Technische Universität Dresden
Fakultät Informatik
Institut für Betriebssysteme, Datenbanken und Rechnernetze
Lehrstuhl Betriebssysteme

GROSSER BELEG

Titel:

Portierung der libSDL auf DROPS/DOpE

Bearbeiter:
Thomas Friebel <tf13@os.inf.tu-dresden.de>

Betreuer:
Dipl. Inf. Jork Löser

verantwortlicher Hochschullehrer:
Prof. Dr. rer. nat. Hermann Härtig

Ziel der Arbeit war es, durch Portierung der libSDL eine Grundlage zur Erstellung und Portierung von Multimediaanwendungen auf/nach DROPS zu schaffen. In diesem Zusammenhang wurde mit DSound auf Basis von dde_linux ein Server zur Audiowiedergabe entwickelt. Am Anfang der Arbeit wird die libSDL und ihr Aufbau vorgestellt, um danach auf die Details der Portierung einzugehen. Dann werden die portierten Beispiel-Anwendungen gezeigt, bevor zum Schluß die Performance einiger Teile betrachtet wird.

Inhaltsverzeichnis

1 Einleitung **1**

2 Vorstellung der libSDL **2**
2.1 Aufbau 2
2.2 SDL-basierte Bibliotheken 5

3 Portierung **7**
3.1 Video 7
3.2 Audio 11
3.3 Threads und Sperren 15
3.4 Dateizugriff 17
3.5 Ausbaumöglichkeiten 17

4 Beispielanwendungen **18**
4.1 Waves 18
4.2 Quake 1 18
4.3 Barrage 19

5 Leistungsbewertung **21**
5.1 Testumgebung 21
5.2 Quake 22
5.3 Video 25
5.4 Audio 26

6 Zusammenfassung/Ausblick **28**

Kapitel 1

Einleitung

SDL steht für Simple DirectMedia Layer, was die libSDL auch darstellt: eine einfach gehaltene und plattformübergreifende Abstraktionsebene für den möglichst direkten Zugriff auf die Multimediaschnittstellen eines PC. Sie wurde primär als Basis für plattformunabhängige Spieleentwicklung geschaffen und so werden zum gegenwärtigen Zeitpunkt (Winter 2003/04) auf der SDL-Website [1] immerhin knapp 500 darauf basierende Spiele und Anwendungen genannt. Dabei enthält die Liste das ganze Spektrum von nicht interaktiven Demos, über mehrere Varianten von Klassikern wie Tetris oder Breakout, bis hin zu großen professionellen Ports, wie der Linux-Version von „Civilization – Call To Power“. Das zeigt, daß die libSDL vielseitig einsetzbar und auch für große und kommerzielle Projekte tauglich ist. Sie steht unter der GNU GPL (General Public License, [2]) frei zur Verfügung.

Ziel der Belegarbeit war es, durch die Portierung der libSDL den Bestand an für DROPS verfügbarer Software zu vergrößern. Die Wahl fiel auf die SDL, da diese bereits auf viele verschiedene Plattformen portiert wurde und als ausgereift und flexibel gilt. Der Erfolg der Portierung sollte anhand der SDL-Version von Quake 1 demonstriert werden, weil es im Quellcode frei verfügbar ist, als Klassiker gilt, sowie als „Blickfang“ wirken könnte.

Quake wurde im Jahr 1996 veröffentlicht und stammt vom Spiele-Hersteller id-software, der unter anderem durch Doom bekannt geworden ist. Schon das Original-Quake – und noch mehr dessen Nachfolger – erfreute sich großer Beliebtheit bei den Online-Spielern. 1999, als die Quake-Engine veraltet war und damit niemand mehr wirtschaftliches Interesse daran hatte, wurde sie unter der GNU GPL der Öffentlichkeit zur Verfügung gestellt. Die Spieldaten dagegen sind weiterhin nicht frei. Doch es besteht noch immer die Möglichkeit die Shareware-Version von Quake 1 auf der id-software Website [3] herunterzuladen und deren Daten zu verwenden.

Kapitel 2

Vorstellung der libSDL

2.1 Aufbau

Die SDL ist in mehrere meist eigenständige Subsysteme unterteilt: Video, Audio, CD-ROM, Joystick, Threads, Timer, File-IO und Events. Die unabhängigen Subsysteme können einzeln initialisiert und benutzt werden.
Ein Subsystem besteht aus einem oder mehreren verschiedenen unteren, architekturabhängigen Adaptern oder Treibern und einem oberen, darauf aufbauenden, architekturunabhängigen Teil. Die architekturabhängigen Teile bieten die Funktionen der Hard- bzw. Software durch eine jeweils einheitliche Schnittstelle an. Der architekturunabhängige Teil stellt einen Rahmen für den komfortablen Zugriff zur Verfügung. Er kann meist nicht direkt unterstützte Funktionen kompensieren, zum Beispiel den Framebuffer zu einer unterstützten Farbtiefe konvertieren. Im Normalfall geschieht das für den Programmierer vollkommen transparent. Er kann eine solche Kompensation aber auch verbieten, um dies als Fehler selbst zu behandeln. Falls keine Kompensation nötig ist, sind die Wege innerhalb der SDL – vom SDL-Funktionsaufruf bis zur darunterliegenden Hard-/Softwareschicht – sehr kurz, was bedeutet, daß die SDL im Normalfall sehr direkt und damit schnell ist.

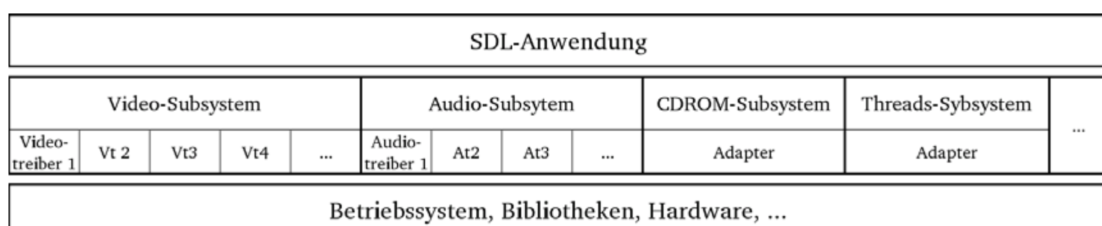

Abbildung 2.1: Aufbau der libSDL

2.1.1 Treiber-Modell

Die unteren Teile des Audio- und des Video-Subsystems sind als Treiber ausgeführt. Davon stehen meist mehrere gleichzeitig zur Verfügung: unter Linux für die Graphikausgabe zum Beispiel wären die Treiber für die aalib [4], DirectFB [5], die SVGAlib [6] und X11 [7] möglich. Bei der Initialisierung des Subsystems wird die Funktion `available()` des ersten vorhandenen Treibers aufgerufen. Bei positivem Ergebnis wird dieser Treiber initialisiert und nachfolgend immer benutzt. Andernfalls wird dies beim jeweils nächsten Treiber versucht, bis ein verfügbarer Treiber gefunden wird. Falls keiner gefunden werden kann, schlägt die Initialisierung des Subsystems fehl.

2.1.2 Video-Subsystem

Der Ausgabebereich (Framebuffer) des Video-Treibers steht als „Surface" zur Verfügung. Ein Surface enthält neben dem eigentlichen Pixel-Array noch weitere Informationen wie die Bildbreite und -höhe und ein „PixelFormat", welches die Farbtiefe und die Bit-Anordnung der Farbinformationen beschreibt. Die Darstellung des Framebuffers erfolgt im Fenster- oder auch im Vollbildmodus. Es können auch weitere Surfaces erstellt werden, die dann zum Beispiel als Speicher für geladene Bilddateien dienen können. Die SDL selbst unterstützt nur das Laden und Speichern im bmp-Format, aber mit der SDL_image Bibliothek (Kapitel 2.2.1) ist der Zugriff auf jpg-, png-, gif- und weitere Dateiformate möglich.
Die Surfaces können mit Hilfe einer Blitting-Funktion ineinander kopiert werden, wobei sie aber im gleichen PixelFormat vorliegen müssen. Zur Vorbereitung der Surfaces stehen neben einer Funktion zum Konvertieren des PixelFormates noch Funktionen zum Setzen des Alpha-Wertes (Transparenz) und des Clipping-Bereiches (zu bearbeitender Bildausschnitt) zur Verfügung.
Desweiteren werden diverse Windowmanager-Funktionen – wie das Setzen des Fenstertitels, Minimieren des Fensters, Setzen des Mauszeigers etc. – angeboten.

Setzen des Videomodus

Der Framebuffer steht erst nach Setzen des Videomodus zur Verfügung. Wird die gewünschte Auflösung bei der vorgegebenen Farbtiefe vom Video-Treiber nicht angeboten, so wird die nächsthöhere Auflösung benutzt. Gibt es keine höhere, wird eine andere Farbtiefe gewählt: zunächst die höheren, danach die niedrigeren. Die Suche scheitert also nur, falls bei keiner Farbtiefe die gewünschte oder eine höhere Auflösung gefunden werden kann. Im Erfolgsfall steht der SDL-Anwendung ein Shadow-Surface zur Verfügung, der genau ihren Vorgaben entspricht. Beim Aufruf der Funktion zum Aktualisieren des Bildschirm-/Fensterinhaltes wird dann der Inhalt des Shadow-Surface in den Puffer des Graphiktreibers kopiert und dabei die PixelFormat-Konvertierung vorgenommen.
Diese Vorgehensweise ist für den Programmierer äußerst komfortabel: er kann sich darauf verlassen, sofern möglich, den gewünschten Videomodus vorzufinden und die SDL übernimmt die Anpassung an die momentanen Gegebenheiten. Die dafür enstehenden

Performance-Kosten werden später betrachtet (Kapitel 5.3.1).

SDL und OpenGL

Folgende Angaben finden sich in [8] über die Zusammenarbeit der SDL mit OpenGL:
Statt auf den Framebuffer zu zeichnen, kann man auch OpenGL-Funktionen benutzen, um damit eine 3D-Anzeige zu erzeugen. OpenGL und die SDL arbeiten sehr gut zusammen: Beim Setzen des Videomodus muß angegeben werden, daß man OpenGL benutzen will und danach kann man beliebige OpenGL-Kommandos ausführen. Auf den Framebuffer darf man danach allerdings nicht mehr direkt zugreifen, dafür stellt OpenGL aber ausgereifte Funktionen bereit. Den Rest des Video-Subsystems und die anderen Subsysteme kann man aber weiterhin uneingeschränkt benutzen, was die SDL zu einer sehr guten Basis für 3D-Spiele macht.

2.1.3 Audio-Subsystem

Das Audio-Subsystem funktioniert nach dem buffer+callback-Prinzip: Die Anwendung übergibt dem Subystem eine Audio-Rendering-Funktion, welche den Audio-Puffer füllt und das Subsystem startet einen Thread, der abwechselnd diese und die Funktion des Treibers zum Abspielen des Puffers aufruft. Die Puffergröße bestimmt die Frequenz dieser Aufrufe und damit die Latenz der Audio-Ausgabe. Darauf wird in Kapitel 3.2.4 genauer eingegangen.
Ähnlich der Vorgehensweise des Video-Subsystems wird der Audio-Puffer konvertiert, falls die Anwendung ein Format verlangt, welches der Treiber nicht unterstützt. So werden gegebenenfalls das Sample-Format (Anzahl der Kanäle, low-/big-endian, signed/unsigned) und die Sample-Rate umgewandelt.
Ebenfalls analog zum Umgang mit Surfaces im Video-Subsystems werden Funktionen zum Laden von wav-Dateien im PCM-Format, zum Konvertieren des Formates eines Audiopuffers und zum Mischen zweier Puffer angeboten.

2.1.4 Thread-Subsystem

Das Thread-Subsystem bietet Funktionen zum Erstellen neuer Threads, zum Warten auf Beendigung und zum erzwungenen Beenden von Threads an. Desweiteren stehen zur Synchronisation Semaphoren, Mutexe und Condition-Variablen (Wartelisten) zur Verfügung. Thread-local storage – Funktionen zum Zugriff auf Thread-spezifischen Speicher – wird nicht angeboten.

2.1.5 Events-Subsystem

Das Events-Subsystem ist die zentrale Ereignis-Verwaltung. Es dient anderen Subsystemen, die bei bestimmten Ereignissen diese zur Event-Liste hinzufügen. Die verschiedenen Ereignistypen sind: Tastatur-, Maus-, Joystick-, Fenstermanager- und benutzerdefinierte

Ereignisse. Jeder Typ kann maskiert werden, sodaß er nicht mehr in die Event-Schlange eingereiht, sondern einfach „vergessen“ wird. Die akzeptierten Ereignisse können dann vom Programmierer per Polling abgeholt werden, oder er erzeugt einen eigenen Thread für die Ereignisbehandlung und benutzt die `SDL_WaitEvent()`-Funktion.

2.1.6 weitere Subsysteme

Joystick

Das Joystick-Subsystem stellt Funktionen zum Abfragen der verschiedenen Achsen und der Knopf-Stati eines oder mehrerer Joysticks zur Verfügung. Falls gewünscht, werden auch bei jeder Veränderung Joystick-Ereignisse generiert und in die Event-Liste eingetragen.

File

Das File-Subsystem dient der Abstraktion von Zugriffen auf Datenquellen. Es werden die Standardfunktionen für Öffnen, Position setzen, Lesen, Schreiben und Schließen angeboten. Unterstützt wird im Moment der Dateizugriff per C-stdio und der Zugriff auf Speicherbereiche.

Timer

Mithilfe des Timer-Subsystems kann man periodische Funktionsaufrufe generieren und verwalten. Dies geschieht durch Anmelden der Callback-Funktionen unter Angabe der Periode. Dafür wird entweder das Thread-Subsystem benutzt oder – falls vorhanden – eine systemspezifische Timer-Implementation.

CD-ROM

Das CD-ROM-Subsystem stellt Funktionen zur Wiedergabe von Audio-CDs zur Verfügung: zum Beispiel `SDL_CDPlay()`, `SDL_CDPause()` und `SDL_CDEject()`.

2.2 SDL-basierte Bibliotheken

Im Moment zählt die Liste der SDL-basierten Bibliotheken auf der SDL-Website [1] knapp 100 Einträge. Sehr häufig werden davon die Erweiterungen der libSDL-Autoren benutzt. Drei wichtige, die oft bei der Implementierung von Spielen genutzt werden, werden hier kurz vorgestellt.

2.2.1 SDL_image

Die SDL_image-Bibliothek erweitert das Video-Subsystem um Funktionen zum Laden verschiedener Bilddateiformate. Sie unterstützt unter anderem jpg-, png-, tif-, gif-, pcx- und

tga-Dateien, wobei die Benutzung der ersten drei jeweils eine zusätzliche Bibliothek erfordert. Zum Dateizugriff wird das File-IO-Subsystem benutzt, sodaß nicht nur aus Dateien, sondern auch aus dem Speicher gelesen werden kann. Das Schreiben der Dateiformate wird nicht unterstützt.

2.2.2 SDL_mixer

Die SDL_mixer-Bibliothek benutzt das Audio-Subsystem um eine komfortablere Schnittstelle zur Sample- und Musikausgabe zur Verfügung zu stellen. Das Audio-Subsystem muß initialisiert sein, darf aber nicht gleichzeitig mit der SDL_mixer-Bibliothek benutzt werden. Zuerst muß sie unter Angabe des Audio-Ausgabeformates und der maximal gewünschten Zahl von Audiokanälen initialisiert werden. Dann kann man die Funktionen zum Laden von wav-, ogg- und voc-Dateien in „Mix Chunks" – einer Speicherstruktur für Audiosample – aufrufen, wobei die Daten automatisch in das Ausgabeformat konvertiert werden.
Die Mix Chunks können in der Lautstärke verändert und auf einem der Audiokanäle ausgegeben werden. Jedem Kanal einzeln und der Summe (dem gemischten Ausgabekanal) können mehrere Effekte zugeordnet werden. Dafür stehen unter anderem eine Balance- und eine Distanz-Funktion zur Verfügung. Weitere benutzerdefinierte Funktionen können auch gesetzt werden.
Parallel dazu steht noch ein Musikkanal zur Verfügung, mit dem wav-, mod-, midi-, ogg- und mp3-Dateien abgespielt werden können. Auch hier werden für die nicht-wav-Formate zusätzliche Bibliotheken benötigt.

2.2.3 SDL_net

Die SDL_net-Bibliothek bietet Funktionen zum Netzwerkzugriff. Sie unterstützt TCP-Verbindungen, das Senden und Empfangen von UDP-Paketen und DNS-Namensauflösung.
Die Adressen von UDP-Kommunikationspartnern können in Klassen gruppiert werden, sodaß ein Paket automatisch an eine Reihe von Adressaten gesandt werden kann. Desweiteren werden Pakete eines Absenders einer höheren Klasse gegenüber denen aus einer niedrigeren Klasse bevorzugt verarbeitet.

Kapitel 3

Portierung

Als Ausgangspunkt wurde die 2003 aktuelle libSDL-Version 1.2.5 benutzt. Eine Aktualisierung des Ports auf eine neuere Version soll mit möglichst geringem Aufwand möglich sein. Dazu werden die angepaßten oder neu hinzugefügten Dateien in einem eigenen Verzeichnis abgelegt. Am Anfang des Build-Vorganges werden dann symbolische Verknüpfungen zu den nicht veränderten Dateien erstellt.

3.1 Video

Die Funktionalität des Video-Subsystems wurde auf das Desktop Operating Environment (DOpE, [9]) – das Fenstersystem des DROPS-Projektes – abgebildet. Das beinhaltet neben der Graphikausgabe die Fenstermanager-Funktionen und die Ereignisbehandlung von Tastatur und Maus. Wie in Kapitel 2.1.1 beschrieben ist der systemspezifische Teil des Video-Subsystems der SDL als Treiber ausgeführt. Es wurde die Treiber-Schnittstelle (`DOPE_bootstrap`, Abbildung 3.1) implementiert und der Liste der Videotreiber hinzugefügt. Die ersten beiden Bestandteile von `DOPE_bootstrap` geben einen Namen und eine

```
VideoBootStrap DOPE_bootstrap = {
    "dope",
    "SDL dope video driver",
    DOPE_Available,
    DOPE_CreateDevice
};
```

Abbildung 3.1: Die Struktur DOPE_bootstrap

kurze Beschreibung an, die anderen beiden enthalten Funktionen zum anfänglichen Zugriff auf den Treiber:

- `int DOPE_Available();`
  wird beim Initialisieren des Videosubsystems aufgerufen, um zu prüfen ob der

Treiber zur Verfügung steht. Dazu wird beim Namensdienst mit Hilfe der Funktion `names_waitfor_name()` überprüft ob der DOpE-Server dort angemeldet ist und gegebenenfalls der Wert 1 zurückgegeben, was bedeutet, daß der Treiber verfügbar ist.

- `SDL_VideoDevice *DOPE_CreateDevice(int devindex);`
  erzeugt eine neue Struktur vom Typ `SDL_VideoDevice` (Abbildung 3.2) und setzt die Funktionszeiger darin auf die DOpE-spezifischen Funktionen. Der Parameter `devindex` wird von der SDL im Moment noch nicht benutzt.

Die Struktur `SDL_VideoDevice` (Abbildung 3.2) umfaßt 41 Zeiger auf Funktionen und 18 weitere, von der SDL benutzte Variablen. Nur einige der Funktionen müssen für einen funktionsfähigen Graphiktreiber implementiert werden, der Großteil ist optional. Zeiger für nicht implementierte Funktionen werden mit `NULL` initialisiert – die SDL überprüft vor Benutzung der optionalen Funktionen den Wert der Zeiger. Die implementierten Funktionen werden in den folgenden Unterkapiteln beschrieben.

```
#define _THIS SDL_VideoDevice *this
struct SDL_VideoDevice {
    void (*free)(_THIS);
    int  (*VideoInit)(_THIS, SDL_PixelFormat *vformat);
    void (*VideoQuit)(_THIS);
    SDL_Rect **(*ListModes)(_THIS, SDL_PixelFormat *format,
        Uint32 flags);
    [...]
}
```

Abbildung 3.2: Die Struktur SDL_VideoDevice

3.1.1 Initialisierung und Deinitialisierung

Die ersten drei der oben angegebenen Zeiger werden auf folgende Funktionen gesetzt:

- `void DOPE_DeleteDevice(SDL_VideoDevice *device);`
  gibt den Speicher der Device-Struktur wieder frei.

- `int DOPE_VideoInit(_THIS, SDL_PixelFormat *vformat);`
  initialisiert die DOpE-Bibliothek mittels dope_init() und setzt `vformat` auf das beste unterstützte Pixelformat – hier auf 16 bpp, da das DOpE VScreen-Widget (siehe unten) im Moment nur dieses und YUV420 (ein Format mit einer Farbinformation pro 2x2-Pixel-Gruppe, Helligkeit für jedes Pixel) anbietet.

- `void DOPE_VideoQuit(_THIS);`
  beendet den Treiber, indem es eventuell geöffnete DOpE-Fenster schließt und danach `dope_deinit()` aufruft, um die DOpE-Bibliothek zu deinitialisieren.

3.1.2 Graphikausgabe

Die Graphikausgabe erfolgt über ein DOpE VScreen-Widget – ein Fensterelement, welches der Anwendung einen Pixel-Speicher zur Verfügung stellt und diesen bei Aufruf der `refresh()`-Funktion in seinen Fensterbereich zeichnet. Der Speicher wird bei der Anwendung eingeblendet, sodaß diese und DOpE gemeinsam darauf zugreifen können.

Die Bereitstellung der Graphikausgabe wird mit folgenden drei Funktionen erreicht:

- `SDL_Rect **DOPE_ListModes(_THIS, SDL_PixelFormat *format, Uint32 flags);`
  wird beim Setzen des Videomodus aufgerufen um herauszufinden, ob die gewünschte Auflösung bei gegebenem PixelFormat unterstützt wird – und falls nicht, welche die nächstbeste unterstützte Auflösung ist. Für ein PixelFormat mit 16 bit Farbtiefe wird der Wert -1 zurückgegeben, was bedeutet, daß jede beliebige Auflösung unterstützt wird. Andere PixelFormate werden nicht unterstützt, weshalb in dem Fall der Wert `NULL` zurückgegeben wird. Die SDL wird dann gegebenenfalls durch die in Kapitel 2.1.2 beschriebene Vorgehensweise einen 16 bpp Modus benutzen und das Format des Pixel-Speichers konvertieren.

- `SDL_Surface *DOPE_SetVideoMode(_THIS, SDL_Surface *current, int width, int height, int bpp, Uint32 flags);`
  dient dem Setzen eines neuen Video-Modus und stellt entsprechend ein neues zugehöriges Surface zur Verfügung. Falls bereits ein Video-Modus gesetzt war, wird zuerst das DOpE-Fenster geschlossen und der dazu reservierte Speicher freigegeben. Dann wird ein neues Fenster entsprechend der gewünschten Auflösung geöffnet und ein VScreen-Widget eingefügt. Der Pixelspeicher des VScreen wird als der des Surface gesetzt. Danach wird für die verschiedenen Tastatur- und Maus-Ereignisse eine Behandlungsfunktion (Kapitel 3.1.4) registriert.

- `void DOPE_UpdateRects(_THIS, int numrects, SDL_Rect *rects);`
  gibt für jeden angegebenen Bereich der Graphikausgabe DOpE den Befehl, diesen Ausschnitt des VScreen-Widgets neu zu zeichnen.

3.1.3 Fenstermanager-Funktionen

- `SDL_GrabMode DOPE_GrabInput(_THIS, SDL_GrabMode mode);`
  setzt oder löscht das `grabmouse`-Attribut des VScreen-Widgets. Ist es gesetzt, erhält der VScreen mit dem nächsten Maus-Click des Benutzers die Kontrolle über die Maus, sodaß der Mauszeiger den Bereich des Widgets nicht mehr verlassen kann.

- `void DOPE_WarpWMCursor(_THIS, Uint16 x, Uint16 y);`
  setzt den Mauszeiger mit Hilfe einer Funktion des VScreen-Widgets an die angegebene Position. Dazu muß dieses das `grabmouse`-Attribut und die Kontrolle über die Maus haben.

- `void DOPE_SetCaption(_THIS, const char *wintitle, const char *apptitle);`
  setzt den Fenstertitel und den Anwendungsnamen, der in der DOpE-Statusleiste dargestellt wird.

3.1.4 Ereignisbehandlung

Am VScreen-Widget wird für die Maus- und Tastaturereignisse eine Behandlungsfunktion registriert, die diese auf die entsprechenden SDL-Ereignisse abbildet. Folgende DOpE-Ereignisse werden behandelt:

- `motion`
  Wenn der Mauszeiger innerhalb des VScreen bewegt wird, werden `motion`-Ereignisse erzeugt. Ein SDL-Event mit den neuen Koordinaten wird mittels `SDL_PrivateMouseMotion()` an die Warteschlange angefügt.

- `press`, `release`
  `press`- und `release`-Ereignisse werden erzeugt, wenn auf Tastatur oder Maus eine Taste gedrückt bzw. wieder losgelassen wird. Handelt es sich um eine Maustaste, wird der neue Zustand mittels `SDL_PrivateMouseButton()` an die Ereignisverwaltung weitergegeben. Bei Tastaturereignissen wird der DOpE-Tastencode zuerst mit Hilfe einer Tabelle und der Funktion `dope_get_ascii()` in den SDL-Code umgewandelt. Falls die Taste erkannt wurde, wird dann mittels `SDL_PrivateKeyboard()` das SDL-Ereignis erzeugt.

Für die Ereignisbehandlung wurden folgende zwei Funktionen implementiert:

- `void DOPE_InitOSKeymap(_THIS);`
  setzt die Werte in einem Array, mit dessen Hilfe die DOpE- in SDL-Tastencodes umgewandelt werden.

- `void DOPE_PumpEvents(_THIS);`
  wird vom Events-Subsystem aufgerufen und dient dazu gegebenenfalls bereitstehende DOpE-Ereignisse abzufragen und die entsprechenden SDL-Ereignisse zu generieren. Diese Funktion hat hier keine Aufgabe, da die Ereignisse von DOpE nicht per Polling abgeholt werden können, sondern die Behandlungsroutine bei Ereignissen asynchron aufgerufen wird.

3.2 Audio

Die Implementierung des Audio-Subsystems basiert auf dem DROPS-Paket DSound. DSound wurde mit im Rahmen dieses Großen Belegs erarbeitet. Es benutzt mit Hilfe von dde_linux Soundkartentreiber von Linux.

3.2.1 dde_linux

Das Linux Device Driver Environment (dde_linux) stellt Linux Treibern eine dem Linux-Kern entsprechende Umgebung zur Verfügung. Wird zu einer Anwendung ein Linux-Soundtreiber und dde_linux dazugelinkt, kann diese über die Funktionen `l4dde_snd_open_dsp()`, ...`open_mixer()`, ...`close()`, ...`read()`, ...`write()` und `l4dde_snd_ioctl()` auf dem unter Linux üblichen Weg (dort wird `open()`, `close()`, etc. benutzt) auf den Soundtreiber zugreifen. Da es nicht komfortabel ist von jeder SDL-Anwendung eine Variante für jeden Soundtreiber zu generieren, entstand DSound – das DROPS Sound System.

3.2.2 DSound

DSound besteht aus einem Server – dem DSound Daemon (DSD) – und bietet eine Bibliothek zum komfortableren Zugriff darauf (DSlib). Der DSD benutzt dde_linux, und da der Soundtreiber nun hier fest hinzugelinkt werden muß, wird eine Variante je Treiber erstellt. Die Anwendung erstellt einen Audiopuffer und gibt ihn für den DSD frei, der ihn dann mit nur-lese-Zugriff in seinen Speicher einblendet. Danach füllt die Anwendung den Puffer immer wieder neu mit den auszugebenden Daten und ruft die Funktion zur Ausgabe auf. Die Funktionen der DSlib im Detail sind:

- `int dslib_init(int timeout);`
  wird zur Initialisierung der DSlib aufgerufen. Dabei wird mittels `names_waitfor_name()` darauf gewartet, daß sich der DSD-Thread beim Namensdienst anmeldet. Der Parameter `timeout` gibt an wie lang gewartet werden soll.

- `int dslib_open_dsp(long bufsize, void **buf);`
  bereitet die Soundwiedergabe vor: Zuerst wird am DSD `dsound_open_dsp()` – die Funktion zum Öffnen des DSP-Device aufgerufen. Danach wird ein Puffer der in `bufsize` angegebenen Größe erstellt und für den DSD freigegeben. Anschließend wird am DSD `dsound_set_buf()` aufgerufen, sodaß dieser den Puffer in seinen Speicher einblendet. `**buf` dient zur Rückgabe der Pufferadresse an die Anwendung.

- `int dslib_close_dsp(void);`
  schließt mittels `dsound_close_dsp()` das Device, was den DSD auch veranlaßt den gemeinsamen Speicher wieder auszublenden und gibt ihn im Anschluß wieder frei.

- `int dslib_set_vol(long volume);`
  ruft am DSD `dsound_set_vol()` auf, um die Wiedergabelautstärke einzustellen. Dieser öffnet dazu das Mixer-Device, setzt die Laustärke mittels `ioctl()` und schließt es wieder.

- `int dslib_set_frag(long fragnum, long fragsize);`
  dient dem Setzen der Anzahl und Größe der Fragmente: Da die wiederzugebenden Daten an den Treiber mittels `write()` übergeben werden, an die Soundkarte dann aber meist über Programmierung des DMA-Controllers, müssen diese zu Blocks/Fragmenten zusammengefaßt werden. Die Anzahl und Größe dieser Fragmente setzt der DSD mittels `ioctl()`. Sie bestimmen maßgeblich die Latenz der Audiowiedergabe (Kapitel 3.2.4).

- `int dslib_get_fmts(long *formats);`
  erfragt mittels `ioctl()` die von der Soundkarte unterstützten Sample-Formate und gibt diese in `*formats` zurück.

- `int dslib_set_fmt(long format);`
  `int dslib_set_chans(long channels)`
  `int dslib_set_freq(long frequency);`
  dienen zum Setzen des Wiedergabe- bzw. des Pufferformats. Mit `dslib_set_fmt()` setzt man das Sample-Format (vorzeichenbehaftet/-frei, 8/16 bit, low-/big-endian), mit `dslib_set_chans()` die Anzahl der Kanäle und mit `dslib_set_freq()` die Samplefrequenz.

- `int dslib_play(long *len);`
  veranlaßt den DSD den Puffer per `write()` auszugeben. Dabei werden maximal `*len` Byte geschrieben. Die Anzahl der erfolgreich geschriebenen Bytes wird ebenfalls über `*len` wieder zurückgegeben.

3.2.3 SDL-Audiotreiber

Wie im Video-Subsystem ist auch hier der systemspezifische Teil als Treiber ausgeführt. Die Bootstrap-Struktur ist mit der des Video-Treibers vergleichbar. Sie enthält je einen Zeiger auf die `Available()`- und die `CreateDevice()`-Funktion:

- `int DROPSAUD_Available(void);`
  initialisiert die DSlib mittels `dslib_init()`. Falls dies gelingt wird durch den Rückgabewert 1 signalisiert, daß der Treiber verfügbar ist.

- `SDL_AudioDevice *DROPSAUD_CreateDevice(int devindex);`
  erzeugt eine neue Struktur vom Typ `SDL_AudioDevice` (siehe unten) und setzt die Funktionszeiger darin auf die DSlib-spezifischen Funktionen. Auch hier wird der Parameter `devindex` von der SDL nicht benutzt.

```
#define _THIS SDL_AudioDevice *_this
struct SDL_AudioDevice {
    void (*free)(_THIS);
    int  (*OpenAudio)(_THIS, SDL_AudioSpec *spec);
    Uint8 *(*GetAudioBuf)(_THIS);
    void (*PlayAudio)(_THIS);
    void (*WaitAudio)(_THIS);
    [...]
}
```

Die Struktur `SDL_AudioDevice` umfaßt 10 Zeiger auf Funktionen und 12 weitere, von der SDL benutzte Variablen. Folgende der Funktionen mußten implementiert werden:

- `void DROPSAUD_DeleteDevice(SDL_AudioDevice *device);`
  gibt den Speicher der Struktur `*device` wieder frei. Der Funktionszeiger `free` in `SDL_VideoDevice` referenziert diese Funktion.

- `int DROPSAUD_OpenAudio(_THIS, SDL_AudioSpec *spec);`
  ruft `dslib_open_dsp()` auf, sucht mit Hilfe von `dslib_get_fmts()` das zu dem in `*spec` angegebenen naheste unterstützte Sampleformat und speichert es wiederum in `*spec`. Danach werden das Format, die Kanalanzahl, die Samplefrequenz und die Lautstärke durch die entsprechenden `dslib_set_`...-Funktionen gesetzt. Mittels `dslib_set_frag()` wird die Fragmentanzahl auf zwei und die -größe auf die in `*spec` angegebene Blockgröße eingestellt (Kapitel 3.2.4).

- `void DROPSAUD_CloseAudio(_THIS);`
  ruft `dslib_close_dsp()` auf, womit das DSP-Device geschlossen und der mit dem DSD gemeinsame Speicher freigegeben wird.

- `Uint8 *DROPSAUD_GetAudioBuf(_THIS);`
  gibt den mit dem DSD gemeinsamen Audio-Puffer zurück.

- `void DROPSAUD_PlayAudio(_THIS);`
  ruft `dslib_play()` auf, womit der Audio-Puffer vom DSD an den Linux-Soundtreiber übertragen und somit wiedergegeben wird.

- `void DROPSAUD_WaitAudio(_THIS);`
  wird von der SDL nach jedem `PlayAudio()` aufgerufen um zu warten, daß wieder ein kompletter Puffer frei ist und neu beschrieben werden kann. Da PlayAudio den Block immer komplett an die Soundkarte überträgt, kann er danach sofort wieder neu beschrieben werden. Es muß also nicht gewartet werden – WaitAudio kehrt immer ohne zu warten zurück.

3.2.4 Puffergröße/-anzahl und Latenz

Die Soundkarte stellt zur Wiedergabe mehrere Puffer als Ringpuffer zur Verfügung. Ständig wird einer der Puffer wiedergegeben, während die nachfolgenden auf die Wiedergabe warten und die vorherigen mit Daten gefüllt werden können. Im Normalfall arbeitet man mit zwei Puffern („Double Buffering"), die dann immer wechselseitig gefüllt bzw. abgespielt werden. Wenn man mit nur einem Puffer arbeitete, würde die Wiedergabe immer wieder kurz unterbrochen, um den Puffer neu zu beschreiben. Bei zwei Puffern ist eine unterbrechungsfreie Wiedergabe möglich. Mehr als zwei Puffer verringern zwar die Wahrscheinlichkeit einer Unterbrechung noch weiter, doch mit jedem zusätzlichen Puffer steigt die Wiedergabelatenz – die Zeit, die vom Schreiben des ersten Samples des Soundpuffers bis zu dessen Wiedergabe vergeht. Eine Vergrößerung der Puffer hat die gleichen Auswirkungen.
Auf Linux-Soundtreiber wird über die `write()`-Funktion zeichenweise zugegriffen. Die SDL erwartet aber, daß ein Puffer bereitgestellt wird, welcher also zusätzlich allokiert werden muß. Im schlimmsten Fall müssen also drei Puffer gespielt werden bis ein Sample wiedergegeben wird. Die Latenz ist also:

$$Latenz \leq 3 \times Puffergröße/(Samplegröße \times Samplefrequenz)$$

Abbildung 3.3 zeigt die Situation, in der die maximale Latenz auftritt: Der erste Puffer der Soundkarte muß noch komplett wiedergegeben werden, der zweite und der SDL-Puffer sind bereits gefüllt. Ein Sample welches jetzt wiedergegeben werden soll, muß also warten bis alle drei Puffer gespielt worden sind. Will man diese Zeitdauer noch verkürzen, müßte man die `WaitAudio()`-Funktion so implementieren, daß sie erst zurückkehrt wenn ein kompletter Puffer an die Soundkarte übertragen werden kann. Im Moment ist das aber mit dde_linux noch nicht möglich.

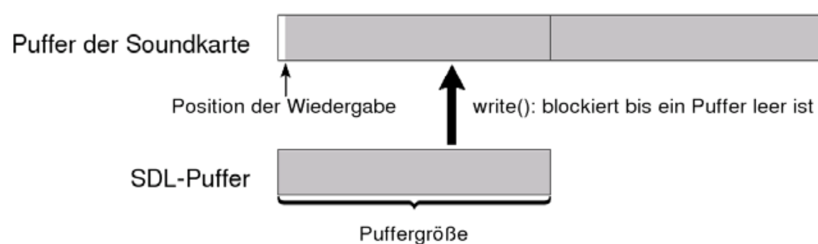

Abbildung 3.3: Soundpuffer bei maximaler Latenz

3.3 Threads und Sperren

3.3.1 Threads

Die von der libSDL für die Verwaltung von Threads benötigten Funktionen wurden auf die im DROPS L4env-Paket [10] „thread“ bereitstehenden abgebildet. Da es keine Funktion zum Warten auf das Ende eines Threads anbietet wurde dafür eine Semaphore benutzt:

- `int SDL_SYS_CreateThread(SDL_Thread *thread, void *args);`
  erstellt mit Hilfe der Funktion `l4thread_create()` einen neuen Thread, der `SDL_RunThread()` mit dem Parameter args aufruft, und speichert dessen id in `thread->threadid`. Desweiteren initialisiert die Funktion noch eine Thread-spezifische Semaphore mit 0, und registriert per `l4thread_on_exit()` eine Exit-Funktion, die bei Beendigung des Threads die Semaphore inkrementiert. Sie wird dazu benutzt um in `SDL_SYS_WaitThread` (siehe unten) auf das Thread-Ende zu warten.

- `void SDL_SYS_SetupThread(void);`
  wird von `SDL_RunThread()` aufgerufen, damit noch einige Arbeiten zur Einrichtung des Threads vorgenommen werden können. Diese Implementation führt hier nichts aus.

- `void SDL_SYS_WaitThread(SDL_Thread *thread);`
  dekrementiert die in `SDL_SYS_CreateThread()` erwähnte Semaphore des angegebenen Threads. Da diese erst bei Beendigung des Threads von 0 auf 1 inkrementiert wird, wartet die Funktion bis dahin. Danach wird der von der Semaphore und der Exit-Funktion belegte Speicher wieder freigegeben.

- `void SDL_SYS_KillThread(SDL_Thread *thread);`
  erzwingt das Beenden des angegebenen Threads mit Hilfe `l4thread_shutdown()`.

3.3.2 Semaphoren

Semaphoren werden nahezu eins zu eins mit Hilfe des L4env-Pakets „semaphore“ implementiert:

- `SDL_sem *SDL_CreateSemaphore(Uint32 initial_value);`
  `void SDL_DestroySemaphore(SDL_sem *sem);`
  `SDL_CreateSemaphore()` allokiert Speicher für eine neue Semaphore, initialisiert sie mit dem gegebenen Wert und gibt einen Zeiger darauf zurück. Mit `SDL_DestroySemaphore()` wird der Speicher wieder freigegeben.

- `int SDL_SemWaitTimeout(SDL_sem *sem, Uint32 timeout);`
  `int SDL_SemWait(SDL_sem *sem);`
  dekrementiert die Semaphore mit Hilfe der Funktion `l4semaphore_down_timed()` bzw. `l4semaphore_down()`.

- `int SDL_SemTryWait(SDL_sem *sem);`
  wartet im Gegensatz dazu, was der Name suggeriert nie, sondern dekrementiert falls möglich die Semaphore per `l4semaphore_try_down()` oder gibt andernfalls einen Fehlerwert zurück.

- `int SDL_SemPost(SDL_sem *sem);`
  ruft die `l4semaphore_up()` auf um die Semaphore zu inkrementieren und damit eventuell einen wartenden Thread wieder anzustoßen.

- `Uint32 SDL_SemValue(SDL_sem *sem);`
  liest den momentanen inneren Zählerstand der Semaphore (`semaphore.counter`) aus und gibt ihn zurück.

3.3.3 Mutexe

Für die Implementierung von Mutexen wurde eine generische Umsetzung auf SDL-Semaphoren, die Bestandteil der SDL ist, angepaßt, um direkt und damit schneller auf die L4env-Semaphoren zuzugreifen.

- `SDL_mutex *SDL_CreateMutex(void);`
  `void SDL_DestroyMutex(SDL_mutex *mutex);`
  `SDL_CreateMutex()` allokiert Speicher für einen neuen Mutex und die zugehörige Semaphore, initialisiert diese mit eins und einen Mutex-internen Zähler mit null. Sie gibt einen Zeiger auf den Mutex zurück. Mit `SDL_DestroyMutex()` kann der Speicher wieder freigegeben werden.

- `int SDL_mutexP(SDL_mutex *mutex);`
  prüft ob der momentane Thread der Besitzer des Mutex ist und inkrementiert den internen Zähler. Andernfalls wird die Semaphore dekrementiert, sodaß gegebenenfalls gewartet wird bis der Mutex freigegeben wird. Danach geht der Mutex in Mutex in den Besitz des momentanen Threads über.

- `int SDL_mutexV(SDL_mutex *mutex);`
  prüft ob der momentane Thread der Besitzer des Mutex ist. Falls nicht kehrt der Aufruf mit gesetztem Fehlerwert zurück. Ist der Test erfolgreich und der interne Zähler größer als null, wird er dekrementiert. Ist der Zähler bereits bei null wird die Semaphore inkrementiert und der Besitzer gelöscht und damit der Mutex wieder freigegeben.

3.3.4 Bedingungsvariablen

Für die Bedingungsvariablen wurde die generische Implementation der SDL, basierend auf auf SDL-Semaphoren und -Mutexes benutzt. Eine Anpassung erfolgte nicht.

3.4 Dateizugriff

Zum Zugriff auf Dateien stehen unter DROPS zwei Schnittstellen zur Verfügung: bootmod und FProv. Über bootmod kann man auf Dateien zugreifen, die durch den Bootloader (GRUB, [11]) als Modul von Festplatte oder per tftp [12] geladen wurden. FProv ist eine allgemeine Schnittstelle zum Zugriff auf Serverprozesse für den Dateizugriff. So gibt es zum Beispiel einen tftp-Client, der diese anbietet. Da sich beide Schnittstellen sehr ähneln, wurde die Nutzung beider implementiert.
Der Zugriff ist denkbar einfach: Man übergibt den Dateinamen an die Funktion zum Öffnen am jeweiligen Server und erhält (falls die Datei zur Verfügung steht) einen „Dataspace", mit dessen Hilfe man den Speicherbereich der Datei im eigenen Speicher einblenden kann. So kann man direkt lesend und schreibend darauf zugreifen. Es wird allerdings keine Möglichkeit geboten, die Größe einer Datei zu ändern. Somit ist auch der schreibende Zugriff auf die ursprüngliche Größe der Datei beschränkt. Desweiteren sind Änderungen an einer Datei nicht persistent, da diese in keinem Fall zurückgeschrieben werden. Beides kann nach Erweiterung der FProv-Schnittstelle und Implementierung eines entsprechenden Servers noch ergänzt werden.

3.5 Ausbaumöglichkeiten

Die Portierung der SDL ist noch nicht vollständig. Sie endet da, wo unter DROPS die entsprechenden Möglichkeiten fehlen. So gibt es keine Unterstützung für Joysticks oder den Zugriff auf Audio-CDs. Die Persistenz von schreibenden Dateizugriffen und die Größenänderung von Dateien sind noch nicht möglich. Desweiteren könnte die Latenz der Audiowiedergabe durch eine Erweiterung des dde_linux verbessert werden (Kapitel 3.2.4).

Kapitel 4

Beispielanwendungen

4.1 Waves

Waves (Abbildung 4.1) ist ein kleines interaktives Demo, das Wellen auf einer Flüssigkeit simuliert und sich mittels Maus beeinflussen läßt. Es wurde als erste SDL-Anwendung portiert, da es nur die Graphik-, Tastatur- und Mausfunktionalität benutzt – welche als erstes zur Verfügung standen – und mit 600 Codezeilen gut überschaubar ist. Der Portierungsaufwand war mit 30 Minuten sehr niedrig.

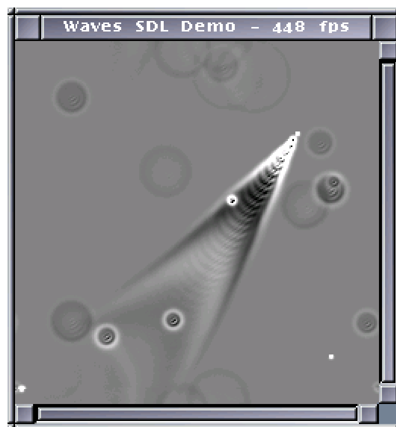

Abbildung 4.1: Waves-Demo

4.2 Quake 1

Die Portierung von Quake 1 (Abbildung 4.2) gestaltete sich aufwendiger, da hier nicht nur SDL-Aufrufe benutzt wurden. Hauptsächlich mußten die Dateizugriffe auf die SDL umgestellt werden. Quake abstrahiert diese zwar, sodaß theoretisch nur die entsprechenden neun

Funktionen – alle zentral an einer Stelle – angepasst werden mußten; doch neben dieser Schnittstelle wurde noch über `fopen()`/`fscanf()`/... und `open()`/`read()`/... an unterschiedlichen Stellen auf Dateien zugegriffen. Desweiteren wurden mehrere Fehler gefunden, die unter anderen Bedingungen noch nicht zum Vorschein kamen.

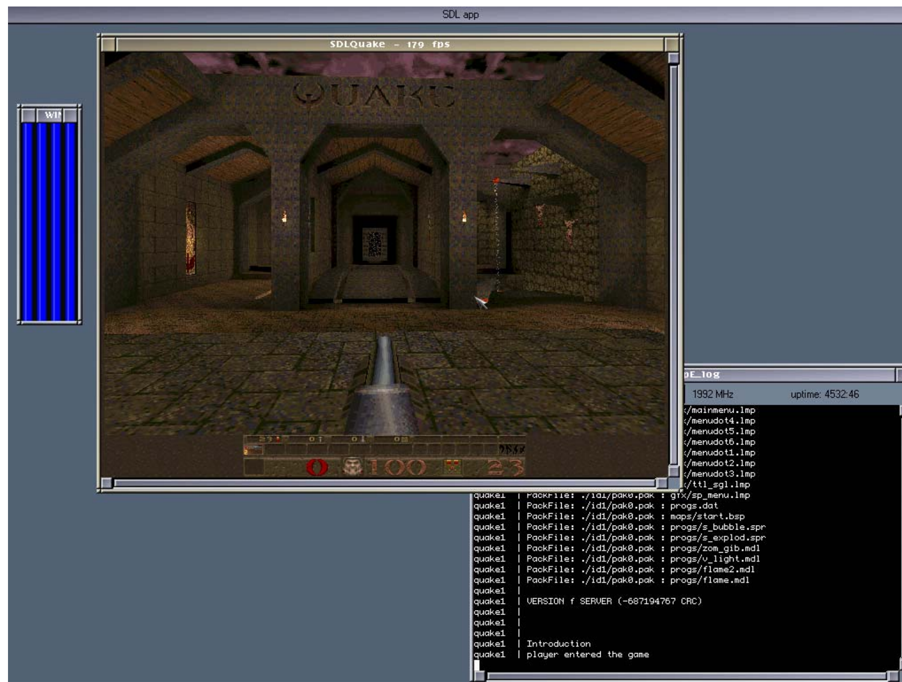

Abbildung 4.2: Quake 1

4.3 Barrage

Barrage (Abbildung 4.3) wurde portiert, um noch ein weiteres Beispiel für die Funktionsfähigkeit der SDL unter DROPS vorzeigen zu können und möglicherweise vorhandene Fehler in der Arbeit zu entdecken. Es benutzt auch die Video-, Audio-, Events- und File-Subsysteme und ließ sich bereits durch minimale Anpassungen portieren.

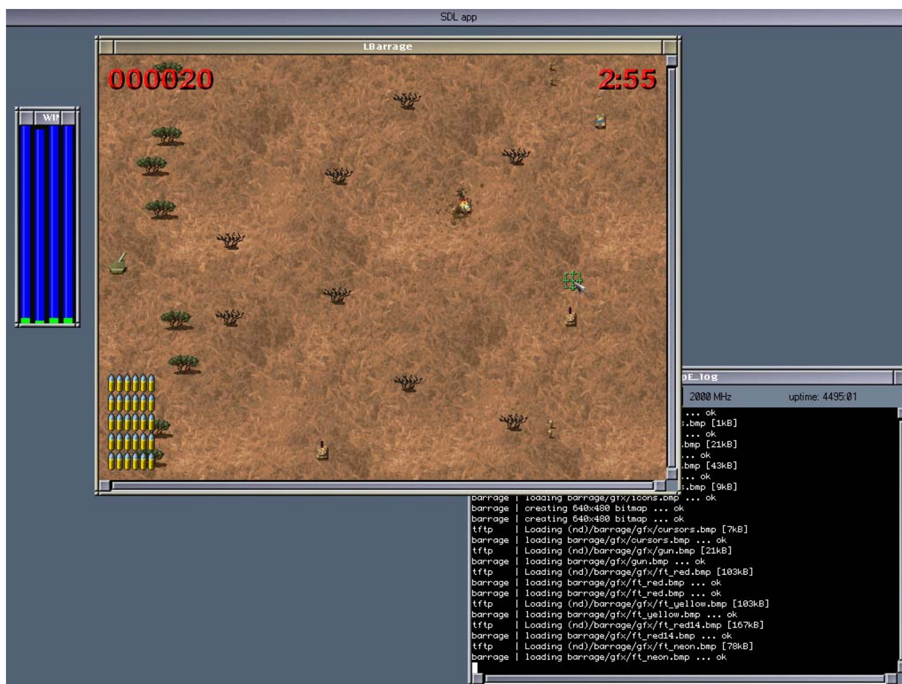

Abbildung 4.3: Barrage

Kapitel 5

Leistungsbewertung

Um eine generelle Einschätzung der Leistungsfähigkeit der SDL unter DROPS zu erhalten, wird in Kapitel 5.2 die Performance von Quake mit der unter DOS verglichen.
In Kapitel 2.1.2 wurde beschrieben, daß die SDL automatisch eine Konvertierung der Farbtiefe des Framebuffers vornehmen kann, falls die von der SDL-Anwendung gewünschte Farbtiefe vom Graphiktreiber nicht angeboten wird. In Kapitel 5.3 soll untersucht werden, welche Performance-Einbußen das Nutzen dieser Fähigkeit mit sich bringt.
Zur Implementierung der Audiowiedergabe wurde das DROPS Sound System geschaffen (siehe Kapitel 3.2.2). Dessen Einfluß auf die Performance wird am Beispiel von Quake in Kapitel 5.4 untersucht.

5.1 Testumgebung

Die Performancemessungen wurden auf einem PC mit folgender Ausstattung durchgeführt:

Mainboard	Gigabyte GA-7IX mit AMD 751 Chipsatz
Prozessor	AMD Athlon, 500 MHz
Arbeitsspeicher	256MB SDRAM, 100 Mhz
Graphikkarte	Creative Labs 3D Blaster (Model CT6930) mit nVidia Riva TNT2 M64 Chip und 32 MB RAM
Soundkarte	Creative Labs Soundblaster 128 PCI mit ES1373 Chip

Es wurde immer fünf Mal mittels `timedemo demo1` gemessen, wie lange Quake braucht, um dieses Demo abzuspielen, und davon die jeweilig beste (kürzeste) Zeit gewertet. Die Abweichung des schlechtesten Wertes gegenüber dem besten betrug nie mehr als ein Prozent.

5.1.1 DOS

Für die Messungen unter DOS wurde das DOS benutzt, welches Windows 98 SE zu Grunde liegt. Windows selbst und andere Anwendungen wurden nicht gestartet, sondern direkt in

den Kommandointerpreter gebootet. Quake benutzte VGA- bzw. VESA-Modi mit 8 bit Farbtiefe. Es wurde mit den Parametern `-nomouse -nosound -nocdaudio` aufgerufen.

5.1.2 DROPS

Unter DROPS wurde mit DOpE mit Stand vom Januar 2004 gearbeitet. Da es nur VESA-Modi mit 16 bit Farbtiefe unterstützt und Quake nur 8 bit Farbtiefe, mußte die SDL die entsprechende Konvertierung vornehmen. Quake wurde mit den Parametern `-nomouse -nosound -nocdaudio` gestartet. Bei den Messungen mit Soundwiedergabe entfiel der Parameter `-nosound`. Die Wiedergabe (2-Kanal, signed/16bit/LSB, 11025Hz) erfolgte und durch einen DSD (siehe Kapitel 3.2.2), zu dem der ES1371/1373-Treiber gelinkt war. Es wurden immer nur die benötigten Server gestartet.

5.1.3 VScreen-Aktualisierung

Bei dem Aufruf von `refresh()` an einem VScreen-Widget zeichnet DOpE es nicht sofort neu, sondern merkt sich die Aktualisierung vor. In regelmäßigen Abständen arbeitet es diese Liste von Aktualisierungen dann ab. Falls ein zweiter `refresh()`-Aufruf erfolgt, bevor die durch einen ersten ausgelöste Aktualisierung begonnen hat, werden die beiden Aktualisierungen zu einer zusammengefaßt (Merging). Wenn diese sich teilweise oder ganz auf den gleichen Darstellungsbereich beziehen, geht die erste Aktualisierung damit entsprechend verloren. Für Quake bedeutet das, daß manche Frames nicht dargestellt werden. Ein direkter Performancevergleich zu Quake auf anderen Plattformen ist anhand der Frameraten also nicht möglich.

5.2 Quake

Hier soll so weit wie möglich die Performance von Quake unter DROPS relativ zu DOS betrachtet werden. Unter DOS sollte Quake sehr performant sein, da zum Beispiel Adressraumwechsel und Semaphoren wegfallen und auf die Hardware, insbesondere die Graphikkarte, direkt zugegriffen werden kann.
Aufgrund des in Kapitel 5.1.3 beschriebenen Effekts kann die wahre Performance unter DROPS nur durch eine Rechnung geschätzt werden. Dazu wurde DOpE so modifiziert, daß es an Stelle des oben genannten Merging die zweite Aktualisierungsanforderung ignoriert. Die Zahl der davon betroffenen und die Gesamtzahl der darzustellenden Pixel wurden aufsummiert. Um die Zeit zu messen, die DOpE für die Darstellung benötigt, wurde für jede Auflösung eine weitere Messung ohne Aktualisierung des Bildschirminhalts durchgeführt. Die Differenz der beiden Messungen ist die von DOpE benötigte Zeit. Aus ihr läßt sich berechnen, wie lange DOpE vermutlich gebraucht hätte, um jede Aktualisierung der Darstellung vorzunehmen.
Getestet wurde bei allen unter DOS zur Verfügung stehenden Auflösungen. DOpE lief beim

Test für 1024x768 bei einer Auflösung von 1280x1024, sonst bei 1024x768. Die Meßwerte in Tabelle 5.1 haben folgende Bedeutung:

t_{DOS} :	Zeit des Tests unter DOS
t_{normal} :	Zeit des Tests unter DROPS, normaler Durchlauf
t_{blind} :	Zeit des Tests unter DROPS, Durchlauf ohne Aktualisierung der Darstellung
p_{soll} :	Anzahl der darzustellenden Pixel (gemessen in der `UpdateRects()`-Funktion des Treibers)
p_{nicht} :	Anzahl der nicht dargestellten Pixel (gemessen in der `add_redraw_action()`-Funktion von DOpE)

DOpE benötigt also $t_{normal} - t_{blind}$ Sekunden für die Darstellung von $p_{soll} - p_{nicht}$ Pixeln. Bei vollständiger Darstellung ist somit eine Durchlaufzeit von Quake in Höhe von

$$t_E = t_{blind} + (t_{normal} - t_{blind}) \times \frac{p_{soll}}{p_{soll} - p_{nicht}}$$

zu erwarten. Dieser Wert kann mit dem Meßergebnis unter DOS verglichen werden. Zum besseren Vergleich der verschiedenen Auflösungen untereinander wurde die Zeit pro einer Million darzustellender Pixel berechnet.

Auflösung	DOS	DROPS [s]		Pixel (Mio.)	
x×y	t_{DOS} [s]	t_{normal}	t_{blind}	p_{soll}	p_{nicht}
320×200	9,3	13,42	9,11	51,1	8,0
360×200	10,3	14,26	9,69	57,4	12,1
320×240	15,0	15,41	10,43	63,6	11,7
360×240	11,8	16,63	11,10	71,4	11,4
640×400	39,7	38,94	23,04	226,0	1,9
640×480	47,2	45,89	26,95	275,7	0,5
800×600	71,2	67,41	38,40	437,5	0
1024×768	113,9	105,42	57,81	726,8	0

Tabelle 5.1: Meßwerte Quake-Performance

Tabelle 5.2 zeigt die Auswertung der Meßergebnisse. Erstaunlich ist, daß unter DROPS in vier der acht Fälle über 100% der DOS-Performance erzielt werden konnte. Zu diesem Gewinn gegenüber DOS führt möglicherweise der Umstand, daß der heutige Compiler sicher besser optimierten Code generiert. Außerdem kann unter DOS nur auf 64kB des Graphikkartenspeichers zugegriffen werden, sodaß dieses Fenster bei hohen Auflösungen sehr häufig verschoben werden muß, wogegen unter DOpE der komplette Speicher eingeblendet wird. Abbildung 5.1 stellt die Performance angegeben in ms/Mio. Pixel graphisch dar.

Auflösung x×y	DROPS t_E [s]	[ms/MPixel] DROPS	DOS	Vergleich DROPS zu DOS
320×200	14,22	278,4	182,0	65,4%
360×200	15,48	269,5	179,3	66,5%
320×240	16,53	260,0	235,9	90,8%
360×240	17,68	247,5	165,2	66,8%
640×400	39,07	172,9	175,7	101,6%
640×480	45,93	166,6	171,2	102,8%
800×600	67,41	154,1	162,7	105,6%
1024×768	105,42	145,0	156,7	108,0%

Tabelle 5.2: Auswertung der Meßergebnisse

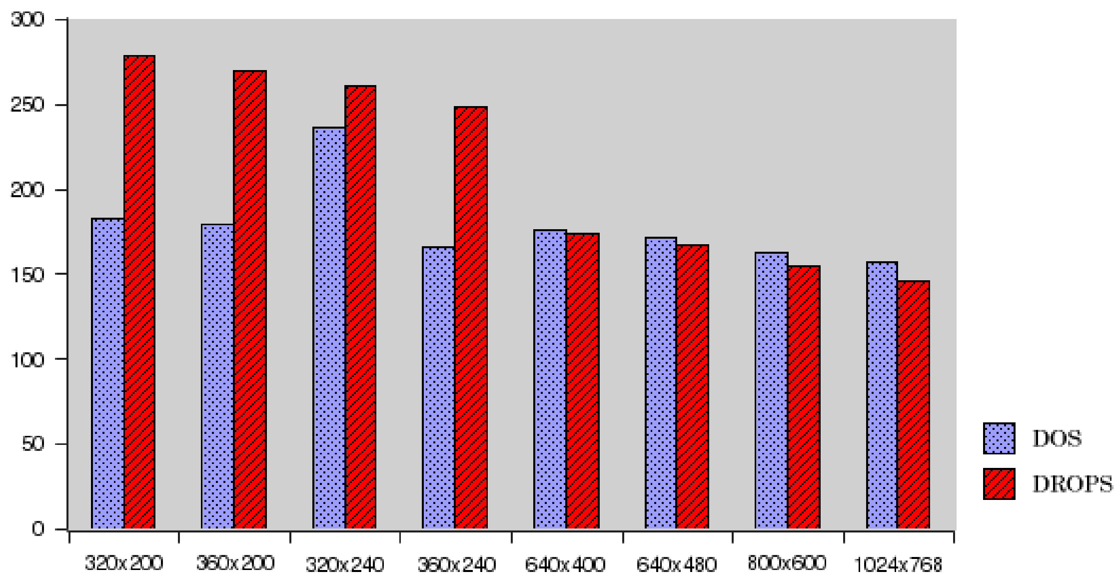

Abbildung 5.1: Quake-Performance unter DOS und DROPS

5.3 Video

Wenn der SDL der gewünschte Videomodus zur Verfügung steht, benutzt die Anwendung zum Zeichnen den mit DOpE gemeinsamen Speicher des VScreen (Kapitel 3.1.2). Die Funktion `SDL_UpdateRects()` zum Aktualisieren des Bildschirminhaltes ruft dann nur die Treiberfunktion `DOPE_UpdateRects()` auf, welche wiederum nur am VScreen-Widget `refresh()` aufruft. Die Performancekosten der zwei zusätzlichen Funktionsaufrufe gegenüber dem direkten Aufruf von `refresh()` sind vernachlässigbar.

5.3.1 Farbtiefen-Konvertierung

Falls der gewünschte Videomodus nicht angeboten wird, erstellt die SDL ein den Anforderungen der Anwendung entsprechendes Shadow-Surface (Kapitel 2.1.2). Bei jedem Aufruf von `SDL_UpdateRects()` wird dann der zu aktualisierende Bildbereich in den VScreen-Speicher kopiert und dabei gegebenenfalls konvertiert. Da dies bei Quake aufgrund der unterschiedlichen Farbtiefen der Fall ist, sollen hier die Performance-Kosten dafür betrachtet werden.
Dazu wurde Quake einmal so compiliert, daß `SDL_UpdateRects()` nicht aufgerufen wurde, und einmal, daß zwar `SDL_UpdateRects()`, aber nicht `refresh()` benutzt wurde. Die Differenz der gemessenen Zeiten ist also die Zeit, welche die SDL für `SDL_UpdateRects()` benötigte – also die Zeit dafür den Bildspeicher zu konvertieren und in den VScreen-Speicher zu kopieren. Zum Vergleich wurde aus Tabelle 5.2 die berechnete theoretische Zeit für einen Durchlauf mit kompletter Darstellung herangezogen. Die unten verwendeten Symbole haben folgende Bedeutung:

Symbol	Quelle	Erläuterung
t_E	Tabelle 5.2	errechneter Wert für volle Darstellung
t_{blind}	Tabelle 5.1	ohne `refresh()`-Aufruf → ohne DOpE
t_{quake}	Messung	ohne `SDL_UpdateRects()`-Aufruf → nur Quake-Engine
t_{konv}	$t_{blind} - t_{quake}$	Kosten der Farbtiefen-Konvertierung
t_{dope}	$t_E - t_{blind}$	Kosten der Darstellung (DOpE)

Auflösung	t_E	t_{blind}	t_{quake}	t_{konv}	t_{dope}
320 × 200	14,22s	9,11s	7,42s	1,69s	5,11s
640 × 400	39,07s	23,04s	17,29s	5,75s	16,03s

Tabelle 5.3: Meßwerte und Auswertung Performance der Farbtiefenkonvertierung

Bei einer Auflösung von 640 × 400 betrugt der Aufwand für den Konvertier- und Kopiervorgang anteilig an der Gesamtlaufzeit also $t_{konv}/t_E = 14,7\%$. Aus Sicht der Anwendung erhöhte sich der Aufwand der Darstellung um $t_{konv}/t_{dope} = 35,9\%$.

Bei 320×200 sind der anteilige Aufwand für die Konvertierung mit $11,9\%$ und die Erhöhung der Darstellungskosten mit $33,1\%$ jeweils etwas niedriger. Die Kosten sind also nicht vernachlässigbar, bei einer aufwendigen Game-Engine aber möglicherweise akzeptabel.

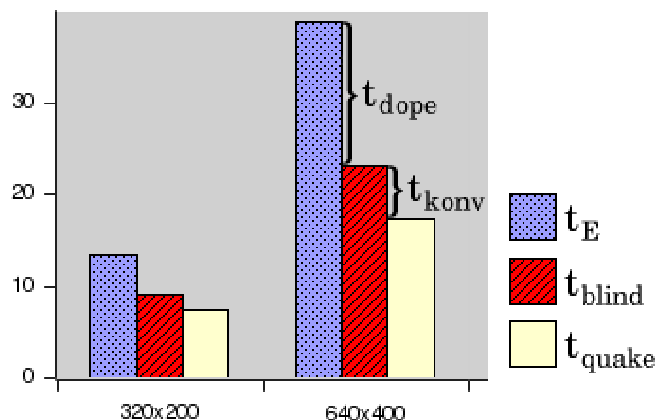

Abbildung 5.2: Kosten der Farbtiefen-Konvertierung

5.4 Audio

Hier soll betrachtet werden wie hoch die Performance-Einbußen durch DSound sind. Zum Vergleich wurde ein SDL-Audiotreiber („null-Audio") geschaffen, der die Daten nur entgegennimmt, aber nicht weiter ausgibt. Dabei wartet die `WaitAudio()`-Funktion so lange, wie die Wiedergabe des Audio-Puffers dauern würde. Ohne diese künstliche Pause würde Quake den Puffer ständig neu mit Hintergrundgeräuschen füllen und dadurch erheblich langsamer laufen. Die Tests erfolgten bei einer Auflösung von 640x400 Pixeln, aber ohne Graphikdarstellung durch `UpdateRects()`, da dies, wie in Kapitel 5.1.3 beschrieben, die Messungen verfälscht hätte.

	Messung/Rechnung	sek
t_{audio}	mit DSound	17,66
t_{null}	mit null-Audio	17,52
$t_{noaudio}$	ohne Audio (-nosound)	17,29
t_E	aus Tabelle 5.2	39,07

Tabelle 5.4: Meßwerte Audio-Performance

Tabelle 5.4 zeigt die Meßergebnisse. Das Mischen der Audiodaten kostet Quake also $t_{mix} = t_{null} - t_{noaudio} = 0,23$s. Die Wiedergabe durch DSound dauert $t_{dsound} = t_{audio} - t_{null} = 0,14$s.

Der Anteil von DSound an der Gesamtausführungszeit beträgt danach

$$\frac{t_{dsound}}{t_E + t_{mix} + t_{dsound}} = 0,35\%$$

und ist somit vernachlässigbar niedrig.

Kapitel 6

Zusammenfassung/Ausblick

Die libSDL wurde mit nahezu allen Aspekten portiert. Für das CDROM- und das Joystick-Subsystem ist im Moment in DROPS noch keine Unterstützung vorhanden. Für die Audiounterstützung wurde mit DSound ein einfacher Sound-Server geschaffen, der als Ausgangspunkt für eine Weiterentwicklung genutzt werden kann. Anhand der drei Anwendungen waves, Quake und barrage konnte gezeigt werden, daß der Port innerhalb der Beschränkungen (Kapitel 3.5) direkt einsetzbar ist. Benchmarks zeigten bei Quake unter DROPS teilweise eine bessere Performance als unter DOS. Mit der SDL steht jetzt also eine funktions- und leistungsfähige Bibliothek zur Erstellung und Portierung von Multimediaanwendungen zur Verfügung. Beispielsweise sollte auch ein DOpE im DOpE-Fenster möglich sein, da die Linuxversion davon nur die SDL benutzt.

Aufbauend auf dieser Arbeit könnte man betrachten, ob es möglich ist, Quake echtzeitfähig zu machen. Dazu müßte man die Verteilung der Zeit messen, die Quake benötigt um ein Frame zu rendern, und untersuchen ob eine niedrige Zeitschranke existiert, die nahezu die gesamte Verteilung abdeckt. Desweiteren müßten die Abhängigkeiten von DSound und DOpE betrachtet und DSound echtzeitfähig gemacht werden. Vorstellbar wäre auch eine echtzeitfähige Netzwerkunterstützung für Quake.

Literaturverzeichnis

[1] http://www.libSDL.org/

[2] http://www.gnu.org/licenses/

[3] http://www.idsoftware.com/

[4] http://aa-project.sourceforge.net/aalib/

[5] http://www.directfb.org/

[6] http://www.svgalib.org/

[7] http://www.x.org/X11.html

[8] John Reeves Hall. *Programming Linux Games*. No Starch Press, 2001

[9] http://os.inf.tu-dresden.de/dope/

[10] http://os.inf.tu-dresden.de/l4env/

[11] http://www.gnu.org/software/grub/, http://os.inf.tu-dresden.de/~fm3/grub.html

[12] http://www.ietf.org/rfc/rfc1350.txt