Projekt FROCOR
Report

Daniel Grün und Fabian Stöhr

19. Januar 2005

Inhaltsverzeichnis

Teil I

Frocor

Kapitel 1

Planung

1.1 Name

Der Name FROCOR steht für **F**ROCOR **R**esonating **O**bjects **COR**relator.

1.2 Idee

Das Projekt FROCOR soll die Programme fms und ocor zu einer Klang- und Bildinstallation verbinden.

Die FMS-API soll genutzt werden um Klänge zu erzeugen, bestehend aus verschiedenklangfarbigen Schwingungen in verschiedener, benutzergesteuert oder zufällig schwankender Frequenz, Lautstärke und Interferenz. Dabei beeinflussen die Eigenschaften des erzeugten Klanges das Aussehen der grafischen Darstellung geometrischer Formen:

```
Klangfarbe -> Form/Typ
    Sinuston - Kreis
    Dreiecksschwingung - Dreieck
    Rechtecksschwingung - Rechteck
    [...]
Lautstärke -> Grösse/Zoom
Frequenz -> Farbe
Interferenzfrequenz -> Drehgeschwindigkeit
```

Der Benutzer kann, z.B. durch Betätigen der Tastatur, auf o.g. Klangeigenschaften Einfluss nehmen und somit Klang wie Bild verändern. Die Kommunikation zwischen den in verschiedenen Programmiersprachen geschriebenen Programmelementen erfolgt über Unix-Domain-Sockets und kann somit auch auf tcp/ip uebertragen werden.

Da Einzelkomponenten des Projektes schon vor Projektbeginn bestanden und zur Erfüllung der Anforderungen von FROCOR nurmehr modifiziert und verbunden werden

mussten, bzw. weitere Einzelkomponenten entwickelt und verbunden werden mussten ist die Entwicklung von FROCOR ein *Bottom-Up*-Prozess.

1.3 Programmbestandteile

1.3.1 FMS-API und Backend – Daniel Grün

Die FMS-API (genauer FMPlayer-API) ist eine selbstgeschriebene Bibliothek zur Erzeugung von Klängen verschiedener Art. Nach etwa dreijähriger Entwicklungszeit sind die Routinen ausgereift genug, um FROCOR zur Klangerzeugung zu dienen.

Um on-the-fly-Variation von Soundeigenschaften und Mitteilung dieser Änderung an weitere Programme zu ermöglichen, muss die API um Funktionalitäten zur Echtzeit-Generierung von Sounds erweitert werden (siehe 2.3.3 auf Seite 18). Diese wurden in den Sommerferien geschrieben und wie es scheint funktionstüchtig, ausreichend schnell und stabil. Unter Umständen werden noch weitere Anpassungen von Nöten sein. So könnte die FMPlayer-API z.B. als dynamische Bibliothek realisiert werden.

Nachtrag 25.11.

Für FROCOR wird ein FMS-Backend-Programm über UDS im fms-Protokoll empfangene Anweisungen lesen und in Klangeigenschaften umsetzen. Dieses Programm soll Anfang Dezember grundlegende Operationen verstehen.

1.3.2 FROCOR-Player – Daniel Grün

Der FROCOR-Player erzeugt Klänge und gibt Informationen darüber mittels des FROCOR-Protokolls an Ocor weiter. Das Programm befindet sich noch in der Planungsphase.

Eine erste Version mit Konsolenzugriff und automatische Modulation der Klangeigenschaften soll bis Mitte November fertiggestellt werden. Spätere Versionen reagieren on-the-fly auf Benutzereingaben und verfügen über eine grafische Oberfläche.

Nachtrag 25.11.

FROCOR-Player muss außer dem FROCOR-Protokoll auch das Fms-Protokoll zur Kommunikation mit dem anderen Backend verstehen. Verschiedene Testversionen können dies bereits, eine multifunktionale Version soll im Monat Dezember abgeschlossen werden.

1.3.3 Ocor – Fabian Stöhr

Ocor besteht seit einem Jahr als Projekt zur Darstellung und Animation geometrischer Formen. In den Sommerferien wurde es von reinen Grafik-Funktionen befreit und aufgeteilt. Es liest im Moment schon Benutzereingaben über die Konsole ein und kommuniziert über UDS mittels des OCOR-Protokolls mit dem SDL-Client, der die geometrischen Formen dann in einem Grafik-Fenster darstellt.

Zur Zusammenarbeit mit dem FROCOR-Player muss Ocor zusätzlich als UDS-Client fungieren um die so empfangenen Anweisungen zur Erzeugung und Animation von Figuren in einfache Grafikbefehle umzuwandeln, die über das OCOR-Protokoll an den SDL-Client weitergegeben werden können.

Um das Programm zu modularisieren soll das alte Eingabe-Frontend in ein seperates Programm ausgelagert werden und ebenfalls über Unix Domain Sockets mit dem Ocor backend kommunizieren. Es ist geplant, die UDS-Fähigkeit bis November zu implementieren.

Im Zuge dieser Modularisierung soll auch die Struktur von Ocor verbessert werden, z.B. durch Aufteilen in Pascal Units und durch teilweise Verwendung von objektorientierter Programmierung. Beide Modifikationen haben keinen Einfluss auf den Funktionsumfang des Programms und werden in Abschnitten implementiert, evtl. auch über das Jahr 2004 hinaus. Jedoch werden spätestens Oktober die erste Schritte in Richtung Objekt-Pascal unternommen werden (das Programm benutzt bereits Records). Bis Ende Oktober soll ein Plan über den Einsatz von OOP erstellt werden, welcher z.B. klarstellen soll, wo der Einsatz sinnvoll ist.

Die benötigten Animationen (z.B. Zoomen) funktionieren bereits, jedoch müssen Funktionen zum Verändern der Farben implementiert werden (siehe 1.3.4, 1.5.2 auf Seite 9). Dies ist spätestens für Dezember geplant.

1.3.4 SDL-Client – Fabian Stöhr

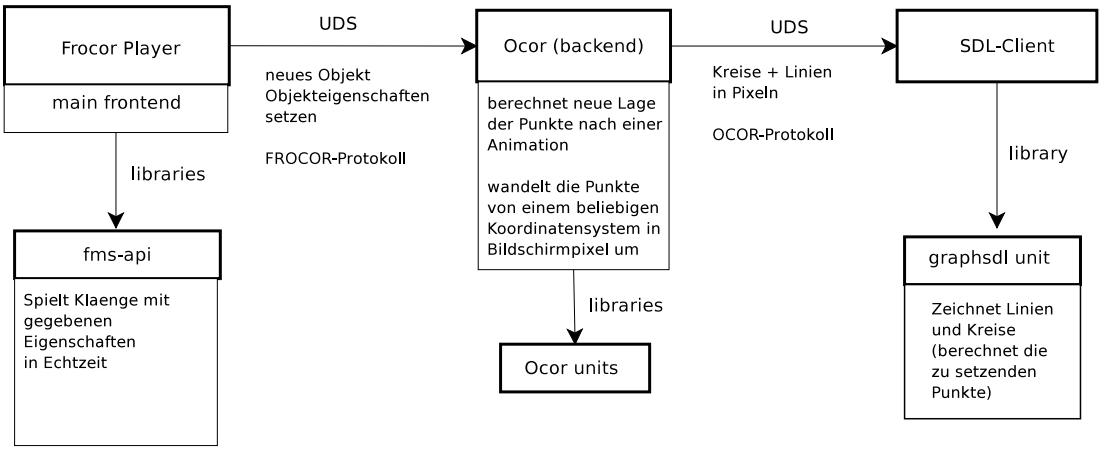
Der SDL-Client setzt einfache Grafik-Befehle (Punkte, Kreise, Linien) in ein Bild um, wobei er zur Berechnung der Bildpunkte selbstgeschriebene Funktionen benutzt. Seine Eingaben erhält er per UDS über das OCOR-Protokoll.

Die Farb-Auswahlmöglichkeit (siehe 1.3.3, 1.5.2 auf Seite 9) wird bis Dezember implementiert werden.

1.4 Struktur

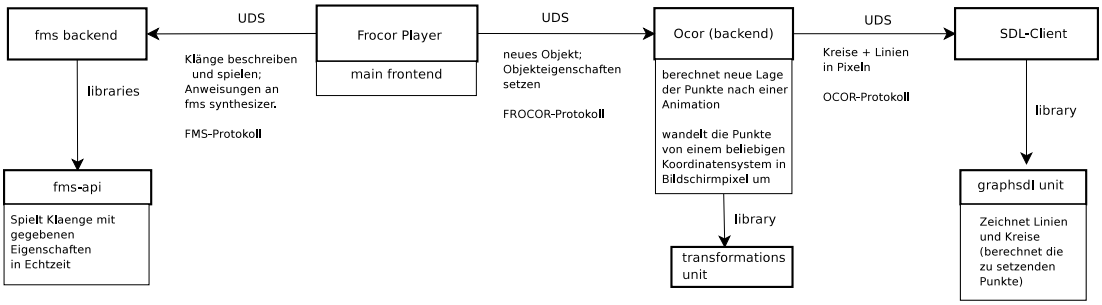
1.4.1 Ursprüngliche Planung

Hier ein Diagramm, welches die Arbeitsaufteilung und Kommunikation zwischen den einzelnen Programmbestandteilen visualisiert:



1.4.2 Version 2

In der heutigen Struktur FROCORs wurden die Aufgaben des frocor-players teilweise an ein fmsbackend abgegeben. Somit gibt frocor-player Anweisungen sowohl an dieses fmsbackend als auch an ocor, welche dann von den ensprechenden backends durchgeführt werden.



1.5 Protokolle

Für die Kommunikation mittels Unix-Domain-Sockets sind bereits in den Sommerferien funktionierende Beispielprogramme in C++, C und Pascal geschrieben worden.

Die benutzten Protokolle zur Beschreibung der grafischen Objekte und Klänge sind die High-Level-Protokolle (d.h. Protokolle mit starker Abstrahierung) zwischen Player und Ocor bzw. FMS und das Low-Level-OCOR-Protokoll (d.h. ein möglichst einfaches Protokoll) zwischen Ocor und SDL-Client. Eine Ausnahme bildet das Adapter-Protokoll zur Umleitung der Kommunikation auf TCP.

1.5.1 FROCOR-Protokoll

Das FROCOR-Protokoll dient zur Übermittlung der geometrischen Daten der Objekte von FROCOR-Player zu OCOR.

Nach dem Starten der Verbindung sendet der Server (FROCOR-Player) die Versionsnummer des Protokolls (aktuell „frocor-pre1“) . Sollte der Client nur niedrigere Versionen beherrschen, wird der Benutzer darauf hingewiesen, und hat die Wahl, fortzufahren oder abzubrechen.

Anschließend sendet der Server Anweisungen in der eigentlichen Ocor-Syntax an den Client.

Jede Anweisung dieser Syntax besteht aus 3 Teilen, die durch „:“(Doppelpunkte) voneinander abgegrenzt werden. Aus den Anweisungsgruppen 1 und 2 muss je genau eine Anweisung, aus Gruppe 3 eine beliebige Anzahl von Anweisungen ausgewählt werden.

Grundoperationen

n neues Graphik-Objekt erzeugen; es werden einige Initialisierungen benoetigt (Typ, Grad des Polygons).

o Veränderung eines existierenden Objektes

Nummer des Objektes

durchlaufende Nummerierung von 1 an

Wertezuweisungen

Werte werden absolut durch ‚=‘und relativ durch ‚+‘und ‚-‘zugewiesen.

t Typ des Objekts [char]

- **p** Polygon
- **l** Linie
- **c** Kreis

n Grad (nur Polygon) [unsigned int]

x X-Koordinaten der Punkte von x0 bis x(n-1) (siehe Beispiel) [real]

y Y-Koordinaten der Punkte von y0 bis y(n-1) (siehe Beispiel) [real]

r Radius (nur Kreis) [unsigned real]

c Farbe [unsigned int]

z Zoomveränderung [unsigned real]

a Winkelgeschwindigkeit [real]

Beispiele

```
n:1:t=p:n=3:x1=-1:y1=2:x2=2:y2=3:c=30000:a=5
```

Erzeugt ein neues Dreieck mit gegebenen Koordinaten und Farbwert, das sich pro Frame um 5 Grad dreht. Die nicht angegebenen Koordinaten werden auf (0,0) gesetzt bzw. zufällig erzeugt.

```
o:2:r+1:x1-7
```

Erhöht den Radius des vorher schon erzeugten Kreises mit Objektnummer 2 um 1 und verschiebt den Mittelpunkt um 7 nach links.

1.5.2 OCOR-Protokoll

Version 0.1

Das Protokoll, das zur Kommunikation zwischen dem Ocor-Backend und dessen SDL-Client genutzt wird ist sehr simpel gehalten. Seine Aufgaben beschränken sich im Wesentlichen auf das Zeichnen von Linien und Kreisen.

Um die Objekte in verschiedenen Farben zu zeichnen (siehe 1.3.3 auf Seite 6 , 1.3.4 auf Seite 6) muss das Protokoll erweitert werden.

Linien werden durch ein ‚l'gefolgt von einem ‚:'und den durch Kommas getrennten Koordinaten der Linie gezeichnet, also:

```
l:x1,y1,x2,y2
l:71,46,97,45
```

Kreise werden durch ein ‚c'gefolgt von einem ‚:', den Koordinaten des Mittelpunktes und dem Radius (letztere auch durch Kommas getrennt):

```
c:x1,y1,r
c:63,417,38
```

update Durch das Kommando „update"werden die Linien und Kreise gezeichnet und anschließend gelöscht (so das sie beim nächsten Mal nicht wieder mitgezeichnet werden).

quit Wird gesendet, wenn sich der Ocor-Server beendet. Nachdem der Client das Kommando empfängt terminiert er ebenfalls.

Version 0.2

Version 0.2 fügt dem OCOR-Protokoll eine Möglichkeit zum Festlegen der Farben hinzu. Dabei kann am Ende einer der OCOR-Anweisungen ‚c:'und ‚l:'optional ein ‚:'gefolgt von

dem Wert der Farbe angehängt werden.

```
l:x1,y1,x2,y2:color
c:63,417,38:28433
```

Mittels der Kommandos `lock` und `unlock` kann das Wiederauffrischen nach jeder Änderung vorübergehen unterbunden werden, was Sinn macht um bei äufwändigen Grafiken eine flüssige Animation zu gewährleisten.

1.5.3 FMS-Protokoll

Mittels des FMS-Protokolls empfängt das FMS-Backend Anweisungen zur Konfiguration und Wiedergabe der gewünschten Klangstrukturen. Das Protokoll ist syntaktisch am FROCOR-Protokoll (siehe 1.5.1 auf Seite 8) angelehnt.

FMSound-Konfiguration

Alle Befehle zur Konfiguration eines FMSound-Objekts bestehen aus einer Zeile, beginnend mit dem Indikator „s “. Es folgen, getrennt durch ‚: ‘eine Folge von Grundoperation, Nummer und einer oder mehreren Wertezuweisungen.

Grundoperationen

n neues FMSound-Objekt erzeugen

o Veränderung eines existierenden Objektes

Nummer

Laufend durchnummerierter Index des FMSound-Objekts.

Wertezuweisungen

Werte werden absolut durch ‚=‘und relativ durch ‚+‘und ‚-‘zugewiesen.

p Pfad der FMS-Sounddatei [string]

f Frequenz [real]

v relative Lautstärke [real]

mi Pfad der FMS-Midi-Instrumentendatei für FMS-Midi-Modus [string]

mh Pfad der FMS-Midi-Hüllkurvendatei für FMS-Midi-Modus [string]

Mixer-Dauer

Mittels der Zeile „m:t=x “kann die Länge des FMMixer-Objekts auf x Sekunden gesetzt werden. Dies ist nur für die Bestimmung der Schlagzahl im Midi-Modus nötig.

1.5.4 TCP-to-UDS-Protokoll

Um verschiedene Teilprogramme von FROCOR auf getrennten Rechnern zu betreiben wurde behufs Umleitung der UDS-Kommunikation mittels TCP ein Protokoll zur Vermittlung zwischen Server und Client entworfen.

Ablauf (TCP-Portnummer 3216):

1. Server öffnet TCP-Socket und wartet auf Client
2. Client öffnet TCP-Socket und verbindet sich mit Server
3. Server verbindet sich als UDS-Client mit dem lokalen UDS-Socket
4. Client verbindet sich als UDS-Server mit dem lokalen UDS-Socket
5. Server sendet `protocol uds2tcp`
6. Client sendet `protocol tcp2uds`
7. Senden der Daten (unidirektional von Server zu Client)
8. UDS-Anwendung versendet `quit`
9. Server sendet `quit` und schließt UDS-Socket
10. Client schließt UDS-Socket
11. Client sendet `tcp2uds quit`
12. Server sendet `uds2tcp quit`
13. Client schließt TCP-Socket
14. Server schließt TCP-Socket

Aufgrund der imperativen Abfolge dieser Prozesse ist eine Strukturierung nach Zuständen nicht nötig.

1.6 Mögliche Probleme

Der Ansatz einer Entwicklung von drei verschiedenen Programmen durch zwei Entwickler in verschiedenen Programmiersprachen auf verschiedenen Linux-Distributionen könnte zu folgenden Problemen führen:

Kommunikation zwischen Programmen schlägt fehl (Protokoll wird nicht richtig implementiert, Inkompatibilitäten aufgrund der verschiedenen Programmiersprachen etc.)

zeitliche und funktionelle Abstimmung der Bestandteile aufeinander schlägt fehl (Projektmitarbeiter können Planung nicht einhalten)

Systemanforderungen können nicht erfüllt werden (Programmierumgebungen für die verschiedenen Programmiersprachen und Bibliotheken, Hardware- und Treiberanforderungen)

...

Als Gegenmaßnahme werden subversion als Version-Controling-System, eine Mailingliste zur Kommunikation und freenx sowie ssh zur Ermöglichung von Testläufen auf dem jeweils anderen System eingesetzt.

13.01.2005: Daniel Grün, Fabian Stöhr,

Kapitel 2

Entwicklung

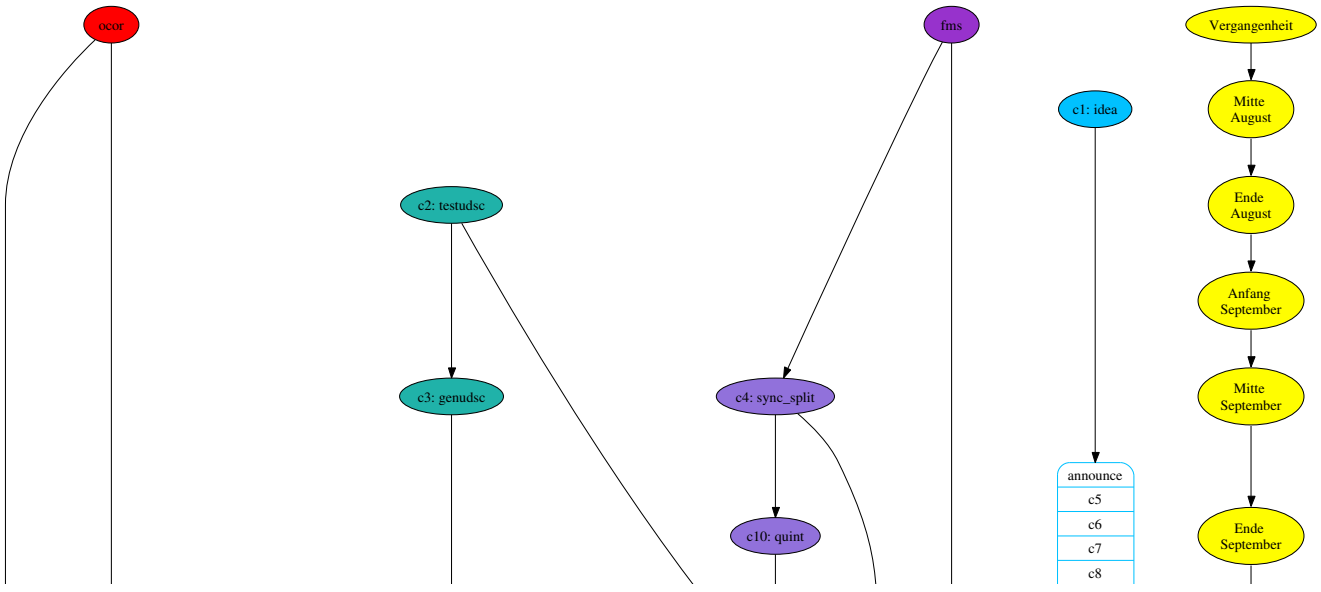
2.1 Zeitbedarf

Die Entwicklung von Frocor erfolgte in über 150 dokumentierten Schritten. Bei einem geschätzten Zeitbedarf von einer Stunde pro Entwicklungsschritt war der Zeitaufwand erheblich und konnte nur durch Nutzung der Herbsferien und sonstiger Freizeit bewältigt werden.

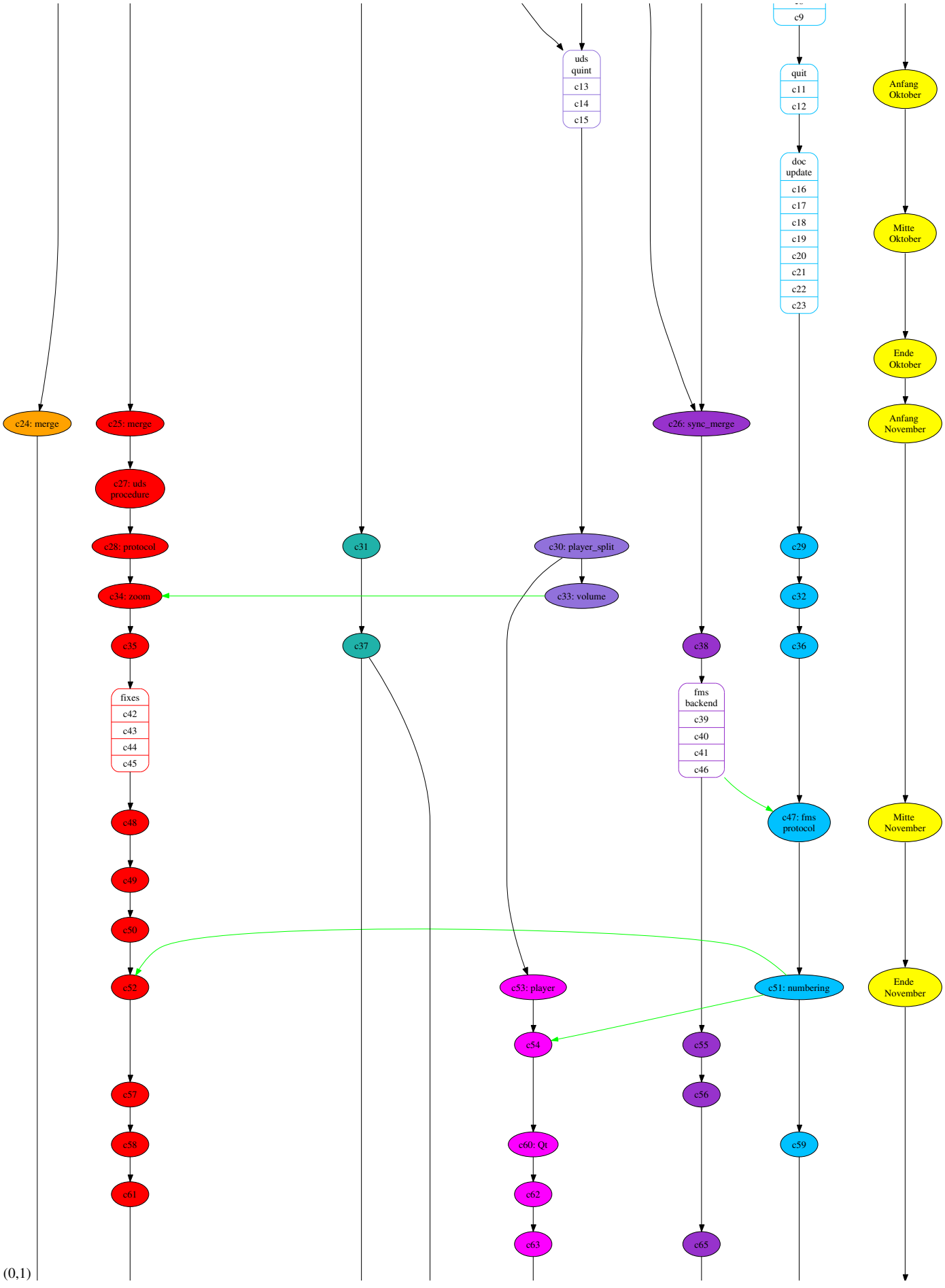
2.2 Commit-Diagramm

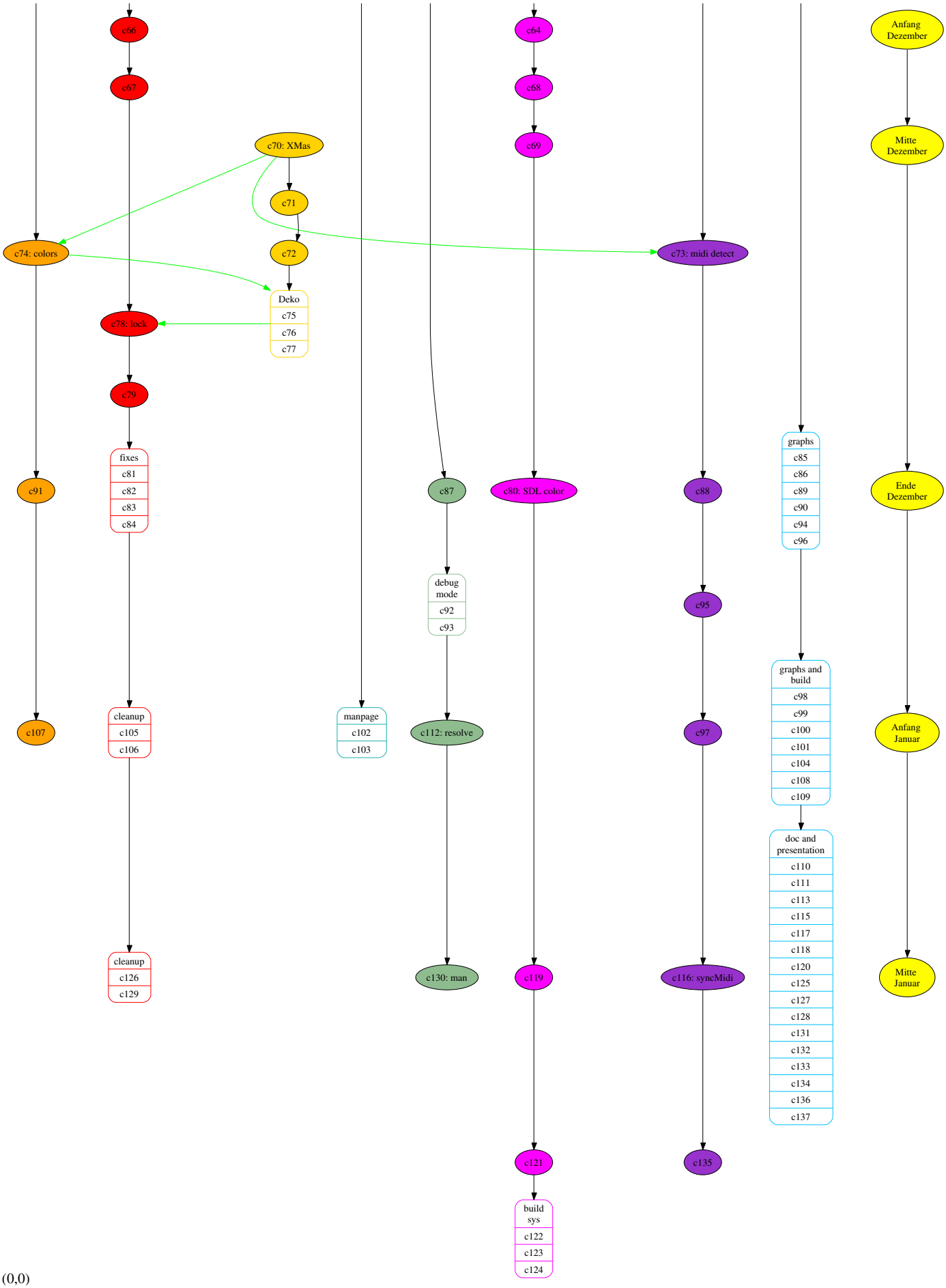
Die Entwicklung von Frocor erfolgte in verscheidenen, weitestgehend voneinander unabhängigen so genannten *Branches*. Mittels eines Version-Control-Systems (svn) wurde die Zusammenarbeit der beiden Programmierer erleichtert. Jede ans System eingesandte Änderung (s.g. *commit*) wird dabei in einem Protokoll festgehalten und ist somit für alle Projektteilnehmer einsehbar, inclusive eines zusammenfassenden Kommentares betreffend der Änderungen.

Aus der Rekonstruktion mittels dieser Logbücher wurde folgendes Diagramm erstellt, das die Entwicklungsschritte in chronologischer und kausaler Folge zeigt.


ocor
c2: testudsc
c3: genudsc
fms
c4: sync_split
c10: quint
c1: idea
announce
c5
c6
c7
c8
Vergangenheit
Mitte August
Ende August
Anfang September
Mitte September
Ende September

(0,2)


c9
uds
quint
c13
c14
c15
quit
c11
c12
Anfang
Oktober
doc
update
c16
c17
c18
c19
c20
c21
c22
c23
Mitte
Oktober
Ende
Oktober
Anfang
November
c24: merge
c25: merge
c26: sync_merge
c27: uds
procedure
c28: protocol
c31
c30: player_split
c29
c34: zoom
c33: volume
c32
c35
c37
c38
c36
fixes
c42
c43
c44
c45
fms
backend
c39
c40
c41
c46
c47: fms
protocol
Mitte
November
c48
c49
c50
c52
c53: player
c51: numbering
Ende
November
c54
c55
c57
c56
c58
c60: Qt
c59
c61
c62
c63
c65
(0,1)


c66
c67
c70: XMas
c71
c72
c74: colors
c73: midi detect
Deko
c75
c76
c77
c78: lock
c79
fixes
c81
c82
c83
c84
c91
c107
cleanup
c105
c106
cleanup
c126
c129
manpage
c102
c103
c87
debug mode
c92
c93
c112: resolve
c130: man
c64
c68
c69
c80: SDL color
c119
c121
build sys
c122
c123
c124
c88
c95
c97
c116: syncMidi
c135
graphs
c85
c86
c89
c90
c94
c96
graphs and build
c98
c99
c100
c101
c104
c108
c109
doc and presentation
c110
c111
c113
c115
c117
c118
c120
c125
c127
c128
c131
c132
c133
c134
c136
c137
Anfang Dezember
Mitte Dezember
Ende Dezember
Anfang Januar
Mitte Januar

(0,0)

Die Entwicklung der einzelnen Programmbestandteile ist – soweit für Außenstehende interessant – im Folgenden beschrieben.

2.3 FMS

An der Entwicklung von FMS lassen sich Programmierstrategien beim Bearbeiten großer *code bases*, also ausgedehnter Programmierprojekte, erkennen.

2.3.1 Ausgangssituation

FMS war zu Projektbeginn bereits ein großes Projekt mit verschiedenen Programmen wie Einlese-, Konvertier-, Anzeige- und Abspieltools, die zusammen einen digitalen Synthsizer bilden.

Während die Funktionalität dieser Programme erhalten werden sollte, musste FMS um zusätzliche Funktionen erweitert werden:

> Synchronität: FMS beruhte auf dem Prinzip der Vorausberechnung (unter Berücksichtigung von Wiederholungen) von Klangwerten im Speicher, die nach Abschluss der Berechnung ausgegeben werden konnten; für FROCOR war es aber nötig, die Eigenschaften des Klanges noch während des Abspielens zu verändern.
>
> FMS-Backend: Da nicht erwünscht war, den FMS-Sourcecode direkt in frocorplayer linken zu müssen, musste eine zusätzliches Backend geschrieben werden, das über UDS mittels eines eigens erarbeiteten Protokolls empfangene Befehle interpretiert und den ausgegebenen Klang entsprechend konfiguriert.

2.3.2 Exportieren und Merge

Um die Funktionalität synchronen Abspielens zu implementieren, ohne die FMS code base irreparabel zu beschädigen wurden die betreffenden Sourcecode-Dateien in die separate Branch *sync_fms* überführt (commit 4).

In der neu erzeugten Branch konnten verschiedenen Änderungen schnell und ohne an anderen FMS-Programmen Änderungen vornehmen zu müssen durchgeführt werden. Nach Überarbeitung der Funktionen zum synchronen Abspielen und Testen mit verschiedenen Testprogrammen mussten die neu geschaffenen Funktionalitäten in die eigentliche FMS code base eingefügt und mit möglicherweise in der Zwischenzeit dort vorgenommenen Änderungen kombiniert werden. Außerdem waren einige Anpassungen nötig, die den übrigen FMS-Tools die Nutzung der Synchronität ermöglichen. Allerdings wurde Rückwärts-Kompatibilität erhalten, so dass die veränderte FMS-API auch mit unveränderten FMS-Toolprogrammen kompilierbar ist.

2.3.3 Synchronität

Der folgende Abschnitt erklärt einige Veränderungen, die zur Schaffung des synchronen Abspielmodus nötig waren.

Das Abspielen erfolgt in einem fünfschrittigen Prozess:

Öffnen der Dateien

Berechnen der Werte

Initialisieren der Sound-Hardware

Abspielen der Werte

Beenden der Sound-Hardware

Bemerkenswert dabei ist vor allem die Trennung von Berechnung und Abspielen der Werte, die zwischen diesen beiden Vorgängen im Arbeitsspeicher gelagert bleiben.

Es folgt ein Diagramm, das den API-Internen Aufruf der Funktionen bei einem einfachen, nicht-synchronen Abspielprozess (ausgelöst durch die Prozedur `FMPlayer::play`) darstellt (gestrichelt – nur optional ausgeführt, gepunktet – Bibliotheksfunktionen):

Für synchrones Abspielen wurde dieser Ablauf verändert, indem die Einzelbestandteile neu aufgeteilt wurden:

Öffnen der Dateien

Initialisieren der Sound-Hardware

Berechnen einiger konstanter Werte (z.B. Modulationsfrequenzen)

Berechnen und Abspielen der Werte (jeder Wert wird einzeln berechnet und abgespielt)

Beenden der Sound-Hardware

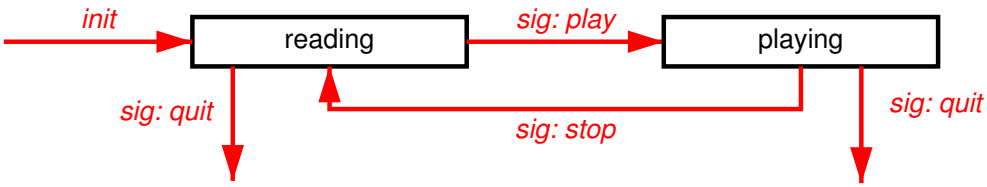
Auf Ebene des FMPlayer- und FMChannel-Objektes mussten hierfür neue Funktionen geschrieben werden, die die bereits vorhandenen Funktionen in richtiger Weise ansteuern. Zudem musste der Berechnungsprozess, der vorher von den FMMixer-Objekten gesteuert wurde, in drei Teile aufgeteilt werden (siehe erstes Diagramm), von denen nur der erste, `FMMixer::computeInit` für synchrones Abspielen von Bedeutung ist.

Die Darstellung der Funktionen in Diagrammform war bei der Lösung der Probleme hilfreich und die einzige Möglichkeit, den Überblick über die doch recht komplexen Abläufe zu behalten. So konnte z.B. ein Fehler, der beim Abspielen von nicht in Dateien befindlichen sondern zur Laufzeit erzeugten FMS-Sounddateien auftrat, nur durch das Diagramm analysiert werden. Kedens `FMPlayer::syncInit` rief `FMChannel::openFiles` auf, was ohne zu öffnende Dateien natürlich zum Fehler führte.

2.3.4 FMS-Backend

Die Entwicklung des FMS-Backends erfolgte, nachdem ein von direkten FMS-Funktionen bereinigter FROCOR-Player geplant worden war (commit 30). Das FMS-Backend wurde entwickelt und war mit commit 46 einsatzbereit, die vollständige Dokumentation des FMS-Protokolls folgte mit commit 47.

Zustandsübergange



Das Empfangen der Protokollversion „fms-protocol 0.90 “macht fmsbackend bereit zum Empfangen der Daten. Mittels der Befehle „play “und „stop “wird der Abspielvorgang gestartet und gestoppt. Die Anweisung „quit “beendet fmsbackend. Vereinfacht können diese Abläufe obigem Zustandsdiagramm entnommen werden.

Threads

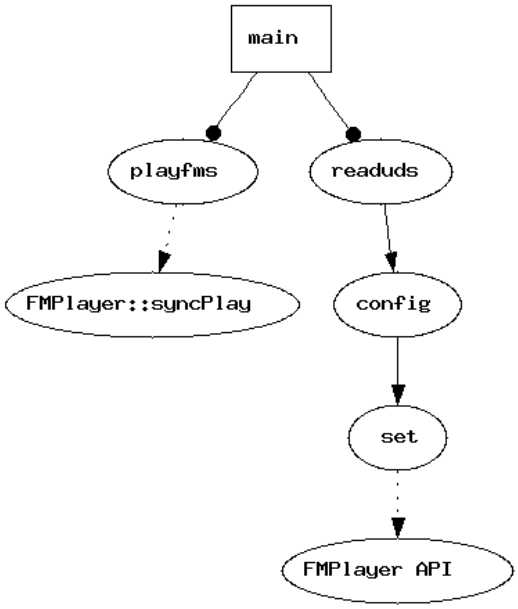
Da beim Lesen von Sockets die Unix-read()-Funktion so lange wartet, bis tatsächlich Daten gelesen werden, kann die parallele Ausgabe von Soundwerten innerhalb des selben Prozesses nur durch Abspaltung eines s.g. *Threads* ermöglicht werden. Es wurden dabei zwei verschiedene Arten von Threads getestet:

1. CLib-Fork: Über das C-Kommando `fork` lässt sich innerhalb eines C-Programmes ein s.g. Child-Prozess erzeugen. Über Abfragen der Prozess-ID lässt sich im Programmablauf feststellen, ob der gerade ausgeführte Prozess ein solcher Child-Prozess oder der Hauptprozess ist, so dass vom Child-Prozess eine andere Aufgabe übernommen werden kann. Problematischerweise werden bei `fork` alle Variablen mitkopiert und es ist so unpraktikabel, die Variablen des jeweils anderen Prozesses (hier: des Abspielprozesses) zu verändern. Im Umgang mit Pointern zeigt sich unerwartetes Verhalten. Daher ist diese Art der Threads für die Anwendung im FMS-Backend nicht geeignet. Sie wird aber z.B. im genudsc zum gleichzeitigen Lesen und Schreiben von und auf dem Socket genutzt.

2. POSIX-Threads (*pthreads*): Ein Prozess kann viele POSIX-Threads enthalten, die sich einen Satz von Variablen teilen. Ein Thread funktioniert dabei wie eine Prozedur, die aufgerufen wird, wobei bei der Fortführung des Programmes nicht auf die Beendigung der Prozedur gewartet wird. Die Bezeichnung POSIX-Threads ergibt sich aus der Geschichte von UNIX.

Um mit POSIX-Threads zu arbeiten, muss der Header `pthread.h` eingebunden und das Programm mit `lpthread` verlinkt werden.

Der Aufruf der Funktionen innerhalb des FMS-Backends erfolgt wie illustiert:



Die folgende Zeile (man beachte die Typenumwandler) starten die Prozedur `fms_player` in einem separaten pthread und übergeben ihr den Pointer auf `Player` als Parameter:

```
pthread_create(&fms_player, 0, (void *(*)(void *))playfms, (void *)
Player);
```

Ein Thread beendet sich selbst durch Aufruf von `pthread_exit(int code)`. Zu beachten ist, dass unter Linux der system call `sleep` nur den aktuellen Thread aussetzt, während nach POSIX-Definition unter anderen UNIX-Derivaten `sleep` den ganzen Prozess unterbricht, weshalb hier `pthread_delay_np(int msec)` verwendet werden muss (`np` steht ironischerweise für *non portable*).

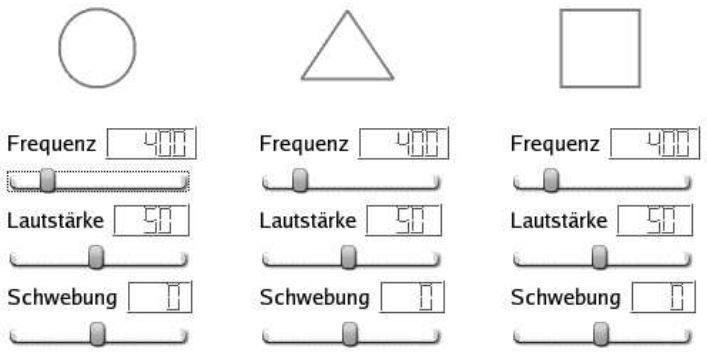
Während der erste Thread in FMS-Backend die Soundwerte abspielt, liest der zweite Thread die UDS-Eingaben und konfiguriert die Soundoptionen (`config` für jede Anweisungs-Zeile ruft für jede Einzeloption `set` auf). Die Kommunikation zwischen den Threads funktioniert mittels globaler Zustandsvariablen (siehe Zustandsdiagramm).

2.4 Frocor-Player

Der Frocor-Player ging mit commit 30 aus der kaum mehr gepflegten sync_fms-Branch hervor. Erst nachdem ein funktionierendes FMS-Backend zur Verfügung stand, machte es Sinn, die Entwicklung des players voranzutreiben. Dies war ab commit 46 der Fall, weshalb mit commit 53 eine erste rein konsolenbasierte Version von frocor-player mit rudimentären Funktionen (einfache Testfigur ohne Benutzersteuerung) entstand.

2.4.1 GUI

Mittels der GUI-Bibliothek Qt wurde mit commit 60 ein Widget geschaffen, das alle zum Wählen der Klang- und Figuroptionen nötigen Steuerelemente enthält. Dieses Widget wurde in die bestehende Umgebung einer UDS-Verbindung integriert und so zum eigentlichen Frocor-Player.



2.4.2 Farben

Probleme bereitete lange Zeit das Farbschema von SDL. Durch Test wurde herausgefunden, dass sich die SDL-Farbwerte als 16bit unsigned integer wie folgt aufbauen:

5 bits für Rotanteil

6 bits für Grünanteil

5 bits für Blauanteil

Konvertierungsfunktionen wurden mit commit 80 eingeführt. Dabei ist besonders auf den fließenden Übergang beim Frequenzwechsel zu achten, wobei auch zu extreme und zu dunkle Farben vermieden werden sollen. Für die Frequenz f gelten daher für die Farbwerte folgende Gleichungen:

$$r = 255 \cdot (1 - o\frac{f}{f_{max}})$$

$$g = 255 \cdot (1 - o\frac{|f - \frac{f_{max}}{2}|}{f_{max}})$$

$$b = 255 \cdot (1 - o\frac{f_{max} - f}{f_{max}})$$

o ist dabei der *overlapping*-Parameter, der für sanfte Übergänge sorgt und in der aktuellen Version auf dem Wert 1.2 steht (es muss gelten $o \leq 2$). f_{max} ist auf 2000Hz gesetzt.

2.5 Build-System

Die code base von frocor umfasst, auch aufgrund der Einbindung großer Teile bereits bestehender Projekte, über 40.000 Zeilen Code. Ein kompletter Compile-Vorgang dauert auch auf recht schnellen Systemen über eine Minute.

Aus diesem Grund wurde das GNU-Build-Tool `make` eingesetzt. In jedem Sourcecodeverzeichnis vefindet sich eine `Makefile` in der verschiedene *Targets* definiert werden. Ein Target ist typischerweise das Ergebnis des Kompilierens, also die ausführbare Datei. Zu jedem Target gehört eine Angabe der Abhängigkeiten (also der Dateien oder anderen Targets, bei deren Änderung das Target neu kompiliert werden muss) und der Kommandozeilen-Befehl, der zum Wiederauffrischen des Targets dient.

Nach Änderung einiger Sourcecodedateien erkennt GNU Make beim Aufruf von `make (target)` automatisch, ob und welche Targets wie neu kompiliert werden müssen. So kann enorm viel Kompilier-Zeit gespart werden.

Über die Definition eines Targets `clean` ohne Abhängigkeiten, dessen Auffrischung einen Befehl enthält, der temporäre Dateien etc. löscht, kann mit dem Kommando `make clean` das Sourcecodeverzeichnis von unerwünschten Dateien, die nicht ins echte Repository übertragen werden sollen (siehe 2.6) bereinigt werden.

2.6 Version-Controlling

Durch die Arbeit an einem großen Projekt wie Frocor mit 2 Programmierern ergaben sich einige Anforderungen an ein organisatorisches System:

- Verteilung der jeweils aktuellsten Version
- Benachrichtigung des anderen Programmierers über die gemachten Änderungen
- Aufzeichnung der Änderungen zum späteren Nachvollziehen und Dokumentieren
- nötigenfalls Rückgängigmachen ungelungener Änderungen
- Kombination gleichzeitig von unterschiedlichen Programmierern vorgenommenen Änderungen

Version Control Systems wie `svn` speichern von Programmierern gemachte Änderungen als Patches mit zugeordnetem Log-Eintrag und ermöglichen so das Herunterladen der aktuellsten Version, das Aufzeichnen von Änderungen mit Kommentaren, das Wiederherstellen alter Dateiversionen und auch die Kombination unterschiedlicher Änderungen (s.g. *merge*). Mehr Informationen zu svn sind auf der Homepage `http://subversion.tigris.org` erhältlich.

2.7 Motivationsmanagement

Der Begriff *Motivationsmanagement* stammt aus der Wirtschaftstheorie und beschreibt die Erforschung menschlichen Motivationsverhaltens. Es sollen hier drei Prinzipien der Motivation aufgeführt werden.

2.7.1 Kreativität - Projekt XMas

Das Lösen scheinbar unwichtiger Aufgaben rein spielerischer Natur kann die Entwicklung bedeutenderer Programmteile erheblich beschleunigen, da es das konkrete Lösen einer überschaubaren Aufgabe nötig macht.

Ein Beispiel hierfür ist das Subprojekt `xmas`. Gegen Ende des Jahres 2004 wurde im CC-Team Stuttgart ein Wettbewerb ausgeschrieben, bei dem es um das möglichst kreative Programmieren eines Weihnachtsbaumes ging.

Zu diesem Zweck wurde frocor-player durch einen Server ersetzt, der das FMS-Backend zum Abspielen eines Weihnachtsliedes ansteuert und Ocor die geometrischen Daten eines einfachen Weihnachtsbaumes aus Drei- und Recheck übermittelt.

Da der SDL-Client zu diesem Zeitpunkt noch keine Farben unterstützte, musste diese Funktionalität, die auch für FROCOR von Bedeutung ist, hinzugefügt werden. Der nun farbenfrohere Baum wurde nun auch mit Kugeln geschmückt. Nach Hinzufügen von *objektorientiertem Schnee* wurde die Animation mit Ocor so langsam, dass über eine Lock-Unlock-Routine das ständige Wiederauffrischen genauso unterbunden werden musste, wie die allzu häufige Ausgabe von Debug-Meldungen. Zudem musste das FMS-Backend erweitert werden, um Abspielen von FMS-Midi-Dateien zu ermöglichen, was letztlich zum Fix eines Bugs führte, der im synchronen Abspielmodus mit musplay auftrat. Ein Bug, der beim automatischen Zoomen von Ocor auftrat, wurde durch die Animation des Sterns offensichtlich und in der Folge gefixt.

Zu diesen Fortschritten am Projekt kam hinzu, dass zwei Rechner in der Musikhochschule, an denen das XMas-Projekt vorgeführt wurde, durch Nachinstallation auf den neuesten Stand der benötigten Bibliotheken gebracht wurden, was vorher viele Monate lang nicht gelungen war.

Diese vielen Fortschritte zeigen, dass auch spielerische Vorgehensweise zum Projekterfolg beigetragen hat.

2.7.2 Deadlines

Von der ersten Planung an wurde bei FROCOR auf klare Zieldaten für die Fertigstellung der einzelnen Komponenten und Funktionalitäten wert gelegt. Dabei zeigt die Erfahrung:

> nicht zu früh: auch wenn ein Entwicklungsschritt leicht zu realisieren scheint - da die weitere Entwicklung davon abhängt, muss in der Projektplanung eine gewisser

Puffer für Eventualitäten gelassen werden

trotzdem gleich anfangen: die großzügige Festlegung der Deadlines soll natürlich nicht zur Verzögerung des Arbeitsbeginnes führen

Zeit für Dokumentation: gegen Ende der hier dokumentierten Projektphase kam es zu sehr vielen commits und einem großen Zeitbedarf wegen der anstehenden Präsentation und Dokumentationsabgabe; dieser Zeitbedarf hätte zu Projektbeginn besser berücksichtigt werden sollen

2.7.3 Zusammenarbeit

Da FROCOR eine Projekt zweier Programmierer ist, waren Synergieeffekte zu beobachten. Nicht nur führte ein von einem Programmierer für die Programme des anderen ausgesprochene Entwicklungs-Wunschliste oft zu schneller Realisierung, auch war die Information über die Fortschritte des anderen ein Ansporn, selbst auch etwas zu programmieren. Diese Effekte wurden durch die Kommunikation über die eingerichtete Mailing-Liste, das Version-Control-System und die persönliche Kommunikation verstärkt.

17.01.2005: Daniel Grün

Kapitel 3

Benutzerdokumentation

FROCOR besteht aus vier Programmteilen, die im einfachsten Fall mittels des Skriptes `frocor` im Verzeichnis `branches` aufgerufen werden.

3.1 Systemvoraussetzungen

FROCOR benötigt folgende softwareseitigen Systemvoraussetzungen:

Linux-Betriebssystem mit üblichen Development-Tools

OSS /dev/dsp (für Soundausgabe)

SDL-Bibliothek (`http://libsdl.org`)

SDL-Pascal-Bindings (`http://sdl4fp.sourceforge.net`)

Qt mit Headern und Tools (`http://www.trolltech.no`)

3.2 FROCOR-Player

3.2.1 Kompilieren

Da FROCOR Qt-Komponenten enthält, muss eine aktuelle Development-Version dieser Bibliothek installiert sein. Das configure-Skript findet diese und richtet die `Makefile` (siehe 2.5 auf Seite 23) dementsprechend ein. Sollte eine andere Installation als die gefundene verwendet werden sollen, kann das zugehörige Basisverzeichnis dem configure-Skript mit dem Kommandozeilen-Argument `--with-Qt-dir=/your/qt/dir` angegeben werden.

3.2.2 Benutzung

Der Player wird mittels des Befehls `frocor` im Unterverzeichnis `frocor-player` gestartet. Es sind keine Kommandozeilen-Argumente anzugeben.

Werden sowohl auf `/tmp/frocor-uds` als auch auf `/tmp/fms-uds` UDS-Clients in korrekter Weise initialisiert, startet eine GUI. Sie ermöglicht die Veränderung der Objekt-Eigenschaften auf einfache Weise. Beenden der GUI beendet FROCOR.

3.3 Ocor

Ocor wird über die ausführbare Datei `ocorbackend` in der Branch `ocor` aufgerufen. Vorher muss ein das FROCOR-Protokoll verstehender Server auf der UDS-Datei `/tmp/-frocor-uds` bereitgestellt werden. Um Ocor und SDL-Client zusammen zu starten, kann das Script `startocor` benutzt werden.

3.4 SDL-Client

Der SDL-Client wird über die ausführbare Datei `ocor-sdlclient` in der Branch `sdl-client` aufgerufen. Vorher muss ein das OCOR-Protokoll verstehender Server auf der UDS-Datei `/tmp/ocorsdl-user.0` bereitgestellt werden. Um einen nicht mehr reagierenden SDL-Client zu beenden, kann der Befehl `killall -9 ocor-sdlclient` benutzt werden.

3.5 FMS-Backend

Der Aufruf des FMS-Backends ist in der FMS-Dokumentation in Kapitel 7.8 auf Seite 43 beschrieben.

3.6 TCP-UDS-Connector

3.6.1 Server

Ablauf:

1. Starten des UDS-Server-Programms (z.B. frocor-player oder ocor)
2. Starten von tucserver [uds file name]

Der TCP-Port 3216 muss freigeschaltet sein, damit Verbindung mit dem Server aufgenommen werden kann:

1. Starten von tucclient [uds file name] [server]

2. Starten des UDS-Client-Programms (z.B. fmsbackend, ocor, sdlclient)

Beim quit-Kommando eines der zu FROCOR gehörenden Protokolle beendet der Adapter selbsttätig.

3.7 Generic UDS

Generic UDS kann als Server oder Client für ein beliebiges UDS-Socket eingesetzt werden, zum Beispiel um die Funktion und Implementierung eines Protokolls zu testen. Er gibt die empfangenen Daten in `stdout` aus und schreibt Konsoleneingaben auf das Socket.

Programmaufruf (in der Branch `genudsc`):

```
[genuds ([--server]|[--client]) (-f /your/uds/socket)
```

Das Programm beendet, wenn die UDS-Kommunikation abbricht oder mit `Strg+C`.

3.8 XMas

XMas wird ohne weitere Kommandozeilen-Argumente gestartet. Die Animation beginnt, wenn mittels des Skriptes `startxmas` die dazugehörigen Clients gestartet wurden.

Mittels der Taste `q` kann das Programm beendet werden, `d/s` startet und unterbricht die Animation des Sterns.

3.9 Man Pages

Unter Linux ist es üblich, jeden installierten Befehl in einer sogenannten *man page* zu dokumentieren.

Die Komplexität dieser Dokumentation hängt vom Autor ab, enthält jedoch in jedem Fall eine kurze Beschreibung des Befehls sowie seiner Kommandozeilenargumente.

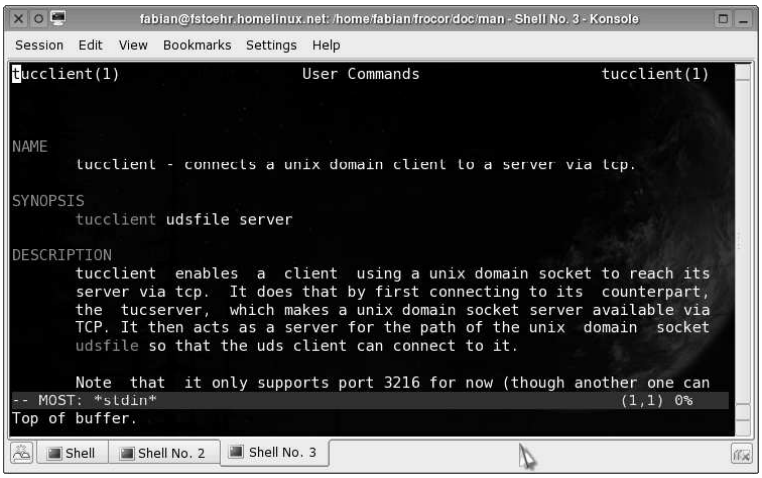
Außerdem wird, so diese existiert, auf weitere Dokumentation verwiesen.

Die man pages werden in einer speziellen, recht eigenwilligen Syntax geschrieben. Dabei enthält eine man page verschiedene Sektionen, für welche es grobe Richtlinien gibt – z.B. die Sektionen synopis, description, options und bugs.

Wie die Syntax interpetiert wird, hängt vom verwendeten Frontend ab. Das Standard-Frontend, der Befehl *man*, wird einfach über die Kommandozeile gestartet und verwendet sogenannte Pager zum Anzeigen der man pages. Einige Frontends unterstützen farbige man pages oder man pages mit verschiedenen Schriftarten und -größen, andere nicht. Den meisten Frontends gemein ist jedoch, das eine man page als ein komplettes Dokumment vom beginn bis zum Ende angezeigt wird, auch bei oft mehreren tausend Zeilen Länge.

Die Dokumentation Frocors durch man pages ist momentan in Arbeit. Bisher wurden die Befehle, die nicht zu den eigentlichen Frocor-Komponenten gehören, entsprechend dokumentiert, denn dies sind auch diejenigen Befehle, die Kommandozeilen-Argumente benötigen (neben vielen der von Frocor nicht benötigten FMS-Bestandteile).

Es folgt ein Screenshot einer gerade angezeigten Manpage:



Kapitel 4

Präsentation

Das Projekt Frocor wird in einer Präsentation vorgestellt.

Dabei wird anhand ausgewählter Beispiele gezeigt

- wie das Projekt aufgebaut ist
- wie die Kommunikation der einzelnen Programme funktioniert
- was beim Projektmanagement beachtet werden musste
- welche grundlegenden Programmiertechniken eingesetzt wurden

Die Präsentation wurde u.a. mit Hilfe des Programmes MagicPoint [**?**] erstellt.

4.1 Outline

	Einleitung	
1	Projektüberblick	Daniel und Fabian
2	Ideenfindung	Daniel
3	Announce-Dokumentation	Fabian
4	bottom-up und top-down Approach mit Diagramm	Daniel
5	Funktionsdemo	Fabian
	Programm-Kommunikation	
6	Unix Domain Sockets	Fabian
7	Protokolle	Daniel
8	State-Diagram	Daniel
9	TCP	Daniel
10	Funktionsdemo TCP	Fabian
	Entwicklungsprozess	
11	Version Control Systems	Fabian
12	Merge	Daniel
13	Commit Diagram	Fabian
14	Rolle der Kreativität	Daniel
15	Xmas-demo	Fabian
	Programmiertechniken	
16	Prozeduren (Vorbereitung OO)	Fabian
17	(Synchronität) Threading	Daniel
18	Objekt-orientierte Programmierung	Fabian

Teil II

Ocor

Kapitel 5

Object Oriented Ocor

Im Rahmen dieses Projekts wurde ocor auf objekt-orientierte Programmierung portiert. Dazu wurden die objekt-orientierten Units des Free Pascal Compilers verwendet. Dieses Kapitel beschreibt jene Portierung.

5.1 Erste Überlegungen

Zur Benutzung der objekt-orientierten Programmierung unter Free Pascal (welches als meines Wissens einziger Pascal-Compiler ocor kompilieren kann) gibt es mehrere Möglichkeiten:

Verwendung der "objects" unit

Verwendung von classes

Nach dem ersten Eindruck schien die objects unit zum Lernen und Konvertieren der aktuellen Programmstrukturen einfacher zu sein. Die Delphi-ähnlichen classes bieten zwar einige wenige zusätzliche Möglichkeiten (z.B. im Bezug auf mehrfache Vererbung), diese wurden in ocor jedoch nicht verwendet.

Die im Bezug auf die Portierung wichtigste Struktur des Programms heißt "shape", speichert die Eigenschaften jeder grafischen Figur und war zuerst durch ein record-array, damals noch auf 100 beschränkt, implementiert (vormals einzelne arrays für jede Eigenschaft).

Die folgenden Möglichkeiten waren zur Portierung dieser Struktur bekannt:

5.1.1 Verkettete Liste

Die einzelnen Figuren sind über den Speicher verteilt. Es existiert eine Pointer-Variable auf die erste Figur, diese enthält einen Zeiger auf die zweite etc.

Vorteile:	**Nachteile:**
Effektivere Nutzung des Speichers	Starke Änderungen am Programm nötig
Leichte Implementierung (leichter als dynamische Arrays)	Auf Objekte mit spezifischen Nummern kann nur kompliziert zugegriffen werden (z.B. für Figur 20: Objekt 0 Objekt 1 Objekt 2 ... 20)
Einfacher Zugriff auf alle Objekte (Objekt 1 Objekt 2)	
Leichter Änderung der Reihenfolge (d.h. Löschen von Objekten)	

5.1.2 Array des Objektes

Vermutlich nicht möglich, da Objektvariablen letztlich nur Pointer sind

5.1.3 Array von Pointern

Ein Array von Pointern wird initialisiert, die Pointer dann wenn nötig auf entsprechend reservierten Speicher gesetzt.

Vorteile:	**Nachteile:**
Effektivere Nutzung des Speichers	Indirektionsoperator nötig.
Es kann leicht auf einzelne Objekte zugegriffen werden	Nicht wirklich unbegrenzt viele, da eine gewisse Menge von Zeigern initialisiert werden muss ev. auch dynamisch, dann aber komplizierter
Änderung der Reihenfolge eher leicht und schnell (shape[i]:=shape[i+1] in Schleife)	

Da diese Methode am sinnvollsten schien, wurde ein erster Versuch mit einem Array von Pointern unternommen. Die Veränderungen im Quelltext (hinzufügen des $\hat{}$) mussten zwar einzeln bestätigt werden (da z.B. beim Vertauschen von Objekten besser die Pointer als die eigentlichen Variablen vertauscht werden sollen), dies konnte jedoch durch Makros stark vereinfacht werden

Danach war der ursprüngliche shape record in ein Objekt konvertiert, damit konnte die Zuteilung der restlichen Variablen zu Objekten sowie die Aufteilung einiger Prozeduren in Methoden beginnen.

5.2 Variablen

Einige der Variablen waren bereits im shape record untergebracht und konnten so leicht in Objekteigenschaften konvertiert werden. Andere hingegen gehoerten keinem record an, daher mussten fuer sie Objekte erstellt werden:

5.2.1 Entfernte Variablen

Einige der globalen Variablen konnten entfernt werden, da sie nur lokal oder überhaupt nicht mehr benötigt wurden:

linenr gibt die Nummer der Linie in einer Datei, die gerade eingelesen wird, an. Wird nur von der Prozedur „loaddrawing “benötigt, daher lokal.

position Position in einer Datei, die gerade gelesen wurde. Wird von mehreren Prozeduren benötigt, kann jedoch statt als globale Variable als Parameter übergeben werden.

istring:string Temporäre string Variable. Großteils bereits lokal, wurde jedoch noch von einer Prozedure (gettillspace) als globale Variable benötigt. Diese Prozedur wurde als Funktion umgeschrieben.

fileline:string Ähnlich istring

5.2.2 sock

Sock enthält alle für den Umgang mit Sockets nötigen Variablen

sock.sdl

Variablen für die Verbindung zum sdl-client. Diese sind momentan:

alter Name	neuer Name	Typ	Funktion
sdlsocket	descriptor	longint	Socket descriptor (dem Socket zugeordnete Nummer)
sdlsocketaddr	path	string	Pfad und Dateiname des Sockets
sdlclientaddr	clientaddress	string	Vermutlich nicht mehr benötigt
sdlbackendin	backendin	text	Textvariable, via derer das ocor backend aus dem socket liest
sdlbackendout	backendout	text	Textvariable, mit der das ocor backend in den socket schreibt
buffer	—	string	Wir nicht mehr benötigt
accepted	accepted	longint	socket descriptor nach dem Verbindungsaufbau (eventuell können fehler anhand dieser Variable festgestellt werden).
addlen	addresslength	integer	Länge des SocketPfades, da die Socketfunktionen sehr alt sind und deshalb das Ende eines strings nicht selbst ermitteln können. Muss als Pointer übergeben werden und kann deshalb nicht direkt berechnet werden. Bald hoffentlich lokal.

5.2.3 strings

Methoden zum Umgang mit strings. Dieses Objekt hat keine Eigenschaften, es sah jedoch zwischendurch so aus, als würden die folgenden nun lokalen Variablen Eigenschaften dieses Objektes werden:

linenr

position

istring

fileline

5.2.4 Das Graph Objekt

Eigenschaften (und Methoden), die sich nicht auf einzelne Objekte sondern auf den Graphen als ganzes beziehen, werden im Graph Objekt untergebracht. Einige dieser

Variablen konnten als lokale Variablen implementiert oder ganz enfernt werden. Ausserdem gibt es Untereigenschaften (z.B. graph.resolution.x) geben.

alter Name	**neuer Name**	**Typ**	**Funktion**
resolx	resolution.x	word	x-resolution
resoly	resolution.y	word	y-resolution
resolxo	—	word	x-resolution vor dem subtrahieren der Seitenränder. Nicht mehr benötigt, da jetzt direkt beim ausrechnen von resmult subtrahiert wird.
resolyo	—	word	y-resolution zu resolxo
xmin	graph.oldxmin	real	Niedrigster X-Wert auf dem Graphen. jetzt Methode graph.xmin, gespeichert in graph.oldxmin, da teils noch der ältere Wert benötigt wird.
xmax	—	real	höchster X-Wert. Jetzt Methode graph.xmax
ymin	—	real	niedrigster Y-Wert. Jetzt Methode graph.ymin
ymax	—	real	höchster Y-Wert. Jetzt Methode graph.ymax
xtotal	—	real	Positives xmax: xmax-xmin. Jetzt direkt ausgerechnet
ytotal	—	real	ymax-ymin. Jetzt direkt ausgerechnet
resmult	graph.resmult	real	Nummer, mit der die Koordinaten multipliziert werden müssen, umd sie in pixel zu konvertieren.
sxresmult	graph.sticky.resmult.x	real	wie resmult, jedoch x-multiplier der unregelmäßigen sticky objects
syresmult	graph.sticky.resmult.y	real	wie sxresmult,nur y-multiplier
saveview	graph.saveview	boolean	Sollen die im Graphik-Modus vorgenommenen Änderungen behalten werden? Wiederhehrstellung der alten Werte funktioniert momentan nicht.
astep	graph.astep	word	aktueller Animationsschritt

30.10.2004: Fabian Stöhr

Teil III

Fms

Kapitel 6

Einleitung

6.1 Was soll FMS leisten?

FMS ist ein virtueller Synthesizer zur Erzeugung von Klängen, Geräuschen und deren Strukturierung. Mögliche Einsatzgebiete sind Sprachsynthese, Klangdemonstrationen, Rauscherzeugung oder Abspielen musikalischer Abläufe. Es handelt sich um ein offenes, modulares System in ständiger Weiterentwicklung FMS soll auch eine leistungsfähige c++-API für Soundsynthesezwecke bieten und anderen Programmierern die Möglichkeit geben, sich an der Weiterentwicklung zu beteiligen.

Deshalb ist FMS unter der GNU General Public License [6] veröffentlicht und für jedermann frei und auch im Quellcode verfügbar [8].

6.2 Was kann FMS bisher leisten?

- Allgemein:

  Speichern der Wellenformen in Tabellen

  grafische Ausgabe (Oszillogramm)

  Wiedergabe (Klänge) über zwei Kanäle (Stereo)

  Unterstützung von .wav-Dateien

- Spezifikationen:

  Erzeugen beliebiger Frequenzen und Lautstärken

  Amplituden- und Ringmodulation

  Frequenzmodulation

  additive Mischung von Klängen

  Rauscherzeugung durch gewichtete Zufallsfunktionen

Abfolge von Klangsequenzen („Soundpaketen“)
bis hin zur Abspielung von Musikstücken

Emulation von Musikinstrumenten

Kapitel 7

User's Guide - Wie man FMS benutzt

7.1 Installation

Das z.B. von [8] heruntergeladene FMS-Sourcepaket wird nach dem Entpacken mit dem Linux-üblichen Installationsablauf von `make` und `make install` installiert. Voraussetzung für eine erfolgreiche Installation ist ein Linux-System mit gcc-g++-Compiler, einer Soundkarte die /dev/dsp unterstützt und den Bibliotheken SDL [10] und SDL_gfx [11] für grafische Anwendungen. Für die eingeschränkte Tcl/Tk-Oberfläche wird die `tclsh` und `wish` benötigt, die Qt-Oberfläche `qfms` läuft ab Qt-Version 3.1, die direkt von Trolltech [12] heruntergeladen werden kann.

7.2 fmautofile

Um die Eingabe natürlicher Klänge zu erleichtern gibt es fmautofile. Es wandelt eine WAV-Datei, die nur ein einzelnes Oszillogramm (also nur eine Iteration eines frequenten Klanges) enthält in eine FMS-Sounddatei im Modus 1 um.

fmautofile (-s x) (-f x)

Das Programm liest die im aktuellen Verzeichnis befindliche WAV-Datei in.wav ein und schreibt die Ausgabe in out.fms. Dabei kann mit -s ein Weichzeichnungsfaktor und mit -f der maximale Unschärfefaktor (Komprimierungsrate) der Ausgabedatei angegeben werden. Es ist empfehlenswert mit diesen Werten etwas herumzuexperimentieren und die Ergebnisse mit xfmdisplay zu überprüfen. Um diese Vorgänge weiter zu vereinfachen, kann man das Script autotest benutzen.

7.3 fmdisplay

Mit fmdisplay kann man FMS-Oszillogramme auf Konsolen-Ebene darstellen. Dabei übernimmt das Programm folgende Parameter:

fmdisplay [FMS-Datei] [x] [y]

Dabei steht x und y für die Anzahl der Zeichen auf der x- bzw. y-Achse. Im Allgemeinen sollten 80 und 24 gewählt werden.

7.4 fmfile

Fmfile liest die Angaben über eine FMS-Sounddatei manuell ein und schreibt die Datei auf die Festplatte. Programmaufruf:

fmfile [FMS-Datei]

Zunächst fragt das Programm nach dem Speicherungsmodus. Die drei möglichen Modi sind:

1. alle Werte
   In diesem Modus werden einzelne x-y-Werte vom Benutzer angegeben. Die dazwischenliegenden Werte werden von fmfile erzeugt und in die Datei geschrieben. Nur in seltenen Fällen lohnt es sich, diesen Modus gegenüber Modus 2 zu bevorzugen.

2. Werte mit Verbindungslinien
   Dieser Modus speichert einzelne x-y-Werte und den Typ der Verbindungslinie zwischen ihnen. Dies spart meist sehr viel Speicherplatz gegenüber Modus 1 und ermöglicht außerdem saubere Erzeugung von Kurven.

3. Midi-Modus
   Dieser Speicherungsmodus sollte gewählt werden, um Musikstücke in FMS-Midi-Dateien umzuwandeln.

Der weitere Programmdialog sollte selbsterklärend sein.

7.5 fmplay

Fmplay spielt FMS-Sounddateien aller Modi gleichzeitig und/oder nacheinander, optional mit schwankender Lautstärke und Frequenz ab. Eine ausführliche Liste der möglichen Abspieloptionen wird asugegeben, wenn man fmplay ohne weitere Parameter aufruft.

7.6 qfms

Um die Funktionen von fmplay auch ohne Konsolenbefehle zugänglich zu machen, arbeitet der Autor im Moment an einer Qt-GUI. Betatester können sich gerne bei mir melden[9].

7.7 xfmdisplay

Auf einfachste Weise stellt xfmdisplay auch gemischte und in der Frequenz verschiedene FMS-Sounddateien gleichzeitig in einer grafischen Umgebung dar. Der Programmaufruf ist dabei wie folgt:

xfmdisplay (-x x) (-y x) (-a) [FMS-Datei] (-r x) (-v x) ([FMS-Datei])

Dabei kann mit -r die Wiederholungen innerhalb des Fensters und mit -v die Lautstärke der zuvor angegebenen FMS-Datei festgelegt werden. Die Optionen -x und -y übergeben dem Programm die gewünschte Größe des geöffneten Fensters (in Pixeln), ist die Option -a gegeben, wird der Aufbau der Ausgabe animiert. Es kann eine beliebig lange Liste von FMS-Dateien mit -r und/oder -v-Optionen angegeben werden.

7.8 fmsbackend

Das FMS-Backend liest mittels UDS Kommandos im FMS-Protokoll (Dokumentation siehe 1.5.3) ein und erzeugt und konfiguriert diesen Befehlen entsprechend die Soundausgabe.

Programmaufruf

fmsbackend (UDS-Datei)

Wird keine UDS-Datei angegeben, so versucht das Backend, sich als Client mit `/tmp-/fms-uds` zu verbinden.

Kapitel 8

Technical Guide - Wie FMS funktioniert

8.1 Theorie

Die von FMS zur Erzeugung eines Klanges genutzten Daten sind die so genannten Oszillogramme der jeweiligen Klänge. Die Angaben über das Oszillogramm eines Klan-

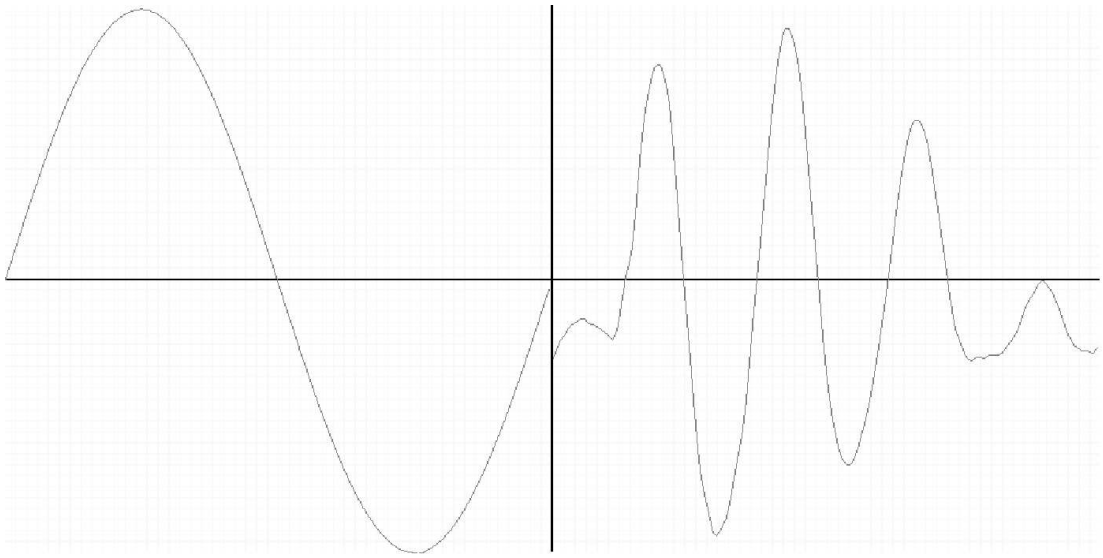


Abbildung 8.1: Oszillogramme von Sinuston und O-Laut

ges werden entweder Wert für Wert in eine Datei gespeichert, oder durch nur wenige Werte mit Angabe ihres Zeitindexes und der Art der Verbindungslinie zwischen ihnen beschrieben und gespeichert. Beim gezeigten Sinuston wären dies nur fünf Werte, z.B. bei den Zeitindices 0, 100, 200, 300 und 400, wobei die Verbindungslinien Sinuskurvenabschnitte sind. Beim O-Laut werden mehr Werte benötigt.

Berechnet man die Werte des so gespeicherten Oszillogramms und gibt sie oft hintereinander als Soundbytes aus, so hört man aus dem Lautsprecher den vorher aufgenommenen Ton mit beliebiger Frequenz (je nachdem, wie viele Werte man pro Wiederholung ausgibt, wie viele ausgegebene Werte also einem Schritt im Zeitindex des Oszillogramms entsprechen). Durch Veränderung dieses Wertes, der bei jedem Zeitschritt zum Zeitindex des aktuell zu berechnenden Wertes hinzugezählt wird, lässt sich Frequenzmodulation realisieren. Die Multiplikation des jeweiligen berechneten Wertes mit dem Wert eines anderen Oszillogramms macht Hüllkurven möglich.

8.2 Schematischer Aufbau

Dabei können die verschiedenen Aufgaben durch den modularen Aufbau von FMS auch von verschiedenen Programmen erledigt werden. So gibt es für das Einlesen schon die Programme fmfile und fmautofile, eine grafische Lösung unter Verwendung der Qt-Bibliothek[12] ist in der Entwicklung. Zur Aufbereitung und Ausgabe der erzeugten FMS-Sounddateien gibt es fmplay (Ausgabe als Klang) oder fmdisplay und xfmdisplay (grafische Darstellung). Auch hier besteht bereits eine noch nicht vollständige GUI-Version.

Programmiertechnisch wird beim Schreiben in FMS-Sounddateien die fmofstream-API benutzt, die ohne große Probleme auch in andere c++-Programme integriert werden könnte. Weitere Informationen hierzu finden sich in dem entsprechenden Abschnitt des Developer's Guide.

Die in entsprechender Form gespeicherten Sounddaten können unter Benutzung der FMPlayer-API bzw. direkt der fmifstream-API gelesen und in tatsächliche Werte umgewandelt werden. Die Ausgabe kann dann in beliebiger Weise, etwa als Klang oder grafisch, erfolgen. Mehr Informationen zu den verwendeten APIs und Programmen können in den entsprechenden Abschnitten des Developer's und dieses Technical Guide gefunden werden.

8.3 fmfile/fmofstream-API

Die einfachste Art der Speicherung (Modus 0, in der Kommunikation mit dem User Modus 1 genannt) eines frequenten Klanges ist die Speicherung sämtlicher Werte, die innerhalb eines Frequenzverlaufes vorkommen. Die daraus resultierenden Sounddateien sind allerdings ziemlich speicherintensiv, weshalb diese Art der Speicherung nur noch aus Kompatibilitätsgründen unterstützt wird.

Der Einlesevorgang verlangt vom Benutzer natürlich nicht die Eingabe jedes einzelnen Wertes. Vielmehr werden bestimmte, frei wählbare x-y-Werte eingegeben und das Programm berechnet die dazwischenliegenden Werte als Gerade mit der aus zwei aufeinanderfolgenden Werten folgenden Steigung.

Ein frequenter Klang kann durch wenige Punkte mit bestimmtem Abstand zum vorhergehenden Wert und Beschreibung der dazwischenliegenden Verbindungslinien beschrieben werden (Modus 1). Dabei genügen vier Bytes (2 Bytes für den Abstand zum vorhergehenden Wert, ein Byte für den Wert selbst und ein Byte für den darauffolgenden Linienstil) um einen kompletten Wert zu speichern und 4 solche Werte um eine komplette Sinuskurve zu beschreiben.

Einem ganz anderen Zweck dient der dritte Speicherungsmodus (Modus 2). Mit ihm kann man die Noten von Musikstücken eingeben um diese zu speichern und später abspielen zu lassen. Die gespeicherten Werte pro Note bestehen daher aus einem 7 Bit großen Wert für die Tonhöhe (mit einem Tonumfang von 120 Halbtönen, also 10 Oktaven), einem 4 Bit großen Wert für den Notenwert und einem 5 Bit großen Wert für das zum Ton gehörige Instrument, einer beim Abspielen frei festlegbaren Kombination aus Klangdatei und Lautstärkenverlauf.

Eine genaue Dokumentation der verschiedenen Dateiformate und der fmofstream-API findet sich im Developer's Guide.

8.4 Berechnen und Abspielen

Die FMPlayer-API übernimmt alle Aufgaben, die für das Berechnen und Abspielen der Soundwerte aus gespeicherten FMS-Sounddateien nötig sind. Die Dokumentation der Programmierschnittstelle ist im Developer's Guide enthalten.

8.4.1 Berechnung der Werte für stimmhafte Klänge

asynchroner Modus

Das Berechnen der Werte ist der zeitaufwendigste Prozess bei der Erzeugung von Klängen. Die Funktion, die dies für jeweils ein einzelnes Soundpaket erledigt, befindet sich unter dem Namen `FMMixer::compute()` in der Datei `fmplayer.cpp`.

Als Erstes begrenzt FMS die Anzahl der berechneten Werte auf die minimale benötigte Anzahl. Diese berechnet sich aus der Samplingrate, also der Zahl der abgespielten Werte pro Sekunde, der Frequenzen der Klänge und der Frequenzen der Kurven, mit denen Lautstärke und Frequenz schwanken. Die Anzahl n der benötigten Werte eines Klanges mit Frequenz f und Samplingrate r wird daher mit folgender Formel berechnet:

$$n = \frac{r}{f} \tag{8.1}$$

Sollten auch Kurven verwendet werden, um die Frequenz und Lautstärke schwanken zu lassen oder Klänge mit anderen Frequenzen gleichzeitig abgespielt werden, so ist der Gesamtwiederholungswert das kleinste gemeinsame Vielfache der Wiederholungswerte der verschiedenen Frequenzen. Dabei muss insbesondere auch darauf geachtet werden,

dass das berechnete n nicht die maximal benötigte Anzahl von Werten für eine Dauer t und eine Samplingrate r überschreitet:

$$n \leq t \cdot r \tag{8.2}$$

Sobald im Speicher Platz für die benötigten Werte geschaffen worden ist, können diese berechnet werden. Dies ist kein Problem bei im Modus 0 (jeder Wert in eine Datei geschrieben) gespeicherten Klängen, da hier lediglich der aktuelle Zeitindex hochgezählt und der entsprechende Wert ausgewählt werden muss. Anders verhält es sich mit Modus 1, bei dem nur wenige Werte mit Angaben zu Zeitindex und darauffolgendem Linienstil in einer Datei gespeichert werden. Hier muss mit Hilfe von je nach Linienstil verschiedenen Formeln der aktuelle Wert berechnet werden. Die folgenden Formeln beschreiben diesen Vorgang für alle unterstützten Linienstile. Dabei ist v_0 der dem zu berechnenden Wert vorhergehende und v_1 der folgende Wert, t_0 und t_1 die entsprechenden Zeitindices sowie v_x und t_x der aktuell zu berechnende Wert und sein Zeitindex.

Die Formel für Linienstil 1, eine gerade Linie, lautet:

$$v_x = \frac{v_1 - v_0}{t_1 - t_0} \cdot (t_x - t_0) + v_0 \tag{8.3}$$

$\frac{v_1 - v_0}{t_1 - t_0}$ ist dabei einfach der Steigungsfaktor $\frac{\Delta y}{\Delta x}$. Dieser wird mit dem Abstand zwischen vorhergehendem und aktuellem Wert multipliziert. Zum Ergebnis wird der vorhergehende Wert addiert.

Die folgenden beiden Formeln berechnen die verschiedenen Sinusabschnitte, entweder mit geradem oder gekrümmtem Anfang (jeweils 90):

$$v_x = v_0 + \sin\left(\frac{t_x - t_0}{t_1 - t_0} \cdot 90 + 0\right) \cdot (v_1 - v_0) \tag{8.4}$$

$$v_x = v_1 + \sin\left(\frac{t_x - t_0}{t_1 - t_0} \cdot 90 + 90\right) \cdot (v_0 - v_1) \tag{8.5}$$

$\frac{t_x - t_0}{t_1 - t_0}$ ist 1, wenn der aktuelle Wert ganz am Ende des Abschnittes steht und 0, wenn er am Anfang steht. Die Gleichungen sollten selbsterklärend sein.

Für die beiden Abschnitte einer Gauss-Kurve (vom x-Wert x_0 bis zur y-Achse bzw. von der y-Achse bis zum x-Wert x_1) sind folgende Formeln nötig:

$$v_x = v_1 + e^{-(x_0 \cdot 1 - \frac{t_x - t_0}{t_1 - t_0})^2} \cdot (v_0 - v_1) \tag{8.6}$$

$$v_x = v_0 + e^{-(x_0 \cdot \frac{t_x - t_0}{t_1 - t_0})^2} \cdot (v_1 - v_0) \tag{8.7}$$

Für die Berechnung von Parabelabschnitten auf der Basis von Angaben des x-y-Startwerts und Endwerts sind drei verschiedene Formeln nötig - je nachdem ob die Parabel symmetrisch ist und ob a 1 oder -1 beträgt. Bei asymmetrischen Parabelabschnitten lautet die Formel wie folgt, wobei x_0 und x_1 die Grenzen der zu berechnenden Abschnitte der Parabel auf der x-Achse bilden:

$$v_x = v_0 + \frac{v_1 - v_0}{x_0^2 - x_1^2} \cdot \left(x_0^2 - \left(x_0 + \frac{t_x - t_0}{t_1 - t_0} \cdot (x_1 - x_0)\right)^2\right) \tag{8.8}$$

Und für symmetrische Parabelabschnitte bei $a = 1$, wobei v_{max} für den Maximalwert und *amp* für die gewählte Amplitude steht

$$v_x = v_{max} - v_1 + amp \cdot \frac{t_a - t_0}{t_1 - t_0} \cdot \left(\frac{t_x - t_0}{t_1 - t_0} - 1 \right) \tag{8.9}$$

beziehungsweise, bei $a = -1$:

$$v_x = v_{max} - v_1 - amp \cdot \frac{t_x - t_0}{t_1 - t_0} \cdot \left(\frac{t_a - t_0}{t_1 - t_0} - 1 \right) \tag{8.10}$$

Sämtliche Formeln sind in der Funktion `fmval()` in `fmval.cpp` enthalten.

Die berechneten Werte müssen natürlich mit den jeweiligen Lautstärkefaktoren (z.B. mit denen einer Hüllkurve) und insbesondere auch mit der relativen Lautstärke des berechneten Klanges zur Gesamtlautstärke aller Klänge multipliziert werden, um Übersteuern zu vermeiden. Alle Lautstärkefaktoren sind dabei Werte zwischen 0 und 1, da die normalerweise maximale Amplitude nicht noch vergrößert werden darf.

Nach Berechnung eines Wertes wird der aktuelle Zeitindex (t_x) weitergezählt, wobei es bei der Ermittlung des hinzugezählten Wertes auf die Frequenz (höhere Freqenz → größerer dazugezählter Wert) ankommt, die natürlich unter Benutzung von schwankenden Frequenzen nicht immer gleich ist.

synchroner Modus

FMS kann, anstatt alle Werte vorher im Speicher zu berechnen und sie erst danach auszugeben, auch ohne Zwischenspeicherung eine somit synchrone Klangausgabe erzeugen.

Zu diesem Zweck muss die Verschachtelung der einzelnen Berechnungsroutinen geändert werden. Die FROCOR-Entwicklungsdokumentation beschreibt diesen Vorgang anhand zweier Funktionsdiagramme (siehe 2.3.3).

8.4.2 Berechnung der Werte für Rauschen

Zur Erzeugung von Rauschen benötigt man Zufallsfunktionen. Erreicht wird das z.B. durch zufällige Variation der Frequenz einer Schwingung. Die C-Zufallsfunktion `rand()` gibt einen zufälligen Integerwert zurück, eine Zahl zwischen -2147483648 und 2147483-648, wobei auf lange Sicht jede Zahl so häufig wie die andere ist (hellgraue Balken im Diagramm). Für den Zweck der Rauscherzeugung ist es jedoch wichtig, eine Verteilungskurve (z.B. Gauss-Verteilung) für den Zufall angeben zu können (dunkelgrau). Die Aufgabe der Erzeugung eines solchen Zufallswertes erledigt die Funktion `fmrand()` in `fmval.cpp`. Es wird ein Standart-Zufallswert erzeugt und berechnet, wie häufig dieser aufgrund der angegebenen Verteilung relativ zurückgegeben werden sollte. Diese Wahrscheinlichkeit wird mit einem weiteren Zufallswert verglichen und je nach Ergebnis zurückgegeben. Dabei wird ein Wert mit der Wahrscheinlichkeit 0.1 nur in zehn Prozent der Fälle akzeptiert. Entscheidet sich die Funktion gegen einen Wert, so ruft sie sich rekursiv selbst auf, um einen neuen Wert zu finden.

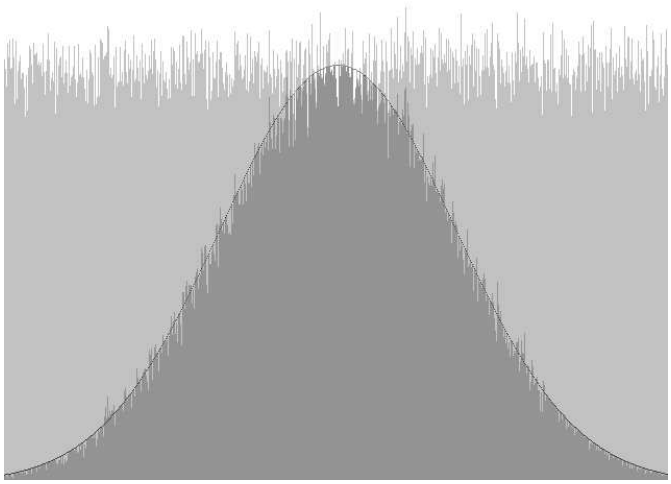


Abbildung 8.2: Zufallsfunktionen im Ereignisdiagramm

8.4.3 /dev/dsp

Linux bietet als Möglichkeit zum Abspielen von Klängen `/dev/dsp`, ein sogenanntes Device, im Prinzip eine Datei, auf die durch Schreib- und Lesezugriffe wie direkt auf die Soundkarte, allerdings durch dahinter stehende Kerneltreiber bequemer und portabler, zugegriffen werden kann. Informationen zur Initialisierung und Programmierung von `/dev/dsp` können entsprechenden HowTos [13] entnommen werden.

Aus der für jeden FMMixer berechneten Reihe von Werten muss beim Abspielen immer der aktuell richtige ausgewählt werden. Dieser wird dann auf `/dev/dsp` geschrieben.

Im synchronen Modus können Wartezeiten oder Underruns die Ausgabe stören, weshalb oft mit einer geringeren Samplingrate gearbeitet werden muss. Ist diese allerdings zu gering, so ist wegen des dsp-Puffers mit einem etwa einsekündigen Vorlauf zu rechnen, bevor Änderungen an den Objekteigenschaften hörbar werden. Es ist also eine der Rechnergeschwindigkeit angemessene Sampling Rate zu wählen

Zusätzliche Schwierigkeiten bietet die Tatsache, dass die Werte eines Soundpakets eventuell mehrfach hintereinander abgespielt werden müssen, sei es vom Benutzer explizit angegeben oder zur Rechenersparnis automatisch bestimmt. Bei Benutzung von zwei Soundkanälen, also im Stereo-Modus, muss darauf geachtet werden, dass immer abwechselnd ein Wert für den linken und ein Wert für den rechten Soundkanal geschrieben werden. All dies erledigt die Funktion `FMPlayer::play()` in `fmplayer.cpp`.

8.4.4 Wave-Dateien

Das Schreiben von Tondaten in eine .wav-Datei bringt generell ein Problem mit sich: das Dateiformat ist ziemlich kompliziert und schlecht dokumentiert, so dass selbst in Suchmaschinen hoch einsortierte Dokumentationen teilweise zumindest schwer verständlich, lücken- oder sogar fehlerhaft sind.

Zuerst muss der sogenannte wav-Header, der Anfang der Wavedatei mit Informationen über Länge, Bitrate, Anzahl der Kanäle u.ä., geschrieben werden. Dann können die rohen Sounddaten einfach als Reihe von Bytes hinzugefügt werden.

Im Developer's Guide zur fmwav-API sind genauere Beschreibungen zum Format und zur Benutzung der API gegeben.

8.5 fmifstream-API

Die fmifstream-API wird intern von FMPlayer genutzt, um FMS-Sounddateien zu öffnen und in verarbeitungsfähigem Format einzulesen. Als einzige Funktion technisch hervorzuheben ist diejenige, welche die Frequenzwerte der Töne bei Dateien im FMS-Midi-Modus berechnet. Da zwölf Halbtonschritte ja eine Verdoppelung der Frequenz bedeuten, muss das Verhältnis v zwischen einem Ton und dem darunterliegenden Halbton folgendermaßen zu berechnen sein:

$$v = \sqrt[12]{2} \tag{8.11}$$

Der n-te Halbton über einem Grundton mit der Frequenz f_0 hat die Frequenz f_1, die man also so berechnet:

$$f_1 = f_0 \cdot v^n \tag{8.12}$$

8.6 andere Elemente

Die anderen Elemente von FMS enthalten technisch nichts Interessantes und müssen deshalb nicht im Technical Guide erwähnt werden.

Kapitel 9

Developer's Guide - Wie FMS programmiert ist

9.1 Vorwort

Dieses Kapitel soll einerseits einen Einblick geben, was der Autor beim Programmieren gelernt hat und andererseits anderen Programmierern eine Möglichkeit geben, mit dem vorhandenen Sourcecode selbst mit FMS verwandte Programme zu schaffen.

FMS ist in c++ programmiert und sollte unter der Betriebssystemplattform Linux mit dem Compiler gcc/g++ 3.2 erfolgreich zu kompilieren sein. Die Einbindung der FMS-Headerdateien (im Unterverzeichnis `include/` des Source Trees) ist in anderen c++-Programmen problemlos möglich. Da FMS allerdings nicht als Bibliothek ausgelegt ist, müssen die zu den jeweiligen Headerdateien gehörenden Sourcecode-Dateien mit ins Programm einkompiliert werden.

Dabei ist natürlich auch das Problem der Lizenzierung zu beachten. Der komplette FMS-Quellcode steht unter der GNU General Public License (GPL) [6], weshalb Programme, die auf FMS basieren, ebenfalls unter der GPL veröffentlicht werden müssen. Als auf FMS basierendes Werk gilt ein Programm, das irgendwelche Sourcecodedateien von FMS benutzt oder mit FMS-Bestandteilen verlinkt ist. Um der GPL gerecht zu werden, muss der Sourcecode eines solchen Programmes der Öffentlichkeit zugänglich gemacht werden, was am Besten über die FMS-Project-Page bei Sourceforge [8] möglich ist. Auf Anfrage per e-mail [9] teile ich gerne weitere Informationen dazu mit.

9.2 Headerdokumentation

Die FMS-APIs und sonstigen Klassen und Funktionen können durch Einbinden der jeweiligen Headerdateien auch in anderen c++-Programmen verwendet werden. Um die Benutzung für den Programmierer so einfach wie möglich zu gestalten, gibt es die folgende Dokumentation.

9.2.1 checkint.h

Die enthaltenen Funktionen `checkint()`, `checkfloat()`, in Sonderfällen auch `chooseint()` gewährleisten in der User-Kommunikation über Kommandozeile, dass eingegebene Zahlen tatsächlich zwischen der gewünschten Minimal- und Maximalzahl liegen, bzw. einem der möglichen Eingaben aus einer Reihe von erlaubten Eingaben entsprechen.

Im einfachsten Fall wird eine Zahl in die Integer-Variable x eingelesen und mit folgender Anweisung verifiziert, wobei `min` und `max` den Minimal- bzw. Maximalwert darstellen:

```
x = checkint(x, min, max);
```

Die Funktion wird sich rekursiv immer wieder selbst aufrufen, bis der Benutzer eine richtige Zahl eingegeben hat.

`chooseint()` übernimmt als Parameter den zu überprüfenden Wert, einen Pointer auf ein Array von erlaubten Werten und die Anzahl der Werte in diesem Array.

Bei Benutzung muss der Header `include/checkint.h` eingebunden und `checkint.o` sowie `fmlanguage_checkint.o` mit ins Programm gelinkt werden.

9.2.2 vtl.h

Der vtl-Header enthält Objekte, die Informationen über FMS-Sounddateien speichern. Um diese zu nutzen, muss nur der Header `include/vtl.h` eingebunden werden, ein Linken mit .o-Dateien ist nicht nötig.

Im Allgemeinen sollte es nicht nötig sein, die fmifstream-Funktionen manuell zu benutzen, sondern es empfiehlt sich vielmehr die Anwendung der komfortableren FMPlayer-API.

VTL

VTL-Objekte (Value-Time-Linetype) enthalten die Informationen über einen Soundwert im Modus 1. Dabei besteht der ganze Wertkomplex aus einem Zeitindex (bei Einlesen der Dateien durch fmifstream: absoluter Zeitindex, sonst relativ zum vorherigen Zeitindex), dem eigentlichen Wert selbst und der Angabe des Linienstils, also des Typs der Verbindungslinie zwischen dem jeweiligen und dem darauffolgenden Wert.

Variable	Bedeutung
`int VTL::time`	Zeitindex
`unsigned char VTL::value`	Wert (Ausschlag)
`unsigned char VTL::linetype`	Linienstil
`int VTL::attr1`	zusätzliche Eigenschaft 1
`int VTL::attr2`	zusätzliche Eigenschaft 2
`int VTL::attr3`	zusätzliche Eigenschaft 3

Der Zeitindex ist dabei beim ersten Wert innerhalb einer Sounddatei 0 und von da an ansteigend, wobei der letzte Wert von Fall zu Fall verschieden sein kann. Der eigentliche Wert ist eine Zahl zwischen 0 und 255, da FMS zur Zeit noch keine andere Byterate als ein Byte pro Wert unterstützt. Die verschiedenen Linienstile sind in der folgenden Tabelle aufgeführt:

Nr.	Bedeutung
0	letzter Wert
1	Gerade
2	Sinuskurvenabschnitt, gerader Anfang
3	Sinuskurvenabschnitt, gekrümmter Anfang
4	Gauss-Kurven-Abschnitt 1, bis zur y-Achse
5	Gauss-Kurven-Abschnitt 2, von der y-Achse an
6	Parabelabschnitt, positives a
7	Parabelabschnitt, negatives a

Für Linienstile 4 - 7 werden offensichtlich noch weitere Parameter benötigt. Diese sind in den `attr`-Variablen gespeichert. Bei den Parabelabschnitten enthalten `attr1` und `attr2` den Start- bzw. Endwert des zu berechnenden Abschnittes auf der x-Achse, `attr3` zusätzlich die Amplitude, die bei symmetrischen Parabelabschnitten benötigt wird. Für Gauss-Kurven muss nur `attr1` als x-Start- bzw. Endwert angegeben werden.

Auf diese `private`-Variablen kann über die Zugriffsfunktionen (get* und set*) zugegriffen werden.

PBS

PBS-Objekte (Pitch-Beats-Style) enthalten die nötigen Informationen über eine Note im FMS-Midi-Modus, die folgendermaßen aus den verschiedenen Variablen besteht:

Variable	Bedeutung
`float PBS::pitch`	Tonhöhe
`float PBS::beats`	Notenwert
`int PBS::style`	Instrumentenstil

Die Tonhöhe wird beim Einlesen berechnet, und gibt das Verhältnis zwischen Kammerton a und Frequenz des jeweiligen Tones an. Der Notenwert beträgt bei einer ganzen Note 1 und bei anderen Notenwerten entsprechende Bruchteile. Der Instrumentenstil schließlich gibt an, welchem beim Abspielen festgelegten Instrument (bestehend aus Oszillogramm + Lautstärkenverlauf) die Note zugeordnet wird. Dabei ist ein Instrument von 0 und 31 möglich.

VTLP

VTLP ist abgeleitet aus VTL und enthält zusätzlich einen `VTLP*` um verkettete Listen zu ermöglichen.

PBSP

Ähnlich verhält es sich mit PBSP, nur dass dieses Objekt nicht von PBS abgeleitet werden konnte, da der Einlesevorgang einige Änderungen am Klassendesign benötigt. Es wäre wünschenswert, dies in Zukunft zu ändern.

9.2.3 fmifstream.h

Das in fmifstream enthaltene Objekt `FMData` dient zum Lesen und Formatieren von FMS-Sounddateien.

Um fmifstream zu benutzen muss der Header `fmifstream.h` eingebunden und `fmifstream.o` mit ins Programm gelinkt werden.

Im Allgemeinen sollte es nicht nötig sein, die fmifstream-Funktionen manuell zu benutzen, sondern es empfiehlt sich vielmehr die Anwendung der komfortableren FMPlayer-API.

FMData

In der Headerdatei `include/fmifstream.h` ist die Klasse FMData enthalten. Sie beinhaltet Funktionen zum Einlesen und Formatieren von in Dateien gespeicherten FMS-Sounddaten.

Normalerweise werden die FMData-Funktionen indirekt über FMPlayer aufgerufen, aber es ist natürlich auch möglich, sie direkt zu verwenden. Allerdings ist dies zeitaufwändiger und fehleranfälliger, weshalb diese Möglichkeit nur in Ausnahmefällen genutzt werden sollte. Vielmehr könnte es sinnvoll sein, die benötigten zusätzlichen Funktionen zu FMPlayer hinzuzufügen. Dazu ist allerdings natürlich auch Kenntnis über fmifstream von Nöten.

Um dem FMData-Objekt a einen Dateinamen „out.fms“ zuzuweisen, diese zu öffnen und auszulesen ist ganz einfach die Anweisung

```
a.init("out.fms");
```

Diese Anweisung kann in einen try-catch-Block eingeschlossen werden, um eventuelle Fehler zu analysieren. Dabei werden als Fehlerindikator Integerwerte verwendet. Zu deren Analyse können die Konstanten `FMD_OPEN_ERR` (Fehler beim Öffnen), `FMD_READ_ERR` (Fehler beim Lesen) und `FMD_FORM_ERR` (Fehler im Dateiformat) verwendet werden.

Nach der erfolgreichen Initialisierung ist die Benutzung der Daten möglich. Dabei ist der Zugriff auf die Bezeichnung, den Speicherungsmodus, die Anzahl der Werte und Bytes pro Wert mit den folgenden Funktionen möglich:

```
char * FMData::getName();

int FMData::getMode();

int FMData::getVlen();

int FMData::getBytes();
```

Die Bezeichnung ist dabei ein zurückgegebener Textstring, der bei der Erstellung der Datei gewählt wurde. Der Modus beschreibt die Art der Speicherung: 0 steht für die Speicherung aller Werte, 1 für die Speicherung einiger Werte mit Angabe der jeweiligen Zeitindices und des Types der Verbindungslinie zwischen ihnen und 2 für die Speicherung im Midi-Modus von FMS. Dabei ist es wichtig, je nach Speicherungsmodus auf die richtigen Werte in fmifstream zuzugreifen. Im Modus 0 befindet sich einfach ein Array von `unsigned chars` hinter dem Pointer `fmifstream::val`. Auf den $n$ten Wert könnte in unserem Beispiel als `a.val[n]`, beginnend mit $n = 0$ zugegriffen werden. Der letzte Wert hat dabei die Ordnungszahl `a.getVlen()-1`. Im Modus 1 dagegen sind die Werte in einer Reihe von VTL-Objekten eingelesen, die sich hinter dem Pointer `fmifstream::vtl` verbergen. Im Modus 2 schließlich sind die einzelnen Tonwerte in je einem PBS-Objekt gespeichert, der Pointer `fmifstream::pbs` zeigt auf das erste. Die VTL- und PBS-Klassen sind im Abschnitt über `vtl.h` beschrieben.

9.2.4 fmlanguage.h

Die fmlanguage-Header und -cpp-Dateien enthalten von den FMS-Programmen verwendete Textstrings. Diese können problemlos in verschiedene Sprachen übersetzt werden, wenn sich ein freiwilliger Übersetzer findet. Vorerst sind sie nur in Deutsch und Englisch verfügbar.

Die jeweiligen Header müssen eingebunden und die dazugehörigen Object Code - Dateien mitgelinkt werden.

9.2.5 fmofstream.h

Wenn man selbst Programme zur Erstellung von FMS-Sounddateien schreiben will, ist die Verwendung von fmofstream ratsam. Der Header enthält einige Funktionen, die hier dokumentiert werden sollen.

`int fmopen(char *)`: Diese Funktion öffnet die Datei mit dem als Parameter übergebenen Dateinamen und gibt die systeminterne Kennziffer für diese Datei zurück. Diese sollte im Programm als Variable gespeichert werden, da sie beim Schreiben der Werte benötigt wird. Es ist zu beachten, das die folgenden Funktionen etwaige schon vorhandene Dateien einfach überschreiben.

`int mode(char, int)`: Die Funktion schreibt die Kennziffer für den Speicherungsmodus an den Anfang der geöffneten Datei. Es müssen eine char-Variable als Modus-Kennziffer (0-2) und ein Integer als systeminterne Kennziffer für die geöffnete Datei übergeben werden.

`bool name(char *, int)`: Schreibt die Klangbezeichnung zusätzlich anderer benötigter Parameter in die Datei. Die Übernommenen Parameter sind die Bezeichnung und die systeminterne Kennziffer für die geöffnete Datei.

`bool values(...)`: Überladene Funktion, die je nach Speicherungsmodus unterschiedliche Variablen als Parameter übernimmt:

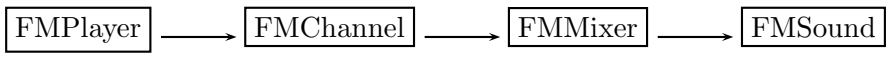
- Modus 0: Die Funktion übernimmt einen Integerwert für die Anzahl der Werte, einen für die Anzahl der Bytes (der 1 sein muss), die systeminterne Kennziffer für die geöffnete Datei und einen `unsigned char *` auf das Wertearray.
- Modus 1: Die Funktion übernimmt einen Integerwert für die Anzahl der Werte, einen für die Anzahl der Bytes (der 1 sein muss), die systeminterne Kennziffer für die geöffnete Datei und einen `VTLP *` auf das erste Element der verketteten Liste.
- Modus 2: Die Funktion übernimmt einen Integerwert für die Anzahl der Noten, die systeminterne Kennziffer für die geöffnete Datei und einen `PBSP *` auf das erste Element der verketteten Liste.

`bool cleanup(int)`: Schließt die Datei mit der übergebenen systeminternen Kennziffer.

Es ist geplant, alle diese Funktionen und Objekte in einer bequemen API, ähnlich der FMPlayer-API, zu vereinen und zusätzliche Fähigkeiten einzubauen, die die Arbeit mit fmofstream weiter erleichtern und weniger fehlerintensiv machen. Um fmofstream zu benutzen, muss `fmofstream.h` eingebunden und das Programm mit `fmofstream.o` verlinkt werden.

9.2.6 fmplayer.h

Ein wirklich praktisches Werkzeug zur Berechnung und zum Abspielen von Werten auf der Basis von FMS-Sounddateien stellt die FMPlayer-API dar. Dabei genügt es, ein FMPlayer-Objekt zu erzeugen und die eingebauten Funktionen zu nutzen. Über dieses Objekt kann man auf alle FMChannel-, FMMixer- und FMSound-Objekte zugreifen. Die Hierarchie der verschiedenen Objekte stellt sich folgendermaßen dar:



Das bedeutet, dass jeder FMPlayer mehrere FMChannel enthalten kann, die ihrerseits mehrere FMMixers enthalten können, die wiederum mehrere FMSounds enthalten können. Dabei ist eine FMSound ein Klang mit Eigenschaften wie z.B. der Frequenz, ein FMMixer ein fertig gemischtes „Soundpaket“ aus verschiedenen Klängen mit Eigenschaften wie z.B. der Dauer, ein FMChannel eine Folge von Soundpaketen die hin-

tereinander abgespielt einen Soundkanal ergeben und der FMPlayer ein Objekt mit ein oder zwei Kanälen und globalen Eigenschaften wie der Samplingrate. Die Reihen der FMMixer und FMSounds sind dabei als verkettete Listen implementiert.

Um die im Folgenden dokumentierten Objekte zu nutzen, muss `fmplayer.h` eingebunden und das Programm mit `fmplayer.o`, `fmwav.o`, `fmval.o` und `fmifstream.o` verlinkt werden.

Die in diesem Kapitel enthaltenen Listen von Funktionen und Variablen erheben keinen Anspruch auf Vollständigkeit. Vielmehr sind weniger wichtige Funktionen, die fast nur API-intern verwendet werden, nicht extra aufgeführt.

Datentypen

Die API verwendet vier eigene Datentypen (enumerations), die zuerst beschrieben werden sollen

`PlayMode`: Abspielmodus des FMPlayers; kann `DSP` (Sound-Ausgabe), `WAV` (Ausgabe in Wave-Datei) oder `NONE` (keine Ausgabe) sein

`FileMode`: Verwendung der Sounddatei; kann `Sound` (normaler Klang), `Noise` (Rauschen durch zufällige Haltewerte, die Haltedauer ist nach dem Oszillogramm verteilt), `RandNoise` (wie `Noise`, aber auch Werte sind nach dem Oszillogramm verteilt) oder `Rand` (wie `RandNoise`, Werte werden aber nicht gehalten) sein

`HFMode`: Modus der Hüllkurve; kann `Envelope` (Ausschlag vom Minimalwert wird moduliert), `AmpMod` (Ausschlag vom Mittelwert wird moduliert) oder `Ring` (Ring-Oszillator) sein.

`ModType`: Eigenschaft, die moduliert wird; kann `Frequency` (Frequenz), `Volume` (LautstÃ rke), `MLen` (HaltelÃ nge der Werte im Rauschmodus), `Minimum` oder `Maximum` fÃ r die Schranken einer anderen Modulation oder `Other` fÃ r sonstige Modulationen sein.

FMPlayer

Das übergeordnete Objekt, das ein Abspielgerät darstellt, ist der FMPlayer. Die enthaltenen Funktionen und als `public` deklarierten Variablen sind:

`FMChannel * firstChannel`: Pointer auf den ersten (oder einzigen) Soundkanal

`FMChannel * secondChannel`: Pointer auf den zweiten Soundkanal im Stereo-Modus

`FMChannel * aChannel`: Pointer auf den aktuell konfigurierten Soundkanal

`void switchChannel()`: Schaltet auf den anderen Soundkanal um bzw. erzeugt diesen, falls noch nicht vorhanden. Danach können dessen Eigenschaften und die Eigenschaften seiner Soundpakete konfiguriert werden. Die kanalinternen Pointer

auf aktuelles Soundpaket und aktuellen Sound des anderen Kanals, sowie die paketinternen Pointer des anderen Kanals bleiben erhalten.

`void gotoFirstChannel()`: Schaltet wieder auf den ersten Kanal um, falls man sich im Programmablauf nicht mehr sicher sein sollte, welcher Kanal gerade konfiguriert wird. Die kanalinternen Pointer auf aktuelles Soundpaket und aktuellen Sound des zweiten Kanals (sofern vorhanden) bleiben erhalten.

`FMMixer * aMixer`: Pointer auf das aktuell konfigurierte Soundpaket, das sich immer im aktuell konfigurierten Soundpaket befindet

`void nextMixer(bool=1)`: Schaltet auf das nächste Soundpaket im aktuellen Kanal um bzw. erzeugt dieses, wenn es nicht besteht und nicht 0 als Parameter angegeben wurde. Die Funktion setzt auch den kanalinternen Pointer auf das neue Soundpaket und seinen ersten Sound. Der paketinterne Pointer auf den aktuellen Sound im vorher aktuellen Paket bleibt erhalten.

`void gotoFirstMixer()`: Schaltet zurück auf das erste Soundpaket im aktuellen Kanal.

`FMSound * aSound`: Pointer auf den aktuell konfigurierten Sound, der sich immer im aktuellen Paket des aktuellen Kanals befindet

`void nextSound()`: Schaltet auf den nächsten Sound im aktuellen Soundpaket um bzw. erzeugt diesen, wenn er nicht besteht. Die Funktion setzt auch den kanalinternen und paketinternen Pointer auf den aktuellen Sound.

`void gotoFirstSound()`: Schaltet zurück auf den ersten Sound im aktuellen Soundpaket des aktuellen Kanals.

`void resetPlayer()`: Entfernt alle untergeordneten Objekte aus dem Speicher und führt den Konstruktor erneut aus.

`void setRate(int)` & `int getRate()`: Zugriffsfunktionen für die Samplingrate; diese wird standartmäßig auf 44100 gesetzt, sollte aber, besonders bei grafischer Darstellung der Werte, oftmals besser andere Werte einnehmen.

`void setBytes(int)` & `int getBytes()`: Zugriffsfunktionen auf die Bytezahl der Soundausgabe

`void setPlayMode(PlayMode)` & `PlayMode getPlayMode()`: Zugriffsfunktionen für die Abspielmodus, siehe oben

`void setWavFile(char *)`: Legt den Dateinamen der Wave-Ausgabedatei fest und setzt gleichzeitig den PlayMode auf WAV.

`void openFiles()`: Öffnet alle Dateien und liest sie ein.

`void compute()`: Berechnet alle Werte.

`void playInit()`: Initialisiert den Abspielprozess.

`void playValue()`: Spielt einen einzelnen Wert ab.

`void playCleanup()`: Beendet den Abspielprozess.

`void play()`: Fasst die drei Abspielfunktionen zusammen.

`void syncPlayInit()` & `void syncPlayValue()` & `void syncPlay()`: analoge Funktionen fÃ r synchronen Berechnungs- und Abspielprozess

FMChannel

Ein FMChannel ist eine Serie von Soundpaketen, die hintereinander abgespielt werden. Folgende Funktionen sind interessant:

`void setPlay(bool)` & `bool play()` Zugriffsfunktionen auf die Bool-Variable, die bestimmt, ob der Kanal abgespielt werden soll

`void setRepeat(int)` & `int getRepeat()` Zugriffsfunktionen auf die Anzahl der Wiederholungen des Kanals; -1 bedeutet Endloswiederholung

`bool syncFin()`: gibt `TRUE` zurück, wenn im synchronen Abspielmodus das Abspielen zeitlich beendet ist

`void ptrInc()`: springt zum nächsten Soundwert im Speicher (asynchroner Modus)

FMMixer

Ein FMMixer ist ein abspielbereites „Soundpaket“ aus gleichzeitig abgespielten Klängen, das folgende Eigenschaften (Variablen) und Funktionen hat:

`FMChannel * channel`: Pointer auf den Soundkanal, zu dem das Paket gehört

`FMMixer * next`: Pointer auf das darauffolgende Soundpaket, um eine verkettete Liste zu ermöglichen; ist `FALSE`, falls es kein nächstes Soundpaket gibt.

`FMMixer * self`: Pointer auf jenes Soundpaket, von dem das jeweilige Paket nur eine Kopie darstellt; ist `this`, falls das Paket keine Kopie ist.

`FMModulator * hvol`: Pointer auf das FMModulator-Objekt, nach dessen Oszillogramm sich die Gesamtlautstärke des Pakets ändert; ist 0, wenn das für das Paket nicht der Fall ist.

`void setHVol()`: Erzeugt ein FMModulator-Objekt aus dem Pointer, wenn das nicht schon geschehen ist. Die Funktion sollte vor jedem Zugriff auf `FMMixer::hvol` benutzt werden, um sicherzustellen, dass `hvol` existiert.

`FMSound * aSound`: Pointer auf den aktuell konfigurierten Sound

`float * values`: Pointer auf das Array der berechneten Werte

`void nextSound()` & `void firstSound()`: siehe FMPlayer

`void setTime(float)` & `float getTime()`: Zugriffsfunktionen auf die Dauer des Soundpaketes (in Sekunden)

`void setVolume(float)` & `float getVolume()`: Zugriffsfunktionen auf die Gesamtlautstärke; es werden Werte zwischen 0 und 1 angenommen, mit denen alle Soundwerte multipliziert werden.

`void setTVol(float)` & `float getTVol()`: Zugriffsfunktionen auf die Summe der Lautstärken aller Klänge

`void setRepeat(float)` & `float getRepeat()`: Zugriffsfunktionen auf die Anzahl der Wiederholungen des Soundpakets

`void setSelf(int)` & `void setSelf(FMMixer *)`: Setzt fest, von welchem Paket das aktuelle Paket eine Kopie ist. Die überladene Funktion kann einen Integerwert als Nummer des Originals übernehmen, wobei bei negativem Wert im anderen Soundkanal nach dem Original gesucht wird. Anderenfalls ist ein `FMMixer *` auf das Original als Argument möglich.

`void deleteMe()`: entfernt dieses Soundpaket

`void computeInit()` & `computeMalloc()` & `computeValues()`: dreistufiger Berechnungsablauf, dessen zweite und dritte Stufe im synchronen Modus nicht benötigt werden

Wenn von einer „Kopie eines anderen Soundpakets“ die Rede ist, so bedeutet das nur, dass die Werte nicht vom Paket selbst berechnet werden, sondern `values` einfach mit dem Wertepointer des Originalpakets gleichgesetzt wird. Die einzige Eigenschaft, die dann noch eine Auswirkung hat ist die Anzahl der Wiederholungen, da sich diese direkt beim Abspielen auswirkt.

Auf die Funktionen und Variablen des aktuellen Soundpakets kann über `FMPlayer::-aMixer->` zugegriffen werden.

FMFile

FMFile ist eine Basisklasse, von der die Klassen FMSound und FMModulator abgeleitet sind. Deshalb enthält sie alles, was für die Berechnung eines Wertes eines Oszillogramms aus einer FMS-Datei nötig ist:

`FMMixer * mixer`: Pointer zum FMMixer-Objekt, zu dem die Datei gehört

`FMModulator * ffreq`: Pointer zum FMModulator-Objekt, nach dessen Oszillogramm die Frequenz schwankt

`void setFFreq()`: Erzeugt ein FMModulator-Objekt aus dem Pointer, wenn das nicht schon geschehen ist. Die Funktion sollte vor jedem Zugriff auf `FMFile::-ffreq` benutzt werden, um sicherzustellen, dass `ffreq` existiert.

`void unsetFFreq()`: entfernt eventuelles ffreq-Objekt

`FMModulator * hvol`: Pointer zum FMModulator-Objekt, nach dessen Oszillogramm die Lautstärke schwankt

`void setHVol()`: Erzeugt ein FMModulator-Objekt aus dem Pointer, wenn das nicht schon geschehen ist. Die Funktion sollte vor jedem Zugriff auf `FMFile::hvol` benutzt werden, um sicherzustellen, dass `hvol` existiert.

`void unsetHVol()`: entfernt eventuelles hvol-Objekt

`void setFilename(char *` & `char * getFilename()`: Zugriffsfunktionen auf den Dateinamen

`void setFreq(float)` & `float getFreq()`: Zugriffsfunktionen auf die Frequenz (in Hz)

`void setVolume(float)` & `float getVolume()`: Zugriffsfunktionen auf die Lautstärke

`void setFileMode(FileMode)` & `FileMode getFileMode()`: Zugriffsfunktionen auf den `FileMode` (siehe oben)

`void setMLen(float)` & `float getMLen()`: Zugriffsfunktionen auf die mittlere Anzahl der Halteschritte bei `Noise`-Filemodi

`void setAtime(float)`: Zugriffsfunktion zum Setzen des aktuellen Zeitindexes

`void incAtime()`: Funktion, die nach der Berechnung eines Wertes den Zeitindex - je nach Samplingrate und Frequenz - hochzählt bzw. zurücksetzt

`void openFile()` & `void closeFile()`: Funktionen zum Öffnen und Auslesen bzw. zum Schließen der Datei

`protected float value(bool=1)`: Gibt den aktuellen Wert der Kurve (von 0 - 255) zurück. Wenn keine 0 als Parameter übergeben wird, wird der Zeitindex weitergezählt. Die Funktion ist `protected` und kann deshalb nur von FMFile-Funktionen oder aus abgeleiteten Klassen (FMSound, FMModulator) aufgerufen werden.

`float real_value()`: Gibt Wert ohne Berücksichtigung von Modulationen zurück ohne den Zeitindex weiterzuzählen.

`float setCopyFrom(FMFile *copy)`: Klont die Eigenschaften der mittels Pointer angegebenen Instanz von FMFile.

Auf die FMFile-Funktionen und Variablen des aktuellen Sounds kann über `FMPlayer::-aSound->` zugegriffen werden.

FMSound

FMSound ist ein von FMFile abgeleitetes Objekt, das für einen Klang steht. Es hat deshalb folgende zusätzliche Eigenschaften:

`FMMidi * midi`: Pointer zum FMMidi-Objekt, das im Falle einer Datei im Modus 2 zum Einsatz kommt

`void setMidi()`: Erzeugt ein FMMidi-Objekt aus dem Pointer, wenn das nicht schon geschehen ist. Die Funktion sollte vor jedem Zugriff auf `FMSound::midi` benutzt werden, um sicherzustellen, dass `midi` existiert. Wird eine FMS-Midi-Datei eingelesen, so erkennt FMS dies jedoch automatisch und ruft `FMSound::midi` vor dem Abspielen auf.

`FMSound * next`: Pointer auf den darauffolgenden Sound im gleichen Soundpaket, um eine verkettete Liste zu ermöglichen; ist `FALSE`, falls es keinen nächsten Sound gibt.

`float value(bool=1)`: Wählt die richtige Funktion zur Bestimmung des aktuellen Soundwerts (entweder `FMFile::value()` oder `midi->MidiValue()`) und gibt deren Ergebnis zurück. Außerdem wird die absolute Lautstärke und Hüllkurvenlautstärke der Datei miteinbezogen. Wird `FALSE` als Parameter übergeben, so zählt die Funktion nicht die jeweiligen aktuellen Zeitindices weiter.

Auf die FMSound-Funktionen und Variablen des aktuellen Sounds kann über `FMPlayer::aSound->` zugegriffen werden.

FMModulator

Immer wenn eine Frequenz oder Lautstärke einem Oszillogramm folgend schwanken soll, ist ein FMModulator-Objekt die praktikabelste Lösung. Zusätzliche Modulationen von Rauscheigenschaften und die rekursive Modulation (z.B. Amplitudenmodulation der Frequenzmodulation) sind ebenfalls möglich. Das Objekt ist von FMFile abgeleitet und enthält zusätzlich folgende Variablen und Funktionen:

`float HFValue(bool=1)`: Die Funktion gibt den aktuellen Wert zwischen dem Minimal- und Maximalwert zurück. Wenn keine 0 als Parameter übergeben wird, wird der Zeitindex weitergezählt.

`float HVolVal(float)`: Führt die Amplitudenmodulation mit dem übergebenen Soundwert durch unter Berücksichtigung der verschiedenen Modulationsmodi (`HFMode::Envelope`, `HFMode::AmpMod`, `HFModeRing`).

`void setMin(float)` & `float getMin()`: Zugriffsfunktionen auf den Minimalwert, der zurückgegeben werden soll. Bei Oszillogrammen, die die Lautstärke bestimmen, so genannten Hüllkurven oder Envelopes, muss dieser Wert kleiner als 1 und mindestens 0 sein.

`void setMax(float)` & `float getMax()`: Zugriffsfunktionen auf den Maximalwert, der zurückgegeben werden soll. Bei Hüllkurven muss dieser Wert kleiner als 1 und mindestens so groß wie der Minimalwert sein.

`void setIter(float)` & `float getIter()`: Zugriffsfunktionen auf die Anzahl der Wiederholungen der Hüll- oder Frequenzkurve während einem Durchgang des Soundpakets.

`void setMinIter(float)` & `float getMinIter()` & `void setMaxIter(float)` & `float getMaxIter()`: Zugriffsfunktionen auf die minimale bzw. maximale Anzahl der Wiederholungen dieser Hüll- oder Frequenzkurve bei Nutzung einer Modulationskurve (`ffreq`) für die Frequenz.

`void setHFMode(HFMode)` & `float getHFMode()`: Zugriffsfunktionen auf den HF-Mode (siehe oben)

`void setCopyFrom(FMModulator *f)`: siehe FMFile

Auf die Funktionen der aktuellen Sound-Hüll- und Frequenzkurve kann über `FMPlayer::aSound->hvol` bzw. `FMPlayer::aSound->ffreq` zugegriffen werden. Bei der Soundpaket-Hüllkurve ist `FMPlayer::aMixer->hvol` zu verwenden. Natürlich muss dazu das jeweilige FMModulator-Objekt existieren.

FMMidi

Das FMMidi-Objekt enthält alle besonderen Funktionen, die beim Abspielen einer FMS-Midi-Datei benötigt werden. Als da wären:

`void initMidi()`: Öffnet alle Dateien, die für das Abspielen benötigt werden.

`void setMidiI(int,char *)`: Zugriffsfunktion auf die Dateinamen der Midi-Instrumente.

`void setMidiH(int,char *)`: Zugriffsfunktion auf die Dateinamen der Midi-Instrument-Hüllkurven.

`void setMidiHMin(int nr, float dhmin)` & `float getMidiHMin(int)`: Zugriffsfunktionen auf die Minimalwerte der Midi-Instrument-Hüllkurven.

`void setMidiHMax(int nr, float dhmax)` & `float getMidiHMax(int)`: Zugriffsfunktionen auf die Maximalwerte der Midi-Instrument-Hüllkurven.

`void setMidiHIter(int nr, float dhiter)` & `float getMidiHIter(int)`: Zugriffsfunktionen auf die Anzahl der Wiederholungen der Midi-Instrument-Hüllkurven.

`void setBps(float)` & `float getBps()`: Zugriffsfunktionen auf die Beats Per Second, die Funktionen werden automatisch aufgerufen.

`float MidiValue(bool=1)`:Die Funktion gibt den aktuellen Midi-Soundwert zurück. Wenn keine 0 als Parameter übergeben wird, wird der Zeitindex weitergezählt.

Dabei muss immer beachtet werden, dass bei Optionen die auf ein bestimmtes Midi-Instrument zutreffen immer die Nummer dieses Instruments als erster Parameter angegeben werden muss. Dies wurde in obiger Aufzählung nicht einzeln beschrieben.

compLock

In den Objekten `FMFile` und insbesondere `FMModulator` gibt es Eigenschaften, die zu Beginn des Abspielens berechnet werden müssen (z.B. Modulationsfrequenz aus Anzahl der Wiederholungen und Dauer des `FMMixer`s.

Über die Funktionen `compLock()` von `FMFile` und `FMModulator` lässt sich der Berechnungsstatus abfragen. Ist der Rückgabewert `TRUE`, so sind die zu berechnenden Eigenschaften nicht auf dem aktuellen Stand. Diese Angaben werden im asynchronen Modus auch genutzt, um überflüssige Wiederberechnung bereits abspielbereiter Sequenzen beim Wiederabspielen zu vermeiden.

Beispiele

Es ist empfehlenswert, sich durch `fmplay.cpp` zu arbeiten, um einen Überblick über die Benutzung der API zu bekommen. Viele der beschriebenen Funktionalitäten werden dabei auf einfache Weise genutzt. Die Beispielprogramme (*test.cpp) können weiteren Einblick liefern.

9.2.7 fmwav.h

Die nicht ganz triviale Aufgabe, Rohwerte im Speicher in eine Wave-Datei zu speichern und auch wieder zu lesen, kann mit Hilfe von fmwav gelöst werden. In fmwav sind nur vier Funktionen enthalten: `writeWavHeader()` zum Öffnen der Datei und Schreiben des so genannten Wav-Headers, wozu es als Parameter den Dateinamen, die Anzahl der Werte, die Samplingrate, die Anzahl der Kanäle (0 für Mono, 1 für Stereo) und die Anzahl der Bytes pro Wert benötigt, außerdem `writeWavByte()` zum Schreiben eines einzelnen Bytes, das als Parameter übergeben wird, in die Wave-Datei und `cleanupWav()`, das keine Parameter benötigt zum Schließen der geschriebenen Wave-Datei.

`wavFile readWav(char *)`: Die Funktion liest die angegebene WAV-Datei ein und gibt ein wavFile-Struct zurück, auf dessen Attribute `length`, einem Integerwert, der die Anzahl der Werte angibt und `data` einem `float *` auf das Array der Werte. Dieses Struct muss wieder mit `delete` aus dem Speicher entfernt werden, um memory leaks zu vermeiden.

9.2.8 version.h

Der version-Header enthält nur zwei Informationen: die Versionsnummer der vorliegenden FMS-Version und den Debug-Level. Setzt man den Debug-Level auf 1 oder 2 und rekompiliert, so werden beim Programmaufruf mehr oder weniger ausführliche Debug-Informationen ausgegeben. Standartmäßig steht der Debug-Level auf 0 und es werden deshalb keine Informationen ausgegeben. Um die Informationen über Versionsnummer

und Debug-Level zu nutzen, muss nur der Header `include/version.h` eingebunden werden.

9.3 Dateiformat

FMS unterhält drei verschiedene Dateiformate, die folgendermaßen aufgebaut sind:

Modus 0: jeder Wert

Das Format 0 (in der Kommunikation mit dem User Format 1 genannt) ist die einfachste denkbare Möglichkeit zur Speicherund eines Oszillogrammes. Es wird dabei einfach jeder Wert (natürlich mit einem bestimmten, mehr oder weniger groben, Raster) in die Datei geschrieben. Da für diese Art der Speicherung relativ viel Festplattenplatz benötigt wird, besonders wenn das Ergebnis genau sein soll, ist es nur noch in Sonderfällen empfehlenswert, Dateien im Format 0 zu speichern. Das komplette Format baut sich byteweise wie folgt auf, wobei Werte in eckigen Klammern für beispielhafte Stellen stehen:

Stelle	Bedeutung	Beispiel
0	Modus, beim Modus 1 natürlich immer 1	1
1	Anzahl der Buchstaben der Bezeichnung	5
2[-6]	Bezeichnung	Sinus
[7]	Anzahl der gespeicherten Werte, 65536er-Stelle	0
[8]	Anzahl der gespeicherten Werte, 256er-Stelle	0
[9]	Anzahl der gespeicherten Werte, 1er-Stelle	5
[10]	Anzahl der Bytes pro Wert	1

Die komplette Anzahl der Werte setzt sich also aus drei Stellen eines 256er-Systems zusammen. Bisher wird keine andere Bytezahl pro Wert als 1 unterstützt.

Modus 1: einzelne Werte, verbunden mit Linien verschiedener Art

Das Format 1 (in der Kommunikation mit dem User Format 2 genannt) ist in Hinsicht auf Speicherplatz und Wiedergabe von Kurven sehr viel besser geeignet. Es werden nur wenige Werte gespeichert, zusammen mit ihrem Zeitindex und dem darauffolgenden Linienstil. Da die eigentlichen Ausgabewerte zur Laufzeit berechnet werden, benötigt das Abspielen mehr Prozessorleistung, allerdings ist die Platzersparnis und die bessere Kurvenwiedergabe das wert. Das Format setzt sich byteweise wie folgt zusammen:

Stelle	Bedeutung	Beispiel
0	Modus, beim Modus 1 natürlich immer 1	1
1	Anzahl der Buchstaben der Bezeichnung	5
2[-6]	Bezeichnung	Sinus
[7]	Anzahl der gespeicherten Werte, 65536er-Stelle	0
[8]	Anzahl der gespeicherten Werte, 256er-Stelle	0
[9]	Anzahl der gespeicherten Werte, 1er-Stelle	5
[10]	Anzahl der Bytes pro Wert	1
[11-35]	Werte	siehe unten

Die Anzahl der Bytes pro Wert gibt nur die Bytes für den eigentlichen Wert, nicht aber die für Zeitindex und Linienstil an. Dabei setzt sich ein Wertekomplex meist aus vier Bytes wie folgt zusammen:

Stelle	Bedeutung	Beispiel
0	Zeitindex-Offset seit letztem Wert, 256er-Stelle	0
1	Zeitindex-Offset seit letztem Wert, 1er-Stelle	10
2	Wert	255
3	darauffolgender Linienstil	siehe unten

Bei Linienstil 6 und 7 werden die zusätzlichen Parameter einfach hinter den oben beschriebenen Wertekomplex in die Datei geschrieben.

Die 8 Linienstile sind:

Nr.	Bedeutung
1	gerade Linie
2	Sinuskurvenabschnitt, linearer Anfang
3	Sinuskurvenabschnitt, gekrümmter Anfang
4	Gauss-Kurven-Abschnitt 1, bis zur y-Achse
5	Gauss-Kurven-Abschnitt 2, von der y-Achse an
6	Parabelabschnitt, positives a
7	Parabelabschnitt, negatives a
0	letzter Wert

Durch diese Linienstile werden die angegebenen Punkte verbunden.

Modus 2: FMS-Midi

Modus 2 besteht aus einer Aneinanderreihung von Informationen über Einzeltöne, also am ehesten einer klassischen Midi-Datei. Binär ist das Format folgendermaßen aufgebaut:

Stelle	Bedeutung	Beispiel
0	Modus, beim Modus 2 natürlich immer 2	2
1	Anzahl der Buchstaben der Bezeichnung	14
2[-15]	Bezeichnung	Menuett - Bach
[16]	Anzahl der gespeicherten Töne, 256er-Stelle	0
[17]	Anzahl der gespeicherten Töne, 1er-Stelle	83
[18-182]	Töne	siehe unten

Ein Ton wird in nur 2 Bytes gespeichert. Diese Komprimierung benötigt „Byte-Sharing“ zwischen verschiedenen Attributen, weshalb das Format sehr kompliziert in einer Tabelle darzustellen ist. Interessierte können die Bitkonfigurationen `fmofstream.cpp` entnehmen.

23.2.2003: Daniel Grün

Teil IV

Anhang

Kapitel 10

Glossar

Amplitudenmodulation: Variation der Schwingungsamplitude

API: Application Programming Interface, Objekte und Funktionen zum Einbau in Programme

Branch: engl. Zweig, Teil des Frocor-Sourcecodes in den Unterverzeichnissen von `frocor/branch/`

Commit: Prozess des Hochladens von Änderungen an Dateien in svn; Synonym zu Version

Frequenzmodulation: Variation der Schwingungsfrequenz

Midi-Mapping: Ausgabe von Mididaten ohne Unterstützung eines klassischen Synthesizers

Merge: Prozess des Wiederein- und Zusammenfügens von Änderungen in den ursprünglichen Sourcecode

Oszillogramm: Visualisierung von Schwingungsvorgängen, bei einem Sinuston z.B. der Graph der Sinusfunktion von 0 bis 360

Ringmodulation: Spezialform der Amplitudenmodulation

Samplingrate: Anzahl der pro Sekunde über die Soundkarte ausgegebenen Werte

Kapitel 11

Dank

Wir danken

Herrn Henke für die Ermöglichung eines solchen Projektes im schulischen Umfeld

Herrn Deffner für die Anregungen und Unterstützung

den Teilnehmern des Schülerseminars im Studio für Elektronische Musik und Computermusik für ihre Ideen

allen Programmierern, auf deren Arbeit wir unsere Programme aufgebaut haben

Ihnen, weil Sie sich das hier bis zum Ende durchgelesen haben

Literaturverzeichnis

[1] Homepage des Seminars für Computer, Musik und neue Medien an der MH Stuttgart
`www.ccteam-bw.de`

[2] MagicPoint, ein alternatives Präsentations-Tool:
`www.member.wide.ad.jp/wg/mgp`

[3] SVN, das für FROCOR verwendet Version Control System:
`www.subversion.tigris.org`

[4] GNU Emacs, der Editor, der sogar einen eingebauten Psychiater hat:
`www.gnu.org/software/emacs/emacs.html`

[5] VI Improved, der Editor für alle, die wie Fabian keinen Wert auf eingebaute Psychater legen: `www.vim.org`

[6] GNU General Public License: `www.gnu.org/copyleft/gpl.html`

[7] nur temporär verfügbare Online-Version des FROCOR-Repositories:
`www.fstoehr.homelinux.net`

[8] FMS Download-Adresse: `www.sourceforge.net/projects/fmsynth`

[9] e-mail-Adressen der Autoren:
`daniel_gruen@web.de`
`fabian@fstoehr.homelinux.net`

[10] SDL: `www.libsdl.org`

[11] SDL_gfx: `www.ferzkopp.net/Software/SDL_gfx-2.0`

[12] Qt/Trolltech: `www.trolltech.com`

[13] z.B. das FMS-DSP-HowTo: `www.fmsynth.sourceforge.net/dsp.dvi`

21.01.2005: Fabian Stöhr, Daniel Grün