Proposal

Norman Feske

5. Februar 2002

Inhaltsverzeichnis

1 Einführung

Dieser Text soll einige meiner Ideen zum Entwurf einer grafischen Benutzeroberfläche für Drops vorstellen. Er enthält einige Kapitel, die nichts unmittelbar mit dem Diplom-Thema zu tun haben, aber Einblick auf den Hintergrund einiger Ideen gewähren. Somit soll dieser Text auch mich und mein bisheriges Schaffen ein wenig vorstellen und dem Leser vermitteln, daß meine Ideen nicht aus der Luft gegriffen, sondern das Ergebnis jahrelanger Suche, sind :-)

Der eigentliche Inhalt des 'Proposals' befindet sich im Kapitel 'Konzept'.

1.1 Erfahrungen mit Benutzeroberflächen

Dieses Kapitel beschäftigt sich weniger mit technischen Details von Fenstersystemen, als mit der Betrachtung verschiedener Benutzerkonzepte. Beim Entwurf einer neuen Benutzeroberfläche möchte ich diese Konzepte im Auge behalten - auch wenn vieles, des hier Beschriebenen über den unmittelbaren Rahmen des Diplomthemas hinausgeht.

Im Laufe der Zeit hatte ich die Gelegenheit eine Vielzahl unterschiedlicher Benutzeroberflächen kennenzulernen. Besonders hervorzuheben sind alternative Betriebssystemplattformen. Oft findet man auf den 'kleinen' Plattformen sehr kreative und innovative Benutzerkonzepte. Im folgenden werde ich auf einige mehr oder weniger bekannte Plattformen eingehen und die Dinge hervorheben, die Einfluß auf Designkriterien einer neuen Benutzeroberfläche nehmen sollten.

Nachfolgend verwende ich häufig die Begriffe 'Betriebssystem' und 'Oberfläche' synonym. Die Oberfläche eines Betriebssystems stellt in vielen der hier aufgeführten Fälle ein wesentliches Element des Betriebssystems dar. Desweiteren ist die Oberfläche der Teil des Betriebssystems, mit dem ich mich hier ausschließlich beschäftige.

1.1.1 X-Windows

Die (meines Erachtens) bedeutenste Eigenschaft von X-Windows ist seine Server-Architektur, die es ermöglicht, Applikationen inkl. grafischen Elementen 'remote' zu bedienen.

Nachteil:

- X-Windows wurde von vornherein nicht auf Multimedia ausgelegt und ist dafür (wieder meines Erachtens) wenig geeignet. Diesem Umstand hat sich Xfree4.0 und DRI angenommen - allerdings gestaltet es sich wohl als sehr schwierig, diese Funktionalität im 'Nachhinein' einzubauen.
- X-Windows stellt keinerlei Desktop-Funktionalität (Buttons etc.) zu Verfügung, sondern beschränkt sich auf die Verwaltung von rechteckigen Bildschirmbereichen (Fenster). (dies wird von einigen durchaus auch als Vorteil angesehen) Hieraus resultiert, daß jede komplexe Anwendung eine Library benötigt, um aufwendige Benutzeroberflächen zu realisieren. Der Benutzer findet sich in einem Chaos von verschieden aussehenden und verschieden

zu bedienenden Anwendungen wieder. (Drag&Drop funktioniert meist nur innerhalb einer Library, also nicht zwischen gtk und qt ...)

1.1.2 RiscOS

RiscOS wurde von der britischen Firma Acorn für deren Rechner 'Archimedes' und 'RiscPC' entwickelt. Es hat viele Eigenschaften eines typischen 'Homecomputer-Betriebssystems'. Dies schließt allerdings auch seine herausragende Bedienbarkeit und Benutzerfeundlichkeit ein.

Ein sehr wesentliches Konzept, welches in RiscOS eine geradezu ideale Verwirklichung findet, ist Drag&Drop. Dateien und Daten können zwischen verschiedenen Anwendungen einfach durch 'herüberziehen' eines entsprechenden Symbols ausgetauscht werden. Durch den konsequenten Einsatz von Drag&Drop wird die Benutzeroberfläche wesentlich vereinfacht und dabei die Flexibilität drastisch erhöht. Dies geht soweit, daß es anstatt weniger großer Applikationen viele kleine Tools gibt, die jeweils eine (oder wenige) Funktionen beinhalten. Die Funktions- vielfalt steckt schließlich in der Kombination dieser kleinen Tools, welche durch Drag&Drop' kinderleicht ist.

Das andere wesentliche Element der Benutzeroberfläche von RiscOS ist das Kontext-Menü-Konzept. Durch einen Mausklick auf ein beliebiges Element der Benutzschnittstelle erscheint ein Menü mit den Operationen, die auf dieses Element anwendbar sind. (Dieses Konzept hat sich offenbar sehr bewährt und findet heute auch in anderen - sehr bekannten - Betriebsystemen seine Umsetzung)

Durch den sehr konsequenten Einsatz dieser beiden Konzepte bleiben die Benutzeroberflächen der Programme sehr übersichtlich und einfach. Menüpunkte zum Öffnen von Dokumenten sucht man in den Programmen z.Bsp. vergeblich - die Dateien werden schließlich einfach auf das entsprechende Programmicon gezogen, um sie zu öffnen.

Das Betriebssystem kennt einige grundlegende Datentypen (z.Bsp. Sprites oder Vektorgrafiken) und stellt bestimmte Funktionen für diese Objekte 'von sich aus' zur Verfügung - beispielsweise können Vektorobjekte immer rotiert und skaliert werden - egal in welcher Applikation sie verwendet werden.

Sehr nützlich ist, daß das Betriebssystem automatisch erkennt, welche Programme mit welchen Datentypen arbeiten und so in der Lage ist, diese Programme bei Bedarf automatisch zu laden. Dieses Erkennen erfolgt wärend der Benutzung und wird nicht - wie etwa unter Windows - beim Installieren von Programmen registriert.

Die Anzahl der verschiedenen Elemente der Benutzerobefläche ist sehr überschaubar und bietet dennoch genug Freiheiten, die verschiedensten Programme zu realisieren:

- Icons
- Buttons
- Text

- Eingabefelder (auch mehrzeilig)
- Radiobuttons, Checkbuttons
- Menüs
- Fenster

1.1.3 TOS/MagiC

Die Benutzeroberfläche von TOS (GEM) ist der Benutzeroberfläche von MacOS (Version < 10.0) sehr ähnlich. Das Drag&Drop-Konzept ist nicht ganz so konsequent vertreten, wie unter RiscOS und findet vor allem beim Zugriff auf das Dateisystem Verwendung.

Jedes Programm kann ein Menü beinhalten, welches am oberen Bildschirmrand dargestellt wird, wenn das oberste Fenster dem Programm gehört. Menüs innerhalb von Fenstern sind nicht üblich. Somit erscheinen die Fenster meist sehr aufgeräumt und übersichtlich.

Häufig auftretende Elemente der Benutzeroberfläche sind im Betriebssystem enthalten und werden von allen Programmen verwendet (z.B. Fileselektor-Box, Fehlermeldungen). Diese Elemente sind vom Benutzer Systemweit austausch- und einstellbar. Beispielsweise gibt es eine Fileselektor-Box, die auch Funktionen wie Kopieren, Löschen, Formatieren etc. beinhaltet. Diese Funktionen sind damit automatisch in allen Programmen verfügbar, die eine Fileselektor-Box nutzen.

Eine sehr interessante Entwicklung unter GEM (mit MiNT als Grundlage) ist der GEM-XServer - eine normale Applikation, die das X-Protokoll zur Verfügung stellt. X-Applikationen laufen damit in normalen GEM-Fenstern neben nativen Atari-Programmen.

1.1.4 NextSTEP

Ein wesentliches Element von NextSTEP ist, ähnlich wie bei RiscOS, Drag&Drop.

NextSTEP erkennt nicht nur, welche Programme mit welchen Datentypen arbeiten können, sondern erlaubt sogar das 'exportieren' von Funktionen aus Programmen. Beispielsweise kann ein Wörterbuch die Funktion 'im Wörterbuch nachschlagen' exportieren. Diese Funktion steht anschließend allen Programmen, die irgendetwas mit Text machen (Editor, Webbrowser, Terminal...) zur Verfügung.

Somit besteht ein wesentlicher Unterschied zwischen NextSTEP und anderen Betriebssystemen darin, daß NextSTEP weiß, mit welchen Daten der Benutzer arbeitet.

Das Konzept von NextSTEP ist durchweg objektorientiert (Auf der Basis von Objective C). Daten können in Dokumente eingebettet und 'verlinkt' werden. Daten, die nicht direkt von einer Applikation 'verstanden' werden, werden beim öffnen automatisch konvertiert, wenn ein Konvertierungsprogramm installiert ist. Dieser 'intelligente' Umgang mit Daten und Programmfunktionen führt zu ähnlich schlanken und übersichtlichen Applikationen, wie unter RiscOS. Auch hier entsteht der riesige Funktionsumfang durch die Kombination verschiedener Programme.

Für diese Funktionalität ist es natürlich notwendig, daß NextSTEP erfährt, welches Programm was kann. Diese Fähigkeiten erfährt NextSTEP nicht etwa durch eine Installationsprozedur, sondern durch einen ausgeklügelten Umgang mit 'Programmen'. Ein Programm befindet sich generell in einem eigenen Verzeichnis (Application Folder). Dieses Verzeichnis behinhaltet alles, was ein Programm benötigt und erscheint nach außen, wie eine einfache Datei, die durch einen Doppelklick gestartet werden kann (soetwas, wie ein ausführbares tar-Archiv). Innerhalb dieses Verzeichnisses befinden sich alle Daten, wie Programmicons, Informationen über die Fähigkeiten des Programmes, das ausführbare Programm selbst (evnt. für verschiedene Plattformen, da es NextSTEP für MC68K, X86 und PA-Risc gab) und seine benötigten Daten. Das Installieren eines Programmes bedeutet lediglich, das Icon des Programmes (welches ja in Wirklichkeit der 'application folder' ist) 'irgendwo hin' zu kopieren. Ein Doppelklick auf das Icon startet das Programm. Damit NextSTEP, weiß wo es nach Programmen suchen muß, gibt es spezielle Verzeichnisse: in jedem 'home'-Verzeichnis befindet sich ein 'App/'-Verzeichnis, in das jeder Nutzer 'seine' Programme installieren kann. Desweiteren gibt es in der Wurzel des Dateibaums ein '/App/'- Verzeichnis, in welches der System-Admin Programme installieren kann, die von allen Nutzern nutzbar sein sollen. NextSTEP untersucht immer diese 'App'-Verzeichnisse nach Programmnen und ihren Funktionen. (RiscOS bietet ein ähnliches 'Application Folder'-Konzept)

Zur Beschreibung eines beliebigen Bestandteils der Benutzeroberfläche wird in NextSTEP ein sehr abstraktes Mittel benutzt - Display-Postscript. Die Ausgabe-Einheit von NextSTEP enthält einen Postscript-Interpreter. Jedes Programm, was Ausgaben auf dem Bildschirm machen möchte, muß dies NextSTEP in Form von Postscript-Befehlen mitteilen.

Dieses Vorgehen bietet natürlich einige Vorteile - vor allem im Hinblick auf klassische Büroanwendungen (ein Dokument sieht auf dem Bildschirm genauso aus, wie auf dem Drucker, da beide Geräte 'die gleiche Sprache sprechen'). Sämtlichen Elementen auf dem Bildschirm stehen alle Möglichkeiten von Postscript zur Verfügung (Rotation, Skalierung, Alphablending etc.) Im Hinblick auf Multimedia-Applikationen stößt man hier allerdings auf deutliche Schranken.

1.1.5 Tcl/Tk

Tcl/Tk ist nicht mit einem konkreten Betriebssystem verknüpft. Es stellt vielmehr ein allgemeines Konzept zur Programmierung von Benutzeroberflächen dar.

Tcl/Tk ist aus dem Bedarf entstanden, effizient - und möglichst plattformunabhängig - grafische Benutzeroberflächen zu programmieren. Dabei besteht Tcl/Tk aus zwei Teilen:

- Tcl ist eine leistungsfähige Scriptsprache

- Tk ist eine Erweiterung von Tcl um die Elemente von grafischen Benutzerschnittstellen. Tk beinhaltet sog. 'Widgets', welche die Grundbausteine jeder Benutzeroberfläche darstellen.

Das Erstellen einer grafischen Benutzerschnittstelle mit Tcl/Tk besteht im Schreiben eines Scriptes, welches die Benutzeroberfläche beschreibt. Ursprünglich wurde Tcl/Tk dazu konzipiert, Benutzeroberflächen für C-Programme zu erstellen. Dabei spielt vor allem die Trennung von Benutzeroberfläche und Programmfunktionalität eine wichtige Rolle. Allerdings ist Tcl als Scriptsprache so leistungsfähig, daß viele Applikationen mittlerweie ohne einen C-Teil auskommen.

Die Nutzung von Tcl/Tk bringt viele Vorteile mit sich:

- Konsistenz: Bedienelemente können sehr einfach mit Variablen verknüpft werden. D.h. eine mit einem 'Scale' verknüpfte Variable besitzt stehts den gleichen Wert, wie der 'Scale'. Wenn der 'Scale' verstellt wird, ändert sich die Variable. Wenn der Variablenwert gesetzt wird, ändert sich automatisch der 'Scale'.
- Abstraktion: Tcl/Tk bietet die sprachlichen Mittel, um eine Benutzeroberfläche effizient und aussagekräftig zu beschreiben. Es wird vollständig von Dingen abstahiert, die für die Gestaltung grafischer Benutzeroberflächen unnötig sind - z.B. Variablentypen
- Portierbarkeit
- leistungsfähige Widgets: Viele Widgets bieten schon allein eine große Funktionalität, die von Anwendungen nur noch genutzt werden muß. Beispielsweise beinhaltet jedes Textfeld prinzipiell die Fähigkeiten eines Texteditors, jedes Canvas kann seinen Inhalt als PostScript speichern usw.

1.1.6 Schlußfolgerungen

- Server-Funktionalität
- Abstraktion von C - das Finden einer ausdruckstarken und dennoch einfachen Form, Oberflächen zu beschreiben.
- Modularität (neue Datentypen, Widgets, Protokolle)
- Drag&Drop

2 Warum nicht ... ?

Es wurde aus mehreren Richtung der Vorschlag an mich herangetragen, eine vorhandene Lösung zu portieren. Die Vorteile einer Portierung liegen auf der Hand:

- Die eigentliche Entwicklungsarbeit und das Entwerfen des Konzeptes entfällt.

- Oft gibt es schon eine Menge verfügbarer Software, die unter der zu portierenden Lösung läuft und die - so hofft man - auch auf der portierten Version läuft.
- Eine gute Portierung kann auch von zukünftigen Weiterentwicklungen der portierten Lösung profitieren.
- Umfangreiche Dokumentation, die nicht selbst geschrieben werden muß

Aus diesem Grund habe ich mir einige, breits bestehende Lösungen, angesehen und fasse im Folgenden meine Einschätzungen bezüglich ihrer Verwendbarkeit für eine grafische Oberfläche von L4 zusammen:

2.1 Microwindows

Microwindows ist ein kleines Fenstersystem, was für DOS und Linux umgesetzt wurde. Es arbeitet direkt auf dem Linux-Framebuffer. Leider wird das Fenstersystem schon mit wenigen Fenstern recht langsam und zeigte sich bei meinen Tests recht absturzfreudig. Microwindows kommt mit mehreren parallel laufenden Applikationen zurecht, produziert allerdings redraw-Fehler. Insgesamt macht es keinen sehr ausgereiften Eindruck. Die Geschwindigkeit läßt zudem Gedanken an Echtzeit-Multimedia-Applikationen in weite Ferne rücken.

Link: HTTP://MICROWINDOWS.ORG
Microwindows Webseite

2.2 EmbeddedQt

EmbeddedQt ist die Portierung von Qt für Embedded Systems mit Linux aus Betriebssystem. Da Qt nicht zuletzt durch den Kde sehr verbreitet ist, existieren schon viele Applikationen, die von Qt gebrauch machen.

Einige Punkte sprechen jedoch gegen den Einsatz von EmbeddedQt unter L4:

- benötigt Linux Infrastruktur (ähnliche Probleme, wie X11 auf L4 zu Portieren)
- nicht Server-basiert
- Echtzeitaspekt fehlt: Es einfacher auf Echtzeitanforderungen einzugehen, wenn diese Anforderungen schon in den Entwurf eingehen als diese Nachträglich (zb. in Qt) hinzuzufügen.
- sehr umfangreich - Zitat von der Trolltech-Webseite: "Qt offers the functionality you need to create professional applications. There are around 250 C++ classes in the Qt API. Most of these classes are GUI specific; however, Qt also provides template-based collections, serialization, file and a general I/O device, directory management, date/time classes, regular expression parsing and more. " 23MB source (tgz)

- kommerzielles Produkt, d.h. sollte auf der Basis von L4 ein kommerzielles Produkt entstehen, müssen Lizenzgebühren an Trolltech gezahlt werden
- keine Trennung von Anwendung und GUI
- basiert auf C++, was ich nicht mag
- besondere Features, wie OpenGL benötigen den Support der speziellen Plattform

Link: http://www.trolltech.com/products/qt/embedded/
EmbeddedQt Webseite

2.3 GTKfb

Als Äquivalent zu EmbeddedQt liegt auch eine Embedded-Version von GTK vor.

Allerdings ähneln sich auch die Punkte, die gegen eine Portierung von GTK auf L4 sprechen, den Kritikpunkten an EmbeddedQt.

- nur ein Prozess möglich (keine Verschiedenen Programme gleichzeitig)
- nicht alle gtk-Programme sind tatsächlich nutzbar, da einige z.T. direkte X-calls benutzen
- Hardwarebeschleunigung wird z.Z. nicht unterstützt
- keine Server-Struktur
- nur Nutzung von gtk-Applikationen
- setzt ebenfalls Linux-Infrastruktur (libc) vorraus

Link: http://www.linuxdevices.com/articles/AT9024868021.html
Informationen zu GTKfb

2.4 Evas

Evas ist eine abstrakte Grafikbibliothek mit integriertem Fenstermanagement. Die einzelnen Grafik-Objekte werden komplett von Evas verwaltet. Die Bibliothek bietet entsprechende Funktionen um auf kontrollierte Art auf die Grafikobjekte zuzugreifen. Als Objekte werden zur Zeit Bilder (mit 'alpha-blending'), Text (mit 'antialiasing'), Linien und Rechtecke unterstützt. Evas nutzt eventuell vorhandene 2D-Beschleunigungshardware.

...aber...

- Evas benötigt X11 (zur Eventbehandlung) und Linux. Ein Zitat der Webseite: "Evas is a canvas abstraction that uses Imlib2, as well as OpenGL, and X11." Hardwarebeschleunigung kommt von OpenGL - kein Echter Vorteil Linux-Umgebung ist notwendig - inkl. X11 !

- beschränkter Umfang: sehr gut geeignet für viele Applikationen, aber nicht flexibel genug
- Es wird 'Freetype' zur Darstellung der Texte verwendet. 'Antialiasing' ist somit vielmehr ein 'feature' von 'Freetype' als von Evas.
- Grundprinzipien sind meinen Vorschlägen ähnlich (ein Widget in meinem Sinne ist ein 'intelligentes' Canvas in Evas Sinn)
- Meine Programmierschnittstelle ist abstrakter und flexibler (Durch dynamisch erzeugte 'Kommandosprache')
- 'Event-Handling' kommt von X und beschränkt sich auf Maus-Aktionen

Link: HTTP://WWW.ENLIGHTENMENT.ORG/PAGES/EVAS.HTML
Evas Webseite

2.5 PicoGUI

2.6 Fazit

Die hier vorgestellten Lösungen wurden hauptsächlich für den Embedded-Bereich konzipiert. Deshalb sind für diese Konzepte Funktionen, wie Display-Umleitung nicht so wichtig. Für eine Benutzeroberfläche, die nicht ausschließlich für den Embedded-Bereicht gedacht ist, erscheit diese Funktion jedoch äußerst sinnvoll.

Desweiteren wird in allen Fällen eine Linux-Infrastruktur vorrausgesetzt. Nach meinem Kenntnisstand existiert für L4 noch keine komplette glibc sowie Dateisystem.

Eine wichtige Eigenschaft der Drops-Benutzeroberfläche soll die 'Echtzeitfähigkeit' sein. Dies bedeutet, daß die Fenster von Echtzeit-Applikationen möglichst unabhängig vom Rest des Fenstersystems aktualisiert werden müssen. Diese Funktionalität müßte nachträglich in ein portiertes Fenstersystem eingebaut werden. Der Aufwand hierfür ist für mich nicht unmittelbar kalkulierbar. In diesem Fall kann (höchstwahrscheinlich) auch nicht unmittelbar von der Weiterentwicklung der portierten Software profitiert werden.

Alle hier vorgestellten Lösungen bieten prinzipiell nichts wirklich neues. Zwar stehen L4 nach der Portierung von (z.B.) EmbeddedQt Qt-Applikationen zur Verfügung - aber das ist nicht wirklich etwas, was L4 anderen Plattformen 'vorraus' hätte. Ein neues Konzept ermöglicht prinzipiell neue Möglichkeiten und die Option, eine Applikation für L4 zu finden, die L4 anderen Plattformen 'vorraus' hätte.

3 Konzept

Zunächst wird auf grundsätzliche Design-Kriterien eingegangen. Anschließend werden einige zentrale Komponenten des Konzeptes näher beleuchtet.

Die nachfolgenden Ausführungen sind in keiner Weise vollständig und sind durchaus nicht fest stehend. Sie sollten vielmehr die Grundlage zu einer konstruktiven Diskussion darstellen.

3.1 Schall und Rauch

Ein passender Name für eine Benutzeroberfläche kann auf zweierlei Weise gefunden werden:

- Man findet ein schönes Wort (am besten mit irgendeiner mystischen Bedeutung). Z.Bsp. Kyuss, Crescent, Eclipse... Wenn dann allerdings jemand fragt, was der Name bedeutet, muß man unweigerlich mit den Achseln zucken (wird man durch 'enlightenment' tatsächlich 'erleuchtet'?).

- Man versucht aus Wörtern, die irgendetwas mit Benutzeroberflächen zu tun haben, eine schicke Abkürzung zu bilden. Leider ist dies mit Wörtern wie: 'Windows', 'Desktop', 'Manager', 'User', 'Environment', 'Menu', 'Graphical' nicht unbedingt leicht und man stößt beim Versuch, einige Permutationen durchzuprobieren, oft auf schon bekannte Namen.

Ich habe mich für die zweite Variante entschieden und bin bei 'DOpE' gelandet. Dies steht für 'Desktop Operating Environment'. Dieser Name ist kurz, klingt nicht peinlich und läßt sich gut merken. (Außerdem fällt mir dazu mindestens ein Desktop-Thema ein) Wer diesen Namen absolut nicht mag, kann sich selbstverständlich mit einem neuen Namensvorschlag bei mir melden.

3.2 Philosophie

Dem hier beschriebenen Konzept liegen einige Prinzipien zu Grunde, die zunächst genannt werden:

- keine Annahmen über das Betriebssystem (z.B.keine direkte Nutzung der libc)

- Das Betriebssystem (die grafische Oberfläche) soll 'wissen', mit 'was' der Benutzer arbeitet.

- Programme sollen nur ihre Funktionen beinhalten und sich nicht mit den Details ihrer grafischen Benutzeroberfläche beschäftigen. (Der Nutzer weiß viel besser, was ihm gefällt, als der Programmautor und kann die Oberfläche konfigurieren, ohne daß das Programm sich darum 'kümmert')

- Programme sollen sich nicht (unnötig) mit der Eingabe und Ausgabe von Daten beschäftigen, sondern Daten verarbeiten.

3.3 'Components'

Jede Komponente von 'DoPE' ist ein Modul, welches über eine einfache und allgemeine Modulschnittstelle in das Gesamtsystem integriert wird. Somit ist jede Komponente für sich 'gekapselt'.

Jede Komponente von DoPE besitzt einige Informationen über sich selbst:

- 'name+version' (Der Name wird später von anderen Komponenten als 'Identifier' verwendet, um mit der Komponente zu arbeiten. Der Versions-String wird verwendet, um mögliche Versionskonflikte zwischen Komponenten aufzulösen. Generell gilt die Regel, daß eine Komponente mit seinen früheren Versionen kompatibel sein muß. Außer....)
- 'conflicts' (Falls eine Komponentenversion zu einer Vorgängerversion inkompatibel ist, muß die entsprechende Vorgängerversion hier angegeben werden.)
- 'uses' (Dies ist eine Liste der benötigten anderen Komponenten)
- 'type' (Beinhaltet Informationen über die Art der Komponente, z.B. Widget-Class, Hardware-Interface, Library)

3.3.1 Art der Modularisierung

Es gibt zwei Möglichkeiten ein solches modulares Konzept umzusetzen:

- Modularität auf Sourcecode-Ebene, d.h. man wählt die Komponenten-Module beim Compilieren/Linken aus. Dies zieht nach sich, daß Namenskonflikte zwischen Modulen vermieden werden müssen. Weiterhin ist das Program nur durch einen erneuten Compilerlauf neu zu konfigurieren. Ein dynamischen Laden/Freigeben von Komponenten ist nicht möglich.
- Modularität auf 'binary'-Ebene, d.h. die einzelnen Komponenten werden einzeln compiliert und die Objekt-Files in Laufzeit geladen und verwendet. Dies hat große Vorteile: Die Modulschnittstelle wird start vereinfacht, da Namenskonflikte zwischen Modulen nicht entstehen können. Das Programm ist auch in binärer Form rekonfigurierbar. Die Komponenten können erst dann geladen werden, wenn sie tatsächlich benötigt werden. (äquivalent zu Kernelmodulen unter Linux) Nicht zuletzt macht dieses Vorgehen den Sourcecode der Module übersichtlicher. (zumindestens lehrt mich dies meine bisherige Erfahrung) Nachteil: 'Loader' für Objektdateien notwendig.

3.3.2 Modulschnittstelle für Komponenten

Es folgt eine kurze Beispiel-Skizze einer Modulschnittstelle für Komponenten. Diese nutzt die Modularität auf 'binary'-Ebene. Dies ist natürlich nur ein Beispiel und soll das wesentliche Prinzip der Modulschnittstelle veranschaulichen.

```
/*******************************/
/* DRAFT OF HEADER FILE DOPE.H */
/*******************************/

struct dope_struct {
  void * (*get_component) (char *comp_name);
  int    (*register_component) (char *comp_name, void *);
  ...
}
```

Bei der Initialisierung bekommt die Komponente die dope-Service-Struktur übergeben, die in erster Linie Funktionen zum Zugriff auf andere Komponenten und zum registrieren der eigenen Komponente enthält. Der Registrierungs-Funktion 'register_component' wird die Service-Struktur der eigenen Komponente übergeben.

```
/*********************/
/* EXAMPLE COMPONENT */
/*********************/

/* header file contains the DOpE service structure */
#include "dope.h"

/* the following strings contain the information */
/* about this component */
char *component_name="Button 1.0";
char *component_conflicts="";
char *component_type="Widget";
char *component_uses="Screen 1.0, WinMan 1.2";

struct dope_struct *dope;

struct my_servic_struct {
  void   (*set_text) (void *button,char *new_text);
  char * (*get_text) (void *button);
  ...
} my_services;

/* function, that is called, when DOpE wants to */
/* initialize the component. DOpE provides its */
/* service structure here, which enables the */
/* component to access other components */
int component_init(struct dope_struct *d) {
  dope=d;
```

```
  ...
  dope->register_component(component_name,&my_services);
  return 1; /* success */
}
```

3.3.3 Beispiele für Komponenten

Einige Beispiele für solche Komponenten sind:

- Screen: Diese Komponente repäsentiert den physischen Bildschirm, ist also eine Hardware-Interface-Komponente.
- Mouse: Diese Komponente enthält einen Maustreiber.
- Widgets: Verwaltet rechteckige Bereiche des Bildschirms
- BaseGfx: Bibliotheks-Komponente, die häufig benötigte Grafik-Funktionen beinhaltet.

Einige dieser Komponenten werden genauer in späteren Kapiteln beschrieben.

3.3.4 Initialisierung der Komponenten

Da einige Komponenten von der Existenz anderer Komponenten abhängen, ist deren Initialisierungsreihenfolge wichtig. Natürlich kann ein Mensch eine feste Initialisierungsreihenfolge der Komponenten vorgeben - dies ist allerdings nicht im Sinne des modularen Konzeptes.

Durch die 'uses', 'version' und 'conflicts'-Informationen der einzelnen Komponenten kennt DOpE die benötigten Abhängigkeiten um eine geeignete Initialisierungs-Topologie zu finden und ggf. Versionskonflikten vorzubeugen.

3.4 'Widgets'

Ein Widget (die Namensgebung wurde inspiriert durch Tcl/Tk) ist der Grundbaustein der Benutzeroberfläche und verwaltet einen rechteckigen Bereich des Bildschirmes. Ein Widget kann weitere Widgets (children) beinhalten. Eine Benutzeroberfläche ist demnach eine Widget-Hierarchie.

Es gibt verschiedene Widget-Typen. Jeder Widget-Typ repräsentiert einen bestimmten Datentyp oder ein bestimmtes Protokoll.

3.4.1 Allgemeine Attribute

Jedes Widget 'w'- egal von welchem Typ - besitzt einige grundlegende Attribute:

- 'class' - Widget-Typ (Tatsächlich ist dies ein Zeiger auf zugehörigen 'Widgethandler')

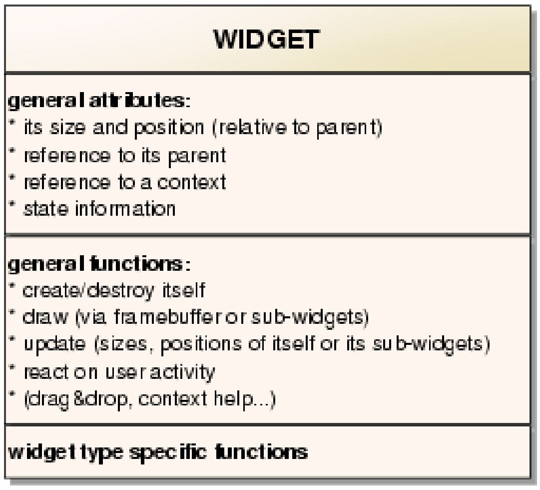

Abbildung 1: Die wesentlichen Attribute und Methoden eines Widgets auf einen Blick

- 'parent' - Widget, welches 'w' beinhaltet
- Position relativ zum 'parent'
- minimale, maximale und aktuelle Größe (Breite, Höhe)
- 'dragtypes' - Namen der Datentypen, die das Widget repräsentieren kann
- 'droptypes' - Namen der Datentypen, die in das Widget 'gezogen' werden können
- 'context' - Zeiger auf einen beliebigen Kontext (der Inhalt ist abhängig vom eigentlichen Typ, der Zeiger auf den Inhalt ist jedoch ein allgemeines Attribut)

3.4.2 Allgemeine Widget-Funktionen (Methoden)

- 'event-handler' - Funktionen zum Behandeln von Maus-, Tastatur-... - Ereignissen
- 'draw' - Funktion zum Zeichnen eines rechteckigen Teils des Widgets
- 'update' - Anpassen an Veränderungen (z.B. wenn ein 'child'-Widget seine Größe ändert)
- 'find' - Suchen eines bestimmten Widgets unter seinen Kindern und Kindeskindern...

Die hier aufgeführten Methoden und Attribute sind sicher nicht vollständig, sollen aber einen Eindruck über das Gesamtkonzept geben.

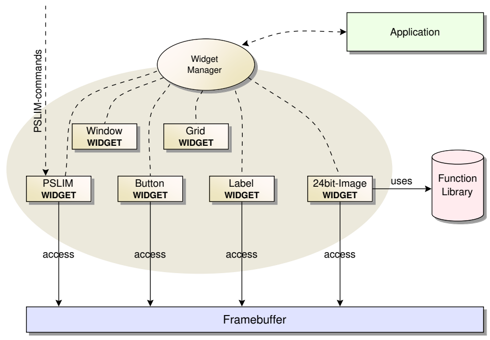

Abbildung 2: Verschiedene Widgets werden vom 'Widget-Manager' verwaltet

3.4.3 Soft widgets - Hard widgets

Generell kann eine Unterscheidung von Widgets in zwei Klassen getroffen werden:

- Soft-Widgets nutzen nur Komponenten, die keinen Zugriff auf Hardware erlauben. Das heißt, daß Soft-Widgets bei einer Portierung von einer auf eine andere Plattform nicht angepaßt werden müssen.
- Hard-Widgets nutzen direkte Hardware-Zugriffe. Jedes Widget, was bespielsweise die 'Screen'-Komponente benutzt, ist von Haus aus nicht portabel.

DOpE hat durch die 'uses'-Informationen einer Komponente (also auch eines Widgets) das Wissen über 'hard'- und 'soft'-Widgets.

3.5 Grundlegende Widget-Typen

In diesem Abschnitt sind die Widget-Typen aufgeführt, die grundlegend für ein Fenstersystem sind aufgeführt.

Das folgende Diagramm stellt einige Widget-Typen dar. Die Widgets werden vom 'Widget Manager' verwaltet (Wo befindet sich welches Widget?, Welches Widget gehört zu welcher Applikation? etc.). Einige Widgets greifen direkt auf den Framebuffer zu.

Die Applikation kommuniziert lediglich mit dem 'Widget Manager'. Die Kommunikation erfolgt auf der Basis einer Mini-Kommandosprache.

Ein wichtiger Aspekt ist, daß das 'Image'-Widget schon von sich aus Funktionen zur Verfügung stellt, die auf jeder Widget des Typs 'Image' anwendbar sind. Diese Funktionen können auch als externe 'Module' Vorliegen. Die Modulschnittstelle zwischen dem 'Image'-Widget und den Funktionen ist Widget-

Spezifisch. Denkbar wäre es (vielleicht sogar mehrere), bereits vorhandene Modulschnittstellen zu integrieren.

3.5.1 Window

spezielle Attribute

- Widgets der Fensterelement ('closer','title','fuller'...)
- 'content' - Fensterinhalt-Widget

3.5.2 Button

Ein 'Button' ist ein Knopf, der einen Zustand repräsentiert.
spezielle Attribute:

- 'text' - Beschriftung
- 'state' - Zustand (0=nicht gedrückt)
- 'command' - Kommandostring

3.5.3 Container

Ein Container ist ein rechteckiger Bereich, der Widget beinhalten kann, was größer ist, als er selbst.
spezielle Attribute:

- 'x_offset'
- 'y_offset'
- 'content' - das Widget im Container

3.5.4 Grid

Ein 'Grid' ist ein Raster (Tabelle), welches aus mxn Zellen besteht. In jeder Zelle kann sich maximal ein 'child'-Widget befinden. Ein 'child'-Widget kann sich über mehrere Zellen erstrecken.
spezielle Attribute:

- Größe der Zeilen und Spalten
- Liste der Kinder inkl. Informationen über deren Position

spezielle Methoden:

- 'adopt' - neues Widget im Gitter platzieren
- 'release' - Widget aus Gitter entfernen

3.5.5 Frame

Ein 'Frame' ist ein rechteckiger gefüllter oder nicht gefüllter Bereich
spezielle Attribute:

- Stil (gefüllt, nicht gefüllt, wie gefüllt?)
- 'content' - Inhalt des 'Frames'

3.5.6 Label

Ein 'Label' ist ein passives Textfeld (d.h. nur zur Anzeige von Text)
spezielle Attribute:

- 'text'
- eventuell Textausrichtung

3.6 Widget-Typen zweiter Stufe

3.6.1 Scale

Ein 'Scale'-Widget repräsentiert einen Zahlenwert

3.6.2 Entry

Ein 'Entry'-Widget repräsentiert einen editierbaren Text-String (es wäre vielleicht überlegenswert, 'Entry' und 'Label' in einem Widget-Typ zu vereinen)

3.6.3 Scrollbar

Ein Scrollbar repräsentiert eine Scrollposition. Das sind genaugenommen zwei fraktionale Zahlenwerte (die Position und die Ausdehnung)

3.6.4 Image

Ein 'Image' repräsentiert ein Bild
spezielle Attribute:

- 'img_w','img_h' - die Größe des Bildes (muß nicht unbedingt mit der Größe des Widgets übereinstimmen)
- 'img_data' - Bilddaten

3.6.5 PSLIM

Das 'PSLIM'-Widget repräsentiert das PSLIM-Protokoll.
spezielle Attribute:

- 'shadow_buffer' - Puffer, in dem tatsächlich gezeichnet wird
- Kommunikationskanal (ip, port?)

3.6.6 RT-PSLIM

Das 'RT-PSLIM'-Widget repräsentiert ebenfalls das PSLIM-Protokoll. Für jedes RT-PSLIM-Widget wird ein eigener Thread erzeugt, der das Widget in gleichmäßigen Abständen neu zeichnet.

zusätzliches spezielles Attribut:

- update-Frequenz

Die genaue Entscheidung, wie es Umgesetzt wird, ist noch nicht getroffen. Eventuell könnten auch zwei Threads je 'RT-PSLIM'-Widget erzeugt werden: einen für die Kommunikation und das zeichnen in den 'shadow-buffer' und einen für das Update des Bildschirmes.

Auf jeden Fall muß dieses Widget sehr eng mit dem 'Windowmanager' zusammenarbeiten, damit das asynchrone Zeichnen nicht zu häßlichen Nebeneffekten führt.

3.7 Memory-Komponente

Die Memory-Komponente beinhaltet Funktionen zum Allokieren und Deallokieren von Speicher. Dies kann z.B. durch ein 'Weiterreichen' der Aufrufe an das Betriebssystem oder durch einen eigenen Speichermanager geschehen. Durch die Nutzung der Memory-Komponente durch andere Komponenten wird vor allem die Portabilität gewahrt, da die Betriebssystem-Schnittstellen zum Allokieren von Speicher unterschiedlich aussehen können.

3.8 Screen

Die Screen-Komponente beinhaltet die physischen Eigenschaften des Bildschirms. Jede andere Komponente, die auf die Screen-Komponente zugreift, hat die Möglichkeit, direkt auf die Videohardware zuzugreifen. (Durch die 'uses'-Informationen der einzelnen Komponenten hat DOpE allerdings stets 'die Übersicht', welche Komponenten auf Screen zugreifen)

Die Screen-Komponente beinhaltet die Attribute:

- Bildschirm-Adresse
- 'Shadow-screen' Adresse (Speicher, der für gepufferte Ausgabe verwendet wird)
- Breite, Höhe in Pixeln
- Farbtiefe
- aktueller Clipping-Bereich
- Position des Mousecursors

Desweiteren stellt er folgende Funktionen zur Verfügung

- Initialisierung der Videohardware
- Wiederherstellen der alten Videohardware-Konfiguration
- Setzen der Position des Mousecursors
- Kopieren vom 'shadow' in den tatsächlichen Bildschirmspeicher

Die Screen-Komponente ist eine 'Hardware-Interface'-Komponente. Durch das Austauschen dieser einen Komponente, ist DOpE sehr leicht an andere Videohardware anzupassen. (z.B. direkte Initialisierung der Grafikkarte etc.)

3.8.1 Font-Engine

Die 'Font Engine'-Komponente hat die Aufgabe, Zeichen unter Beachtung des 'clipping'-Bereiches auf den Bildschirm auszugeben.

Wie dies genau geschiet, bleibt im inneren der Font-Engine 'verborgen'. Zunächst wird es nur eine Unterstützung einfacher Bitmap-Fonts geben. Die Integration von 'Freetype' könnte in einem nächsten Schritt folgen. Die eventuelle Verwaltung von Font-Caches findet ebenfalls hier statt.

3.9 Variablen

Für jeden 'client' verwaltet DOpE eine Liste von Variablen. Zunächst gibt es nur einen Variablen-Typ - Widgets. Widgets wiederum repräsentieren jeweils verschiedene Datentypen. (Beispielsweise repräsentiert ein 'Scale'-Widget eine Zahl oder ein 'Image'-Widget ein Bild)

Somit entsteht die Vielfalt an Datentypen aus der Vielfalt von Widgets. Verschiedene Widget-Typen können dabei durchaus gleiche Datentypen repräsentieren. (es sind beispielsweise mehrere Widgets vorstellbar, die eine Zahl repräsentieren)

Variablen können genutzt werden, um Widgets miteinander zu synchronisieren (ähnlich wie in Tcl/Tk) - Variablen werden an Widgets gekoppelt.

(Über die Details bin ich mir hierbei noch nicht ganz im Klaren)

3.10 Kommandosprache

Um eine Benutzeroberfläche effizient zu beschreiben, soll eine winzige Kommandosprache Verwendung finden.

Der Sprachumfang soll keineswegs von vornherein festgelegt sein. Vielmehr bestimmen die Widgets selbst den Sprachumfang.

Eine solche Kommandosprache ist sehr anschaulich und soll es dem Entwickler einfach ermöglichen, in kurzer Zeit (und vor allem mit kompaktem code)) eine Benutzeroberfläche zu definieren.

Zudem schützt sie vor groben Versionskonflikten. (Wenn ein Attribut gesetzt wird, was nicht existiert, kann es ja ignoriert werden - hat aber keine schwerwiegenden Konsequenzen)

3.10.1 Syntax

Die Syntax der Kommandosprache ist sehr einfach gehalten. Jedes Kommando besteht aus einer Zeile, die durch ein ';' oder einen Zeilenumbruch beendet wird.

Beispielhaft sollen die wichtigsten Szenarien dargestellt werden:

Das Erzeugen eines neuen Widgets:

```
but1 = new Button(-width 50 -height 30 -text "Exit");
```

Ein neues Buttonwidget wird erzeugt. Es können beliebige Parameter übergeben werden. Die Variable but1 enthält das erzeugte Widget.

Setzen von Widget-Attributen:

```
but1.configure(-text "Hallo" -width 60 -variable a);
```

Die 'configure'-Methode erlaubt das Setzen mehrerer Parameter auf einmal. Das Widget wird erst nach dem Setzen aller Parameter aktualisiert.

Das Aufrufen einer Widget-Methode:

```
but1.destroy();
```

Das Widget but1 wird zerstört. Ein Methodenaufruf besitzt stehts einen Rückgabewert.

Das Liefern eines Variablenwertes:

```
but1
```

Diese wenigen Sprachelemente genügen, um eine Benutzeroberfläche zu beschreiben.

3.10.2 Definition von Sprachelementen

Widgets haben die Möglichkeit, der Kommandosprache eigene Methoden hinzuzufügen, sowie Attribute zu registrieren.

Die Registrierung von Widget-Attributen und Methoden könnte wie folgt geschehen:

```
/* EXAMPLE WIDGET */

...

  /* somewhere inside the component initialisation */
  /* routine... */

  /* Define how the widget can be accessed via script commands. */
  /* a widget has attributes...*/
  register_attribute("long width",&get_width,&set_width,&update_widget);
```

```
/* ...and functions */
register_function("long adopt(widget child,long row=-1,long column=-1)",
                  &adopt);
```

Ein Attribut besitzt einen Typ und einen Namen. Diese Informationen werden DOpE im ersten Parameter mitgeteilt. Die folgenden Parameter sind Zeiger auf die Behandlungsroutinen zum Setzen bzw. Ermitteln des Attributwertes. Der letzte Parameter ist ein Zeiger auf eine Update-Funktion, die aufgerufen werden sollte, damit die neuen Attributwerte wirksam werden. Es können somit mehrere Attribute mit einer 'configure'-Anweisung gesetzt und anschließend (!) einmal das Widget aktualisiert werden.

Das Registrieren von Widget-Methoden geschiet ähnlich einfach. Im ersten Argument wird die Art der Methode beschrieben. Jeder möglicher Methodenparameter besitzt einen Typ und Namen. Zusätzlich kann ein Default-Wert angegeben werden. Die Angabe eines solchen Parameters beim Methoden-Aufruf ist optional.

Anschließend können die registrierten Funktionen 'von außen' aufgerufen werden, bzw. Attribute mittels 'configure' gesetzt werden.

3.10.3 Beispiel-Anwendung

Das folgende code-Fragment soll illustrieren, wie ein Anwendungsprogramm eine Benutzeroberfläche erzeugt:

```
/* Example Definitions of event types */
/* ...should be defined in a dedicated header file */
#define DOPE_NOTHING 0;
#define DOPE_COMMAND 1;
#define DOPE_WIDGET_CHANGE 2;
#define DOPE_DROP 3;
#define DOPE_TIMER 4;
#define DOPE_EXIT 1000;

/* Pointer to event information buffer */
/* This Pointer points to a buffer with additional */
/* information about the occured event */
char *event_buf;

/* Pointer to DOpE command result buffer */
/* Each executed DOpE command returns a result, which */
/* can be interpreted by the application */
char *result_buf;

/* Identifier for the occured event type */
int  event_type;
```

```
...

int main(void) {

  ...

  /* build window with some widgets */
  /* finally the window will contain a grid with     */
  /* two rows. The upper row contains a text field.  */
  /* The lower row contains a button. The button has */
  /* a 'command' as attribute. When the button is    */
  /* pressed, this 'command'-string will be returned */
  /* to the event buffer (see mainloop)              */
  dope_Cmd("\
    win1 = new Window( -width 200 -height 200 -title Testwin )\n\
    win1.configure( -droptypes {Text Image24 Color} )\n\
    but1 = new Button( -text klickhier -command but1_clicked] )\n\
    txt1 = new Text()\n\
    grid1 = new Grid() \n\
    grid1.adopt( txt1 -row 0 )\n\
    grid1.adopt( but1 -row 1 )\n\
    win1.adopt( grid1 ) \n\
    win1.open()\n\
  ",result_buf);

  /* here the mainloop just begins */
  while (!quit) {

    /* wait for an event - there would also be a */
    /* non-blocking version possible */
    event_type=dope_GetEvent(event_buf);

    /* check what type of event occured */
    switch (event_type) {

      /* HANDLE COMMAND MESSAGES */
      case DOPE_COMMAND:

        /* event_buf contains command from dope */
      /* the strcmp call can be eliminated by */
      /* giving the commands 'better' names   */
      /* ...just for illustration... */
        if (!strcmp(event_buf,"but1_clicked")) {

        /* reaction - append some text to the */
```

```
          /* text widget */
            dope_Cmd("txt1.append( irgendwas )",result_buf);
          }
          break;

        case DOPE_EXIT:
          quit=0;
      } /* end of switch */
    } /* end of while */
  } /* end of main() */
```

3.11 Beispiel-Szenario

Bildwidget Html-Widget Pslim-Widget

3.12 Echtzeit

4 Was bisher geschah...

Mein Interesse an grafischen Benutzeroberflächen reicht schon einige Jahre zurück. Im Laufe der Zeit habe ich einige Fenstersysteme entwickelt - allerdings zum Großteil nicht mit einem ernsthaften Hintergrund, sondern eher aus 'Spieltrieb'.

4.1 Modulation III

'Modulation III' ist eine Sound-Demo, die ein funktionierendes Fenstersystem beinhaltete. Mein Ziel war, eine Oberfläche zu schaffen, die dem Benutzer das Gefühl gibt, vor einem wirklich schnellen Rechner zu sitzen (obwohl der Atari Falcon nur 16Mhz hat). Aus diesem Grund wurde das rechte Viertel des Bildschirmes genutzt, um einen Demoeffekt flüssig anzuzeigen - auch wenn der Benutzer Fenster verschiebt usw. (Dies zu erreichen, war nicht sonderlich schwer, da der Demoeffekt einfach in einem Timer-Interrupt lief - schlimm war, daß der Effekt etwa 50% Rechenzeit benötigte, also noch weniger Zeit für die eigentliche GUI zur Verfügung stand)

Das Fenstersystem war noch recht einfach aufgebaut und bestand im wesentlichen aus Bildern - für seinen Zweck völlig ausreichend.

4.2 Voodoo & Whip!

Das Fenstersystem 'Voodoo' wurde ursprünglich für einen alternativen Rechner ('Phenix' von der französischen Firma Centek) programmiert. Leider ist aus diesem Rechner nichts geworden. Das Fenstersystem funktionierte bereits sehr gut und war um ein Mehrfaches flüssiger zu bedienen, als das originale Atari-Fenstersystem 'GEM'. Außerdem war es bunter :-)

Abbildung 3: Screenshot der Sounddemo 'Modulation III'

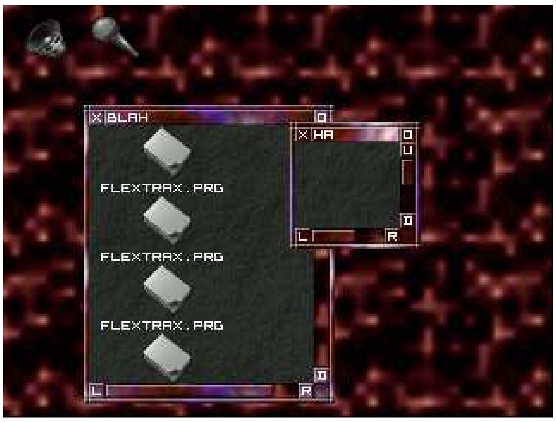

Abbildung 4: Screenshot von 'Voodoo' in einer Entwicklungsversion

Abbildung 5: Screenshot des 'Whip!'-Menüs: es stehen verschiedene Bedienelemente zur Verfügung (Buttons,Scales,Scrollbars...)

Alle originalen Betriebssystem-Funktionen des Atari-Betriebssystems (bis auf die Filesystem-Routinen) wurden durch eigene Routinen ersetzt (Tastatur, Maus, Grafik-Treiber,Grafikroutinen, Speichermanagement). Desweiteren wurden keinerlei Standard-C-Libraries verwendet - ein Umstand der das Fenstersystem sehr portierbar machte. Es wurde beispielsweise eine zeitige Version von 'Voodoo' auf die Playstation portiert.

Später wurde der Kern von 'Voodoo' für die grafische Benutzeroberfläche von 'Whip!' verwendet. 'Whip!' ist ein sog. 'Sound-To-Light' Programm für den Atari Falcon. Ein Soundsignal wird mittels DSP56001 analysiert und zur Parametrisierung von klassischen Demoeffekten verwendet. Das Programm besitzt eine Modulschnittstelle, die Effektparameter u.a. erlaubt.

Link: http://www.inf.tu-dresden.de/˜nf2
Escape Homepage - der Link zur Whip!-Seite befindet sich im Menü links

5 Fahrplan