Audio-Plugins

Allgemein

Audio-Plugins sind modulare Programme, die digitale Signale erzeugen und verändern. Sie können dadurch als Effekt (z.B. Reverb, Equalizer, Kompressor) oder Instrument (z.B. Synthesizer, Sampler) eingesetzt werden. Außerdem gibt es auch Plugins, die das Signal nicht verändern. Dies ist sinnvoll bei Visualisierungs- und Messfunktionen.
Der Name “Plugin” kommt aus dem Englischen für auf- bzw. einstecken und steht für die Modularität der Programme, wodurch sie auf eine Vielzahl von Fällen angewendet werden können. Ein Beispiel sind Audio-Editoren, wie z.B. *Audacity* oder *Adobe Audition*.
Ihr wahres Potential können Audio-Plugins mithilfe von Programmen entfalten, die mehrere Plugins in komplexen Anordnungen und in Echtzeit anordnen und ausführen können. Die wichtigsten Vertreter sind die *Digital Audio Workstations* (DAWs). Sie bieten nicht nur die Funktion des Plugin-Hosts, sondern auch Multit-track-Aufnahmen, Mixing, Mastering und viele andere, zur Musikproduktion notwendige Eigenschaften.

Geschichte

Vor der Digitalisierung durch Computer waren Tonstudios ausschließlich mit analogen Elementen ausgestattet. Entsprechend musste für alle Aufgaben ein physisches Gerät vorhanden sein, was die Kosten stark in die Höhe trieb. Außerdem waren Aufnahmen auf Tonbändern aufwendig und teuer, sodass Multi-track-Aufnahmen nur sehr selten möglich waren. Weil alles gleichzeitig gespielt, gemischt und aufgenommen wurde brauchte jedes Instrument mindestens einen Mischpult-Kanal.
Noch bevor Computer leistungsfähig genug waren, um in Software mit digitalen Signalen zu arbeiten, zeigten sich die Vorteile digitaler Geräte in Form von Samplern und Effektgeräten.

Die vollständige Digitalisierung begann mit den ersten Sequencern, die in der Lage waren verschiedene Samples anhand eines programmierten Musters (Pattern) abzuspielen. So entstandt die Möglichkeit, Musikstücke ausschließlich am Computer zu erstellen. Je

größer mit der Zeit die Leistungskapazität wurde, desto komplexere Aufgaben konnte die Software übernehmen.

Vorteile

Im Vergleich zu analogen Pendanten ist es sehr viel günstiger digitale Audio-Plugins zu erwerben und einzusetzen. Dadurch sinkt die Einstiegsschwelle und mehr Menschen wird ermöglicht selbst Musik am Computer zu machen und aufzunehmen. Auch können kleinere Studios eine professionelle Umgebung bieten, ohne dass sie auf teures Equipment und große Räumlichkeiten angewiesen sind.
Ein weiterer großer Vorteil ist, dass digitales Material nicht nur gespeichert und arrangiert werden, sondern auch in Echtzeit bearbeitet werden kann. Aufnahmen können beliebig oft wiederholt und Effekte können live genutzt werden.
Außerdem sind heutzutage sogar aktuelle Laptops leistungsstark genug, um Musik zu machen, was die künstlerische Freiheit stark erhöht. Mithilfe von Sample-Libraries und Software-Synthesizern enthfällt auch das Hinderniss der Größe einiger Instrumente.
Anwendungsgebiete erstrecken sich von kleinen Wohnungen in denen kein Platz für ein Klavier ist bis Orchester-Komponisten, die auch ohne echte Künstler ihre Musik testen und hören können.

Formate und Technische Umsetzung

Um die Kompatibilität aller verschiedenen Software-Plugins und -Hosts zu gewährleisten sind gemeinsame Format-Standards notwendig. Zu den am meisten genutzen zählen VST (Virutal Studio Technoligy, Steinberg), AU (Audio Unit, Apple), RTAS (Real Time AudioSuite, Avid) und LADSPA (quelloffenes Format, LGPL). Alle Plugins besitzen mehrere Schnittstellen zum Host-Programm. Im Einfachsten Fall bestehen diese aus jeweils einem Audio-Ein- und Ausgang sowie einer Benutzeroberfläche (User-Interface). Für Instrument- und komplexere Effekt-Plugins kommen noch MIDI Ein- bzw. Ausgänge hinzu.

Digitale Darstellung von Signalen

Zur digitalen Darstellung von analogen Signalen, wird dieses mehrere tausend Male pro

Sekunde in Form einer Spannungsmessung abgetastet. Der diskrete Wert wird als digitales Sample, oft in Form einer vorzeichenlosen Ganzzahl (unsigned integer) gespeichert. Um die Signale möglichst originalgetreu zu halten werden die Abtastrate und Größe des Speichertyps so hoch gewählt, dass der Unterschied zwischen digitalem und alanogem Signal für den Menschen nicht mehr höhrbar ist.

Da beim Arbeiten mit vielen Audiospuren mehrere Signale aufsummiert werden, verwenden DAWs in der Regel 32-bit Fließkommazahlen zu Darstellung der Samples um so Clipping zu vermeiden. Außerdem können aktuelle Prozessoren nativ mit 32-bit Variablen rechnen, sodass diese Wahl keine negativen Auswirkungen auf die Ausführungsgeschwindigkeit hat.

Technische Umsetzung

In digitalen Audio-Plugins wird das Signal auf Sample-Ebene kreiert bzw. bearbeitet. Dazu wird in jedem Plugin eine *process*-Funktion aufgerufen, die das Eingangssignal als Parameter erhält und das Ausgangssignal zurückgibt. Damit diese Funktionen nicht einmal pro Sample aufgerufen werden, werden Audio-Buffer in Form eines Arrays benutzt. Dies hat den Grund, dass jeder Funktionsaufruf und damit verbundene Kontextwechsel Mehraufwand (Overhead) für den Prozessor bedeutet. Die process-Funktionen können dann in einer Schleife alle Samples bearbeiten und das gesamte Array zurückgeben. Je größer das Array ist, desto weniger Aufrufe werden gemacht. Der Nachteil dabei ist, dass der Audioausgang des Computers erst Informationen erhält, wenn der gesamte Buffer zurückgeben wurde. Dies ist für Echtzeit-Anwendungen wichtig, da Verzögerungen über 20 Millisekunden von Menschen deutlich wahrgenommen werden. Entsprechend wird versucht eine Balance zwischen Buffergröße und Performance-Einbußen zu erreichen.

Meistens werden als Größe Zweierpotenzen verwendet, da Computer das Binärsystem nutzen und noch weniger Overhead entsteht. Außerdem verwenden viele Algorithmen das Divide-and-conquer-Prinzip und können so die Arrays beliebig oft in zwei gleich große Teile spalten.

Ein Sample besteht desweiteren effektiv aus zwei oder mehr 32-bit Fließkommazahlen (ein Wert pro Kanal), sodass ein typischer Buffer den folgen Aufbau besitzt.

Buffer = [s_{l0}, s_{r0}, s_{l1}, s_{r1}, s_{l2}, s_{r2}, …]

Obwohl zur Entwicklung theoretisch jede Programmiersprache verwendet werden kann, sind Audio-Plugins aus Performance-Gründen oft in systemnahen Sprachen, wie z.B. C oder C++ geschrieben.

Die Funktionen des User-Interface werden generell in einem separaten Thread ausgeführt, da sie nur einige Male pro Sekunde aufgerufen werden müssen, um eine flüssige Darstellung zu gewährleisten. Dies ist essentiell, da das Zeitfenster, das der process-Funktionen für die Berechnung der Daten zur Verfügung steht sehr gering ist. Nur das absolut wichtigste wird dort ausgeführt.

Limitierungen bei der Programmierung

Um die Echtzeitanforderung zu erfüllen muss möglichst genau vorhersehbar sein, wie lange die Berechnung eines Audio-Plugins benötigt. Um dies zu gewährleisten wird auf folgende Dinge geachtet, um unvorhersehbare Berechnungszeit zu minimieren:

Vermeidung von IO-Funktionen

Jegliche Kommunikation außerhalb der CPU und des RAMs wird vermieden bzw. in einen Thread ausgelagert. Andernfalls ergeben sich undeterministische Bereiche im Algorithmus, was dazu führen kann, dass eine Berechnung zu lange dauert und der Audio-Buffer verloren geht. Hierzu gehören Zugriffe auf Festspeicher und Netzwerke.

Vermeidung von Mutex-Locks

Zur Kontrolle der gleichzeitigen Ausführung mehrerer Programmteile, z.B. dem UI-Thread, ist es notwendig den lesenden und vor allem schreibenden Zugriff auf gemeinsam genutzte Teile zu kontrollieren. Ansonsten entsteht eine Wettlaufsituation (Race Condition) und die Korrektheit der Daten ist nicht gewährleistet. Eine klassische Lösung für dieses Problem sind Mutex-Locks, deren Nahme sich aus *mutually-exclusive,* also „gegenseitig ausgeschlossen“ herleitet. Bei ihrer Verwendung bekommt ein Programmteil bei Bedarf das exlusive Recht auf den Zugriff der Daten. Falls ein anderer Teil diese gleichzeitig nutzen will, muss er solange warten, bis sie wieder freigegeben wurden. Da sich allerdings nicht verhersagen lässt, wie lange gewartet werden muss, sind Mutex-

Locks für die verwendung in Audio-Plugins ungeignet. Stattdessen bieten *atomic types* eine Alternative. Sie garantieren den Zugriff auf entsprechend implementierte Variablen ohne Race Conditions und meistens trotzdem in einem einzigen Taktzyklus des Prozessors. Der einzige Nachteil dabei ist, dass ein Prozess potentiell nicht den aktuellsten Wert einer Variable erhält, falls die jeweiligen Lese- und Schreibzugriffe gleichzeitig stattfinden. Dies ist aber für Audio-Plugins unerheblich, da die Daten beim nächsten Zugriff zur verfügung stehen. Bei einer Verwendung eines 512-samples großen Buffers bei 44.1kHz entspricht das einer Verzögerung von 11,6ms.

Nicht-Blockierender Code

Generell wird darauf geachtet, dass während der Sample-Berechnung jeder Funktionsaufruf sofort einen Wert zurückgibt. Diese Kriterium ist wichtiger, als alle anderen Faktoren, wie die Genauigkeit oder Aktualität der Daten. Eine weitere Möglichkeit zur Kommunikation mit anderen Programmteilen ist hierfür eine Nachrichten-Warteschlange (message-queue). Anstatt Funktionen direkt aufzurufen oder Variablen direkt zu ändern, werden Nachrichten in die Warteschlange eingefügt, die dann vom jeweils anderen Prozess ausgeführt wird. Dadurch ist ein vorhersehbarer Zugriff auf das System gewährleistet.

Praxisbeispiel

Um die konkrete Entwicklung eines Audio-Plugins zu veranschaulichen, wird ein minimales Beispiel implementiert. Hierfür wird das VST-Format verwendet, da es weit verbreitet ist und eine gute Dokumentation besitzt. Um nur die wichtigsten Aspekte zu beleuchten, wird das Beispiel „Again“ aus den VST-SDK Beispielen verändert und nur der Teil der process-Methode betrachtet, der die eigentlichen Samples bearbeitet. Das Ziel ist es, die Funktion des Plugins von einer einfachen Lautstärkekontrolle auf eine kontrolliere Verzerrung zu erweitern. Der folgende Code-Ausschnitt zeigt den unveränderten Teil aus dem Again-Projekt, in dem die Berechnung stattfindet. Nachdem in der for-Schleife die aktuelle Position der Samples im Buffer in *ptrIn* bzw *ptrOut* gespeichert wird, multipliziert die Zeile

```
tmp = (*ptrIn++) * gain;
```

den Gain-Wert mit dem Eingangswert.

```
template <typename SampleType>
SampleType AGain::processAudio (SampleType** in, SampleType** out, int32 numChannels, int32
sampleFrames, float gain)
{
	SampleType vuPPM = 0;

	// in real Plug-in it would be better to do dezippering to avoid jump (click) in gain
value
	for (int32 i = 0; i < numChannels; i++)
	{
		int32 samples = sampleFrames;
		SampleType* ptrIn = (SampleType*)in[i];
		SampleType* ptrOut = (SampleType*)out[i];
		SampleType tmp;
		while (--samples >= 0)
		{
			// apply gain
			tmp = (*ptrIn++) * gain;
			(*ptrOut++) = tmp;

			// check only positive values
			if (tmp > vuPPM)
			{
				vuPPM = tmp;
			}
		}
	}
	return vuPPM;
}
```

Um die Verzerrungsfunktion (Clip-Distortion) zu realisieren, wird der Eingangswert zusätzlich mit 10 multipliziert und anschließend, falls das Ergebnis größer ist, auf den maximalen Wert 1 limitiert.

```
tmp *= 10;
	if (tmp > 1) {
		tmp = 1;
	}
```

Dies veranschaulicht den direkten Zugriff auf jedes einzelne Sample des Audio-Signals.

Quellen

[1] https://en.wikipedia.org/wiki/Audio_plug-in#List_of_plug-in_architectures

[2] Lessons Learned from a Decade of Audio Programming
(https://www.youtube.com/watch?v=Vjm--AqG04Y)

[3] VST SDK (https://www.steinberg.net/en/company/developers.html)