Schriftliche Ausarbeitung zur Vorlesung
Grafische Datenverarbeitung 2004/05

OpenGL-Plugins für das X Multimedia System

Felix J. Ogris

28. September 2006

Inhaltsverzeichnis

1. Einleitung

Diese Ausarbeitung zeigt die Programmierung von OpenGL basierten Visualisierungsplugins für das *X Multimedia System*, kurz *XMMS*. Sie entstand mit 12 Monaten Verspätung im Anschluss an die 2semestrige Vorlesung *Grafische Datenverarbeitung* im WS/SS 2004/05 an der Fachhochschule Bielefeld. Unter anderem wird das im Praktikum erstellte Programm bzw. Modell eines Klick-Klack- oder auch Kügelchen-Spiels (5 Metallkugeln an dünnen Nylonfäden zur Demonstration von Impuls- und Energieerhaltung) in einer modifizierten Variante verwendet. Als Programmiersprache dient aus diversen Gründen C. Zum einen wurde in der Vorlesung das OpenGL-API in C behandelt, zum anderen sind XMMS und somit auch alle Headerdateien in C programmiert. Ausserdem wurde das erwähnte Klick-Klack-Spiel ebenfalls in dieser vom Autor ohnehin favorisierten Programmiersprache erstellt.

2. XMMS

XMMS ist ein quelloffener Player für diverse Multimediaformate. Er wird unter Unix-artigen Betriebssystemen hauptsächlich zum Abspielen von Audiodaten im MP3-, Ogg/Vorbis- und WAV-Format verwendet. Aufgrund ähnlicher Optik, Bedienung und Funktionsumfang kann er als Pendant zum unter Windows häufig eingesetzen Winamp angesehen werden.

Abbildung 1: XMMS

XMMS kann mit folgenden Typen von Plugins erweitert werden:

Inputplugins gewinnen aus Streams wie Dateien oder Netzwerkverbindungen unkomprimierte Daten und stellen diese den Output-, Effekt- oder Visualisierungsplugins zur Verfügung

Outputplugins schreiben PCM-Daten in Dateien, auf Geräte wie Soundkarten oder stellen sie zum Abruf über einen Netzwerkserver bereit

Effektplugins werden zwischen Input- und Output-Plugins eingesetzt und versehen die PCM-Daten mit zusätzlichen akkustischen Informationen wie z.B. einem Echo oder simulierten Raumklang

Visualisierungsplugins bekommen die PCM-Daten und das Frequenzspektrum bereitgestellt, um sie für den Anwender grafisch aufzubereiten; dieses stellt den Schwerpunkt dieser Ausarbeitung dar

Generische Plugins steuern XMMS, indem sie z.B. den Empfänger einer Infrarotfernbedienung oder die Ansteuerung eines Joysticks auslesen.

Ferner besteht die Möglichkeit, die Oberfläche von XMMS über eigene Skins anzupassen. Ausserdem kann der Quellcode gemäß der GNU Public Lizenz (GPL) frei modifiziert oder als Grundlage für eigene Projekte genutzt werden.

3. Visualisierungsplugins

3.1. Format

Alle Pluginarten werden als dynamische Bibliothek realisiert. Der Dateiname eines Plugins endet unter den meisten Unix-artigen Betriebssystemen auf *.so*, was abkürzend für *shared object* steht; diese müssen zwingend mit den Compilerflags *-dynamic -fPIC* übersetzt werden. Bei Programmstart lädt das XMMS diese Dateien, welche hierzu entweder in einem globalen Verzeichnis oder unterhalb der Ordnerstruktur *.xmms/Plugins* im Heimatverzeichnis des Anwenders liegen müssen. Das globale Pluginverzeichnis lässt sich über das Hilfsprogramm *xmms-config* mit dem Parameter *–visualization-plugin-dir* (oder –input-plugin-dir, –output-plugin-dir usw. für die anderen Plugintypen) ermitteln. Da man dort als normaler Benutzer in der Regel keine Schreibrechte, ist zumindest während der Entwicklungsphase eine Installation im Verzeichnis *.xmms/Plugins/Visualization* (oder analog *.xmms/Plugins/Input*, *.xmms/Plugins/Output* usw.) unterhalb des eigenen Heimatverzeichnisses vorzuziehen. Zudem kann ein fehlerhaftes Plugin zum Absturz führen, so dass es - wenn es im globalen Pluginverzeichnis installiert wurde - XMMS für alle Benutzer unbrauchbar machen kann.

3.2. Schnittstelle

Als Einsprungpunkt erwartet XMMS die parameterlose Funktion *get_vplugin_info*, welche einen Zeiger auf eine Struktur vom Typ *VisPlugin* zurück liefern muss. Diese Struktur beinhaltet vornehmlich Zeiger auf weitere Funktionen, über die das Plugin z.B. PCM-Daten erhält. Der Typ *VisPlugin* bzw. die eigentlich sich dahinter verbergende C-Struktur *_VisPlugin* sind in der Headerdatei *plugin.h* wie folgt deklariert:

```
typedef struct _VisPlugin
{
  void *handle;
  char *filename;
  int xmms_session;
  char *description;
  int num_pcm_chs_wanted;
  int num_freq_chs_wanted;
  void (*init)(void);
  void (*cleanup)(void);
  void (*about)(void);
  void (*configure)(void);
  void (*disable_plugin)(struct _VisPlugin *);
  void (*playback_start)(void);
  void (*playback_stop)(void);
  void (*render_pcm)(gint16 pcm_data[2][512]);
  void (*render_freq)(gint16 freq_data[2][256]);
} VisPlugin;
```

Die einzelnen Variablen haben folgende Bedeutungen:

handle wird vom XMMS gesetzt und enthält nach der Initialisierung des Plugins das Handle, welches die Systemfunktion *dlopen* zum Laden einer dynamischen Bibliothek zurück liefert

filename wird ebenfalls vom XMMS gesetzt und enthält den vollständigen Pfadnamen des Plugins, der ferner als eindeutige Identifikation dient, so dass ein Plugin nicht mehrfach geladen wird

xmms_session wird auch vom XMMS initialisiert und enthält eine eindeutige Verbindungsnummer, mit der ein Plugin Steuerfunktionen wie Start, Stop, nächstes Lied u.ä. ausführen kann

description sollte vom Programmierer auf eine kurze und prägnante Beschreibung seines Plugins gesetzt werden, welche in der Übersichtsliste aller Visualisierungsmodule als Infotext angezeigt wird (s. Abbildung 2)

num_pcm_chs_wanted zeigt dem XMMS an, wie viele PCM-Kanäle das Plugin erhalten möchte, wobei der Wert 0 die Verarbeitung von PCM-Daten deaktiviert, der Wert 1 für Mono-Daten und Werte grösser 1 für Stereo-Daten steht

num_freq_chs_wanted gleicht *num_pcm_chs_wanted*, allerdings wird hiermit die Anzahl der Frequenzspektren festgelegt

init kann, wenn vom Programmierer gewünscht, auf eine Funktion zeigen, die nach dem Laden des Plugins ausgeführt werden soll

disable ist das Gegenstück zu *init* und kann auf eine Funktion zeigen, die vor dem Deaktivieren, gemeinhin vor dem Beenden des Programmes ausgeführt wird

about wird ausgeführt, wenn der Anwender in der Übersichtsliste das Plugin angewählt und den *About*-Knopf gedrückt hat

configure gleicht der Variablen *about*, ist aber mit dem *Configure*-Knopf verknüpft

disable_plugin wird vom XMMS auf eine Funktion gesetzt, die das Plugin aufrufen kann, um sich selbst aus der Liste der aktiven Plugins zu entfernen; als Parameter muss ein Zeiger auf die Plugin-spezifische VisPlugin-Struktur übergeben werden

playback_start kann vom Programmierer auf eine Funktion gesetzt werden, die aufgerufen wird, nachdem der Anwender die Wiedergabe gestartet hat

playback_stop wird dementsprechend aufgerufen, nachdem die Wiedergabe gestoppt wurde

render_pcm wird aufgerufen, sobald das gerade aktive Input-Plugin unkomprimierte PCM-Daten bereitstellt

render_freq gleicht *render_pcm*, allerdings wird hier das mit Hilfe einer Fast Fourier Transformation aus den PCM-Daten gewonnene Frequenzspektrum übergeben.

Hat einer der Funktionszeiger wie z.B. *playback_start* den Wert NULL, so wird diese Funktion nicht ausgeführt und das abrupte Programmende durch einen *Segmentation fault* vermieden.

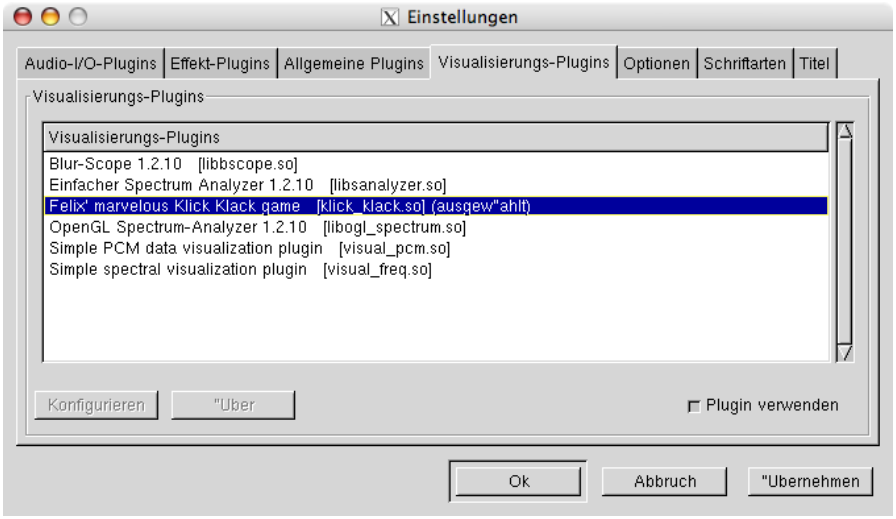

Abbildung 2: Liste der installierten Visualisierungsplugins

3.3. Die Funktionen render_pcm und render_freq

Die beiden Funktionen *render_pcm* und *render_freq* werden aufgerufen, sobald das Input-Plugin weitere Daten decodiert hat und das Output-Plugin vorangegangene Daten abgearbeitet hat. Im Idealfall werden die Funktionen mit der Häufigkeit

$$\frac{f_s}{512} \tag{1}$$

also bei einer Samplingfrequenz f_s von 44100 Hz rund 86 Mal pro Sekunde aufgerufen. Allerdings liefern Inputplugins wie z.B. jenes für MP3-Dateien pro Decodierungsdurchgang mehr als 512 Samples: Bei einer in CD-Qualität, also mit 44,1 kHz und 16 Bit in Stereo aufgenommenen und MP3 codierten Datei werden 1152 Samples geliefert. Das XMMS verwirft hierbei vor dem Aufruf von *render_pcm* und *render_freq* die überflüssigen 640 Samples, so dass ein Visualisierungsplugin eine Audiodatei nicht originalgetreu darstellen kann. Die Funktion *render_freq* erhält über den Parameter *freq_data* das aus den PCM-Daten mittels einer FFT berechnete Frequenzspektrum. Als indirekte Folge des Nyquist-Theorems reicht der dargestellte Frequenzraum von 0 Hz bis zur Hälfte der Samplingrate, meist also bis 22050 Hz. Somit stellt jeder Wert n des übergebenen Arrays *freq_data* die Intensität der Frequenzen $n * \frac{f_s}{2*256}$ bis $(n+1) * \frac{f_s}{2*256}$ dar.

Beide Funktionen sind sehr zeitkritisch, sie sollten auf keinen Fall die eigentliche Berechnung und Darstellung einer Grafik vornehmen, da ansonsten der Abspielprozess ins Stocken geraten könnte. Um dies zu vermeiden, werden alle grafischen Operationen in einem seperaten Thread vorgenommen, welcher quasi-parallel zum XMMS und somit zu *render_pcm* und *render_freq* läuft. Beide Funktionen kopieren lediglich die ihnen übergebenen Daten in globale Variablen, die wiederum von der im seperaten Thread ausgeführten Funktion *display_thread_func*, die das OpenGL-System ansteuert, ausgelesen werden. Der Zugriff auf die globalen Variablen wird mit einem Mutex synchronisiert, der von *playback_start* angelegt und initial gesperrt wird. Die Funktion *display_thread_func* versucht, bevor sie etwaige OpenGL-Befehle absetzt, ebenfalls diesen Mutex zu sperren, welcher jedoch nur von *render_pcm* bzw. *render_freq* freigegeben wird. Somit dient der Mutex lediglich als Benachrichtigungsmechanismus für *display_thread_func*, der neue Audiodaten signalisiert. Auf ein explizietes Sperren der

globalen PCM- bzw. Frequenzdatenvariablen wurde verzichtet, da jeweils nur genau eine Funktion lesend (*display_thread_func*) bzw. schreibend (*render_{pcm,freq}*) darauf zugreifen. Ausserdem reproduzieren die bereitgestellten Daten nach obigen Betrachtungen in der Regel nicht originalgetreu den gesamten Audiostream, so dass es bei mehreren Aufrufen pro Sekunde unerheblich ist, wenn von der Funktion *display_thread_func* Samples zweier aufeinander folgender Blöcke dargestellt werden.

3.4. X Window

Das X Window-System, kurz *X11* oder auch einfach nur *X*, ist unter Unix-artigen Systemen die Standardschnittstelle zur Ansteuerung grafischer Oberflächen. Es umfasst unter anderem Treiber für Grafikkarten, Tastaturen und Mäuse und stellt eine Programmierschnittstelle in der Form der *Xlib* samt diverser Headerdateien bereit. Da es als Client-Server-Anwendung konzipiert wurde, ist X11 generell netzwerkfähig. Der *X-Server* steuert dabei die Hardware an und läuft auf dem Rechner, der die grafische Oberfläche darstellt und Maus- und Tastatureingaben verarbeitet. Programme wie z.B. ein Webbrowser verbinden sich auf den X-Server und setzen Zeichenbefehle ab, fungieren also als Client. Auf einer normalen Workstation laufen Server und Clients auf dem selben Rechner. Allerdings unterstützt X11 nur grundlegende Funktionen wie Öffnen und Schliessen von Fenstern, ein durch Eingabegeräte bedingtes Eventhandling und das Zeichnen von Primitiven von Linien und Rechtecken. Höhere Funktionen einer grafischen Oberfläche werden hingegen von *Window Managern* übernommen, die ihrerseits meist auch Programmierschnitstellen anbieten, sich aber immer der Xlib bedienen. Eine besondere Form von Grafikbibliothek ist die GLUT-Library, die das Ansteuern der grafischen Oberfläche abstrahiert und somit den Einsatz eines OpenGL basierten Programmes auf unterschiedlichen Betriebssystemen wie Unix, Windows etc. ermöglicht. Jedoch ist es nicht möglich, in das vorgegebene Eventhandling von GLUT einzugreifen und andere Ereignisse ausser Tastatureingaben, Mausbewegungen oder Verändern der Fenstergrösse auszulösen. Dieses ließe sich umgehen, indem die Funktion *glutMainLoop* ebenfalls in einem eigenen Thread ausgeführt wird. Allerdings erwies sich GLUT als nicht threadsafe: Der Aufruf von *glutPostRedisplay* aus der Funktion *display_thread_func* bewirkte eben nicht das Aktualisieren des OpenGL-Fensters. Als mögliche Alternative bietet sich der unelegante Umweg über eine Timerfunktion an, die mit *glutTimerFunc* eingerichtet wird. Da XMMS ohnehin eine Unix-artige Umgebung voraussetzt, wurde auf GLUT verzichtet und das X Window-System direkt angesteuert.

3.4.1. X Window-Fenster anlegen

Jedes Programm, welches X11 ansteuern will, muss die Datei *X11/Xlib.h* einbinden, die grundlegende Datentypen und Funktionsdeklarationen bereitstellt. Ereignisse wie das Schliessen eines Fensters werden nicht von X11 selbst ausgelöst, sondern vom eingesetzten Window Manager, der neben Fensterdekorationen auch das gängige Kreuz in der rechten, oberen Fensterecke bereitstellt. Um solche Events verarbeiten zu können, muss ein Programm auch mit dem Window Manager kommunizieren und hierfür die Datei *X11/Xatom.h* inkludieren.

Die Verbindung zum X-Server wird über die Funktion *XOpenDisplay* hergestellt. Sie erwartet die Hostadresse des Servers als Parameter. Übergibt man statt dessen *NULL*, so wählt das X Window-System automatisch die bestmögliche Verbindungsart zum lokalen X-Server. Als Rückgabe erhält man im Erfolgsfall einen Zeiger auf eine Variable vom Typ *Display*, ansonsten den Wert *NULL*:

```
display = XOpenDisplay(NULL);
if (!display) {
  /* Fehlerbehandlung */
  ...
}
```

Ein X-Server kann mehrere Bildschirme ansteuern; die Nummer des Hauptbildschirms erhält man über das Macro *DefaultScreen*, welches einen Integer liefert:

```
screen = DefaultScreen(display);
```

Über die Funktion *glXChooseVisual* wird ermittelt, ob OpenGL-Parameter wie z.B. eine bestimmte Farbtiefe von der aktuellen Grafikkonfiguration unterstützt werden. Neben der Displayvariablen und der Bildschirmnummer erwartet die Funktion ein nullterminiertes Integer-Array, welches die gewünschten OpenGL-Optionen darstellt. Man erhält einen Zeiger auf eine *XVisualInfo*-Struktur oder *NULL*, sofern die gewünschte Konfiguration nicht unterstützt wird:

```
int attributes[] = { GLX_RGBA, GLX_DOUBLEBUFFER, GLX_DEPTH_SIZE, 16, 0 };
visualinfo = glXChooseVisual(display, screen, attributes);
if (!visualinfo) {
  /* Fehlerbehandlung */
  ...
}
```

Als nächstes muss das Handle des Rootfensters, sprich des Desktops ermittelt werden. Hierzu verwendet man das Macro *RootWindow*, welches eine Variable vom Typ *Window* liefert:

```
rootwindow = RootWindow(display, screen);
```

Jedem Fenster muss eine Farbtabelle zugeordnet werden. Diese wird beim Erzeugen des Fensters zusammen mit weiteren Parametern in einer Attributliste vom Typ *XSetWindowAttributes* übergeben:

```
windowattributes.colormap = XCreateColormap(display, rootwindow,
                                            visualinfo->visual, AllocNone);
```

Um über Veränderungen in der Fenstergrösse informiert zu werden, setzt man in der Attributliste ein entsprechendes Flag:

```
windowattributes.event_mask = StructureNotifyMask;
```

Mit der Funktion *XCreateWindow* erzeugt man nun das Fenster und erhält das entsprechende Handle in Form einer Variable vom Typ *Window*:

```
window = XCreateWindow(display, rootwindow,
                       0,                      /* linke obere Ecke, X-Koord.*/
                       0,                      /* "     "     "     Y-Koord.*/
                       400,                    /* Fensterbreite             */
                       300,                    /* Fensterhöhe               */
                       0,                      /* Breite des Fensterrahmens */
                       visualinfo->depth,      /* Farbtiefe                 */
                       InputOutput,            /* Fensterklasse             */
                       visualinfo->visual,     /* Grafikkontext             */
```

```
                    CWColormap | CWEventMask, /* spezifiziert, welche
                                                 Felder in windowattributes
                                                 gesetzt sind             */
                    &windowattributes);
```

Der Titel des Fensters wird mit der generischen Funktion *XChangeProperty* geändert:

```
XChangeProperty(display, window,
                XA_WM_NAME,      /* der Titel des Fensters wird geändert    */
                XA_STRING,       /* Format des neuen Titels                 */
                8,               /* Bitgrösse eines Zeichens                */
                PropModeReplace, /* Titel ersetzen; ebenfalls möglich:
                                    PropModePrepend: neuen Titel dem be-
                                                     stehenden voranstellen
                                    PropModeAppend: neuen Titel anhängen    */
                (unsigned char*) TITLE,
                                 /* neuer Fenstertitel                      */
                strlen(TITLE));
```

Dem OpenGL-Subsystem muss nun angezeigt werden, dass es seine Darstellung im neu erzeugten Fenster vornehmen soll. Hierzu legt man zunächst einen OpenGL-Kontext an und verknüpft diesen mit dem gewünschten Fenster:

```
/* Kontext anlegen */
context = glXCreateContext(display, visualinfo,
                           NULL,  /* neuen OpenGL-Kontext anlegen */
                           True); /* direkter Hardwarezugriff     */

/* Kontext ins Fenster legen */
glXMakeCurrent(display, window, context);
```

Die *XVisualInfo*-Struktur wird ab hier nicht weiter benötigt, so dass der Speicherbereich freigegeben werden kann:

```
XFree(visualinfo);
```

Um das durch den Anwender ausgelöste Schliessen des Fensters zu verarbeiten, muss dem Window Manager angezeigt werden, das entsprechende Ereigniss an das Programm zu senden. Hierzu bestimmt man mit der Funktion *XInternAtom* zunächst die ID des Ereignisses *WM_DELETE_WINDOW* und speichert sie in einer Variablen vom Typ *Atom*:

```
wm_delete_window_atom = XInternAtom(display, "WM_DELETE_WINDOW", False);
```

Diese Ereigniss-ID wird nun mit dem Fenster verknüpft:

```
XSetWMProtocols(display, window, &wm_delete_window_atom, 1);
```

Per Aufruf von *XMapWindow* wird das neue Fenster letztendlich aktiviert und somit auf dem Bildschirm dargestellt:

```
XMapWindow(display, window);
```

3.4.2. X Window-Ereignisse verarbeiten

Die Funktion *XPending* liefert die Anzahl der unverarbeiteten Events. Dies lässt sich für eine Ereignissschleife nutzen:

```
while (XPending(display)) {
  /* Ereignisse verarbeiten */
```

Mit der Funktion *XNextEvent* holt man sich das nächste Ereigniss aus der Warteschlange und speichert dieses in einer Variablen vom Typ *XEvent*:

```
  XEvent event;

  XNextEvent(display, &event);
```

Das Feld *type* der *XEvent*-Struktur gibt Aufschluss über den Ereignisstyp. Der Wert *ConfigureNotify* zeigt eine Änderung der Fenstergeometrie an, während *ClientMessage* ein generisches Ereigniss bedeutet. Im letzteren Fall muss hier lediglich überprüft werden, ob es sich um das *WM_DELETE_WINDOW*-Event handelt:

```
  switch (event.type) {

    case ConfigureNotify:
      /* Viewport für OpenGL anpassen */
      glViewport(0, 0, event.xconfigure.width, event.xconfigure.height);
      break;

    case ClientMessage:
      /* Will der Anwender das Fenster schliessen? */
      if ((Atom) event.xclient.data.l[0] == wm_delete_window_atom) {
        /* Programm / Thread / Ereignissschleife verlassen */
        ...
      }
      break;

    default:
      /* ohne default-Label meckert der Compiler */
      break;
  }
} /* XPending() */
```

3.4.3. X Window-Fenster schliessen

Um das X11-Fenster zu schliessen, muss lediglich der OpenGL-Kontext zerstört und die Verbindung zum X-Server beendet werden. Zwecks sauberem Programmierstil sollte man auch allozierte Speicherbereiche explizit freigeben, gleichwohl dies der X-Server beim Beenden der Verbindung automatisch vernehmen würde:

```
/* OpenGL-Kontext verwerfen */
glXMakeCurrent(display, 0, NULL);
glXDestroyContext(display, context);

/* Fenster schliessen */
```

```
XDestroyWindow(display, window);

/* Farbtabelle freigeben */
XFreeColormap(display, windowattributes.colormap);

/* Verbindung zum X-Server schliessen */
XCloseDisplay(display);
```

3.5. OpenGL unter X11

Das Absetzen von OpenGL-Befehlen unter X11 unterscheidet sich zu denen bei Verwendung der GLUT-Bibliothek. Lediglich der Aufruf von *glutSwapBuffers* muss durch *glXSwapBuffers* ersetzt werden:

```
glXSwapBuffers(display, window);
```

Somit ist es relativ einfach, OpenGL-Programme von GLUT auf natives X11 zu portieren.

3.6. Threads und Mutexe

Threads sind quasiparallel verlaufende Funktionen innerhalb eines Prozesses. Bildlich gesprochen verhalten sich Threads zum Prozess wie Prozesse zum Betriebssystem. Ein Thread verfügt zwar über einen eigenen Stack, greift aber auf den gemeinsamen globalen Prozessspeicher, sprich Heap zu. Zwar ist echte Parallelität nur auf einem Multiprozessorsystem gegeben, dennoch kann jeder Thread zu jedem Zeitpunkt zugunsten eines anderen Threads unterbrochen werden. Daher müssen beim Zugriff auf gemeinsame Daten besondere Maßnahmen getroffen werden, um Inkonsistenzen zu vermeiden. Mutexe sind atomar veränderbare boolsche Variablen, die entweder gesperrt oder frei sind. Ein Thread darf nur einen nicht gesperrten Mutex sperren. Der Versuch, einen gesperrten Mutex erneut zu sperren, blockiert den Thread solange, bis der Mutex durch einen anderen Thread freigegeben wird. Daher werden Mutexe oftmals zum Schutz kritscher Abschnitte verwendet, z.B. beim Zugriff auf gemeinsame Speicherbereiche. Bei komplexeren Abläufen kann hierbei ohne sorgfältige Planung ein Deadlock entstehen, wenn z.B. zwei Threads auf die Freigabe eines Mutexes warten, den der jeweils andere gesperrt hat.

Der Rumpf einer Threadfunktion ist sehr allgemein gehalten:

```
void *thread_function (void *arg)
{
  ...
  return (void*) (... Rückgabewert ...);
}
```

Mit der Funktion *pthread_create* wird ein Thread angelegt und gestartet; sie ist in *pthread.h* definiert und erwartet vier Parameter:

- einen Zeiger auf eine Variable vom Typ *pthread_t*, die nach erfolgreichem Aufruf von *pthread_create* das Handle für den neu angelegten Thread darstellt
- einen Zeiger auf eine *pthread_attr_t*-Struktur, über die der Thread mit erweiterten Attributen angelegt werden kann, oder *NULL*, falls die Standardoptionen verwendet werden sollen
- die eigentliche Threadfunktion

- einen Void-Zeiger, der der Threadfunktion als Parameter übergeben wird

Somit ergibt sich folgender Aufruf:

```
#include <pthread.h>
pthread_t thread;

if (pthread_create(&thread, NULL, thread_function, NULL)) {
  /* Fehlerbehandlung */
  ...
}
```

Per *pthread_join* kann auf das Beenden eines Threads gewartet werden:

```
void *ergebnis;

if (pthread_join(thread, &ergebnis)) {
  /* Fehlerbehandlung */
  ...
}
```

Die Variable *ergebnis* im obigen Beispiel erhält den Rückgabewert der Funktion *thread_function*. Natürlich muss durch die Ablauflogik gewährleistet sein, dass sich der Thread selbst beendet, z.B. durch ein entsprechendes Flag. Andernfalls blockiert die Funktion *pthread_join* unendlich lange.

Der Umgang mit Mutexen ähnelt sehr dem mit Threads. Per *pthread_mutex_init* legt man einen neuen, nicht gesperrten Mutex an:

```
pthread_mutex_t mutex;

if (pthread_mutex_init(&mutex, NULL)) {
  /* Fehlerbehandlung */
  ...
}
```

Der zweite Parameter kann auf eine *pthread_mutexattr_t*-Struktur zeigen, um auch den Mutex mit erweiterten Attributen anzulegen, oder *NULL*, um die Standardoptionen zu übernehmen. Mittels *pthread_mutex_lock* und *pthread_mutex_unlock* sperrt bzw. entsperrt man einen Mutex:

```
/* kann den aufrufenden Thread blockieren! */
if (pthread_mutex_lock(&mutex)) {
  /* Fehlerbehandlung */
  ...
}
```

Analog:

```
if (pthread_mutex_unlock(&mutex)) {
  /* Fehlerbehandlung */
  ...
}
```

Einen nicht mehr benötigten Mutex löscht man mit der Funktion *pthread_mutex_destroy*:

```
if (pthread_mutex_destroy(&mutex)) {
  /* Fehlerbehandlung */
  ...
}
```

3.7. Makefile

Ein *Makefile* beinhaltet Anweisungsschritte, wie eine Zieldatei aus zugeordneten Quelldateien zu erstellen ist, zum Beispiel:

```
test:   test.c
        gcc -o test test.c
```

Das gezeigte Makefile gibt an, dass die Datei *test* neu zu erstellen ist, sobald die Datei *test.c* ein jüngeres Änderungsdatum aufweist. Die zweite Zeile stellt die hierzu notwendige Anweisung dar, meist ein Compileraufruf. Um ein solches Makefile auszuführen, genügt in der Regel der Aufruf von *make* oder *gmake*. Bei modularer Programmierung kann ein sinnvoll eingerichtetes Makefile Übersetzungszeit sparen. Besteht z.B. das Programm *test* aus den Teilen *modul1* und *modul2*, so muss bei Änderung einer Quelldatei nur ein Modul neu übersetzt und mit dem schon vorhandenen Modul zum Hauptprogramm gelinkt werden:

```
test:   modul1.o modul2.o
        gcc -o test modul1.o modul2.o

modul1.o:       modul1.c
        gcc -c -o modul1.o modul1.c

modul2.o:       modul2.c
        gcc -c -o modul2.o modul2.c
```

Beim ersten Aufruf von *make* würden nun die Dateien *modul1.o*, *modul2.o* und *test* erzeugt. Ändert man dann *modul1.c* und ruft *make* erneut auf, würden nur *modul1.o* und *test* neu kompiliert.

4. Erstellte Plugins

Die im Zuge dieser Ausarbeitung erstellten Plugins obliegen dem gleichen äusseren Aufbau: Die Funktion *playback_start* legt einen Mutex an, sperrt diesen und startet die Funktion *display_thread_func* in einem neuem Thread. Analog entsperrt die Funktion *playback_stop* den Mutex und wartet auf das Beenden das Threads. Die Hauptarbeit findet in *display_thread_func* statt. Hier wird zunächst das X11 Fenster geöffnet und in einer Schleife die Darstellung der Audiodaten per OpenGL vorgenommen, sofern der Mutex zwischenzeitlich durch *render_pcm,freq* freigegeben wurde. Das Beenden dieser Schleife wird durch ein globales Flag gesteuert, welches von *playback_start* auf 1 bzw. von *playback_stop* auf 0 gesetzt wird. Da zwischen zwei Threads Daten ausgetauscht werden, lassen sich globale Variablen nicht vermeiden. Diese sind zwecks besonderer Beachtung mit vorangestelltem *global_* benannt. Zudem sind alle globalen Variablen und Funktionen ausser *get_vplugin_info* als *static* deklariert, so dass sie ausserhalb des Moduls nicht sichtbar sind.

4.1. visual_pcm

Das Plugin *visual_pcm* (Abbildung 3) stellt Stereo-PCM-Daten dar, wobei der linke Kanal in der oberen und der rechte Kanal in der unteren Bildhälfte abgebildet wird. Um einen fliessenden Eindruck zu erwecken, werden zunächst von den 512 von *render_pcm* bereitgestellten Samples je 16 per Mittelwert zusammengefasst. Im Fenster werden nun immer 32 dieser komprimierten Blöcke zu je 32 Samples dargestellt, wobei der älteste jener Blöcke überschrieben wird und somit der Eindruck entsteht, die PCM-Daten würden von rechts nach links durch das Fenster wandern. Um immer die volle Fensterhöhe auszunutzen, werden alle darzustellenden Werten über ein Maximum normiert, welches aus dem grössten bisher vorgekommenen Sampledatum gewonnen wird.

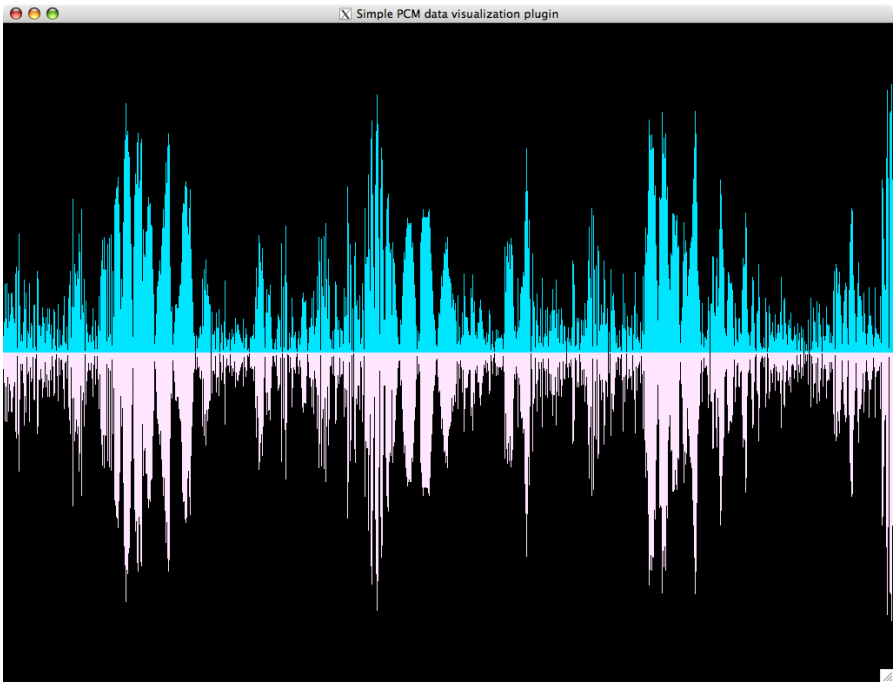

Abbildung 3: *visual_pcm* in Aktion

4.2. visual_freq

Das Plugin *visual_freq* (Abbildung 4) zeigt analog zu *visual_pcm* das Frequenzspektrum der beiden Audiokanäle, allerdings erfolgt hierbei keine weitere Verarbeitung der Daten, sie werden einfach von *render_freq* übernommen und lediglich zur sauberen Darstellung über einen Maximalwert normiert.

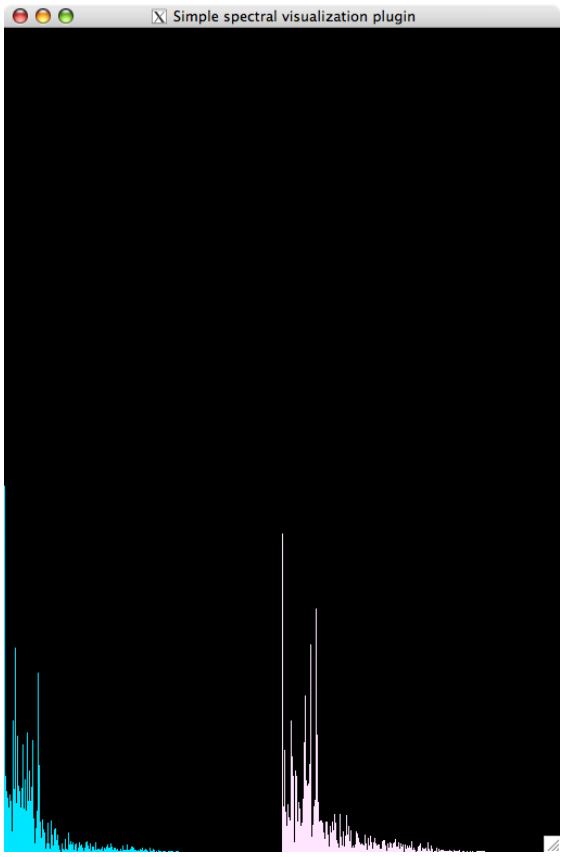

Abbildung 4: Das Modul *visual_freq*

4.3. klick_klack

Das Plugin *klick_klack* basiert auf dem im Praktikum zur Vorlesung GD erstellten Programm *manager*, welches ein statisches Modell des bekannten Kügelchenspiels darstellt. Dieses wurde weitreichend verändert: Die komplette *main*-Routine samt dem GLUT-basierten Eventhandling entfiel, ebenso alle Menüfunktionen. Die interne Sinus-/Cosinus-Lookuptabelle wurde verbessert: Initial werden nur noch Sinuswerte von 0 bis $\frac{\pi}{2}$ in eine statische Tabelle geschrieben, so dass sich Sinuswerte aus anderen Bereichen und Cosinuswerte durch Verschiebung oder Spiegelung ergeben. Ferner wurde die eigentliche Zeichenroutine *manager_display* mit weiteren Parametern versehen, so dass nun folgende Werte übergeben werden müssen:

ypos bestimmt die Position der Kamera in Y-Richtung

zpos wie *ypos*, nur in Z-Richtung

r gibt die Rotation der ganzen Szene in Winkelgrad an

lr bezeichnet die Rotation der Taschenlampe (Spotlight) in Grad

dlr, dlg, dlb steht für den Rot-, Grün- bzw. Blauwert des diffusen Lichts

auslenkung gibt die Auslenkung der rechten Kugel in Winkelgrad an

auslenkung2 dito für die linke Kugel.

Für die dargestellten Nylonfäden der ausgelenkten Kugeln wurde die Funktion *manager_ausgelenkter_faden* hinzugefügt.

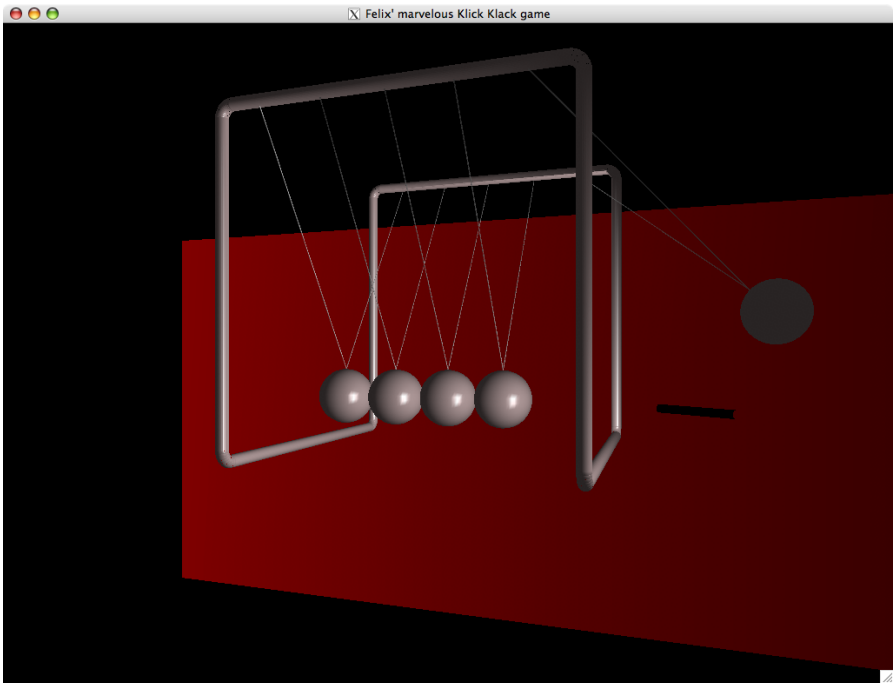

Abbildung 5: Das Modul *klick_klack*

Die Lichtintensität wird in *display_thread_func* aus den Frequenzspektren errechnet unter der Annahme, dass tiefe Frequenzen beim Hörer den Eindruck grösserer Energie erwecken. Es gilt:

$$w = \sum_{k=0}^{1} \sum_{b=0}^{255} f_{k,b} * (256 - i) \tag{2}$$

mit
w: Schallenergie
k: Kanal
b: Frequenzband
f: Intensität des Frequenzbandes wie von *render_freq* bereitgestellt

Der so ermittelte Wert für w wird durch den um den Faktor 100 reduzierten theoretischen Mittelwert für die Schallenergie geteilt, um so zu grosse Werte für die Lichtintensität zu vermeiden. Die Division durch 100 ergab sich als Erfahrungswert, da keines der in der Entwicklungsphase verwendeten Musikstücke an jenen theoretischen Mittelwert heranreichte. Die Lichtintensität wird ferner zur Auslenkung der beiden äussersten Kugeln herangezogen. Befinden sich beide Kugeln in Ruhe, wird eine (die rechte) mit konstanter Winkelgeschwindigkeit von 3° pro Aufruf von *render_freq* bis zum Winkel von

$\frac{w}{w_{max}} * 90°$ ausgelenkt, wobei w_{max} den Maximalwert der bisher aufgetretenen Lichtintensitäten darstellt. Erreicht die Kugel die gewünschte Auslenkung, pendelt sie mit stetig wachsender Geschwindigkeit zurück, bis sie wieder in der Ausgangsposition ist. In der Realität würde nun wegen Energie- und Impulserhaltung die linke Kugel angestossen werden. Im Modell wird sie hingegen mit stetig sinkender Geschwindigkeit ausgelenkt, bis nur noch eine Fortbewegung von unter 1.000001° pro Durchlauf der Hauptschleife erzielt wird (dies war nötig, um Darstellungsfehler mit Float-Werten zu vermeiden). Anschliessend sinkt die linke Kugel mit erneut wachsender Geschwindigkeit in ihre Ruhelage zurück. Nun wird wieder die rechte Kugel ausgelenkt usw. Um diesen Bewegungsablauf der beiden Kugeln abzubilden, wurde mittels eines switch-case-Konstruktes ein simpler Zustandsautomat umgesetzt.

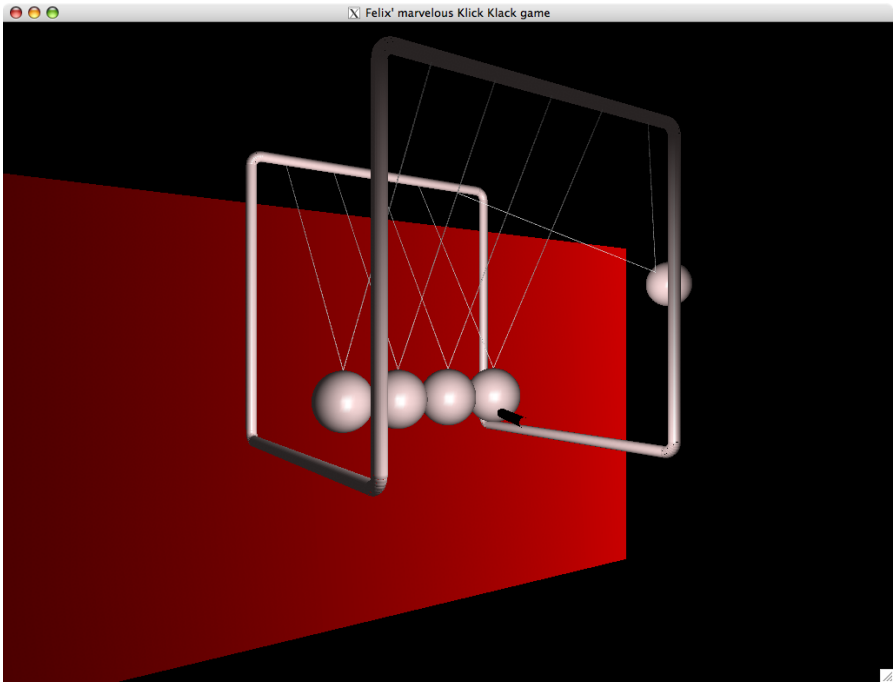

Abbildung 6: Das Modul *klick_klack*, im Vordergrund die angedachte Taschenlampe

Eine absolut realitätsgetreue Darstellung wird nicht erzielt, da zum einen die Nylonfäden im Programm als starre Drähte erscheinen und zum anderen total elastische Stösse zwischen den Kugeln angesetzt werden. Ferner blieben sämtliche Reibungseffekte unbeachtet. Hierzu müsste man ein umfangreiches physikalisches Modell entfernen und z.B. jede Kugel als eigenen Thread ausführen. Grafisch anspruchsvoll wären zudem die in sich gekrümmten Nylonfäden.

A. Literatur

[Birkett 2002] Birkett, Andrew: *XMMS Plugin tutorial at nobugs.org*. Version: Dezember 2002. http://www.xmms.org/docs/vis-plugin.html, Abruf: 2006-09-28

[Diverse 2006] Diverse: *Schnelle Fourier-Transformation*. Version: 19. 2006. http://de.wikipedia.org/wiki/Schnelle_Fourier-Transformation, Abruf: 2006-09-28

[Schebella 2002] Schebella, Marius: *FFT und Pd*. Version: Januar 2002. http://www.parasitaere-kapazitaeten.net/Pd/fft_und_pd.htm, Abruf: 2006-09-28

[Wolf 2004] Wolf, Jürgen: *Linux-Unix-Programmierung*. Version: 2004. http://www.pronix.de/pronix-28.html, Abruf: 2006-09-28

[Wright u. Lipchak 2004] Wright, Richard S. ; Lipchak, Benjamin: *OpenGL Superbible, Third Edition*. 2004. ISBN 0–672–32601–9

B. visual_pcm.c

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <GL/gl.h>
#include <GL/glx.h>
#include "plugin.h"

#define TITLE "Simple PCM data visualization plugin"
#define PCM_SAMPLES 512
#define PCM_CHANNELS 2
#define MERGE 16

#define WIDTH 1024
#define HEIGHT 768

/* globale Variablen */
static gint16 global_pcm_data[PCM_CHANNELS][PCM_SAMPLES];
static pthread_mutex_t global_pcm_data_mutex;
static int global_playing_flag;
static pthread_t display_thread;

/* Forward-Definition */
static void disable_myself (char *);

/* OpenGL-Zeichenroutine im eigenen Thread */
static void *display_thread_func (void *p)
{
  int i, j, channel;
  gint16 pcm_data[PCM_CHANNELS][WIDTH], max;
  gint64 value;

  /* Variablen für X11 */
  Display *xdpy;
  Window root, win;
  XVisualInfo *xvinfo;
  GLXContext context;
  XSetWindowAttributes winattrs;
  int screen, attributes[] = { GLX_RGBA, GLX_DOUBLEBUFFER, 0 };
  XEvent event;
  Atom wm_delete_window_atom;

  /* make compiler happy */
  p = p;

  /* zum X-Server verbinden */
  if (!(xdpy = XOpenDisplay(NULL))) {
```

```
  disable_myself("can not open display");
  return NULL;
}

/* Bildschirm auswählen */
screen = DefaultScreen(xdpy);
if (!(xvinfo = glXChooseVisual(xdpy, screen, attributes))) {
  disable_myself("can not choose visual");
  XCloseDisplay(xdpy);
  return NULL;
}

root = RootWindow(xdpy, screen);

winattrs.colormap = XCreateColormap(xdpy, root, xvinfo->visual,
                                    AllocNone);
winattrs.event_mask = StructureNotifyMask;

/* Fenster anlegen */
win = XCreateWindow(xdpy, root, 0, 0, WIDTH, HEIGHT, 0, xvinfo->depth,
                    InputOutput, xvinfo->visual,
                    CWColormap | CWEventMask,
                    &winattrs);

/* Fenstername */
XChangeProperty(xdpy, win, XA_WM_NAME, XA_STRING, 8, 0,
                (unsigned char*) TITLE, strlen(TITLE));

/* OpenGL-Kontext anlegen */
context = glXCreateContext(xdpy, xvinfo, 0, True);
glXMakeCurrent(xdpy, win, context);

XFree(xvinfo);

/* Handle zum Fenster schliessen */
wm_delete_window_atom = XInternAtom(xdpy, "WM_DELETE_WINDOW", False);
XSetWMProtocols(xdpy, win, &wm_delete_window_atom, 1);

/* Fenster anzeigen */
XMapWindow(xdpy, win);

/* OpenGL initialisieren */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0.0, (GLfloat) WIDTH, 0.0, (GLfloat) HEIGHT, 1.0, -1.0);
glClearColor(0.0, 0.0, 0.0, 0.0);

/* Variablen initialisieren */
max = 1000;
memset(pcm_data, 0, sizeof(gint16) * PCM_SAMPLES * PCM_CHANNELS);
```

```
/* Grafik darstellen */
while (global_playing_flag) {
  /* auf Daten warten */
  pthread_mutex_lock(&global_pcm_data_mutex);

  /* X11 Events verarbeiten */
  while (XPending(xdpy)) {
    XNextEvent(xdpy, &event);
    switch (event.type) {
      /* Fenstergrösse wurde verändert */
      case ConfigureNotify:
        glViewport(0, 0, event.xconfigure.width,
                   event.xconfigure.height);
        break;
      /* Fenster schliessen? */
      case ClientMessage:
        if ((Atom) event.xclient.data.l[0] == wm_delete_window_atom)
          disable_myself(NULL);
        break;
      default:
        break;
    }
  }

  /* alte Daten schieben */
  for (channel = 0; channel < PCM_CHANNELS; ++channel)
    memmove(&pcm_data[channel][0],
            &pcm_data[channel][PCM_SAMPLES / MERGE],
            sizeof(gint16) * (WIDTH - PCM_SAMPLES / MERGE));

  /* neue Daten zusammenfassen, in lokale Variable schreiben und
     dabei das Maximum bestimmen */
  for (channel = 0; channel < PCM_CHANNELS; ++channel)
    for (i = 0; i < PCM_SAMPLES / MERGE; ++i) {
      value = 0;
      for (j = 0; j < MERGE; ++j)
        value += global_pcm_data[channel][i * MERGE + j];
      value /= MERGE;
      if (value < 0) value *= -1;
      pcm_data[channel][WIDTH - PCM_SAMPLES / MERGE + i] = value;
      if (max < value) max = value;
    }

  /* PCM-Daten anzeigen */
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(0.0, 0.9, 1.0);
  for (i = 0; i < WIDTH; ++i)
    glRectf((GLfloat) i,
            HEIGHT / 2.0 + pcm_data[0][i] * HEIGHT / (2.0 * max),
```

```
                (GLfloat) (i + 1), HEIGHT / 2.0);

      glColor3f(1.0, 0.9, 1.0);
      for (i = 0; i < WIDTH; ++i)
        glRectf((GLfloat) i, HEIGHT / 2.0,
                (GLfloat) (i + 1),
                HEIGHT / 2.0 - pcm_data[0][i] * HEIGHT / (2.0 * max));

      glXSwapBuffers(xdpy, win);
    }

    /* aufräumen, Fenster schliessen, Verbindung zum X-Server schliessen */
    glXMakeCurrent(xdpy, 0, NULL);
    glXDestroyContext(xdpy, context);
    XDestroyWindow(xdpy, win);
    XFreeColormap(xdpy, winattrs.colormap);
    XCloseDisplay(xdpy);

    return NULL;
}

static void playback_start ()
{
    global_playing_flag = 1;
    pthread_mutex_init(&global_pcm_data_mutex, NULL);
    pthread_mutex_lock(&global_pcm_data_mutex);
    pthread_create(&display_thread, NULL, display_thread_func, NULL);
}

static void playback_stop ()
{
    global_playing_flag = 0;
    pthread_mutex_unlock(&global_pcm_data_mutex);
    pthread_join(display_thread, NULL);
    pthread_mutex_destroy(&global_pcm_data_mutex);
}

static void render_pcm (gint16 pcm_data[2][PCM_SAMPLES])
{
    memcpy(global_pcm_data, pcm_data,
           sizeof(gint16) * PCM_SAMPLES * PCM_CHANNELS);
    pthread_mutex_unlock(&global_pcm_data_mutex);
}

static VisPlugin plugin = {
    .handle              = NULL,
    .filename            = NULL,
    .xmms_session        = 0,
    .description         = TITLE,
    .num_pcm_chs_wanted  = PCM_CHANNELS,
```

```
  .num_freq_chs_wanted = 0,
  .init                = NULL,
  .cleanup             = NULL,
  .about               = NULL,
  .configure           = NULL,
  .disable_plugin      = NULL,
  .playback_start      = playback_start,
  .playback_stop       = playback_stop,
  .render_pcm          = render_pcm,
  .render_freq         = NULL,
};

VisPlugin* get_vplugin_info ()
{
  return &plugin;
}

static void disable_myself (char *msg)
{
  if (msg) fputs(msg, stderr);
  if (plugin.disable_plugin) plugin.disable_plugin(&plugin);
}
```

C. visual_freq.c

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <GL/gl.h>
#include <GL/glx.h>
#include "plugin.h"

#define TITLE "Simple spectral visualization plugin"
#define FREQ_SAMPLES 256
#define FREQ_CHANNELS 2

#define WIDTH (2 * FREQ_SAMPLES)
#define HEIGHT 768

/* globale Variablen */
static gint16 global_freq_data[FREQ_CHANNELS][FREQ_SAMPLES];
static pthread_mutex_t global_freq_data_mutex;
static int global_playing_flag;
static pthread_t display_thread;

/* Forward-Definition */
static void disable_myself (char *);

/* OpenGL-Zeichenroutine im eigenen Thread */
static void *display_thread_func (void *p)
{
  int i, channel;
  gint16 freq_data[FREQ_CHANNELS][FREQ_SAMPLES], max;

  /* Variablen für X11 */
  Display *xdpy;
  Window root, win;
  XVisualInfo *xvinfo;
  GLXContext context;
  XSetWindowAttributes winattrs;
  int screen, attributes[] = { GLX_RGBA, GLX_DOUBLEBUFFER, 0 };
  XEvent event;
  Atom wm_delete_window_atom;

  /* make compiler happy */
  p = p;

  /* zum X-Server verbinden */
  if (!(xdpy = XOpenDisplay(NULL))) {
    disable_myself("can not open display");
    return NULL;
```

```
}

/* Bildschirm auswählen */
screen = DefaultScreen(xdpy);
if (!(xvinfo = glXChooseVisual(xdpy, screen, attributes))) {
  disable_myself("can not open visual");
  XCloseDisplay(xdpy);
  return NULL;
}

root = RootWindow(xdpy, screen);

winattrs.colormap = XCreateColormap(xdpy, root, xvinfo->visual,
                                    AllocNone);
winattrs.event_mask = StructureNotifyMask;

/* Fenster anlegen */
win = XCreateWindow(xdpy, root, 0, 0, WIDTH, HEIGHT, 0, xvinfo->depth,
                    InputOutput, xvinfo->visual,
                    CWColormap | CWEventMask,
                    &winattrs);

/* Fenstername */
XChangeProperty(xdpy, win, XA_WM_NAME, XA_STRING, 8, 0,
                (unsigned char*) TITLE, strlen(TITLE));

/* OpenGL-Kontext anlegen */
context = glXCreateContext(xdpy, xvinfo, 0, True);
glXMakeCurrent(xdpy, win, context);

XFree(xvinfo);

/* Handle zum Fenster schliessen */
wm_delete_window_atom = XInternAtom(xdpy, "WM_DELETE_WINDOW", False);
XSetWMProtocols(xdpy, win, &wm_delete_window_atom, 1);

/* Fenster anzeigen */
XMapWindow(xdpy, win);

/* OpenGL initialisieren */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0.0, (GLfloat) WIDTH, 0.0, (GLfloat) HEIGHT, 1.0, -1.0);
glClearColor(0.0, 0.0, 0.0, 0.0);

/* Variablen initialisieren */
max = 1000;
memset(freq_data, 0, sizeof(gint16) * FREQ_SAMPLES * FREQ_CHANNELS);

/* Grafik darstellen */
```

```
  while (global_playing_flag) {
    /* auf Daten warten */
    pthread_mutex_lock(&global_freq_data_mutex);

    /* X11 Events verarbeiten */
    while (XPending(xdpy)) {
      XNextEvent(xdpy, &event);
      switch (event.type) {
        /* Fenstergrösse wurde verändert */
        case ConfigureNotify:
          glViewport(0, 0, event.xconfigure.width,
                     event.xconfigure.height);
          break;
        /* Fenster schliessen? */
        case ClientMessage:
          if ((Atom) event.xclient.data.l[0] == wm_delete_window_atom)
            disable_myself(NULL);
          break;
        default:
          break;
      }
    }

    /* Frequenzspektren in lokale Variable kopieren und
       dabei das Maximum bestimmen */
    for (channel = 0; channel < FREQ_CHANNELS; ++channel)
      for (i = 0; i < FREQ_SAMPLES; ++i) {
        freq_data[channel][i] = global_freq_data[channel][i];
        if (freq_data[channel][i] < 0) freq_data[channel][i] = 0;
        if (max < freq_data[channel][i]) max = freq_data[channel][i];
      }

    /* Spektren anzeigen */
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 0.9, 1.0);
    for (channel = 0; channel < FREQ_CHANNELS; ++channel) {
      for (i = 0; i < FREQ_SAMPLES; ++i)
        glRectf((GLfloat) (channel * FREQ_SAMPLES + i),
                (GLfloat) (freq_data[channel][i] * HEIGHT / max),
                (GLfloat) (channel * FREQ_SAMPLES + i + 1), 0.0);

      glColor3f(1.0, 0.9, 1.0);
    }

    glXSwapBuffers(xdpy, win);
  }

  /* aufräumen, Fenster schliessen, Verbindung zum X-Server schliessen */
  glXMakeCurrent(xdpy, 0, NULL);
  glXDestroyContext(xdpy, context);
```

```
  XDestroyWindow(xdpy, win);
  XFreeColormap(xdpy, winattrs.colormap);
  XCloseDisplay(xdpy);

  return NULL;
}

static void playback_start ()
{
  global_playing_flag = 1;
  pthread_mutex_init(&global_freq_data_mutex, NULL);
  pthread_mutex_lock(&global_freq_data_mutex);
  pthread_create(&display_thread, NULL, display_thread_func, NULL);
}

static void playback_stop ()
{
  global_playing_flag = 0;
  pthread_mutex_unlock(&global_freq_data_mutex);
  pthread_join(display_thread, NULL);
  pthread_mutex_destroy(&global_freq_data_mutex);
}

static void render_freq (gint16 freq_data[2][FREQ_SAMPLES])
{
  memcpy(global_freq_data, freq_data,
         sizeof(gint16) * FREQ_SAMPLES * FREQ_CHANNELS);
  pthread_mutex_unlock(&global_freq_data_mutex);
}

static VisPlugin plugin = {
  .handle              = NULL,
  .filename            = NULL,
  .xmms_session        = 0,
  .description         = TITLE,
  .num_pcm_chs_wanted  = 0,
  .num_freq_chs_wanted = FREQ_CHANNELS,
  .init                = NULL,
  .cleanup             = NULL,
  .about               = NULL,
  .configure           = NULL,
  .disable_plugin      = NULL,
  .playback_start      = playback_start,
  .playback_stop       = playback_stop,
  .render_pcm          = NULL,
  .render_freq         = render_freq,
};

VisPlugin* get_vplugin_info ()
{
```

```
  return &plugin;
}

static void disable_myself (char *msg)
{
  if (msg) fputs(msg, stderr);
  if (plugin.disable_plugin) plugin.disable_plugin(&plugin);
}
```

D. klick_klack.c

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <GL/gl.h>
#include <GL/glx.h>
#include "plugin.h"

#define TITLE "Felix' marvelous Klick Klack game"
#define FREQ_SAMPLES 256
#define FREQ_CHANNELS 2

/* globale Variablen */
static gint16 global_freq_data[FREQ_CHANNELS][FREQ_SAMPLES];
static pthread_mutex_t global_freq_data_mutex;
static int global_playing_flag;
static pthread_t display_thread;

/* in manager.c */
extern void manager_init ();
extern void manager_display (GLfloat, GLfloat, GLfloat, GLfloat, GLfloat,
                             GLfloat, GLfloat, GLfloat, GLfloat);
extern void manager_reshape (int, int);

/* Forward-Definition */
static void disable_myself (char *);

/* OpenGL-Zeichenroutine im eigenen Thread */
static void *display_thread_func (void *p)
{
  int i, channel;
  gint16 freq_data[FREQ_CHANNELS][FREQ_SAMPLES];
  guint64 power, rotation;
  GLfloat light, new_light, max_light,
          auslenkung, auslenkung2, auslenkung_speed, auslenkung_ziel;
  enum {
    RECHTE_KUGEL_STEIGT,
    RECHTE_KUGEL_SINKT,
    LINKE_KUGEL_STEIGT,
    LINKE_KUGEL_SINKT,
  } status;

  /* Variablen für X11 */
  Display *xdpy;
  Window root, win;
  XVisualInfo *xvinfo;
  GLXContext context;
```

```
  XSetWindowAttributes winattrs;
  int screen, attributes[] = { GLX_RGBA, GLX_DOUBLEBUFFER,
                               GLX_DEPTH_SIZE, 16, 0 };
  XEvent event;
  Atom wm_delete_window_atom;

  /* make compiler happy */
  p = p;

  /* zum X-Server verbinden */
  if (!(xdpy = XOpenDisplay(NULL))) {
    disable_myself("can not open display");
    return NULL;
  }

  /* Bildschirm auswählen */
  screen = DefaultScreen(xdpy);
  if (!(xvinfo = glXChooseVisual(xdpy, screen, attributes))) {
    disable_myself("can not open visual");
    XCloseDisplay(xdpy);
    return NULL;
  }

  root = RootWindow(xdpy, screen);

  winattrs.colormap = XCreateColormap(xdpy, root, xvinfo->visual,
                                      AllocNone);
  winattrs.event_mask = StructureNotifyMask;

  /* Fenster anlegen */
  win = XCreateWindow(xdpy, root, 0, 0, 1024, 768, 0, xvinfo->depth,
                      InputOutput, xvinfo->visual,
                      CWColormap | CWEventMask,
                      &winattrs);

  /* Fenstername */
  XChangeProperty(xdpy, win, XA_WM_NAME, XA_STRING, 8, 0,
                  (unsigned char*) TITLE, strlen(TITLE));

  /* OpenGL-Kontext anlegen */
  context = glXCreateContext(xdpy, xvinfo, 0, True);
  glXMakeCurrent(xdpy, win, context);

  XFree(xvinfo);

  /* Handle zum Fenster schliessen */
  wm_delete_window_atom = XInternAtom(xdpy, "WM_DELETE_WINDOW", False);
  XSetWMProtocols(xdpy, win, &wm_delete_window_atom, 1);

  /* Fenster anzeigen */
```

```
  XMapWindow(xdpy, win);

  /* Klick-Klack Spiel initialisieren */
  manager_init();
  manager_reshape(1024, 768);

  /* Variablen initialisieren */
  light = 0.0f;
  rotation = 600;
  auslenkung = auslenkung2 = auslenkung_speed = auslenkung_ziel = 0.0f;
  status = RECHTE_KUGEL_STEIGT;
  max_light = 1.0f;
  memset(freq_data, 0, sizeof(gint16) * FREQ_SAMPLES * FREQ_CHANNELS);

  /* Grafik darstellen */
  while (global_playing_flag) {
    /* auf Daten warten */
    pthread_mutex_lock(&global_freq_data_mutex);

    /* X11 Events verarbeiten */
    while (XPending(xdpy)) {
      XNextEvent(xdpy, &event);
      switch (event.type) {
        /* Fenstergrösse wurde verändert */
        case ConfigureNotify:
          manager_reshape(event.xconfigure.width,
                          event.xconfigure.height);
          break;
        /* Fenster schliessen? */
        case ClientMessage:
          if ((Atom) event.xclient.data.l[0] == wm_delete_window_atom)
            disable_myself(NULL);
          break;
        default:
          break;
      }
    }

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 0.9, 1.0);

    power = 0;

    /* Frequenzspektren in lokale Variable kopieren,
       dabei Energie bestimmen */
    for (channel = 0; channel < FREQ_CHANNELS; ++channel)
      for (i = 0; i < FREQ_SAMPLES; ++i) {
        freq_data[channel][i] = global_freq_data[channel][i];
        if (freq_data[channel][i] < 0) freq_data[channel][i] = 0;
        power += freq_data[channel][i] * (256 - i);
```

```
    }

  /* Lichtintensität bestimmen */
  new_light = power / (65792.0 * 655.360);
  if (new_light >= light) light = new_light;
  else if (light > 0.05) light -= 0.05;
  else light = 0.0;

  if (light > max_light) max_light = light;

  switch (status) {
    case RECHTE_KUGEL_STEIGT:
      auslenkung += 3.0f;
      if (auslenkung >= auslenkung_ziel) {
        status = RECHTE_KUGEL_SINKT;
        auslenkung_speed = 1.0f;
      }
      break;
    case RECHTE_KUGEL_SINKT:
      auslenkung_speed *= 1.2f;
      auslenkung -= auslenkung_speed;
      if (auslenkung <= 0.0f) {
        auslenkung = 0.0f;
        status = LINKE_KUGEL_STEIGT;
      }
      break;
    case LINKE_KUGEL_STEIGT:
      auslenkung_speed /= 1.2f;
      auslenkung2 += auslenkung_speed;
      if (auslenkung_speed <= 1.000001f) status = LINKE_KUGEL_SINKT;
      break;
    case LINKE_KUGEL_SINKT:
      auslenkung_speed *= 1.2;
      auslenkung2 -= auslenkung_speed;
      if (auslenkung2 <= 0.0f) {
        auslenkung2 = 0.0f;
        status = RECHTE_KUGEL_STEIGT;
        auslenkung_ziel = light / max_light * 90.0f;
      }
      break;
  }

  /* Klick-klack Spiel zeichnen */
  manager_display(-1.0, -7.0, 0.1 * rotation, 0.2 * rotation,
                  light, light, light, auslenkung, auslenkung2);
  if ((rotation += 5) >= 3600) rotation = 0;

  glXSwapBuffers(xdpy, win);
}
```

```
  /* aufräumen, Fenster schliessen, Verbindung zum X-Server schliessen */
  glXMakeCurrent(xdpy, 0, NULL);
  glXDestroyContext(xdpy, context);
  XDestroyWindow(xdpy, win);
  XFreeColormap(xdpy, winattrs.colormap);
  XCloseDisplay(xdpy);

  return NULL;
}

static void playback_start ()
{
  global_playing_flag = 1;
  pthread_mutex_init(&global_freq_data_mutex, NULL);
  pthread_mutex_lock(&global_freq_data_mutex);
  pthread_create(&display_thread, NULL, display_thread_func, NULL);
}

static void playback_stop ()
{
  global_playing_flag = 0;
  pthread_mutex_unlock(&global_freq_data_mutex);
  pthread_join(display_thread, NULL);
  pthread_mutex_destroy(&global_freq_data_mutex);
}

static void render_freq (gint16 freq_data[FREQ_CHANNELS][FREQ_SAMPLES])
{
  memcpy(global_freq_data, freq_data,
         sizeof(gint16) * FREQ_SAMPLES * FREQ_CHANNELS);
  pthread_mutex_unlock(&global_freq_data_mutex);
}

static VisPlugin plugin = {
  .handle              = NULL,
  .filename            = NULL,
  .xmms_session        = 0,
  .description         = TITLE,
  .num_pcm_chs_wanted  = 0,
  .num_freq_chs_wanted = FREQ_CHANNELS,
  .init                = NULL,
  .cleanup             = NULL,
  .about               = NULL,
  .configure           = NULL,
  .disable_plugin      = NULL,
  .playback_start      = playback_start,
  .playback_stop       = playback_stop,
  .render_pcm          = NULL,
  .render_freq         = render_freq
};
```

```
VisPlugin* get_vplugin_info ()
{
  return &plugin;
}

static void disable_myself (char *msg)
{
  if (msg) fputs(msg, stderr);
  if (plugin.disable_plugin) plugin.disable_plugin(&plugin);
}
```

E. manager.c

```
#include <stdio.h>
#include <math.h>
#include <GL/gl.h>
#ifdef MACOS
#  include <GLUT/glut.h>
#else
#  include <GL/glut.h>
#endif

#define TABLE_SIZE 9000

enum Listobjects {
  LIST_ROPE,
  LIST_BAR,
  LIST_CURVE,
  LIST_MAX,
};

static GLfloat diffuseLight[4] = { 1.0f, 1.0f, 1.0f, 1.0f };

static const GLfloat myWidth = 1.0f,
                     myHeight = 1.0f;

static GLuint lists[LIST_MAX];

static GLfloat* mySin_init () {
  static GLfloat table[TABLE_SIZE];
  unsigned int i;

  for (i = 0; i < TABLE_SIZE; ++i)
    table[i] = sin(M_PI * i * 0.5f / TABLE_SIZE);
  return table;
}

static GLfloat mySin (GLfloat deg) {
  static GLfloat *table;
  int degree, degree2;

  if (!table) table = mySin_init();
  degree = ((int) (deg * TABLE_SIZE / 90)) % (4 * TABLE_SIZE);
  degree2 = 1 + ((int) (deg * TABLE_SIZE / 90)) % TABLE_SIZE;
  if (degree >= 3 * TABLE_SIZE) return -table[TABLE_SIZE - degree2];
  if (degree >= 2 * TABLE_SIZE) return -table[degree2];
  if (degree >= TABLE_SIZE) return table[TABLE_SIZE - degree2];
  return table[degree2];
}

static GLfloat myCos (GLfloat deg) {
```

```
  return mySin(deg + 90.0f);
}

/* malt ein Rohr der Länge l entlang der x-Achse,
 * Schwerpunkt im Ursprung,
 * mit Radius r und Tesselationfaktor d
 */
static void myTube (GLfloat l, GLfloat r, GLint d) {
  GLfloat p[4][3], n[4][3], angle;
  GLint i;

  glPushMatrix();

  /* Idee: Vektoren p[4] definieren eine "Deckplatte" des Rohres */
  /* bzw. im Querschnitt ein "Tortendreieck"
*/

  /* x-Koordinaten der Eckpunkte */
  p[0][0] = l / 2;
  p[1][0] = l / 2;
  p[2][0] = -l / 2;
  p[3][0] = -l / 2;

  /* ...der Normalenvektoren */
  n[0][0] = 0.0f;
  n[1][0] = 0.0f;
  n[2][0] = 0.0f;
  n[3][0] = 0.0f;

  /* y-Koordinaten der Eckpunkte */
  p[0][1] = myCos(360.0f / d) * r;
  p[1][1] = p[0][1];
  p[2][1] = p[0][1];
  p[3][1] = p[0][1];

  /* ...der Normalenvektoren */
  n[0][1] = p[0][1];
  n[1][1] = p[1][1];
  n[2][1] = p[2][1];
  n[3][1] = p[3][1];

  /* z-Koordinaten der Eckpunkte */
  p[0][2] = mySin(360.0f / d) * r;
  p[1][2] = -p[0][2];
  p[2][2] = -p[0][2];
  p[3][2] = p[0][2];

  /* ...der Normalenvektoren */
  n[0][2] = p[0][2];
  n[1][2] = p[1][2];
```

```
  n[2][2] = p[2][2];
  n[3][2] = p[3][2];

  angle = 360.0f / d;

  for (i = 0; i < d; ++i) {
    glBegin(GL_QUADS);
      glNormal3fv(n[0]);
      glVertex3fv(p[0]);
      glNormal3fv(n[1]);
      glVertex3fv(p[1]);
      glNormal3fv(n[2]);
      glVertex3fv(p[2]);
      glNormal3fv(n[3]);
      glVertex3fv(p[3]);
    glEnd();
    /* TortenstÃ¼ck um die x-Achse weiterdrehen */
    glRotatef(angle, 1.0f, 0.0f, 0.0f);
  }

  glPopMatrix();
}

/* konstruiert ein um 90 Grad gebogenes Eckstück eines Rohres
 * Radius r1 der Hauptkrümmung, Radius r2 des Rohres (=r1 von myTube())
 * Tesselationfaktor d
 * Mittelpunktslinie beginnt im Ursprung und verläuft in der positiven
 * xy-Ebene endet bei x=r1, y=r1, z=0
 */
static void myMuffenStuck (GLfloat r1, GLfloat r2, GLint d) {
  GLfloat p[4][3], n[4][3], angle, angle_deg, costmp, sintmp,
          sintmp_r1, costmp_r1;
  GLint i, j;

  glPushMatrix();

  angle = 2.0f * M_PI / d;
  angle_deg = 360.0f / d;
  costmp = 1.0f - myCos(angle_deg);
  sintmp = mySin(angle_deg);
  sintmp_r1 = sintmp * r1; /* = r1 * sin(2 Pi / d)      */
  costmp_r1 = costmp * r1; /* = r1 - r1 * cos(2 Pi / d)
                              = Höhenanstieg durch Hauptkrümmung
                                pro Deckplatte */

  /* Idee: Vektoren p[4] definieren "Deckplatten" des gekrümmten Rohres
   * zur Vorstellung gehen wir von einem Punkt mit grossem positiven y-Wert
   * aus und blicken in negative y-Richtung auf die xz-Ebene
   * jede Deckplatte wird dann ccw definiert mit p[0] links unten
   * der Normalenvektor für jeden Vertex ergibt sich als Differenz zwischen
```

```
   * dem Vertex und dem korrelierenden Punkt auf der Mittelpunktslinie
   */
  for (i = 0; i < d/4; ++i) { /* Schleife für Hauptkrümmung
*/
    p[1][2] = 0.0f;           /* z-Koordinate rechte untere Ecke
*/
    p[1][1] = r2;             /* y-Koordinate rechte untere Ecke
*/
    for (j = 0; j < d; ++j) { /* Schleife für Rohr
*/
      /* z */
      /* linke untere Ecke ist rechte untere Ecke der Vorgängerplatte */
      p[0][2] = p[1][2];
      /* rechte untere Ecke (im Querschnitt wie dreieckiges Tortenstück */
      p[1][2] = mySin(angle_deg * (j + 1)) * r2;
      /* rechte obere = rechte untere Ecke (da parallel zur x-Achse) */
      p[2][2] = p[1][2];
      /* linke obere = linke untere Ecke */
      p[3][2] = p[0][2];

      /* y */
      /* linke untere Ecke ist rechte untere Ecke der Vorgängerplatte */
      p[0][1] = p[1][1];
      /* rechte untere Ecke (im Querschnitt wie dreieckiges Tortenstück */
      p[1][1] = myCos(angle_deg * (j + 1)) * r2;
      /* rechte obere = rechte untere Ecke (da parallel zur x-Achse) */
      p[2][1] = p[1][1];
      /* PLUS Höhenanstieg durch Hauptkrümmung */
      p[2][1] += costmp_r1 - p[2][1] * costmp;

      /* linke obere = linke untere Ecke */
      p[3][1] = p[0][1];
      /* PLUS Höhenanstieg durch Hauptkrümmung */
      p[3][1] += costmp_r1 - p[3][1] * costmp;

      /* x */
      /* Hauptlinie beginnt im Ursprung */
      p[0][0] = 0.0f;
      p[1][0] = 0.0f;
      /* x-Koordinaten der rechten und linken oberen Ecke jetzt in xy-Ebene
         betrachtet ergibt Tortendreick der Hauptkrümmung UNTER Beachtung
         der Höhe des jeweiligen Punktes
*/
      p[2][0] = sintmp_r1 - p[2][1] * sintmp;
      p[3][0] = sintmp_r1 - p[3][1] * sintmp;

      /* Normalenvektoren */
      /* z = z des Vertex */
      n[0][2] = p[0][2];
      n[1][2] = p[1][2];
```

```
      n[2][2] = p[2][2];
      n[3][2] = p[3][2];

      /* y = y des Vertex - r1*(1-cos) */
      n[0][1] = p[0][1] - costmp_r1;
      n[1][1] = p[1][1] - costmp_r1;
      n[2][1] = p[2][1] - costmp_r1;
      n[3][1] = p[3][1] - costmp_r1;

      /* x = x des Vertex - r1*sin */
      n[0][0] = p[0][0] - sintmp_r1;
      n[1][0] = p[1][0] - sintmp_r1;
      n[2][0] = p[2][0] - sintmp_r1;
      n[3][0] = p[3][0] - sintmp_r1;

      glBegin(GL_QUADS);
        glNormal3fv(n[0]);
        glVertex3fv(p[0]);
        glNormal3fv(n[1]);
        glVertex3fv(p[1]);
        glNormal3fv(n[2]);
        glVertex3fv(p[2]);
        glNormal3fv(n[3]);
        glVertex3fv(p[3]);
      glEnd();
    }
    /* in xy-Ebene nach rechts oben verschieben... */
    glTranslatef(sintmp_r1, costmp_r1, 0.0f);
    /* ...und passend reindrehen                   */
    glRotatef(angle_deg, 0.0f, 0.0f, 1.0f);
  }

  glPopMatrix();
}

void manager_ausgelenkter_faden (GLfloat auslenkung, int pos)
{
  glPushMatrix();
  glColor4f(1.0f, 1.0f, 1.0f, 0.55f);
  glTranslatef(0.275f,
    1.06f - myCos(auslenkung) * 0.275f * mySin(55.0f) / myCos(55.0f),
    mySin(auslenkung) * 0.275f * mySin(55.0f) / myCos(55.0f) * pos);
  glRotatef(55.0f, 0.0f, 0.0f, 1.0f);
  glRotatef(auslenkung * pos, -myCos(55.0f), mySin(55.0f), 0.0f);
  myTube(0.55f / myCos(55.0f), 0.002f, 18);
  glPopMatrix();
}

void manager_display (GLfloat ypos, GLfloat zpos, GLfloat r, GLfloat lr,
                      GLfloat dlr, GLfloat dlg, GLfloat dlb,
```

```
                        GLfloat auslenkung, GLfloat auslenkung2)
{
  GLfloat scale = 10.0f - fabs(zpos);
  GLfloat posLight[4] = { .0f, 0.20f, .0f, 1.0f },
    dirLight[3] = { .0f, -.20f, .0f };

  diffuseLight[0] = dlr;
  diffuseLight[1] = dlg;
  diffuseLight[2] = dlb;

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  glTranslatef(0.0f, ypos, zpos);
  glRotatef(r, 0.0f, 1.0f, 0.0f);
  glScalef(scale, scale, scale);

  glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
  /* Licht */
  posLight[0] = mySin(lr) * 1.0f;
  posLight[2] = myCos(lr) * 1.0f;
  dirLight[0] = -posLight[0] / (sqrt(posLight[0] * posLight[0] +
                                     posLight[2] * posLight[2]));
  dirLight[2] = -posLight[2] / (sqrt(posLight[0] * posLight[0] +
                                     posLight[2] * posLight[2]));
  glLightfv(GL_LIGHT0, GL_POSITION, posLight);
  glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, dirLight);

  /* Taschenlampe */
  glPushMatrix();
  glTranslatef(posLight[0], posLight[1], posLight[2]);
  glRotatef(lr-90.0f, 0.0f, 1.0f, 0.0f);
  glColor3f(0.0f, 0.0f, 0.0f);
  myTube(0.2f, 0.01f, 36);
  glPopMatrix();

  /* linke Wand */
  glColor3f(1.0f, 0.0f, 0.0f);
  glBegin(GL_POLYGON);
    glNormal3f(1.0f, 0.0f, 0.0f);
    glVertex3f(-myWidth, -myHeight, +2.0f);
    glVertex3f(-myWidth, -myHeight, -2.0f);
    glVertex3f(-myWidth, +myHeight, -2.0f);
    glVertex3f(-myWidth, +myHeight, +2.0f);
  glEnd();
  /* hintere Wand */
  glBegin(GL_POLYGON);
    glNormal3f(0.0f, 0.0f, 1.0f);
```

```
    glVertex3f(-myWidth, -myHeight, -1.90f);
    glVertex3f(+myWidth, -myHeight, -1.90f);
    glVertex3f(+myWidth, +myHeight, -1.90f);
    glVertex3f(-myWidth, +myHeight, -1.90f);
  glEnd();
  /* rechte Wand */
  glBegin(GL_POLYGON);
    glNormal3f(-1.0f, 0.0f, 0.0f);
    glVertex3f(+myWidth, -myHeight, -2.0f);
    glVertex3f(+myWidth, -myHeight, +2.0f);
    glVertex3f(+myWidth, +myHeight, +2.0f);
    glVertex3f(+myWidth, +myHeight, -2.0f);
  glEnd();
  glPushMatrix();

  /* Gestell */
  /* hinten unten */
  glTranslatef(0.0f, 0.0f, -0.54f);
  glCallList(lists[LIST_BAR]);
  glTranslatef(0.5f, 0.0f, 0.0f);
  glCallList(lists[LIST_CURVE]);
  glTranslatef(-1.0f, 0.0f, 0.0f);
  glRotatef(180.0f, 0.0f, 1.0f, 0.0f);
  glCallList(lists[LIST_CURVE]);

  /* vorne unten */
  glTranslatef(-0.5f, 0.0f, -1.08f);
  glCallList(lists[LIST_BAR]);
  glTranslatef(0.5f, 0.0f, 0.0f);
  glCallList(lists[LIST_CURVE]);
  glTranslatef(-1.0f, 0.0f, 0.0f);
  glRotatef(180.0f, 0.0f, 1.0f, 0.0f);
  glCallList(lists[LIST_CURVE]);

  glPopMatrix();
  glPushMatrix();

  /* hinten rechts */
  glTranslatef(0.54f, 0.54f, -0.54f);
  glRotatef(90.0f, 0.0f, 0.0f, 1.0f);
  glCallList(lists[LIST_BAR]);

  /* hinten links */
  glTranslatef(0.0f, 1.08f, 0.0f);
  glCallList(lists[LIST_BAR]);

  /* vorne links */
  glTranslatef(0.0f, 0.0f, 1.08f);
  glCallList(lists[LIST_BAR]);
```

```
/* vorne rechts */
glTranslatef(0.0f, -1.08f, 0.0f);
glCallList(lists[LIST_BAR]);

glPopMatrix();
glPushMatrix();

/* links oben */
glTranslatef(-0.54f, 1.08f, 0.0f);
glRotatef(90.0f, 0.0f, 1.0f, 0.0f);
glCallList(lists[LIST_BAR]);
glTranslatef(-0.54f, -0.04f, 0.0f);
glRotatef(180.0f, 1.0f, 1.0f, 0.0f);
glCallList(lists[LIST_CURVE]);
glTranslatef(0.0f, 1.08f, 0.0f);
glRotatef(180.0f, 1.0f, 0.0f, 0.0f);
glCallList(lists[LIST_CURVE]);

glPopMatrix();
glPushMatrix();

/* rechts oben */
glTranslatef(0.54f, 1.08f, 0.0f);
glRotatef(90.0f, 0.0f, 1.0f, 0.0f);
glCallList(lists[LIST_BAR]);
glTranslatef(-0.54f, -0.04f, 0.0f);
glRotatef(180.0f, 1.0f, 1.0f, 0.0f);
glCallList(lists[LIST_CURVE]);
glTranslatef(0.0f, 1.08f, 0.0f);
glRotatef(180.0f, 1.0f, 0.0f, 0.0f);
glCallList(lists[LIST_CURVE]);

glPopMatrix();
glPushMatrix();

/* Bömmels */
glTranslatef(0.0f,
             1.06f - myCos(auslenkung) * (0.1f + mySin(55.0f) * 0.55f /
             myCos(55.0f)),
             0.4f + mySin(auslenkung) * (0.1f + mySin(55.0f) * 0.55f /
             myCos(55.0f)));
glutSolidSphere(0.1f, 30, 30);

glPopMatrix();
glPushMatrix();

glTranslatef(0.0f, .96f - mySin(55.0f) / myCos(55.0f) * 0.55f, 0.2f);
glutSolidSphere(0.1f, 30, 30);
glTranslatef(0.0f, 0.0f, -0.2f);
glutSolidSphere(0.1f, 30, 30);
```

```
  glTranslatef(0.0f, 0.0f, -0.2f);
  glutSolidSphere(0.1f, 30, 30);
//  glTranslatef(0.0f, 0.0f, -0.2f);

  glPopMatrix();
  glPushMatrix();
  glTranslatef(0.0f,
               1.06f - myCos(360.0f - auslenkung2) *
               (0.1f + mySin(55.0f) / myCos(55.0f) * 0.55f),
               -0.4f + mySin(360.0f - auslenkung2) *
               (0.1f + mySin(55.0f) / myCos(55.0f) * 0.55f));
  glutSolidSphere(0.1f, 30, 30);

  glPopMatrix();
  glPushMatrix();

  /* Seile */
  glCallList(lists[LIST_ROPE]);
  glTranslatef(0.0f, 0.0f, -0.2f);
  glCallList(lists[LIST_ROPE]);
  glTranslatef(0.0f, 0.0f, -0.2f);
  //glCallList(lists[LIST_ROPE]);
  manager_ausgelenkter_faden(360.0f - auslenkung2, 1);
  glTranslatef(0.0f, 0.0f, 0.6f);
  glCallList(lists[LIST_ROPE]);
  glTranslatef(0.0f, 0.0f, 0.2f);
  //glCallList(lists[LIST_ROPE]);
  manager_ausgelenkter_faden(auslenkung, 1);

  glPopMatrix();

  glRotatef(180.0f, 0.0f, 1.0f, 0.0f);
  glCallList(lists[LIST_ROPE]);
  glTranslatef(0.0f, 0.0f, -0.2f);
  glCallList(lists[LIST_ROPE]);
  glTranslatef(0.0f, 0.0f, -0.2f);
  //glCallList(lists[LIST_ROPE]);
  manager_ausgelenkter_faden(auslenkung, -1);
  glTranslatef(0.0f, 0.0f, 0.6f);
  glCallList(lists[LIST_ROPE]);
  glTranslatef(0.0f, 0.0f, 0.2f);
  //glCallList(lists[LIST_ROPE]);
  manager_ausgelenkter_faden(360.0f - auslenkung2, -1);
}

void manager_reshape (int w, int h) {
  GLfloat aspectRatio;

  glViewport(0, 0, w, h);
```

```
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  aspectRatio = ((w == 0.0f)?(1.0f):(w)) / ((h == 0.0f)?(1.0f):(h));
  if (w < h)
    glFrustum(-myWidth, myWidth, -myHeight / aspectRatio,
              myHeight / aspectRatio, 2.0f, 20.0f);
  else
    glFrustum(-myWidth * aspectRatio, myWidth * aspectRatio,
              -myHeight, myHeight, 2.0f, 20.0f);
}

void manager_init () {
  const GLfloat ambientLight[4] = { .2f, .2f, .2f, 1.0f },
    specularLight[4] = { 1.0f, 1.0f, 1.0f, 1.0f },
    specRef[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
  int i;

  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glClearDepth(1.0f);

  glEnable(GL_DEPTH_TEST);
  glEnable(GL_NORMALIZE);

  glEnable(GL_CULL_FACE);
  glCullFace(GL_BACK);
  glFrontFace(GL_CCW);

  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glEnable(GL_BLEND);

  glEnable(GL_MULTISAMPLE);
  glEnable(GL_POINT_SMOOTH);
  glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
  glEnable(GL_LINE_SMOOTH);
  glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
#if 0
  glEnable(GL_POLYGON_SMOOTH);
  glHint(GL_POLYGON_SMOOTH_HINT, GL_NICEST);
#endif
  glShadeModel(GL_SMOOTH);

  glEnable(GL_LIGHTING);
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLight);

  glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
  glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);
  glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 60.0f);
  glEnable(GL_LIGHT0);
```

```
  glEnable(GL_COLOR_MATERIAL);
  glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

  glMaterialfv(GL_FRONT, GL_SPECULAR, specRef);
  glMateriali(GL_FRONT, GL_SHININESS, 128);

  /* Displaylisten anlegen */
  lists[0] = glGenLists(LIST_MAX);
  for (i = 1; i < LIST_MAX; ++i) lists[i] = lists[i - 1] + 1;

  /* dünne Seile */
  glNewList(lists[LIST_ROPE], GL_COMPILE);
    glPushMatrix();
    glColor4f(1.0f, 1.0f, 1.0f, 0.55f);
    glTranslatef(0.275f, 1.06f - mySin(55.0f) / myCos(55.0f) * 0.275f,
                 0.0f);
    glRotatef(55.0f, 0.0f, 0.0f, 1.0f);
    myTube(0.55f / myCos(55.0f), 0.002f, 18);
    glPopMatrix();
  glEndList();

  /* Silberstangen */
  glNewList(lists[LIST_BAR], GL_COMPILE);
    glColor3f(.8f, .7f, .7f);
    myTube(1.0f, 0.02f, 36);
  glEndList();

  /* Kurvenstück */
  glNewList(lists[LIST_CURVE], GL_COMPILE);
    myMuffenStuck(0.04f, 0.02f, 36);
  glEndList();

  printf("%s\n%s\n%s\n",
   glGetString(GL_VENDOR), glGetString(GL_RENDERER),
   glGetString(GL_VERSION));
}
```

F. Makefile

```
.PHONY: all install clean

GLIB=$(shell pkg-config --cflags glib-2.0)

ifeq (Darwin,$(shell uname -s))
  GLUT=-framework GLUT
  LDFLAGS=-dynamiclib -L/usr/X11R6/lib -lX11 -lGL
  CFLAGS=-DMACOS -W -Wall -O3 -fPIC $(GLIB) -I/usr/X11R6/include \
         -I/opt/local/include/xmms
else
  GLUT=-lglut -lGLU
  LDFLAGS=-shared -L/usr/X11R6/lib -lX11 -lGL
  CFLAGS=-W -Wall -O3 -fPIC $(GLIB) -I/usr/X11R6/include \
         -I/usr/include/xmms
endif


DEST=$(HOME)/.xmms/Plugins/Visualization

all:    klick_klack.so visual_pcm.so visual_freq.so

klick_klack.so: Makefile klick_klack.o manager.o
        $(CC) $(CFLAGS) $(LDFLAGS) $(GLUT) -o $@ klick_klack.o manager.o

klick_klack.o:  Makefile klick_klack.c
        $(CC) $(CFLAGS) -c -o $@ klick_klack.c

manager.o:      Makefile manager.c
        $(CC) $(CFLAGS) -c -o $@ manager.c

visual_pcm.so:  Makefile visual_pcm.c
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ visual_pcm.c

visual_freq.so: Makefile visual_freq.c
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ visual_freq.c

install:        all
        install -m 0755 -d $(DEST)
        install -m 0755 *.so $(DEST)

clean: ; rm -f *.so *.o
```