LEHRSTUHL FÜR MULTIMEDIAKOMMUNIKATION UND SIGNALVERARBEITUNG

UNIVERSITÄT ERLANGEN-NÜRNBERG

Entwicklung eines Steinberg Virtual Studio Technology (VST) Plugins als nativen Echtzeitdemonstrator zur Klangsynthese

Bachelorarbeit

von

Daniel Maaß

Hochschullehrer:
Priv. Doz. Dr.-Ing. habil. Rudolf Rabenstein

Betreuer:
Dipl.-Ing. Stefan Petrausch

Erlangen, 18. März 2005

hier kommt die aufgbenstellung hin!

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 18. März 2005

Daniel Maaß
Hofmannstr. 29
91052 Erlangen

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Zusammenfassung

1.1 Was versteht man unter VST

Abbildung 1.1: VST Logo

VST ist als Akronym für Virtual Studio Technologie zu verstehen und dient als Schnittstelle für virtuelle Instrumente und Effektgeräte. Durch die VST wird eine reale und echtzeitfähige Studioumgebung im Rechner nachgeahmt. Diese Schnittstelle stellte die Firma Steinberg bereit und ermöglicht so mithilfe des VST SDK's die Pluginimplementierung für Drittentwickler. Eine relativ umfangreiche C++ Bibliothek kann bei Steinberg heruntergeladen werden. Die erste Version der VST Schnittstelle wurde 1996 veröffentlicht und revolutionierte den damaligen Stand von Musikproduktionssoftware. 1999 erweiterte man schließlich die Schnittstelle auf die Version 2.0. So war es nun unter anderem auch möglich MIDI(Musical Instruments Digital Interface) [1] Signale zu empfangen und zu verarbeiten. Die aktuelle Version 2.3 habe ich im Zuge dieser Arbeit zur Implementierung genutzt.
VST beruht auf dem Prinzip von Host und Plugin. Einige Hostanwendungen sind zum Beispiel Cubase VST, Logic und Fruity Loops. Leider unterscheiden sich manche in der Hostseitigen

[1] Standard-Schnittstelle für Noten- und Parameter-Signale, die z.b. von Synthesizern in Töne gewandelt werden oder von Eingabegeräten (Klaviaturen, Dreh-/Schiebe-/Fußregler, etc.) erzeugt werden.

Implementierung. So stellen manche Hostanwendungen nicht alle Funktionen bereit oder haben eine unterschiedliche Aufrufreihenfolge der Plugin - Funktionen.

1.2 Physical Modeling

Physical Modeling ist eine elektronische Methode der Klangerzeugung basierend auf der Simulation von schwingenden Körpern. Diese Körper werden nach ihren physikalischen Eigenschaften analysiert und mathematisch beschrieben. Das entstehende mathematische Modell beinhaltet alle physikalischen Parameter des simulierten Musikinstruments, wie zum Beispiel die Länge, den Querschnitt oder die Dichte bei Saiteninstrumenten oder auch den Rohrdurchmesser eines Blasinstrumentes. Darin besteht ein wesentlicher Vorteil gegenüber anderen Syntheseverfahren, in denen auf vorher aufgenommene Samples oder Wavetabels zurückgegriffen werden muss. Dem Musiker steht damit ein erheblich größerer Parameterraum zur Verfügung. Er kann zum einen Klänge erzeugen, die realen Instrumenten sehr nahe kommen, zum anderen kann er sogar virtuelle Musikinstrumente erschaffen, die in der Realität nur sehr schwer zu bauen wären. So kann er die Saite einer Gitarre unrealistisch lang oder mit einem sehr großen Querschnitt versehen und so einen völlig neuen Klang erzeugen. Außerdem wird kein Speicherplatz für Samples oder Ähnliches benötigt, da der Klang nur auf Basis der eingestellten Parameter errechnet wird. Die physikalische Modellierung benötigt jedoch einen hohen Rechenaufwand, da partielle Differentialgleichungen zu lösen sind. Durch die rasch ansteigende Rechenleistung in den letzten Jahren ist es heutzutage aber schon möglich solche Syntheseverfahren auf gebräuchlichen Personal Computern in Echtzeit durchzuführen.

1.3 Motivation

Im Zuge dieser Bachelorarbeit möchte ich diese beiden Themen zusammenbringen. Es soll ein Programm entstehen, welches einen Physical modeling Algorithmus in die Welt der VSTs transferiert. Somit soll dieses Plugin mit allen gängigen Musikproduktionsprogrammen kompatibel sein und die Vielfalt der Möglichkeiten für Musiker ein kleines Stück erweitern. Da der Algorithmus, der am Lehrstuhl für Multimediakommunikation und Signalverarbeitung der Universität Erlangen Nürnberg entwickelt wurde, das Schwingungsverhalten einer Saite simuliert ist der Musiker damit in der Lage, alle nur erdenklichen Klänge einer Saite zu kreieren. Gleichzeitig sollen alle Eigenschaften der Saite in Echtzeit verändert werden können, um so auch bei einem Liveeinsatz des Plugins eine möglichst hohe Klangvielfalt zu erreichen.

Kapitel 2

VST Schnittstellenprogrammierung

2.1 VST Plugin

Wie in Abschnitt 1.1 bereits erwähnt, hat sich das VST SDK in den letzten Jahren weiterentwickelt.
Abbildung 2.1 zeigt das Zusammenspiel der wichtigsten Klassen und Dateien unterschiedlicher Versionen des VST SDK's.

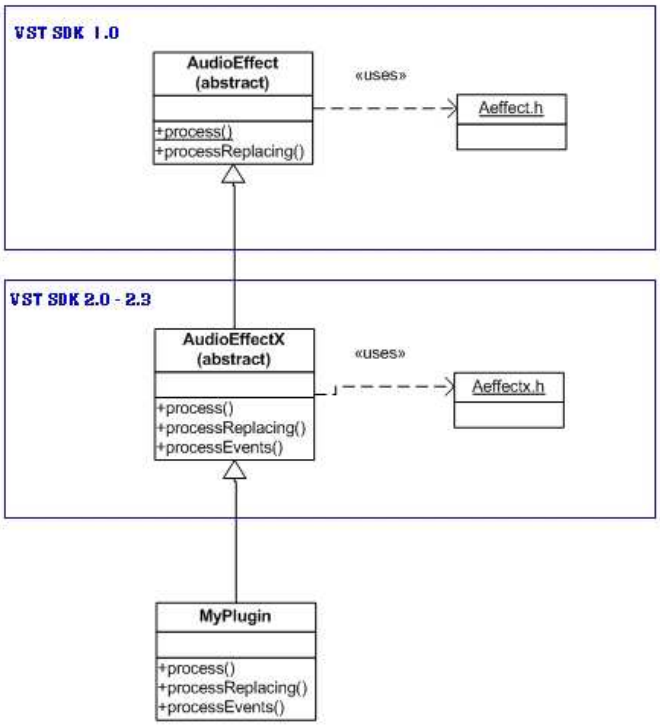

Abbildung 2.1: SDK Versionsaufbau

Ein VST Plugin ist eine ausführbare Datei die von einer Hostanwendung geladen wird. Aufgrund der Plattformunabhängigkeit gibt es natürlich unterschiedliche Dateitypen.
Tabelle 2.1 listet für verschiedene Betriebssysteme die entsprechenden Dateitypen auf.

PLATTFORM	EXECUTABLE
Windows	.dll
MAC	Raw Code Recource
BeOS	Library
SGI/Unix	Library

Tabelle 2.1: Plugin Dateitypen

Man unterscheidet grundsätzlich zwischen Effektplugins und Instrumentenplugins (VSTi), wobei ein Effekt schon vorhandene Audiodaten verändert, ein Instrument neue Audiodaten generiert. Instrumente müssen daher bei einer möglichen Kopplung am Anfang der entstehenden Kette stehen. Eine beliebige Anzahl von Effekten kann dem nachgeschaltet werden. Mit der VST Version 1.0 konnten ursprünglich nur Effekte implementiert werden. Später als auch Midi Signale empfangen werden konnten, war die Möglichkeit geben auch Instrumente zu erstellen. Der Klang wird dann abhängig von den empfangen Noten-, Regler- und Steuerdaten erzeugt.
Für die Audioverarbeitung kann ein Plugin zwei Funktionen zur Verfügung stellen. *process(float **inputs, float **outputs, long sampleFrames)* und *processReplacing(float **inputs, float **outputs, long sampleFrames)*. Während die Erster unterstützt werden muss, ist die Zweite optional. Es ist also nicht zwingend notwendig beide Funktionen zu implementieren, wobei Steinberg dies ausdrücklich empfiehlt. Der Unterschied dieser Funktionen besteht darin, dass die *process(...)* Funktion seine Audiodaten auf den Ausgabepuffer akkumuliert und die *processReplacing(...)* Funktion den Ausgabepuffer überschreibt. Daher ist *process(...)* effizienter, wenn mehrere Prozessoren auf einem Ausgang arbeiten (aux-send bus) und *processReplacing(...)* ist effizienter bei verketteten Verbindungen (inserts). Der Ein- und Ausgabepuffer in Form von Funktionsparametern *float **inputs* und *float **outputs* , sowie die Anzahl der zu bearbeitenden Samples werden von der Hostanwendung übergeben. Der Wertebereich der Audiosignale ist float im Bereich von -1.0 bis 1.0.

2.1.1 Implementierungsdetails

2.1.1.1 Parameterhandling

Ein VST Plugin kann Parameter, Programme und Bänke bereitstellen. Parameter sind die einzelnen Veränderlichen des Plugins. Die so genannten Programme sind komplette Parametersätze, und Bänke beinhalten eine bestimmte Anzahl von Programmen. Beim Initialisieren des Plugins wird die Anzahl der Veränderlichen festgelegt. Es stehen eine Reihe von Funktionen zur Verfügung, die alle relevanten Daten der einzelnen Parameter zwischen Host und Plugin kommunizieren. Der Datentyp und Wertebereich von Parametern ist *Float [0.0, 1.0]*. Die interne Abbildung dieser Parameter bleibt dem Entwickler überlassen. So werden einige Parameter des FTM-String Plugins exponentiell auf den Wertebereich der physikalischen Parameter abgebildet. Dafür wurden die Funktionen *EXPparam2value* und *EXPvalue2param* implementiert. Sie

gewährleisten die bidirektionale Abbildung der VST-Parameter und der physikalischen Parameter.

Listing 2.1: Logarithmische Abbildung der Parameter

```
/*
    function:     EXPparam2value
    logarithmic mapping between VST Parameter to internal
       physical parameter
    VST-Parameter -> physical Parameter

    arguments:
        -     min         :     minimum border
        -     max         :     maximum border
        -     param     :     parametervalue
    output:
        -     double     :     internal physical value
*/
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
double VstXSynth::EXPparam2value(double min, double max, double
param)
{
    double logmin = log(min);
    double logmax = log(max);
    double logTemp = (param * (logmax -logmin)) + logmin;
    return exp(logTemp);
}

/*
    function:     EXPparam2value
    logarithmic mapping between VST Parameter to internal
       physical parameter
    physical Parameter -> VST-Parameter

    arguments:
        -     min         :     minimum border
        -     max         :     maximum border
        -     param     :     internal physical value
    output:
        -     double     :     paramtervalue
*/
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
double VstXSynth::EXPvalue2param(double min, double max, double
value)
{
    double logmin = log(min);
```

```
    double logmax = log(max);
    double logTemp = log(value);
    return (logTemp – logmin) / (logmax – logmin);
}
```

Tabelle 2.2 zeigt die einzelnen Funktionen sowie deren Beschreibung. Alle diese Funktionen werden im Plugin implementiert und vom Host aufgerufen. Weiterhin sei erwähnt, dass die Funktionen *getParameterDisplay* und *getParameterLabel* bei einer eigenen Implementierung einer GUI überflüssig sind.

FUNKTION	BESCHREIBUNG
getParameter (long index)	liefert den durch den index gekennzeichneten Parameterwert
setParameter (long index, float value)	Setzt den jeweiligen Paramterwert
getParameterDisplay (long index, char *text)	liefert einen spezifischen Wert des jeweiligen Parameters. (wird im FTM-String, ohne GUI, für die physikalischen Werte der Parameter genutzt)
getParameterLabel (long index, char *label)	liefert ein Label des jeweiligen Parameters. (Wird im FTM-String, ohne GUI, für die Maßeinheit genutzt)
getParameterName (long index, char *label)	liefert den Namen des jeweiligen Paramters.

Tabelle 2.2: Funktionen für das Parameterhandling

2.1.1.2 Voice Controlling

Das FTM String Plugin bietet die Möglichkeit mithilfe des Parameters *voices* eine maximale Anzahl von zugleich ertönenden Saitenschwingungen festzulegen. Der Benutzer hat damit eine weitere Möglichkeit sein Plugin auf die ihm zur Verfügung stehende Rechenleistung einzustellen. Die Realisierung dieser Funktion wurde durch einen Stealalgorithmus durchgeführt. Ein Array mit einer dynamischen Länge verwaltet die aktuell klingenden Saiten. So werden bei einem NoteOn Event Elemente in dieses Array eingefügt und bei einem NoteOff Event das entsprechende Element entfernt. Ist jedoch die maximale Anzahl erreicht und es erfolgt ein weiters NoteOn Event, wird die schon am längsten klingende Saite entfernt.

2.1.1.3 MIDI Verarbeitung

Um MIDI Events zu empfangen, muss das Plugin im Konstruktor die Funktion *isSynth()* aufrufen. Damit wird dem Host signalisiert, dass es sich bei diesem Plugin um ein VSTi handelt. Außerdem wird an einigen geeigneten Programmpunkten *wantEvents()* aufgerufen, um den Empfang von VstMidiEvents zu beginnen. Die eigentliche Verarbeitung der VstMidiEvents wird in der Funktion *processEvents(VstEvents* ev)* durchgeführt. Es werden allerdings nur NoteOn und NoteOff Events verarbeitet. Andere Events, wie zum Beispiel Program Change, Channel Pressure und Pitch Bend werden in dieser Implementierung ignoriert. Für detailierte Informationen

FUNKTION	BESCHREIBUNG
NotemasterInit(long Voices)	generiert und initialisiert das Array
NotemasterInsert(long voice)	fügt Elemente ein und führt gegebenenfalls Stealalgorithmus durch
NotemasterNoteOn(long voice, long velocity, long delta)	fügt Elemente ein, setzt NoteIsOn Flag und führt gegebenenfalls Stealalgorithmus durch
NotemasterNoteOff(long voice)	entfernt Elemente und löscht NoteIsOn Flag
NotemasterAllNoteOff()	leert das Array und löscht alle NoteIsOn Flags

Tabelle 2.3: Funktionen für das Voice Controlling

bezüglich des Midi Standarts empfehle ich [4] [6] [7] [8]. Listing 2.2 soll die Verarbeitung der Midi Daten näher dokumentieren.

Listing 2.2: Midi Verarbeitung mit processEvents(VstEvents* ev)

```
/*
    function:    processEvents
    handle the VstEvents (only NoteON and NoteOff Events are
       catched)

    arguments:
        -    Zahl : pointer to VstEvents
    output:
        -    1 (want more Events)
*/
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
long VstXSynth::processEvents (VstEvents* ev) {
    for (long i = 0; i < ev->numEvents; i++)
    {
        // we only look for VsiMidiEvents
        if ((ev->events[i])->type != kVstMidiType)
            continue;
        VstMidiEvent* event = (VstMidiEvent*)ev->events[i];
        char* midiData = event->midiData;
        long status = midiData[0] & 0xf0;        // ignoring
           channel
        // 0x80  Note-off  0x90  Note-on
        if (status == 0x90 || status == 0x80)    // we only
           look at notes
        {
            long note = midiData[1] & 0x7f;
            long velocity = midiData[2] & 0x7f;
```

```
            // some Extern controls use NoteOn with Velocity=0
               to send NoteOff
            if ((status == 0x80) || (velocity == 0))
            {
                NotemasterNoteOff(note);
            }
            else
            {
                NotemasterNoteOn(note, velocity, event->
                   deltaFrames);
            }
        }
        event++;
    }
    return 1;    // want more
}
```

2.1.1.4 Erregungsfunktion

Für die Erregungsfunktion des rekursiven Systems stehen im FTM String mehre Funktion zur Auswahl. Man kann die Saiten mit einer $Impuls$, $Rect$, $Sin^2(x)$ mit $x \in [0, \pi]$ oder $Sinc(x)$ mit $x \in [-\pi, \pi]$ Funktion erregen. Dabei sind Art und Länge dieser Erregungsfunktion Parameter des Plugins. Damit einer Übersteuerung vorgebeugt wird, sind alle diese Funktionen auf 1 energienormiert. So werden zum Beispiel die diskreten Werte der Sin^2 Funktion mit einem Faktor a multipliziert.

$$\sum_{k=0}^{N-1} a^2 * \sin^4(\frac{k\pi}{N}) = 1 \tag{2.1}$$

$$a = \sqrt{\frac{8}{3N}} \tag{2.2}$$

Listing 2.3: Initialisierung der Erregungsfunktion

```
/*
    function:     initInputFunction
    calculates the different Exitation functions

    arguments:
        -     index:     Which function should be used
                         Impulse Rect Sin^2 or Sinc
    output:
        -     none
*/
//-------------------------------------------------------

```

```
void VstXSynth::initInputFunction(int index)
{
  double fA;
  if(InputFunction) delete [] InputFunction;
  InputFunction = new double[InputFunctionLength];

  int InputFunctionIndex = 0;
  switch(index)
  {
    case kImpulse:
      InputFunction[InputFunctionIndex] = 1.0;
      InputFunctionIndex++;
      while(InputFunctionIndex < InputFunctionLength)
      {
        InputFunction[InputFunctionIndex] = 0.0;
        InputFunctionIndex++;
      }
      InputType = kImpulse;
      break;
    case kRect:
      for(InputFunctionIndex =0;
          InputFunctionIndex<InputFunctionLength;
          InputFunctionIndex++)
      {
        InputFunction[InputFunctionIndex] = sqrt(1.0/
           InputFunctionLength);
      }
      InputType = kRect;
      break;
    case kSin2:
      fA = sqrt(8.0 / (3.0 * InputFunctionLength));
      for(InputFunctionIndex =0;
          InputFunctionIndex<InputFunctionLength;
          InputFunctionIndex++)
      {
          InputFunction[InputFunctionIndex] =
          fA * sin(InputFunctionIndex * M_PI /
             InputFunctionLength)
             * sin(InputFunctionIndex * M_PI /
                InputFunctionLength);
      }
      InputType = kSin2;
      break;
    case kSinc:
      double temp;
```

```
      fA = 0.0;
      for(InputFunctionIndex =0;
          InputFunctionIndex<InputFunctionLength;
          InputFunctionIndex++)
      {
        temp = (double)InputFunctionIndex /
               (double)InputFunctionLength
               * 2.0 * M_PI - M_PI;
        if(temp == 0.0)
        {
          InputFunction[InputFunctionIndex] = 1.0;
        }
        else
        {
          InputFunction[InputFunctionIndex] = sin(temp)/temp;
        }
        fA += InputFunction[InputFunctionIndex]
              * InputFunction[InputFunctionIndex];
      }
      // normalize sinc
      fA = sqrt(fA);
      for(InputFunctionIndex =0;
          InputFunctionIndex<InputFunctionLength;
          InputFunctionIndex++)
      {
        InputFunction[InputFunctionIndex] /= fA;
      }
      InputType = kSinc;
      break;
    }
}
```

2.1.1.5 Die Saite

Der eigentliche Algorithmus, der die Schwingung der Saite beschreibt wurde in der Klasse *voice* gekapselt. Wird das Plugin instantiiert, werden auch 128 Instanzen der Klasse *voice* angelegt. Jede Instanz repräsentiert dabei genau eine der 128 Noten, die mit Hilfe von Midi darstellbar sind. Außerdem wird jede einzelne Instanz initialisiert, was bedeutet, dass die zugehörige Grundfrequenz der Saite errechnet wird. (z.B. 246,94Hz für die B3-Midi Note, d.h. Note “h“ der kleinen Oktave [8]) Daraufhin wird die Initialisierung des rekursiven Systems jeder Saite angestoßen. Das inkrementelle Bearbeiten des rekursiven Systems, Sample für Sample, wird in der *processReplacing()* Funktion der Pluginklasse durchgeführt. Diese Funktion ist das Herzstück der Audioverarbeitung (siehe Abschnitt 2.1). Hier werden alle errechneten Audiodaten der zur Zeit aktiven Stimmen(*voices*) akkumuliert. Außerdem werden auch die Audiodaten der Release-Phase akkumuliert. Die Release-Phase ist eine parametri-

sierte Zeitspanne nach dem NoteOff Event, in der die Amplitude der Audiodaten exponentiell auf -60dB gesenkt wird. Dieses Verfahren, das auch aus der Hüllkurvenjustierung gängiger Synthesiser bekannt ist, wirkt einem hohen Amplitudenunterschied zweier aufeinander folgender Samples entgegen. Dieser Amplitudenunterschied wäre als “Knacksen“ deutlich hörbar.

Werden die saitenrelevanten Parameter, die physikalischen Größen und die Dämpfungsparameter, bei der Wiedergabe des Plugins automatisiert oder live verändert, wird in jeder Stimmeninstanz das Flag *ParameterChanged* gesetzt. Dies bedeutet, dass alle Saiten neu initialisiert werden müssen. Eine gleichzeitige Neuberechnung aller 128 Saiten würde einen enormen Rechenaufwand mit sich bringen. Aufgrund dessen werden die Saiten erst unmittelbar vor einer neuen Erregung initialisiert.

2.2 GUI Editor

Da bei der Erstellung einer graphischen Bedienoberfläche meist betriebssystemspezifische Methoden verwendet werden, wurde die VST GUI Bibliothek erstellt. Dies ermöglicht es, plattformunabhängig zu bleiben, indem die zur Verfügung stehenden Funktionen für alle gängigen Betriebssysteme implementiert wurden.
Für ein VST Plugin ist es grundsätzlich nicht erforderlich eine GUI zu erstellen. Der Host übernimmt dies auf seine individuelle Art und Weise. Durch die verschiedenen Parameterattribute ist er in der Lage, ein Userinterface zu erstellen, welches dem Anwender alle Parameter justieren lässt. Abbildung 2.2 und 2.3 zeigen verschiedene Standard GUI's einiger Hosts.

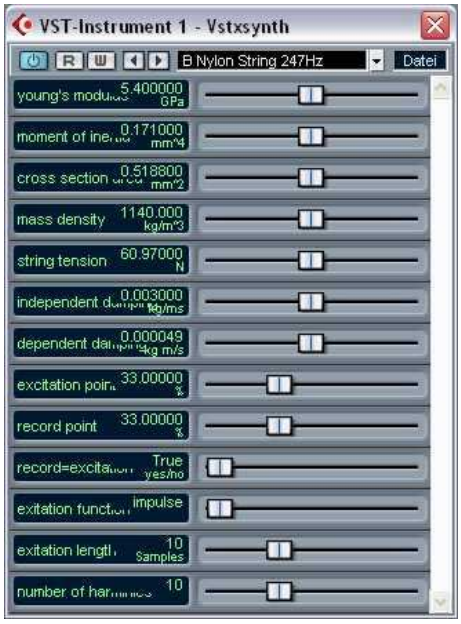

Abbildung 2.2: Standard GUI von Cubase SX

Abbildung 2.3: Standard GUI von Fruity Loops

Um dem Plugin eine eigene GUI zu verleihen, die nicht mehr vom Host beeinflusst wird, besteht die Möglichkeit mittels der VST GUI Bibliothek eine eigene Oberfläche zu schaffen. Es werden

verschiedene Schalter, Regler und andere Bedienelemente zur Verfügung gestellt, die man nach Belieben gestalten kann. Man kann diese Bibliothek natürlich auch erweitern, falls der gegebene Komfort der Bedienelemente nicht ausreichen sollte.
In Abbildung 2.4 wird das Beispielplugin der VST GUI Bibliothek gezeigt.

Abbildung 2.4: Beispiel einer GUI der VSTGUI Bibliothek

2.2.1 Implementierungsdetails

Um dem Projekt eine eigene GUI hinzuzufügen, implementiert man eine Editorklasse welche von *AEffGUIEditor* und *CControlListener* abgeleitet ist. Dem Konstruktor wird die Plugininstanz übergeben und im Konstruktor der Pluginklasse wird die Editorklasse instantiiert. Damit können nun Plugin und Editor bidirektional kommunizieren und die Schnittstelle zwischen Host und Plugin wird um einige Funktionen erweitert. Es ergibt sich eine wie in Abbildung 2.5 dargestellten Klassenhierarchie.

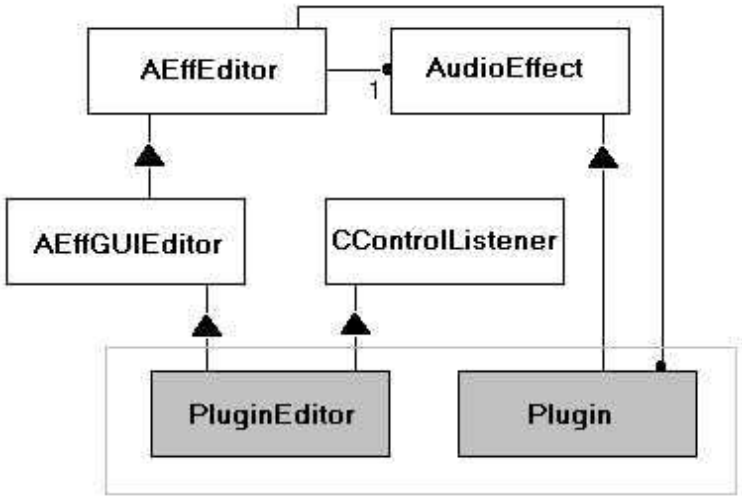

Abbildung 2.5: Klassenhierarchie mit Editorklasse

2.2.1.1 Bedienelemente und Windowhandling

Die Implementierung der GUI beruht auf einer Art des Model-View-Controller Konzepts [9]. Dabei wird das Programm in die drei Einheiten Datenmodell(Model), Präsentation(View) und Programmsteuerung(Controller) geteilt. Ziel des Modells ist ein flexibles Programmdesign um spätere Änderung oder Erweiterung einfach zu halten und die Wiederverwendbarkeit der einzelnen Komponenten zu ermöglichen. Außerdem ist aufgrund dieser Trennung die Visualisierung der Daten leicht auszutauschen. Der größte Teil der Bedienelemente der VST GUI Bibliothek vereinigt Controller und View in einer Klasse. Da sie einen Zustand besitzen und auch zur Steuerung eingesetzt werden.
Abbildung 2.6 zeigt die Klassenhierarchie der GUI Elemente.

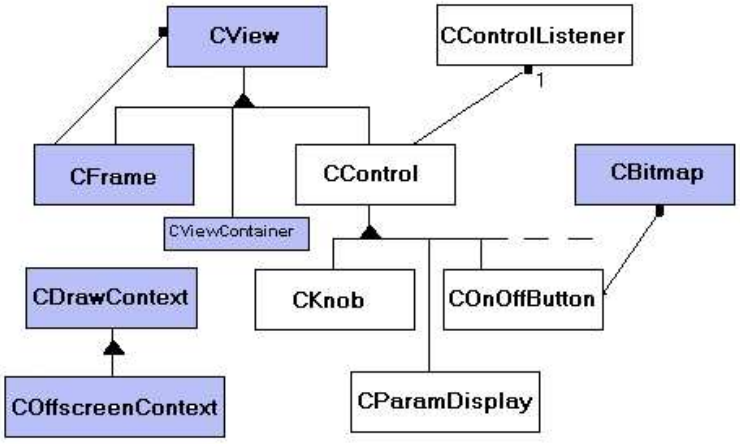

Abbildung 2.6: Klassenhierarchie der GUI - Elemente

In der Editorklasse wird eine Instanz von einem Frame angelegt, welche die Größe des Visuellen Plugins festlegt und einen Container für alle darzustellenden Objekte darstellt. Weiterhin können in diesen Frame beliebige Views hinzugefügt oder entfernt werden. Soll nun die grafische Oberfläche des Plugins angezeigt werden, ruft die Hostapplikation die *open()* Funktion des Editors auf. In dieser Funktion kann der Entwickler beliebige Views instantiieren und dem Frame mit *addView()* hinzufügen. Im Gegensatz dazu werden in der Funktion *close()* alle Views und der Frame freigegeben. Die individuelle Gestaltung der Bedienelemente beruht größtenteils auf dem Einbinden von Bitmaps.

Listing 2.4: Beispiel der open Funktion

```
/*
    function:    open
    instantiates all controls and views; is called from the
       host when the Plugin is visible

    arguments:
        -    void *ptr
    output:
        -    none
*/
//-------------------------------------------------------
long VstXSynthEditor::open (void *ptr)
{
    // always call this !!!  to postUpdate();
    AEffGUIEditor::open (ptr);

    // get version
    int version = getVstGuiVersion ();
    int verMaj = (version & 0xFF00) >> 16;
```

```
    int verMin = (version & 0x00FF);

    isOpen = true;          //set Flag

    // init the background bitmap
    CBitmap *background = new CBitmap (kBackgroundBitmap);

    //--CFrame-----------------------------------------------
    CRect size (0, 0, background->getWidth () + 100, background
       ->getHeight ());
    frame = new CFrame (size, ptr, this);
    frame->setBackground (background);
    background->forget ();
    CPoint point (0, 0);
    CBitmap *movieKnobBitmap = new CBitmap (kMovieKnobBitmap);

    //--CAnimKnobE-------------------------------------------
    size (0, 0, movieKnobBitmap->getWidth (), movieKnobBitmap->
       getHeight () / 61);
    size.offset (47, 25);
    point (0, 0);
    cAnimKnobE = new CAnimKnob (size, this, kEE, 61,
       movieKnobBitmap->getHeight () / 61, movieKnobBitmap,
       point);
    frame->addView (cAnimKnobE);
    movieKnobBitmap->forget();

    //--CParamDisplay----------------------------------------
    size (0, 0, 50, 15);
    size.offset (35, 55);
    cParamDisplayE = new CParamDisplay (size);
    if (cParamDisplayE)
    {
        cParamDisplayE->setFont (kNormalFontSmall);
        cParamDisplayE->setFontColor (kWhiteCColor);
        cParamDisplayE->setBackColor (kBlackCColor);
        frame->addView (cParamDisplayE);
    }
    //init the Knobvalue
    cAnimKnobE->setValue(effect->getParameter(kEE));
    //init ParamterDisplays
    effect->setParameter(kEE, effect->getParameter(kEE));
    return true;
}
```

Listing 2.5: Beispiel der close Funktion

```
/*
    function:    close
    free the Frame and all controlls and views
*/
//-------------------------------------------------------------------------------
void VstXSynthEditor::close()
{
    // remove the frame!
    if (frame) delete frame;
    frame = 0;
    isOpen = false;         //reset Flag
    cAnimKnobE = 0;
    cParamDisplayE = 0;
}
```

2.2.1.2 Parameterhandling

Wie in Tabelle 2.2 beschrieben, können bei einer Editorimplementierung die Funktionen *getParameterDisplay()* und *getParameterLabel()* nicht mehr benutzt werden. Da nun nicht mehr der Host sondern der Editor die Parameterveränderungen verwaltet. Das stringbasierende Interface wird durch die Funktion *setParameterAutomated()*, die vom Editor zum Host gerichtet ist, ersetzt. Der Host leitet die Parameteränderungen dann an die Pluginklasse weiter, welche ihrerseits eine Aktualisierung der Bedienelemente des Editors auslöst.
Abbildung 2.7 und 2.8 sollen die Änderungen des Konzepts näher erläutern und Listing 2.6 den Umgang mit *setParameterAutomated()* verdeutlichen.

Listing 2.6: Beispiel der valueChanged Funktion

```
/*
    function:    valueChanged
    handles the changes of the contols
    arguments:
        -    context:  Plattform specific drawing context
        -    control:  control which was changed
    output:
        -    none
*/
//-------------------------------------------------------------------------------
void VstXSynthEditor::valueChanged (CDrawContext* context,
   CControl* control)
{
    // called when something changes in the UI (mouse, key..)
    switch (control->getTag ())
    {
        case kEE:
```

```
            //the host will send setParameter after
               setParameterAutomated
            effect->setParameterAutomated (control->getTag (),
               control->getValue ());
            control->update (context);
            break;
        case k..
            .
            .
            .
    }
}
```

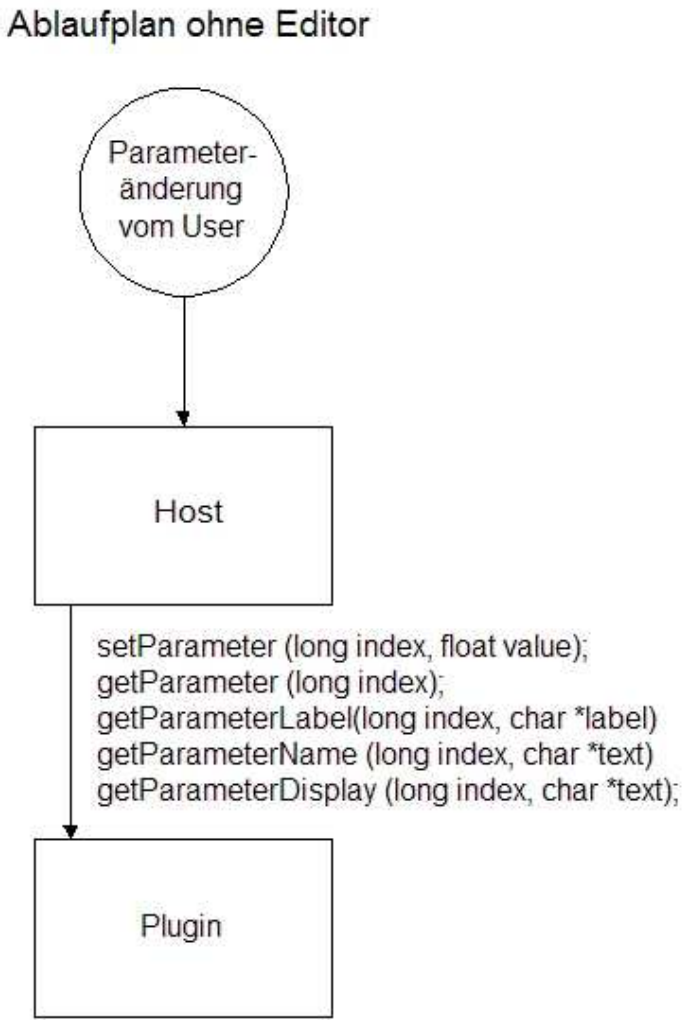

Abbildung 2.7: Ablaufplan ohne Editor

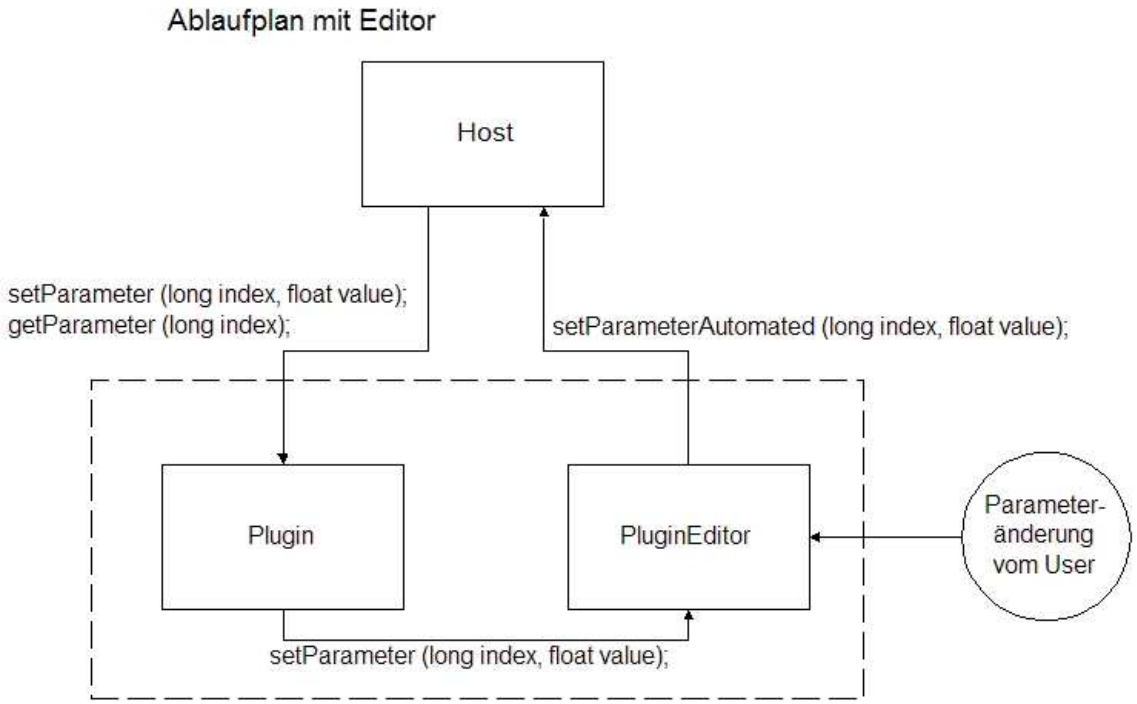

Abbildung 2.8: Ablaufplan mit Editor

2.2.2 Beschreibung der Oberfläche

Abbildung 2.9 zeigt die Oberfläche des Plugins und ihre Aufteilung. So wurde versucht gleichartige Parameter strukturiert anzuordnen.
Mithilfe der Parameter “maximale Oberwellen“, “maximale Stimmen“ und auch “Auskling-dauer“ sollte der Benutzer das Plugin auf die zur Verfügung stehende Rechenleistung einstellen können. Dadurch muss er gegebenenfalls aber klangliche Güte oder Melodiekomplexität einbüssen.
Durch die Schaltfläche “borders“ kann der Benutzer den Wertebereich der physikalischen Parameter beliebig einstellen. Es werden die oberen und unteren Grenzwerte angezeigt und können mithilfe der Pfeiltasten jeweils verdoppelt oder halbiert werden. Damit können sowohl beliebig große als auch beliebig kleine Wertebereiche für die Drehregler eingestellt werden.

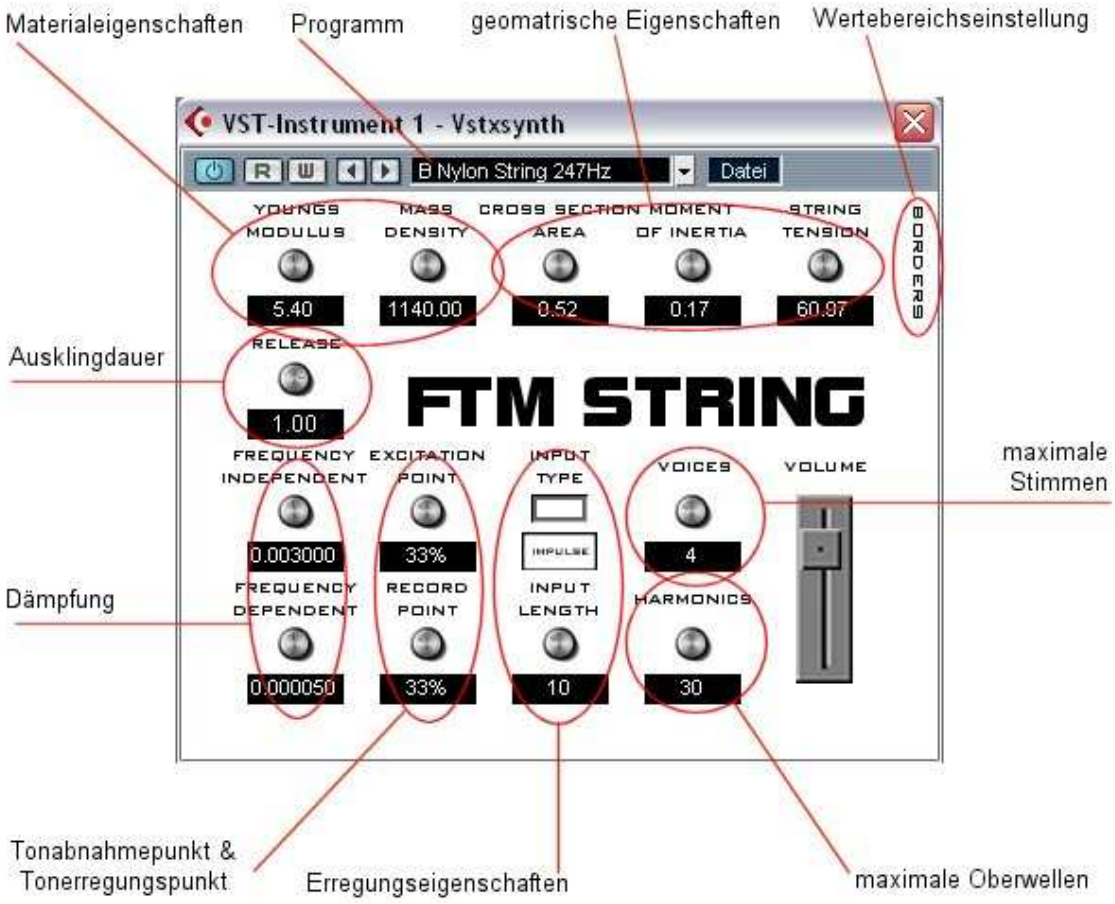

Abbildung 2.9: Oberfläche des FTM String Plugins

Kapitel 3

Physical Modeling mithilfe der FTM

Der zugrundeliegende Algorithmus, der am Lehrstuhl für Multimediakommunikation und Signalverarbeitung an der Friedrich-Alexander-Universität Erlangen-Nürnberg entwickelt wurde vollzieht die physikalische Modellierung mittels der Funktionaltransfomationsmethode (FTM). Diese Methode ist nicht so rechenintensiv wie andere Verfahren beispielsweise die Finite Differenzen Methode. Außerdem ermöglicht sie den direkten Zugriff auf die physikalischen Parameter der simulierten Objekte, was mit der Modalen Synthese und die Digitale Wellenleitermethode bei gleicher Rechenleistung nicht möglich ist [2].

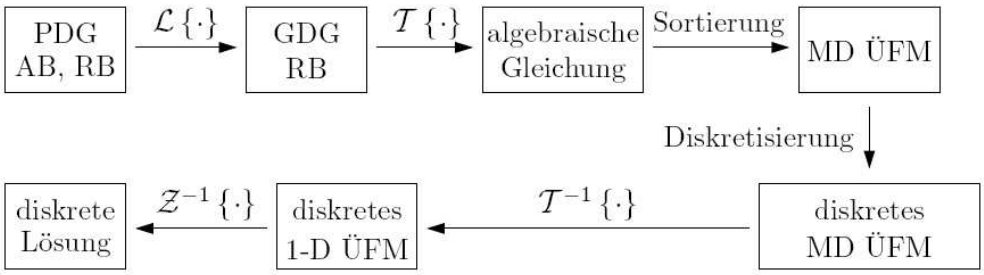

Abbildung 3.1: Lösungsweg zur Simulation mithilfe der FTM

In Abbildung 3.1 wird der Lösungsweg für die Simulation von schwingenden Objekten mittels der FTM gezeigt.
Dabei wird die zugrundeliegende partielle Differentialgleichung (PDG) bezüglich der Zeitvariable t in den Laplacebereich transformiert. Infolgedessen entsteht die gewöhnliche Differentialgleichung (GDG). Aus dem Randbedingungs (RB)- und Anfangswertproblem (AB) ergibt sich ein RB Problem, welches dann bezüglich der Ortsvariablen x mithilfe der Sturm-Liouville-Transformation in den örtlichen Frequenzbereich gebracht wird. Nach diesen zwei Transformationen kann das Problem in eine mehrdimensionale Übertragungsfunktion umgewandelt werden, welches eine ideale Form für eine Diskretisierung, und somit auch eine Implementierung darstellt. Um aber direkt auf einzelne Frequenzkomponenten zugreifen zu können, wird die mehrdimensionale diskrete Übertragungsfunktion bezüglich der vorher angewendeten Transformationen invers transformiert. Dadurch können nun auch psychoakustische Effekte ausgenutzt werden und somit die Rechenleistung minimiert werden.

Gleichung 3.1 wurde aus [3] Gleichungsnummer (39) entnommen. Sie führt durch die im folgenden dargestellten Operationen auf eine mathematische Darstellung des implementierten Algorithmus.

$$y(t) = \sum_{\mu=1}^{\infty} \frac{2}{l\rho A\omega_\mu} \sin(\frac{\mu\pi}{l}x)e^{-\sigma_\mu t} \sin(\omega_\mu t) * \sin(\frac{\mu\pi}{l}x_e)f_e(t) \tag{3.1}$$

$$y(x_b) = 0 \qquad y''(x_b) = 0 \qquad x_b \in \{0; l\}$$

$$f_e(x,t) = \gamma_0(x - x_e) \cdot f_e(t)$$

mit der Dämpfung σ_μ und der Frequenz ω_μ

$$\sigma_\mu = \frac{d_1}{2\rho A} + \frac{d_3}{2\rho A}(\frac{\mu\pi}{l})^2 \tag{3.2}$$

$$\omega_\mu = \text{sign}(\mu)\sqrt{\left(\frac{EI_b}{\rho A} - \left(\frac{d_3}{2\rho A}\right)^2\right)\left(\frac{\mu\pi}{l}\right)^4 + \left(\frac{T_s}{\rho A} - \frac{d_1 d_3}{2\left(\rho A\right)^2}\right)\left(\frac{\mu\pi}{l}\right)^2 - \left(\frac{d_1}{2\rho A}\right)^2}$$

Durch eine Diskretisierung $t = kT$ und $y(t) = \sum_{\mu=1}^{\infty} y_\mu(t)$ erhalten wir:

$$y_\mu^{(d)}(kT) = \frac{2}{l\rho A\omega_\mu} \sin(\frac{\mu\pi}{l}x)e^{-\sigma_\mu kT} \sin(\omega_\mu kT) * \sin(\frac{\mu\pi}{l}x_e)f_e(kT) \tag{3.3}$$

Nachfolgende Z-Transformation mit $e^{-\sigma_\mu kT}\sin(\omega_\mu kT) \circ\!\!-\!\!\bullet \frac{ze^{\sigma_\mu T}\sin(\omega_\mu T)}{z^2e^{2\sigma_\mu T} - 2ze^{\sigma_\mu T}\cos(\omega_\mu T)+1}$ ergibt:

$$Y_\mu(z) = \frac{2}{l\rho A\omega_\mu} \sin(\frac{\mu\pi}{l}x)\frac{ze^{\sigma_\mu T}\sin(\omega_\mu T)}{z^2e^{2\sigma_\mu T} - 2ze^{\sigma_\mu T}\cos(\omega_\mu T) + 1}F_e(z)\sin(\frac{\mu\pi}{l}x_e) \tag{3.4}$$

Im diskreten Zeitbereich lautet das Rekursive System wie folgt:

$$y_\mu[k] = y_\mu[k-1]2e^{-\sigma_\mu T}\cos(\omega_\mu T) + y_\mu[k-2]e^{-2\sigma_\mu T} + \tag{3.5}$$

$$+ f_e[k-1]e^{-\sigma_\mu T}\frac{2}{l\rho A\omega_\mu}\sin(\omega_\mu T)\sin(\frac{\mu\pi}{l}x)\sin(\frac{\mu\pi}{l}x_e)$$

Abbildung 3.2 zeigt das entstehende System.

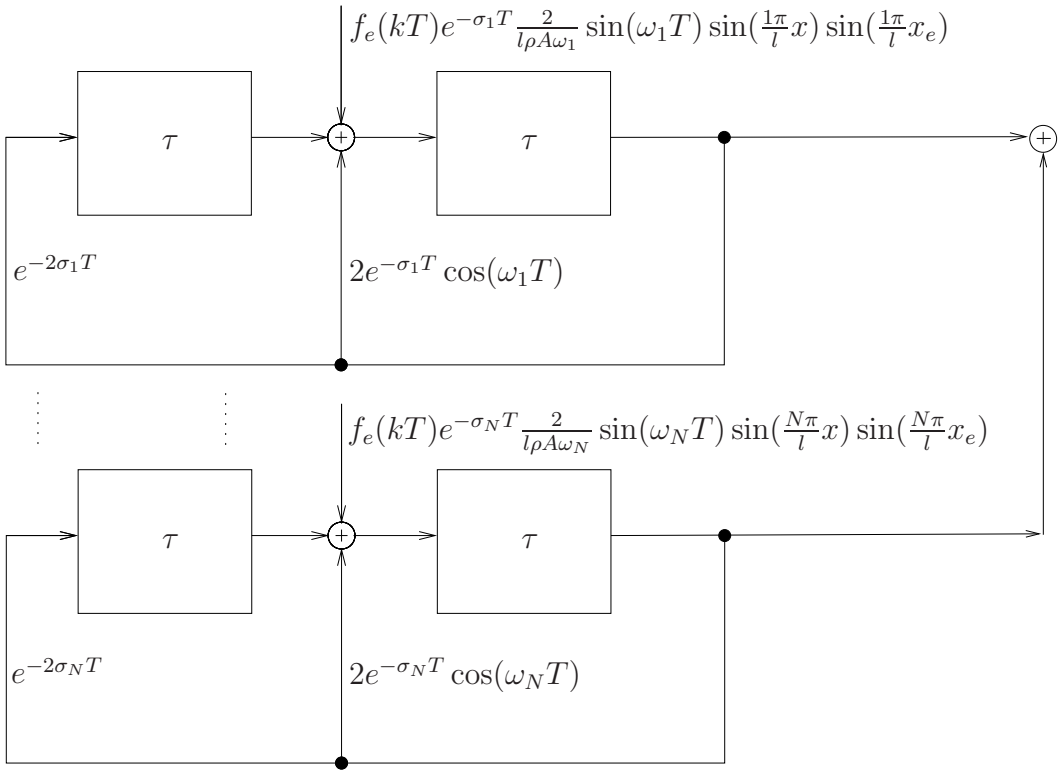

Abbildung 3.2: rekursives System

Kapitel 4

Anhang

4.1 Ausblick

Die VST Schnittstelle hat sich auf dem Markt der Produktionssoftware erfolgreich durchsetzten können. Das hat dazu geführt, dass inzwischen eine unüberschaubare Menge von Plugins entwickelt wurden. Da die Dokumentation der Schnittstelle doch etwas dürftig ausgefallen ist, bleibt dem Entwickler oft nur die spannende “try and error“ Methode übrig. Auch die unterschiedlichen Implementierungen der Hostapplikationen führen oft zu Verwirrungen. Nichts desto trotz ist es eine hervorragende Möglichkeit Klangerzeugende oder Klangverändernde Algorithmen der Allgemeinheit zugänglich zu machen. Außerdem möchte ich anmerken, dass der enorme Zusammenhalt und die Hilfsbereitschaft der VST-Entwickler in Form von Foren, Maillinglists (z.B vst-plugins@lists.steinberg.net) und privaten Webseiten bemerkenswert ist. Bei Problemen oder Fragen zum VST SDK bekommt man schnell professionelle Hilfe und gute Ratschläge.

4.1.1 Tool zur Gestaltung der Oberfläche

Eine professionelle Oberflächengestaltung erweist sich mit normalen grafischen Werkzeugen als sehr mühsam. Aufgrund dessen gibt es einige Tools im Web die sich auf diese Aufgabe spezialisiert haben.
Das Tool “RLH s Knob Render v3“ hat sich auf das Rendern von 3D Bitmaps für animierte Knöpfe *CAnimKnob* spezialisiert. Diese Art von Reglern arbeitet mit unterteilten Bitmaps. Das heißt, es wird je nach Parameterwert ein bestimmter Ausschnitt des Bildes gezeigt. Es ist als Action Script für Adobe Photoshop (V.7 und höher) realisiert und generiert als Vollversion 61 Bilder für einen Drehregler. Unter `http://www.otiumfx.com/knobrender.php` kann man sich eine Demoversion mit Dokumentation und Installationsanweisungen herunterladen. Die PRO-Version ist für 25 US-Dollar zu haben.

Abkürzungen

VST	Virtual Studio Technology
VSTi	VST Instrument
SDK	Software Devellopment Kit
MIDI	Musical Instruments Digital Interface
	Standard-Schnittstelle für Noten- und Parameter-Signale, die z.b. von Synthesizern in Töne gewandelt werden oder von Eingabegeräten (Klaviaturen, Dreh-/Schiebe-/Fußregler, etc.
dll	Dynamic Link Library
GUI	Graphical User Interface
FTM	Funktional Transfomation Method
PDG	partielle Differentialgleichung
GDG	gewöhnliche Differentialgleichung
AB	Anfangsbedingung
RB	Randbedingung
MVC	Model View Controller

Literaturverzeichnis

[1] *Lutz Trautmann.* Digital Sound Synthesis by Physical Modeling of Musical Instruments using Functional Transformation Models. Doktor Arbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Multimediakommunikation und Signalverarbeitung ,2002.

[2] *R. Rabenstein, L.Trautmann, S. Petrausch.* Digitale Klangsynthese durch physikalische Modellierung. ,Mar. 2003

[3] *Stefan Petrausch.* FTM Solution for a Dispersiv String with Frequency-Dependent and Frequency-Independent Damping. , Apr. 2004

[4] *Steinberg Media Technologies GmbH.* VST Plug-Ins SDK 2.3 Documentation `http://www.steinberg.de/steinberg/ygrabit/vstsdk/OnlineDoc/vstsdk2.3/index.html`, 2004

[5] *Steinberg Media Technologies GmbH.* VSTGUI Library Documentation V2.2. `http://www.steinberg.de/steinberg/ygrabit/vstgui/V2.2/doc/index.html`, 2004.

[6] Musical Information. Raw MIDI Data. `http://www.ccarh.org/courses/253/lab/cinmidi/`, 2004.

[7] Zem College - Institut für Elektronische Musik. `http://www.zem-college.de/`, 2004.

[8] *Hans Georg Britz, Franz Martin Löhle* Zem College - Institut für Elektronische Musik. *MIDI-Kompendium* `http://www.zem-college.de/midi/index.htm`, 2004.

[9] *MVC-Modell* `http://www.computerbase.de/lexikon/Model_View_Controller`, 2004.