    [SerializeField] Object m_targetAsset;
#endif
    [SerializeField] string m_localFilePath;

#if UNITY_EDITOR
    public string localFilePath { get {
        if(m_targetAsset == null){
            return string.Empty;
        }
        return UnityEditor.AssetDatabase.GetAssetPath(m_targetAsset);
    } }
#else
    public string localFilePath => m_localFilePath;
#endif
```

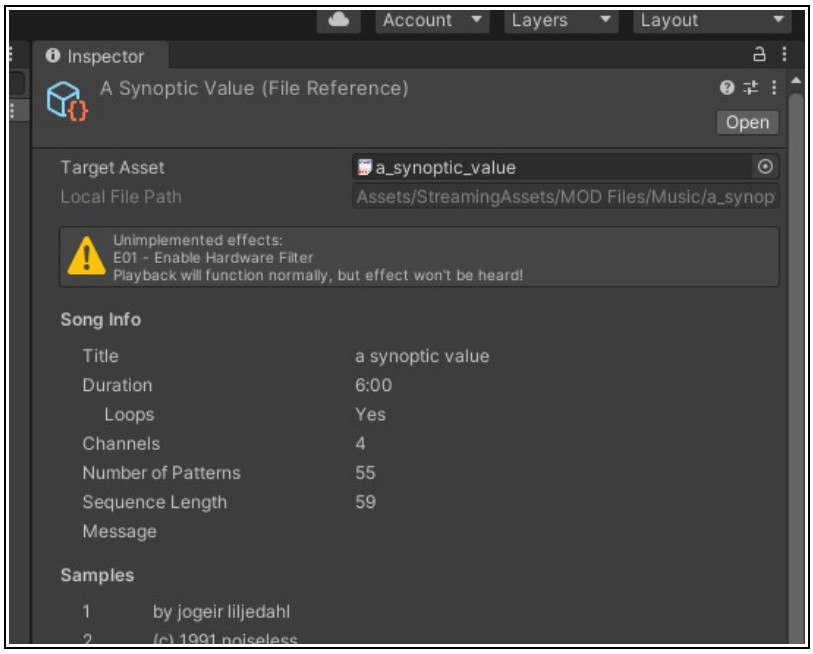
Das obige Code-Snippet zeigt, dass solange im Editor agiert wird, der Pfad der referenzierten Datei aus der AssetDatabase bei Abfragen zurückgegeben wird, in Builds jedoch der als string fest gespeicherte Wert. Eine weitere Klasse, die das Interface `IPreprocessBuildWithReport` implementiert, sorgt dafür, dass bei Builds in allen `FileReference`-Objekten das `m_localFilePath`-Feld den korrekten Wert enthält.

FileReferences können wie normale Unity-Assets per Rechtsklick im Inspector oder über das Create-Menu erstellt werden. Dabei muss dann die Trackerdatei manuell in das `m_targetAsset`-Feld gezogen werden. Um die Usability zu verbessern, wurde zusätzlich implementiert, dass beim Rechtsklick auf solch ein Trackerdatei-Asset zusätzlich die Option auftaucht, eine FileReference aus der Selektion zu erstellen.

Letztlich wurde die Klasse TrackerMusicPlayer noch so modifiziert, dass sie sowohl zur Laufzeit direkt mit gelesenen Dateien, als auch mit FileReferences funktioniert.

Custom Editors

Wie zuvor erwähnt, sind Teile des Unity Editor Interfaces modifizierbar, wie mit dem Custom PropertyDrawer für das Attribut, das das Auswählen eines Dateipfads per Dateibrowser ermöglichte. Eine weitere Möglichkeit, den Editor zu verändern sind sogenannte CustomEditors, mit denen die gesamte Repräsentation eines Objekts im Inspektor definiert werden kann.



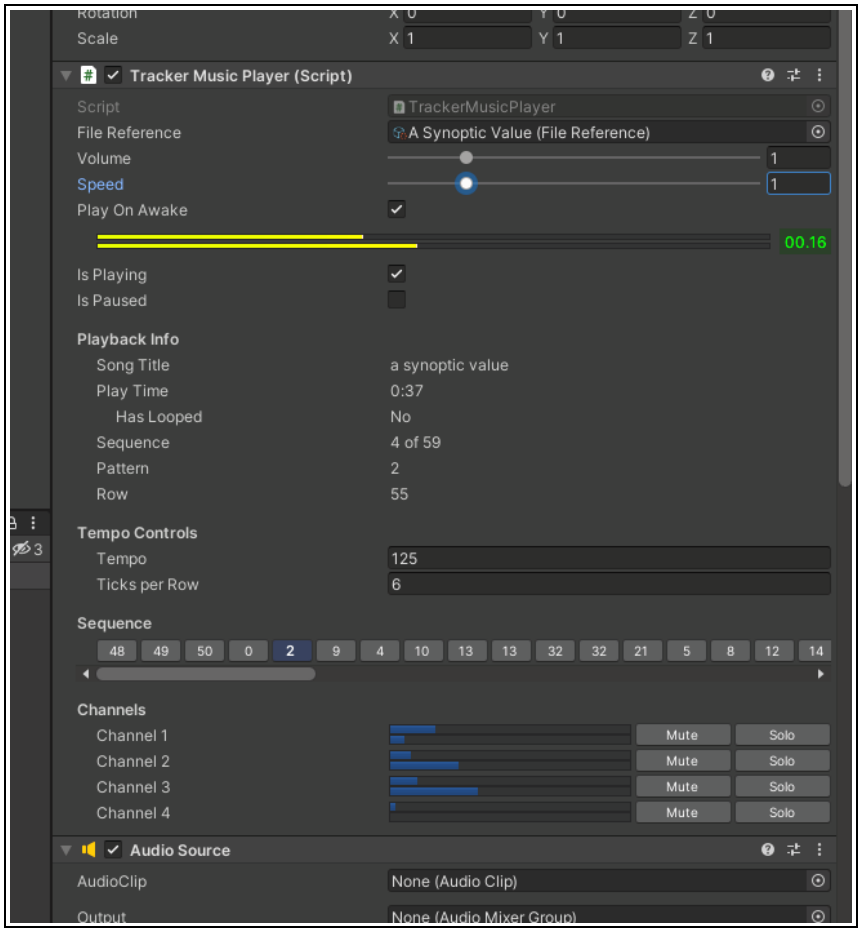
CustomEditor für FileReference-Objekte

Der Editor für FileReference-Objekte hat somit nicht nur sichtbare Felder für die referenzierte Datei und deren Pfad, sondern sorgt auch dafür, dass der Pfad sowohl korrekt als auch nicht veränderbar ist. Zusätzlich werden Hinweise in der Form von Warnungen, sowie die Metadaten der Datei angezeigt. Warnungen gibt es beispielsweise, wenn eine Datei nicht abspielbare Effekte beinhaltet, wie oben angezeigt. Zusätzlich wird gewarnt, wenn die referenzierte Datei nicht im StreamingAssets-Ordner liegt, da dies ein besonderer Ordner ist, der von Unity so wie er im Editor ist, in Builds übertragen wird. Normalerweise werden alle Assets in große binäre Archive komprimiert und sind nur noch über unityinterne Methoden erreichbar. Da die Trackerdateien allerdings von der Festplatte gelesen werden müssen und ohnehin nicht als Assets in Builds existieren würden, ist der StreamingAssets-Order die beste Lösung.

Einige Metadaten wie Songtitel, Anzahl der Kanäle und Samplenamen waren direkt aus der gelesenen Datei entnehmbar. Um die Dauer zu berechnen, und eventuelle nicht implementierte Effekte zu finden, wurde der PlaybackHandler mit einer statischen Methode ausgestattet, bei der der gesamte Song ohne Audioausgabe durchgespielt wird, die Dauer gemessen und eventuelle Fehler notiert. Hierfür war es zusätzlich notwendig, Loops zu erkennen, da sonst die Erkennung der Länge den Editor einfrieren würde, was während der Implementierung dieses Features mehrmals der Fall war.

```
    public static void ScanForDurationAndUnimplementedEffects (ModFile file, out float
duration, out bool loops, out IEnumerable<string> unimplementedEffects) {
        var player = new ModFilePlaybackHandler(file);
        var iPlayer = (IFilePlaybackHandler)player;
        iPlayer.onLooped += iPlayer.Stop;
        ((IFilePlaybackHandler)player).InitializePlayback(0);
        try{
            var sw = new System.Diagnostics.Stopwatch();
            sw.Start();
            while(sw.ElapsedMilliseconds < 1000){
                player.NextTick();
                player.m_currentTick += (player.m_ticksPerRow - 1);
            }
            sw.Stop();
            player.m_playtimeCounter = double.NaN;
        }catch(JustFinishedException){
            player.onFinished();
        }
        duration = iPlayer.playTime;
        loops = iPlayer.hasLooped;
        unimplementedEffects = player.unimplementedEffectsEncountered;
    }
```

Letztlich erhielt auch der TrackerMusicPlayer einen CustomEditor. Per Code war es bereits möglich, während der Song spielte, Kanäle stummzuschalten, vorzeitig Patterns zu wechseln und mehr.



Inspector des TrackerMusicPlayers während ein Song spielt.

Unter dem Feld „Play On Awake“ ist eine von Unity für Audio-Filter bereitgestellte Stereovisualisierung der ausgegeben Lautstärke zu sehen mit der Rechendauer für den letzten Output-Buffer als Zahl in Millisekunden rechts daneben. Bevor ähnliche Visualisierungen für die einzelnen Kanäle implementiert wurden, lief ein Song mit zwölf Kanälen bei durchschnittlich 0.12ms pro `OnAudioFilterRead`-Aufruf. Mit dem sogenannten „Peaking“ implementiert ging dies hoch auf 0.6ms, was immer noch weit unter den ungefähr 20ms ist, die `OnAudioFilterRead` maximal dauern darf, bevor Probleme auftreten.

Letztlich wurde noch zusätzlich zu Tempo und Ticks per Row ein Speed-Wert eingebaut, um die Abspielgeschwindigkeit justieren zu können. Die ersteren zwei Werte können vom Song selbst gesetzt werden und somit sind überschriebene Werte nicht unbedingt persistent. Der Speed-Wert, dessen Implementierung ebenso viele Aufhänger wie die Berechnung der Spieldauer forderte, ist rein extern, genau wie der Volume-Wert.

Ausblick

Wie zuvor erwähnt, ist das Abspielen tatsächlich unabhängig von der Unity-API implementiert, was Portabilität in andere Anwendungen vereinfacht. Weiterhin bemerkenswert ist, dass die einzige Gleitkommaarithmetik in den Channels dazu dient, Lookup-Tables zu ersetzen. Von Lautstärke bis zu Periodenwerten sind alle Variablen Integer.

Ein Feature, welches in der Zukunft wahrscheinlich implementiert werden wird, ist der Export einer Datei als Wav-Audiodatei in Unity. Dies hat zwar keinen wirklichen Zweck im Plugin, wird allerdings das A/B-Testen deutlich vereinfachen, wenn nicht erst der Song bis zum Ende laufen muss während Audacity den Windows-Stereomix aufnimmt. Die Implementierung wäre ähnlich zur oben gezeigten `ScanForDurationAndUnimplementedEffects` -Methode, nur dass tatsächlich ein Audio-Buffer gefüllt werden würde. Dieser wäre dann nur in die Binärdaten einer Wav-Datei umzuwandeln und auf die Festplatte zu schreiben.

Letztlich erlaubt es die Struktur des Plugins mit seinen bereits verwendeten abstrakten Klassen und Interfaces, einfach weitere Formate hinzuzufügen. Es wären lediglich ein neues File und ein neuer Reader vonnöten, sowie ein neuer PlaybackHandler mit den Feinheiten des neuen Formats. Wird der Reader in der Klasse File vermerkt, lassen sich sogar direkt FileReferences zu diesen neuen Dateien erstellen ohne Refactoring.

```
private static bool TryGetReader (
    string extension,
    out System.Func<string, File> readFile
){
    extension = extension.ToLower();
    if(extension.StartsWith(".")){
        extension = extension.Substring(1);
    }
    switch(extension){
        case "mod":
            readFile = (path) => TrackerMusic.Mod.ModFileReader.ReadFile(path);
            return true;
        default:
            readFile = default;
            return false;
    }
}

public static bool FormatIsSupported (string extension) {
    return TryGetReader(extension, out _);
}
```

Es besteht die Hoffnung, die Entwicklung des Plugins im nächsten IC fortsetzen zu können, mit dem Ziel, das XM-Format, einer Weiterentwicklung von Mod, zu implementieren.

Zeiterfassung

Datum	Beschreibung	Zeit (Std)
07/04/22	Nachforschung bezüglich des Mod-Formats, Aufsetzen des Projekts	3.5
21/04/22	Experimente mit der Erstellung von AudioClips, Audiogenerierung mit OnAudioFilterRead	1.5
22/04/22	Weitere Experimente mit Audiogenerierung, Determinieren der Unity Samplerate	1
29/04/22	Anfang der Implementierung des ModFileReaders	3.5
02/05/22	Refactoring des ModFileReaders, Erstellung von consts für Magic Numbers	2.5
03/05/22	Vollständiges Auslesen von Mod-Dateien implementiert	4
07/05/22	Anfang der Implementierung des TrackerMusicPlayers und ModFilePlaybackHandlers	4
19/05/22	Timing von Divisions und Patterns im PlaybackHandler korrekt implementiert	2.5
20/05/22	Erste Ausgabe von Sound aus einer abgespielten Mod-Datei, Fixen vieler Bugs, die dabei auffielen, Experimente mit einem SamplePlayer, der gecancelled wurde	6
23/05/22	Werte zur korrekten Samplingrate von Samples bestimmt und eingebaut	3
24/05/22	Experimente mit Bresenham zum Sampling	3
29/05/22	Bresenham beim Sampling implementiert	2
30/05/22	Finetune mittels Gleitkommaarithmetik implementiert, Start mit Effekten	4
07/06/22	CustomPropertyDrawer zur Dateiauswahl, weitere Effekte implementiert	4
08/06/22	Versuchte Implementierung von Effekt 3xy (Portamento)	1
27/06/22	Fix der Implementierung von Slide-Effekten (1xy, 2xy, 3xy) mittels Non-Zero-Ticks	0.5
28/06/22	Korrekte Implementierung von Effekt 3xy und Axy (Volume Slide)	3
02/07/22	Anfang der Implementierung von Vibrato und Tremolo (4xy, 7xy)	3
03/07/22	Fortsetzung der Implementierung von Vibrato und Tremolo	2
04/07/22	Fertigstellung der Implementierung von Vibrato und Tremolo, Finden und Fixen von Bugs nachdem zum ersten Mal das Abspielen von a_synoptic_value.mod versucht wurde	8
07/08/22	Überlegung bezüglich Scriptable Objects für Referenzen zu Trackerdateien, weitere Effekte implementiert	2

08/08/22	Fine Slide-Effekte implementiert, Anfang Implementierung Glissando	2
09/08/22	Fertigstellung Implementierung Glissando	5
11/08/22	Weitere Effekte implementiert, Variablen in Channels auf Integer umgeschrieben	3
12/08/22	Weitere Effekte implementiert, Nachforschung bezüglich Sample-Retriggering in OpenMPT bei Samplewechseln ohne mitgelieferte Periode	4
13/08/22	Weitere Effekte implementiert, Refactoring	4
14/08/22	Letzte Effekte implementiert (E6x, EEx), Verwendung von Sample-Volume eingebaut, Verifiziert, dass Output-Level identisch ist mit OpenMPT	3
15/08/22	A/B-Testing mit Songs, Fixen von dabei gefundenen Problemen	2
16/08/22	Mehr A/B-Testing, mehr Bugfixing	2
18/08/22	Mehr A/B-Testing, keine Bugs mehr gefunden, Refactoring	3
20/08/22	Anfang Implementierung FileReference, Anfang Implementierung von File-Scanning, Probleme mit unendlichen Loops aufgrund von falscher Implementierung von Effekt EEx	10
21/08/22	Fixen der Probleme mit unendlichen Loops, Peaking eingebaut für Editor	4
22/08/22	Abkopplung von Unity-API im Playbackhandler und weiterem (größtenteils Ersatz von UnityEngine.Mathf mit System.Math), Refactor des File-Scannings, Hinzufügen von Events, Erweiterung des Editors mit Tempo und Ticks per Row, Erstellung von BuildHandler, um in FileReferences bei Builds die korrekten Pfade zu speichern, Builds gemacht um Funktionalität zu verifizieren	12
23/08/22	FileReference aus Selektion erstellen implementiert, Mod-Datei mit nicht implementiertem Effekt EFx hinzugefügt, der Vollständigkeit halber	1
26/08/22	Channel-Interface-Refactor, um Volume-Fading zu erlauben, statt nur Stummschaltung	1
	Gesamtdauer	120

Links

[1] FamiTracker
http://www.famitracker.com/

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

[3] MilkyTracker | News
https://milkytracker.org/

[4] Audacity ® | Free, open source, cross-platform audio software for multi-track recording and editing.
https://www.audacityteam.org/

[5] Protracker Module Format
http://www.aes.id.au/modformat.html

[6] Protracker Module - MultimediaWiki
https://wiki.multimedia.cx/index.php/Protracker_Module

[7] Protracker - ExoticA
https://www.exotica.org.uk/wiki/Protracker

[8] Algorithm for computer control of a digital plotter
https://www.cse.iitb.ac.in/~paragc/teaching/2010/cs475/papers/bresenham_line.pdf (toter Link)
https://web.archive.org/web/20160117124048/https://www.cse.iitb.ac.in/~paragc/teaching/2010/cs475/papers/bresenham_line.pdf

[9] nim-mod/renderer.nim at master · johnnovak/nim-mod · GitHub
https://github.com/johnnovak/nim-mod/blob/master/src/renderer.nim

Alle Links wurden zuletzt abgerufen am 05.09.2022.