



Marius
Apetri

3D-Grafik mit **OpenGL**

Das umfassende Praxis-Handbuch



inklusive CD-ROM

Teil I

Grundlagen der OpenGL- Programmierung

In diesem Teil:

- **Kapitel 1**
Programmsoftware und Grafikhardware 27
- **Kapitel 2**
Darstellung von Punkten 39
- **Kapitel 3**
Globale Zustände 71
- **Kapitel 4**
Darstellung von Linien 81
- **Kapitel 5**
Der Auswertungsmechanismus von OPENGL 111

Der erste Teil des Buches gibt einen ausführlichen Einblick in die Hauptmerkmale des OPENGL-Standards, wie beispielsweise die Verwendung globaler Zustände, die gepufferte Ausgabe, das Fehlererkennungssystem oder die verschiedenen OPENGL-Implementierungen.

Das zweite Hauptthema sieht die Diskussion der Richtlinien vor, die bei der Erstellung von OPENGL-Programmen zu berücksichtigen sind; in diesem Zusammenhang werden die Benennung von Funktionen und die Verwendung von OPENGL-Datentypen beschrieben. Der praktische Einsatz dieser Vorgaben erfolgt anhand zahlreicher Beispielprogramme mit zweidimensionalen Ausgaben zur Darstellung von Punkten, Linien und Annäherungen an Bézierkurven.

Programmsoftware und Grafikhardware

Die Idee, die uns umgebende dreidimensionale Welt auf einem Computer nachzuahmen, wurde bereits in den sechziger Jahren ausführlich diskutiert, als die modernsten Rechner noch ganze Räume ausfüllten und nur wenigen Personen zur Verfügung standen. Algorithmen, mit deren Hilfe künstliche dreidimensionale Umgebungen sich auf eine fotorealistische Weise generieren lassen, unter anderem Polygonschattierung, Texturprojektion oder die Modellierung von Semitransparenz- und Reflexionsdarstellungen, wurden bereits in dieser frühen Zeit entwickelt.

Die geringe Leistungsfähigkeit der verfügbaren Computersysteme erlaubte die breite Anwendung dieser spektakulären, jedoch rechenintensiven Grafikeffekte allerdings über einen längeren Zeitraum nicht. Stattdessen griff man bei der Programmierung zwei- und dreidimensionaler Welten zunächst auf wesentlich einfachere Algorithmen zurück, die im weiteren Verlauf immer komplexer werden konnten und, in enger Anlehnung an den technischen Fortschritt, bessere Annäherungen an die reale Welt zu generieren vermochten. Die Grundstruktur dieser Programme blieb jedoch unverändert.

1.1 Parallele Durchführung von Berechnungen

Die Darstellung einer künstlichen Umgebung lässt sich im Allgemeinen in zwei klar voneinander getrennten Phasen unterteilen. In der ersten Phase werden die Eingaben des Benutzers entgegengenommen, bei Bedarf erfolgt die Bewegung bestimmter virtueller Gegenstände, und die im aktuellen Bildaufbau sichtbaren Polygone werden ausgewählt. Die eigentliche Darstellung der ausgewählten Polygone auf dem Bildschirm erfolgt erst während der zweiten Phase; diese ist aufgrund der beschriebenen komplexen Visualisierungsverfahren im Allgemeinen um ein Vielfaches rechenaufwendiger als die erste.

Die Berechnungen, die während des Darstellungsvorganges virtueller Welten auftraten, wurden zunächst vollständig von dem **Hauptprozessor** durchgeführt. Mit steigender Leistung der verfügbaren CPUs wurden auch die Ausgaben immer realistischer; doch trotz dieser rasanten Entwicklung erkannte man bereits sehr früh, dass der Rechenaufwand, den die Echtzeitdarstellung einer fotorealistischen dreidimensionalen Umgebung erfordert, viel zu groß ist, um in naher Zukunft von dem Hauptprozessor allein bewältigt werden zu können. Diese Tatsache gilt selbst für die modernsten Multicore-CPU's mit Geschwindigkeiten von mehreren Gigahertz.

Arbeitsteilung lautet die Lösung dieses Problems: Während der Hauptprozessor weiterhin die erste Phase der Berechnung des aktuellen Bildaufbaus übernimmt, weist man die aufwendigen, in der zweiten Phase durchzuführenden Berechnungen einem zusätzlichen, prozessorähnlichen

Baustein zu, dem so genannten **Grafikprozessor**; im Gegensatz zur CPU, die auf der Hauptplatine des Computers angebracht ist, befindet dieser sich auf der **Grafikkarte**.

Die vom Grafikprozessor durchzuführenden Berechnungen werden so ausgewählt, dass diese **parallel**, das heißt **zur selben Zeit** wie die Berechnungen erfolgen können, die nach wie vor in den Aufgabenbereich des Hauptprozessors übertragen werden: Während der Grafikprozessor die zweite Phase des aktuellen Bildaufbaus verarbeitet, kann die CPU bereits die in der ersten Stufe des jeweils nächsten Frames auftretenden Berechnungen durchführen. Der entscheidende Geschwindigkeitsvorteil, der selbst die Visualisierung hochkomplexer Umgebungen ermöglicht, ist auf diese Halbierung des gesamten Rechenaufwandes zurückzuführen.

1.1.1 Der nicht programmierbare Grafikprozessor

Während die Aufgabe der CPU darin besteht, eine möglichst große Anzahl voneinander unterschiedlicher Aufgaben zu bewältigen, übernimmt der nicht programmierbare Grafikprozessor grundsätzlich nur sehr spezielle Aufgaben: entweder einen Punkt setzen oder eine Linie zeichnen oder ein Polygon darstellen. Die Erweiterungsmöglichkeiten dieser Aufgaben, wie die benutzerdefinierte Einstellung der Größe des Punktes oder die Projektion einer Textur auf die Oberfläche des Polygons sind ebenfalls sehr beschränkt.

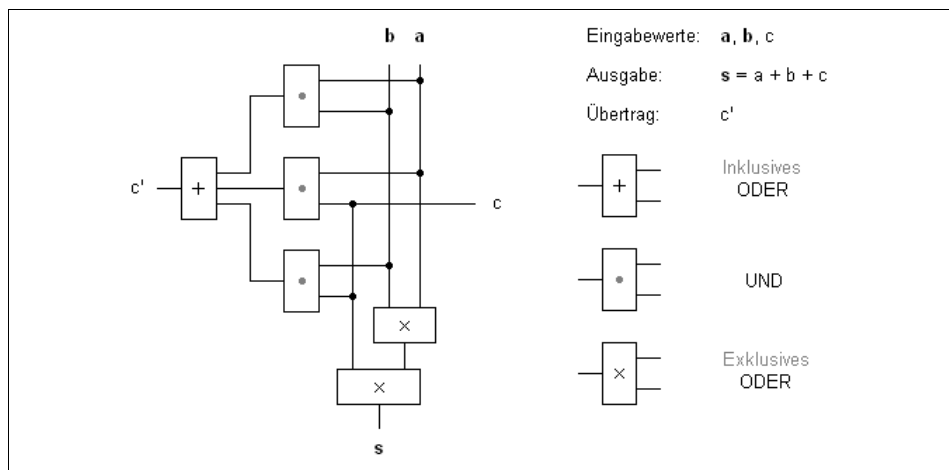


Abb. 1.1: Schematische Darstellung einer Schaltung, die zwei Bitwerte **a** und **b** unter Berücksichtigung des aus der vorherigen Berechnung stammenden Übertrags **c** zusammenaddiert

Es ist daher naheliegend, für jede dieser Aufgaben eine eigene **Schaltung** zu entwerfen, die aus einer entsprechenden Verkettung von Bausteinen besteht, die die einfachen logischen Operationen UND, ODER und NICHT durchführen. Der Grafikprozessor ist somit nichts anderes als die Zusammenfassung einer großen Anzahl voneinander getrennter Schaltungen, wobei jede eine bestimmte Aufgabe übernimmt: Es gibt beispielsweise eine Schaltung zur Darstellung eines einfach gefärbten Polygons, eine andere, die ein texturiertes Polygon zeichnen kann, usw.

Der große Vorteil spezialisierter Schaltungen besteht darin, dass diese ihre entsprechenden Berechnungen in einer wesentlich kürzeren Zeit durchzuführen vermögen als die für breitgefä-

cherte Aufgaben konzipierte CPU. Ein weiterer Vorteil ist, dass komplexe Schaltungen mit wenig Aufwand aus einfacheren zusammengestellt werden können – Abbildung 1.2 zeigt beispielsweise den schematischen Aufbau eines Ganzzahlenaddierers, der aus der viermaligen Parallelausführung der in Abbildung 1.1 dargestellten Schaltung besteht.

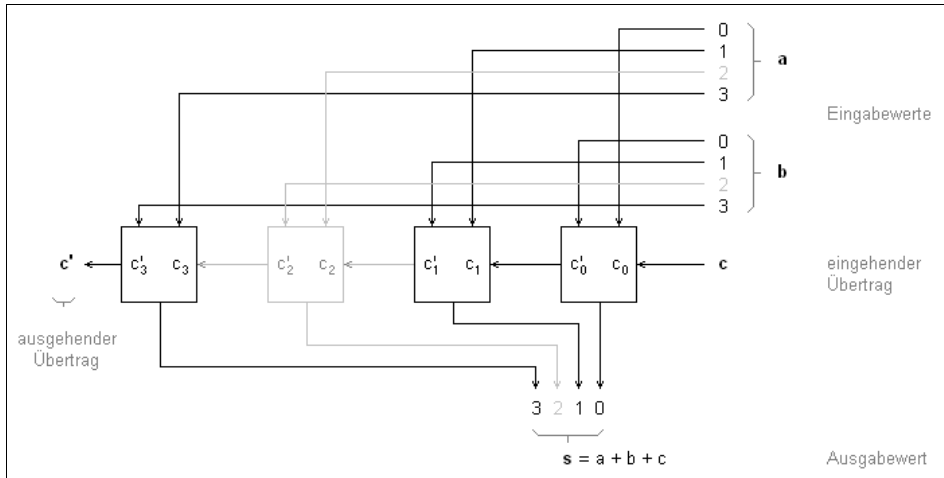


Abb. 1.2: Eine Schaltung zur Addition von zwei Zahlen mit einer Breite von jeweils vier Bit lässt sich durch Hintereinanderschaltung mehrerer einfacherer Additionsbausteine zusammenstellen

Schaltungen, die wesentlich komplexere Aufgaben wie Division, Wurzelberechnung mit Fließkommazahlen oder rechenintensive trigonometrische Funktionen berechnen können, lassen sich grundsätzlich nach demselben Prinzip zusammenstellen. Da jedem Grafikeffekt eine mathematische Formel zugrunde liegt, die ihrerseits nichts anderes als eine bestimmte Verkettung der beschriebenen Operationen darstellt, sind die im Grafikprozessor enthaltenen Schaltungen nichts anderes als wesentlich komplexere Versionen des in Abbildung 1.2 vorgestellten Schaltbildes.

1.2 Der OpenGL-Standard

Die Implementierung eines theoretisch formulierten Algorithmus in Form einer Schaltung lässt sich auf viele verschiedene Arten durchführen – die Schaltungen unterschiedlicher Grafikkartenhersteller, die zur Realisierung eines bestimmten Effektes entworfen worden sind, können sich demnach teilweise sehr stark voneinander unterscheiden.

Dieser Umstand stellt den Programmierer vor ein großes Problem: Verschiedene Schaltungen werden in unterschiedlicher Form von der Software angesprochen – aus diesem Grund müssten mehrere verschiedene Versionen desselben Programms erstellt werden, wobei jede Version die Grafikkarte eines bestimmten Herstellers unterstützt. Hinzu kommt, dass jede Hardwareänderung seitens des Herstellers die Bereitstellung eines entsprechenden Softwareupdates erforderlich machen würde. Dieser Aufwand ist jedoch bei Weitem zu groß.

Gesucht ist somit ein Standard-Befehlssatz, der es dem Programmierer erlaubt, mit **demselben** Befehl die Schaltungen sämtlicher Hersteller anzusprechen, die einen bestimmten Grafikeffekt hervorrufen. Heute sind mehrere Standards vorhanden, die diese Aufgabe erfüllen – der zweifellos bedeutendste unter ihnen ist die OPEN GRAPHICS LIBRARY, kurz OpenGL; dieser besitzt mehrere Vorteile, die die anderen Standards nicht aufweisen können.

Bei der Entstehung und Weiterentwicklung von OpenGL sind sowohl die großen Grafikkartenhersteller als auch namhafte Programmierer maßgeblich beteiligt, wodurch lückenlose Unterstützung und einwandfreie Funktionsfähigkeit seitens der Hardware und große Übersichtlichkeit und Programmierkomfort seitens der Software gewährleistet sind. Ein weiterer Vorteil ist, dass die OpenGL-Befehle mit mehreren weitläufig bekannten Programmiersprachen wie C++, JAVA oder DELPHI aufgerufen werden können.

Das wichtigste Alleinstellungsmerkmal ist jedoch die Plattformunabhängigkeit – ein Quelltext, der auf OpenGL zurückgreift, lässt sich in unveränderter Form auf unterschiedlichen Betriebssystemen wie WINDOWS, LINUX, MAC usw. kompilieren, wodurch Verbreitungsgrad und Popularität eines Programms stark vergrößert werden können.

1.2.1 Das Grundprinzip des OpenGL-Standards

Der OpenGL-Standard besteht lediglich aus der Zusammenfassung mehrerer Funktionsdeklarationen ohne Definition bzw. einer Ansammlung schlichter Funktionsnamen, zusammen mit der detaillierten Beschreibung der Arbeitsweise dieser Funktionen, einfachen Richtlinien bezüglich ihres Einsatzes und Programmierbeispielen. Die eigentliche Definition dieser Funktionen sowie die Art, wie die entsprechenden Schaltungen anzusprechen sind, bleibt hingegen den Grafikkartenherstellern überlassen und ist für den Programmierer irrelevant. So kommt es, dass viele verschiedene Implementierungen des OpenGL-Standards vorhanden sind.

Die Definition dieser Schnittstelle zwischen Programmierer und Grafikhardware wurde unter Berücksichtigung der beiden folgenden Vorgaben konzipiert: Einerseits müssen die Funktionen möglichst hardwarenah sein, um eine schnelle Ausführungsgeschwindigkeit der Programme zu ermöglichen, andererseits sind diese aber auch so allgemein wie möglich zu halten, um die Übersichtlichkeit der Quelltexte und die Unterstützung verschiedener Programmierstile zu gewährleisten und den Grafikkartenherstellern möglichst große Freiheit bei der Entwicklung der Schaltungen zu geben.

1.2.2 Richtlinien für die Benennung von Funktionen

Eine der Richtlinien des OpenGL-Standards sieht die Berücksichtigung mehrerer globaler Zustände bei der Darstellung von Gegenständen vor. Bei dem ersten im Verlauf dieses Buches vorgestellten Zustand handelt es sich um die **aktuelle Farbe**: Einmal festgelegt, nehmen sämtliche Punkte, Linien oder Polygone automatisch diese Farbe an, bis dieser Zustand von Neuem verändert wird.

Um die Verwendung von OpenGL möglichst angenehm zu gestalten, kann der Programmierer bei der Erfüllung einer bestimmten Aufgabe auf eine von mehreren Funktionen zurückgreifen, die alle denselben Zweck erfüllen. Die Namen dieser eng miteinander verwandten Funktionen werden nach einer festen Gesetzmäßigkeit gebildet: Zuerst erscheint der Name der gesamten Gruppe, gefolgt von der Anzahl der Parameter und schließlich von einem Suffix, das den gemeinsamen Datentyp der entgegengenommenen Werte angibt.

```
void glColor3 < b, ub, s, us, i, ui, f, d >
(
    GLtype red, GLtype green, GLtype blue
);
```

Diese Funktionen legen die aktuelle Farbe fest, die in Form der Komponenten `red`, `green` und `blue` definiert ist. Sämtliche im Anschluss gezeichneten Gegenstände nehmen automatisch diese Farbe an.

Diese Funktion legt den Wert der Zustandsvariablen `GL_CURRENT_COLOR` fest.

Abfrage des globalen Zustandes:	<code>glGetFloatv()</code>
vordefinierter Wert:	<code>(1, 1, 1, 1)</code>
Kennzeichen der Attributgruppe:	<code>GL_CURRENT_BIT</code>

Die in den spitzen Klammern angegebenen Ausdrücke `ub` und `d` geben beispielsweise an, dass die Funktionen `glColor3ub()` bzw. `glColor3d()` jeweils drei Variablen vom Typ `GLubyte` bzw. `GLdouble` entgegennehmen. `GLtype` ist lediglich ein Platzhalter bzw. eine formelle Bezeichnung, die auf die Typgleichheit der Parameter hinweist. Die vollständigen Prototypen dieser beiden Funktionen sind somit:

```
void glColor3ub( GLubyte r, GLubyte g, GLubyte b );
void glColor3d( GLdouble r, GLdouble g, GLdouble b );
```

Die eigentliche Darstellung von Objekten erfolgt durch die Angabe der räumlichen Positionen der Punkte, die diese Gegenstände definieren. Die Übermittlung der Koordinaten eines Punktes an die Hardware erfolgt durch den Aufruf einer Funktion aus der Gruppe `glVertex*()` – im Fall des zweidimensionalen Raumes sind diese Funktionen wie folgt deklariert:

```
void glVertex2 < s, i, f, d >
(
    GLtype sx, GLtype sy
);
```

Diese Funktionen übermitteln die zweidimensionale Position (`sx`, `sy`) an die Grafikhardware.

Sämtliche Elemente der Funktionsgruppe `glVertex*()` dürfen lediglich innerhalb eines Quelltextblocks aufgerufen werden, der von `glBegin()` und `glEnd()` begrenzt wird.

1.2.3 OpenGL-Datentypen

Wie bereits beschrieben, sieht das Grundkonzept des OPENGL-Standards vor, den Hardwareherstellern möglichst großen Freiraum bei der Implementierung der vorgegebenen Funktionen einzuräumen; auf diese Weise kann eine sehr breite Palette an Geschwindigkeitsoptimierungen in die Praxis umgesetzt werden.

Ein wesentlicher Bestandteil dieser Funktionen sind die **Datentypen** der Variablen, mit denen sie arbeiten, genauer die Größe des jeweils belegten Speicherplatzes. Je mehr Speicherplatz eine Variable in Anspruch nimmt, umso länger dauert es, bis das Ergebnis einer aufwendigen, mit dieser Variablen durchgeführten mathematischen Operation vorliegt. Die genaue Festlegung der Größe des Arbeitsspeichers, den jeder OpenGL-Datentyp in Anspruch nimmt, ist daher den Grafikkartenherstellern überlassen.

Die in der nachfolgenden Tabelle aufgeführten Größen dienen somit lediglich der Orientierung und sind daher nicht bindend: Während man sich bei den einfachen Datentypen wie `GLbyte` oder `GLshort` noch beinahe sicher sein kann, dass der vorgeschlagene Speicherplatz und damit auch die dazugehörigen Minimal- und Maximalwerte eingehalten werden, unterscheiden sich die verschiedenen OpenGL-Implementierungen hauptsächlich hinsichtlich des Speicherplatzes, der von den beiden Fließkommazahlendatentypen belegt wird.

Suffix	Datentyp	Speicherplatz	Minimaler Wert	Maximaler Wert
b	<code>GLbyte</code>	8 Bit	-128	127
ub	<code>GLubyte</code>	8 Bit	0	255
S	<code>GLshort</code>	16 Bit	-32 768	32 767
us	<code>GLushort</code>	16 Bit	0	65 535
i	<code>GLint</code>	32 Bit	-2 147 483 648	2 147 483 647
ui	<code>GLuint</code>	32 Bit	0	4 294 967 295
f	<code>GLfloat</code>	32 Bit	<i>Abhängig von der eingesetzten OpenGL-Implementierung</i>	
d	<code>GLdouble</code>	64 Bit		

Wichtig ist auch, dass bestimmte Funktionen nicht den gesamten Wertebereich ausnutzen, der ihren Parametern zur Verfügung steht: Die bereits vorgestellte Funktion `glColor3d()` ist beispielsweise nur für Farbkomponenten größer oder gleich 0.0 und kleiner oder gleich 1.0 definiert.

Die beschriebenen Optimierungen können in der Praxis wie folgt aussehen: Wird im Zuge einer beliebigen Implementierung des OpenGL-Standards entschieden, der Ausführungsgeschwindigkeit eines Programms zu Lasten der Genauigkeit der Darstellung den Vorzug zu geben, soll auf die Verwendung des Datentyps `GLdouble` verzichtet werden können:

```
#define GLfloat float
#define GLdouble float
```

Durch diese Definition stellt `GLdouble` nichts anderes als eine zusätzliche Bezeichnung für den Standard-C++-Datentyp `float` dar und belegt somit 32 Bit Speicherplatz. Diese einfache Optimierung kann aber auch versteckt implementiert sein: Übergibt der Benutzer Werte vom Typ `double` an eine Funktion, die mit `GLdouble`-Parametern definiert ist, können diese intern von der Funktion nach `float` konvertiert und an die entsprechende OpenGL-Funktion weitergeleitet werden, die mit `GLfloat`-Werten arbeitet.

Bei der Ausführung bestimmter zeitkritischer Berechnungen können aber selbst `float`-Werte zu viel Speicherplatz belegen; in diesem Fall kann die Hardware auf Fließkommazahlen zugreifen, die 30 oder nur 28 Bit in Anspruch nehmen.

Alternative Bezeichnungen der OpenGL-Standarddatentypen

Nicht alle OPENGL-Funktionen sind unter Verwendung der beschriebenen Datentypen deklariert; in einigen Fällen werden alternative Bezeichnungen eingesetzt, da auf diesem einfachen Weg dem Programmierer hilfreiche Zusatzinformationen hinsichtlich der Arbeitsweise bzw. des Anwendungszwecks der entsprechenden Funktion bereits anhand des Prototyps übermittelt werden können.

Die Alternativbezeichnungen der OPENGL-Datentypen verkürzen nicht zuletzt die Zeitspanne, die zur Lösung einer bestimmten Aufgabe mit einer noch unbekannteren OPENGL-Funktion erforderlich ist. Hierzu ein Vorgriff auf das zehnte Kapitel: Anstelle des Standardparameters `GLdouble` wird `glDepthRange()` beispielsweise unter Verwendung von `GLclampd` deklariert, wodurch gleichzeitig ausgedrückt wird, dass die beiden Parameter dieser Funktion lediglich Werte zwischen 0.0 und 1.0 annehmen sollen.

Hierbei handelt es sich um einen deutlichen Hinweis darauf, dass diese Funktion mit z-Koordinaten arbeitet, die am Ende des geometrischen Verarbeitungsprozesses auf das beschriebene Intervall abgebildet worden sind, und keineswegs mit gültigen dreidimensionalen z-Koordinaten, deren Werte zwischen den benutzerdefinierten Konstanten `z_min` und `z_max` liegen. Die nachfolgende Tabelle enthält eine vollständige Auflistung der alternativen Bezeichnungen der OPENGL-Datentypen:

Datentyp	alternative Bezeichnung	Zusatzinformation, die in Form der alternativen Bezeichnung übermittelt werden soll
GLbyte	GLchar	Bei dem Parameter handelt es sich um ein einzelnes Zeichen vom Typ <code>char</code> oder, im Fall eines Arrays, um eine Zeichenkette.
	GLboolean	Der entsprechende Parameter darf lediglich den Wert <code>GL_TRUE</code> bzw. <code>GL_FALSE</code> annehmen.
GLint	GLsizei	Der Parameter gibt eine <i>Anzahl</i> von Gegenständen an und ist somit ganzzahlig und stets größer oder gleich 0.
GLuint	GLenum	Der Parameter darf lediglich vordefinierte Werte annehmen, deren formelle Bezeichnungen mit dem Präfix <code>GL_</code> gekennzeichnet sind.
	GLbitfield	Einzelne Bits in dem entsprechenden Integerwert dienen der Steuerung bestimmter Mechanismen.
GLfloat	GLclampf	Wird der jeweiligen Fließkommazahl ein Wert kleiner 0.0 zugewiesen, wird dieser automatisch durch die Konstante 0.0 ersetzt; Zahlenwerte größer 1.0 werden nach demselben Prinzip durch 1.0 ausgetauscht.
GLdouble	GLclampd	

1.2.4 Weiterentwicklung des OpenGL-Standards

Moderne Grafikhardware ist in einem immerwährenden Prozess der Weiterentwicklung begriffen. Jeder neue, von einem Hersteller in die von ihm entwickelte Hardware aufgenommene Effekt lässt sich mit Hilfe entsprechender Funktionen nutzen, die im Allgemeinen nach den bereits vorgestellten Prinzipien benannt werden. Um anzugeben, dass diese zusätzliche Funkti-

onalität sich zunächst nur von einer speziellen Grafikhardware ausführen lässt, werden die beschriebenen Funktionsnamen allerdings mit einer zusätzlichen Gruppe von Großbuchstaben markiert, die auf den entsprechenden Hersteller hinweist. Für den Fall, dass die neue Funktionalität lediglich auf einer bestimmten Grafikkarte des Herstellers vorhanden ist, folgt den Großbuchstaben eine in Kleinbuchstaben angegebene Abkürzung des Namens der jeweiligen Grafikkarte.

Entscheiden mehrere Hersteller, den beschriebenen Effekt in ihre Grafikhardware aufzunehmen, wird er dem OpenGL-Standard in Form einer experimentellen Erweiterung hinzugefügt, und die entsprechenden Funktionen werden mit EXT, der Abkürzung für *Extension* gekennzeichnet. Diese Funktionen lassen sich allerdings noch nicht auf allen Grafikkarten ausführen.

Hardwarehersteller kooperieren untereinander, so dass eine von NVIDIA herausgegebene Grafikkarte beispielsweise neben eigenen, mit NV gekennzeichneten Erweiterungen auch Funktionen unterstützt, die ursprünglich auf Grafikkarten von IBM, SUN MICROSYSTEMS, HEWLETT-PACKARD und anderen, erkennbar an Signaturen wie IBM, SUN, HP usw. vorgestellt worden sind.

Informationen bezüglich der aktuellen Grafikhardware, insbesondere eine vollständige Auflistung der jeweils unterstützten, über den regulären OpenGL-Befehlssatz hinausgehenden Erweiterungen, lassen sich über `glGetString()` abfragen. Der Aufruf dieser Funktion darf erst nach erfolgreich durchgeführter Verbindung des aktuellen Programmfensters mit OpenGL erfolgen:

```
const GLubyte *glGetString( GLenum info_signature );
```

Diese Funktion ermöglicht die Abfrage spezifischer Informationen bezüglich der Grafikhardware, die zur Ausführung des aktuellen Programms eingesetzt wird.

Alle möglichen Werte des Parameters `info_signature`:

<code>GL_VENDOR</code>	Name des Grafikkartenherstellers
<code>GL_RENDERER</code>	Name der Grafikkarte
<code>GL_VERSION</code>	Version der OpenGL-Implementierung, die auf der aktuellen Grafikkarte vorhanden ist
<code>GL_EXTENSIONS</code>	Diese Konstante wird zur Abfrage einer vollständigen Liste mit den Signaturen der jeweils unterstützten Zusatzfunktionen eingesetzt. Die Signaturen selbst beinhalten keine Leerzeichen, werden jedoch durch Leerzeichen voneinander getrennt.

Die jeweils angeforderten Informationen werden in Form eines Strings an die aufrufende Instanz übermittelt, dessen Ende mit `'\0'` gekennzeichnet ist. Tritt während der Ausführung von `glGetString()` jedoch ein Fehler auf, besitzt diese Funktion den Rückgabewert `NULL`, und intern wird eine der folgenden Fehlersignaturen generiert, die sich mit `glGetError()` abfragen lässt:

<code>GL_INVALID_ENUM</code>	Dieser Fehler tritt auf, wenn <code>glGetError()</code> mit einem ungültigen Parameter aufgerufen wird.
<code>GL_INVALID_OPERATION</code>	Dieser Fehler wird durch den Aufruf der Funktion innerhalb eines durch <code>glBegin()</code> und <code>glEnd()</code> gebildeten Quelltextblocks generiert.

Hat sich eine Erweiterung bewährt, wird sie von dem OPENGL ARCHITECTURE REVIEW BOARD eingehenderen Untersuchungen unterzogen. Hierbei handelt es sich um eine aus Programmierern, Ingenieuren und Vertretern der Grafikkartenhersteller zusammengesetzte Kommission, deren Aufgabe die reibungslose Weiterentwicklung der Grafikhardware vorsieht. In dieser Testphase, die sich über einen längeren Zeitraum erstrecken kann, lässt sich der Grafikeffekt mit Hilfe von OPENGL-Funktionen mit dem Suffix ARB nutzen, die nunmehr von vielen Grafikkarten unterstützt werden.

Wird die Testphase erfolgreich beendet, findet schließlich die Aufnahme des Grafikeffekts in den regulären OPENGL-Standard statt, und die Benennung der dazugehörigen Funktionen erfolgt mit den ab Seite 30 vorgestellten Standard-Richtlinien. Die Deklaration von Funktionen kann während dieses Vorgangs größere Veränderungen erfahren: `void glUseProgram(GLuint)` ging beispielsweise aus `void glUseProgramObjectARB(GLhandleARB)` hervor. Ab diesem Zeitpunkt sind sämtliche Hardwarehersteller zur Unterstützung des beschriebenen Grafikeffektes verpflichtet.

Die Erweiterung der bestehenden Grafikhardware ist ein aufwendiger Vorgang. Der geschilderte mehrstufige Entwicklungsprozess soll aus diesem Grund sicherstellen, dass die neue Funktionalität eine sinnvolle Ergänzung der bereits vorhandenen Möglichkeiten darstellt – demnach dürfen weder Widersprüche mit der bereits vorhandenen Hardware auftreten, noch soll der entsprechende Effekt auf einfache Weise mit den bestehenden Möglichkeiten generiert werden können. Die Absicht ist, dass die neue Funktionalität für eine möglichst lange Zeit nutzbar bleibt und nicht durch zukünftige Erweiterungen überholt wird.

Die Vorteile einer stabilen Grafikarchitektur überwiegen den Nachteil, der sich aus der langen Testphase ergibt: Eine Funktion, die mittlerweile einen regulären OPENGL-Namen besitzt, kann in der Literatur sowie im Internet in Quelltexten, die zu unterschiedlichen Zeitpunkten erstellt worden sind, noch immer unter Bezeichnungen mit der Signatur ARB bzw. EXT auftreten.

1.3 Grundarchitektur der Grafikkarte

Moderne Grafikkarten besitzen vielfältige Architekturen, die sich teilweise sehr stark voneinander unterscheiden können. Die Befehle des OPENGL-Standards werden daher im weiteren Verlauf des Buches anhand eines einfachen theoretischen Hardwaremodells vorgestellt, das eng an die Bauweise handelsüblicher Grafikkarten angelehnt ist und eine solide Grundlage für sämtliche Rasteralgorithmen darstellt.

1.3.1 Der Videospeicher

Der erste Schritt bei der Darstellung einer virtuellen Umgebung ist die Auswahl einer geeigneten Ausgabefläche; diese kann sich entweder über den gesamten Computerbildschirm erstrecken oder, bei Verwendung eines Multitasking-Betriebssystems, lediglich ein kleines Programmfenster in Anspruch nehmen. Diese Ausgabefläche wird durch zwei Faktoren festgelegt, die eng mit dem Anwendungszweck des jeweiligen Programms verbunden sind: Während die **Auflösung** die Gesamtanzahl der Bildpunkte bzw. Pixel des Ausgabefensters angibt, legt die **Farbtiefe** die Anzahl der gleichzeitig auf dem Bildschirm sichtbaren Farben fest.

Unser theoretisches Modell der Grafikhardware legt fest, dass bei einer Auflösung von 32 Bit die Farbe jedes Pixels direkt mit einer roten, grünen und blauen Komponente anzugeben ist.

Jede dieser Komponenten muss durch einen vorzeichenlosen, ganzzahligen Wert mit einer Breite von 8 Bit bzw. einem Byte angegeben werden. Zusätzlich zu den Farbkomponenten steht jedem Pixel ein weiteres Byte zur Verfügung, **Alphawert** genannt, der unterschiedliche Informationen ausdrücken kann; hierbei handelt es sich zumeist um Semitransparenzangaben. Die entsprechenden Techniken werden ab dem 23. Kapitel ausführlich diskutiert, bis dahin bleibt der Alphawert ungenutzt.

Die vier Komponenten der Farben sämtlicher auf dem Bildschirm sichtbaren Pixel werden in einem ausreichend großen, zusammenhängenden Speicherbereich abgelegt, der sich auf der Grafikkarte befindet und als **Videospeicher** bezeichnet wird. Bei der Auflösung 640×480 Pixel und der beschriebenen Farbtiefe umfasst der Videospeicher somit $640 \times 480 \times 4 = 1.228.800$ Byte. Der Videospeicher ist demnach genau wie ein gewöhnliches eindimensionales Array aufgebaut, das aus einer langen Reihe von 32-Bit-Werten bzw. aus 307200 zusammenhängenden Gruppen von jeweils 32 Bit besteht, die direkt aufeinanderfolgen.

Die Elemente des Videospeichers lassen sich nach demselben Prinzip wie die Komponenten eines gewöhnlichen Arrays verändern – weist man einer bestimmten Gruppe von **32** Bit einen neuen Wert zu, ist die Veränderung der Farbe des entsprechenden Pixels für den Betrachter sofort auf dem Bildschirm sichtbar.

1.3.2 Die Position eines Pixels

Die Oberfläche des Bildschirms weist aufgrund ihrer Länge und Breite die Eigenschaften eines zweidimensionalen Gegenstandes auf – die einfachste Art, die Position eines Pixels auf dem Bildschirm anzugeben, besteht demnach in der Spezifikation einer x- und einer y-Koordinate. Der Videospeicher muss aufgrund der linearen Anordnung seiner Elemente jedoch durch eine eindimensionale Struktur beschrieben werden, die zwar über eine Länge, aber keine Breite verfügt.

Gesucht ist demnach eine Methode, mit deren Hilfe die zweidimensionale Position eines Pixels auf dem Bildschirm in die eindimensionale Position umgewandelt werden kann, an der die Farbe des entsprechenden Bildpunktes sich im Videospeicher befindet. Kennt man diese eindimensionale Position, lässt sich die Farbe des gegebenen Pixels problemlos verändern.

Die gesuchte Rechenvorschrift ergibt sich sofort aus der speziellen Art, wie die Positionen der beschriebenen Gruppen von jeweils 32 Bit mit der räumlichen Lage der jeweiligen Pixel auf dem Bildschirm zusammenhängen: Die ersten vier Byte des Videospeichers definieren das Aussehen des ersten Pixels des Bildschirms, der sich in der **unteren linken Ecke** bzw. in der ersten Zeile und ersten Spalte befindet. Anschließend folgen die vier Farbkomponenten des zweiten Bildpunktes, der rechts daneben in der ersten Zeile, zweiten Spalte liegt, dann die Komponenten des dritten Pixels usw.

C++, JAVA und andere Programmiersprachen legen fest, dass das erste Element eines eindimensionalen Arrays an der Position 0 gespeichert sein soll; verweist der Zeiger `*screen` auf den Anfang des Videospeichers, gibt `screen[0]` demnach die Position der vier Farbkomponenten des ersten Pixels des Bildschirms an. An den Positionen 1, 2, 3 usw. befinden sich die Komponenten des zweiten, dritten und der nachfolgenden Bildpunkte der ersten Zeile – es ist daher naheliegend, den **Ursprung** des zweidimensionalen Koordinatensystems des Bildschirms in der **unteren linken Ecke** zu platzieren.

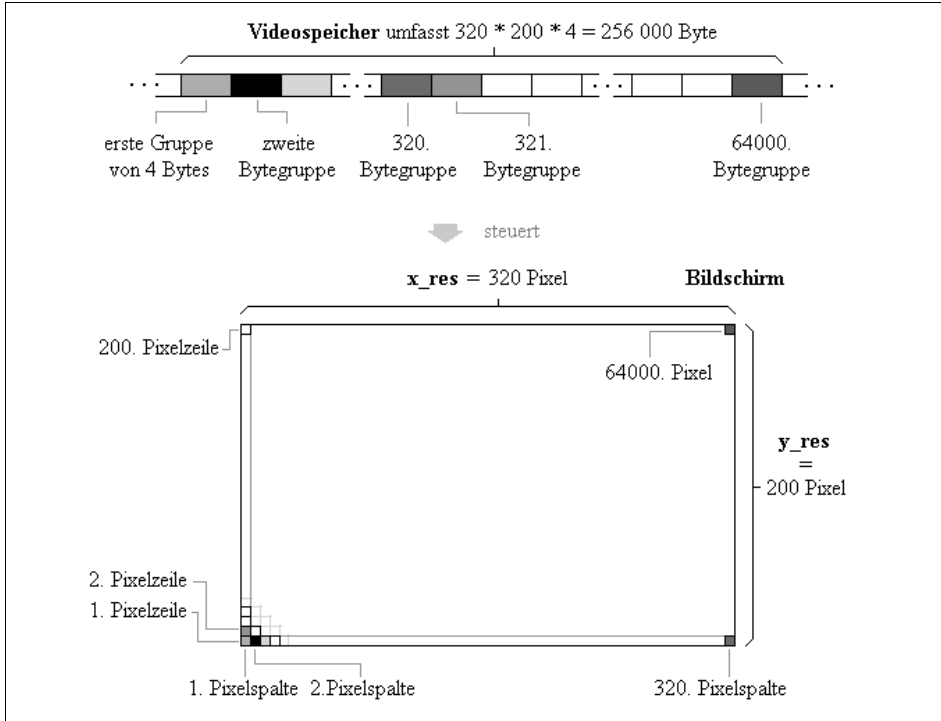


Abb. 1.3: Die Farbe jedes auf dem Bildschirm sichtbaren Pixels ist in Form eines **32-Bit-Zahlenwerts** definiert, der im Videospeicher eingetragen ist

Eine Auflösung von 640×480 Bildpunkten legt fest, dass jede Zeile 640 Pixel enthalten muss – das Ende der ersten Zeile wird somit durch den Bildpunkt in der rechten unteren Ecke des Bildschirms gekennzeichnet, dessen Farbe im Videospeicher an der Position 639 enthalten ist. 640, die darauffolgende Position, gibt das Aussehen des Anfangspunktes der *zweiten* Zeile an, der auf dem Bildschirm direkt *oberhalb* des Pixels am Anfang der ersten Zeile angezeigt wird.

Die zwei beschriebenen Gesetzmäßigkeiten setzen sich über den gesamten Verlauf des Bildschirms fort: Je weiter ein Pixel in einer Zeile vom linken Bildschirmrand entfernt ist, umso größer ist der Wert der eindimensionalen Position der 32-Bit-Gruppe der entsprechenden Farbe im Videospeicher – die x-Achse unseres Koordinatensystems muss demnach **nach rechts** verweisen. Für die y-Koordinaten gilt, dass mit größer werdendem Abstand eines Bildpunktes von dem unteren Rand des Bildschirms sich auch der Wert der dazugehörigen Position im Videospeicher verändert – die y-Achse des Bildschirms verweist **nach oben**.

Die Farben der Anfangspunkte der ersten Zeilen besitzen demzufolge die eindimensionalen Positionen 0, 640, 1280, 1920 usw. Gibt die y-Koordinate **sy** den vertikalen Abstand zwischen einem bestimmten Pixel und dem unteren Rand des Bildschirms an, lässt sich die Position der Farbe des Bildpunktes am *Anfang* der entsprechenden Zeile über den Ausdruck **(sy * 640)** berechnen.

Die anfangs gesuchte Formel zur Berechnung der eindimensionalen Position, an der sich die Farbe eines Pixels mit den zweidimensionalen Koordinaten (sx, sy) im Videospeicher befindet, lässt sich wie folgt formulieren:

$$Position (sx, sy) = sy * x_res + sx$$

Die **horizontale** und **vertikale Auflösung**, das heißt die Anzahl der Bildpunkte in den Zeilen bzw. Spalten des Bildschirms, werden in Form der Konstanten **x_res** und **y_res** angegeben und besitzen in unserem Fall die Werte 640 und 480. Um zu verhindern, dass Speicherstellen außerhalb des Videospeichers beschriftet werden können, legen wir fest, dass die x-Koordinate **sx** eines beliebigen Pixels lediglich Werte größer oder gleich 0 und kleiner oder gleich (**x_res - 1**) annehmen darf, während für die y-Koordinate **sy** dementsprechend gilt: $0 \leq sy \leq (y_res - 1)$. Sowohl **sx** als auch **sy** müssen ganzzahlig sein.

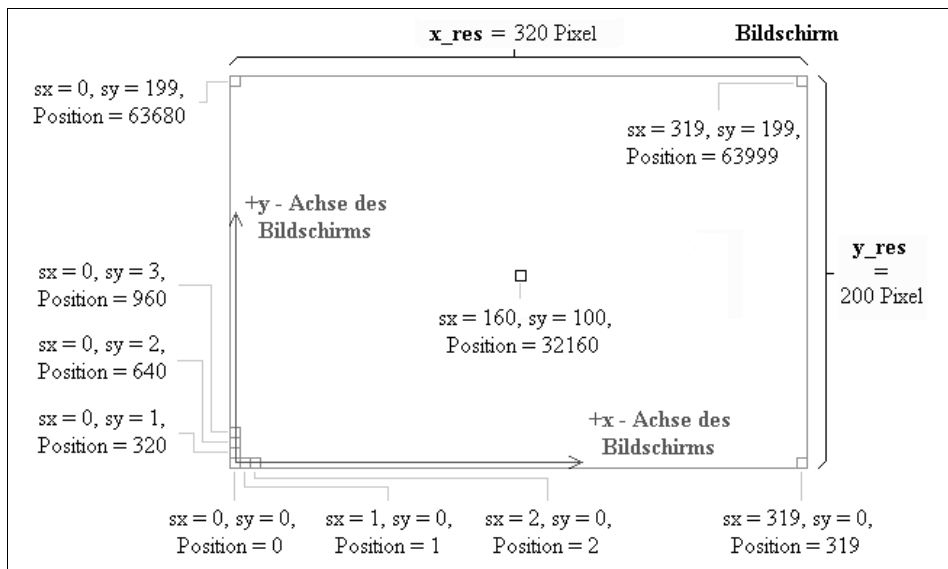


Abb. 1.4: Der Zusammenhang zwischen der Position eines Pixels auf dem Bildschirm und der Position seines Farbwerts im Videospeicher