

# DRI und 3D unter Linux



Daniel Mangold  
Universität Ulm

- Linux Proseminar Sommersemester 2002 -

# **Inhaltsverzeichnis:**

## **1. Vorwort**

## **2. OpenGL**

### **2.1. Was ist OpenGL ?**

### **2.2. Kurz zur Geschichte**

### **2.3. Das ARB (Architecture Review Board)**

### **2.4. OpenGL 2.0**

### **2.5. Einiges zur Architektur**

### **2.6. Die „Graphics Rendering Pipeline“**

#### **2.6.1. Der Anwendungsabschnitt**

#### **2.6.2. Der Geometrieabschnitt**

#### **2.6.3. Der Rasterungsabschnitt**

## **3. Mesa**

### **3.1. Mesa ...**

### **3.2. Geschichte**

## **4. DRI – Die Direct Rendering Infrastructure**

### **4.1 Einleitung**

### **4.2 Motivation**

### **4.3. Low Level Design Document**

#### **4.3.1. Der Aufbau**

#### **4.3.2. Initialisierung**

#### **4.3.3. Der Bereitschaftsstatus**

#### **4.3.4. Prozess Abschließung**

## **5. Glossar**

# 1. Vorwort

Ein kleines Vorwort zum Aufbau dieses Proseminars. DRI (Direct Rendering Infrastructure) ist die in XFree86 4.x integrierte 3D Hardwareunterstützung für Linux. Es ist eine Infrastruktur, die verschiedene Module und Bibliotheken bereitstellt und die wesentlichen Komponenten koordiniert. Eine wichtige Rolle für das allgemeine Verständnis spielen dabei OpenGL, bzw. Mesa. Das sind 3D Rendering Bibliotheken, die in die Struktur der DRI integriert sind. OpenGL und Mesa sind sich sehr ähnlich. In Wirklichkeit wird für die DRI Mesa verwendet, aber OpenGL ist weiter verbreitet ist und bekannter. Aus diesen Gründen befasst sich ein großer Teil des Proseminars mit OpenGL, obwohl OpenGL nicht Linux spezifisch ist, sondern Plattformunabhängig.

Der Teil über DRI geht den gesamten Prozess von der Initialisierung der DRI Ressourcen, über den Bereitschaftsstatus zum Rendern von 3D Grafiken, bis zum Abschluss des Prozesses durch. Zusätzlich wird dort beschrieben, warum man diese Struktur gewählt hat.

## 2. OpenGL

### 2.1. Was ist OpenGL ?

OpenGL (Open Graphics Library) ist eine Open Source Grafikbibliothek, was bedeutet, dass man mit dessen Hilfe Grafiken im 2D und 3D Bereich erzeugen kann. Typische Anwendungen für 2D sind z.B. Bilder oder Filme. 3D Anwendungen, die OpenGL nutzen sind z.B. CAD Programme, komplexe Grafikprogramme, Spiele und andere Multimedia Anwendungen.

OpenGL ist stark optimiert und wurde (ursprünglich) speziell für Hochleistungszwecke entworfen. Da aber inzwischen gute Grafikkarten sehr billig geworden sind, findet man OpenGL fast überall, wo Endanwendungen grafische Ansprüche haben.

OpenGL wird durch eine API (Application Programming Interface) angesprochen. Konkret sind das ca. 120 Routinen, die Grafik Operationen und Manipulationen implementieren.

Es ist weltweiter Industriestandard, der von einem unabhängigen Konsortium (OpenGL Architecture Review Board - ARB) überwacht und geformt wird. Momentan sind in diesem Konsortium als aktive Mitglieder 3Dlabs, Apple, ATI, Dell Computer, Evans & Sutherland, Hewlett-Packard, IBM, Intel, NVIDIA, Microsoft, SGI, Sun (Stand Dezember 2001).

OpenGL ist auf die meisten relevanten Betriebssysteme portierbar. Das hat zur Folge, dass OpenGL basierte Anwendungen auf jedem System laufen können, wo es OpenGL gibt. Das kann von Workstation, PC, Konsole bis zum Supercomputer ziemlich viel sein. Zusätzlich ist es einfach zu bedienen, da mit intuitivem Design und logischen Befehlen gut strukturiert. Und es gibt eine ausführliche Dokumentation und Literatur.

Das sind vermutlich die Gründe, warum sich OpenGL durchgesetzt hat und inzwischen die Industrielle Grundlage für Hochleistungsgrafiken ist. Natürlich gibt es auch noch Mesa, aber dazu später.

## 2.2. Kurz zur Geschichte

Die Idee von OpenGL bestand seit 1989 von der Firma SGI (Silicon Graphics Inc.) und wurde in den folgenden 3 Jahren umgesetzt. 1992 gab es dann die erste Version 1.0.

OpenGL ist ein direkter Nachfolger von IRIS GL, einer Hardware unabhängigen Grafikbibliothek, die speziell für 3D Grafikworkstations von SGI entworfen wurde. IRIS GL wird von über 1500 3D Anwendungen genutzt, aber es gibt keine Abwärtskompatibilität von OpenGL zu IRIS GL. Das einzige, was wirklich erhalten wurde, war die Idee der API.

## 2.3. Das ARB (Architecture Review Board)

Wie wir vorher gehört haben ist das ARB eine interessante Einrichtung, die im wesentlichen sehr dazu beigetragen hat, dass sich OpenGL so bewährt hat und auch auf lange Sicht hin sehr stabil ist und nicht dauernd überarbeitet werden muss.

Alles ist standardisiert und man hat ein API, das plattformunabhängig agieren kann. Doch es gibt auch Nachteile, die das ARB mit sich bringt, und diese kommen inzwischen auch zum tragen.

Wenn sich auf dem Markt etwas tut und es neue Erkenntnisse über die Grafik erfordern würden, dass man neue Routinen implementieren müsste, dann muss im Konsortium zuerst diskutiert werden, ob überhaupt und in welcher Form dies stattfinden soll.

Da sich in letzter Zeit in der Hardware Entwicklung sehr viel getan hat, entstanden neue Grafikstandards, wozu es aber in OpenGL keine direkten Methoden zum umsetzen gab. Das war vor allem im 3D Spiele Bereich der Fall. Die logische Konsequenz daraus ist, dass verschiedene Firmen OpenGL Erweiterungen anbieten, die aber nicht zum offiziellen Standard der ARB gehören. Es sind dennoch offizielle Erweiterungen. Das hängt damit zusammen, dass es keiner Grafikkartenfirma viel bringt, eigene Routinen zu schreiben, die eine eigene Anwendung zu allen anderen Grafikkarten inkompatibel machen würde.

Anders herum hätten man mit eigenen Erweiterungen nur Demos oder Grafikkarten spezifische Anwendungen oder Spiele, die wunderbar zeigen, was die neue Grafikkarte alles kann, hätte aber ansonsten keine Anwendungen auf dem freien Markt, die die neuen Features der Grafikkarte nutzen können.

Der OpenGL 1.x Standard hat sich jetzt schon seit über 10 Jahren gehalten. In dieser Zeit hat sich natürlich sehr viel Entwicklung von Seiten der Grafikkartenhersteller vollzogen. Das hat dazu geführt, dass OpenGL 1.x inzwischen einfach nicht mehr zeitgemäß ist. Aus diesem und obig erwähnten Gründen ist hauptsächlich 3D Labs gerade damit beschäftigt eine neue OpenGL Spezifikation zu schreiben - OpenGL 2.0 - die so zeitgemäß werden soll, dass sie das gesamte Grafikspektrum für über 10 Jahre wieder abdecken soll. Ob das wohl funktioniert?

## 2.4. OpenGL 2.0

3Dlabs hat ca. Mitte 2001 angefangen, einen neuen OpenGL 2.0 Standard in die Wege zu leiten. Dieser ist aber zum jetzigen Zeitpunkt (Mai 2002) noch nicht fertig.

Der vorwiegende Grund ist, dass OpenGL 1.x ein Update benötigt, um dem Trend zu programmierbarer Grafikhardware entsprechen zu können. Zusätzlich zu dieser Unterstützung bietet es sich hier auch gleich an, das ursprüngliche OpenGL auf den neuesten Stand zu bringen, indem einige Aspekte überarbeitet werden, die doch schon seit 10 Jahren existieren.

OpenGL 2.0 wird abwärtskompatibel sein. Von den Mitgliedern des ARB ist die bisherige Arbeit von 3Dlabs positiv aufgenommen worden.

## 2.5. Einiges zur Architektur

Obwohl die OpenGL Spezifikationen eine einheitliche 'Graphics Processing Pipeline' definieren, haben Betriebssystemhersteller die freie Möglichkeit, sich einzelne OpenGL Implementierungen zurechtzuschneiden. Dadurch können Kosten gespart werden (man muss das Rad nicht jedes mal neu erfinden) und es hat natürlich weitere Vorteile, da OpenGL schnell und relativ ausgereift ist. Individuelle Befehle können auf der Grafik Hardware ausgeführt werden, als Standard Software Routinen auf jeder CPU laufen oder als eine Kombination von beidem implementiert werden. Diese Flexibilität ermöglicht, dass OpenGL Hardwarebeschleunigung von einfachem Rendern bis zu komplexen Geometrie Operationen gehen kann.

Auf der anderen Seite kennen Anwendungsprogrammierer die Befehle der API (Doku), können also ihre Programme entwickeln, ohne sich um irgendwelche Plattform- oder Hardware- Implementierungen kümmern zu müssen.

## 2.6. Die „Graphics Rendering Pipeline“

Der Vorgang des Renderings in OpenGL findet über eine Pipeline statt. Das bedeutet erst mal nicht viel mehr, als dass eine OpenGL Anfrage von einem Anwendungsprogramm gestartet wird und in verschiedenen Abschnitten sequentiell abgearbeitet wird. Wenn man das jetzt auf eine Pipeline bezieht, dann fällt folgendes auf:

Teilt man die Pipeline in  $n$  Verarbeitungsabschnitte auf, die jeweils einen gewissen Teil verarbeiten, dann wäre es ideal, wenn jeder der  $n$  Teile genau  $t$  Zeit (also jeweils gleichviel Zeit) benötigt, um abgearbeitet zu werden. Eine Pipeline ist nur so schnell, wie ihr langsamstes Verbindungsstück an Durchsatz hat. Benötigt also von  $n$  Teilen das  $i$ -te Teil ( $i < n$ ) zum Beispiel  $t+1$  Sekunden statt  $t$  Sekunden, dann benötigt eine Anfrage genau  $(t+1)*n$  Zeit, um verarbeitet zu werden. Deswegen sind innerhalb der Pipeline größere Abschnitte auch wieder klein unterteilt, um dem Ideal weitgehendst zu entsprechen.

Die Grundlegende Struktur der Rendering Pipeline besteht aus 3 Abschnitten: Anwendungsabschnitt, Geometrieabschnitt und Rasterungsabschnitt.

### 2.6.1. Der Anwendungsabschnitt

Der Entwickler hat hier die volle Kontrolle über das, was passieren soll, da diese Schicht immer von der Software aus ausgeführt wird. In diesem Abschnitt kann man 3D Szenen erzeugen und z.B. mit Eingabegeräten abstimmen. (Bei Spielen ist das die Graphics Engine). Am Ende dieses Abschnitts werden die Geometriedaten, die gerendert werden sollen an den nächsten Abschnitt der Pipeline weitergegeben. Die Geometriedaten sind Rendering Primitiven, wie z.B. Punkte, Linien und Dreiecke.

## 2.6.2. Der Geometrieabschnitt

Hier finden die ganzen Operationen auf den Grafikprimitiven statt. Der Geometrieabschnitt ist weiter unterteilt in funktionale Abschnitte. Die Unterteilung kann jedoch mehr oder weniger ausgeprägt sein. Das kommt ganz darauf an, ob verschiedene Schritte von der Hardware (z.B. einem Gleitzahlenprozessor auf der Grafikkarte) abgenommen werden können oder ob alles durch Software gemacht wird. Hier unterscheiden sich die Implementierungen zum Teil stark. Der Geometrieabschnitt ist übrigens sehr rechenintensiv!

### Die funktionalen Abschnitte:

#### *1. Modell und Sichttransformation*

Wenn man sich nun also eine Szene aufbauen will, z.B. einen Wohnraum mit Tisch und 5 Stühlen. Dann muss man nicht 5 Stühle programmieren. Es reicht, einen zu gestalten. Dieser ist dann anfangs untransformiert, also in seinem sogenannten eigenen „model space“. Jetzt kann man verschiedene Instanzen auf dieses Modell setzen, mit jeweils einer definierten Modelltransformation. Das bedeutet, dass die 5 Stühle in gewünschten Positionen um den Tisch stehen können, aber alle aus einem einzigen programmierten Stuhl resultieren. Nach diesen Transformationen befinden sich alle Modelle im sogenannten „world space“. Dieser ist einheitlich, und auch der Tisch, weitere Einrichtung, ..., befinden sich darin.

Anschließend findet eine weitere Transformation des world space statt. Man nimmt den Punkt, wo die Kamera steht und normt ihn so ein, dass die Kamera auf dem Nullpunkt des Koordinatensystems steht und in Richtung negative z-Achse schaut. Das macht für die Szene keine Änderung, ist nur für die folgenden Abschnitte einfacher zu rechnen.

#### *2. Belichtung und Schattierung*

Um die Szene realistischer wirken zu lassen, kann man sie mit einer oder mehreren Lichtquellen ausstatten. Um die neuen Farben nach dem Lichteinfall zu berechnen, gibt es zwei Möglichkeiten. Für die erste gibt es eine Gleichung, die eine Approximation an die echten Verhältnisse unserer Welt zwischen Photonen und Oberflächen darstellt.

Da dies bei Echtzeitgrafiken zur Berechnung zu viel Zeit kosten würde, ist die zweite Methode eine Interpolation der Farbverläufe über ein Dreieck (die kleinste rendering Primitive in einer Szene).

#### *3. Projektion*

Nachdem die Szene mit Lichtquellen versehen wurde, wird das Sichtfeld (also nur der Teil, der durch die Kamera erfasst wird) in einen „Einheitswürfel projiziert, deren extreme Punkte  $(-1 \ -1 \ -1)^T$  und  $(1 \ 1 \ 1)^T$  sind. Danach ist immer noch die ganze Szene vorhanden, mit den „genormten Koordinaten“, aber alles was die Extrempunkte des Einheitswürfels überschreitet, ist nicht innerhalb des Sichtfeldes.

Hier gibt es im wesentlichen 2 Methoden, die orthografische (oder parallele) Methode und die perspektivische. Bei der orthografischen bleiben alle ursprünglich parallelen Geraden auch danach noch parallel, d.h der Einheitswürfel bleibt auch ein parallelkantiger Einheitswürfel.

Bei der perspektivischen Methode hat man einen Fluchtpunkt, so dass ursprünglich parallele Geraden sich am Horizont schneiden. Hierbei projiziert man also den Einheitswürfel in eine zur Spitze hin abgeschnittene Pyramide. Nach einer dieser Projektionen haben alle Modelle sogenannte normierte Gerätekoordinaten.

#### 4. Clipping (Zurechtschneiden)

Es werden letztendlich nur die Koordinaten gerendert, die auf dem Bildschirm ausgegeben werden sollen. Modelle, die innerhalb des Einheitswürfels oder außerhalb sind, müssen nicht zurechtgeschnitten werden. Nur Modelle, die einen Teil innerhalb und einen außerhalb dieses Würfels haben, müssen zurechtgeschnitten werden. Hierzu müssen für die Schnittebene mit dem Einheitswürfel neue Randkoordinaten berechnet werden.

#### 5. Screen Mapping (Bildschirm anpassen)

In diesem Stadium werden die x- und y-Koordinaten auf das Bildschirmfenster umgerechnet, denn man kann ja nicht davon ausgehen, dass immer in einem Vollbild gerendert wird, außerdem sind die Koordinaten ja normiert. Die z-Koordinaten bleiben dabei vollkommen unberührt, da sie ja nicht auf den Bildschirm kommen können und somit nur Träger der Tiefeninformation sind (Erklärung folgender Abschnitt). Jetzt hat man endlich die Bildschirmkoordinaten und kann diese an den nächsten Abschnitt der Pipeline weitergeben.

### 2.6.3. Der Rasterungsabschnitt

In diesem Abschnitt findet die Abbildung der Szene von  $\mathbb{R}^3 \rightarrow \mathbb{R}^2$  statt (also vom 3D nach 2D). Er ist also verantwortlich, dass das fertige Bild korrekte Farben hat und korrekt gerendert wurde.

Dieser Abschnitt des Renderings findet ausschließlich in Puffern statt. Um den unschönen Nebeneffekt zu vermeiden, dass man auf dem Bildschirm den Aufbau der Polygone und Dreiecke sehen kann, gibt es das sogenannte „Double Buffering“. Es gibt hier den Vordergrundpuffer und den Hintergrundpuffer. Der Vordergrundpuffer ist immer auf dem Bildschirm sichtbar und in dem Hintergrundpuffer wird das neue Bild zusammengebaut.

Das bedeutet, dass das eigentliche Rendering im Hintergrund stattfindet und nicht auf dem Bildschirm (also off screen).

Wenn eine Szene fertig gerendert ist, dann werden einfach der Vordergrund- und der Hintergrundpuffer getauscht. Dies geschieht zu einem Zeitpunkt, wenn der Elektronenstrahl des Bildschirms das Bild nicht stören kann.

Weiterhin gibt es den Z-Puffer (oder Tiefenpuffer). Es passiert ja recht oft, dass sich 3D Modelle aus Sicht der Kamera überlagern, also das eine verdeckt das andere. Natürlich darf man ausschließlich die Vorderseite desjenigen Modells sehen, welches am nächsten zur Kamera ist. Sonst gibt es Überlappungen, die das Bild insofern zerstören, dass das Modell, welches zuletzt gerendert wird auch sichtbar ist und nicht das, welches am nächsten zur Kamera ist.

Der Algorithmus ist sehr einfach und funktioniert folgendermaßen: Beim Rendern einer Szene wird der Z-Wert eines Modells mit dem Z-Wert im Z-Puffer verglichen. Ist der Z-Wert kleiner, dann wird das Modell, welches näher zur Kamera ist, als das, welches sich momentan im Z-Puffer befindet gerendert. Der Wert im Z-Puffer mit dem neuen Wert überschrieben. Im umgekehrten Fall, also der aktuelle Z-Wert ist größer als der im Z-Puffer, wird der Z-Puffer unberührt gelassen.

Gleichzeitig mit dem setzen des Z-Puffers wird auch der Farbwert im Farb-Puffer gesetzt. Dieser enthält die entsprechende Farbinformation der jeweiligen Grafikprimitiven.

Durch den Z-Puffer spielt die Reihenfolge der gerenderten Modelle eigentlich keine Rolle mehr. Deswegen ist diese Methode auch so beliebt. Dennoch muss man die Reihenfolge beachten, wenn man Transparente Modelle vor undurchsichtigen hat (z.B. Wasser).

Dann gibt es noch einige andere Puffer, z.B. für verschiedene Effekte, die ich an dieser Stelle aber nicht erwähnen will.

Alle diese Puffer werden allgemein „Frame Buffer“ genannt.

In diesem Abschnitt findet zusätzlich auch noch „Texturing“ statt, also es werden (meist 2 Dimensionale) Bilder auf 3D Modelle gelegt, wodurch eine Szene einfach realistischer wirkt. Nachdem eine Szene dann fertig gerastert ist, werden die sichtbaren Grafikprimitiven auf dem Bildschirm dargestellt.

## **3. Mesa**

### **3.1. Mesa ...**

... der Name hat wieder Erwartens nichts zu sagen!!! Es ist wie OpenGL eine 3D Grafik Bibliothek mit einer API, die der von OpenGL sehr verwandt ist (um nicht fast gleich zu sagen).

### **3.2. Geschichte**

Warum wurde Mesa überhaupt entwickelt?

Der Grund lag darin, dass OpenGL nicht immer so offen und frei war, wie es jetzt ist. Für den Vertrieb von OpenGL waren Lizenzen notwendig, die man kaufen konnte. Das „Open“ von OpenGL kommt nicht daher, dass es Frei, oder Open Source ist, sondern weil es von Anfang an ein offener Standard (siehe ARB) war.

1992 kam die erste Version 1.0 von OpenGL auf den Markt. 1993 hat Brian Paul angefangen, eine Implementierung der OpenGL Specification zu schreiben. Das hatte praktische und persönliche Gründe. Paul benützte einen GL Emulator, genannt VOGL für die Visualisierung eines wissenschaftlichen Projektes, als OpenGL angekündigt wurde. Er dacht sich, dass es sicher Spaß machen würde, eine einfache 3D Grafik-Bibliothek zu schreiben, die das OpenGL API benutzen würde und VOGL ersetzen könnte. Nach 18 Monaten Teilzeitentwicklung und der Klärung der Rechtsfrage mit SGI (er hat ja dieselben Schnittstellenbefehle benutzt, wie OpenGL), durfte Paul sein Projekt im Internet veröffentlichen. Sein Projekt hat viele Menschen angesprochen und es haben auch sofort einige zu der Entwicklung beigetragen. Paul nannte seine 'Graphics Library' Mesa.

Das Ziel von Mesa war in erster Linie eine sichere und zuverlässige GL bereitzustellen. Korrektheit hatte also einen höheren Stellenwert als Leistung (Geschwindigkeit). Die Endbenutzer haben diese Entwicklung von Mesa positiv aufgenommen und somit war Mesa recht früh eine gute Alternative zu OpenGL.

1997 wurde die erste Hardware Grafikerunterstützung zu Mesa hinzugefügt. Es war ein Glide Treiber für die neu entwickelten 3Dfx Voodoo Grafikkarten. Und wieder einmal waren die Augen auf Linux gerichtet, denn auch hier wollte man die neuen Features der 3D Grafikkarten für den täglichen Gebrauch nutzen können. Aber in diesem Stadium war klar, dass es ein langer Weg sein würde, bis Linux mit den aktuellen Grafikworkstations (oder auch nur Windows PC's) würde mithalten können. Zu der Zeit war die Hardware aber auch noch nicht

perfekt. Die 3Dfx Voodoo Karte war nur auf Vollbild Grafiken beschränkt und hatte zudem nur relativ geringe Auflösungen - nicht nur Pixel, sondern auch bei der Farbpalette.

Der Mesa/Glide Treiber war NICHT in das X-Window System integriert und nur eine Minderheit der OpenGL Anwendungen konnte von der Hardwarebeschleunigung profitieren. Kurz darauf war die Hardware dabei, die entscheidenden Einschränkungen (s.o.) abzulegen, aber auf der anderen Seite stand bei der Software (für Linux) eine unglaubliche Herausforderung bevor. Eine wirkliche Lösung für eine 3D Workstation würde Arbeit im Linux Kernel, im X Server, den Hardware Treibern und dem rendering Kern erfordern.

## 4. DRI – Die Direct Rendering Infrastructure

(...man sagt auch 3D für XFree86 oder 3D für Linux)

### 4.1 Einleitung

Ziemlich lange Zeit gab es also für Linux keine 3D Hardwareunterstützung, zumindest keine, die frei erhältlich war (also z.B. in XFree86 integriert). Es gibt natürlich wie unter Unix käuflich zu erwerbende X Server, die dann eine Hardwarebeschleunigung für die gewünschte Grafikkarte unterstützen. Die interessieren uns aber nicht.

Inzwischen hat quasi jeder PC Besitzer eines aktuellen Standardmodells die Hardwaretechnischen Voraussetzungen (also eine tolle Grafikkarte) und will die natürlich nicht nur z.B. unter Windows, sondern auch unter Linux zum Einsatz bringen.

Die DRI erlaubt es, sicher und effizient auf die 3D Grafikhardware zuzugreifen, wie es z.B. unter Windows DirectX tut.

DRI ist nicht ein einzelner Treiber, oder ein einzelnes Programm, dass dann für die gesamte Grafik in Linux verantwortlich ist. Vielmehr sind es verschiedene Bibliotheken und Module, die den X11 Client, den X Server und den Kernel erweitern oder ergänzen.

Die wichtigste Funktion der DRI ist, OpenGL Anfragen schnell zu verarbeiten. Anwendungsprogramme, die OpenGL als Grafik API benutzen sollen über die DRI die Möglichkeit bekommen, direkt die Features der 3D Grafikhardware anzusprechen.

Eine Konsequenz des direct Rendering ist, dass so erzeugte Grafiken nicht mehr über das XFree86 GLX Protokoll (also Netzwerk) portierbar sind. Ist aufgrund der Leistungsanforderung bis jetzt auch gar nicht machbar.

Die Precision Insight Inc. wurde gegründet, um die notwendigen Hardwaretreiber für XFree86 zu entwickeln und die Entwicklungen der DRI voranzutreiben. Zusätzlich wollten sie sich im Bereich der 3D Grafiken ausbreiten. Mesa Autor Brian Paul ist dort an der Treiberentwicklung beteiligt.

Da die Entwicklungen zum jetzigen Zeitpunkt noch recht jung sind, werden hauptsächlich neuere AGP Standardgrafikkarten unterstützt, wie ATI, Matrox, 3Dfx. Es gibt auch 3D Hardware Unterstützung für Nvidia Karten (GeForce Chipsätze), aber Nvidia hat eigene Treiber, die nicht Open Source sind. Somit kennt man auch deren Implementierung nicht.

## 4.2 Motivation

Prinzipiell funktioniert das Grafiksystem unter Linux folgendermaßen. Es gibt 3 wesentliche Komponenten: Den Kernel, den X Server und einen X11 Client. Der Kernel stellt die Hardwarespezifischen Details bereit. Der X Server bietet eine Abstraktionsebene, über die die Clients (also irgendwelche Anwendungsprogramme) komfortabel auf Grafikroutinen zugreifen können. Die Kommunikation zwischen einem Client und dem X Server wird wie schon erwähnt über das X Protokoll (ein Netzwerkprotokoll) aufgebaut. Man hat also die Möglichkeit zum Remote Betrieb, also Programme auf einem Server zu starten und das Display über Netzwerk zu verschicken.

Beim direct Rendern (also das was die DRI tut) versucht man, den Weg über das X Protokoll auszusparen, da es zu langsam ist, und direkt über den Kernel Treiber zu rendern.

Eine entscheidende Frage, die sich jetzt vielleicht stellt ist, warum man überhaupt so viel Aufwand betreibt, die 3D Hardware Beschleunigung einerseits in den X Server zu integrieren, diesen dann aber andererseits wieder zu übergehen versucht. Weiterhin stellt sich vielleicht sogar die Frage, warum man überhaupt über den Kernel geht, so wertvolle Zeit verliert und nicht gleich im Client Bibliotheken hat, die direkt auf die Grafikhardware zugreifen können. So hat man das z.B. früher unter DOS gelöst. Das hat natürlich Geschwindigkeitsvorteile.

Die Antworten darauf und gleichzeitig die Motivation für die DRI sind folgende. DOS Methodik kommt nicht in Frage, denn das wäre ein Rückschritt, weg vom Multiuser und Multitasking Betriebssystem. Beim diesem Beispiel hätte man nur noch die Möglichkeit einen einzigen Client zu haben, der auf die Hardware zugreifen kann. Zusätzlich hat man ein Problem, wenn sich dieser Client nicht „sauber“ beendet (also z.B. nicht alle Speicherbereiche freigibt o.ä.) und man hat ein noch größeres Problem, wenn dieser Client aus irgendwelchen Gründen abstürzt. Dann ist das ganze Betriebssystem lahmgelegt.

Weiterhin will man die Option, mehrere 3D Clients parallel laufen zu lassen, also z.B. mehrere Fenster offen zu haben und diese auch verschieben zu können oder ähnliches, was ja nicht über den Client läuft, sondern über den X Server.

Auf der anderen Seite ist das X Protokoll einfach zu langsam, denn die ganzen Datenpakete müssen zuerst enkodiert, dann verschickt (wenn auch nur lokal), dann wieder dekodiert werden, ausgewertet werden,... . Das kostet alles zu wertvolle Zeit, wenn man die Mengen an Datenfluss betrachtet, die vor allem bei 3D Echtzeitgrafiken im Umlauf sind.

Man sitzt also in der Klemme. Einerseits will man einen hohen, möglichst direkten und somit ungebremsten Datenfluss zur Hardware haben. Andererseits will man die Vorzüge einer Abstraktionsebene und die Tatsache, dass das Betriebssystem zu jedem Zeitpunkt den vollen Überblick und die volle Kontrolle hat.

Die DRI implementiert einen Mittelweg. Der X Server behält den Überblick. Er weist alle benötigten Ressourcen zu, die ein Client benötigt. Zusätzlich hat er zu jeder Zeit Zugriff auf Statusinformationen, die sich der Client, der Kernel und der X Server teilen. Im Endeffekt kann der Client jedoch nachdem ihm alle nötigen Ressourcen zugewiesen wurden quasi direkt über den Kernel Anfragen an die Hardware schicken. Das ist das Konzept. Der nächste Abschnitt macht deutlich, wie dies relativ Hardware nahe funktioniert.

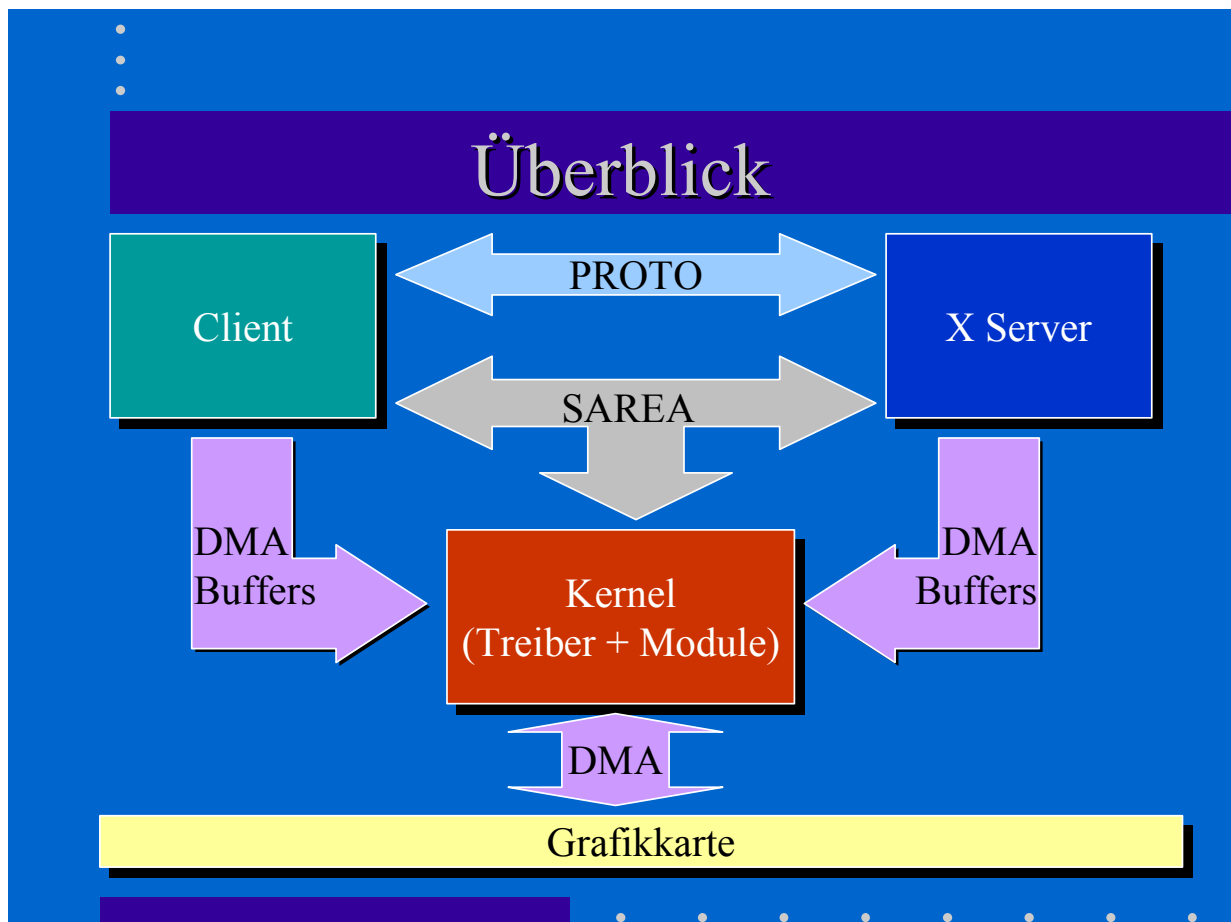
### 4.3. Low Level Design Document

Im folgenden soll erklärt werden, in welchen Modulen DRI verankert ist, wie diese Module untereinander kommunizieren und wie verschiedene grundlegende Grafikkonzepte hier zum tragen kommen und funktionieren. Der folgende Abschnitt heißt „Low Level Design Document“, da er sich mit dem Zusammenspiel der Komponenten auf relativ niedriger Ebene, also Treiber und Hardwarenahe, beschäftigt.

Die grundlegende Architektur der DRI beinhaltet 3 separate Komponenten:

- den X-Server
- einen direct rendering Client
- einen Gerätetreiber auf Kernelebene

Und so sieht das ganze bildlich aus:



(Abbildung 4.2a)

Innerhalb des Client und X Server fehlen natürlich noch Module und Abstraktionsschichten, die zur Übersicht weggelassen wurden.

### 4.3.1. Der Aufbau

Der gesamte Verlauf, vom initialisieren der entsprechenden Module und Komponenten, bis zur deren Nutzung durch eine 3D Anwendung, geht also grob folgendermaßen vonstatten:

Wenn man allgemein etwas mit Grafik unter Linux machen will - egal ob 3D oder 2D - so läuft dies grundsätzlich erst mal über den *X Server*. Der X Server ist daher die erste Anwendung, die gestartet werden muss und mit Direct Rendering in Berührung kommen kann. Um die DRI zu unterstützen musste der X Server durch verschiedene Modifikationen angepasst werden. Weiterhin kann er mit weiteren Modulen verlinkt werden, um direktes Hardware rendering zu unterstützen. Darunter fallen Bibliotheken wie libDRI – Grafikkarten- und Betriebssystemabhängiger Treiber libH2D, ebenfalls Geräteabhängig. Diese Bibliothek kann hardwarespezifisches 2D rendering ermöglichen und verschiedene 3D Initialisierungs- und Finalisierungsroutinen, die vom Client nicht benötigt werden, beinhalten.

Nachdem der *X Server* seine eigenen Quellen initialisiert hat, startet er den *Kernel Gerätetreiber* und wartet, bis sich *Clients* (s.u.) mit ihm verbinden.

Der *Kernel Treiber* ist ein normaler Gerätetreiber auf Kernelebene, der die Masse der DMA Operationen ausführt und verschiedene Schnittstellen für die Synchronisation bereitstellt.

Eine Bemerkung: Obwohl der Treiber von seiner Funktion her hardwareabhängig ist, kann eine aktuelle Implementierung vielleicht - wie allgemein gebräuchlich - so sein, dass alle Hardware spezifischen Details in die libH3D Bibliothek abstrahiert werden und diese dann beim initialisieren der DRI in den Kernel Treiber geladen wird. Eine Implementierung dieser Art wäre wünschenswert, da man den Kernel Treiber dann nicht für jede neue Grafikkarte neu Updaten müsste. (Laut der offiziellen DRI-Seite sind diese Überlegungen im Gange, an dieser Stelle aber nicht von entscheidender Wichtigkeit.)

*Clients* können irgendwelche Anwendungsprogramme mit grafischer Oberfläche sein, in unserem Fall eben 3D Anwendungen. Der Client ist ein sogenannter X11 Client, der auch mit verschiedenen Bibliotheken verlinkt ist, um hardwareabhängiges direct rendering zu unterstützen:

libGL – das ist die Standard OpenGL (oder Mesa) Bibliothek, mit der Geräte- und Betriebssystemunabhängigen Beschleunigung und den GLX Modifikationen (Erklärung später).

libDRI – Grafikkarten- und Betriebssystemabhängiger Treiber

libHW3D – Grafikkartenabhängiger Treiber.

Wie man Abbildung 4.2a entnehmen kann, müssen, um den Daten- und Kontrollfluss zwischen diesen Komponenten zu gewährleisten, verschiedenen Kommunikationsprotokolle geöffnet werden.

- PROTO ist die Standard X Protokoll Transportschicht.
- SAREA (shared memory area) ist ein spezieller Speicherbereich. Dieser Bereich wird vorwiegend dazu genutzt, um Informationen zwischen X Server und Client austauschen zu können. Aber auch der Kernel hat zugriff auf diesen Bereich und kann z.B. verschiedene Status Informationen abfragen und manipulieren.
- MMIO (Memory Mapped Input Output), DMA (Direct Memory Access) sind die Möglichkeiten, direkt mit der Grafikkarte zu kommunizieren. Mit MMIO kann man etwas mehr erreichen, als nur Rendering - Daten an die Grafikkarte zu schicken. Man kann die Karte programmieren, indem man verschiedene Register setzt, die als

Speicherbereiche Codiert sind. DMA ist ein fest definierter Speicherbereich, der als Schnittstelle zwischen Grafikkarte und Treiber dient. Die Daten, die hier abgelegt werden, werden von der Grafikkarte direkt aufgenommen und verarbeitet. Diese Methode wird hier vorwiegend verwendet, auch wenn sie nicht mehr ganz zeitgemäß ist. Es ist aber zu jeder Grafikkarte kompatibel und deswegen vorerst einfacher zu implementieren.

- DMA BUFFERS sind Speicherbereiche, die genutzt werden, um Grafikkarten Befehle zu puffern, welche über den DMA zur Grafikkarte geschickt werden. Diese Bereiche werden nicht gebraucht, falls MMIO genutzt wird.

### 4.3.2. Initialisierung

Bis jetzt haben wir gesehen, was es für verschiedene Komponenten gibt und durch welche Transportprotokolle sie miteinander kommunizieren. Im folgenden Abschnitt werden die Initialisierung und die damit verbundenen Analysen beschrieben.

Zu diesem Zeitpunkt finden alle notwendigen Operationen statt, um das System startklar zu machen. Der X Server findet heraus, was ihm hardwaretechnisch zur Verfügung steht und weist den verschiedenen Quellen Speicherbereiche zu.

### X Server Initialisierung

Wenn der X Server gestartet wird und das GLX Modul geladen wird, müssen verschiedene Ressourcen im X Server und im Kernel initialisiert werden. GLX: Siehe Glossar!

Der X Server kann mit der 3D Grafikkarte offensichtlich nichts anfangen, bevor das GLX Modul geladen ist. Dieses muss in der XFree86 Konfigurationsdatei angegeben sein. Wenn dieses Modul dann geladen ist – es beinhaltet auch das XFree86-GLX Protokoll mit den Dekodierungs- und Ereignisbehandlungsroutinen – wird das Grafikkartenunabhängige DRI Modul ebenfalls geladen. Dieses DRI Modul wird das Grafikkartenabhängige Modul starten, um die weiter unten beschriebene Zuweisung der Quellen durchzuführen.

Das Grafikkartenabhängige Modul beinhaltet den 2D und 3D Initialisierungscode.

Was passiert nun im Detail? Verschiedene globale X Ressourcen müssen zugewiesen werden, um die 3D Anfragen des DRI Client behandeln zu können. Diese Quellen beinhalten den Frame Buffer, Texture Memory, weitere eventuell benötigte Puffer, den Platz für die Display list und SAREA.

Der **Frame Buffer** wurde ja schon bei der Funktionsweise von OpenGL erklärt. Dessen Speicherbereich wird beim initialisieren statisch zugewiesen. Wenn die Grafikkarte z.B. Vorder-, Rück- und Z-Puffer im Frame Buffer unterstützt, dann wird der Frame Buffer in 4 Teile aufgeteilt. Die ersten drei Teile sind gleich der Größe der Auflösung des sichtbaren Bereichs und werden für die drei genannten Puffer verwendet. Der verbleibende Teil des Frame Buffers wird dann nicht zugewiesen und bleibt für weitere Dinge wie Hardware Cursor, Texturen, ... frei. Bei dieser Methode wird also, wenn Clients Fenster anlegen, das ganze Feld an verfügbaren Puffern vorreserviert

**Der Speicherplatz für Texturen** wird von allen 3D rendering Clients geteilt – normalerweise auf dem Speicher der Grafikkarte. Hier gibt es entweder spezielle Speicherbereiche für Texturen, die eventuell mit anderen Ressourcen geteilt werden können, oder es gibt einen Speicherbereich, den sich Texturen und Puffer teilen müssen.

Beim initialisieren müssen die jeweiligen Größen herausgefunden werden, was nicht immer ganz einfach ist und noch nicht durchgängig sauber gelöst ist.

(Da Speicher ein begrenztes Mittel ist, wäre es am besten, wenn es einen Mechanismus gäbe, der den Speicher begrenzt, der für Texturen vorgesehen ist. Das ist aber nicht der Fall und zusätzlich wird die Speicherorganisation auf verschiedenen Grafikkarten unterschiedlich gehandhabt, meist anders als die lineare Adressierung der Puffer. Die Größe des Texturspeichers ist geräteabhängig unterschiedlich groß.)

Anschließend kann je nach **verbleibendem Speicherplatz** der bis dahin noch ungenutzte Speicherbereich für weitere (nicht zwingend notwendige) Puffer genutzt werden. Manche können sind allerdings nur Geräteabhängig genutzt werden. Dieser Platz wird bei der X Server Initialisierung berechnet.

**DISPLAY LISTS** werden bei Grafikkarten, die das unterstützen gleich behandelt, wie Texturspeicher. Sie sind normalerweise im Speicher der Grafikkarte. Ansonsten befinden sie sich im virtuellen Adressraum des Client. In Display Lists kann man verschiedene 3D Modelle setzen, die man z.B. oft benötigt. Das hat einen entscheidenden Zeitvorteil, da man sich dann nur auf die Display List beziehen muss und nicht wieder das Modell an die Grafikkarte schicken muss.

Nach den Display Lists wird der **SAREA** Platz zugewiesen, und somit sind die Speicherzuweisungen abgeschlossen.

Jetzt wird noch herausgefunden, welche Hardwaremethode des Double Bufferings zur Verfügung steht. Hier gibt es verschiedene Methoden, deren wesentlicher Unterschied in der Geschwindigkeit liegt. Welche gewählt wird ist Hardwareabhängig, aber das ist an dieser Stelle nicht so wichtig.

## **Kernel Treiber Initialisierung**

Wenn der X Server auf den Kernel Treiber zugreift, muss dieser eventuell erst noch geladen werden. Falls das der Fall ist, wird das Modul vom kerneld initialisiert und die Initialisierungsroutine wird gestartet. In jedem Fall wird die „open Routine“ dann aufgerufen und beendet die Initialisierung.

Hier müssen wieder verschiedene Dinge geprüft und zugewiesen werden. Im wesentlichen werden die *DMA Puffer* zugewiesen und die Interrupt Behandlung initialisiert.

Die DMA Puffer werden für die Kommunikation mit der Grafikkarte benötigt. Diese Puffer werden dann dem direct rendering Client zur Verfügung gestellt. Zusätzlich gibt es noch die *Client DMA Warteschlangen*. Jeder direct rendering Kontext (also jedes Fenster in dem gerendert wird) benötigt eine Warteschlange, die vom DMA Puffer abgearbeitet werden kann. Diese Warteschlangen werden vom X Server zugewiesen, wenn `glXCreateContext()` aufgerufen wird, aber der Kernel muss sie verwalten.

Dann muss noch die *Interrupt Behandlung* initialisiert werden. Nachdem ein DMA Puffer von der Grafikkarte Verarbeitet wurde, setzt die Grafikkarte einen Interrupt. Um diesen behandeln zu können, muss der Treiber wissen, welches Register er mit welchem Wert setzen muss. Hier sollte es am besten ein Interface geben, dass die Information mit dem X Server teilt, da der X Server dem Kernel dann mitteilen kann, wie er auf die Interrupts reagieren soll.

## Client Initialisierung

Die Grundlage für die Initialisierung des direct rendering Client ist, dass die GL/GLX Bibliothek geladen ist. Dann werden verschiedene Abfragen an den X Server gemacht, die z.B. feststellen, ob direct rendering überhaupt zur Verfügung steht. Dann können GLX Kontexte erstellt werden. Ein GLX Kontext ist z.B. ein Fenster, in dem gerendert werden soll. Man kann aber auch in einem Fenster mehrere GLX Kontexte haben. Ein Beispiel dafür wären z.B. verschiedene Sichten aus einem Flugsimulator heraus: Sicht aus dem Cockpit, Sicht vom Flügel, Sicht nach hinten. Wenn das alles in einem Fenster stattfindet, dann hätte man in diesem Fenster 3 GLX Kontexte, die allerdings nicht parallel, sondern nur sequentiell abgearbeitet werden können. Das kommt aber nicht allzu oft vor.

Die Reihenfolge der Initialisierung setzt voraus, dass der X Server und der Kernel Gerätetreiber bis dahin schon geladen sind. Einige von den Client Konfigurationsanfragen beziehen sich auf Verschiedene Daten der GLX, beispielsweise Version, Erweiterungen, weitere wichtige fragen den X Server, was die Grafikkarte bietet, z.B. wie viele Farben verwendet werden können, was für zusätzliche Puffer (zu den nicht grundsätzlich notwendigen) vorhanden sind, was für ein Standardtransportprotokoll benützt wird (Unix Domain oder TCP/IP sockets).

Es muss mindestens ein GLX Kontext existieren. Ist ja auch klar, weil wo soll sonst gerendert werden. Es können aber natürlich auch mehrere vorhanden sein. Beim Erzeugen des ersten passiert folgendes: die DRI Bibliothek initialisiert die direct rendering Schnittstelle für den Client. Diese wird im X Server und im Client geladen und initialisiert. Dabei baut sie einen privaten Kommunikationsmechanismus, das XFree86-GLX Protokoll zwischen X Server und Client auf. Der X Server schickt dem Client die SAREA Segment ID über dieses Protokoll zu und der Client fügt dies dann hinzu. Als nächstes schickt der X Server dem Client auch per XFree86-GLX Protokoll den Namen des Geräteabhängigen 3D Grafikkartentreibermodul, welches dann vom Client geladen und initialisiert wird. Der X Server schickt dem Kernel Modul einen Aufruf, eine DMA Warteschlange zu erzeugen und den GLX Kontext zu behandeln.

Fazit: Es werden also erst mal die grundlegenden Strukturen (Speicherbereiche, Puffer, ...) vom X Server und Kernel initialisiert und bereitgestellt. Der Client aktiviert und verbindet dann verschiedene Bibliotheken und Protokolle zwischen sich, vor allem dem X Server und auch dem Kernel, stellt die benötigten 3D Schnittstellen her und somit erreicht unser Initialisierungsvorgang den Bereitschaftsstatus, der nun endlich 3D direct rendering zulässt.

### 4.3.3. Der Bereitschaftsstatus

Nachdem der X Server gestartet, Kernel Treiber Module geladen sind und sich ein oder mehrere 3D Clients eingeklinkt haben, können verschiedene Situationen auftreten.

#### **Ein einzelner 3D Client (1 GLXContext, 1 GLXWindow), X Server inaktiv**

Annahme: Es gibt keine X Server Aktivität. Das ist der Optimalfall, denn das primäre Ziel ist ja, Grafikkarten spezifische Befehle zu erzeugen, und diese so schnell wie möglich in den DMA Puffer zu setzen.

*Laufende Rendering Anfragen* können im einfachsten Fall per DMA Puffer direkt an die Grafikkarte geschickt werden. Wenn der Puffer voll ist und von der Grafikkarte verarbeitet werden soll, kann er sofort vom Kernel behandelt werden.

Verschiedene Dinge noch beachtet werden:

Es kann unter Umständen ein *Software Fallback* benötigt werden. Nicht jede OpenGL Grafikprimitive kann von jeder Hardware beschleunigt werden. Deswegen stellen Mesa und OpenGL einen Mechanismus bereit, um diese Software Fallbacks zu realisieren.

Falls indirect rendering benötigt wird, stellt GLX Befehle bereit, die direct rendering und indirect rendering *synchronisieren* (glFlush).

Die wesentlichen Mechanismen für das Rendering müssen im Client und im Kernel implementiert sein, dass direktes Hardware Rendern funktioniert. Deswegen muss man dem Client auch ein gewisses Maß an „Low Level“ Operationen zugestehen, da er ja verschiedene Gebiete des X Servers übernehmen muss, um ein Leistungsfähiges direct rendering zu ermöglichen. Gute Hilfsmittel dafür sind GLX und die SAREA, die es ermöglichen, die rendering Anfragen zu managen und die benötigten Informationen schnell zwischen den Komponenten zu teilen.

### **Weitere Situationen**

Weitere Situationen können sein, dass der X Server zeichnen kann (z.B. 2D rendering in anderen Fenstern), aber er kann das 3D Fenster nicht bewegen. Dies ist ein allgemeiner Fall. Der große Unterschied zum vorigen Fall besteht darin, dass der Hardware Lock geschickt gesetzt werden muss, um zu verhindern, dass der X Server verschiedene 3D Anfragen software rendert, aber andererseits muss man sich ja auch um Software Fallbacks, also Anfragen kümmern, die die Hardware nicht rendern kann.

### **4.3.4. Prozess Abschließung**

In diesem Abschnitt wird beschrieben, was passiert, wenn ein 3D rendering Prozess terminiert wurde. Wenn die zu zeichnende Oberfläche ein Fenster ist, kann es vom Window Manager zerstört werden. Falls dies geschieht, muss der X Server auf jeden Fall registrieren, dass das Fenster zerstört wurde. Der X Server muss vor dem endgültigen schließen auf jeden Fall noch warten, bis alle außenstehenden DMA Puffer anfragen, die mit diesem Fenster in Verbindung stehen, verarbeitet wurden (die liegen ja schon auf der Grafikkarte!). Wenn der Client noch einmal versucht, auf dieses Fenster zu zeichnen, dann stellt er fest, dass es nicht mehr gültig ist, gibt seinen internen Status, der mit diesem Fenster verbunden ist auf und gibt einen Fehler zurück.

Dann müssen eventuell noch verschiedene Zustände zurückgesetzt werden, z.B. wenn ein GLX Kontext noch aktuell ist, Referenzen müssen gelöscht werden und die rendering Quellen, die für den Client genutzt wurden müssen freigegeben werden.

Im schlimmsten Fall stellt man fest, dass ein Client tot ist, also irgendwie unerwartet beendet wurde. Hier kann der Fall sein, dass der Client durch einen Kill Befehl abgeschossen wurde, oder der Programmierer einfach nicht sauber programmiert hat und nicht alle Ressourcen freigegeben hat. Falls der Client noch einen Hardware Lock gesetzt hat, muss dieser vom Kernel entfernt werden. Hierbei kann es passieren, dass die Grafikkarte in einen unbrauchbaren Zustand gerät und einen Reset benötigt. Nach diesem Reset kann der Prozess dann eventuell normal beendet werden.

Das war das „Leben“ eines 3D rendering Client Prozesses.

## 5. Glossar

### **GLX:**

GLX verbindet OpenGL und das X Window System<sup>TM</sup>. OpenGL ist ein reines 3D API. Unter Linux und Unix wird GLX verwendet, um OpenGL in die darunter liegende graphische Oberfläche des Systems einzubetten (z.B. Fenster zum rendern anzulegen). Das bedeutet für das rendern in einem Fenster, dass eventuell die gerenderten OpenGL Koordinaten einer Szene auf die Koordinaten eines Fensters umgerechnet werden müssen und auch eventuelle Größenänderungen des Fensters (also Streckungsfaktoren) berücksichtigt werden müssen. Eventuell kann sogar die Auflösung in diesem Fenster eine andere sein, als auf den restlichen Fenstern des X-Server. Auch Spiele, die unter Windows laufen werden in einem Fenster gerendert. Man sieht nur den Rahmen nicht. Alte 3Dfx Voodoo Karten unterstützen nur Vollbildrendering, aber das ist inzwischen überholt und z.B. im Bildbearbeitungsbereich oder CAD nicht sehr sinnvoll.

Daher wird GLX von jeder OpenGL Implementierung in Zusammenhang mit X benötigt (also auch z.B. von Mesa). SGI bringt seine GLX implementierung unter einer öffentlichen Lizenz heraus. Das bedeutet, dass, zusammen mit einigem zusätzlichem Arbeitsaufwand, welcher von Precision Insight Inc. getätigt wird, die Reichweite von OpenGL in eine neue Ebene von Plattformen ausgeweitet wird: Auf Linux, BSD und andere Betriebssysteme, welche die bekannte XFree86 Implementierung von X benutzen. Die GLX selber ist auf Entwickler abgezielt, welche mit XFree86 arbeiten. Diese Arbeit sollte zu Hardware beschleunigten 3D Treibern für XFree86 führen (GLX aus einem Dokument von 1998!!!)

### **GLUT**

Ist ein OpenGL Utility Toolkit, ein Window System unabhängiges Entwicklungshilfsmittel, mit dem man OpenGL Programme schreiben kann. Ist hauptsächlich für kleinere bis mittelgroße Programme gedacht. Man kann damit gut den Umgang mit OpenGL lernen.

### **Direct / indirect rendering**

Direct rendering ist der Vorgang, 3D Daten Hardwarebeschleunigt als 2D Bild auf den Bildschirm zu bringen (das entscheidende ist, dass die Grafikspezifischen Berechnungen vom Grafikprozessor (auf der Grafikkarte) abgenommen werden).

Indirect rendering oder auch software rendering läuft über die CPU-Leistung des Computers und wird erst im letzten Arbeitsschritt an die Grafikkarte geschickt.

Wenn man einen beliebig schnellen Prozessor hätte, dann wäre indirect rendering vermutlich nicht langsamer als direct rendering. Da dies aber nicht der Fall ist, muss die Prozessorleistung für das Ausführen des eigentlichen Programms und die aufwändige Berechnung der 3D Grafiken geteilt werden, was bei komplexen Grafiken normalerweise nur mit einer einbuße der Geschwindigkeit (ruckeln) einhergeht.

### **Hardware Lock**

Nur ein Prozess kann gleichzeitig auf die Grafikkarte zugreifen. Deswegen kann man einen Hardware Lock setzen, damit, falls mehrer Prozesse auf die Grafikkarte zugreifen wollen, kein Unglück passiert und plötzlich wildes Bildchaos entsteht, wenn jeder unkoordiniert ins Bild zeichnet.