DirectX vs. GameAPI

Unterschiede, Vor- und Nachteile der Programmierung für Windows PC und Windows-Pocket-PC

Diplomarbeit von
Denny Bertram

Leipzig, Juni 2003

Inhaltsverzeichnis

Einleitung

Personal Computer sind nun schon seit mehreren Jahren fester Bestandteil des täglichen Lebens. Sie finden zum Beispiel Anwendung in der Industrie, im Handel aber auch in der Entertainmentbranche. Gerade die Computerspielindustrie entwickelte sich rasch zu einem der Zugmotoren für die Entwicklung neuer und leistungsfähigerer PC-Komponenten, da immer komplexere Anwendungen wie zum Beispiel in Echtzeit laufende 3D-Spiele die Anforderungen an die Hardware in die Höhe treiben. Mittlerweile sind die meisten Hardwareentwickler dazu übergegangen, bestimmte Funktionalitäten vom Hauptprozessor weg in eigene Komponenten auszulagern. So haben zum Beispiel Sound- oder Grafikkarten Ihren eigenen Speicher und spezielle Zusatzprozessoren bzw. fest implementierte Algorithmen, die für die jeweilige Komponente typische Operationen bedeutend schneller als der Hauptprozessor ausführen können.

Die Computerspielindustrie ist es auch, die immer ausgefallenere Eingabegeräte für den PC hervorbringt. Wurden dereinst Nutzereingaben allein über die Tastatur gesteuert, werden inzwischen Joysticks, Gamepads, Trackballs, Force-Feedback-Lenkräder und in naher Zukunft sogar Cyberanzüge angeboten.

Auch in Zukunft wird die Computerindustrie immer modernere und auch neue Komponenten für die verschiedensten Teilbereiche entwickeln und somit die Angebotspalette ständig erweitern. Die Vielzahl der Hardwarekomponenten bringt aber auch ein großes Problem mit sich. Anwendungen für den PC sollen flexibel sein, d.h. auf möglichst vielen Systemen laufen, um sie einem recht großen Kundenkreis anbieten zu können. Viele gleichartige Systemkomponenten verfolgen aber in ihrem Aufbau oft unterschiedliche Ansätze, so dass sie einzeln in einer Anwendung berücksichtigt werden müssten. Bei der Unmenge an Komponenten wäre dies ein äußerst zeit- und kostenintensives Unterfangen, um zumindest einen großen Teil der Hardware in die Anwendung zu integrieren.

Die Firma Microsoft hat dem jedoch Abhilfe geschaffen und mit DirectX eine Funktionsbibliothek entwickelt, die in der Lage ist, jede Hardware anzusprechen, für die ein entsprechender DirectX-Treiber vom Hersteller bereitgestellt wird. Wie dies im einzelnen funktioniert, wird die vorliegende Arbeit am Beispiel von 2D-Grafikdarstellung, Eingabeverwaltung und Soundintegration demonstrieren.

Und noch ein weiterer wesentlicher Vorteil von DirectX soll aufgezeigt werden. Die Standardfunktionen unter Windows bieten keinen Direktzugriff auf die Hardware und unterbinden bzw. blockieren somit das zielgerichtete Zugreifen auf Funktionen oder Speicherbereiche der betreffenden Komponente. Dies führt allerdings zu deutlichen Geschwindigkeitseinbußen, da zum einen die Hardwarebeschleunigung einer solchen Komponente nicht genutzt wird und zum anderen über bereits bestehende Funktionen zugegriffen werden muss, die mitunter intern Aufrufe starten, die für die aktuelle Anwendung gar nicht relevant sind. DirectX bietet allerdings Möglichkeiten, die Standardsystemschnittstellen zu umgehen, um so direkt auf die Hardware zugreifen zu können. Gerade bei Hochleistungsapplikationen, bei denen es auf die schnelle

Ausführung des Programms ankommt, lassen sich so deutliche Geschwindigkeitsvorteile erzielen.

Doch nicht nur auf dem Sektor für Desktop-PCs schreitet die Entwicklung voran. In der modernen Geschäftswelt ist immer mehr Mobilität, Flexibilität und ständige Erreichbarkeit gefordert. Gerade für diese Zielgruppe wurden in den letzten Jahren Geräte entwickelt, die die Funktionalität eines Desktop-PCs in einem möglichst kleinen, handlichen Format umsetzen sollen. Pocket-PCs, Geräte in der Größe eines Gameboys, aber mit der Leistungsfähigkeit und dem Funktionsumfang eines Pentium-II-PCs, sind da nur ein Beispiel dafür. Bereits jetzt gibt es knapp 8 Millionen Pocket-PC-Besitzer und es wird erwartet, dass sich diese Zahl aufgrund der weiter fallenden Gerätepreise und der fortwährenden Weiterentwicklung noch deutlich erhöhen wird. In den Visionen der Hersteller wird der Pocket-PC aufgrund seines Leistungsspektrums als das Gerät angesehen, welches sowohl die kleineren Spielkonsolen wie den Gameboy, als auch in vielen Bereichen den Laptop ersetzen wird.

Zahlreiche Anwendungen wurden bereits für den Pocket-PC entwickelt und mit steigender Leistungsfähigkeit wuchs auch der Bedarf an einer Software, die analog DirectX direkten Zugriff auf die Hardware ermöglicht - die GameAPI.

Diese Arbeit bietet nun einen Überblick über das Programmieren von Windows-Anwendungen im allgemeinen, demonstriert anhand einer instruktiven Beschreibung und ausgewählten Beispielen die Möglichkeiten von DirectX und der GameAPI und liefert letztendlich über einige Geschwindigkeitstests einen Einblick in die Stärken und Schwächen beider Implementationen. Dabei wird der Vergleich aber nur auf Ebenen geführt, die sowohl in DirectX als auch in der GameAPI berücksichtigt werden: 2D-Grafikdarstellung, Eingabeverwaltung und Klang- bzw. Geräuschintegration. Der Funktionsumfang von DirectX ist zwar deutlich größer, jedoch ist die Leistung der momentan aktuellen Pocket-PC-Generation noch recht begrenzt, so dass zum Beispiel 3D-Anzeige und -Darstellung in der GameAPI noch nicht umgesetzt wurden.

Das erste Kapitel dieser Arbeit beschäftigt sich damit, wie Pocket-PCs hinsichtlich ihrer Leistungsfähigkeit im Vergleich zu einem Desktop-PC einzuordnen sind. Weiterhin werden Gemeinsamkeiten und Unterschiede beider Geräte bzgl. der Hardware und des Betriebssystems aufgezeigt.
Das zweite Kapitel befasst sich mit den theoretischen Grundlagen zur Implementierung einer Applikation für das Betriebssystem Windows.
Im dritten Kapitel werden die Grundlagen des COM (Component Object Modells), von DirectX und dessen Pocket-PC-Äquivalent GameAPI näher beleuchtet.
Innerhalb der Kapitel 4 bis 7 wird anhand ausgewählter Beispiele eine Anleitung zum Programmieren von DirectX- bzw. GameAPI-basierten Anwendungen für die jeweilige Plattform gegeben.
Zuerst wird in Kapitel 4 die Fenstererstellung und Basisprogrammierung für Windowsprogramme erläutert, Kapitel 5 enthält eine Einführung in die 2D-Grafikdarstellung und in Kapitel 6 wird die Nutzereingabeerkennung und -verwaltung behandelt. Innerhalb von Kapitel 7 werden diese Beispiele noch durch eine Soundkomponente ergänzt.

Im Kapitel 8 sind einige Geschwindigkeitstests zu finden, die sich auf die Programmierbeispiele der Kapitel 4 bis 7 beziehen und die Performance von Desktop- und Pocket-PCs auf verschiedenen Gebieten untersuchen und vergleichen.
Kapitel 9 fasst schließlich die Ergebnisse dieser Arbeit noch einmal zusammen und offenbart einen Ausblick auf die zukünftige Entwicklung von DirectX, der GameAPI und des Pocket-PCs im allgemeinen.

Ziel dieser Arbeit ist es nun, einen Vergleich der beiden Plattformen in den Punkten Programmiertechnik, Hardwareunterstützung und Leistungsfähigkeit zu liefern und eine Grundlage für weiterführende DirectX- bzw. GameAPI-basierte Anwendungen zu schaffen.

Kapitel 1: Desktop-PC und Pocket-PC im Vergleich

1.1. Der Entwicklungsstand aktueller Desktop-PCs

Der rasante Fortschritt in der Computerbranche hat die Leistung moderner PCs in Dimensionen getrieben, die vor wenigen Jahren noch kaum vorstellbar waren. Prozessoren erreichen bereits 3 GHz Taktfrequenz, Arbeitsspeichergrößen von 256 MB (Megabyte) sind mittlerweile Standard und Computerkomponenten wie z.b. Grafikkarten warten mit eigenem Arbeitsspeicher in der Größenordnung früherer Festplatten auf. Auch die Kapazität der Datenträger ist wesentlich verbessert worden. CD-Roms können beispielsweise bis zu 800 MB speichern, DVDs gar bis zu 6 GB (Gigabyte).

Aufgrund der immer weiter fortschreitenden Verbesserung der Hardware für Desktop-PCs bietet sich dem PC-Besitzer eine immer breiter werdende Palette von Nutzungsmöglichkeiten. DOS hat in frühen Versionen nur 16 verschiedene Farben und eine Bildschirmauflösung von 320x240 Pixeln verwalten können, seit Erscheinen von Windows95 sind es bereits 16.Mio Farben bei einer Bildschirmauflösung von 800x600 bis 1600x1200 Bildpunkten. 3D-Beschleuniger-Karten erlauben es, virtuelle 3D-Welten in Echtzeit darzustellen, zahlreiche Zusatzsteuergeräte wie ForceFeedback-Lenkräder oder Joysticks erschaffen neue Interaktionsmöglichkeiten mit dem PC.

1.2. Der Entwicklungsstand aktueller Pocket-PCs

PDAs wurden ursprünglich als elektronische Terminplaner mit integriertem Adressbuch entwickelt. Die aktuelle Gerätegeneration ist mittlerweile als Pocket-PC bekannt. Doch was können sie leisten, was bieten sie für Möglichkeiten gegenüber einem Desktop-PC, in welchen Bereichen ist ein Pocket-PC ebenbürtig oder gar leistungsstärker?

Aktuelle Prozessoren für Pocket-PCs sind in Ihrer Leistung mit der eines PentiumII-Prozessors vergleichbar, besitzen aber einen etwas geringeren Funktionsumfang. So werden zum Beispiel Fließkommazahlen von der Hardware nicht unterstützt und müssen per Software emuliert werden.
Das für die Arbeit verwendete Testgerät ist ein Yakumo Delta (siehe Abbildung 1.2.A) und besitzt einen StrongArm-Prozessor mit 206 MHz Taktfrequenz. Ursprüngliche Prozessorarchitekturen wie MIPS gelten inzwischen als veraltet, so dass sich Arm-Prozessoren als Standard etabliert haben.

Im Bereich Arbeitsspeicher (RAM) und Festplattengröße ist folgendes bemerkenswert: Es gibt weder Festplatte noch RAM im eigentlichen Sinne. Pocket-PCs besitzen nur einen temporären Speicher, (das Testgerät hat 36,45 MB), wobei das System oder auch der Nutzer festlegen können, zu welchen Anteilen der zur Verfügung stehende Platz in Arbeits- bzw. Datenspeicher unterteilt wird. Temporärer Speicher bedeutet in diesem Fall, dass der gesamte Speicher mitsamt der Programme gelöscht werden würde, sollte die Stromzufuhr unterbrochen werden. Geräteinterne Lithium-Ionenakkus sorgen aber für hinreichende Lademöglichkeiten und Standby-Zeiten von ca. einer Woche, diverse Erweiterungskarten bieten Festspeicher auch für Pocket-PCs.

Abbildung 1.2.A: Der Pocket-PC Yakumo Delta

Die Übertragung von Daten auf einen Pocket-PC wird mittels USB oder auch durch eine Infrarot-Schnittstelle realisiert. Während der Installation der Treibersoftware für das Gerät wird auf dem als Datenquelle dienenden Desktop-PC ein virtuelles Laufwerk erstellt. Dieses Laufwerk wird automatisch mittels einer speziellen Software (Active Sync) mit dem Inhalt des Pocket-PC-Ordners "My Documents" synchronisiert, also die Daten in diesem Ordner werden denen des virtuellen Laufwerkes angepasst, nötigenfalls ergänzt, überschrieben oder gelöscht. Die meisten Installationsprogramme sind sogar in der Lage, direkt vom CD-ROM-Laufwerk des Desktop-PCs aus Anwendungen auf den PDA zu kopieren und dort natürlich auch einen Link im Startmenü oder ähnliches einzurichten.

Des Weiteren bietet ein moderner Pocket-PC ein 240 x 320 Display mit 16-Bit-Farbtiefe, also über 65536 verschiedene Farben, die dargestellt werden können. Das Display fungiert zudem als Touchscreen und bietet so dem Nutzer eine bequeme Möglichkeit, mit dem Gerät zu interagieren.

Zusätzlich stehen dem Nutzer drei, an manchen Geräten auch vier Programmtasten (program buttons) und ein Navigationsrad (navigation pad) zur Verfügung. Die Steuertasten dienen zum Schnellstart PDA-typischer Funktionen wie dem Terminkalender oder dem Adressbuch, können aber auch innerhalb von Programmen neue Funktionen zugewiesen bekommen. Das Navigationsrad über nimmt zumeist Scrolling-Aufgaben, kann aber auch zum Bewegen/Steuern verschiedener Objekte innerhalb von Applikationen verwendet werden.

Viele Steuerfunktionen sind bei einem Desktop-PC Maus-basiert. Bei einem Pocket-PC wurde die Maus durch den sogenannten Stylus ersetzt, eine Art Zeigestab, der die

direkte Interaktion mit dem Touchscreen des Pocket-PCs ermöglicht. Dieser Stylus ist allerdings nicht elektronisch, optisch oder anderweitig an den Touchscreen gebunden, so dass ein Fingerdruck die gleiche Reaktion hervorruft, wie die Berührung mit dem Stylus.

Für einen eventuellen Neustart des Gerätes ist eine Resettaste im Gehäuse integriert, ebenso eine Aufnahme-Taste, um Sprachmitteilungen direkt auf dem PDA als Klangdatei zu speichern.

Auch wurde auf einen Soundchip nicht verzichtet, der in der Lage ist, Klangdateien wie .wav wiederzugeben. Eine Kopfhörerbuchse und ein im Gehäuse integrierter Lautsprecher dienen dabei als Schnittstelle zur Außenwelt.

1.3. Aufbau des Pocket-PC-Betriebssystems Windows CE

Windows CE basiert auf dem Kernel der Windows-Versionen für den Desktop-PC und arbeitet weitgehend nach den selben Prinzipien. Die grafische Eingabeoberfläche und die überwiegend tastaturunabhängige Steuerung findet sich genau so wieder wie das Fenster-Konzept. Auf einige Features wie zum Beispiel den "Arbeitsplatz" und das "Papierkorbsystem" haben die Entwickler hingegen verzichtet. Die Funktionen des "Arbeitsplatzes", insbesondere das Darstellen der Datei- und Verzeichnisstruktur, lässt sich hier nur über den Datei-Explorer realisieren, auf den "Papierkorb" wurde indessen aufgrund der relativ geringen Speichermöglichkeiten gänzlich verzichtet.

An deren Stelle treten PDA-typische Applikationen wie Kalender, Terminplaner, Notizblöcke und Adressbücher. Links zu diesen Applikationen sind auf dem Desktop des Pocket-PC integriert bzw. direkt mit speziellen Starttasten auf dem Gehäuse verbunden.

Darüber befindet sich die Taskleiste, deren Startmenü-Einträge allerdings auf maximal neun durch den Nutzer selbst definierbare Programme fixiert sind. Abgesehen von den vier direkt auf dem Desktop integrierten Anwendungen bietet die Taskleiste die einzige Möglichkeit, auf Programme mittels Links zuzugreifen, direkte Zugriffe auf Dateien sind nur mittels Dateiexplorer zu realisieren.

Doch bei all den Veränderungen und Zusätzen ist trotzdem zu erkennen, dass das Grundprinzip der Desktop-Windows-Varianten auch hier Anwendung findet und sich aufgrund der ähnlichen Windows-Oberfläche beider Systeme keine programmiertechnischen Hindernisse ergeben, die ein komplett neues Denken bei der Anwendungsentwicklung und -anpassung erfordern.

Kapitel 2: Windows - Theoretische Grundlagen

2.1. Multitasking/Multithreading

Um den Aufbau und die Funktionsweise der im folgenden verwendeten Routinen zu verstehen, ist es notwendig zu erklären, wie das Betriebssystem Windows intern arbeitet, welche Architektur es aufweist und welche Dinge ein typisches Windows-Programm ausmachen.

Windows ist im Gegensatz zu DOS ein sogenanntes "Multitasking Operating System", dass es erlaubt, mehrere Anwendungen und/oder Prozesse simultan ablaufen zu lassen. Dies bedeutet, dass unter Windows jederzeit mehrere Prozesse laufen können, ohne dass eine einzelne Anwendung das System komplett übernehmen bzw. blockieren kann. Obwohl sich die Windows-Versionen in einer Anzahl technischer Details unterscheiden, arbeiten sie doch alle nach dem selben Schema. Da zudem noch die DirectX-Schnittstelle für jede dieser Windows-Ausführungen ohne Änderungen oder Anpassungen verwendbar ist, sind diese Details vernachlässigbar.

Was diese Versionen jedoch gemeinsam haben, ist zum Beispiel der Scheduler. Ein Scheduler ist in diesem Fall ein systeminterner Plan, mit dem festgelegt wird, in welcher Reihenfolge den Prozessen eine bestimmte Zeitspanne gegeben wird, um die CPU zu nutzen. Bei komplexeren Schedulern lässt sich auch die Priorität eines Prozesses festlegen, ob dieser evtl. mehr Rechenzeit zugeteilt bekommt als ein anderer vielleicht weniger "wichtiger" Prozess und ob ein Prozess vorzeitig abgebrochen wird, weil eine höher priorisierte Anfrage die CPU benötigt. Windows verwendet einen solchen komplexen Scheduler.

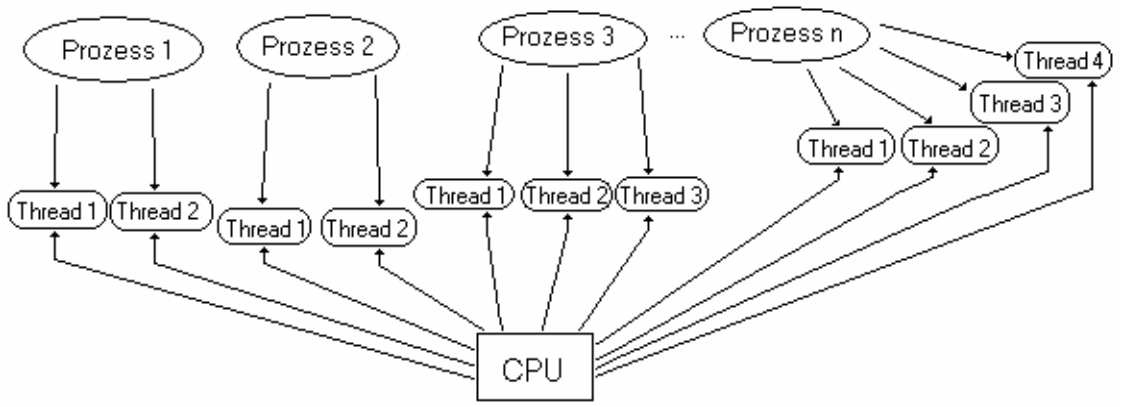
Bei näherer Betrachtung ist festzustellen, dass Prozesse unter Windows aus einem oder mehreren Threads bestehen (siehe Abb.2.1.A.), wobei der Windows-interne Scheduler diesen Threads CPU-Zeit zuweist und nicht direkt den Prozessen. Das ist ein ganz entscheidender Unterschied zu DOS. Unter DOS gibt es nur einen einzigen Thread. Wird dort ein Programm gestartet, ist dass der einzige laufende Prozess. Alle anderen Prozesse werden dort unterbunden (außer dem Interrupt-Handler (interrupt handler), der system- oder programmwichtige Eingaben z.B. die eines externen Eingabegerätes wie Maus oder Tastatur registriert und an das System weitergibt).

Sollte zum Beispiel eine Spiele-Applikation unter DOS verschiedene Objekte gleichzeitig über den Bildschirm bewegen, musste Multithreading bzw. Multitasking vom Programmierer selbst simuliert werden. Mit Windows wird dem Programmierer nun ein Instrument in die Hand gegeben, das eben diese Features schon von Haus aus unterstützt.

Allerdings sei hier angemerkt, dass, solange ein Rechner nur eine CPU hat, kein echtes Multitasking möglich ist. Windows schaltet bei einem einzelnen Prozessor nur so schnell zwischen den einzelnen Threads hin und her, dass es nur so scheint, als würde alles parallel abgearbeitet. Steht jedoch zum Beispiel ein Dual Pentium Board mit zwei CPUs

zur Verfügung, so können auch zwei vom Typ her gleiche Anweisungen parallel abgearbeitet werden.

Abbildung 2.1.A: In Threads aufgeteilte Prozesse unter Windows



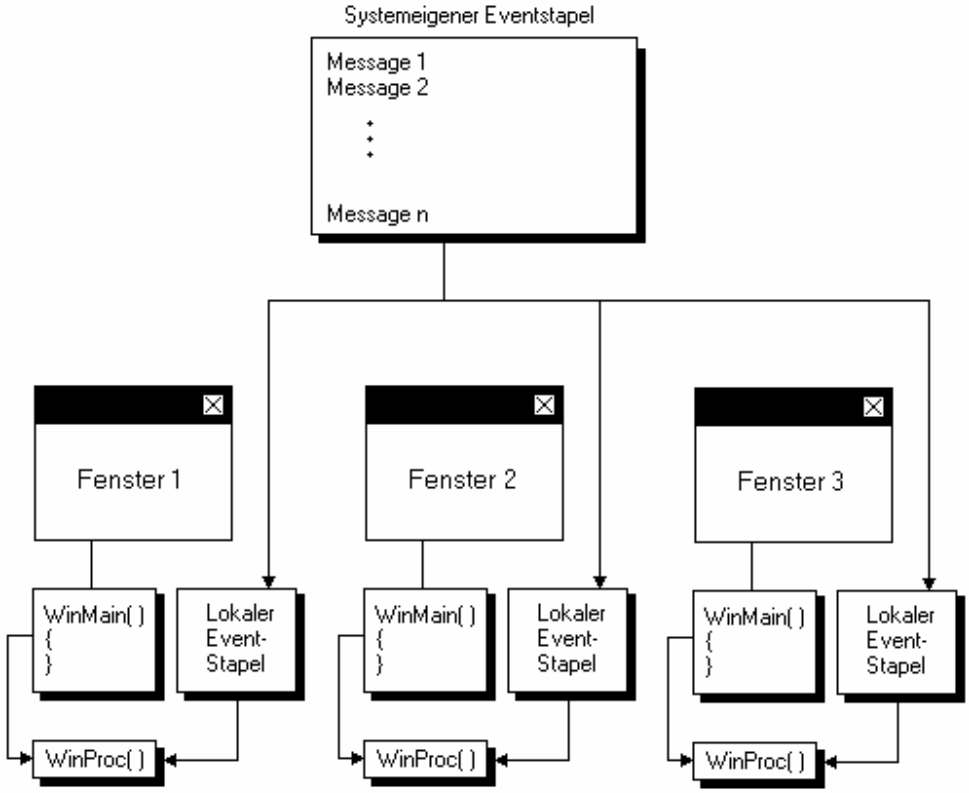
2.2. Das Event-Modell

Eine weitere Eigenheit von Windows ist das Event-Modell. Im Gegensatz zu DOS warten die meisten Windows-Programme auf eine Eingabe des Nutzers oder auf ein bestimmtes Ereignis (Event) und reagieren entsprechend. Abbildung 2.2. verdeutlicht dies an einem Beispiel mit drei Fenstern.

Hier werden eine Anzahl von Anwendungsfenstern (window) dargestellt, wobei jedes seine Nachrichten bzw. Events an Windows schickt. Manche der Nachrichten wird Windows selbst verarbeiten, der Großteil jedoch wird weitergeleitet und in der Applikation selbst ausgewertet.

Was nun im einzelnen Events sind, welche es gibt und wie Windows darauf reagiert, wird Bestandteil des Abschnittes "Der Windows-Eventhandler" im 4.Kapitel sein.

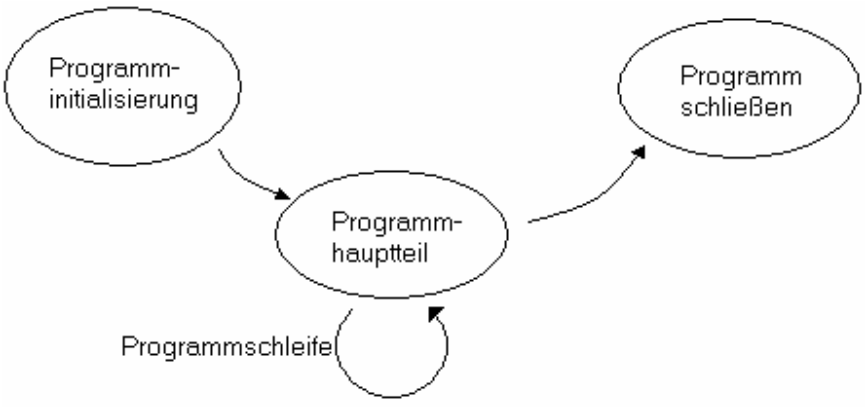
Abbildung 2.2.A: Ereignisbehandlung unter Windows-Betriebssystemen



2.3. Ereignisorientierte Programmierung

Da Programme nicht nur ein einziges Event auslösen und das auch nicht nur zu einem bestimmten Zeitpunkt, bietet es sich an, in bestimmten Zeitabschnitten immer wieder aufs neue zu testen, ob Ereignisse eingetreten sind. Das können zum Beispiel die im Abschnitt 2.2. erwähnten Nutzereingaben wie Mausklicks und -bewegungen, Tastatureingaben oder Tastendruck bei Geräten wie dem Joystick sein, aber auch Nachrichten wie "Programm starten" und "Programm beenden", "Fokus setzen" oder "auf das Hauptausgabegerät zeichnen". Abbildung 2.3.A zeigt, wie ein solcher Zyklus aussehen kann:

Abbildung 2.3.A: Transitionsdiagramm eines Ereignistest-Zykluses



Hierbei haben die einzelnen Funktionen folgende Aufgabe:

Programminitialisierung:

Hier werden alle für das Programm benötigten Ressourcen geladen, Bilder eingelesen, Speicher reserviert und viele weitere Dinge initialisiert, die in späteren Kapiteln noch näher beschrieben werden.

Hauptprogramm:

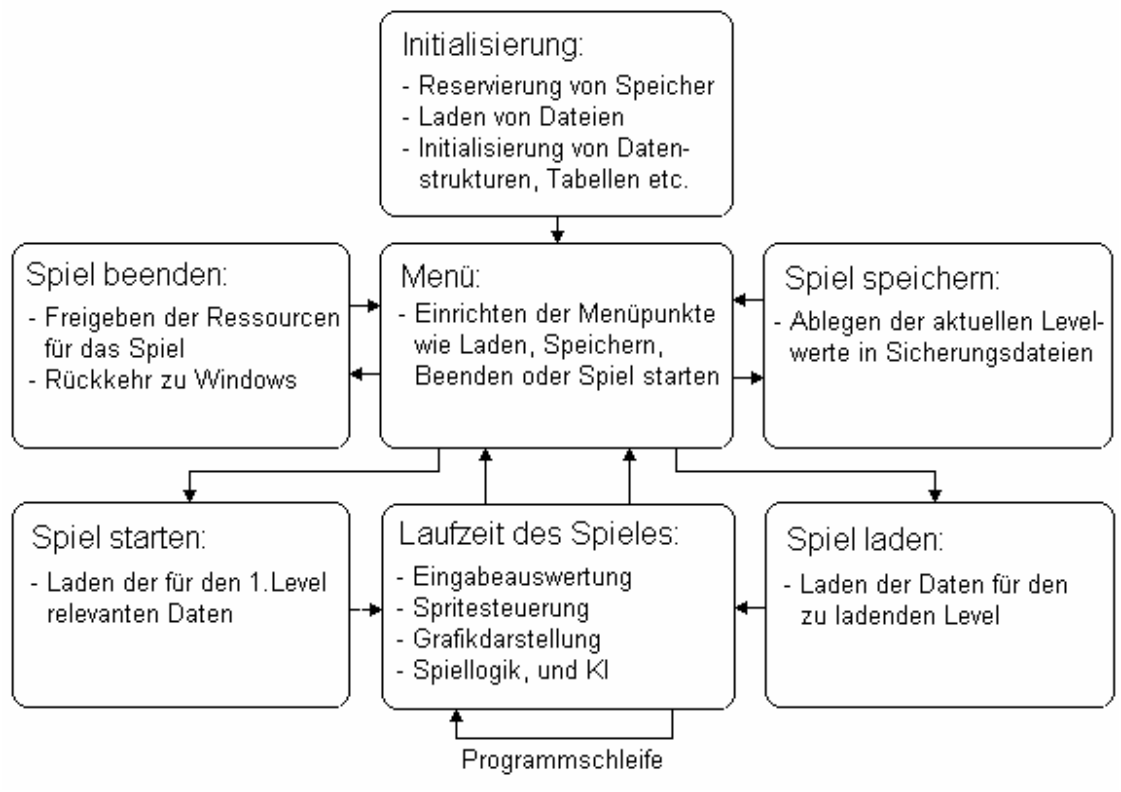
In diesem Teil wird die Programmlogik definiert, also alles was während des Programmablaufes immer wieder passieren soll. So wird hier zum Beispiel errechnet, was gerade auf dem Bildschirm zu sehen sein muss (Screenmanagement), gewisse programminterne Bedingungen überprüft und für bestimmte Nutzerangaben oder Systemnachrichten die entsprechende Ereignisbehandlung definiert. Ebenso bietet es sich an, Routinen zur zeitlichen Synchronisation der Programmschleife an dieser Stelle mit unterbringen.

Programm schließen:

Dieser Programmabschnitt sorgt dafür, dass reservierter Speicher für zum Beispiel Klassen, Strukturen Listen usw. wieder freigegeben wird, dass eventuell geöffnete Dateien geschlossen und bestimmte Einstellungen wie das Zurücksetzen auf Standardschriften bzw. Mauszeiger realisiert werden. Nach Beenden von *App_exit()* wird das Programm geschlossen und der Nutzer kehrt zur Windows-Oberfläche zurück.

Natürlich lässt sich dieser Zyklus beliebig erweitern, mehrere Hauptprogramm-Funktionen in verschiedenen Schleifen unterbringen, Menüs einfügen und noch vieles mehr, je nach Fantasie des Programmierers oder den Erfordernissen der Applikation. Abbildung 2.3.B zeigt eine mögliche Architektur eines Computerspieles.

Abbildung 2.3.B: Beispielarchitektur eines Computerspieles

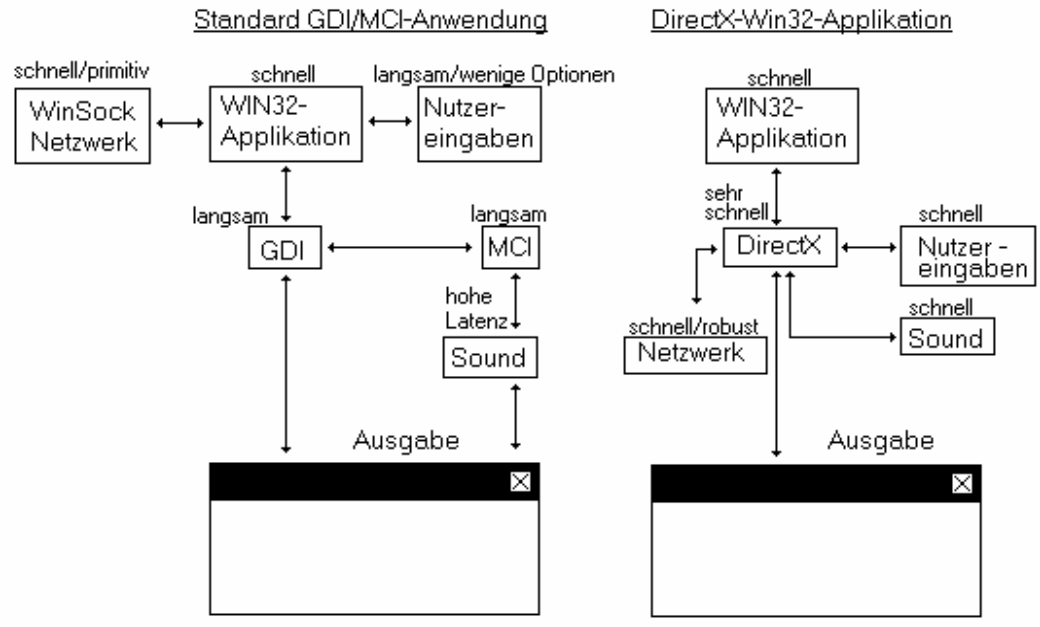


Kapitel 3: Die Grundlagen von COM, DirectX und der GameAPI

3.1. DirectX- Grundlagen

Das GDI (grafics device interface) und das MCI (media control interface) sind die beiden Schnittstellen (Interfaces) zu den die Hardware ansteuernden Funktion, die in Windows schon standardmäßig integriert sind. Ein Programmierer muss sich dieser beiden Schnittstellen bedienen, wenn er unter Windows Programme entwickeln will, die sich auf bestimmte Hardware stützt. Da GDI und MCI dem Anwendungsentwickler aber nur vergleichsweise langsame Funktionen bereitstellen, aber keinen Direktzugriff auf den Videospeicher und die Eingabegeräte des Systems unterstützen, bieten verschiedene Hersteller Programmpakete an, die GDI und MCI umgehen und mitunter direkten Zugriff auf die Hardware gewähren. Ein solches Paket aus dem Hause Microsoft ist zum Beispiel DirectX. Abbildung 3.1.A. zeigt die Unterschiede im Aufbau einer GDI/MCI-Anwendung und einer DirectX-Applikation:

Abbildung 3.1.A: Performance-Vergleich des GDI (grafics device interface) und des MCI (media control interface) mit DirectX



DirectX ist ein Softwaresystem, das als Schnittstelle zwischen den Hardwaretreibern und einer DirectX-unterstützenden Anwendung dient. Es stellt Funktionen bereit, die unabhängig von der Hardware immer anwendbar sind, auch dann, wenn Hardwaregeräte direkt angesprochen werden. DirectX-Funktionen ermitteln automatisch die verwendeten Systemkomponenten und greifen dann auf die installierten Hardware-Treiber zu. Unabhängig von der Gerätekonfiguration können so immer die selben allgemeinen DirectX-Funktionen für Audio, Video, Eingaben, Netzwerke usw. verwendet werden, ohne dass sich der Programmierer um die Hardwarekomponenten zu kümmern braucht. Der große Vorteil von DirectX ist, dass es wesentlich schnellere Routinen bietet als GDI und MCI, und einem Programmierer die Freiheit lässt, fast auf Hardware-Ebene

zu programmieren. Dies ist möglich durch die sogenannte COM-Technologie (COM = Component Object Model) und einer Reihe von Treibern und Bibliotheken, die sowohl von Microsoft als auch von Hardwareherstellern erstellt wurden. Microsoft gab eine Reihe von Bestimmungen vor - Funktionen, Variablen, Datenstrukturen usw. - und diese mussten von den Hardwareherstellern verwendet werden, wenn sie Windows-Treiber für Ihre Geräte implementierten. Solange diese Konventionen eingehalten werden, wird DirectX in der Lage sein, die Hardware anzusprechen und zu managen.

DirectX selbst besteht aus einer Anzahl verschiedener Komponenten:
- DirectDraw: DirectDraw ist die primäre Render- und 2D-Darstellungsroutine, die die Grafikanzeige steuert. Alle Grafiken einer DirectX-basierten Anwendung werden mittels dieses Funktionspaketes dargestellt. DirectDraw steuert die Grafikkartenfunktionen innerhalb des Systems und stellt u.a. Routinen zum direkten Zugriff auf den Videospeicher der Grafikkarte zur Verfügung.
- Direct3DRM: Direct3D Retained Mode ist ein Objekt- und framebasiertes Hochleistungs-3D-System, dass zum Programmieren von 3D-Programmen eingesetzt werden kann. Es unterstützt 3D-Beschleunigung, ist aber aufgrund seiner relativ langsamen Funktionen nur für im Automodus laufende 3D-Demonstrationsprogamme, Modelldarsteller oder extrem langsam laufende Applikationen zu empfehlen.
- Direct3DIM: Direct3D Immediate Mode ist ein Low-Level-3D-System, d.h. es bietet nur grundlegende 3D-Funktionen wie DrawPrimitive() an, die einfache geometrische Objekte darstellt. Direct3DIM ist weniger komplex, aber dafür deutlich schneller als Direct3DRM.
- DirectInput: DirectInput behandelt die Nutzereingaben und verwaltet alle diesbezüglichen Geräte wie Maus, Keyboard, Tastatur, Joystick, Force-Feedback-Lenkrad, Zeichenbrett usw..
- DirectPlay: DirectPlay behandelt die Netzwerkaspekte in DirectX. Es gibt dem Programmierer die Möglichkeit, abstrakte Verbindungen mittels Internet, Modem, Direct Link (Direktverbindung über Null-Modem-Kabel) oder irgendeine andere Art von Medium erschaffen und unterstützt dabei die Konzepte *sessions* (Differenzierung mehrerer Ausführungen einer laufenden Anwendung) und *lobbies* (virtueller Versammlungsraum, Ausgangspunkt für zum Beispiel Multiplayer-Online-Spiele). DirectPlay erlaubt es, diese Verbindungen ohne näheres Wissen der Netzwerkarchitektur, Netzwerktreiber, Sockets o.ä. anzulegen. Es sendet und empfängt lediglich Datenpakete, deren Inhalt und Funktionsfähigkeit aber dem Programmierer überlassen bleiben.
- DirectSetup: DirectSetup/Autoplay sind Quasi-DirectX-Komponenten, die es einem Programm erlauben, DirectX von einer Applikation aus auf den Nutzerrechner zu installieren und ein Programm direkt zu starten, sobald die entsprechende Programm-CD in das CDROM-Laufwerk eingelegt wird.
DirectSetup ist eine Sammlung von Funktionen, die die Laufzeit-Komponenten von DirectX auf einen Nutzerrechner laden und diese in der Windows-Registrierungsdatei (windows registry) registrieren.
Autoplay ist das Standard-Untersystem bei CDs, dass nach der autoplay.inf-Datei schaut und wenn diese existiert, deren Batch-Kommandos ausführt.
- DirectShow: DirectShow wird im allgemeinen für Datenströme (media streams) verwendet und bietet Funktionen für hochqualitatives Datenrecording (Capturing) und die Wiedergabe von Audio- und Videodaten. Es unterstützt dabei eine Vielzahl von Formaten wie zum Beispiel AVI, ASF, MP3, WAV oder WDM. DirectShow unterstützt zudem Hardware-Beschleunigung, sofern vorhanden.

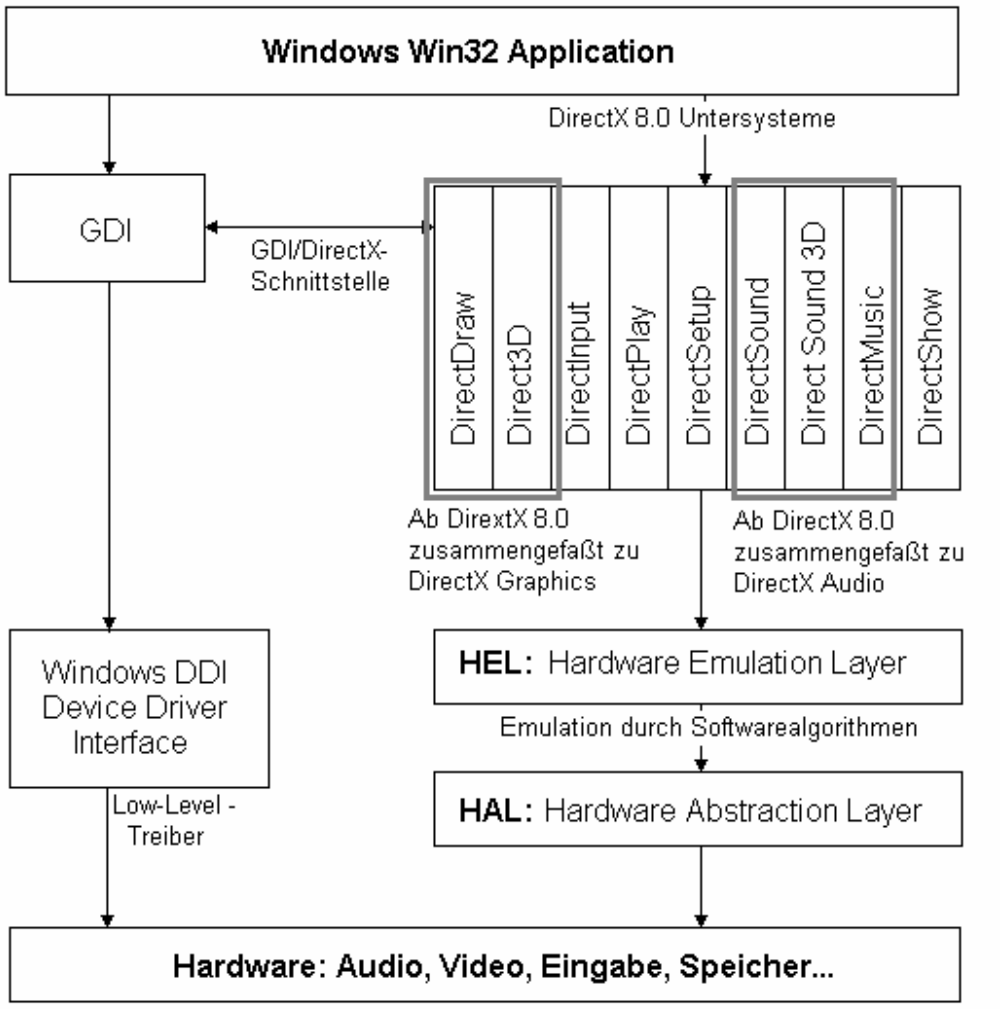
- DirectSound: DirectSound ist die Klangkomponente für digitalen Sound in DirectX und unterstützt demnach nur digitale Formate wie WAV, MIDI hingegen nicht.
- DirectSound3D: DirectSound3D ist die 3D-Raumklangkomponente (3DSound) in DirectX, die es erlaubt, 3D-Klänge so im Raum zu positionieren, als würde sich die Tonquelle bewegen. Aufgrund der Komplexität dieser Funktionen ist jedoch Hardwareunterstützung der 3D-Effekte seitens der Soundkarte erforderlich.
- DirectMusic: DirectMusic ist die Klangkomponente für MIDI in DirectX und beherbergt Funktionen zur Bearbeitung und Wiedergabe von MIDI-Dateien. Zudem bietet DirectMusic ein DLS-System (DLS = downloadable sounds), mit dem sich digitale Repräsentationen von Instrumenten erstellen und als MIDI abspielen lassen. Mit anderen Worten gesagt, lassen sich die Standard-General-Midi-Instrumente durch beliebige digitale Klänge ersetzen und mittels der MIDI-Controller manipulieren.

Hinweis: Seit DirectX 8.0 hat Microsoft die Funktionspakete DirectDraw, Direct3DRM, Direct3DIM als DirectX Graphics und die Funktionspakete DirectSound, DirectSound3D, DirectMusic als DirectX Audio zusammengefasst. Auf ältere DirectX-Routinen lässt sich aber nach wie vor problemlos zugreifen, denn Microsoft behält die älteren Funktionsvarianten weiterhin bei. So lässt sich zum Beispiel das Funktionspaket DirectDraw7 aufrufen, obwohl nur DirectX 8.0 installiert ist.

Wie die DirectX-Komponenten zusammenhängen und wie sie zwischen Hardware und Software eingebettet sind, demonstriert Abb.3.1.B.

Die HEL oder Hardware-Emulationsschicht (Hardware Emulation Layer) wird angesprochen, wenn ein Feature nicht von der Hardware unterstützt wird. Bietet zum Beispiel die Grafikkarte keine Möglichkeit, eine 2D-Grafik zu rotieren, wird HAL übergangen und in HEL die Rotation durch einen Software-Algorithmus emuliert. Die Softwareemulation ist zwar zumeist deutlich langsamer als ein entsprechender Hardware-Support, aber eine Applikation würde trotz fehlender Hardware immer noch lauffähig sein, sofern für die erforderlichen Features Software-Algorithmen mit gleicher Funktionalität existieren.

Abbildung 3.1.B: Die Architektur von DirectX



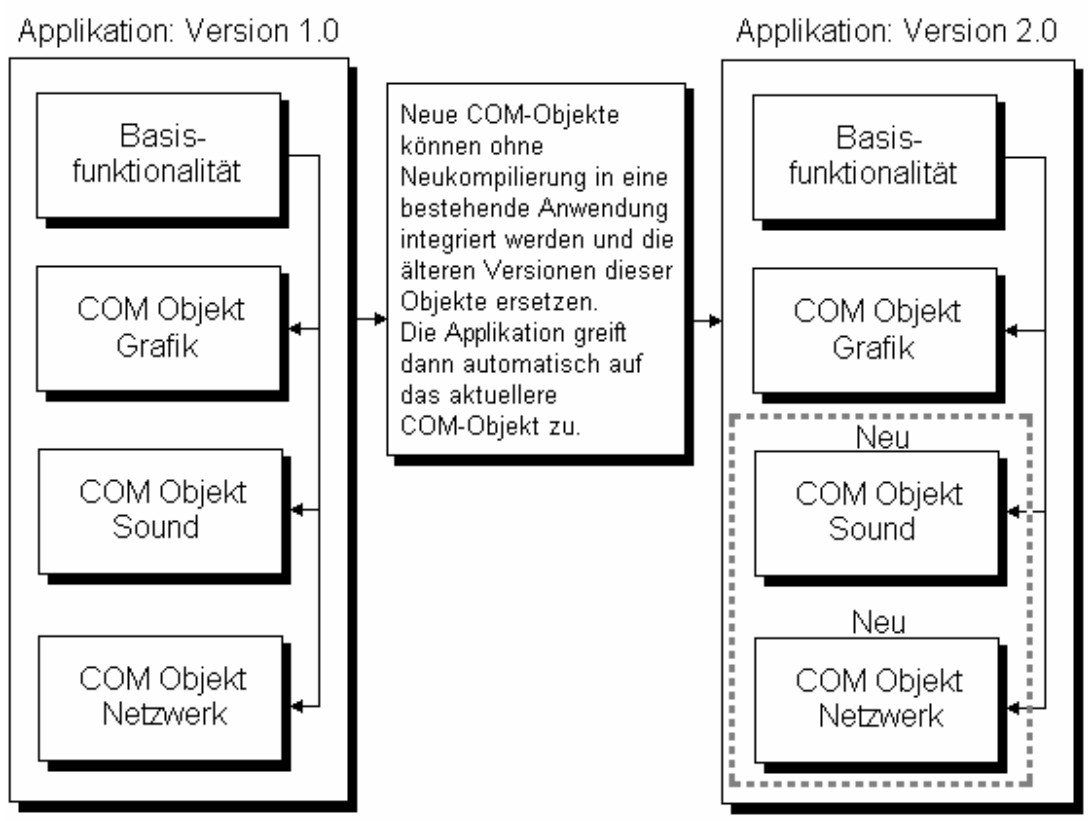
3.2. COM als Basis von DirectX

COM - das Component Object Model - wurde von der Firma Microsoft entwickelt und dient als Basis von DirectX. Dieses Modell soll es ermöglichen, bestimmte Teile eines Programms modular anzulegen, so dass diese Module unabhängig von den anderen Komponenten des Programms entwickelt, kompiliert und gegen andere Module gleicher Funktionalität ausgetauscht werden können. Arbeiten mehrere Personen an einem größeren Softwareprojekt zusammen, so braucht jeder nur die Schnittstellen der anderen Module zu kennen und sein eigenes Modul darauf abstimmen. Weiterhin können die Module im Laufe der Entwicklungszeit immer wieder durch die jeweils neueste Version ersetzt werden, ohne dass die anderen Programmierer davon beeinträchtigt werden. Die Schnittstellen eines solchen Moduls dürfen sich während der Entwicklung aber nicht ändern, deshalb sind im COM die Konventionen für solche Modulschnittstellen definiert.

Oft werden Neuerungen oder Erweiterungen mit sogenannten Patches oder Updates zum Programm dazuinstalliert. Dies hat aber den Nachteil, das entweder ganze Dateien von mitunter mehreren Megabyte Größe komplett ersetzt oder Quelltextabschnitte neu kompiliert werden müssen.

COM steuert dem entgegen, indem es einen fest definierten Rahmen für ein Softwaremodul und dessen Schnittstellenentwurf vorgibt. Ein als COM-Objekt entwickeltes Softwaremodul kann so leicht durch ein neueres ersetzt werden, denn die Schnittstellen dieses Objektes bleiben unverändert. Voraussetzung dafür und deshalb auch Bedingung für ein COM-kompatibles Modul ist allerdings, dass die Funktionen des älteren Moduls im neueren auch mit integriert sind. Bestes Beispiel dafür ist DirectX selbst, denn es enthält in Version 8.0 noch immer alle Funktionen und Schnittstellen, die seit DirectX1.0 programmiert wurden. Abbildung 3.2.A illustriert die Wiederverwendbarkeit von COM-basierten Softwarekomponenten an einem Beispiel mit drei COM-Objekten:

Abbildung 3.2.A: Austausch von COM-Objekten an einem Beispiel



Ein COM-Objekt ist eine Klasse oder eine Sammlung von Klassen, die zumeist als Dynamic Link Library (.dll-Datei) verpackt ist. Es lässt sich in jeder Programmiersprache erstellen, vorausgesetzt, der Compiler erzeugt dasselbe Binärfile wie der C-Compiler von Microsoft. Viele Compiler bieten aber spezielle Optionen, um ein COM-Objekt zu

erzeugen. Zudem sind COM-Objekte auch auf dem MAC oder unter Linux einsetzbar, sofern sie die COM-Normen erfüllen.

Ein COM-Objekt kann ohne weitere Kompilierung an die Stelle der älteren COM-Variante (.dll-Datei) kopiert werden und über dessen festdefinierte Schnittstellen lässt sich mit dem COM-Objekt genauso kommunizieren wie mit der älteren Version dieser Komponente.

Abbildung 3.2.A zeigt ein einzelnes COM-Objekt mit drei Schnittstellen: IGRAPHICS, ISOUND und IINPUT. Über jede dieser Schnittstellen sind eine Reihe von Funktionen aufrufbar, die im COM-Objekt definiert worden sind. Ein COM-Objekt kann mehrere Schnittstellen haben und ein Programmierer kann auch mehrere COM-Objekte in seine Applikation integrieren. Überdies legt die COM-Spezifikation fest, dass alle COM-Objekt-Schnittstellen vom Basis-Klassen-Interface IUnknown abgeleitet sein müssen. Listing 3.2.A zeigt den Prototypen der Struktur IUnknown, die Abbildung 3.2.B die Schnittstellen des in Abbildung 3.2.A dargestellten Beispiels:

Listing 3.2.A: Die Struktur des Basis-Klassen-Interface IUnknown

```
struct IUnknown {
  virtual HRESULT __stdcall QueryInterface(const IID &iid,
                                           (void **)ip) = 0;
  virtual ULONG __stdcall AddRef() = 0;
  virtual ULONG __stdcall Release() = 0;
};
```

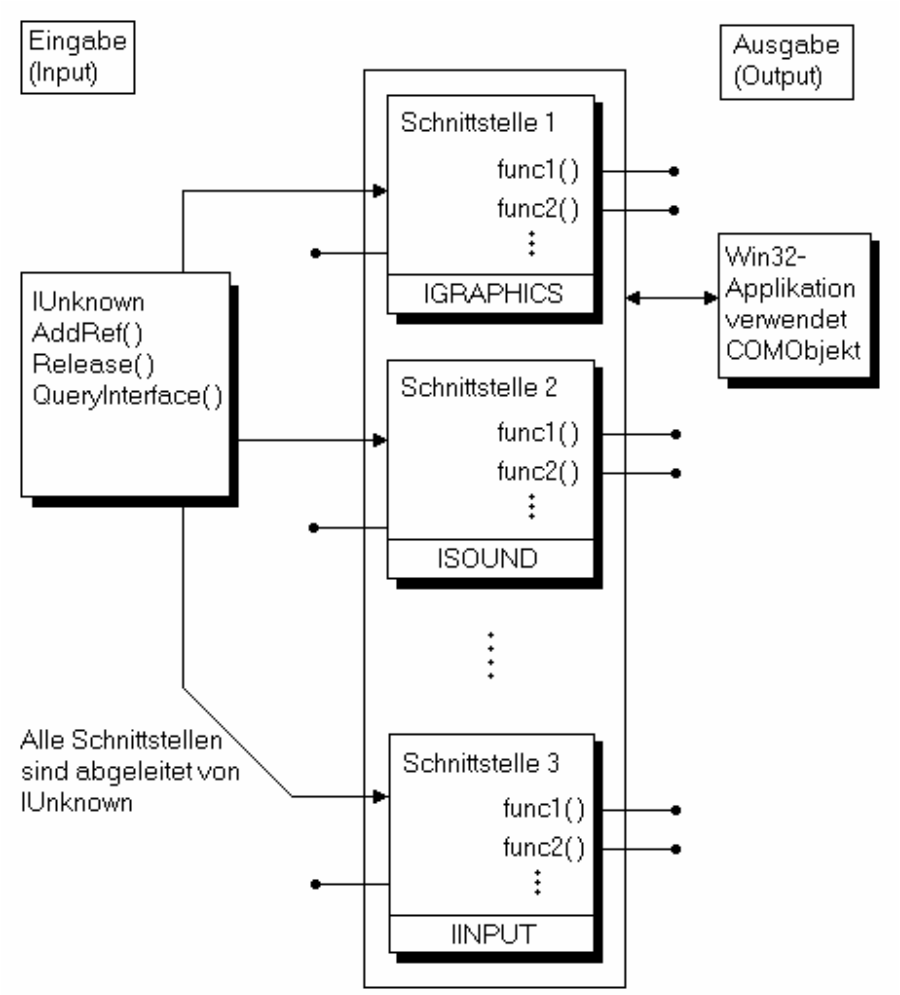
Alle Schnittstellen, die von IUnknown abgeleitet wurden, müssen mindestens die Funktionen QueryInterface(), AddRef() und Release() enthalten.

QueryInterface() stellt einen Pointer zur Schnittstelle bereit. Jede COM-Schnittstelle hat eine eigene 128-Bit-lange Identifikationsnummer, die InterfaceID oder auch GUID, über die mittels des Pointers auf die Funktionen innerhalb des COM-Objektes zugegriffen werden kann. Sie wird mit dem Microsoftprogramm GUIDGEN.EXE generiert, wobei ein bestimmter Algorithmus im Programm dafür sorgt, dass niemals eine Schnittstellen-ID zweimal erstellt wird. Zudem lässt sich mit dem Pointer einer Schnittstelle auf jede andere Schnittstelle zugreifen, sofern diese zum selben COM-Objekt gehört.

AddRef() wird jedes Mal dann aufgerufen, wenn ein COM-Objekt oder eine Schnittstelle zu diesem erstellt wird. Bei jedem Aufruf erhöht es den internen Referenzzähler um 1, wodurch sich überwachen lässt, wie viele Referenzen es auf das COM-Objekt gibt.

Release() ist das Gegenstück zu AddRef(). Es dekrementiert den Referenzzähler um 1, sobald der Programmierer Release () aufruft. Fällt der Zähler auf 0, werden die Referenzobjekte intern freigegeben bzw. zerstört.

Abbildung 3.2.B: Die Schnittstellen eines COM-Objekts

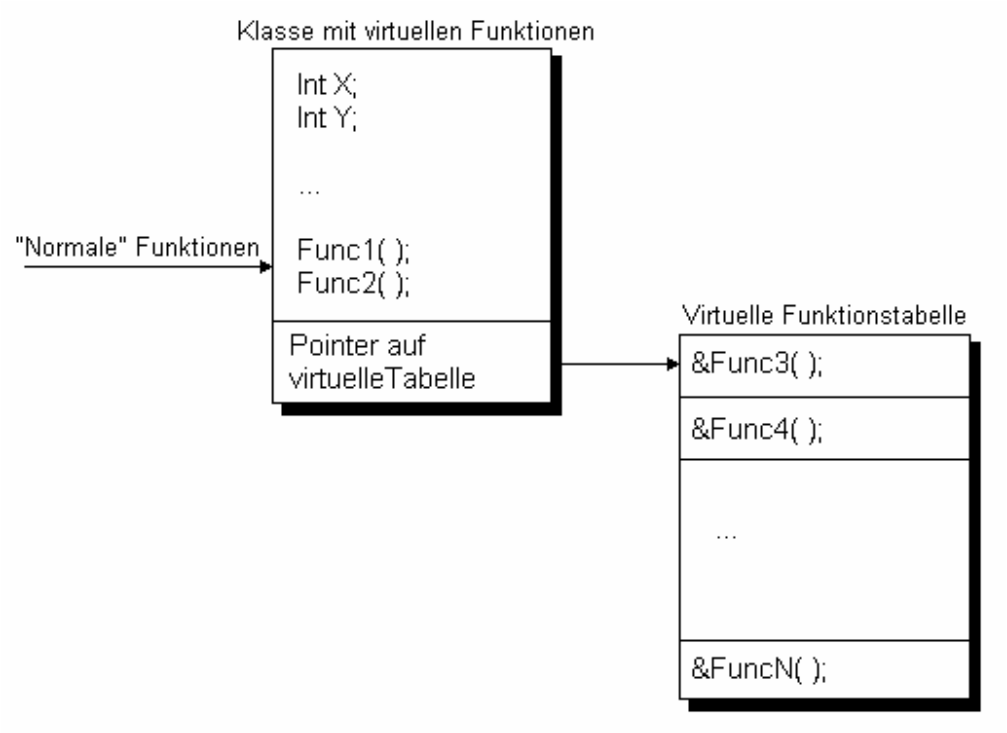


DirectX basiert nun auf einer Reihe derartiger COM-Objekte. Diese sind als .dlls im System enthalten, sobald DirectX installiert wird. Microsoft hat allerdings DirectX so angelegt, dass der Anwendungsprogrammierer nur recht wenig mit COM-Objekten in Berührung kommt. Fast jede Kommunikation zwischen DirectX und seinen Basis-COM-Objekten geschieht über die DirectX-Komponenten-Bibliotheken DDRAW.LIB, DSOUND, DINPUT, DINPUT8, DSETUP, DPLAYX, D3DIM, D3DRM, die je nach Bedarf in ein Projekt mit eingebunden werden müssen.

Wenn ein COM-Objekt erstellt wurde und es einen Schnittstellenpointer zurückgibt, so ist das eigentlich ein VTable-Pointer, also ein Zeiger auf eine Virtuelle Funktionstabelle (siehe Abb. 3.2.C).

Virtuelle Funktionen werden so benutzt, dass sie bis zur Laufzeit eines Programms noch nicht fest an Werte bzw. Aufrufe gebunden sind. Die eigentlichen Funktionen werden erst zur Laufzeit mittels sogenannter Funktionspointer den virtuellen Funktionen zugewiesen.

Abbildung 3.2.C: Die Architektur virtueller Funktionstabellen



Ein Funktionspointer ist ein bestimmter Typ von Zeiger, der gebraucht wird, um Funktionen aufzurufen. Die Funktionen können fest an einen bestimmten Code gebunden sein, es ist aber auch möglich, mit nur einem Funktionspointer auf verschiedene Funktionen zuzugreifen, sofern diese den gleichen Funktionsprototyp haben. Das nun folgende Beispiel wird diesen Mechanismus noch etwas näher beleuchten.

Gegeben sei dazu eine Funktion *SetPixel()*, die einen einzelnen Bildpunkt auf dem Monitor darstellen soll. Da Grafikkarten unterschiedlicher Marken und Hersteller unterschiedlich angesprochen werden müssen und gewisse Funktionalitäten auf verschiedene Art und Weise umsetzen, ist es notwendig, *SetPixel()* für jede Grafikkarte einzeln anzupassen. Listing 3.2.B zeigt, wie dafür der Quellcode aussehen kann:

Listing 3.2.B: Die Funktion *SetPixel()* als fest implementierte Funktion

```
int SetPixel(int x, int y, int color, int card){
  switch(card){      //Test des Grafikkartentyps
    case ATI: {
                /*Hardware-spezifischer Code*/
    } break;
    case VOODOO: {
                /*Hardware-spezifischer Code*/
    } break;
    case SIII: {
                /*Hardware-spezifischer Code*/
    } break;
```

```
    ...
  }
}
```

Diese Variante ist sehr langsam, fehleranfällig und gerade für geschwindigkeitsabhängige Programme absolut ungeeignet. Zum einen wird bei jedem Aufruf von *SetPixel()* von neuem geprüft, welche Grafikkarte Benutzung findet, obwohl eine einzige diesbzgl. Abfrage reichen würde. Des Weiteren wird alles in einer Funktion definiert, was bedeutet, dass jede neu einzubindende Grafikkarte im Programm-Quelltext anstatt als dll-Modul zu integrieren ist, was bei jedem Update ein nochmaliges Kompilieren zur Folge hat. Eine bessere Lösung ist an dieser Stelle der Einsatz eines Funktionspointers (siehe Listing 3.2.C)

Listing 3.2.C: Zugriff auf spezielle Grafikkartenfunktionen mittels Funktionspointer

```
//SetPixel() als Funktionspointer
int (* SetPixel) (int x, int y, int color);

//spezifische SetPixel-Funktionen
int SetPixel_ATI (int x, int y, int color){
    // Quellcode für ATI-Karte
}


int SetPixel_VOODOO (int x, int y, int color){
    // Quellcode für VOODOO-Karte
}

int SetPixel_SIII (int x, int y, int color){
    // Quellcode für SIII-Karte
}
```

Wenn das Programm gestartet wird, ist nur ein einziges Mal zu prüfen, welche Grafikkarte sich im System befindet und mit dem Aufruf

```
SetPixel = SetPixel_ATI;
```

wird zum Beispiel die ATI-spezifische Funktion für weitere SetPixel-Aufrufe ausgewählt. Würde nun
SetPixel(10,20,4); aufgerufen werden, so wird in Wirklichkeit *SetPixel_ATI(10,20,4);* aufgerufen. Der Punkt ist der, dass der Quellcode auf diese Weise immer gleich aussieht, obwohl bei unterschiedlicher Systemkonfiguration eigentlich komplett andere Funktionen aufgerufen werden. Aufgrund dieser Technologie und des vorausblickenden Designs der Entwickler ist DirectX in der Lage, sämtliche Hardware zu erkennen und allgemeinen Funktionspointern die jeweiligen hardwarespezifischen Routinen zuzuweisen, so dass sich der Programmierer nicht mehr um die Systemkonfiguration zu kümmern braucht. Und genau das ist die große Stärke von DirectX.

3.3. Die Pocket-PC-Funktionsbibliothek GameAPI

DirectX wird vom Betriebssystem Windows CE in seinem vollen Umfang unterstützt, es ist jedoch viel zu umfangreich für die Möglichkeiten eines Pocket-PCs. Deshalb entschloss sich Microsoft, für diese Geräte-Klasse eine eigene Lösung als Alternative zum GDI (siehe Abschnitt 3.1) anzubieten: Die GameAPI.

Die GameAPI ist mit der ersten Version von DirectX vergleichbar und kommt vor allem den Spieleprogrammieren entgegen, denn gerade in dieser Branche sind schnelle, hardwarenahe Funktionen von großer Wichtigkeit und Möglichkeiten wie der Direktzugriff auf den Videospeicher unerlässlich. Jedoch finden Features wie die 3D-Programmierung in der GameAPI keine Verwendung, weil Pocket-PCs diese weder hardware- noch softwaretechnisch unterstützen.

In der aktuellen Version 1.2. beinhaltet die GameAPI nur zwölf Funktionen, (Zum Vergleich: DirectX bietet ca. 1200 Funktionen) doch ist zu erwarten, dass einige der Features von DirectX in zukünftigen Versionen verfügbar sein werden.

Kapitel 4: Windows-Programmierung - Die ersten Schritte

4.1. Hello World - Das erste Programm

In der Literatur hat es sich durchgesetzt, zum Einstieg in eine neue Programmiersprache mit einer sogenannten "Hello-World"-Anwendung zu beginnen, die lediglich eine Zeichenkette (wie z.B.: "Hello World!") auf dem Bildschirm ausgibt. Listing 4.1.A zeigt den C-Quelltext eines solchen Programms:

Listing 4.1.A: Quelltext einer Standard-DOS-"Hello-World"-Applikation

```
//Headerdatei für DOS Eingabe/Ausgabefunktionen
#include <stdio.h>

void main() {
  printf("\nHello World!\n");
}
```

Die Funktion *main()* ist dabei der Standard-Eintrittspunkt in ein DOS/Konsolen-Programm. Wie der Quelltext unter Windows aussieht, zeigt das Listing 4.1.B. Da unter Windows die Konsole nur noch selten Verwendung findet, ist hier der *print*-Befehl durch die Funktion *MessageBox()* ersetzt worden, die ein kleines Nachrichtenfenster mit der Textausgabe "Hello World!" und einem Okay-Button öffnet.

Listing 4.1.B: Quelltext einer Standard-Windows-Hello-World-Applikation

```
#define WIN32_LEAN_AND_MEAN   //Compiler exkludiert MFC-
Klassenheader

#include <windows.h>          //Einbinden der Fenster-Bibliothek
#include <windowsx.h>         //Einbinden weiterer Fenster-
                              //eigenschaften

int WINAPI WinMain(HINSTANCE hinstance,
                   HINSTANCE hprevinstance,
                   LPSTR lpcmdline,
                   int ncmdshow){

  MessageBox(NULL,
             "Windows -HelloWorld!- application",
             "Hello World!",
             MB_OK | MB_ICONEXCLAMATION);

  //Generierung des Rückgabewertes (optional)
  return(0);
} // end WinMain
```

Das Programm in Listing 4.1.B zeigt das Äquivalent zu der in Listing 4.1.A beschriebenen Applikation. Allerdings wirkt es komplizierter und übernimmt zudem auch einige Parameter, doch muss dem Rechnung getragen werden, dass das Erzeugen

eines Windows-Fensters deutlich komplexer ist als das Abbilden von weißen Standard-Lettern auf einem schwarzen Bildschirm.

Ein Windows-Fenster basiert auf mehreren Basisklassen, die bestimmen, wie sein Rahmen definiert ist, welche Ausmaße und Farbgebung ihm eigen sein sollen, wie er bei gewissen Ereignissen reagieren soll und vieles mehr. Durch die beiden Header "windows.h" bzw. "windowsx.h" werden die Windows-Basisklassen in das Programm eingebunden. Im Gegensatz dazu sollen die Basisklassen für eine typische MFC-Anwendung nicht mit ins Programm übernommen werden, denn diese werden nur bei Applikationen gebraucht, die die vordefinierten Windows-internen Dialogfelder verwenden. Ist das Macro WIN32_LEAN_AND_MEAN im Quelltext eingefügt, wird der Compiler gezwungen, die MFC-Klassen-Header beim Compilieren zu überspringen.

Die Windows-Funktion *WinMain()* ist der Einstiegspunkt eines Windows-Programms, dass heißt, sie ist die erste Funktion, die aufgerufen wird. Sie sollte vom Typ WINAPI sein, da ihre Parameter von links nach rechts ausgewertet werden müssen, anstatt in der normalen "rechts-nach-links-Ordnung". Ihr zusätzlich noch den Typ integer zuzuweisen, hat den Vorteil, über den return-Wert, den eine Nicht-void-Funktion zurückgibt, festzustellen, ob diese erfolgreich ausgeführt werden konnte.

Doch nun zu den Parametern der Funktion WinMain:

hinstance:
- Dieser Parameter ist das Instanzenhandle, dass Windows für diese Applikation generiert. Instanzen sind Pointer oder Zahlen, die genutzt werden, um Ressourcen zu verfolgen und zu verwalten. In diesem Fall wird hinstance dazu verwendet, einen Bezugspunkt auf die Anwendung zu haben, ähnlich einem Namen oder einer Adresse, über den Windows auf das Programm zugreifen kann.

prevhinstance:
- In previnstance wurde ursprünglich die vorhergehende Applikation gespeichert, also dass Programm, welches das derzeit laufende gestartet hat. Auf die Nutzung dieses Parameters wird aber weitgehend verzichtet, da er sich in vielen Fällen als zu umständlich und eher überflüssig erwiesen hat.

lpcmdline:
- Dies ist ein Null-terminierter String, ähnlich den Kommando-Zeilen-Parametern einer Standard-C/C++- main(int argc, char **argv) Funktion, mit der Ausnahme, dass es keinen separaten Parameter gibt, der die Anzahl der Argumente speichert. Als Beispiel soll hier eine Anwendung PARATEST.EXE dienen, die mit folgenden Parametern aufgerufen wird:

PARATEST.EXE pa1 pa2 pa3

lpcmdline würde dann folgenden String enthalten:

"pa1 pa2 pa3"

Sie speichert also nur die Parameter, inkludiert aber nicht die PARATEST.EXE.

ncmdshow:
- Der letzte Parameter ist ein einfacher Integer-Wert, der angibt, wie die Anwendung geöffnet wird. Die Tabelle 4.1.A zeigt einige mögliche Werte, die dieser Parameter annehmen kann:

Tabelle 4.1.A: Flags der Funktion *WinMain()* zum Festlegen des Darstellungsmodus für ein Fenster

Wert	Beschreibung
SW_SHOWNORMAL	Aktiviert ein Fenster und stellt es in seiner ursprünglich definierten Größe dar.
SW_SHOW	Aktiviert ein Fenster und stellt es in seiner derzeit aktuellen Größe und Position dar.
SW_SHOWNOACTIVATE	Stellt ein Fenster in seiner derzeit aktuellen Größe und Position dar, jedoch bleibt das vorher aktuelle Fenster aktiviert.
SW_SHOWMINIMZIED	Aktiviert ein Fenster und stellt es im minimierten Modus dar.
SW_SHOWMAXIMIZED	Aktiviert ein Fenster und stellt es im maximierten Modus dar.

Alle weiteren Möglichkeiten sind in der Hilfe-Dokumentation des Microsoft Visual Studios unter dem Punkt *MessageBox* zu finden.

4.2. Fenster - Eigenschaften der Basiselemente unter Windows

Während DOS-Nutzern nur die Konsole als Eingabeoberfläche zur Verfügung stand, verfolgt Windows ein komplett anderes Konzept. Hier bildet ein Fenstersystem die Grundlage der gesamten Windows-Oberfläche. Ein einzelnes Fenster ist dabei nicht nur als der typische Windows-Rahmen zu sehen, in dem eine Applikation läuft. Es ist viel mehr ein Arbeitsbereich (workspace), der Informationen wie Text und Grafik darstellt, mit denen ein Anwender interagieren kann. So gelten auch Buttons, Listenboxen, Textfelder, Menüs etc. als jeweils eigenständiges "Fenster".

Zum Erstellen eines einfachen Fensters sind fünf Hauptschritte nötig:
1. Erstellen einer Windows-Klasse
2. Registrieren dieser Klasse in Windows
3. Erstellen eines Fensters basierend auf der unter Punkt 1 erstellten Klasse
4. Erstellen eines Eventhandlers
5. Erstellen eines Main Event Loop (Programmschleife,Ereignistestzyklus:siehe Kapitel 2), der Windows Messages (Events) abfragt und an den Event Handler verschickt.

4.3. Fenstererstellung für Windows-Desktop-PCs:
4.3.1. Die Windows-Klasse (Windows class)

Windows ist ein objektorientiertes Betriebssystem. Viele seiner Konzepte und Prozeduren haben ihre Wurzeln im C++. Eines dieser Konzepte ist "Windows classes". Alle Fenster, Kontrollboxen, Listboxen, Dialogboxen, Menüs, Geräte usw. sind tatsächlich Fenster, die sich nur in einem unterscheiden - ihrer Windows-Klasse. Eine Windows-Klasse ist die Beschreibung eines Fenster-Typs, vergleichbar mit einem Bauplan, nach dem Windows ein Fenster zusammensetzt.

Es existieren bereits einige vordefinierte Windows-Klassen wie Buttons, Listboxen, Farb- oder Dateidialoge, es steht dem Programmierer aber auch frei, weitere zu erstellen. Tatsächlich ist für jedes neue Programm, das nicht auf MFC basiert, mindestens eine neue Windows-Klasse anzulegen.

Es sind zwei Datenstrukturen verfügbar, die die *Windows class* -Informationen aufnehmen können. Zum einen ist das die WNDCLASS und zum anderen die WNDCLASSEX (windows class extended). Beide sind im Aufbau sehr ähnlich, jedoch enthält die WNDCLASSEX zwei Einträge mehr als die WNDCLASS-Struktur, die Parameter *cbSize* und *hIconSm*. Listing 4.3.A zeigt die Struktur der WNDCLASSEX, wie sie in den Windows-Header-Dateien definiert ist:

Listing 4.3.1.A: Die Struktur der WNDCLASSEX

```
typedef struct _WNDCLASSEX {
    UINT    cbSize;
    UINT    style;
    WNDPROC lpfnWndProc;
    int     cbClsExtra;
    int     cbWndExtra;
    HANDLE  hInstance;
    HICON   hIcon;
    HCURSOR hCursor;
    HBRUSH  hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
    HICON   hIconSm;
} WNDCLASSEX;
```

Um nun eine Windows-Klasse anzulegen, wird eine Instanz dieser Struktur, etwa

```
WNDCLASSEX winclass;
```

erstellt. Im Anschluss sind alle Eigenschaften dieser Klasse zu definieren.

Der erste Parameter beschreibt die Größe der WINCLASSEX Struktur und wird standardmäßig auf

```
winclass.cbSize = sizeof(WNDCLASSEX);
```

gesetzt. Der Compiler erhält so schon im Vorfeld die Angaben zur Größe der Struktur und kann für sie entsprechend Speicherplatz reservieren.

Das nächste Feld beinhaltet ein Style Flag, dass die generellen Eigenschaften des Fensters beschreibt. Es lassen sich mehrere Eigenschaften mittels logischem OR verknüpfen, so dass jeder erdenkliche Fenstertyp damit erschaffen werden kann. Tabelle 4.3.1.A zeigt die wichtigsten Flags.

Tabelle 4.3.1.A: Style Flags der Windows-Klasse

Wert	Beschreibung
CS_HREDRAW	Zeichnet das gesamte Fenster neu, wenn eine Bewegung oder Größen-anpassung die Breite des Fensters ändert.
CS_VREDRAW	Zeichnet das gesamte Fenster neu, wenn eine Bewegung oder Größen-anpassung die Länge des Fensters ändert.
CS_OWNDC	Legt fest, dass jedes auf der Windowsklasse basierende Fenster seinen eigenen Geräte-kontext (Device Context) bekommen soll.*
CS_DBLCLKS	Sendet ein Doppelklick-Ereignis an die Windows-Prozedur (Callback-Funktion), sobald der User mit der Maus doppelklickt, während er sich in dem Fenster befindet, zu dem diese Klasse gehört.

Die MSVisualC++-Hilfe enthält eine Übersicht zu allen weiteren Style-Flags unter dem Punkt WNDCLASSEX. Gesetzt werden sie folgendermaßen:

* Ein Geräte-Kontext ist eine Struktur, die ein Set von Grafikobjekten und ihre assoziierten Attribute definiert und den Grafikmodus, der die Ausgabe bestimmt. Die grafischen Objekte beinhalten einen Zeichenstift (pen) zum Linienzeichnen, eine Bürste (brush) zum Flächen zeichnen bzw. füllen, ein Bitmap, um Teile des Bildschirms zu kopieren oder zu scrollen, eine Palette, um ein Set verfügbarer Farben zu definieren, eine Region für Bildschirm-Clipping und andere Operationen, und einen Pfad für Mal- und Zeichenoperationen.

```
winclass.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC | CS_DBLCLKS;
```

Im dritten Punkt ist der Funktionspointer für die Callback - Funktion des Fensters anzugeben. Diese Funktion wird im Abschnitt "Der Windows-Eventhandler" noch genau beschrieben.

An dieser Stelle soll die Zuweisung

```
winclass.lpfnWndProc = WinProc;
```
aber genügen.

Die nächsten beiden Felder wurden ursprünglich angelegt, um in der Klasse noch Platz für zusätzliche Laufzeitinformationen unterzubringen. Da dieses Feature aber kaum genutzt wird, soll auch hier nur der Standard-Wert gesetzt werden:

```
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
```

Das nächste in der Reihenfolge ist das hInstance-Feld. Es ist die selbe aus der Winmain() bekannte hinstance, die die Instanz bzw. den Identifier des Fensters enthält und so übernommen werden kann:
```
winclass.hInstance = hinstance;
```

Die folgenden Felder beziehen sich auf hauptsächlich grafische Aspekte des Fensters, wie
- das Anwendungs-Icon:

```
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

lädt das Standard-Icon für Anwendungen. (Näheres dazu im Abschnitt Ressourcen)

- der Cursur:

winclass.hCursor = LoadCursor(NULL, IDC_ARROW);

lädt den Standard-Cursor. (Näheres dazu im Abschnitt Ressourcen)

- die Farbe des Hintergrundes:

```
winclass.hbrBackground = (HBRUSH)GetstockObject(BLACK_BRUSH)
```

weist dem Hintergrund die Farbe schwarz zu.
Wenn ein Fenster neu gezeichnet oder aktualisiert wird, zeichnet Windows als erstes das gesamte Fenster in der mittels hbrBackground vordefinierten Farbe. Detailiert betrachtet wird hier eigentlich das Werkzeug auswählt, mit dem gearbeitet werden soll, in diesem Fall z.B. eine Füllbürste (BRUSH vom Typ HBRUSH), die mit schwarz (BLACK_BRUSH) den Hintergrund zeichnet. Statt BLACK_BRUSH sind auch noch die in Tabelle 4.3.1.B verzeichneten Eintrage möglich:

Tabelle 4.3.1.B: Übersicht möglicher zuweisbarer Hintergrundfarbenflags

Wert	Beschreibung
WHITE_BRUSH	weisse Bürste
GRAY_BRUSH	graue Bürste
LTGRAY_BRUSH	hellgraue Bürste
DKGRAY_BRUSH	dunkelgraue Bürste

Weitere lassen sich in der MS-VisualC++-Hilfe finden, sind aber für ein Fenster nicht sinnvoll.

- das Menü des Fensters:

```
winclass.lpszMenuName = NULL;
```

gibt an, dass dieses Fenster kein Menü zugewiesen bekommen soll. (Näheres dazu im Abschnitt Ressourcen)

- der Name der Windows-Klasse:

```
winclass.lpszClassName = "WINCLASS1";
```

weist dem Fenster den Namen WINCLASS1 zu.

- das Titelleisten-Icon:

```
winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
```

lädt ein kleines Icon (hier wieder das Standard-Icon), dass in der Titelleiste des Fensters oder evtl. als Icon in der Taskleiste wieder auftaucht. (Näheres dazu im Abschnitt Ressourcen)

Nun, da die Windows-Klasse in *winclass* definiert und gespeichert wurde, muss Windows mitgeteilt werden, dass eine solche Klasse existiert. Dies wird mit der Funktion *RegisterClassEx()* erreicht, der ein Pointer auf *winclass* als Parameter zu übergeben ist:

```
RegisterClassEx(&winclass);
```

Wurde die Struktur WNDCLASS anstelle der neueren WNDCLASSEX verwendet, so ist zur Klassenregistrierung die Funktion *RegisterClass()* mit dem gleichen Parameter aufzurufen.

4.3.2. Das Erstellen des eigentlichen Fensters

Nachdem die Eigenschaften des Fensters festgelegt sind, und die Windows-Klasse registriert ist, muss dass Fenster nun noch angelegt werden. Das wird mit der Funktion

```
HWND CreateWindowEx(
```

```
  DWORD dwExStyle,          // erweitereter Fenster-Style
  LPCTSTR lpClassName,      // Pointer auf registrierte Klasse
  LPCTSTR lpWindowName,     // Pointer auf den Fensternamen
  DWORD dwStyle,            // Fenster-Style
  int x,                    // horizontale Position des Fensters
  int y,                    // vertikale Position des Fensters
  int nWidth,               // Fensterbreite
  int nHeight,              // Fensterhöhe
  HWND hWndParent,          // Adresse des übergeordneten Fensters
  HMENU hMenu,              // Adresse eines untergeordneten
                            // Fensters oder eines Menüs
  HINSTANCE hInstance,      // Adresse der Applikationsinstanz
  LPVOID lpParam            // Pointer auf CREATESTRUCT,
                            // NULL setzen
);
```

realisiert, die folgende Parameter benötigt:
- dwExStyle: Dies ist ein erweiterter Style Flag, mit dem sich einige Stil-Eigenschaften des Fensters wie zum Beispiel die Ausrichtung des Fensterinhaltes oder das Vorhandensein und Verhalten von Scrollbars festlegen lassen. Für eine Anwendung im Vollbildmodus wird dieser Parameter aber auf NULL gesetzt.
- lpClassName: Das ist der Name der Windows-Klasse, auf der das zu erstellende Fenster basieren soll. Angelehnt am Beispiel oben wäre dass "WINCLASS1".
- lpWindowName: Mit diesem Parameter wird dem Fenster ein Titel (nullterminierter String) zugeordnet.
- dwStyle: Dies ist der allgemeine Style Flag, mit dem generelle Eigenschaften des Fensters festgelegt werden. Tabelle 4.3.2.A zeigt eine Auswahl gültiger Werte für diesem Parameter, die auch mittels logischem OR verknüpft werden können. Alle weiteren sind in der MSVisualC++-Hilfe zu finden.

Tabelle 4.3.2.A: Übersicht über die Style-Parameter der Funktionen CreateWindow() und CreateWindowEx()

Wert	Beschreibung
WS_POPUP	Ein Popup-Fenster.
WS_OVERLAPPED	Ein überlappendes Fenster, dass eine Titelleiste und ein einen Rahmen (border) aufweist.
WS_VISIBLE	Ein Fenster, dass anfänglich als sichtbar deklariert ist.
WS_BORDER	Ein Fenster, dass einen Rahmen (border) aufweist.
WS_CAPTION	Ein Fenster, dass eine Titelleiste aufweist. Beinhaltet WS_BORDER.
WS_SYSMENU	Ein Fenster, dass ein Windows-Menü aufweist. WS_CAPTION muss gesetzt sein.
WS_THICKFRAME	Ein Fenster, dass eine in Länge und Breite verstellbare Größe aufweist.
WS_MINIMIZEBOX	Ein Fenster, dass in der Titelleiste ein Minimiersymbol beherbergt.
WS_MAXIMIZEBOX	Ein Fenster, dass in der Titelleiste ein Maximiersymbol beherbergt. WS_SYSMENU muss gesetzt sein.
WS_OVERLAPPEDWINDOW	Vereinigt die Flags WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX und WS_MAXIMIZEBOX.

- x, y: Diese Variablen sind die Positionskoordinaten des Fensters, also der Pixel, auf dem die linke obere Ecke des Fensters liegt.
- nWidth, nHeight Diese zwei Werte geben an, wie viele Pixel lang bzw. breit das Fenster sein soll.
- hWndParent: Hier erwartet die Funktion ein Handle zu seinem übergeordneten Fenster (Parent Window). Fall kein "Parent Window" existiert, ist dieser Wert NULL.
- hMenu: Hier erwartet die Funktion ein Handle zu einem zum Fenster gehörenden Menü. Fall kein Menü existiert, ist dieser Wert NULL.
- hInstance: Dies ist die Instanz der Anwendung. Der Wert hinstance aus der Windows-Klasse kann an dieser Stelle eingesetzt werden.
- lpParam: Das ist ein Pointer auf eine CreateStruct-Struktur, die Informationen über das Fenster enthält, das erzeugt werden soll. Die Parameter von CREATESTRUCT sind identisch mit denen der Funktion *CreateWindowEx()*. Der Parameter *lpParam* wird standardmäßig auf NULL gesetzt.

Listing 4.3.2.A demonstriert den Aufruf der Funktion an einem Beispiel:

Listing 4.3.2.A: Aufruf der Funktion *CreateWindowEx()* an einem Beispiel

```
hwnd = CreateWindowEx(NULL,
                      "WINCLASS1",
                      "WINDOWSTITLE",
                       WS_OVERLAPPEDWINDOW,WS_VISIBLE,
```

```
                        0,0,
                        400,400,
                        NULL,
                        NULL,
                        hinstance,
                        NULL
);
```

4.3.3. Der Windows-Eventhandler

Der Event Handler (wörtlich: Ereignisbehandler /-auswerter) ist eine Funktion, die immer dann aufgerufen wird, wenn ein Ereignis (Event) auftritt, auf das das Fenster reagieren muss. Dem Programmierer obliegt es nun, ob und in welcher Weise seine Applikation auf solche Ereignisse reagieren soll. Ein unbehandeltes Ereignis wird an das Betriebssystem weitergeleitet, das dann die Nachrichten nach eigenem Ermessen auswertet.

Im Detail sieht das folgendermaßen aus: Wenn ein Nutzer oder Windows Prozesse anstößt, werden durch diese Prozesse Ereignisse (Events) bzw. Nachrichten (Messages) generiert. Alle diese Nachrichten werden auf einem Ereignisstapel (queue) abgelegt, die ein bestimmtes Fenster betreffenden auf einem fenstereigenen Stapel. Der "Main Event Loop" eines Fensters liest diese Nachrichten dann aus und leitet sie an den Eventhandler des Fensters weiter.

Jede Windows-Klasse, die erstellt wird, kann eine eigene ereignisbehandelnde Routine (Eventhandler) haben. Diese soll im folgenden Windows' Procedure, kurz WinProc genannt werden. Listing 4.3.3.A zeigt den Prototypen der WinProc:

Listing 4.3.3.A: Prototyp der Callback-Funktion WinProc

```
LRESULT CALLBACK WinProc(
  HWND hwnd,          //"Windows handle" -
                      // Zugriffsname für das Fenster
  UINT msg,           // Ereignis-ID
  WPARAM wparam,      // Parameter, die bestimmte Eigenschaften
  LPARAM lparam       // eines Ereignisses speichern,
                      // z.B. Wert einer gedrückten Taste
);
```

Die Funktion benötigt folgende Parameter:
hwnd - Das ist das handle des Fensters.
msg - Hier wird die Kennung der Message, also deren ID übergeben.
wparam und lparam - Diese Parameter qualifizieren bzw. klassifizieren die Message, die in *msg* gesendet wurde, noch weiter.
Es existieren eine Reihe vordefinierter Ereignisse, die entweder vom System, durch eine Nutzereingabe oder durch ein Programm generiert werden. Tabelle 4.3.3.A zeigt einige wichtige Ereignisse:

Tabelle 4.3.3.A: Auswahl möglicher System- und Nutzerereignisse

Message ID	Beschreibung
WM_ACTIVATE	Wird gesendet, wenn ein Fenster aktiviert wird bzw. den Fokus Fokus bekommt.
WM_CREATE	Wird gesendet, wenn ein Fenster das erste Mal erstellt wird.
WM_MOVE	Wird gesendet, wenn ein Fenster bewegt wurde.
WM_SIZE	Wird gesendet, wenn ein Fenster in seiner Größe verändert wurde.
WM_PAINT	Wird gesendet, wenn ein Fenster neu gezeichnet werden muss (z.B. nach WM_SIZE / WM_MOVE).
WM_CLOSE	Wird gesendet, wenn ein Fenster geschlossen wird.
WM_DESTROY	Wird gesendet, wenn ein Fenster geschlossen und gelöscht wird.
WM_MOUSEMOVE	Wird gesendet, wenn die Maus bewegt wurde.
WM_KEYDOWN	Wird gesendet, wenn eine Taste gedrückt wurde.
WM_KEYUP	Wird gesendet, wenn eine Taste wieder losgelassen wurde.
WM_TIMER	Wird gesendet, wenn ein Timer-Ereignis auftritt, wenn also eine zeitabhängige Funktion den Ablauf einer bestimmten Frist feststellt.
WM_USER	Erlaubt das Senden einer Message zu einem Zeitpunkt oder Ereignis, dass der Nutzer/Programmierer generiert.
WM_QUIT	Wird gesendet, wenn eine Applikation beendet wird.

Listing 4.3.3.B zeigt den prinzipiellen Aufbau einer *WinProc()*-Funktion am Beispiel der Ereignisse WM_CREATE, WM_PAINT, WM_DESTROY.

Listing 4.3.3.B: Die Funktion *WinProc()*

```
LRESULT CALLBACK WinProc(HWND hwnd,UINT msg,
                         WPARAM wparam,LPARAM lparam){
HDC hdc;                // Handle zu einem Gerätekontext
PAINTSTRUCT ps;         //Struktur, die in WM_PAINT gebraucht wird

 switch(msg){            //Auswahl der Nachricht
  case WM_CREATE: {

    //Platz für Aufrufe zur Programminitialisierung

    //Rückgabewert für erfolgreiche Ausführung der Funktion
    return(0);
  } break;
  case WM_PAINT: {
    hdc = BeginPaint(hwnd,&ps);     //Zeichnen Anfang

    EndPaint(hwnd,&ps);             //Zeichnen Ende
    return(0);
  } break;

  case WM_DESTROY: {
    //Eine WM_QUIT-Message wird gesendet
```

```
    PostQuitMessage(0);

    return(0);
   } break;
  default:break;
 } // end switch

//Leitet die Nachrichten weiter, die die nicht in der WinProc()
// behandelt wurden
return (DefWindowProc(hwnd, msg, wparam, lparam));
} // end WinProc
```

Zuerst werden zwei Strukturen definiert, die eine ist der Gerätekontext des Fensters (hdc), die andere eine Zeichenstruktur, die in WM_PAINT gebraucht wird.

Im nächsten Schritt folgt eine Case-Anweisung, die die ankommenden Messages selektiert. In WM_CREATE wird dann definiert, was beim ersten Erstellen des Fensters initialisiert werden soll. Der Case-Zweig gibt in diesem Fall zwar nur ein *return(0)* zurück, doch Windows wurde der automatischen Behandlung von WM_CREATE entledigt und somit ungewollte Initialisierungen von Seiten des Systems unterbunden.

WM_PAINT wird immer dann aufgerufen, wenn ein Fenster aktualisiert bzw. neu gezeichnet werden muss. Das Konstrukt

```
hdc = BeginPaint(hwnd,&ps);       // Zeichnen beginnen
EndPaint(hwnd,&ps);               // Zeichnen beenden
```

füllt den Hintergrund des Fensters mit der in der hbrBackground-Variable der Windows-Klasse festgelegten Hintergrundfarbe und dem entsprechenden Werkzeug. Beiden wird das Handle des betreffenden Fensters (hwnd) übergeben und eine PAINTSTRUCT, die unter anderem ein Rechteck enthält, dass die Größe der neuzuzeichnenden Fläche angibt. Ist zum Beispiel der gesamte Bildschirm neu zu zeichnen, so enthält diese Rechteckstruktur die Eckpunkte des Bildschirmes.

WM_DESTROY wird gesendet, wenn der Nutzer das Fenster schließt. Die Applikation, die das Fenster aufgerufen hat, läuft jedoch weiter. So scheint es in den meisten Fällen geraten, an dieser Stelle eine weitere Nachricht (WM_QUIT) zu senden, die Windows mitteilt, dass die Applikation auch beendet werden soll, sobald ein Nutzer das Fenster schließt. Dafür wird die Funktion *PostQuitMessage(0)* aufgerufen, die eine entsprechende WM_QUIT-Message generiert.

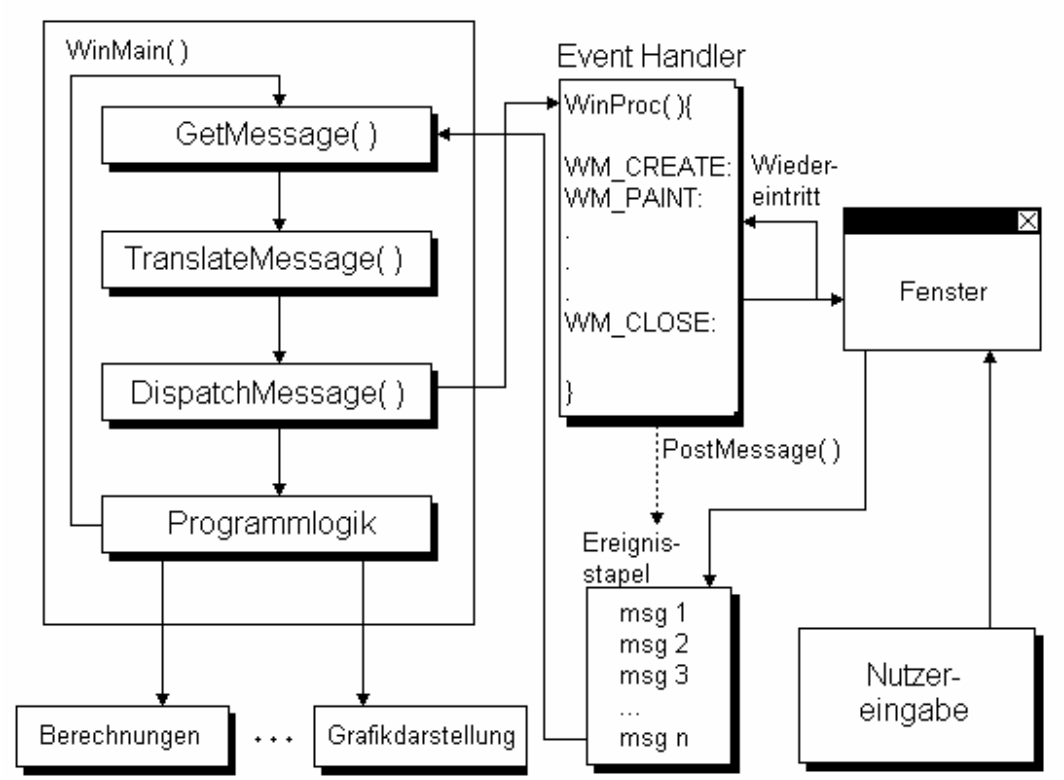
Letztlich wird noch die Funktion *DefWindowProc(hwnd, msg, wparam, lparam)* aufgerufen, die alle nicht in der Funktion *WinProc()* behandelten Ereignisse an das System zur automatischen Auswertung weiterleitet.

4.3.4. Main event loop

Im Main-Event-Loop werden beständig alle das Applikationsfenster betreffenden Nachrichten abgefangen und an die Funktion WinProc weitergeleitet.

Abbildung 4.3.4.A stellt das Prinzip des Event-Loops dar.

Abbildung 4.3.4.A: Das Prinzip des Event-Loops



Programmiertechnisch lässt sich ein solcher Loop folgendermaßen umsetzen:

```
App_init();              //Programm initiieren
while(TRUE) {            //Eintrittspunkt in die Programmschleife

  //Test,ob eine Nachricht im Messagestapel ist,
  //wenn ja, wähle diese aus
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)){

      // Test auf Programmabbruch durch WM_QUIT-Event
      if (msg.message == WM_QUIT){
        break;
      }
      TranslateMessage(&msg); //Übersetzen der Nachricht
      DispatchMessage(&msg);  //Senden der Nachricht an WinProc()
  } // end if
  App_main();
//Hauptprogrammteil
} // end while

//Programm schließen
App_exit();
```

App_init(), *App_main() und App_exit()* sind Funktionen, in denen später die Routinen der Applikation definiert werden. Vorerst noch sind sie noch leere Hüllen nach dem Schema:

```
void App_xxxx(void){
};
```

PeekMessage liest die Messages/Events vom Systemqueue und bekommt folgende Parameter:

```
PeekMessage(
  &msg,             //Adresse der Message, die gerade
                    //oben auf dem Message-Queue liegt
  NULL,             //"Adresse" des Fensters, wo Event auftrat.
                    //NULL, da Message vom System
  0,                //Message Filter Untergrenze,
                    //wähle Nachrichten von 0 an
  0,                //Message Filter Obergrenze,
                    //wähle Nachrichten bis 0
  PM_REMOVE         //Flag, Message soll nach Weiterleitung
gelöscht werden
);
```

TranslateMessage generiert in erster Linie aus einem Tastendruck das zur Taste gehörende ASCII-Zeichen und bekommt die Adresse der Message übergeben.
DispatchMessage übernimmt die Adresse der Message und sendet diese an die Callback - Funktion des Fensters (*WinProc()*) weiter.

Das Beispiel zum Erstellen eines Fensters unter Windows (1stWindow.cpp) und die dazugehörige ausführbare Datei (1stWindow.exe) befinden sich auf der beigelegten CD im Verzeichnis \Kapitel4\

4.3.5. Die Ressourcen eines Fensters

Menüs, Mauscursor und kleine als auch große Icons, die mit einem Fenster assoziiert sind, gelten als Ressourcen des Fensters. Es ist durchaus möglich, in der Windows-Klasse User-spezifische Ressourcen zu integrieren, die dann Teil des Fensters sind. Soll zum Beispiel der selbst entworfene Maus-Cursor (IDC_MYCURSOR) eingebunden werden, so ist die Zeile

```
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
```

aus der Windows-Klasse durch

```
winclass.hCursor = LoadCursor(hinstance,
                              MAKEINTRESOURCE(IDC_MYCURSOR));
```

zu ersetzen. Voraussetzung dazu sind
a) ein Header-File, zum Beispiel resources.h, in dem die ID des Cursors definiert wird

```
#define IDC_MYCURSOR 100     //Cursor bekommt ID 100
```

b) ein Ressourcen-File, zum Beispiel resources.rc.

```
IDC_MYCURSOR    CURSOR    mycursor.cur
```

"IDC_MYCURSOR" sei also eine Ressource vom Typ "CURSOR" mit "mycursor.cur" als Datenquelle.

Um die Icons einer Applikation zu ändern, ist analog zu verfahren.

```
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

wird durch

```
winclass.hIcon = LoadIcon(hinstance,
                          MAKEINTRESOURCE(IDI_MYICON));
```

ersetzt. Ebenso müssen die Icons im Ressourcen-Header

```
(#define IDI_MYICON 200     //Icon bekommt ID 200)
```

und in der Ressourcen-Datei

```
(IDI_MYICON ICON myicon.ico)
```

definiert werden.

Das Einbinden eines Menüs soll hier übersprungen werden, da dieses Feature für das weitere Vorgehen in der vorliegenden Arbeit nicht von Relevanz ist und zudem auch den Rahmen dieser Ausarbeitung sprengen würde. Nur soviel sei gesagt: Mit den Funktionen in Microsoft Visual C++ und Microsoft embedded C lassen sich relativ einfach Ressourcen jeder Art erstellen, das Anlegen dazugehöriger Header- und Ressourcenfiles übernehmen diese Programme automatisch.

4.3.6. Das Öffnen mehrerer Fenster

An dieser Stelle noch ein kleiner Hinweis. Soll ein Programm mehrere Fenster öffnen, so ist zu beachten, in welcher Beziehung diese Fenster zueinander stehen sollen. Dem Programmierer stehen zwei Wege offen. Zum einen lässt sich mit einem erneuten Aufruf der Funktion

```
hwnd = CreateWindowEx(...);
```

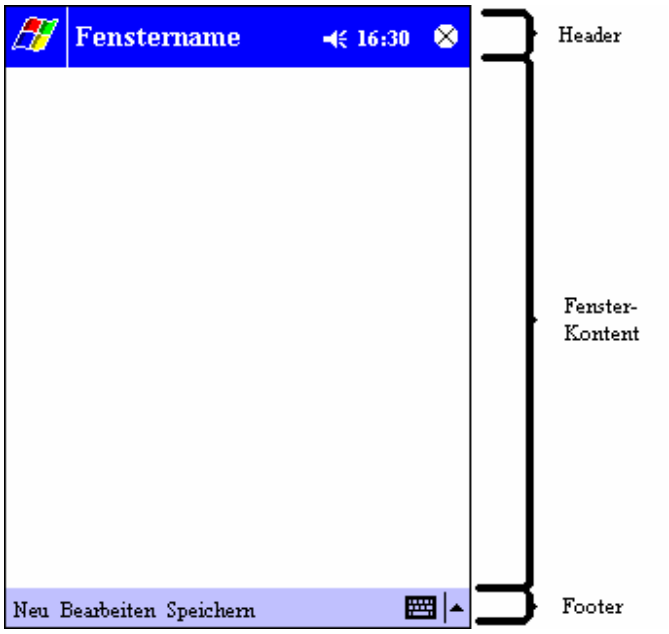
ein weiteres Fenster erzeugen. Wird als Windows-Klasse wieder die des zuerst erzeugten Fensters angegeben, dann basieren diese zwei Fenster auf der selben Windows-Klasse, was bedeutet, dass sich diese den selben Event-Queue teilen und so eine Message, die für ein Fenster generiert wurde, auch das andere mit involviert. Sind zum Beispiel zwei Fenster geöffnet, und der Nutzer schließt eines davon, so wird eine WM_QUIT-Message an die Windows-Klasse gesendet, die letztlich beide Fenster zugleich schließt.
Zum anderen ist es möglich, für jedes Fenster eine eigene Windows-Klasse zu definieren und somit einen eigenen Message-Queue für jedes Fenster zu generieren.

4.4. Fenstererstellung für Windows-Pocket-PC
4.4.1. Die Windows Class

Das Erstellen eines Anwendungs-Fensters für den Pocket-PC läuft nach dem gleichen Muster wie beim Desktop-PC ab. Jedoch weist Windows CE aufgrund seiner Architektur einige Unterschiede auf.

Ein Fenster in Windows CE wird meist nur in zwei Modi dargestellt, im maximierten oder im Vollbildmodus. Abb.4.4.1.A zeigt das typische Grundgerüst eines maximierten Fensters.

Abbildung 4.4.1.A: Grundgerüst eines Windows CE-Fensters



Das Grundgerüst besteht grob gesehen aus drei Teilen. Der erste ist ein Header, der den Fensternamen, die Uhrzeit, die Statusanzeige des Lautsprechers und einen Button zum Schließen des Fensters enthält. Über das Windowssymbol lässt sich ein Menü öffnen, über dass sich andere Programme starten lassen, deren Anwendungsfenster dann das aktuelle Fenster komplett überdecken.

Im zweiten Teil wird der eigentliche Fensterkontent dargestellt.

Der dritte Teil ist der Footer. Er beinhaltet die mit dem Fenster assoziierten Haupt-Menüpunkte bzw. Icons und einen Button zum Öffnen der virtuellen Tastatur.

Auf Minimier- / Maximierfunktionen von Fenstern wurde verzichtet. Im Vollbildmodus fallen sowohl Header als auch Footer weg. In diesem Falle obliegt dem Programmierer die Gestaltung des gesamten Screens und der Fenstersteuermöglichkeiten.

Als Basis-Struktur für die Windowsklasse auf einem Pocket-PC dient die WNDCLASS. Listing 4.4.1.A zeigt den Prototypen dieser Struktur:

Listing 4.4.1.A: Prototyp der Struktur WNDCLASS

```
typedef struct WNDCLASS {
    UINT    style;
    WNDPROC lpfnWndProc;
    int     cbClsExtra;
    int     cbWndExtra;
    HANDLE  hInstance;
    HICON   hIcon;
    HCURSOR hCursor;
    HBRUSH  hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS;
```

Da die Funktionen der WNDCLASS-Parameter mit denen der Desktop-Windows-Variante identisch sind, wird auf eine nochmalige Erläuterung verzichtet. Zwei Besonderheiten sind jedoch zu beachten. Zum einen muss die der WNDPROC zugewiesene Funktion in den Typ WNDPROC konvertiert werden, etwa so:

```
winclass.lpfnWndProc   = (WNDPROC) WinProc;
```

Des Weitern müssen ganz im Gegensatz zu MS Visual C++ in Hochkommas stehende Zeichenketten (Strings) extra noch mittels der Funktion bzw. dem Macro _T(*string*) in null-terminierte (LPCTSTR)-Strings umgewandelt werden, so zum Beispiel bei der Zuweisung des Namens der Windows-Klasse:

```
winclass.lpszClassName = _T("WINCLASS1");
```

Sobald WNDCLASS definiert ist, wird sie mittels

```
RegisterClass(&winclass);
```

registriert und damit vom Betriebssystem als Windows-Klasse anerkannt.

4.4.2. Das Erstellen des eigentlichen Fensters

Zum Kreieren eines Fensters stehen dem Programmierer zwei Funktionen zur Verfügung, zum einen die Funktion *CreateWindow()* und zum anderen *CreateWindowEx()*. *CreateWindowEx()* wurde schon im Abschitt 4.3.2 verwendet, womit sich eine nochmalige Erklärung der einzelnen Parameter erübrigt. Die Funktion *CreateWindowEx()* unterscheidet sich auch nur durch den Parameter dwExStyle von der Funktion *CreateWindow()*. *dwExStyle* spezifiziert den erweiterten Stil für Windows-Popup-Fenster, für Anwendungen im Vollbildmodus ist dieser Parameter jedoch irrelevant, so dass die Funktion *CreateWindow()* an dieser Stelle genügt. Listing 4.4.2.A zeigt den Prototyp dieser Funktion.

Listing 4.4.2.A: Prototyp der Funktion *CreateWindow()*

```
HWND CreateWindow(
  LPCTSTR lpClassName,      // Pointer auf registrierte Klasse
  LPCTSTR lpWindowName,     // Pointer auf den Fensternamen
  DWORD dwStyle,            // Fenster-Style
  int x,                    // horizontale Position des Fensters
  int y,                    // vertikale Position des Fensters
  int nWidth,               // Fensterbreite
  int nHeight,              // Fensterhöhe
  HWND hWndParent,          // Adresse des übergeordneten Fensters
  HMENU hMenu,              // Adresse eines untergeordneten
                            // Fensters oder eines Menüs
  HINSTANCE hInstance,      // Adresse der Applikationsinstanz
  LPVOID lpParam            // Pointer auf CREATESTRUCT,
                            // NULL setzen
);
```

Der Aufruf der Funktion im Quelltext kann zum Beispiel so aussehen:

```
hwnd = CreateWindow(_T("WINCLASS1"),      // Klasse
                    _T("WINDOWSTITLE"),   // Titel
                     WS_VISIBLE           // Window-Stil
                     0,0,                 // Koordinaten der
                                          // linken, oberen
                                          // Ecke des Fensters
                     240,320,             // Länge u. Breite
                                          // des Fensters
                     NULL,                // Adr. zu
                                          // übergeordn.Fenster
                     NULL,                // Adresse zu einem Menü
                     hinstance,           // Adr. zu einer
                                          // Applikationsinstanz
                     NULL                 // Erw. Stilparameter
);
```

4.4.3. Der Windows Event-Handler - WinProc

Der Prototyp der Callback-Funktion ist der gleiche wie der der Desktop-Windows-Variante. Ebenso lässt sich das Listing 4.3.3.B als Beispieldeklaration komplett übernehmen.

Allerdings ist es nötig, noch einen weiteren Event zu behandeln - WM_LBUTTONDOWN. Wird das Fenster über das im Header dafür vorgesehene Kreuz geschlossen, so ist zwar das Fenster ausgeblendet, aber nicht die Applikation geschlossen.

Eine Möglichkeit, dies zu umgehen, ist, das Programm bei Bildschirmberührung zu schließen. WM_LBUTTONDOWN wird gesendet, sobald eine Berührung des Bildschirms festgestellt wird. Ist das der Fall, so werden zwei Nachrichten an das System gesendet:

```
SendMessage(hwnd,WM_ACTIVATE,MAKEWPARAM(WA_INACTIVE,0),
                             (LPARAM)hwnd);
```

blendet das aktuelle Fenster aus und setzt seinen Status auf inaktiv,

```
SendMessage(hwnd,WM_CLOSE,0,0);
```

sendet eine WM_CLOSE-Nachricht, die die Applikation schließt. Listing 4.4.3.A zeigt die *WinProc()*-Funktion für WindowsCE:

Listing 4.4.3.A: Quellcode der Callback-Funktion *WinProc()*

```
LRESULT CALLBACK WinProc(HWND hwnd,UINT msg,WPARAM wparam,
                                            LPARAM lparam) {
  PAINTSTRUCT ps;
  HDC         hdc;

  switch(msg) {
    case WM_LBUTTONDOWN: {
      SendMessage(hwnd,WM_ACTIVATE,MAKEWPARAM(WA_INACTIVE,0),
                                   (LPARAM)hwnd);
      SendMessage(hwnd,WM_CLOSE,0,0);

      return(0);
    } break;

    case WM_CREATE: {

      // Platz für Initialisierungen

      return(0);
    } break;

    case WM_PAINT: {
      hdc = BeginPaint(hwnd,&ps);

      // Platz für Funktionen, die auf den Bildschirm zeichnen

      EndPaint(hwnd,&ps);
      return(0);
    } break;

    case WM_DESTROY: {
      // Applikation schließen

      PostQuitMessage(0);
      return(0);
    } break;

  default:break;
  } // end switch

  // DefWindowProc (default window procedure) leitet alle
  // Messages an das System weiter, die hier nicht verarbeitet
```

```
  // wurden
  return(DefWindowProc(hwnd,msg,wparam,lparam));

} // Ende: callback function WinProc
```

4.4.4. Main-Event-Loop

Ohne Veränderung lässt sich der Main-Event-Loop der Desktop-Windows-Varianten (siehe Abschnitt 4.3.4) übernehmen.

Das Beispiel zum Erstellen eines einzelnen Fensters unter Windows CE (1stWindowCE.cpp) und die dazugehörige ausführbare Datei (1stWindowCE.exe) befinden sich auf der beigelegten CD im Verzeichnis \Kapitel4\

4.4.5. Die Ressourcen eines CE-Fensters

Ebenso wie bei einem Desktop-Windows-Fenster lassen sich auch bei CE-Fenstern Ressourcen wie Icons, Mauscursor und Menüs einbinden. Dazu sind die gleichen Schritte nötig. Am Beispiel des Icons noch einmal demonstriert, sieht der Quelltext so aus:

```
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

wäre dann durch

```
winclass.hIcon = LoadIcon(hinstance,MAKEINTRESOURCE(IDI_MYICON));
```

zu ersetzen. Ebenso müssen die Icons im Ressourcen-Header

```
(#define IDI_MYICON  200     //Icon bekommt ID 200)
```

und in der Ressourcen-Datei

```
(IDI_MYICON ICON myicon.ico)
```

definiert werden.

Die Integration von Cursorn wird zwar auch unterstützt, jedoch erscheint es wenig sinnvoll. Bei einem Pocket-PC ist keine Maus vorhanden, die Eingaben werden entweder über die Steuertasten des Gerätes oder über den Stylus (Eingabestift) registriert, wobei letzterer direkt auf dem Bildschirm wirkt und nicht wie die Maus als relatives Eingabegerät.

Das Einbinden von WindowsCE-basierten Menüs soll hier übersprungen werden, da dieses Feature für das weitere Vorgehen in der Arbeit nicht von Relevanz ist und zudem auch den Rahmen dieser Ausarbeitung sprengen würde.

4.4.6. Das Öffnen mehrerer Fenster

Oftmals füllen die Fenster einer Anwendung die gesamte Bildschirmfläche aus, so dass es vorkommen kann, dass mehrere Fenster direkt übereinander liegen. Leider bietet Windows CE kein Feature, um direkt zwischen den Fenstern hin- und herzuschalten, wie es etwa mit der Taskleiste vom Windows für Desktop-PCs möglich ist. Zudem sind auch wie bereits erwähnt keine Minimier- bzw. Maximierfunktionen für die Fenster vorgesehen. Auf eine Applikation, die unter einer anderen "liegt", lässt sich demnach nicht ohne weiteres zugreifen. Für genau diesen Fall empfiehlt es sich, in jedem Programm zu überprüfen, ob das zugehörige Fenster nicht schon geöffnet ist und dieses Fenster dann ggf. in den Vordergrund setzen. Eine Überprüfung, ob ein Fenster schon existiert, lässt sich zum Beispiel auf diese Weise realisieren:

```
const TCHAR cszClassName[] = _T("Winclass1");
const TCHAR cszTitle[]     = _T("Mein erstes Fenster");

hWnd = FindWindow(cszClassName, cszTitle);
if (hWnd) {
  SetForegroundWindow ((HWND) (((DWORD) hWnd) | 0x01));
  return 0;
}
```

FindWindow() erkennt anhand des Namens der Windows-Klasse und des Fenstertitels, ob ein Fenster schon existiert. Ist dies der Fall, so wird dieses Fenster mittels *SetForeGroundWindow()* fokusiert bzw. in den Vordergrund gesetzt, anstatt es neu zu generieren. *SetForeGroundWindow()* übernimmt als Parameter das Typ-angepaßte (type casted) Window-Handle (hier: hWnd).

Zudem empfiehlt es sich nach dem Aufruf der *CreateWindow()* bzw. *CreateWindowEx()* - Funktionen, die Funktion *ShowWindow(hWnd, SHOWNORMAL)* aufzurufen. hWnd ist in diesem Falle das Handle des Fensters und SHOWNORMAL ein Flag, dass das aktuelle Fenster aktiviert, darstellt und fokusiert.

Ist beabsichtigt, mit einer Applikation mehrere Fenster zu öffnen, so geschieht dies über den mehrfachen Aufruf der Funktion *CreateWindow()*, analog zu der in Abschnitt 4.3.6. beschriebenen Weise.

Kapitel 5: Grafikdarstellung

5.1. Möglichkeiten des Desktop-PC

5.1.1. Bildschirmauflösungen und Farbmodi für Desktop-PCs

Ein zentraler Aspekt bei der Programmierung ist das Darstellen von Informationen oder Grafiken auf dem Bildschirm. Die Monitore der ersten Rechner konnten lediglich weiße Zeichen auf schwarzem Hintergrund darstellen, und erst mit der Einführung der Grafikkarte ließen sich dann farbige Objekte auf dem Monitor anzeigen.

Der Bildschirm selbst kann als ein großes Rechteck-Raster betrachtet werden, dass je nach Bildschirmauflösung in bis zu 1.920.000 einzelne Raster- bzw. Bildpunkte, sogenannte Pixel, unterteilt ist. Eine Grafikkarte für einen Desktop-PC der Pentium 4-Generation unterstützt eine Bildschirmauflösung von 320x240 (76.800 Rasterpunkte) bis hin zu 1600x1200 Pixeln (1.920.000 Rasterpunkte) und bis zu vier verschiedene Farbmodi.

Der Farbmodus bzw. die Farbtiefe repräsentiert die pro Pixel zum Codieren einer Farbe verwendete Anzahl der Bits (siehe Tabelle 5.1.1.A). Mit der Formel
Farbanzahl = 2^(Bit pro Pixel)
lässt sich ausrechnen, wie viele Farben in Abhängigkeit von den verwendeten Bits codiert werden können,
mit einem einzelnen Bit zum Beispiel 2 Farben, mit 2 Bits 4 usw.

Tabelle 5.1.1.A: Übersicht über die Anzahl von Farben in Abhängigkeit von den verwendeten Bits

Anzahl Bits	Codierbare Farben
1	2
2	4
4	16
8	256
16	65.535
24	16.777.216

Genau genommen gibt es zwei Wege, eine Farbe auf einem Bildschirm darzustellen: direkt oder indirekt.

Direkte Farb- bzw. RGB-Modi stellen jedes Pixel mit entweder 16, 24 oder 32 Bit dar, die die roten, grünen und blauen Komponenten einer Farbe repräsentieren. Dies ist durch die additive Natur der Farben rot, grün und blau möglich, also der Tatsache, dass sich mit bestimmten Mischungen aus diesen drei Farben jede andere sichtbare Farbe ableiten lässt. Es fällt dabei aber auf, dass 16 und 32 nicht durch 3 teilbar sind, so dass den drei Farben nicht die gleiche Anzahl Bits zur Verfügung stehen.
Im 16-Bit-RGB-Modus (65.536 Farben) gibt es mehrere Strategien, das "Rest-Bit" zu verwenden. Die gängigste und von den meisten Grafikkarten unterstützte ist:

RGB (5.6.5) - 5 Bit rot, 6 Bit grün, 5 Bit blau
Das grüne Farbspektrum wird noch genauer unterteilt, da dass Auge grüne Flächen und Farben am besten differenzieren kann.

Im 24-Bit-Modus (16.777.216 Farben) stehen jeder Farbkomponente (rot, grün, blau) 8 Bit zur Verfügung. Dies ist der für den Programmierer einfachste Modus. Er wird aber von den meisten Grafikkarten nicht unterstützt, und soll daher nicht weiter betrachtet werden.

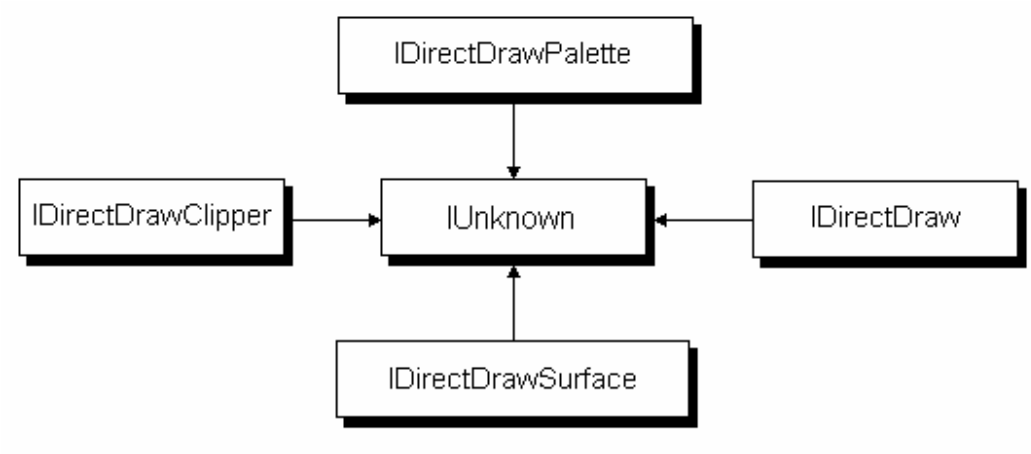
Im 32-Bit-Modus (16.777.216 Farben) wird anstatt noch weiterer Farbenuntergliederung Alpha-Informationen gespeichert. Diese Alpha-Informationen bestimmen, wie transparent das jeweilige Pixel bezüglich seines Hintergrundes ist. Da 8-Bit zum Codieren der Transparenz zur Verfügung stehen, ergeben sich daraus 256 Transparenz-Abstufungen von komplett durchsichtig bis völlig überdeckend (opak).

Bei indirekten Farbmodi wird intern in der Grafikdatei eine Farbpalette gespeichert. Eine Palette ist eine Tabelle mit 2^Farbtiefe Einträgen, im Falle einer 8-Bit-Grafik also 256 Möglichkeiten, um bestimmte Farben zu definieren. Einer solchen Palette können beliebige RGB-Farben zugeordnet werden, zum Beispiel auch nur blaudominierte Farben, Graustufen, Rottöne usw. Jeder Farbe in einer Palette ist ein Index zugeordnet, über den sich dann die Farbe auswählen lässt. Es wird also nicht direkt die Farbe angesprochen, sondern deren Indexwert, was bei zwei Grafiken mit unterschiedlicher Farbpalette zwei verschieden colorierte Bilder ergibt.

5.1.2. Initialisierung von DirectDraw

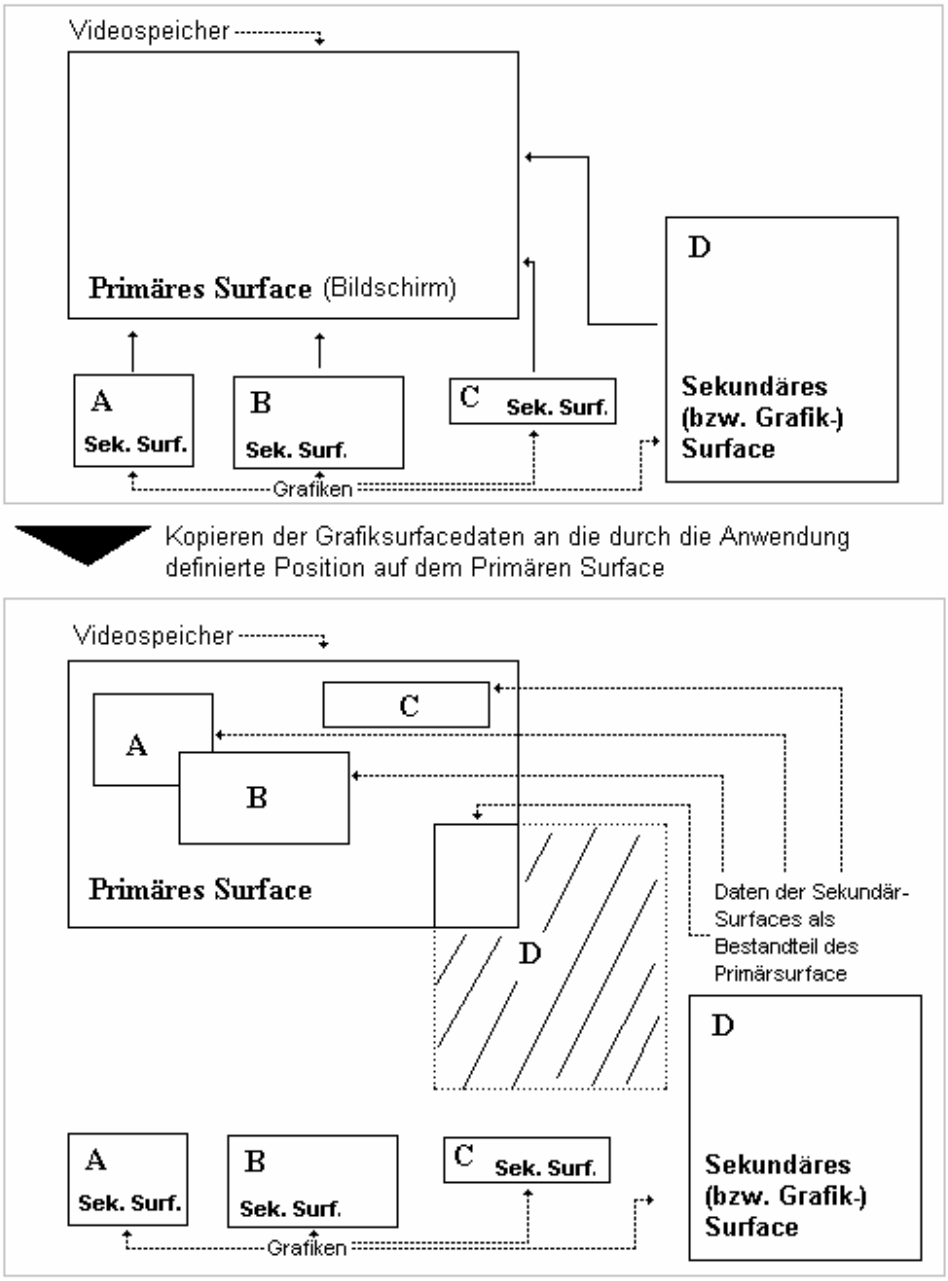
Wie schon in Kapitel 3 erwähnt, besteht DirectX aus einer Reihe von Komponenten, die wiederum COM-Objekte sind. DirectDraw, das 2D-Grafikdarstellungsmodul von DirectX, bildet da keine Ausnahme. Es wurde zwar ab Version 8.0 mit den Direct3D-Komponenten zu DirectGraphics zusammengefaßt, Thema der Arbeit sind aber die 2D-Routinen dieses Paketes, weshalb im weiteren ausschließlich auf die letzte, am weitesten entwickelte DirectDraw-Version (7) Bezug genommen wird. Wie Abb. 5.1.2.A zeigt, besteht DirectDraw aus fünf Schnittstellen:

Abbildung 5.1.2.A: Aufbau des COM-Objektes DirectDraw



- IUnknown: Alle COM-Objekte müssen von diesem Basisinterface abgeleitet werden. IUnknown enthält die Funktionen *QueryInterface()*, *AddRef()* und *Release()*, die Kommunikationsschnittstellen des COM-Objektes.
- IDirectDraw: Dies ist die Hauptschnittstelle des DirectDraw-Objektes. Sie muß erstellt werden, um mit DirectDraw-Funktionen arbeiten zu können. IDirectDraw erkennt die Grafikhardware und repräsentiert den Videospeicher einer der Grafikkarten im System. Mit MMS (Multiple Monitor Support) lassen sich auch mehrere Grafikkarten im System mit DirectX verwalten, für jede weitere Karte ist dann ein neues IDirectDraw-Objekt zu erzeugen.
- IDirectDrawSurface: Ein Surface (Oberfläche, hier: Speicherbereich für Grafikobjekt) existiert eigentlich nur virtuell, in Wirklichkeit repräsentiert es Speicherbereiche im Videospeicher oder auf dem System, in dem Grafikobjekte liegen. Ein Grafikobjekt kann ein einfaches Bild sein, es kann aber ebenso gut die gesamte Bildschirmanzeige repräsentieren. Es gibt grundsätzlich zwei Surface-Typen: Primäre und sekundäre Surfaces.
Ein primäres Surface dient als tatsächlicher Videopuffer und speichert die (Grafik-)Daten, die durch die Grafikkarte dargestellt und durch den Monitor angezeigt werden sollen.
In sekundären Surfaces werden alle Grafikobjekte abgelegt und je nach Bedarf im Primären Surface zum auf dem Monitor sichtbaren Bild kombiniert (siehe Abb. 5.1.2.B.)

Abbildung 5.1.2.B: Beziehungen zwischen dem primären und den sekundären Grafiksurfaces



Das Prinzip der Surfaces, deren Erstellung und Anwendung wird im Laufe dieses Abschittes noch näher beleuchtet.

- IDirectDrawPalette: DirectDraw ist so konzipiert, dass es mit jeder Farbtiefe von 1 Bit bis 32 Bit umgehen kann. Indirekte Farbmodi mit 8-Bit Farbtiefe oder geringer speichern Ihre Farben in einer Palette (siehe Abschnitt 5.1.1.). Mit IDirectDraw können solche Paletten erstellt, geladen und manipuliert werden. Weiterhin besteht die Möglichkeit, bestehenden Surfaces eine Palette zuzuordnen.
- IDirectDrawClipper: Clipping ist der Prozess des Begrenzens der Ausgabe auf eine bestimmte Region. Grafik-Informationen außerhalb dieser Region werden nicht

dargestellt. Im unteren Teil der Abb.5.1.2.B. ist ein solches Clipping zu sehen. Von der Grafik des sekundären Surfaces D wird nur die linke obere Ecke dargestellt, der schraffierte Bereich aber weggeschnitten.
Die Schnittstelle IDirectDrawClipper bietet nun Funktionen, um einen solchen Clipper anzulegen, einem Surface zuzuordnen und zu verwalten. Sie greift dabei auf die auf der Grafikkarte integrierten Clipping-Funktionen zu, sofern die Hardware dieses Feature unterstützt.

Um ein DirectDraw-Objekt zu erstellen, wird ein Pointer auf das Objekt angelegt und mit der Funktion DirectDrawCreateEx() eine IDirectDraw7-Schnittstelle zum Objekt erstellt.

```
LPDIRECTDRAW7 lpdd;   //Pointer auf DirectDraw7-Objekt

//erstellen einer DirectDraw7 - Objektschnittstelle
DirectDrawCreateEx(NULL,(void **)&lpdd, IID_IDirectDraw7, NULL);
```

DirectDrawCreateEx(Parameter1,Parameter2,Parameter3,Parameter4) benötigt dabei folgende Parameter:
- Parameter1: Das ist der GUID (Global Unique Identifier) des Anzeigentreibers. Der Anzeigentreiber ist ein COM-Objekt und besitzt daher auch einen 128-Bit-Identifizierungsschlüssel, den GUID (siehe Kapitel 3.2.). Übergibt man diesem Parameter *NULL*, so wird das aktive Display ausgewählt.
- Parameter2: Hier wird der "Empfänger" des Interfaces angegeben, also der Pointer auf das DirectDraw7-Objekt.
- Parameter3: Dies ist die InterfaceID der Schnittstelle, die angesprochen wird.
- Parameter4: Dieser Parameter wurde für zukünftige Erweiterungen hinsichtlich COM oder der DirectDrawCreateEx()-Funktion angelegt, wird aber meist nicht gebraucht und daher *NULL* gesetzt.

Als nächstes sollte Windows mitgeteilt werden, wie es das Fenster der Anwendung handhaben soll. Dafür gibt es zwei grundsätzliche Möglichkeiten: Den Vollbildmodus und den Standard-Fenstermodus.

Im Vollbildmodus (full screen mode) wird der ausführenden Anwendung der gesamte Bildschirm zugeteilt. Weiterhin darf auch nur diese Anwendung auf den Videospeicher bzw. die Grafikkarte zugriffen.

Im Standard-Fenstermodus (windowed mode) steht einer ausgeführten Anwendung nur eine bestimmte Region auf dem Bildschirm zur Verfügung. Sie muss daher stärker mit Windows kooperieren und sich in vielerlei Hinsicht (z.B. Fokussetzung: Aktive Anwendung im Vordergrund) nach anderen Applikationen richten.

Um den für die Anwendung relevanten Kooperationsmodus zu setzen, hat Microsoft die Funktion SetCooperativeLevel() entwickelt. Sie benötigt nur zwei Parameter, das Windowshandle (hwnd, Kap.4) und eine Anzahl von sogenannten Kontrolflags, die mit logischem ODER (|) verbunden sein können. Möglichkeiten für diese Flags zeigt Tabelle 5.1.2.A.

Tabelle 5.1.2.A: Kontrolflags der DirectDaw-Funktion SetCooperativeLevel()

Wert	Beschreibung
DDSCL_FULLSCREEN	Zeigt an, dass der Vollbildmodus genutzt werden soll. Muss zusammen mit DDSCL_EXCLUSIVE verwendet werden.
DDSCL_ALLOWREBOOT	Die Ctrl+Alt+Del-Kombination zum Neustart des Rechners kann mit diesem Flag auch im Vollbildmodus genutzt werden.
DDSCL_EXCLUSIVE	Zeigt an, dass die Bildschirmanzeige exklusiv von einer Anwendung genutzt werden soll. Muss zusammen mit DDSCL_FULLSCREEN verwendet werden.
DDSCL_NORMAL	Bestimmt, dass das Fenster im Standard-Fenstermodus ausgeführt werden soll. Kann nicht in Zusammenhang mit DDSCL_FULLSCREEN oder DDSCL_EXCLUSIVE gesetzt werden.

Alle weiteren Möglichkeiten sind in der Hilfe-Dokumentation des MS Visual C-Studios unter dem Punkt SetCooperativeLevel für DirectDraw zu finden.

In den meisten Fällen würde der Funktionsaufruf für eine Vollbildanwendung so aussehen:

```
ldpp->SetCooperativeLevel(hwnd,
                          DDSCL_FULLSCREEN,
                          DDSCL_ALLOWREBOOT,
                          DDSCL_EXCLUSIVE
);
```

und für eine Standard-Fenster-Applikation so:

```
ldpp->SetCooperativeLevel(hwnd,DDSCL_NORMAL);
```

Soll eine Applikation im Vollbildmodus erstellt werden, schließt sich an den Aufruf der SetCooperativeLevel()-Funktion die Definition des Anzeigenmodus (display mode) an. Mit der Funktion SetDisplayMode() wird Windows mitgeteilt, in welcher Auflösung und Farbtiefe die Vollbildanwendung laufen soll. Der Aufruf

```
lpdd->SetDisplayMode(640,480,8,0,0);
```

würde zum Beispiel eine Anwendung mit einer Auflösung von 640x480 Pixeln und 8 Bit Farbtiefe erstellen. Parameter 4 der Funktion legt die Rate fest, in der der Bildschirm neu gezeichnet wird, dabei steht 0 für die vom System vorgegebene Standardeinstellung. Parameter 5 ist für spätere Weiterentwicklungen reserviert und muss auf 0 gesetzt werden.

Erwähnenswert ist hierbei, dass theoretisch auch Werte wie 400x400 möglich sind, sofern die Grafikkarte solche Auflösungen bewältigen kann. Jedoch ist es ratsam, auf gängige Werte zurückzugreifen (siehe Tabelle 5.1.2.B), da diese von nahezu jedem Windowssystem unterstützt werden.

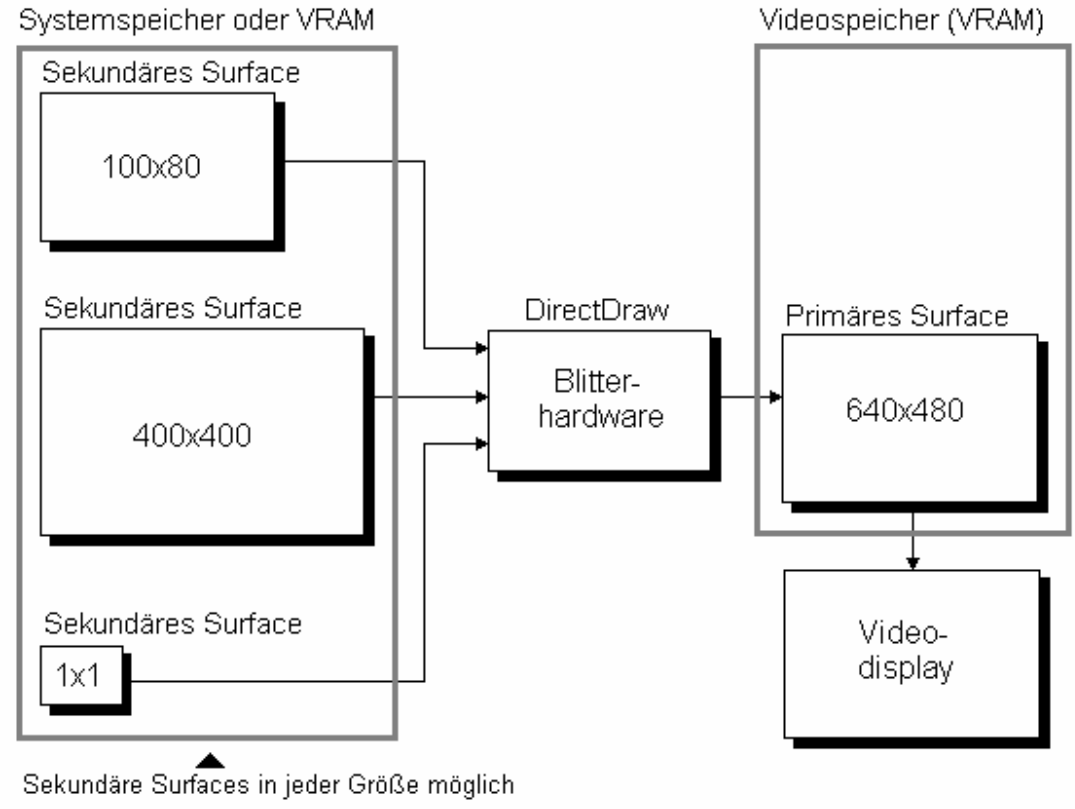
Tabelle 5.1.2.B: Gängige Werte für Bildschirmauflösungen bei einem Desktop-Windows-PC

Bildschirmweite	Bildschirmhöhe	Farbtiefe(Bits pro Pixel)
640	480	8, 16, 24, 32
800	600	8, 16, 24, 32
1024	768	8, 16, 24, 32
1280	1024	8, 16, 24, 32

Bei vielen Grafikkarten sind auch höhere Auflösungen möglich.

Je nach Auswahl der Auflösung und Farbtiefe wird die Grafikkarte nun die sichtbare Bildfläche einrichten. Wie bereits beschrieben, ist das Bild, das auf dem Monitor zu sehen ist, eine Matrix farbiger Pixel, die im Videospeicher der Grafikkarte abgelegt wird. Um das angezeigte Bild zu verändern, muss diese Pixelmatrix manipuliert werden. DirectDraw stellt Funktionen bereit, die die Erstellung, Manipulation und den Zugriff auf solche Pixelmatrizen, sogenannte Surfaces, erlauben. Wie die Abbildung. 5.1.2.C zeigt, sind Surfaces rechteckige Speicherregionen, die Bitmap-Daten beinhalten.

Abbildung 5.1.2.C: Rechteckige DirectDraw-Surfacestrukturen als Speicher für Bitmapdaten



Es gibt zwei Arten von Surfaces: Primäre und sekundäre.

Ein primäres Surface korrespondiert direkt mit dem tatsächlichen Videospeicher (VRAM) der Grafikkarte und ist jederzeit sichtbar. Es hat die gleichen Dimensionen wie die Bildschirmanzeige. Wurde mit SetDisplayMode zum Beispiel eine Auflösung von 640x480 mit 16 Bit festgelegt, so weist auch das primäre Surface diese Werte bzw. die damit verbundene Größe auf.
Sekundäre Surfaces sind flexibler, da sie jede beliebige Größe annehmen können, entweder im VRAM der Grafikkarte oder im Systemarbeitsspeicher (RAM) ablegbar sind und in theorethisch unbegrenzter Anzahl zur Verfügung stehen. Im Gegensatz zum primären Surface sind sekundäre Surfaces nie direkt sichtbar (offscreen), sie existieren nur virtuell im Hintergrund und enthalten Grafiken, die bei Bedarf in den Speicher des Primären Surfaces übertragen werden können.

Die Funktion *CreateSurface()* legt ein neues Surface an. Als Parameter übernimmt sie:
1. die Adresse der DirectDraw Surface Description (DDSD), einer Datenstruktur, in der alle Surface-Eigenschaften definiert werden können. In Listing 5.1.2.A werden die wichtigsten Eigenschaften der DDSD für ein primäres Surface festgelegt, eine detailiertere Hilfe ist in der MS-VisualC++-Hilfe unter dem Punkt DDSURFACEDESC zu finden:

Listing 5.1.2.A: Beispieldefinition der Surface-Beschreibung für ein primäres Surface

```
DDSURFACEDESC2 ddsd;          // Initialisierung der Surface-
                              // Beschreibungs-Struktur (DDSD)

memset(&ddsd,0,sizeof(ddsd)); // Speicher für die Struktur in
                              // Größe der DDSD reservieren

ddsd.dwSize = sizeof(ddsd);   // in DDSD dessen Größe speichern
ddsd.dwFlags = DDSD_CAPS;     // DDSD mitteilen, dass DDSD_CAPS
                              // gesetzt werden sollen
                              // über DDSD_CAPS der DDSD
                              // mitteilen, dass das DirectDraw-
                              // Surface ein primäres Surface ist
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
```

2. die Adresse eines Directdraw-Surfaces, dem die DDSD zugewiesen werden soll:

```
LPDIRECTDRAWSURFACE7 lpddsprimary = NULL;
```

ist im Vorfeld festzulegen.

3. *NULL*, da dieses Feld für spätere Erweiterungen reserviert ist.

```
lpdd->CreateSurface(&ddsd,&lpddsprimary,NULL);
```

zeigt den Aufruf von CreateSurface(), wenn ein primäres Surface mit der DDSD aus Listing 5.1.2.A generiert werden soll. Das Anlegen sekundärer Surfaces als Grafikspeicher oder Hintergrundpuffer wird im Abschnitt 5.1.3. ausführlich behandelt.

Wurde in *SetDisplayMode()* der 8-Bit-Farbmodus festgelegt, bei dem die Farbeinträge in einer 256-Farben-Palette gespeichert sind (siehe Abschnitt 5.1.1.), ist zusätzlich noch folgendes zu tun:
1. Erstellen einer oder mehrerer Palettenstrukturen als Arrays vom Typ PALETTEENTRY mit 256 Einträgen.
PALETTEENTRY ist eine Struktur, die aus folgenden Einträgen besteht:

```
typedef struct PALETTEENTRY {
  BYTE peRed;
  BYTE peGreen;
  BYTE peBlue;
  BYTE peFlags;
}
```

Es werden die roten, grünen und blauen Farbkomponenten eines Pixels als 8-Bitwert übergeben, für peFlags empfiehlt es sich, *PC_NOCOLLAPSE* zu übergeben, damit die Pixelfarbe nicht an die Systemfarben angepasst und somit verfälscht dargestellt wird.
Mit

```
PALETTEENTRY palette[256];
```

wird der Palettenarray erzeugt, Listing 5.1.2.B demonstriert das Erstellen einer Zufallspalette:

Listing 5.1.2.B: Zugriff auf Variablen der Struktur PALETTEENTRY am Beispiel einer zufällig generierten 256-Farben-Palette

```
for (int i=0; i<256; i++){
  //Füllen der Palette mit zufälligen RGB-Werten
  palette.peRed    = rand()%256;
  palette.peGreen = rand()%256;
  palette.peBlue   = rand()%256;
  // Flag auf PC_NOCOLLAPSE setzen, um Farb-Manipulationen
  // durch das System zu vermeiden
  palette.peFlags = PC_NOCOLLAPSE;
}
```

Hinweis: Unter manchen Systemen wie zum Beispiel Windows NT sind die Paletteneinträge 0 und 255 schon fest definiert. Wurden die Einträge durch den Nutzer anders festgelegt, überschreibt das System diese und ersetzt sie durch die Farben weiß (Eintrag 0) bzw. schwarz(Eintrag 255).

2. Erstellen einer Palettenschnittstelle (IDirectDrawPalette) zum DirectDraw-Objekt.
Die Funktion CreatePalette kreiert ein Palettenobjekt und weist Ihm bestimmte Eigenschaften zu. Der Prototyp dieser Funktion (siehe Listing 5.1.2.C) zeigt, welche Parameter erwartet werden.

Listing 5.1.2.C: Prototyp der DirectDraw7-Funktion CreatePalette

```
HRESULT CreatePalette(
           DWORD dwFlags,               //Eigenschaftenflags
           LPPALETTEENTRY lpColorTable, //Palettendaten (-array)
           LPDIRECTDRAWPALETTE FAR *lplpDDPalette,
                                        //Palettenschnittstelle
           IUnknown FAR *pUnkOuter      //auf NULL setzen
);
```

Für *dwFlags* können zum Beispiel folgende evtl. ODER (|)-verknüpften Werte stehen:
DDPCAPS_8BIT: 256 Einträge in der Farbpalette
DDPCAPS_ALLOW256: Alle 256 Einträge, auch die auf manchen Systemen festgelegten Einträge 0 und 256 sind frei definierbar.
DDPCAPS_INITIALIZE: Die Palettenfarben werden der Hardwarefarbpalette zugeordnet. Empfohlen, falls die Palette dem primären Surface zugeteilt werden soll.

Als *lpColorTable* ist das Palettenarray *palette* zu setzen.

Die Palettenschnittstellenpointer ist im Vorfeld mit

```
LPDIRECTDRAWPALETTE lpddpal = NULL;
```

festzulegen, *lpddpal* wird dann als **lplpDDPalette* übergeben.

Der Pointer **pUnkOuter* ist reserviert für spätere Erweiterungen und wird auf *NULL* gesetzt.

3. Zuweisen des Palettenobjekts zu dem Surface, dass sich auf diese Palette beziehen soll (z.B das Primäre Anzeigesurface).
Die Palette für ein Surface wird mit der Funktion *SetPalette()* zugewiesen, die als Parameter einen Pointer auf die jeweilige Palette übernimmt. Mit

```
lpddsprimary->SetPalette(lpddpal);
```

wird zum Beispiel die Palette, auf die *lpddpal* zeigt, dem primären Display zugewiesen.

Hinweis: Die Zuweisung von Paletten zu Surfaces ist nicht absolut. Es ist durchaus möglich, während der Laufzeit eines Programms die Paletten generieren zu lassen, zu manipulieren oder auszutauschen.

Hinweis: SystemRessourcen sollten prinzipiell freigegeben werden, sobald sie nicht mehr gebraucht werden, am besten in umgekehrter Reihenfolge ihrer Erstellung. Dies trifft auch auf die DirectDraw-Schnittstellen zu. Listing 5.1.2.D demonstriert die Freigabe dieser Schnittstellen an einem Beispiel:

Listing 5.1.2.D: Freigabe der DirectDraw-InterfaceRessourcen

```
if (lpddpal){
  lpddpal->Release();           // gibt die Palette frei,
  lpddpal = NULL;
} // end if

if (lpddsprimary){              // ...das primäre Surface...
  lpddsprimary->Release();
  lpddsprimary = NULL;
} // end if

if (lpdd){
  lpdd->Release();              // ...und zum Schluß das
  lpdd = NULL;                  // DirectDraw-Objekt selbst
} // end if
```

Es wird jeweils geprüft, ob der Pointer auf die Objekte wirklich (noch) gesetzt ist. Das Freigeben von Ressourcen, die nicht existent sind, kann unvorhersehbare Fehler beim Aufruf des betreffenden Programms hervorrufen.

Um generell das Resultat von Funktionsaufrufen zu testen, zum Beispiel, ob ein DirectDraw-Objekt erfolgreich angelegt wurde, bietet DirectX zwei boolsche Funktionen: FAILED() und SUCCEEDED().

```
if (FAILED( FunktionA() )){
  //Fehlerbehandlung
}
```

und

```
if (SUCCEEDED( FunktionB() )){
  //kein Fehler, Programm weiter ausführen
}
```

sind zwei Möglichkeiten, diese Funktionen für Fehlertestzwecke anzuwenden.

5.1.3. DirectX-unterstütztes Manipulieren einzelner Pixel (Pixel-Plotting)

Um auf Surfaces wie zum Beispiel das primäre Surface zugreifen zu können, muss auf diesem eine bestimmte Region gesperrt werden, auf die dann kein anderer Prozess zugreifen kann. Zudem soll auch die Grafikkarte den Videospeicheradressraum, der durchaus nicht immer konstant bleibt, während des Sperrens nicht verschieben. Dieser Sperrzustand wird mit der Funktion *Lock()* eingeleitet und mit *Unlock()* wieder aufgehoben.

Lock() übernimmt vier Parameter:
1. Eine Rechteckstruktur vom Typ RECT, in der definiert ist, auf welchen (rechteckigen) Teil des Surfaces zugegriffen werden soll. Zum Zugriff auf das gesamte Surface ist dieser Parameter *NULL* zu setzen.

2. Eine DirectDrawSurfaceDescription (DDSD), wie sie im Abschnitt 5.1.2. beschrieben wurde. Die DDSD sollte an dieser Stelle nur das Feld *dwSize* mit *sizeof(ddsd)* definiert haben, ansonsten aber "leer" sein.
3. Flags in logischer ODER-Verknüpfung (|), die bestimmte Eigenschaften der Sperrung definieren. Zwei Flags sind besonders hervorzuheben:
DDLOCK_SURFACEMEMORYPTR: Legt fest, das ein gültiger Pointer auf die linke obere Ecke der RECT-Struktur aus Parameter 1 zurückgegeben werden soll. Wurde für die RECT-Struktur NULL angegeben, zeigt dieser Pointer auf die linke, obere Ecke des jeweiligen Surfaces.
DDLOCK_WAIT: Legt fest, dass im Falle des Fehlschlagens der Funktion Lock() deren Aufruf so oft wiederholt wird, bis er erfolgreich ist oder ein anderer Fehler auftaucht, wie zum Beispiel DDERR_SURFACEBUSY, der anzeigt, dass ein anderer Prozess gerade auf die Anwendung zugreift.
4. NULL, da dieser Parameter laut Microsoft nicht genutzt wird.

Unlock() empfängt als Parameter die gleiche Rechteck-Struktur, die beim Aufruf der Funktion Lock() als erster Parameter übergeben wurde.

Listing 5.1.3.A demonstriert die Anwendung der beiden Funktionen:

Listing 5.1.3.A: Anwendung der Funktionen Lock() und Unlock() der IDirectDrawSurface-Schnittstelle zum Sperren einer bestimmten Region auf dem primären Surface

```
DDSURFACEDESC2 ddsd;

memset(&ddsd,0,sizeof(ddsd));    //"leeren" bzw. neu
                                 // initialisieren der DDSD
ddsd.dwSize = sizeof(ddsd);      // Das Feld dwSize einer DDSD
                                 // sollte immer gesetzt sein
lpddsprimary->Lock(NULL,
                   &ddsd,
                   SURFACEMEMORYPTR | DDLOCK_WAIT,
                   NULL
);

  //Funktionen zur Manipulation des Surfaces

lpddsprimary->Unlock(NULL);
```

Nachdem ein DirectDraw-Objekt angelegt wurde, die Bildschirmeigenschaften feststehen und ein primäres Surface mit evtl. zugewiesener Farbpalette bereitsteht, lässt sich das Bild auf dem Monitor individuell je nach Bedarf bearbeiten. Die einfachste Form, dies zu tun, ist das Setzen einzelner Pixel im primären Surface (Pixel-Plotting).

Das primäre Surface (auch: Videopuffer), im vorherigen Abschnitt als rechteckige Struktur beschrieben, ist eigentlich linear angelegt, vergleichbar mit einem Array aller Rasterpunkte. Die Surfaces abstrahieren lediglich das lineare Konzept, damit sich Programmierer die Struktur des Surfaces besser vorstellen können.

Mit der Formel

```
videobuffer8[x + y * memoryPitch] = pixelcolor8;
```

lässt sich auf jedes Pixel des Videopuffers zugreifen, sofern die Anwendung im 8-Bit-Modus läuft.

videobuffer8[] ist ein Array vom Typ *unsigned char*, dessen Inhalt dem Videopuffer zugewiesen werden muss. *x* ist als x-Koordinate eines Pixels auf dem Bildschirm zu verstehen, *y* als y-Koordinate des Pixels. In *memoryPitch* ist festgelegt, aus wie vielen Bytes eine Pixel-Reihe besteht. In *pixelColor8* wird dem betreffenden Pixel eine bestimmte Farbe zugeordnet bzw. im 8-Bit-Modus der Palettenindex der Farbe dem Pixel zugewiesen.

Diese Formel ist aber nur für 8-Bit-Anwendungen einsetzbar. Für 16-Bit zum Beispiel werden statt einem 2 Byte zur Codierung einer Pixelfarbe verwendet. *videobuffer16,* das Pixelarray für eine Applikation im 16-Bit-Modus, sollte demnach vom Typ *unsigned short* sein. Zudem bildet nur jedes ungerade Byte den Anfang eine 2-Byte-Pixel-Codierung, so dass *memoryPitch* durch 2 geteilt werden muss, um auf die Anfänge der jeweiligen Pixeldaten zuzugreifen. In *pixelcolor16* wird diesmal nicht der Palettenindex, sondern ein 16-Bit-Farbwert übergeben. Das Macro

```
#define RGB565(r,g,b) ((r >> 3)|((g >> 2)<< 5)|((b >> 3)<< 11))
```

konvertiert einen 24-Bit-RGB-Wert (ca. 16,7 Mio. Farben) in eine 16-Bit-Zahl, die einer der 65.536 darstellbaren Farben im 16-Bit-Modus entspricht.

```
videobuffer16[ x + y * (memoryPitch/2)] = pixelcolor16;
```

ist die Formel für den Zugriff auf Pixel im 16-Bit-Modus.

Für 32-Bit-Applikationen ist die Verfahrensweise analog.

```
videobuffer32[ x + y * (memoryPitch/4)] = pixelcolor32;
```

ist die Formel für diesen Modus, *unsigned long* ist als Datentyp für *videobuffer32* am besten geeignet und als MACRO zur Farbkonvertierung dient:

```
#define RGB(r,g,b)   ((a)|(r<<8)|(g<<16)|(b << 24))
```

Mit dem Aufruf von *Lock()* und dessen Flag DDLOCK_SURFACEMEMORYPTR stehen zwei Eigenschaften des in Lock übergebenen DDSD zur Verfügung: *ddsd.lPitch* und *ddsd.lpSurface*. Mit *ddsdlPitch* wird *memoryPitch* initialisiert, der Pointer *ddsd.lpSurface,* der auf die Adresse des am weitesten links-oben stehenden Elementes innerhalb der *Lock()* übergebenen Rechteckstruktur zeigt, wird hingegen *videobuffer* zugewiesen. Listing 5.1.3.B demonstriert dies noch deutlicher:

Listing 5.1.3.B: 8-Bit-Pixelplotting für eine 640x480-Vollbildapplikation

```
DDSURFACEDESC2 ddsd;

memset(&ddsd,0,sizeof(ddsd));      //"leeren" bzw. neu
                                   // initialisieren der DDSD
ddsd.dwSize = sizeof(ddsd);        // Das Feld dwSize einer DDSD
                                   // sollte immer gesetzt sein
lpddsprimary->Lock(NULL,&ddsd,
                   SURFACEMEMORYPTR|DDLOCK_WAIT, NULL);
  int memoryPitch = ddsd.lPitch;
  unsigned char *videobuffer8 = ddsd.lpSurface;

//Zeichne 1000 Pixel mit zufällig ausgewählter Farbe und
//zufällig ermittelte Breiten und Höhenkoordinaten
//in den Speicher des primären Surfaces
  for (int i = 0; i <1000; i++){
    unsigned char color  = rand()%256;
    int x = rand()%640;

    videobuffer8 = [x + (y * mempitch)] = color;
  }
lpddsprimary->Unlock(NULL);
```

Ein Beispielprogramm zum Pixelplotting im 8-Bit-Modus unter Windows (Pixelplotter8.cpp) und die dazugehörige ausführbare Datei (Pixelplotter8.exe) befinden sich auf der beigelegten CD im Verzeichnis \Kapitel5\

Ebenfalls in diesem Verzeichnis befindet sich eine Pixelplotter-Applikation im 16-Bit-Modus (Pixelplotter16.cpp bzw. Pixelplotter16.exe).

5.1.4. Grafikdarstellung mittels DirectX-unterstütztem Bitmap-Blitting

Als Alternative zum Pixel-Plotting besteht die Möglichkeit, außerhalb des Programms vordefinierte Pixelmatrizen, also Grafiken, in die Applikation zu integrieren (Bitmap-Blitting). BMP ist eines der von Haus aus unterstützten Formate in DirectX und das am einfachsten verständliche, weshalb es in dieser Arbeit als Beispiel verwendet werden soll.

Um eine .BMP-Grafik zu laden, sind im wesentlichen folgende Schritte abzuarbeiten
1. Laden der Grafik, dabei auslesen des Bitmap-Headers.
2. Erstellen eines Surfaces in den aus dem Header gelesenen Dimensionen.
3. Kopieren des geladenen Bitmaps in das Surface.
4. Freigeben der Grafik
Zum Darstellen einer Grafik gibt es zwei Möglichkeiten:
A: Kopieren der relevanten Region des Bitmap-Surfaces in die Zielregion des primären Surfaces.
oder
B: Kopieren der relevanten Region des Bitmap-Surfaces in die Zielregion eines Hintergrund-Surfaces, in dem Grafiken zum nächsten anzuzeigenden Bild kombiniert

werden (double buffering). Anschließend vertauschen der Pointer des Hintergrund-Surfaces mit dem des primären Surfaces (page flipping).

Laden und Freigeben von Bitmap-Grafiken:
An dieser Stelle soll nur anhand grober Stichpunkte erläutert werden, wie das Laden und Freigeben von Grafiken funktioniert. Die Implementation der letztlich verwendeten Funktionen *LoadBitmapFile(),UnloadBitmapFile()* und *FlipBitmap()* ist in den Beispielprogrammen zum Bitmap-Blitting (CD: "/Kapitel5/LoadBitmap8.cpp" oder "/Kapitel5/LoadBitmap16.cpp") zu finden.

LoadBitmapFile() - öffnet eine Bitmap-Datei, liest deren Header-Informationen aus und speichert diese in der temporären Struktur *BITMAP_FILE_TAG* (siehe Listing 5.1.4.A). Anhand der Daten der *BITMAP_FILE_TAG*-Struktur lassen sich Informationen bzgl. der Art und Größe der Grafik ableiten und z.B. auf ein Surface übertragen, dass die Bitmap enthalten soll.

Listing 5.1.4.A: Die Struktur *BITMAP_FILE_TAG* zum Speichern von Header-Informationen einer BMP-Datei

```
typedef struct BITMAP_FILE_TAG {       // Containerstruktur für
                                       // .BMP-Dateien
  BITMAPFILEHEADER bitmapfileheader;   // Informationen über den
                                       // Typ, die Größe und das
                                       // Layout der Datei
  BITMAPINFOHEADER bitmapinfoheader;   // Informationen über die
                                       // Dimensionen und das
                                       // Farbformat der Bitmap
                                       //(beinhaltet die Palette)
  PALETTEENTRY     palette[256];       // Array zum Speichern
                                       // evtl.vorhandener
                                       // 256-Farben-Paletten
  UCHAR            *buffer;
// Pointer auf Daten des BMP-Files
} BITMAP_FILE, *BITMAP_FILE_PTR;
```

LoadBitmapFile() eignet sich für Bitmap-Dateien mit der Farbtiefe 8, 16 und 24 Bit, wobei 24-Bit-Grafiken auf 16-Bit konvertiert werden, denn der 24-Bit-Modus wird, wie schon erwähnt, von den wenigsten Grafikkarten unterstützt.

FlipBitmap() - dreht die Bitmap um. Es ist üblich, dass BMP-Dateien vertikal gespiegelt gespeichert werden, also die unterste bzw. letzte Zeile wurde zuerst gespeichert, dann die vorletzte usw. In *FlipBitmap()* wird die Bitmap wieder so gedreht, dass die Grafik richtig herum im Puffer der *BITMAP_FILE_STRUCT* abgelegt ist.

UnloadBitmapFile() - gibt den Pufferspeicher der Struktur *BITMAP_FILE_TAG* (*buffer*) wieder frei.

Um nun eine Grafik für ein Programm verfügbar zu machen, muss sie in einem sekundären Surface mit den Dimensionen der Bitmap abgelegt werden. Für das

Anlegen eines sekundären Surfaces und das Kopieren der Bitmap-Daten in dieses Surface sind folgende vier Schritte nötig:

1.Schritt: Initialisierung einer DirectDrawSurfaceDescription (DDSD):
Wie schon in Abschnitt 5.1.2. demonstriert, wird ein Surface mittels der Funktion *CreateSurface()* erstellt. *CreateSurface()* erwartet unter anderem eine Surface-Beschreibungsstruktur vom Typ DDSD als Parameter, in der die Eigenschaften des Surfaces definiert sind. Listing 5.1.4.B zeigt, welche Eigenschaften der DDSD für ein sekundäres Surface festzulegen sind:

Listing 5.1.4.B: Eigenschaften einer "DirectDrawSurfaceDescription" für ein sekundäres Surface

```
DDSURFACEDESC2  ddsd;

memset(&ddstruct,0,sizeof(ddstruct));
ddstruct.dwSize=sizeof(ddstruct);
ddsd.dwFlags   = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT;
ddsd.dwWidth  = bitmapWidth;
ddsd.dwHeight = bitmapHeight;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
```

bitmapWidth und *bitmapHeight* sind Variablen, in denen die Größe der Bitmapgrafik gespeichert ist. Bis jetzt wurde von dem Standpunkt ausgegangen, dass in einer Bitmap-Grafik nur ein Bild gespeichert ist, dass im Gesamten in ein Surface übertragen werden soll. In diesem Fall können die Werte *biWidth* und *biHeight* der *BITMAP_FILE_TAG*-Struktur der DDSD zugewiesen werden (Hinweis: *biWidth* und *biHeight* repräsentieren die Länge und Breite der Bitmap in Pixeln). Es ist aber durchaus möglich, dass eine Grafik aus lauter einzelnen Stücken besteht, die mitunter unterschiedlich groß sein können (siehe Abbildung 5.1.4.A). Um für diese Objekte dann ein Surface zu erstellen, muss der Programmierer sowohl deren Größe als auch deren Lage in der Bitmap-Datei kennen und in der Applikation berücksichtigen, denn die in der *BITMAP_FILE_TAG*-Struktur gespeicherten Werte beziehen sich ja nur auf die gesamte Bitmap und sind so nicht verwendbar.

Abbildung 5.1.4.A: Zusammenfassung von Animationsgrafiken und Bildstücken in einer BMP-Datei

Zudem wurde der Struktur *ddsd* die Eigenschaft DDSCAPS_OFFSCREENPLAIN zugewiesen, dass zu erzeugende Surface soll also ein sekundäres Surface sein, dass

im Gegensatz zum primären Surface nur im Speicher existiert und nicht auf dem Bildschirm zu sehen ist.

2.Schritt: Anlegen des Surfaces und eventuelles Festlegen eines Farbschlüssels (color key):
Mittels *CreateSurface()* wird nun das sekundäre Surface erstellt (siehe Listing 5.1.4.C).

Listing 5.1.4.C: Erstellen eines sekundären Surfaces und setzen des Farbschlüssels (color key)

```
lpddsGraphic = NULL;
lpdd->CreateSurface(&ddsd,&lpddsGraphic,NULL);

DDCOLORKEY color_key;
color_key.dwColorSpaceLowValue  = color_key_value;
color_key.dwColorSpaceHighValue = color_key_value;
lpddsGraphic->SetColorKey(DDCKEY_SRCBLT, &color_key);
```

Optional kann noch ein Farbschlüssel (color key) definiert werden. Wird der Inhalt dieses Surfaces in ein anderes Surface übertragen, wird der als Farbschlüssel angegebene Wert nicht mit kopiert. Farbschlüssel können entweder RGB-Farbbereiche in den Grenzen von *dwColorSpaceLowValue* und *dwColorSpaceHighValue* sein oder einzelne RGB-Farbwerte. Bei letzterem sind die Farbobergrenze und -untergrenze identisch. In Abbildung 5.1.4.B ist die Wirkung eines Farbschlüssels deutlich zu sehen. Dort wurde ein Objekt (Flugzeug), dass ursprünglich auf einen schwarzen Hintergrund gezeichnet wurde, von einem sekundären Surface in das primäre Surface kopiert. Im oberen Teil des Bildes wurde kein Farbschlüssel angegeben, im unteren Teil wurde Schwarz (RGB: 0,0,0) als Farbschlüssel definiert.

Abbildung 5.1.4.B: Visualisierung der Wirkung von Farbschlüsseln

3.Schritt: Kopieren der Bitmap-Daten aus dem Puffer der *BITMAP_FILE_TAG*-Struktur in das Surface:
Da das Verfahren, dass in Abbilding 5.1.4.A gezeigt wird, häufig Anwendung findet, soll auch im folgenden davon ausgegangen werden, dass mehrere Animationsphasen in einer Grafik zusammengefaßt sind. So soll in den Variablen *cx* und *cy* die Position einer Animationsphase (Frame) in der Grafik gespeichert sein und in *frameWidth* bzw. *frameHeight* die Länge und Breite des Frames. Listing 5.1.4.D zeigt, wie sich auf diesen Werten basierend eine Bitmap vom *BITMAP_FILE_TAG*-Puffer in ein Surface übertragen lässt.

Listing 5.1.4.D: Übertragen einer Bitmap aus dem Puffer einer *BITMAP_FILE_TAG*-Struktur in ein Surface

```
unsigned char *source_ptr;     // Pointer auf die Quellstruktur
                               // der Grafik
                               // (BITMAP_FILE_TAG-Puffer)
unsigned char *dest_ptr;       // Pointer auf die Zielstruktur
                               // der Grafik
                               // (Surface Puffer:ddsd.lpSurface)

// bitmap sei ein Pointer auf eine BITMAP_FILE_TAG-Struktur,
// die Grafikdaten enthält

// extrahiert Bitmapdaten
source_ptr = bitmap->buffer +
                     2*cy*bitmap->bitmapinfoheader.biWidth+2*cx;
```

```
//sperrt das gesamte Surface, um die Bitmap zu übertragen
lpddsGraphic->Lock(NULL,&ddsd,
                   DDLOCK_WAIT | DDLOCK_SURFACEMEMORYPTR,NULL);
dest_ptr = (UCHAR *)ddsd.lpSurface;

for (int j=0; j<frameHeight; j++){
  // iterieren durch jede Zeile der Grafik und kopieren dieser
  // in das Zielsurface
  memcpy(dest_ptr, source_ptr, 2*frameWidth);

  // Quell- und Zielpointer auf die nächste Zeile setzen
  dest_ptr   = dest_ptr   + (ddsd.lPitch);
  source_ptr = source_ptr + 2*bitmap->bitmapinfoheader.biWidth;
} // end for j

// Entsperren des Surfaces
lpddsGraphic->Unlock(NULL);
```

4.Schritt: Freigeben der Bitmap-Grafik
Zu guter letzt sollte noch die Grafik mit der Funktion *UnloadBitmap()* wieder freigegeben werden, denn sie wird im Laufe der Anwendung nicht mehr gebraucht.

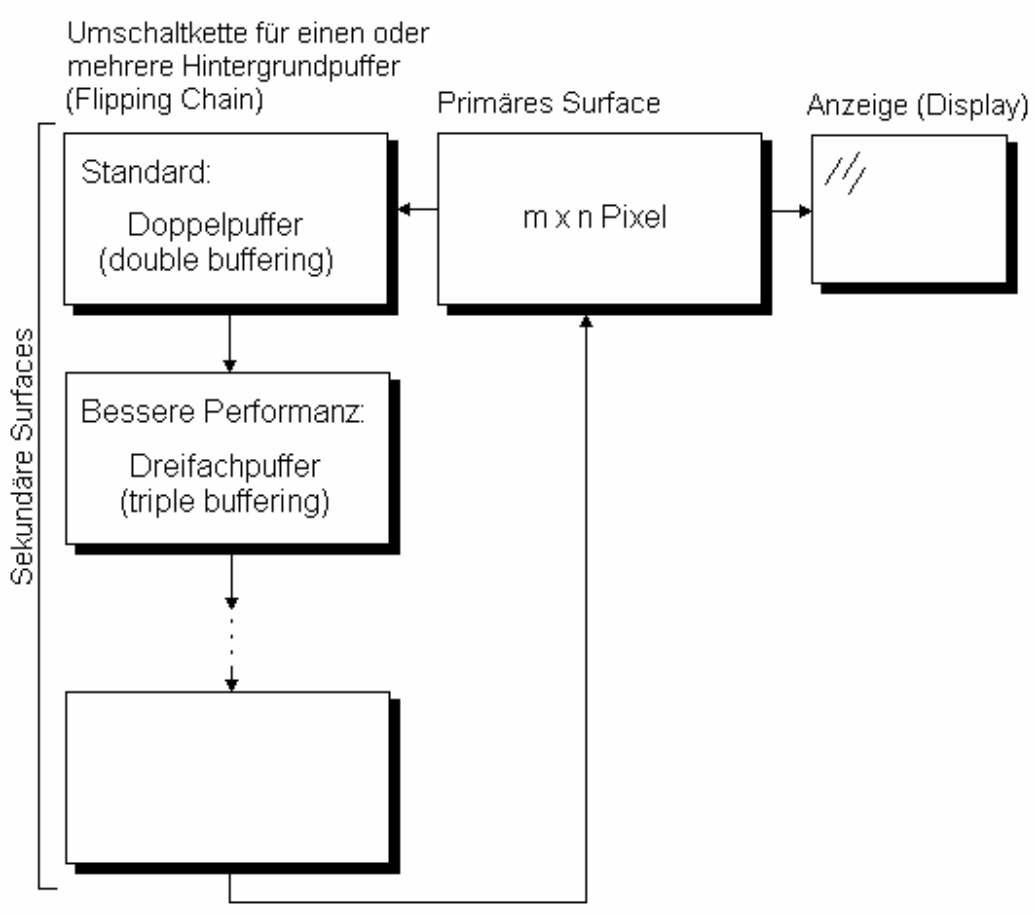
Sind alle benötigten Grafiken auf diese Weise initialisiert, können sie auf einem Surface zu einem Bild kombiniert und danach angezeigt werden. Wie schon erwähnt, gibt es mehrere Varianten, wie das arrangiert werden kann.

Die einfachste, aber auch langsamste Methode ist, alle Grafiken direkt in das primäre Surface zu kopieren. Für Applikationen mit vielen Bildwechseln ,wie zum Beispiel Computerspielen, ist es üblich, neben dem primären Surface noch ein weiteres sekundäres Surface mit der selben Geometrie und Farbtiefe wie das primäre zu erstellen. In diesem wird das Gesamtbild des Monitors aus einzelnen Surfacegrafiken kombiniert und dann die Pointer des primären Surfaces und des sekundären Surfaces vertauscht. Im nächsten Schritt wird dann auf den Speicherplatz des vorherigen primären Surfaces geschrieben, wieder die Pointer getauscht usw., jedoch bleibt das im Vordergrund stehende Surface immer das primäre und das im Hintergrund immer das sekundäre Surface. Diese Technik, die ein sekundäres Surface als Hintergrundpuffer einsetzt, heisst "double buffering" (engl., Doppelpuffertechnik). Werden für schnellere Performance zwei Surfaces als Hintergrundpuffer verwendet, in denen abwechselnd das nächste anzuzeigende Bild erstellt wird, ist das "triple buffering" engl., Dreifachpuffertechnik. Abbildung 5.1.4.C demonstriert das Prinzip des "buffering" für mehrere Hintergrundpuffer.

Einiges ist beim Erstellen der Hintergrundpuffer-Surfaces noch zu beachten. Bei einer Auflösung von 640*480 und einer Farbtiefe von 16 Bit benötigt das primäre Surface 614,400 Byte freien Speicher, jedes sekundäre Surface nocheinmal genausoviel. Für double buffering bedeutet das einen Verbrauch von immerhin 1,2MB, bei 1200*1024 und 32 Bit triple buffering sogar knapp 14 MB!! Wirkliche Geschwindigkeitsvorteile bringen die "buffering"-Techniken aber nur dann, wenn sowohl das primäre als auch die Hintergrundpuffer-Surfaces im Videospeicher (VRAM) der Grafikkarte liegen, die über entsprechend viel Speicher verfügen sollte. Kann aufgrund von Speichermangel ein

Surface nicht im VRAM untergebracht werden, lagert es DirectDraw automatisch in den Systemspeicher (RAM) aus. Zugriffe auf diese Surfaces sind deutlich langsamer, da die Hardwarebeschleunigung nur innerhalb des Videospeichers der Grafikkarte wirksam ist.

Abbildung 5.1.4.C: Prinzip der Mehrfachpufferung beim Erstellen des nächsten Bildes für das Anzeigegerät



Den Zyklus des Vertauschens der Pointer primärer und sekundärer Hintergrundsurfaces nennt man "flipping chain" (engl., etwa: Umschalt-Kette). Um eine solche zu initialisieren, sind einige Erweiterungen bei der Erstellung des primären Surfaces nötig:

1. In der DDSD-Struktur des primären Surfaces muss zusätzlich neben DDSD_CAPS noch das Flag DDSD_BACKBUFFERCOUNT gesetzt und die Anzahl der Hintergrundpuffer *dwBackBufferCount* zugewiesen werden:

```
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;

ddsd.dwBackBufferCount = 1;
// 1 für einen Hintergrundpuffer-Surface, 2 für zwei usw.
```

2. An dwCaps werden nun die Flags übergeben, die ein komplexes (), die "flipping chain" unterstützendes primäres Surface beschreiben.

```
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                      DDSCAPS_COMPLEX |
                      DDSCAPS_FLIP;
```

Bei einem komplexen Surface wird mehr als ein Surface erstellt. Die zusätzlichen Surfaces werden an das Basissurface angehangen bzw. gebunden, wie zum Beispiel die Hintergrundpuffer an das primäre Surface.

3. Wurde nach Punkt zwei dann das primäre Surface mittels *CreateSurface()* erstellt (siehe Abschnitt 5.1.2), kann die vorhandene DDSD-Struktur für die Hintergrund-Surfaces angepasst und weiter verwendet werden, um diese Surfaces dem primären zuzuweisen:

```
ddsd.ddsCaps.dwCaps = DDSCAPS_BACKBUFFER;

//Pointer auf Hintergrundpuffer
LPDIRECTDRAWSURFCAE7 lpddsback = NULL;

lpddsprimary->GetAttachedSurface(&ddsd.ddsCaps,lpddsback);
```

GetAttachedSurface() übernimmt als Parameter die ddsCaps der beschreibenden DDSD und einen Pointer auf das Hintergrundpuffer-Surface.

Um nun letztlich Grafiken kombinieren und auf dem Bildschirm anzeigen zu können, müssen die Bitmaps geblittet, also von einem auf ein anderes Surface kopiert werden. Blitten ist ein Kunstwort, dass sich von "Bit Block Transfer" ableitet und das Verschieben von Datenblöcken zwischen Surfaces meint. DirectDraw stellt zwei verschiedene Blitting-Routinen bereit: *Blt()* und *BltFast()*.

Blt() ist die umfangreichere der beiden Blit-Funktionen. Sie bietet mehr Optionen, ist dafür aber langsamer *als BltFast()*. Als Parameter sind anzugeben:
1. Eine Rechteck-Struktur, die beschreibt, in welche Region des Zielsurfaces gezeichnet werden soll.
2. Ein Pointer auf das Quellsurface.
3. Eine Rechteck-Struktur, die beschreibt, aus welcher Region des Quellsurfaces kopiert werden soll.
4. Logisch - Oder - verknüpfte Kontrolflags für das Blitting
Tabelle 5.1.4.A zeigt eine Auswahl gültiger Werte für diesen Parameter. Alle weiteren sind in der MSVisualC++-Hilfe zu finden.
5. Ein Pointer auf eine DDBLTFX-Struktur, ähnlich der DirectDrawSurfaceDescription (DDSD), in der weitere Optionen zum Blitten definiert werden können. Dort wird zum Beispiel der Einsatz und die Verwendung von Alphamasken festgelegt, also Masken, die den Opazitätswert (Transparenz) einzelner Pixel bestimmen. Einige der Kontrolflags unter Punkt 4 beziehen sich auf eine solche DDBLTFX-Struktur, zum Beispiel DDBLT_COLORFILL, dass gesetzt wird, wenn das Zielsurface mit der im DDBLTFX-Parameter dwFillColor definierten Farbe gefüllt werden soll. Soll keine DDBLTFX-Struktur verwendet werden, ist hier *NULL* zu übergeben.

Tabelle 5.1.4.A: Auswahl möglicher Werte für die dwFlags der Funktion Blt()

Wert	Beschreibung
DDBLT_WAIT	wartet, bis das Blitting durchgeführt werden kann und nicht mehr die Fehlermeldung DDERR_WASSTILLDRAWING zurückgibt
DDBLT_KEYSRC	Verwendet den mit dem Quellsurface assoziierten Farbschlüssel

Der Aufruf von *Blt()* kann zum Beispiel so aussehen:

```
lpddsback->Blt(&dest_rect,lpddsGraphic,
               &source_rect,(DDBLT_WAIT | DDBLT_KEYSRC), NULL);
```

In das Hintergrund-Surface *lpddsback* wird vom Grafiksurface lpddsGraphic die in *source_rect* festgelegte Speicherregion an die in *dest_rect* übergebene Stelle kopiert. Sind *source_rect* und *dest_rect* unterschiedlich groß, so wird die Grafik in *lpddsback* skaliert dargestellt.

BltFast() ist weniger mächtig als *Blt()*, dafür aber auch schneller. Es übernimmt als Parameter:
1. Die x-Koordinate im Zielsurface, ab der gezeichnet werden soll.
2. Die y-Koordinate im Zielsurface, ab der gezeichnet werden soll.
3. Einen Pointer auf das Quellsurface.
4. Eine Rechteck-Struktur, die beschreibt, aus welcher Region des Quellsurfaces kopiert werden soll.
5. Logisch - Oder - verknüpfte Kontrolflags, die näher beschreiben, wie geblittet werden soll.

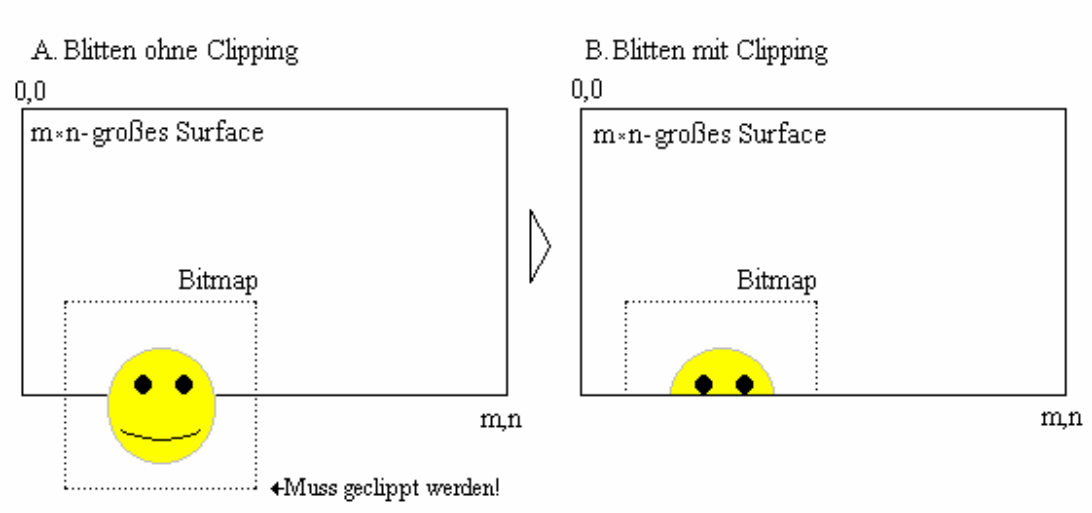
Der Aufruf von *BltFast()* könnte zum Beispiel so aussehen:

```
lpddsback->BltFast(x, y, lpddsGraphic,
                   &source_rect, (DDBLT_WAIT | DDBLT_KEYSRC));
```

BltFast() führt nur einen einfachen Bitblock-Transfer durch, es ist also zum Beispiel nicht in der Lage, zu skalieren oder Alphamasken zu verwenden. Diese Funktion sollte nur dann verwendet werden, die Grafik des Quellsurfaces 1:1 übernommen werden soll und dabei nicht geclippt (Clipping, siehe Abschnitt 5.1.2. IDirectDrawClipper) wird.

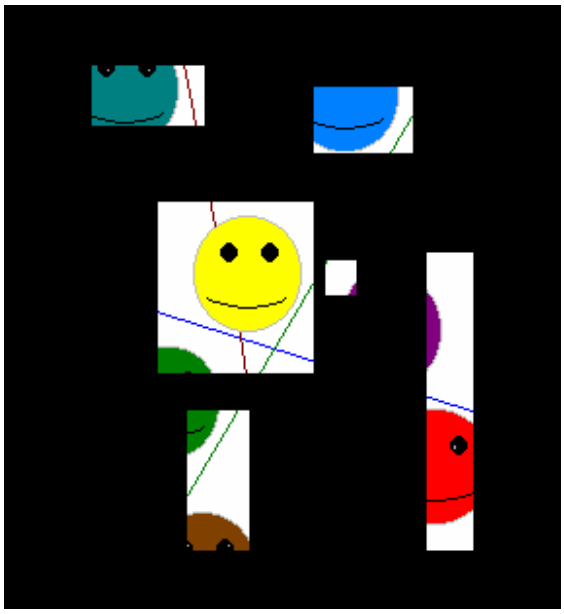
Wie schon erwähnt, stellt DirectDraw auch Funktionen für das Clipping von Grafiken zur Verfügung. Warum Clipping notwendig ist, soll Abbildung 5.1.4.D demonstrieren:

Abbildung 5.1.4.D: Das Problem des Bitmap-Clipping



Bei einer Bitmap, die über die Grenzen des Zielsurfaces hinausgeht, würden nicht geclippte Teile in ungültige Speicherbereiche geschrieben werden. Clipper in DirectDraw haben die Aufgabe, nur die Bereiche einer Bitmap aus dessen Quellsurface zu übernehmen, die sich innerhalb der Grenzen des Clipping-Bereiches befinden. Abbildung 5.1.4.D zeigte ein Beispiel für Rand-Clipping. Es ist aber auch möglich, Clipper so zu setzen, dass nur bestimmte Regionen eines Surfaces dargestellt werden, wie Abbildung 5.1.4.E zeigt:

Abbildung 5.1.4.E: Festlegen mehrerer Clipping-Regionen für ein Surface



Prinzipiell werden Rechtecke angegeben, die die Grenzen des darzustellenden Bereiches definieren. Alles außerhalb dieser Rechtecke wird geclippt. Um einen Clipper zu initialisieren, ist folgendes zu tun:

1. Einen DirectDraw-Clipper-Objekt erstellen:

```
LPDIRECTDRAWCLIPPER lpddclipper;
lpdd->CreateClipper(NULL,&lpddclipper,NULL);
```

An *CreateClipper()* wird eigentlich nur die Adresse des Schnittstellen-Pointers übergeben, Parameter 1 und 3 hingegen werden nicht verwendet und sind auf *NULL* zu setzen.

2. Eine Liste mit Clipping-Rechtecken definieren. Die Anzahl dieser Rechtecke muss von vornherein bekannt sein und in *num_rect* zur Verfügung stehen. In clip_list ist die eigentliche Liste der Clipping-Rechtecke gespeichert. Sie könnte zum Beispiel so aussehen:

```
Rect clip_list[3] = {{10,10,50,50},
                     {100,100,200,200},
                     {300,300,500,450}};
```

3. Dem Clipper-Objekt die Clipping-Liste zuweisen und die Grenzen der gesamten Clipping-Region ermitteln:

Listing 5.1.4.E: Definition der Clipping-Regionen für DirectDraw-Clipper

```
LPRGNDATA region_data;
region_data = (LPRGNDATA)malloc(sizeof(RGNDATAHEADER) +
               num_rects*sizeof(RECT));
memcpy(region_data->Buffer, clip_list, sizeof(RECT)*num_rects);
region_data->rdh.dwSize          = sizeof(RGNDATAHEADER);
region_data->rdh.iType           = RDH_RECTANGLES;
region_data->rdh.nCount          = num_rects;
region_data->rdh.nRgnSize        = num_rects*sizeof(RECT);
region_data->rdh.rcBound.left    =
region_data->rdh.rcBound.top     =  64000;
region_data->rdh.rcBound.right   =
region_data->rdh.rcBound.bottom = -64000;

for (int i=0; i<num_rects; i++){
  if (clip_list[i].left < region_data->rdh.rcBound.left){
    region_data->rdh.rcBound.left = clip_list[i].left;
  }
  if (clip_list[i].right > region_data->rdh.rcBound.right){
    region_data->rdh.rcBound.right = clip_list[i].right;
  }
  if (clip_list[i].top < region_data->rdh.rcBound.top){
    region_data->rdh.rcBound.top = clip_list[i].top;
  }
  if (clip_list[i].bottom > region_data->rdh.rcBound.bottom){
    region_data->rdh.rcBound.bottom = clip_list[i].bottom;
  }
} // end for i
```

Wurden die Parameter der RGNDATA-Struktur festgelegt, ist die Funktion *SetClipList()* aufzurufen. *SetClipList()* weist dem in Punkt 1 erstellten Clipper-Objekt die mittels RGNDATA generierte Clipliste zu. Anschließend kann die RGNDATA-Struktur freigegeben werden.

```
lpddclipper->SetClipList(region_data, 0);
free(region_data);
```

4. Den Clipper einem Fenster oder einem (sekundären) Surface zuweisen.

```
lpddsback->SetClipper(lpddclipper);
```

weist dem Hintergrundsurface *lpddsback* die eben definierten Clipping-Regionen zu.

5.2. Möglichkeiten für Pocket-PCs

5.2.1. Bildschirmauflösungen und Farbmodi für Pocket-PCs

Ein Standard-Pocket-PC hat eine Auflösung von 240x320 Bildpunkten (=76.800 Rasterpunkte), also ein Display (hier: Anzeigebildschirm eines Pocket-PC), dessen Höhe größer ist als dessen Breite, ganz entgegen dem Bildschirm des Desktop-PCs. Dieses Pixelraster ist fixiert, eine Umstellung auf eine andere Auflösung ist nicht möglich. Für Displays, die nicht den Standardkonventionen entsprechen, stellt die GameAPI ab Version 1.2. die Funktionen *GXIsDisplayDRAMBuffer()* und *GXSetViewport()* bereit. *GXIsDisplayDRAMBuffer()* überprüft, ob das Display des Pocket-PCs die Standardauflösung von 240x320 Bildpunkten aufweist und gibt einen boolschen Wert zurück. GXSetViewport() wird dafür verwendet, einen Sichtbereich (Viewport) innerhalb des verfügbaren Displays zu definieren, in dem das tatsächlich sichtbare Bild dargestellt wird. Diese Funktion ist aber noch nicht ausgereift und zeigt auf den meisten Pocket-PCs keine Wirkung.
Des Weiteren haben aktuelle Geräte standardmässig eine Farbtiefe von 16-Bit, so dass 65.536 Farben zur Verfügung stehen. Auch 8-Bit-Grafiken lassen sich darstellen, indem sie intern konvertiert werden, doch dazu mehr in Abschnitt 5.2.4.

5.2.2. Initialisierung der GameAPI-Grafikfunktionen

Während bei DirectDraw ein komplexes Set von Funktionen eine Surface-Architektur simuliert (siehe Abschnitt 5.1.2.), bietet die GameAPI lediglich einige Hardware-nahe Funktionen, über die zum Beispiel direkt auf den Videospeicher zugegriffen werden kann. Sie wird über die Headerdatei gx.h in eine Applikation eingebunden. Da Pocket-PCs keine Grafikkarten mit eigenem VRAM besitzen, ist der Videospeicher auf diesen Geräten als Teil des Systemspeichers zu betrachten.
Surfaces müssen vom Programmierer mittels GDI-basierter Funktionen selbst entwickelt werden, nur das primäre Surface lässt sich über die GameAPI-Routinen *GXOpenDisplay()* und *GXCloseDisplay()* initialisieren. Über *GXGetDisplayProperties()* werden die Bildschirmeigenschaften ermittelt bzw. manipuliert. Listing 5.2.2.A zeigt, wie für eine Anwendung der Vollbildmodus festgelegt wird.

Listing: 5.2.2.A Festlegen des Vollbildmodus für eine GameAPI-Applikation

```
GXDisplayProperties gdp;              //GameAPI-Struktur für
                                      //Displayeigenschaften
HWND  ghwnd;                          //"Windowshandle"

GXOpenDisplay(ghwnd,GX_FULLSCREEN);   //Initialisieren des
                                      //Vollbildmodus

// Speichern der Displayeigenschaften in der GXDisplayProperties-
// Struktur gdp
gdp = GXGetDisplayProperties();

// Testen auf 16-Bit-Display und 5.6.5.RGB-Format
if(!(gdp.cBPP = 16) || !(gdp.ffFormat | kfDirect565)){

  // Nachrichtenfenster mit Fehlermeldung
  MessageBox(ghwnd,
             _T("16 Bit Farbdisplay erforderlich!"),
             _T("Problem: "),
              MB_ICONEXCLAMATION | MB_OK);

  // Zugriff auf Display(-eigenschaften) beenden
  GXCloseDisplay();
  return(0);
}

// Zugriff auf Display(-eigenschaften) beenden
GXCloseDisplay();
```

Die Struktur GXDisplayProperties enthält sechs Parameter, die für bestimmte Eigenschaften des Displays stehen:

```
struct GXDisplayProperties{
  DWORD cxWidth;          // Anzahl sichtbarer Pixel pro Linie
                          // des Displays
  DWORD cyHeight;         // Anzahl sichtbarer Pixel entlang
                          // der Höhe des Displays
  long cbxPitch;          // Breite des Videopuffers
                          //(wichtig für Pixelzeilenwechsel)
  long cbyPitch;          // Höhe des Videopuffers
                          //(wichtig für Pixelspaltenwechsel)
  long cBPP;              // Farbtiefe in Bits pro Pixel
  DWORD ffFormat;         // Informationen über die Bitaufteilung
                          // pro Pixel, bei kfDirect565 z.B.
}                         // 5 rot, 6 grün, 5 rot
```

Das primäre Surface, das unter DirectX den Videopuffer repräsentiert hat, wird in GameAPI-Applikationen nicht durch ein eigenes Objekt definiert, sondern es werden vielmehr durch den Aufruf von GXOpenDisplay die Formateigenschaften des primären Gerätekontextes (PDC) bestimmt. Ein Gerätekontext (device context: DC) ist eine GDI-verwaltete Struktur, die Informationen über die Operationsmodi und die derzeit gültige Geräteauswahl (z.B. Display, Drucker,...) enthält. Die Klasse, in der Gerätekontexte implementiert sind, heisst CDC und beinhaltet Funktionen, mit denen sämtliche

grafischen Ausgaben dargestellt werden. Der primäre Gerätekontext entspricht vom Prinzip her einem primären Surface unter DirectDraw. Die Funktion *GXOpenDisplay()* legt ihn an und definiert zudem einen bestimmten Bereich im Videospeicher, der den Videopuffer repräsentiert.

5.2.3. GameAPI-unterstütztes Pixelplotting

Das Gegenstück zu den DirectDraw-Funktionen *Lock()* und *Unlock()* ist das Funktionspaar
GXBeginDraw() und *GXEndDraw().* Wird *GXBeginDraw()* aufgerufen, gibt es einen Pointer auf den Videopuffer zurück und sperrt diesen, um Zugriffe anderer Prozesse auf den Puffer zu unterbinden. *GXEndDraw()* muss aufgerufen werden, um die Ressourcen, die *GXBeginDraw()* erzeugt hat, wieder freizugeben. Ansonsten würde die Applikation "abstürzen". Es ist empfehlenswert, beide Routinen immer paarweise aufzurufen, besser noch innerhalb der gleichen Funktion, so wie im Beispiel in Listing 5.2.3:

Listing 5.2.3.A: Einsatz der GameAPI-Funktionen *GXBeginDraw()* und *GXEndDraw()* zum Sperren des Videopuffers

```
struct COLOR {
//Struktur für 24-Bit-RGB-Werte
  BYTE red;
  BYTE green;
  BYTE blue;
};

COLOR color;
color.red   = color.green = color.blue  = 0;    //Schwarz

unsigned char *VidMem;
unsigned int x, y;          //x,y-Koordinaten eines Pixels
                            // auf dem Display

int PlotPixel(int x, int y, COLOR color){

  //Adresse des Pixels relativ zur (0,0)-Koordinate
  int adress = (x * gdp.cbxPitch) + (y * gdp.cbyPitch);

  //Variable für auf 16-Bit konvertierten RGB-Wert
  unsigned short usColor;

  // Beginne Zeichnen in Videospeicher
  VidMem = (unsigned char *)GXBeginDraw();

  //Konvertierung der RGB-Werte
  usColor = (unsigned short) (((color.red & 0xf8) << 8) |
                              ((color.green & 0xfc) << 3) |
                              ((color.blue & 0xf8) >> 3));
  *(unsigned short *)(VidMem + adress) = usColor;

  GXEndDraw();        // Beende Zeichnen in Videospeicher
}
```

Der in Listing 5.2.3.A gezeigte Quellcodeabschnitt kann zum Beispiel für einen Pixelplotter (siehe Abschnitt 5.1.3) verwendet werden. In einer solchen Applikation sind die Farbwerte der Struktur *COLOR* per Zufall zu generieren, der Videopuffer mittels *GXBeginDraw()* und *GXEndDraw()* zu sperren, und zufällig ermittelten Pixeln Farben zuzuordnen. Eine Beispielapplikation zu diesem Thema (Pixelplotter16CE.cpp) und die dazugehörige ausführbare Datei (Pixelplotter16CE.exe) befinden sich auf der beigelegten CD im Verzeichnis \Kapitel5\.

5.2.4. GameAPI-unterstütztes Bitmap-Blitting

Wie bei DirectX findet auch unter WindowsCE das Surface-Prinzip Anwendung. In Abschnitt 5.2.2. wurde schon erwähnt, dass der Videopuffer als Äquivalent zum primären Surface durch die Funktion *GXOpenDisplay()* initialisiert wird. Für nicht auf dem Bildschirm sichtbare Speicherbereiche ähnlich den sekundären DirectDraw-Surfaces gibt es zwei Ansatzmöglichkeiten, je nachdem, welche Funktionalität diese besitzen sollen. Die erste Möglichkeit besteht darin, einen Hintergrundpuffer zu schaffen, wie er bei der Doppelpuffer-Technik (siehe Abschnitt 5.1.4.) Verwendung findet.
Der Codeabschnitt

```
HDC hBackbufferDC;          // Instanz eines Gerätekontextes

// einen zum primären DC kompatiblen Gerätekontext definieren
hBackbufferDC = CreateCompatibleDC(hPDC);
```

legt eine Instanz eines Gerätekontextes (HDC) an und erstellt mittels *CreateCompatibleDC()* einen neuen zum *PDC* kompatiblen Gerätekontext (memory device context). Kompatibel bedeutet in diesem Fall, dass beide Gerätekontexte in ihren Eigenschaften (z.b. Vollbildmodus) identisch sind. Wird mittels *CreateCompatibleDC()* ein neuer Gerätekontext erstellt, setzt GDI den Pufferspeicher automatisch auf die Größe eines 1x1-Pixel großen monochromen Standard-Bitmaps. Daher muss ein Speicherbereich reserviert werden, der die tatsächlichen Dimensionen des Videopuffers besitzt. Dieser Bereich ist vom Typ *HBITMAP*, er soll ja später auch Bitmap-Grafiken beinhalten.

```
HBITMAP hBackbufferBitmap;
//Bitmap-Pointer

// einen zum primären DC kompatiblen Speicherblock definieren.
hBackbufferBitmap = CreateCompatibleBitmap(hPDC, Width, Height);
```

CreateCompatibleBitmap() erstellt einen PDC-identischen Speicherbereich, als weitere Parameter ist die Größe, also die Länge und Breite des Bereiches anzugeben.

Des Weiteren ist der Speicherbereich noch auf den gewünschten Gerätekontext zu übertragen und das anschließend nicht mehr gebrauchte *hBackbufferBitmap* zu löschen.

```
hOldBitmap = SelectObject(hBackbufferDC, hBackbufferBitmap);
DeleteObject(hBackbufferBitmap);
```

Die zweite Art von nicht sichtbaren Gerätekontexten sind solche, die Grafiken aufnehmen sollen. Wie schon bei den DirectDraw-Routinen werden auch unter WindowsCE Grafiken aus Dateien geladen, in einem speziellen Speicherbereich abgelegt und die Dateien anschließend wieder freigegeben. Das Laden von Grafiken stützt sich nicht auf Funktionen der GameAPI und soll daher nur grob tangiert werden. Eine mögliche Umsetzung wird in der Funktion *LoadDIB()* demonstriert. *LoadDIB()* wurde vorrangig für 8- und 16-Bit-Grafikdateien entworfen, liest dort den Bitmapheader aus, reserviert dementsprechend Speicher für die Grafikdaten, konvertiert evtl. vorhandene Palettenfarben in RGB-Werte, legt die Bilddatei-Header temporär ab und liefert einen Pointer *lpSrcDIB* auf die BITMAPINFOHEADER-Struktur der geladenen BMP-Datei. Als Parameter übernimmt *LoadDIB()* den vollen Pfad und Namen einer Grafikdatei:

```
LPBITMAPINFO lpSrcDIB;
LoadDIB("\\unterverzeichnis\\beispiel.bmp");
```

Als nächstes wird mittels *CreateDIBSection()* ein Bereich definiert, der zum einen die Dimension der geladenen Grafik haben soll und zum anderen die Bilddaten in ein DIB konvertiert. DIB steht für device independent bitmap, also geräteunabhängiges Bitmap. Damit ist gemeint, dass die Farben des Bildes intern fest definiert sind, zum Beispiel als RGB-Wert, und damit auf jedem System gleich aussehen. Im Gegensatz dazu legt das System die Farben für geräteabhängige Bitmaps (DDB) fest, so dass diese schwer zu portieren sind. *CreateDIBSection()* gibt einen Pointer auf diesen Bereich zurück und übernimmt als Parameter:
1. Einen Pointer zu einem Gerätekontext, dessen logische Farbpalette verwendet wird, für den Fall, dass eine palettengestützte Grafik geladen werden soll.
2. Den Pointer, der innerhalb der Funktion *LoadDIB()* ermittelt wurde, und auf die BITMAPINFOHEADER-Struktur der geladenen BMP-Datei.
3. Ein Flag, das angibt, ob die geladene Datei Farbpaletten verwendet oder konkrete RGB-Werte.
LoadDIB() wurde bereits so umgesetzt, dass es Palettenfarben in RGB-Farben konvertiert, so dass an dieser Stelle *DIB_RGB_COLORS* angegeben werden muss.
4. Einen Pointer zu einer Variablen, die einen Pointer auf die Adresse der tatsächlichen Bilddaten empfangen soll.
5. + 6. NULL, da diese Parameter für spätere Nutzung reserviert wurden, momentan aber noch nicht implementiert sind.
Der Aufruf von *CreateDIBSection()* für dieses Beispiel ist folgender:

```
HBITMAP hSourceBitmap;
LPBYTE lpSourceBits;
hSourceBitmap = CreateDIBSection(hPDC(),
                                 lpSrcDIB,
                                 DIB_RGB_COLORS,
                                (void **)&lpSourceBits,
                                 NULL,NULL);
```

Für das in *CreateDIBSection()* erzeugte Bitmap muss nun ein Gerätekontext geschaffen werden, der zum primären kompatibel ist. In diesen wird dann die DIB-Grafik abgelegt. Dazu wird erst die Größe der Grafik errechnet, anschließend in den durch

CreateDIBSection() definierten Bereich kopiert und letztlich in den Gerätekontext der Grafik übernommen:

```
//Erstellen des kompatiblen Gerätekontextes
hSourceDC = CreateCompatibleDC(hPDC());

DWORD dwSourceBitsSize;
dwSourceBitsSize = lpSrcDIB->bmiHeader.biHeight *
                   BytesPerLine(&(lpSrcDIB->bmiHeader));
memcpy(lpSourceBits, BitmapDataIndex((LPSTR)lpSrcDIB),
dwSourceBitsSize);
SelectObject(hSourceDC, hSourceBitmap);
```

Für spätere Nutzung des Bitmap-Headers ist dieser noch zu speichern:

```
BITMAP bmBitmap;

//Speichern des Bitmap-Header in bmBitmap
GetObject(hSourceBitmap, sizeof(BITMAP), &bmBitmap);
```

Letztlich sind noch die nicht mehr gebrauchten Ressourcen freizugeben:

```
DeleteObject(hSourceBitmap);
free(lpSrcDIB);
```

Nachdem auf diese oder ähnliche Weise alle Grafiken in Gerätekontexte geladen wurden, können sie nun auf dem Bildschirm dargestellt werden. Dafür stehen vier verschiedene Blitting-Operationen zur Verfügung, die Bitblöcke von jedem beliebigen Gerätekontext in jeden anderen kopieren können. Wie bei DirectDraw können die Grafiken auch hier entweder direkt in den primären Gerätekontext "geblittet" oder zuerst in einem Hintergrundpuffer kombiniert und erst anschließend in den Videopuffer übernommen werden. Allerdings müssen die Daten des Hintergrundpuffers direkt in den Videopuffer kopiert werden, "page flipping", also das Vertauschen der Pointer der beiden Puffer, wird an dieser Stelle nicht unterstützt. Die vier Blitting-Operationen sind:

```
1. BOOL BitBlt(HDC hdcDest,              //Zielgerätekontext
               int nXDest,
               int nYDest,               //Ziel-
               int nWidth,
               int nHeight,              //rechteck
               HDC hdcSrc,               //Quellgerätekontext
               int nXSrc, int nYSrc,     //Ausgangspunkt für
                                         // Quellrechteck
               DWORD dwRop );            //Flag für Raster-
                                         //Operationen
```

BitBlt() ist die Standard-Blit-Operation und findet dann Anwendung, wenn ein bestimmter Rechteck-Bereich, definiert durch den Punkt (*nXSrc*, *nYSrc)* und die Länge (*nWidth*) und Breite(*nHeight*), in der Größe unverändert von einem in einen anderen Gerätekontext verschoben werden soll. Über den Parameter *dwRop* wird angegeben, in welcher Art und Weise die im Zielkontext vorhandenen Pixel genutzt werden sollen, mit

SRCCOPY werden sie zum Beispiel überschrieben, mit *SRCAND* werden die Pixel beider Kontexte logisch-AND-verknüpft.

```
2. BOOL StretchBlt(HDC hdcDest,          //Zielgerätekontext
                   int nXOriginDest,
                   int nYOriginDest,     //Ziel-
                   int nWidthDest,
                   int nHeightDest,      //rechteck
                   HDC hdcSrc,           //Quellkontext
                   int nXOriginSrc,
                   int nYOriginSrc,      //Quell-
                   int nWidthSrc,
                   int nHeightSrc,       //rechteck
                   DWORD dwRop );        //Flag für Raster-
                                         //Operationen
```

Mit *StretchBlt()* können Grafiken gestreckt oder gestaucht werden. Dafür wird für das Zielrechteck, durch den Punkt (*nXOriginDest*, *nYOriginDest)* und der Länge (*nWidthDest*) und Breite (*nHeightDest*) beschrieben, eine andere Größe angegeben wie für das Quellrechteck (*nXOriginSrc*, *nYOriginSrc)* und der Länge (*nWidthSrc*) und Breite (*nHeightSrc*) angegeben. *StretchBlt()* skaliert dann die Grafik entsprechend.

```
3. BOOL MaskBlt(HDC hdcDest,             //Zielgerätekontext
                int nXDest,
                int nYDest,              //Ziel-
                int nWidth,
                int nHeight,             //rechteck
                HDC hdcSrc,              //Quellgerätekontext
                int nXSrc,
                int nYSrc,               //Ausgangspunkt für
                                         //Quellrechteck
                HBITMAP hbmMask,         //Pointer zu einer
                                         //Alphamaske (ext.Datei)
                int xMask, int yMask,    //Ausgangspunkt für
                                         //Maskenrechteck
                DWORD dwRop );           //Flag für Raster-
                                         //Operationen
```

MaskBlt() funktioniert im Prinzip genauso wie *BitBlt()*, bietet aber die Möglichkeit, eine Maske anzugeben, die Alphainformationen enthält. Diese Transparenz-Informationen werden dann mit den Pixeln des Quellrechtecks kombiniert und ergeben im Zielrechteck das eigentliche Bild.

```
4. BOOL TransparentImage(HDC hdcDest,    //Zielgerätekontext
                         LONG DstX,
                         LONG DstY,      //Ziel-
                         LONG DstCx,
                         LONG DstCy,     //rechteck
                         HANDLE hSrc,    //Quellgerätekontext
                         LONG SrcX,
                         LONG SrcY,      //Quell-
                         LONG SrcCx,
```

```
                    LONG SrcCy,     //rechteck
                    COLORREF TransparentColor );
                                    //Farbschlüssel
```

Soll ein Farbschlüssel eine bestimmte Farbe nicht mit zeichnen, so ist *TransparentImage()* einzusetzen. Es werden wie bei *StretchBlt()* zwei Rechteck-Bereiche über einen Punkt und die jeweiligen Längen und Breiten definiert und der Zielkontext *hdcDest* übergeben. Zusätzlich wird als TransparentColor ein Farbwert festgelegt, der beim Kopieren zwischen den Gerätekontexten nicht mit übernommen wird.

Der vollständige Quelltext, die Anwendung der einzelnen Funktionen und die Implementation der verwendeten Hilfsfunktionen sind in einer Beispielanwendung (loadBitmap16CE.cpp / loadBitmap16CE.exe) auf der CD im Verzeichnis \Kapitel5\ zu finden.

Kapitel 6: Eingabeerkennung und -auswertung

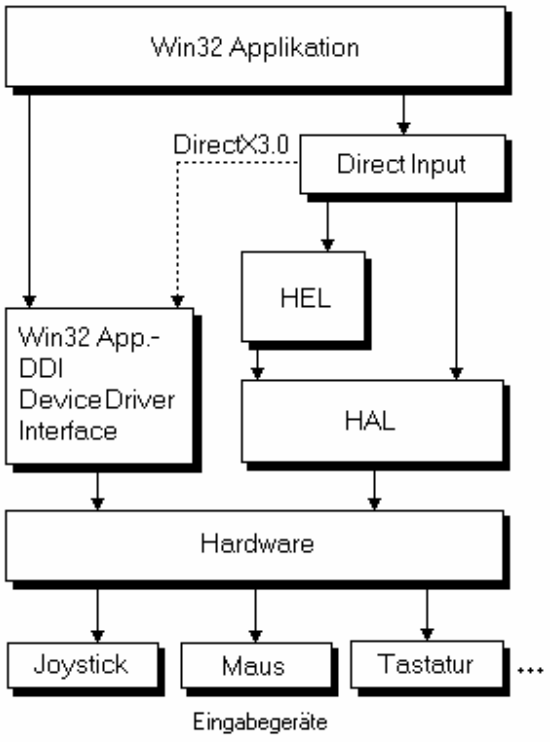
6.1. Eingabeverwaltung mittels DirectInput

6.1.1. Einführung in DirectInput

DirectInput ist die Komponente für Eingabebehandlung in DirectX. Entwickelt für die typische Input-Hardware wie Tastatur, Maus und Joystick, unterstützt DirectInput mittlerweile jedes erdenkliche Eingabegerät, für das der Hersteller einen DirectX-Treiber bereitstellt.

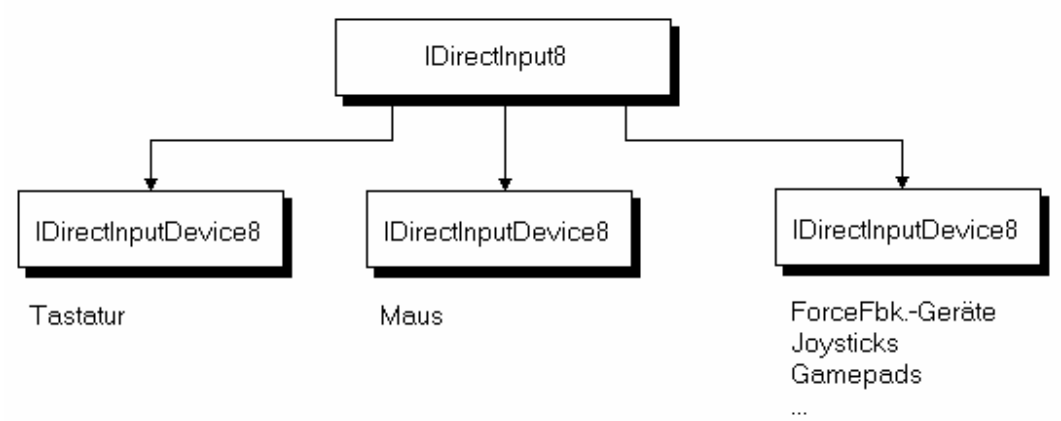
Dabei fungiert DirectInput als hardware-unabhängiges virtuelles Eingabesystem, dass im Prinzip genauso funktioniert wie DirectDraw. Die Hardwarehersteller entwickeln Treiber, DirectInput bietet die Schnittstellenfunktionen, die die Programmroutinen in Hardware-spezifische Aufrufe umwandeln. Die Beziehungen zischen DirectInput, den Treibern und den physikalischen Eingabegeräten werden in Abbildung 6.1.1.A verdeutlicht.

Abbildung 6.1.1.A: Einbindung von DirectInput als Schnittstelle zwischen Hardware und Software



Wie jedes DirectX-Subsystem besteht auch DirectInput aus einer Reihe von COM-Schnittstellen. Wie in Abbildung 6.1.1.B zu erkennen ist, enthält die hier verwendete Version 8.0 eine Hauptschnittstelle, IDirectInput8, und eine weitere Schnittstelle, IDirectInputDevice8.

Abbildung 6.1.1.B: Die DirectInput-Schnittstellen



IDirectInput8 ist das Haupt-COM-Objekt, dass zu erstellen ist, um DirectInput zu starten. Dies geschieht über die Funktion *DirectInputCreate()*. Über IDirectInput8 werden die allgemeinen Eigenschaften von DirectInput festgelegt und die Eingabegeräte initialisiert und aktiviert.

IDirectInputDevice8 ist die Schnittstelle, über die die Eigenschaften einzelner Eingabegeräte definiert werden und über die mit den Geräten kommunizert wird. Jede Tastatur, jede Maus, jeder Joystick usw. gilt als Eingabegerät (input device), dessen Eigenschaften durch je eine IDirectInputDevice8-Schnittstelle festgelegt werden können.

In den folgenden Abschnitten (6.2 und 6.3) wird die Verwendung von DirectInput am Beispiel von Tastatur und Maus erläutert. Die Initialisierung von Joystick-Geräten, Game-Pads, Force-Feedback-Lenkrädern oder ähnlichem soll an dieser Stelle aber übersprungen werden, da für Pocket-PCs keine vergleichbaren Geräte existieren.

6.1.2. Tastatureingabeauswertungen über DirectInput

Das Standardeingabegerät für Desktop-PCs ist die Tastatur, über die nicht nur alphanumerische Zeichen, sondern auch Funktionen (über Windows-Funktionstasten) und Ereignisse (zum Beispiel ESC) direkt aufgerufen werden können.

Um Eingabegeräte wie die Tastatur ansprechen zu können , muss zuerst die Hauptschnittstelle IDirectInput8 definiert werden. Wie schon erwähnt, erschafft die Funktion *DirectInputCreate()* eine solche Schnittstelle auf Basis von fünf Parametern:
1. Die Hauptinstanz der Applikation, wie sie in der Funktion *WinMain()* angegeben wurde (siehe Kapitel 4.1).
2. Die DirectInput-Version, die verwendet werden soll. Die in der Headerdatei definierte Konstante DIRECTINPUT_VERSION enthält die höchstmögliche DirectInput-Version, die auf dem System installiert ist.
3. Die Referenz-ID für die Schnittstellenversion, die verwendet werden soll.
4. Die Adresse des Schnittstellen-Pointers, der die COM-Schnittstelle zu DirectInput übernehmen soll.
5. NULL, da dieser Parameter für COM-Erweiterungen reserviert ist.

```
LPDIRECTINPUT8 = lpdi;
//Pointer zur DirectInput-COM-Schnittstelle
DirectInputCreate(hInstance, DIRECTINPUT_VERSION,
                  IID_IDirectInput8, (void **)&lpdi,NULL);
```

Für jedes Eingabegerät, dass in eine Applikation eingebunden werden soll, ist dann eine IDirectInputDevice8-Schnittstelle zu erstellen, die durch drei Parameter bestimmt wird:
1. Der GUID (Global Unique Identifier, siehe Kapitel 3.2.) für das Gerät. Für Tastatur und Maus wurden bereits Konstanten mit den entsprechenden GUID-Werten definiert, GUID_SysKeyboard für die Standard-Tastatur mit 101 Tasten, GUID_SysMouse für die Standard-Maus.
2. Der Adresswert eines Pointers auf die jeweilige IDIRECTINPUTDEVICE8-Schnittstelle.
3. NULL, da dieser Parameter für COM-Erweiterungen reserviert ist.

Der Aufruf zur Initialisierung der Tastatur wäre dann wie folgt zu realisieren:

```
//Macro, dass die Standard-GUIDs für Tastatur und Maus einbindet
#define INITGUID

LPDIRECTINPUTDEVICE8 lpdikey = NULL;
//Pointer zur Tastatur

//Der Hauptschnittstelle lpdi wird das hier erstellte Gerät
//(Tastatur) zugewiesen
lpdi -> CreateDevice(GUID_SysKeyboard, &lpdikey, NULL);
```

Als nächstes sind die Eigenschaften des Eingabegerätes und das Datenformat für die Eingabenachrichten zu definieren. Dafür stehen die Funktionen *SetCooperativeLevel()* und *SetDataFormat()* zur Verfügung. Typisch für die Tastatur wäre der Aufruf

```
lpdikey->SetCooperativeLevel(mainwinhandle,
                             DISCL_NONEXCLUSIVE |
                             DISCL_BACKGROUND);
lpdikey->SetDataFormat(&c_dfDIKeyboard);
```

Über *SetCooperativeLevel()* wird zum einen definiert, ob der Zugriff einer Applikation auf ein Eingabegerät exklusiv (keine andere Anwendung kann auf das Gerät zugreifen) oder nicht-exklusiv (andere Anwendungen können die Eingabedaten eines Gerätes auch auslesen, z.B. das System die Tastenkombination STRG+ALT+DEL) erfolgt und zum anderen, ob die Eingaben nur dann für die Applikation gültig sind, wenn das Programmfenster im Vordergrund steht, oder auch dann, wenn das Programm im Hintergrund läuft.
SetDataFormat() legt das Format fest, in dem die Eingabedaten empfangen werden sollen. Dabei werden diese Daten in verschiedenen, den Geräteklassen angepassten Strukturen abgelegt. Für die Tastatur ist dies ein 256-Zeichen großes "unsigned char"-Array.

Nachdem die Initalisierung des Eingabegerätes abgeschlossen ist, muss es noch acquiriert werden. Die dazu verwendete Funktion *Acquire()* ist mit der Funktion *Lock()* von DirectDraw zu vergleichen. Sie signalisiert, dass das Eingabegerät jetzt verwendet

wird (Voraussetzung für den Abruf der Eingabedaten) und sperrt im Exklusivmodus dieses Gerät gegenüber Zugriffen anderer Programme.

```
lpdikey->Acquire();
```

acquiriert zum Beispiel das Gerät lpdikey.

Das eigentliche Auslesen der Daten geschieht über die Funktion *GetDeviceState()*. Ihr werden als Parameter die Größe und der Name der Struktur angegeben, in die die ausgelesenen Daten abgelegt werden sollen.

```
unsigned char keystate[256];
lpdikey->GetDeviceState(256, (LPVOID)keystate);
```

Im Anschluss können die relevanten Tasten auf Ihren Status hin untersucht werden. Dazu ist zu überprüfen, ob das höchste Bit des 8-Bit-Tastenwertes (0x80) gesetzt ist, was dann zutrifft, wenn die entsprechende Taste gerade gedrückt wird. Für die Taste ESCAPE hat dieser Test zum Beispiel folgende Form:

```
// Beispiel: Ist die Taste ESCAPE gedrückt?
if (keystate[DIK_ESCAPE] & 0x80){
  /* Quelltext für Ereignisbehandlung*/
}
```

Die Bezeichnungen und Werte für die einzelnen Tasten sind in der Datei DInput.h des DirectX-Paketes definiert und dort unter dem Suchbegriff *DirectInput keyboard scan codes* zu finden.

Nach dem Auslesen der Daten bzw. spätestens bei Beenden einer Applikation sollten die Ressourcen wieder freigegeben werden. Für Eingabegeräte sind dafür drei Schritte nötig:
1. Freigeben des Zugriffs auf das Gerät:

```
  lpdikey->Unacquire();
```

2. Freigeben des Gerätes selbst:

```
  lpdikey->Release();
```

3. Freigeben des Haupt-DirectInput-Objektes:

```
  lpdi->Release();
```

6.1.3. Mauseingabeauswertung über DirectInput

Das wohl wichtigste Eingabegerät neben der Tastatur ist die Maus. Sie hat entweder zwei oder drei Tasten (buttons) und zwei Bewegungsachsen: X und Y. Wird die Maus bewegt, werden Informationspakete erstellt, die die Statusveränderungen der Maus beschreiben und (zumeist) seriell an den PC übertragen werden. Diese Daten werden

dann vom Maustreiber empfangen, bearbeitet und an Windows bzw. DirectX weitergesendet. Ein DirectX-Programmierer muss dann nur noch wissen, wie Tastendruck und Mausbewegungen auszuwerten sind.

Das Ansprechen einer Maus über DirectInput funktioniert im Prinzip genauso wie bei der Tastatur. Voraussetzung ist die Hauptschnittstelle IDirectInput8 aus dem vorherigen Kapitel (6.2). Darauf basierend wird die Maus initialisiert, aquiriert und deren Eingabedaten ausgelesen, die Reihenfolge der Schritte ist dabei dieselbe wie bei der Tastatur:
1. Erstellen der Mausschnittstelle mit *CreateDevice()*.

```
//Macro, dass die Standard-GUIDs für Tastatur und Maus einbindet
#define INITGUID

LPDIRECTINPUTDEVICE8 lpdimouse = NULL;
//Pointer zur Maus
lpdi -> CreateDevice(GUID_SysMouse, &lpdimouse, NULL);
```

2. Setzen des Kooperalionslevels mit *SetCooperativeLevel()*.

```
lpdimouse->SetCooperativeLevel(mainwinhandle,
                               DISCL_NONEXCLUSIVE |
                               DISCL_BACKGROUND);
```

3. Definieren des Datenformates mit *SetDataFormat()*.

```
DIMOUSESTATE mousestate;                    // enthält den Status
der Maus

// c_dfDIMouse bestimmt, dass die Mausdaten in einer
// DIMOUSESTATE-Struktur abgelegt werden
lpdimouse->SetDataFormat(&c_dfDIMouse);
```

Das Datenformat muss hierbei aus eine Struktur zeigen, die Mausdaten aufnehmen kann, wie zum Beispiel DIMOUSESTATE, die in DirectInput standardmässig enthalten ist. Listing 6.1.3.A zeigt den Prototypen dieser Struktur:

Listing 6.1.3.A: Prototyp der DirectInput-Struktur DIMOUSESTATE, die Mausstatusdaten speichert

```
typedef struct DIMOUSESTATE {
    LONG lX;                //Informationen über die Maus-X-Achse
    LONG lY;                //Informationen über die Maus-Y-Achse
    LONG lZ;                //Informationen über die Maus-Z-Achse
                            //(evtl. Mausrad)
    BYTE rgbButtons[4];     //Status der Maustasten
} DIMOUSESTATE, *LPDIMOUSESTATE;
```

4. Aquirieren der Maus mit *Acquire()*.

```
lpdimouse->Acquire();
```

5. Einlesen des Mausstatus über *GetDeviceState()*. Die asugelesenen Daten werden in der in Punkt 2 definierten Struktur *mousestate* abgelegt:

```
lpdimouse->GetDeviceState(sizeof(DIMOUSESTATE),
(LPVOID)mousestate);


#define MOUSE_LEFTBUTTON 0;
#define MOUSE_RIGHTBUTTON 1;

// object_x und object_y seien die Koordinaten
// eines fiktiven Objektes,
// dass in Geschwindigkeit und Richtung der Maus folgt
object_x  = object_x + mousestate.lX;
object_y  = object_y + mousestate.lY;

if (mousestate.rgbButtons[MOUSE_LEFTBUTTON] & 0x80){
    /* Anweisungen für den Fall,
       dass linke Maustaste gedrückt wurde */
}

if (mousestate.rgbButtons[MOUSE_RIGHTBUTTON] & 0x80){
    /* Anweisungen für den Fall,
       dass rechte Maustaste gedrückt wurde */
}
```

6. Freigeben der Maus über *Unaquire()* und *Release()*.

```
lpdimouse->Unacquire();
lpdimouse->Release();
/* Freigeben der Hauptschnittstelle lpdi */
```

6.2. Auswertung von Eingabe-Events auf dem Pocket-PC

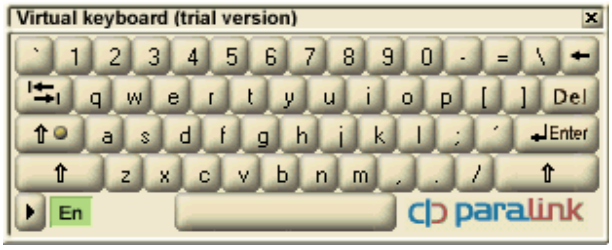
6.2.1. Einführung in die Pocket-PC-Eingabemöglichkeiten

Die Benutzerschnittstelle von Pocket-PCs bietet zwei primäre Formen der Eingabe, die Hardwareschnittstelle, die über die im Gehäuse des Gerätes integrierten Tasten oder die Berührung des Touchscreens Eingabedaten empfängt, und die Softwareschnittstelle, für die virtuelle Tastaturen und Zeichenerkennungssoftware entwickelt wurden.

Für die Hardwareschnittstelle stehen je nach Gerätetyp ein Navigationspad (oder auch Pfeiltasten), drei bzw. vier Programmtasten und diverse andere Funktionstasten wie der Startknopf und der Aufnahmeknopf für Sprachmitteilungen zur Verfügung. Diese Tasten sind meist schon mit Funktionen belegt, die GameAPI bietet dem Programmierer aber die Möglichkeit, diese vordefinierte Funktionalität zu umgehen und die Tasten je nach Bedarf selbst an eigene Funktionen zu koppeln. Auch für den Stylus, das Äquivalent der Maus am Desktop-PC, kann der Programmierer eingabesensitive Bereiche festlegen, die als virtuelle Tasten (Buttons) fungieren, oder er kann die Bewegung und Position des Stylus registrieren und auswerten.

Der Stylus ist gerade für die Softwareschnittstelle wichtig, da über ihn die im System integrierte virtuelle Tastatur bedient wird. In WindowsCE wird die vom Desktop-PC her bekannte Tastatur durch eine Software ersetzt, die eine solche Tastatur virtuell nachbildet (siehe Abbildung 6.2.1.A).

Abbildung 6.2.1.A: Beispielprogramm einer virtuellen Tastatur für Pocket-PCs

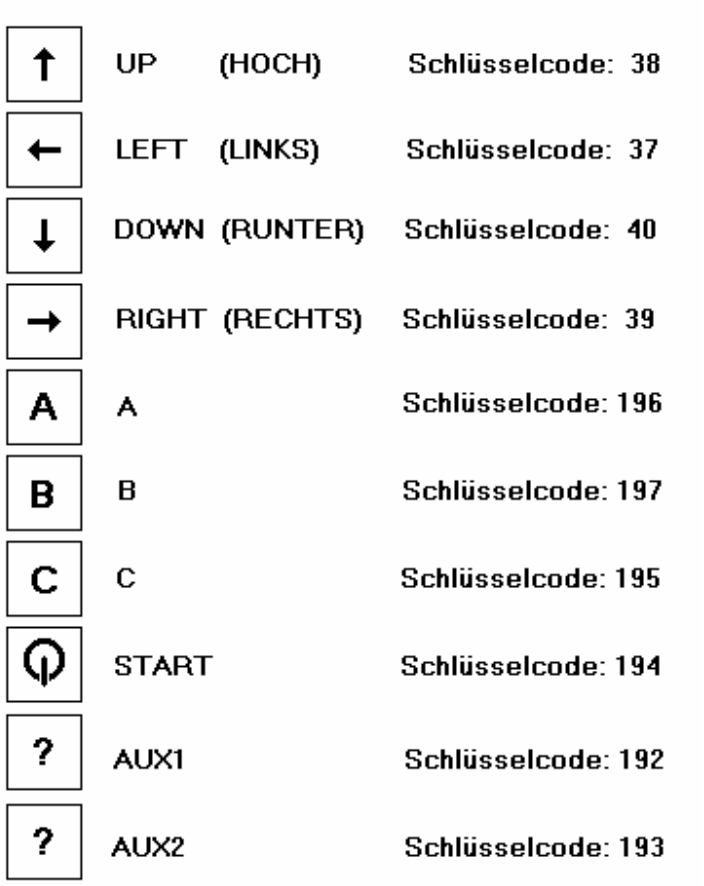


Wird mit dem Stylus ein Feld der Tastatur berührt, fügt WindowsCE das entsprechende Zeichen in die momentan im Vordergrund laufende Anwendung ein, sofern diese Zeicheneingaben erwartet (z.B. in Word für Pocket-PC). Ähnlich funktioniert auch der Zeichenerkenner (character recognizer) für den Pocket-PC. Der über den Stylus eingegebene Schriftzug wird innerhalb dieser Software in alphanummerische Zeichen konvertiert, die für das System weiterverwertbar sind. So lassen sich handschriftliche Notizen mit diesem Werkzeug in ASCII-Texte verwandeln. Da die Softwareschnittstelle schon eine fertige Interpretationssoftware für Eingaben darstellt, aber jede Anwendung Eingabeereignisse möglicherweise anders auswertet, soll in den folgenden beiden Abschnitten 6.2.2. und 6.2.3. nur auf die Initialisierung und Benutzung der durch die GameAPI unterstützten Funktionstasten und des Stylus eingegangen werden.

6.2.2. Eingabeauswertung der Funktionstasten über die GameAPI-Input-Funktionen

Die Funktionstasten eines Pocket-PCs sind meist so vorprogrammiert, dass sie auf Tastendruck eine bestimmte Applikation starten. Dies ist zwar eine nützliche Option, doch nicht immer ist die Vorbelegung der Tasten erwünscht. In einer Anwendung im Vollbildmodus wirkt es sich oft störend aus, wenn während der Laufzeit der Anwendung andere Programme gestartet werden. Desweiteren ist eine benutzerdefinierte bzw. programmspezifische Belegung der Funktionstasten wünschenswert, doch das WindowsCE-GDI bietet dafür keinerlei Funktionen. Dieses Problem behebt die GameAPI, die das Betriebssystem umgeht und hardwarenahe Funktionen zur Eingabeerkennung bereitstellt.

Um Zugriff auf die Eingabetasten zu bekommen, ist zuerst die Funktion *GXOpenInput()* aufzurufen. *GXOpenInput()* zwingt das System dazu, eingabe- bzw. tastenspezifische Systemnachrichten direkt und ungefiltert an den Eventhandler der Anwendung zu senden. Wurde zum Beispiel die Taste "hoch" gedrückt, empfängt der Eventhandler den Schlüsselcode dieser Taste, hier 38. Die GameAPI unterstützt 10 solcher Tasten:



AUX1 und AUX2 sind nicht in der GameAPI definiert, repräsentieren aber zusätzliche Tasten wie die Aufnahmetaste und eine vierte Programmtaste D, die evtl. in den Pocket-PC integriert sein könnten.

Das Gegenstück zu*GXOpenInput()* ist die Funktion *GXCloseInput()*, die den Eingabezugriff beendet und die Systemnachrichten für Eingaben wieder direkt an das System weiterleitet. *GXCloseInput()* sollte immer aufgerufen werden, wenn die Eingabeerkennung abgeschlossen wurde.

Der Aufruf

```
GXKeyList GXKeys;
GXKeys = GXGetDefaultKeys(GXNormalKeys);
```

ermittelt die ersten acht Funktionstasten und speichert diese in die *GXKeyList*-Struktur der GameAPI. Von dort können sie mit

```
short vkKeyXXX = GXKeys.vkXXX;
```

ausgelesen werden. XXX steht dabei für den Namen der Taste. Das Beispiel in Listing 6.2.2. zeigt, wie die Eingabetasten A, START und AUX2 darauf geprüft werden, ob sie im Moment gedrückt sind.

Listing 6.2.2: Abfrage auf Eingabeereignisse innerhalb des Eventhandlers am Beispiel der Funktionstasten A, START und AUX2

```
GXOpenInput();
GXKeyList GXKeys;
GXKeys = GXGetDefaultKeys(GXNormalKeys);

short vkKey;

LRESULT CALLBACK WinProc(HWND hwnd,UINT msg,WPARAM wparam,
                                            LPARAM lparam) {
 switch (message){
    case WM_KEYDOWN:             //Ereignis: Wurde Taste gedrückt?
       vkKey = (short) wParam;   //wParam enthält den virtuellen
                                 //Schlüsselcode

  //für die gedrückte Taste
  if(vkKey = = GXKeys.vkA){
    // A-Button gedrückt
    break;
  }
  if(vkKey = = GXKeys.vkStart){
    // Start-Button gedrückt
    break;
  }
  if(vkKey = = 193){             //hier Schlüsselcode für AUX2
                                 //verwenden, da AUX2 in GXKeyList
    /* AUX2 gedrückt */          //nicht definiert ist
    break;
  }
       ...
  break;
 }
}
GXCloseInput();
```

Analoge Abfragen lassen sich nun auch für die anderen Funktionstasten und das Systemereignis WM_KEYUP definieren.

6.2.3. Eingabeauswertung von Styluseingaben

Im Gegensatz zu den Funktionstasten wird die Auswertung des Stylus nicht über die GameAPI initialisiert, sondern an bestimmte Systemereignisse geknüpft. So schickt das System die Nachricht WM_LBUTTONDOWN, wenn eine Berührung des Touchscreens registriert wird, WM_MOUSEMOVE, wenn der Stylus über den Screen bewegt wird und WM_LBUTTONUP, sobald er den Touchscreen nicht mehr berührt. Für die Auswertung dieser Nachrichten kann zum Beispiel im Eventhandler für jedes Ereignis eine Funktion aufgerufen werden, die auf Styluseingaben entsprechend reagiert und die Position des Stylus auf dem Touchscreen als x- bzw. y-Koordinate übernimmt (siehe Listing 6.2.3.A).

Listing 6.2.3.A: Abfragen der Stylus-Eingabeereignisse innerhalb des Eventhandlers

```
LRESULT CALLBACK WinProc(HWND hwnd,UINT msg,WPARAM wparam,LPARAM
lparam) {
  switch (message){
    case WM_LBUTTONDOWN:
      StylusDown(LOWORD(lParam), HIWORD(lParam));
      break;
    case WM_MOUSEMOVE:
      StylusMove(LOWORD(lParam), HIWORD(lParam));
      break;
    case WM_LBUTTONUP:
      StylusUp(LOWORD(lParam), HIWORD(lParam));
      break;
  }
}
```

In den Funktionen *StylusDown()*, *StylusMove()* und *StylusUp()* ist definiert, wie auf das jeweilige Ereignis reagiert werden soll. Soll zum Beispiel bei Berühren des Bildschirms die Applikation beendet werden, so ist die Funktion *StylusDown()* wie folgt zu implementieren:

```
void StylusDown(int x, int y) {

  //selbst entwickelte Funktion, die das Programm beendet
  ApplicationShutdown();
}
```

Die Funktion *ApplicationShutdown()* soll dabei alle zum Beenden des Programms nötigen Aufrufe enthalten.

Kapitel 7: Sound

7.1. Klangintegration mittels DirectSound

Moderne Softwareapplikationen unterstützen bestimmte Programmereignisse oft akustisch, dass heisst, durch einzelne Geräusche oder auch durch Musik. Diese Klänge können in zweierlei Form existieren, digitalisiert und synthetisiert (künstlich erzeugt). Digitalisierte Klänge sind zumeist Aufnahmen aus der Natur, die in digitaler Form in verschiedenen Formaten wie wav, voc, au, aiff usw. gespeichert werden. Synthetische Klänge werden durch Tongeneratoren auf der Soundkarte und durch diverse Algorithmen erzeugt.

Für Geräusche, Effekte oder Sprachausgaben eignen sich am besten digitale Aufnahmen, denn diese sind synthetisch schwer nachzubilden. Sollen aber Musikstücke wiedergegeben werden, empfielt sich ein anderes Format: MIDI. MIDI steht für Musical Instrument Digital Interface und ist ein Dateiformat, dass musikalische Kompositionen als eine Funktion der Zeit beschreibt. MIDI selbst enthält keinerlei Klänge, sondern nur die Steueranweisungen, wann welches Instrument welche Note in welcher Länge spielt. Dabei greift MIDI auf Klangsamples zurück, die entweder einem Standard wie General MIDI entsprechen und von der Soundhardware breitgestellt werden, oder die als digitale Klangdateien vorliegen und zusammengefasst in einer sogenannten Soundbank MIDI als Instrumentenset zugewiesen werden können.
DirectMusic ist als die eigentliche Soundkomponente in DirectX anzusehen. Über DirectMusic lässt sich jedes Musikdateiformat auslesen und abspielen, speziell für MIDI und ähnlich aufgebaute Dateiformate ist sie deshalb am geeignetsten. Das von Windows wohl am besten unterstützte Format WAV lässt sich allerdings mittels der DirectSound-Schnittstelle besser verwalten.

DirectSound ist näher an der Hardware orientiert und gestattet über die Soundpuffertechnik das einfache Importieren, Speichern, Auslesen und Manipulieren von WAV-Dateien. Soundpuffer (sound buffer) sind mit den Surfaces in DirectDraw zu vergleichen. Sie sind reservierte "Stellen" im Speicher des Systems oder der Soundkarte, in die importierte Sounds abgelegt werden. Klangmanipulationen betreffen immer die Daten innerhalb der Soundpuffer und können, unterstützt durch integrierte Prozessoren wie den DSP (Digital Signal Prozessor), über die Soundhardware beschleunigt werden.

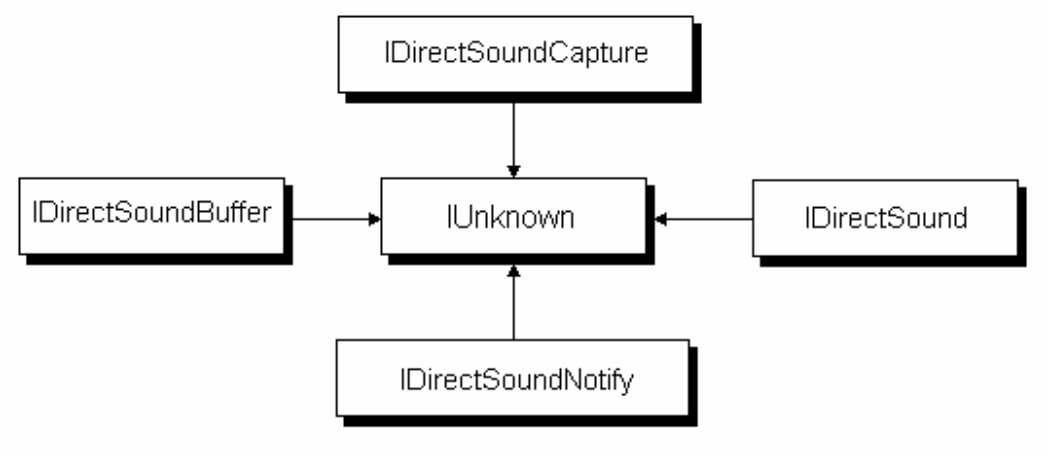
Als Beispiel für diesen Abschnitt soll binnen sechs Schritten die Funktionsweise von DirectSound anhand der Integration (Schaffen eines Soundpuffers, laden der Wav-Datei und kopieren der Daten in den Puffer) und Verwendung (Abspielen und Stoppen) einer WAV-Datei erklärt werden:

1. Schritt: Aufruf von DirectSoundCreate() zum Anlegen eines neuen DirectSound-Objekts

DirectSound ähnelt im Aufbau DirectDraw (siehe Abb. 7.1.A und auch Abb.5.1.2.A). Neben der Hauptschnittstelle IDirectSound existieren noch drei weitere:

IDirectSoundBuffer, die den Zugriff auf die Mixer-Hardware und das Soundbuffermanagement realisiert, IDirectSoundCapture, das für Aufnahmen und Audiostreaming gedacht ist, und IDirectSoundNotify, mit dem sich die Aufnahme- und Abspielfunktionen von DirectSound durch bestimmte Ereignisse steuern lassen.

Abbildung 7.1.A: Die Schnittstellen von DirectSound



Für jede Soundkarte im System ist eine IDirectSound-Schnittstelle zu definieren, über die sämtliche weiteren Eigenschaften und Optionen im Zusammenhang mit der Soundkarte gesteuert werden können. Über die Funktion *DirectSoundCreate()* wird eine neue IDirectSound-Schnittstelle angelegt. *DirectSoundCreate()* übernimmt dabei als Parameter:
- den GUID der Soundkarte, für das Standardgerät ist hier NULL anzugeben
- den Schnittstellenpointer zum Objekt
- NULL, da dieser Parameter für spätere COM-Erweiterungen reserviert ist

```
LPDIRECTSOUND  lpds;
// DirectSound-Schnittstellen-Pointer
DirectSoundCreate(NULL, &lpds, NULL);
```

ist der Aufruf für die Initialisierung der IDirectSound-Schnittstelle der Stanndard-Soundkarte im System.

2. Schritt: Setzen des Kooperationslevels für die Soundkarte

Über SetCooperativeLevel() wird im Anschluss festgelegt, wie die Soundkarte mit evtl. "konkurrierender" Hardware kooperieren soll. Dafür gibt es vier verschiedene Priorsierungsstufen:
- Normale Kooperation: In dieser Stufe kann innerhalb der zugehörigen Applikation Sound abgespielt werden, sofern die Anwendung den Fokus besitzt. Ebenso können aber auch andere parallel laufende Anwendungen über die gleiche Soundkarte Sounds abspielen. DirectSound legt in diesem Modus automatisch einen primären 22 KHz-, 8-Bit-Stereo-Soundpuffer an (siehe Schritt 3), unterbindet aber gleichzeitig den Schreibzugriff auf diesen Puffer. Wird über den Flag DSSCL_NORMAL gesetzt.

- Prioritätskooperation: Wie normale Kooperation, nur dass hier das Format des primären Soundpuffers verändert und der Soundpuffer selbst manuell beschrieben werden kann. Wird über den Flag DSSCL_PRIORITY gesetzt.
- Exklusive Kooperation: Wie Prioritätskooperation, nur mit dem Zusatz, dass die Applikation den exklusiven Zugriff auf die Soundkarte bekommt, sobald sie den Fokus hat. Keine andere Applikation kann während dieser Zeit auf die Soundkarte zugreifen. Wird über den Flag DSSCL_EXCLUSIVE gesetzt.
- "Write_Primary"-Kooperation: Dieser Modus überlässt dem Programmierer die totale Kontrolle über alle Eigenschaften und Optionen der Soundkarte. So ist zum Beispiel der primäre Soundpuffer eigenhändig anzulegen und zu verwalten. Wird über den Flag DSSCL_WRITEPRIMARY gesetzt.

Über

```
lpds->SetCooperativeLevel(hwnd,DSSCL_NORMAL);
```

wird zum Beispiel der normale Kooperationslevel für das Fenster *hwnd* gesetzt.

3. Schritt: Anlegen der primären und sekundären Soundpuffer:
Analog zu den Surfaces in DirectDraw werden geladene oder während der Laufzeit des Programms erzeugte Sounds in sekundären Puffern abgelegt und im primären Puffer zur Klangausgabe kombiniert. Wurde der normale Kooperationslevel als Modus festgelegt, wird der primäre Soundpuffer von DirectSound automatisch erzeugt.

Für das Anlegen weiterer Soundpuffer steht die Funktion *CreateSoundBuffer()* zur Verfügung. Als ersten Parameter übernimmt *CreateSoundBuffer()* einen Pointer zu einer Struktur, die beschreibende Daten bzgl. des Soundpuffers enthält. Vor Aufruf von *CreateSoundBuffer()* sind vier Eigenschaften dieser DSBUFFERDESC-Struktur zu definieren:

```
DSBUFFERDESC dsbd;

dsbd.dwSize        = sizeof(DSBUFFERDESC);
dsbd.dwFlags       = DSBCAPS_CTRLVOLUME |
                     DSBCAPS_STATIC |
                     DSBCAPS_LOCSOFTWARE;
dsbd.dwBufferBytes = 22050;
dsbd.lpwfxFormat   = &pcmwf;
```

dwSize speichert die Größe der DSBUFFERDESC-Struktur in Bytes.
dwFlags bestimmen die Kontroll-Attribute des Soundpuffers:
- DSBCAPS_VOLUME legt fest, dass die Lautstärke des Sounds in diesem Puffer verändert werden kann
- DSBCAPS_STATIC zeigt an, das die Daten im Soundpuffer statisch sind, der Puffer also nicht für Audiostreaming verwendet wird
- DSBCAPS_LOCSOFTWARE bestimmt, das der Soundpuffer im Speicher des Systems angelegt wird und nicht im Soundkartenspeicher, auch wenn entsprechende Hardware vorhanden ist

Alle weiteren Möglichkeiten sind in der Hilfe von MSVisual-C++ unter dem Suchbegriff DSBUFFERDESC zu finden.
dwBufferBytes bestimmt die Größe des Soundpuffers in Bytes nach der Formel:
BufferBytes = Zeit * Samplingrate
22050 Bytes sind demnach für einen 2-sekündigen 11025Hz-Klang zu reservieren.
lpwfxFormat zeigt auf eine weitere Struktur (WAVEFORMATEX), in der die Daten bzw. Eigenschaften des für den Puffer bestimmten Sounds gespeichert sind.
Die Definition der Eigenschaften der Struktur WAVEFORMATEX hat in diesem Beispiel folgendes Aussehen:

```
memset(&pcmwf, 0, sizeof(WAVEFORMATEX));

// Pulse-Code-Modulation-Format
pcmwf.wFormatTag      = WAVE_FORMAT_PCM;

// 1 Kanal = mono, 2 = Stereo
pcmwf.nChannels       = 1;

// 11025 Hz Sampling-Rate
pcmwf.nSamplesPerSec  = 11025;

// 1 Kanal * 1 Byte pro Sample = 1,
// 2 Kanäle (Stereo) und 16 bit  = 4 usw.
pcmwf.nBlockAlign     = 1;

pcmwf.nAvgBytesPerSec = pcmwf.nSamplesPerSec * pcmwf.nBlockAlign;

//Bits pro Sample (entspricht Farbtiefe bei Grafiken)
pcmwf.wBitsPerSample  = 8;

//immer 0
pcmwf.cbSize          = 0;
```

Der zweite Parameter, der *CreateSoundBuffer()* übergeben werden muss, ist ein Pointer auf den Soundpuffer selbst. Dieser ist zum Beispiel als

```
LPDIRECTSOUNDBUFFER lpdsbuffer;
```

festzulegen.

Als letztes ist der Funktion *CreateSoundBuffer()* NULL zu übergeben, da dieser Parameter für spätere Verwendung reserviert ist.

Durch

```
lpds->CreateSoundBuffer(&dsbd, &lpdsbuffer, NULL);
```

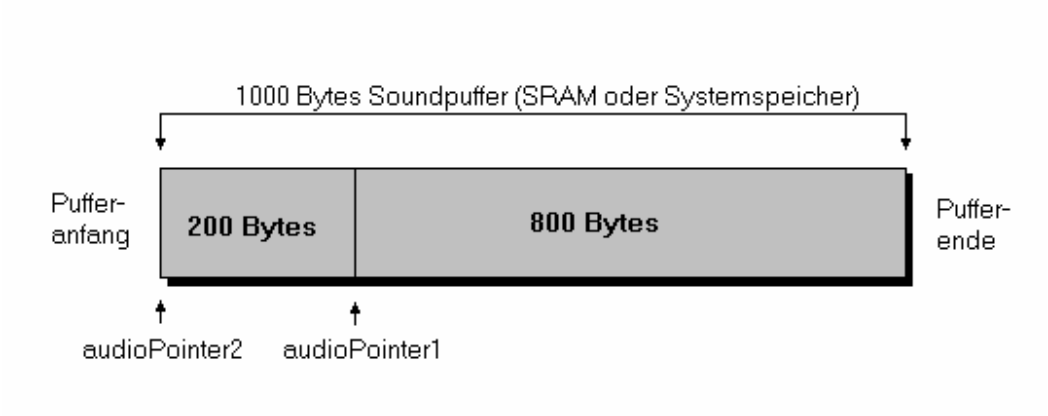
wird der Soundpuffer für dieses Beispiel angelegt.

4. Schritt: Sperren und Entsperren der Soundpuffer zum Laden oder Generieren eines Sounds

Um die Soundbuffer zu beschreiben, müssen diese ebenso wie die Surfaces in DirectDraw vor Fremdzugriff während des Schreibvorgangs geschützt werden. Die Funktion *Lock()* aktiviert diese Sperre, über *Unlock()* wird sie wieder aufgehoben.

Der Soundpuffer ist im Gegensatz zu den Grafik-Surfaces zirkulär aufgebaut, er besteht also aus zwei Teilen (siehe Abb. 7.1.B).

Abbildung 7.1.B: Der zirkuläre Aufbau eines DirectSound-Soundpuffers



In Abbildung 7.1.B. ist zuerkennen, dass der Soundpuffer in zwei Bereiche unterteilt wird, die über je einen Schreibcursor (audioPointer1 bzw. audioPointer2) angesprochen werden können. Eine solche Unterteilung wurde deshalb geschaffen, um Audioströme (Audiostreams) im Puffer aufnehmen und wiedergeben zu können. Soll zum Beispiel die Musik einer gesamten CD wiedergegeben werden, so ist es derzeit noch unmöglich, die ca. 650 MB Speicherkapazität in einem Soundpuffer unterzubringen. Demnach können nur einzelne Abschnitte (chunks) in den Puffer geschrieben werden. Ist nun wie im Beispiel der Abbildung 7.1.B ein 1000-Bytes-langer Sound in den Puffer zu übertragen und der *audioPointer1* schon durch frühere Operationen soweit an das Ende des Pufferspeichers gerückt, dass die 1000 Bytes nicht mehr in Bereich zwischen *audioPointer1* und Pufferende passen, so werden ab audioPointer2 die restlichen Bytes in den Puffer geschrieben. Die Verantwortung, dass sich der Puffer dabei nicht mehrfach überschreibt, bevor die darin gespeicherten Daten überhaupt erst einmal abgerufen werden können, liegt allerdings beim Programmierer.

Für statische Sounds genügt es jedoch, einmalig den gesamten Puffer zu sperren und zu beschreiben. Dafür muss der Funktion *Lock()* neben der Startposition des Schreibcursors, der Länge des zu sperrenden Bereiches und den vier Rückgabewerten von *Lock()* das Flag *DSBLOCK_ENTIREBUFFER* als siebtenter Parameter übergeben werden. Listing 7.1.A. beschreibt das Sperren und Entsperren von Soundpuffern für statische Sounds.

Listing 7.1.A: Beispiel zum Sperren und Entsperren von Soundpuffern unter DirectSound

```
unsigned char *audioPointer1,
              *audioPointer2;
```

```
int audioLength1,
    audioLength2;
//Start und Länge des zu sperrenden Pufferabschnitts
lpdsbuffer->Lock(0,1000,
                 (void **)&audioPointer1,  // Pointer zum ersten
                                           // Abschnitt des Sounds
                  &audioLength1,           // Länge des ersten
                                           // Abschnittes in Bytes
                  (void **)&audioPointer2, // Pointer zum zweiten
                                           // Abschnitt des Sounds
                   &audioLength2,          //Länge des zweiten
                                           // Abschnittes in Bytes
                   DSBLOCK_ENTIREBUFFER);

//DSBLOCK_ENTIREBUFFER für Sperren des gesamten Puffers;
//DSBLOCK_FROMWRITECURSOR zum Sperren ab Schreibcursorposition
//(1.Parameter wird dabeiignoriert)

//Schreibfreigabe für den Klangpuffer zum Beispiel für
//Aufruf(e) zum Kopieren eines generierten oder geladenen Sounds
//in den Puffer,z.B. DSound_Load_Wav() ->
//(siehe Beispielapplikation sound.cpp auf CD)

lpdsbuffer->Unlock(audioPointer1, audioLength1,
                   audioPointer2, audioLength2);
```

5. Schritt: Abspielen und Stoppen eines Sounds
Um den Sound aus einem bestimmten Soundpuffer abzuspielen, stellt DirectSound die Funktion *Play()* bereit. *Play()* übernimmt als Parameter zweimal 0, da diese Parameter reserviert sind, und einen dritten Wert, der bestimmt, ob der betreffende Sound nur einmal (*NULL*) oder dauerhaft (*DSBPLAY_LOOPING*) abgespielt werden soll.

```
lpdsbuffer->Play(0, 0, DSBPLAY_LOOPING);
```

Über die Funktion *Stop()* wird der derzeit "spielende" Sound angehalten und der Lesecursor des Soundpuffers entweder auf den Anfang zurückgesetzt, sofern der Puffer der primäre Soundpuffer der Soundkarte ist. Bei einem sekundären Soundpuffer verbleibt der Lesecursor an seiner Position und ein erneuter Aufruf von *Play()* setzt das Abspielen des Sounds an der Stelle fort, an der es unterbrochen wurde.

```
lpdsbuffer->Stop();
```

6. Schritt: Freigeben der Soundressourcen und des DirectSound-Objektes
Werden die Soundkartenressourcen nicht mehr gebraucht, empfielt es sich, diese wieder freizugeben. Dies geschieht sowohl für den Soundbuffer als auch für das DirectSound-Objekt über die Funktion *Release()*:

```
lpdsbuffer->Release();
lpds->Release();
```

Ein Beispiel für die Verwendung der DirectSound-Komponenten unter Windows (Sound.cpp) und die dazugehörige ausführbare Datei (Sound.exe) befinden sich auf der beigelegten CD im Verzeichnis \Kapitel7\. Dort wurde die Funktion *DSound_Load_WAV()* implementiert, die das Laden einer WAV-Datei enthält und sowohl das Anlegen eines Soundpuffers als auch das Sperren und Entsperren desselben beinhaltet. Da alle DirectX-relevanten Teile dieser Funktion bereits in diesem Abschnitt Beachtung fanden, wurde auf eine differenziertere Erklärung verzichtet. Ebenfalls verzichtet wurde auf die Anwendungsbeschreibung von DirectMusic, da sie aufgrund ihrer Komplexität den Rahmen dieser Arbeit sprengen würde.

7.2. Integration von Sounds in WindowsCE-Applikationen

Die Wiedergabe von digitalisierten Klängen oder Musik wird auch von vielen Pocket-PCs unterstützt, muss jedoch auf andere Funktionsbibliotheken als die GameAPI zurückgreifen, da diese für Soundintegration keinerlei Funktionen bietet. Steht keine Erweiterungsbibliothek eines Drittanbieters zur Verfügung, empfiehlt sich der Zugriff über die GDI/MCI-Schnittstellen von WindowsCE. Dieser Zugriff ist zwar vom Aufwand her einfacher umzusetzen, da keine Soundpufferstruktur (siehe Abschnitt 7.1) zu initialisieren ist, aber dafür ist der Funktionsumfang sehr eingeschränkt. Es werden zum Beispiel keine Klangmanipulationen wie Frequenz- oder Lautstärkeveränderung und kein Soundmixing (Kombinieren von Einzelklängen zu einem Gesamtklangbild) unterstützt. Letzteres bedeutet, dass keine Routinen zur Verfügung stehen, um mehr als einen einzelnen Sound zu einer bestimmten Zeit abzuspielen. Ein weiterer Nachteil besteht darin, dass die GDI/MCI-Funktionen unter WindowsCE als einziges Dateiformat das unkomprimierte PCM-Wave-Format unterstützen. Für alle weiteren Klangformate müssen unter WindowsCE entsprechende Routinen erst entwickelt werden.

Das Beispiel für diesen Abschnitt soll in vier Schritten zeigen, wie WAV-Sounds in WindowsCE-Applikationen integriert werden können. Dazu gehört das Laden einer WAV-Datei, das Abspielen und Stoppen von Sounds und die Freigabe aller dafür verwendeten Ressourcen.

1. Schritt: Das Laden einer WAV-Datei:
Die hier demonstrierte Quelltextversion zum Laden einer WAV-Datei ist deutlich einfacher angelegt als sein Pedant aus dem Abschnitt 7.1. Es wird davon ausgegangen, dass die zu ladenden Daten in Form einer unkomprimierten PCM-Wav-Datei vorliegen.

```
LPTSTR lpFile;                  //Pointer auf die zu ladende
                                //WAV-Datei
LPCTSTR lpWave = NULL;          //Pointer auf den Speicherplatz,
                                //in den WAV-Datei kopiert wird
HANDLE hFile;                   //Zugriffsvariable für die
                                //geöffnete (WAV)-Datei
DWORD dwSize = 0;               //Größe des Sounds in Bytes
DWORD dwBytesRead = 0;          //Bereits eingelesene Bytes
BOOL bReturn;                   //Erfolgsstatus bzgl. des Ladens
                                //der WAV-Datei:
                                //TRUE = erfolgreich
```

```
//Öffnen der WAV-Datei
hFile = CreateFile(lpFile,       //Pointer zu Null-terminiertem
                                 //String, der die zu ladende
                                 //Datei inklusive Pfad,
                                 //eine Ressource o.ä. enthält
                GENERIC_READ,    //Lesezugriff auf die Datei
                                 //GENERIC_WRITE für
                                 //Schreibzugriff
                FILE_SHARE_READ, //andere Prozesse dürfen nur im
                                 //Lesemodus auf die geöffnete
                                 //Datei zugreifen/
                                 //0 für kein Zugriff,
                                 //FILE_SHARE_WRITE für Schreiben
                NULL,            //immer NULL
                OPEN_EXISTING,   //Spezifiziert die Aktion, wie
                                 //mit existierenden oder
                                 //nichtexistenten Dateien
                                 //umgegangen werden soll.
                                 //Hier: Öffnen einer Datei,
                                 //Funktion schägt fehl, wenn
                                 //Datei nicht existiert
          FILE_ATTRIBUTE_NORMAL, //Spezifiziert die Attribute für
                                 //die zu öffnende Datei.
                                 //Hier:Keine speziellen Attribute
                NULL);           //Template-Datei.
                                 //Hier: NULL für keine

//Bestimmen der Länge des Sounds
dwSize  = GetFileSize(hFile, NULL);

//Reservieren von Speicher für die WAV-Datei
lpWave = (LPCTSTR) malloc (dwSize);

//Einlesen der Datei
bReturn = ReadFile(hFile,          //Dateiadresse(="Filehandle")
                   (void *) lpWave, //Pointer auf Zielstruktur
                   dwSize,          //Anzahl einzulesender Bytes
                   &dwBytesRead,    //Anzahl tatsächlich
                                    //eingelesener Bytes
                   NULL);           //Nicht unterstützt,
                                    //darum NULL

//Bei Fehlschlagen des Lesens Freigeben der Ressourcen
if ((bReturn == FALSE) || (dwBytesRead != dwSize)){
  CloseHandle(hFile);
  free((void *)lpWave);
  dwBytesRead = 0;
}

//Schliessen des "Filehandles"
CloseHandle(hFile);
```

2. Schritt: Abspielen eines Sounds:
Über die Funktion *sndPlaySound()* lassen sich Wave-Daten wiedergeben. Sie übernimmt als Parameter zum einen einen Pointer auf die Stelle im Speicher, wo die Wav-Daten abgelegt wurden, zum anderen eine Anzahl von Flags, die den Abspielvorgang steuern. So wird zum Beispiel durch SND_MEMORY dem Programm mitgeteilt, dass die Wavedaten in einem Bereich des Hauptspeichers abgelegt wurden, SND_LOOP bestimmt, dass ein Sound zyklisch wiederholt werden soll und SND_ASYNC legt das asyncrone Abspielen dieses Sounds fest, was Voraussetzung für SND_LOOP ist.

```
BOOL bRepeat = FALSE;
//Looping-Status: TRUE  = Sound einmal abspielen,
                  FALSE = Sound unendlich oft abspielen

if (lpWave != NULL){
  if (bRepeat) {
    sndPlaySound(lpWave, SND_MEMORY | SND_ASYNC | SND_LOOP);
  } else {
    sndPlaySound(lpWave, SND_MEMORY | SND_ASYNC);
  }
}
```

Als Alternative zu *sndPlaySound()* bietet sich die Funktion *PlaySound()* an, die vor allem dann zum Einsatz kommt, wenn andere Soundformate als WAV abgespielt werden sollen.

3. Schritt: Stoppen eines Sounds:
Wird der Funktion sndPLaySound() NULL als erster Parameter übergeben und kein Flag gesetzt, so wird die momentan laufende Wiedergabe gestoppt.

```
sndPlaySound(NULL, 0);
```

4. Schritt: Freigeben der Ressourcen für die Sounds:
Der Speicherbereich, der von den geladen Wav-Daten belegt wird, ist durch

```
free((void *)lpWave);
```

wieder freizugeben.

Ein Beispiel für die Verwendung der DirectSound-Komponenten unter Windows (SoundCE.cpp) und die dazugehörige ausführbare Datei (SoundCE.exe) befinden sich auf der beigelegten CD im Verzeichnis \Kapitel7\. Dort wurde die Funktion *DSound_Load_WAV()* implementiert, die das Laden einer WAV-Datei enthält und sowohl das Anlegen eines Soundpuffers als auch das Sperren und Entsperren desselben beinhaltet. Da alle DirectX-relevanten Teile dieser Funktion bereits in diesem Abschnitt Beachtung fanden, wurde auf eine differenziertere Erklärung verzichtet. Ebenfalls verzichtet wurde auf die Anwendungsbeschreibung von DirectMusic, da sie aufgrund ihrer Komplexität den Rahmen dieser Arbeit sprengen würde.

8. Untersuchungen zur Geschwindigkeit von DirectX- und GameAPI-basierten Funktionen

8.1. Vergleich der Geräteleistung unabhängig von DirectX und der GameAPI

Während sich die vorherigen Kapitel mit dem Vergleich der Programmiertechniken auf beiden Plattformen beschäftigt haben, soll dieses 8.Kapitel vor allen Dingen dazu dienen, Einblicke in die Leistungsfähigkeit der Hardware zu gewinnen. Zwei verschiedene Arten von Tests sollen verdeutlichen, inwiefern die Hardware eines Pocket-PCs mit einem 206 MHz-Prozessor der eines Desktop-PCs mit einer ähnlich hohen Prozessorleistung von 233 MHz ebenbürtig ist bzw. in welchen Bereichen der Pocket-PC über- oder unterliegt. Die erste Art von Tests misst die Geschwindigkeit, mit der die beiden Geräte ein DirectX- bzw. GameAPI-unabhängiges Programm ausführen, bei der zweiten Art werden die Vor- und Nachteile der speziellen Strukturen unter DirectX oder der GameAPI untersucht.

In der Beispielapplikation für Test 1 wird ein einfaches Windowsfenster geöffnet. Variante a ist die Version für den Desktop-PC, b für den Pocket-PC, das Programm ist allerdings für beide Plattformen nahezu identisch.

Test 1: Initialisieren eines einfachen Windows-Standardfensters

Tabelle 8.1.A: Zeit (in Millisekunden) für das Initialisieren eines Fensters unter Windows
a) für Desktop-PCs*
b) für Pocket-PCs**

(MWT = Arithmetischer Mittelwert der Ergebnisse, MQA = Mittlere quadrat. Abweichung)

Durchlauf	Desktop-PC	Pocket-PC
1	9	157
2	4	173
3	3	153
4	6	159
5	11	154

MWT	6,6	159,2
MQA	1,3	3,2

* (Quelltext der Anwendung und zugehörige .exe-Datei siehe CD im Verzeichnis: Testprogramme\Kapitel4\1stWindow\ )

** (Quelltext der Anwendung und zugehörige .exe-Datei siehe CD im Verzeichnis: Testprogramme\Kapitel4\1stWindowCE\ )

Die Werte aus Test 1 zeigen, dass der Aufbau eines Fensters auf dem Desktop-PC ungleich schneller geht als auf einem Pocket-PC. Gleichartige bzw. gleiche Funktionen werden auf einem Pocket-PC offenbar langsamer ausgeführt als auf einem

vergleichbaren Desktop-PC, was vermuten lässt, dass ein Pocket-PC trotz frequenzgleichen Prozessors nicht dieselbe Geschwindigkeit erreicht wie ein Desktop-PC.

8.2. Vergleich der Darstellungsgeschwindigkeit von generativen Grafiken

Eines der obersten Ziele von DirectX und der GameAPI war es, Routinen zu entwickeln, die durch Umgehung der vorhandenen Betriebssystem-internen Funktionen einen Direktzugriff auf die Hardware ermöglichen,womit deutliche Geschwindigkeitssteigerungen vor allem im Bereich der Grafikdarstellung erreicht werden.

Während die GameAPI Geschwindigkeitsvorteile vor allem durch die Grafikfunktionen erreicht, die die direkte Manipulation des Videospeichers ermöglichen, baut DirectX zudem noch auf die Unterstützung besonderer Hardware, die speziell entwickelt wurde, um bestimmte Operationen deutlich schneller durchzuführen, als es ein Softwarealgorithmus könnte.

Die Untersuchungen dieses Abschnittes (8.2) beschäftigen sich mit der Frage, wie schnell DirectX generative Grafiken darstellen kann, wenn diese entweder direkt in den Videospeicher der Grafikkarte kopiert oder aus einem sekundären Surface heraus in das primäre Surface (Videospeicher) verschoben werden. Zudem soll die Geschwindigkeit untersucht werden, mit der die GameAPI ihrerseits generative Grafiken erzeugen und anzeigen kann.

Für den Test 2 wurde eine 16-Bit-Anwendung erstellt, die in jedem Schleifendurchlauf der Hauptprogrammfunktion (*App_Main()*) zufällig Position und Farbe einzelner Pixel generiert und diese auf dem Bildschirm anzeigt. Variante a (Desktop-PC) generiert die Pixel direkt im primären Surface, in Variante b (Desktop-PC) werden in einem sekundären Surface die Pixel generiert und anschließend von da aus in das primäre Surface verschoben (blit). Variante c (Pocket-PC) verwendet die GameAPI-Funktionen, um die Pixel direkt im primären Gerätekontext zu generieren.

Gemessen wurde die Dauer der Initialisierung des Anwendungsfensters und die Zeit, die benötigt wird, um 1000 Mal die Funktion *App_Main()* aufzurufen.

Test 2: 16-Bit-Pixelplotter (PP16)

Tabelle 8.2.A: Zeit (in ms) für den Aufbau des Fensters in PP16

a) für Desktop-PCs: Generieren der Pixel direkt im primären Grafiksurface*
b) für Desktop-PCs: Generieren der Pixel in sekundärem Grafiksurface und Blitten in Primäres Surface**
c) für Pocket-PCs: Generieren der Pixel im primären Gerätekontext***

(MWT = Arithmetischer Mittelwert der Ergebnisse, MQA = Mittlere quadrat. Abweichung)

Durchlauf	Desktop-PC a	Desktop-PC b	Pocket-PC
1	715	833	88
2	737	821	94
3	714	823	88
4	728	821	83
5	732	825	89

MWT	725,2	824,6	88,4
MQA	4,1	2,0	1,6

Tabelle 8.2.B: Zeit (in ms) für 1000 Durchläufe der Funktion App_Main() in PP16

a) für Desktop-PCs: Generieren der Pixel direkt im primären Grafiksurface*
b) für Desktop-PCs: Generieren der Pixel in sekundärem Grafiksurface und Blitten in Primäres Surface**
c) für Pocket-PCs: Generieren der Pixel im primären Gerätekontext***

(MWT = Arithmetischer Mittelwert der Ergebnisse, MQA = Mittlere quadrat. Abweichung)

Durchlauf	Desktop-PC a	Desktop-PC b	Pocket-PC
1	649	2369	1247
2	652	2366	1301
3	648	2359	1248
4	662	2362	1228
5	649	2358	1358

MWT	652	2362,8	1276,4
MQA	2,3	1,9	21,2

* (Quelltext der Anwendung und zugehörige .exe-Datei siehe CD im Verzeichnis:
Testprogramme\Kapitel5\Pixelplotter16 - Zeichnen in primäres Surface\ )

** (Quelltext der Anwendung und zugehörige .exe-Datei siehe CD im Verzeichnis:
Testprogramme\Kapitel5\Pixelplotter16 - Zeichnen in sekundäres Surface und Blit\ )

*** (Quelltext der Anwendung und zugehörige .exe-Datei siehe CD im Verzeichnis:
Testprogramme\Kapitel5\Pixelplotter16CE - Zeichnen in Videospeicher\)

Test 2 zeigt, das in Variante b sowohl die Initialisierung des Fensters und der Surfacestrukturen von DirectX als auch die Dauer für 1000 Schleifendurchläufe erheblich mehr Zeit in Anspruch nimmt als in Variante a. Dazu ist zu sagen, dass in b noch zusätzlich ein sekundäres Surface angelegt wird und die Pixel, anstatt direkt angezeigt zu werden, noch in jedem Schleifendurchlauf das primäre Surface kopiert werden müssen. Daraus ist zu schlussfolgern, das generative Grafiken direkt im primären Surface erstellt werden sollten, da das Umkopieren von einem Surface in ein anderes ohne weiteren Nutzen nur noch erhöht.

Auf dem Pocket-PC geschieht die Initialisierung deutlich schneller als bei der DirectX-Variante. Aber obwohl die GameAPI auch den direkten Zugriff auf den Speicher der Anzeige (hier: primärer Gerätekontext) ermöglicht, die Programme sich also vom Typus (Generieren der Pixel direkt im Videospeicher) her gleichen, wird in Variante c ca. doppelt soviel Zeit für 1000 Schleifendurchläufe benötigt wie in der DirectX-Variante a.

8.3. Vergleich der Darstellungsgeschwindigkeit von Bitmap-Dateien

Oftmals ist Software so konzipiert, dass das gesamte Layout aus Einzelgrafiken besteht, die in externen Editoren erstellt werden. Dieser Abschnitt (8.3) beschäftigt sich nun damit, wie schnell DirectX solche externen Grafiken laden und darstellen kann, wenn diese entweder direkt in das primäre Surface kopiert oder aus einem sekundären Surface heraus in das primäre Surface verschoben werden. Zudem soll untersucht werden, mit welcher Geschwindigkeit die GameAPI Bitmap-Grafiken in den Systemspeicher lädt und anzeigt.

Als Beispiel für Test 3 steht eine 16-Bit-Anwendung zur Verfügung, die zuerst eine BMP-Datei lädt und diese dann in jedem Schleifendurchlauf der Hauptprogrammfunktion (*App_Main()*) auf dem Bildschirm anzeigt. Variante a (Desktop-PC) lädt eine 24-Bit-Bitmapdatei in den Systemspeicher, konvertiert diese in das 16-Bit-Format und kopiert in jedem Schleifendurchlauf die Bitmapdaten in das primäre Surface. In Variante b (Desktop-PC) wird ebenfalls eine 24-Bit-Bitmapdatei geladen und in das 16-Bitformat konvertiert, jedoch werden die Daten hier in einem sekundären Surface abgelegt und innerhalb der Funktion *App_Main()* vom sekundären Surface in das primäre verschoben (blit). Variante c für den Pocket-PC lädt eine 8-Bit-BMP-Datei in einen eigenen Gerätekontext, konvertiert diesen in das 16-Bit-Format und blittet das Bild in den primären Gerätekontext.

Gemessen wurde die Dauer der Initialisierung des Anwendungsfensters inkl. der Ladezeit für die Bitmap-Datei, die Zeit, die benötigt wird, um 1000 Mal die Funktion

App_Main() aufzurufen und wie lange es dauert, bis die Programmressourcen wieder freigegeben wurden.

Test 3: 16-Bit-Anwendung zum Laden und Darstellen von Bitmaps (LB16)

Tabelle 8.3.A: Zeit (in ms) für den Aufbau des Fensters in LB16

a) für Desktop-PCs: Laden einer Bitmapgrafik direkt in den Systemspeicher und Blitten in das Primäre Surface*
b) für Desktop-PCs: Laden einer Bitmapgrafik in ein sekundäres Surface und Blitten in das Primäre Surface**
c) für Pocket-PCs: Laden einer Bitmapgrafik aus einem sekundären in den primären Gerätekontext***

(MWT = Arithmetischer Mittelwert der Ergebnisse, MQA = Mittlere quadrat. Abweichung)

Durchlauf	Desktop-PC a	Desktop-PC b	Pocket-PC
1	743	751	125
2	741	752	117
3	732	769	111
4	745	752	112
5	750	753	117

MWT	742,2	755,4	116,4
MQA	2,6	3,1	2,2

Tabelle 8.3.B: Zeit (in ms) für 1000 Durchläufe der Funktion App_Main() in LB16

a) für Desktop-PCs: Laden einer Bitmapgrafik direkt in den Systemspeicher und Blitten in das Primäre Surface*
b) für Desktop-PCs: Laden einer Bitmapgrafik in ein sekundäres Surface und Blitten in das Primäre Surface**
c) für Pocket-PCs: Laden einer Bitmapgrafik aus einem sekundären in den primären Gerätekontext***

(MWT = Arithmetischer Mittelwert der Ergebnisse, MQA = Mittlere quadrat. Abweichung)

Durchlauf	Desktop-PC a	Desktop-PC b	Pocket-PC
1	10036	443	41901
2	10026	443	41989
3	10049	440	41931
4	10040	439	42002
5	10084	443	42005

MWT	10047	441,6	41965,6
MQA	8,9	0,8	18,8

Tabelle 8.3.C: Zeit (in ms) für die Freigabe der Ressourcen in LB16

a) für Desktop-PCs: Laden einer Bitmapgrafik direkt in den Systemspeicher und Blitten in das Primäre Surface*
b) für Desktop-PCs: Laden einer Bitmapgrafik in ein sekundäres Surface und Blitten in das Primäre Surface**
c) für Pocket-PCs: Laden einer Bitmapgrafik aus einem sekundären in den primären Gerätekontext***

(MWT = Arithmetischer Mittelwert der Ergebnisse, MQA = Mittlere quadrat. Abweichung)

Durchlauf	Desktop-PC a	Desktop-PC b	Pocket-PC
1	55	0	0
2	58	1	0
3	20	0	0
4	54	0	0
5	29	0	0

MWT	43,2	0,2	0
MQA	7,0	0,2	0,0

* (Quelltext der Anwendung und zugehörige .exe-Datei siehe CD im Verzeichnis: Testprogramme\Kapitel5\LoadBitmap16 (Laden des Bitmap aus Datei in Primäres Surface)\ )

** (Quelltext der Anwendung und zugehörige .exe-Datei siehe CD im Verzeichnis: Testprogramme\Kapitel5\LoadBitmap16 (Laden des Bitmap aus sek. Surface in Primäres Surface)\ )

*** (Quelltext der Anwendung und zugehörige .exe-Datei siehe CD im Verzeichnis: Testprogramme\Kapitel5\LoadBitmap16CE (Laden des Bitmap aus Datei in den prim. Gerätekontext)\ )

An dieser Stelle zeigt sich sehr deutlich der Geschwindigkeitsvorteil, der durch DirectX erreicht wird, wenn anstatt generativer Grafiken extern vordefinierte Bilddateien importiert werden. Die Ladezeit für die BMP-Datei ist in Variante b nur geringfügig höher als in Variante a, da ja in b das sekundäre Surface noch zusätzlich initialsiert und "gefüllt" werden muss, allerdings wird das Anzeigen der Grafik durch die Surface-Architektur in Variante b auf ca. 1/25 der Blit-Zeit von a reduziert.

In Variante c ist zu erkennen, dass zwar die Ladezeit für die BMP-Datei deutlich geringer ist, (Ladezeit in Variante c ist mit 3 zu multiplizieren, um einen Vergleich zu erhalten, wie schnell eine 24-Bit-Datei anstatt der 8-Bit-Grafik zum Laden gebraucht hätte), allerdings ist die Zeitdauer für 1000 Schleifendurchläufe, in denen das Bild jeweils einmal aus einem sekundären Gerätekontext in den primären Kontext kopiert wurde, etwa 4 mal größer als bei der DirectX-Variante a und ca. 100 mal größer als bei b.

Weiterhin fällt auf, dass nur in Variante a eine relativ lange Zeit (ca. 60ms) für die Ressourcenfreigabe benötigt wird. In b und c hingegen ist diese Zeit nur in der Größenordnung von Microsekunden (μs) erfassbar, vermutlich weil in diesen Fällen die Speicherregionen nicht explizit gelöscht, sondern nur für andere Prozesse zum Beschreiben freigegeben wurden.

8.4. Vergleich der Initialisierungs- und Freigabezeiten für Wav-Dateien

Ebenso wie bei Grafiken werden auch im Bereich Sound die Klänge, Geräusche und die Musik für eine bestimmte Applikation meistens nicht innerhalb des Programms generiert, sondern liegen in speziellen Formaten als Datei vor. In diesem Abschnitt (8.4) wird die Zeit untersucht, die benötigt wird, um eine WAV-Datei zu laden sowie die Dauer für die Freigabe der Programmressourcen.

In Test 3 wird eine 16-Bit-Anwendung verwendet, die zuerst eine WAV-Datei (11.025 Hz; 8 Bit; Mono) lädt und diese abspielt. Bei Beenden des Programms wird die Wiedergabe des Sounds gestoppt. In Variante a (Desktop-PC) und c (Pocket-PC) wird die Datei in den Systemspeicher geladen und von dort aus abgespielt. Variante b (Desktop-PC) legt die Datei in einem Soundpuffer ab und startet aus diesem heraus die Wiedergabe.

Gemessen wurde die Dauer der Initialisierung des Anwendungsfensters inkl. der Ladezeit für die WAV-Datei und wie lange es dauert, bis die Programmressourcen wieder freigegeben wurden.

Test 4: 16-Bit-Anwendung zum Laden und Darstellen von Wav-Dateien (LW16)

Tabelle 8.4.A: Zeit (in ms) für das Laden einer Wav-Datei LW16

a) für Desktop-PCs: Laden der Wav-Datei in Systemspeicher*
b) für Desktop-PCs: Laden der Wav-Datei in Soundpuffer**
c) für Pocket-PCs: Laden der Wav-Datei in Systemspeicher***

(MWT = Arithmetischer Mittelwert der Ergebnisse, MQA = Mittlere quadrat. Abweichung)

Durchlauf	Desktop-PC a	Desktop-PC b	Pocket-PC
1	4	4	5
2	4	4	5
3	4	4	5
4	4	4	6
5	4	4	5
MWT	4	4	5,2
MQA	0,0	0,0	0,2

Tabelle 8.4.B: Zeit (in ms) für den Aufbau des Fensters in LW16
a) für Desktop-PCs: Laden der Wav-Datei in Systemspeicher*
b) für Desktop-PCs: Laden der Wav-Datei in Soundpuffer**
c) für Pocket-PCs: Laden der Wav-Datei in Systemspeicher***

(MWT = Arithmetischer Mittelwert der Ergebnisse, MQA = Mittlere quadrat. Abweichung)

Durchlauf	Desktop-PC a	Desktop-PC b	Pocket-PC
1	736	880	270
2	740	874	262
3	743	870	248
4	736	880	237
5	40	874	247

MWT	599	875,6	252,8
MQA	125,0	1,7	5,2

Tabelle 8.4.C: Zeit (in ms) für das Freigeben der Ressourcen in LW16
a) für Desktop-PCs: Laden der Wav-Datei in Systemspeicher*
b) für Desktop-PCs: Laden der Wav-Datei in Soundpuffer**
c) für Pocket-PCs: Laden der Wav-Datei in Systemspeicher***

(MWT = Arithmetischer Mittelwert der Ergebnisse, MQA = Mittlere quadrat. Abweichung)

Durchlauf	Desktop-PC a	Desktop-PC b	Pocket-PC
1	20	81	3
2	55	89	3
3	57	68	4
4	38	47	4
5	59	59	3

MWT	45,8	68,8	3,4
MQA	6,7	6,7	0,2

* (Quelltext der Anwendung und zugehörige .exe-Datei siehe CD im Verzeichnis: Testprogramme\Kapitel7\Sound - Laden der Wav-Datei in Systemspeicher\ )

** (Quelltext der Anwendung und zugehörige .exe-Datei siehe CD im Verzeichnis: Testprogramme\Kapitel7\Sound - Laden der Wav-Datei in Soundpuffer\ )

*** (Quelltext der Anwendung und zugehörige .exe-Datei siehe CD im Verzeichnis: Testprogramme\Kapitel7\SoundCE - Laden der Wav-Datei in Systemspeicher\ )

Dem Test 4 ist zu entnehmen, dass in Variante b mehr Zeit gebraucht wird, um das Programm zu initialisieren und zudem noch die WAV-Datei zu laden, als in a, da in b ein zusätzlicher Soundpuffer angelegt wird, der die WAV-Datei enthalten soll. Die eigentliche Ladezeit für die WAV-Datei ist bei beiden erwartungsgemäss identisch.

Anders bei der Pocket-PC-Variante c. Dort wird die WAV-Datei nicht in einen vorreservierten, DirectX-verwalteten Pufferspeicher abgelegt, sondern im Systemspeicher untergebracht. Das Laden der Wav-Datei dauert demnach auch etwas länger. Allerdings beträgt die Gesamtzeit für die Programminitialisierung inkl. des Dateiladens nur ca. 1/3 der Zeit der Varianten a und b. Auch beim Freigeben der Ressourcen ist c deutlich schneller, nur 3-4 ms anstatt der ca. 50 bzw. 70ms der beiden anderen Varianten.

Wie Test 4 zeigt, sollte in Desktop-PC-Programmen, die nur eine einzige Wav-Datei zu einem bestimmten Zeitpunkt abspielen, auf DirectX verzichtet und die Applikation analog der Variante c aufgebaut werden.
DirectSound lohnt nur dann, wenn Sounds aus mehreren sekundären Soundpuffern in einem primären Puffer kombiniert und erst dann ausgegeben werden sollen (Soundmixing).
Für die Wiedergabe von einzelnen Sounds wie z.B. bei Audiostreaming ist DirectSound nicht geeignet. Eine gute Alternative bietet hier DirectShow, dass zwar nicht Bestandteil dieser Arbeit ist, aber speziell für Multimediadatenströme entwickelt wurde.

Kapitel 9: Zusammenfassung

Diese Arbeit bietet einen Ausblick über den Nutzen, die Möglichkeiten und den Funktionsumfang von DirectX. Dazu wurde anhand einer Auswahl der Unterkomponenten von DirectX demonstriert, wie 2D-Grafiken in eine Applikation eingebunden und in derselben dargestellt werden, wie über DirectX Eingabeereignisse verschiedenster Geräte abgefragt und ausgewertet werden können, und welche Schritte nötig sind, um Sounddateien in ein Programm einzubinden und wiederzugeben. Des Weiteren wurde in der Arbeit aufgezeigt, auf welche Weise die in Windows integrierten Standardfunktionen umgangen werden können, um direkten Zugriff auf einzelne Hardwarekomponenten zu erhalten, um so Performancevorteile für leistungsintensive Anwendungen zu erzielen. DirectX enthält über 1200 Funktionen, die fast alle Hardware-relevanten Bereiche einer Applikation abdecken. Ebenso ist es DirectX möglich, auf jedwede Hardware zuzugreifen, für die ein entsprechender Treiber bereitsteht.

Gleichzeitig wurde die GameAPI, das Gegenstück zu DirectX auf dem Pocket-PC, vorgestellt. Dabei wird durch einen direkten Vergleich sowohl der Leistungsstand aktueller Pocket-PCs, als auch die Möglichkeiten, Vor- und Nachteile der GameAPI verdeutlicht. Analog zur Erläuterung von DirectX wurden auch hier Beispiele aufgezeigt, um 2D-Grafiken und Sound in eine Anwendung zu integrieren und Nutzereingaben zu verwalten.

Im ersten Kapitel wurde untersucht, inwieweit Desktop- und Pocket-PCs eine gemeinsame Grundlage bilden auf der man sie vergleichen kann. Dazu wurde Hardware und das Betriebssystem des Pocket-PCs mit dem eines Desktop-PCs verglichen und es zeigte sich, dass dieser aufgrund seiner Prozessorleistung mit einem Gerät der Pentium-II-Klasse vergleichbar ist. Jedoch ist die Speicherarchitektur so angelegt, dass der auf Desktop-PCs existierende Arbeitsspeicher und die Festplatte beim Pocket-PC in nur einem Speicher zusammengefasst wurden, die gesamte Speicherkapazität aber deutlich unter der eines vergleichbaren Desktop-PCs liegt. Die Nutzereingabe geschieht beim Pocket-PC über ins Gehäuse integrierte Tasten, zusätzlich fungiert das Display als Touchscreen, was die Funktionalität einer Computermaus auf einen Pocket-PC abbildet.
Nachdem festgestellt wurde, dass für viele Funktionen auf den Desktop-PCs Äquivalente für Pocket-PCs existieren, wurden im zweiten Kapitel die Grundlagen von Microsoft Windows diskutiert, dass in einer plattformspezifischen Ausführung auf beiden Geräten als Betriebssystem installiert ist.
Näher beleuchtet wurde Funktionionsumfang und Leistungsvermögen von DirectX und der GameAPI im dritten Kapitel. Anhand des Component Object Models wurde gezeigt, nach welchem Prinzip DirectX funktioniert.
Als erste Anwendung wurde im vierten Kapitel eine Applikation entwickelt, die ein einfaches Windows-Fenster öffnet. Anhand weiterer Beispielapplikationen zur 2D-Grafikdarstellung in Kapitel 5, zur Nutzereingabeerfassung und -verwaltung in Kapitel 6 und zur Soundintegration in Kapitel 7 wurden wesentliche Gemeinsamkeiten und Unterschiede bei der Programmierung für beide Plattformen deutlich gemacht. Nach Kapitel 7 stehen nun Routinen zur Verfügung, die in einer Bibliothek zusammengefasst den Grundstock für jedes DirectX- oder GameAPI-basierende Programm bilden.

Schließlich wurde in Kapitel 8 mittels einiger Geschwindigkeitstests untersucht, inwiefern die in Kapitel 4 bis 7 entwickelten Programme auf beiden Plattformen vom Laufzeitverhalten her identisch sind und an welcher Stelle sich der Einsatz von DirectX oder der GameAPI nur bedingt oder auch garnicht lohnt. Dabei stellte sich heraus, das auf dem Pocket-PC gleiche Funktionen etwas langsamer ausführt werden als auf einem von der Prozessorleistung her ebenbürtigen Desktop-PC. Da für Pocket-PCs bestimmte Dienste, Optionen und Aufgaben nicht vorgesehen sind, geraten sie in Bereichen wie Hardwarebeschleunigung, 3D-Darstellung und Netzwerkunterstützung deutlich ins Hintertreffen. Allerdings sollte der Zweck von Pocket-PCs nicht aus den Augen verloren werden, denn sie wurden vor allem entwickelt, um die Grundfunktionalitäten von Windows und die zugehörigen Standardprogramme wie z.B. Microsoft Word auf einem mobilen, tragbaren Gerät zur Verfügung zu stellen.

Zukünftig ist zu erwarten, dass die Hardwarekomponenten für Desktop-PCs noch weiter verbessert werden. DirectX existiert momentan in der Version 9.0, jedoch ist von Seiten der Softwarefirma Microsoft binnen der nächsten Jahre keine Weiterentwicklung geplant.

Der Pocket-PC wird sich, so die Hoffnung der Hersteller, weiter auf dem Markt etablieren. Bestrebungen, die die Kommunikation zwischen den Geräten und deren Anschluss ans Internet vorsehen, sind bereits im Gange und scheinen neben weiterhin fallenden Gerätepreisen die Herstellerhoffnungen zu rechtfertigen, dass Pocket-PCs bald kleinere Spielekonsolen wie den Gameboy verdrängen und in vielen Bereichen Laptops oder gar Handys ersetzen werden.

Die GameAPI wurde ursprünglich von Microsoft als eine Interimslösung entwickelt, um ein DirectX-Äquivalent auch für den Pocket-PC anbieten zu können. Mittlerweile werden aber immer mehr Komponenten von DirectX auf den Pocket-PC portiert, so dass zu erwarten steht, dass DirectX bald in annähernd gleichem Umfang für die kleineren Geräte zur Verfügung steht wie auf dem Desktop-PC. Auch bei Komponenten wie der 3D-Unterstützung ist es wohl nur noch eine Frage der Zeit, bis auch Pocket-PCs diese ähnlich gut unterstützen wie Desktop-PCs.

Im Zuge dieser Entwicklung dient die vorliegende Arbeit als Grundlage für weitergehende Untersuchungen zum Beispiel im Hinblick auf Kommunikationsfähigkeit zwischen einzelnen Pocket-PCs oder die Fähigkeit zum Aufnehmen und Wiedergeben von Audio- und Videoströmen. Daraus sind Hinweise bzgl. Der Eignung von Pocket-PCs als mobile Radios oder TV-Geräte zu erwarten. Der Erfolg von 3D-Spielen im Desktop-PC-Sektor wirft zudem die Frage auf, ob evtl. auch Pocket-PCs die nötigen Anforderungen für derartige Anwendungen aufbieten können. Nicht zuletzt ist auch eine Untersuchung in der Richtung interessant, inwieweit sich die Funktionalitäten von Handys und Pocket-PCs ähneln, und ob diese beiden Branchen in naher Zukunft gar fusionieren, weil sich die Funktionen beider Gerätetypen dann so sehr überschneiden, dass eine Fusion gerechtfertigt scheint.

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Literaturverzeichnis

Als Quelle zur Programmierung von Desktop-PCs diente das Buch:

Andre LaMothe: Tricks of the Windows Programming Gurus (Second Edition)
Sams Publishing, Indianapolis, Indiana, 2002

Als Quelle zur Programmierung von Pocket-PCs diente das Buch:

J. S. Harbour: Pocket PC Game Programming (using the Windows CE Game API)
Prima Publishing, Rosville, California, 2001

Verwendete Hard- und Software für die Diplomarbeit

Desktop-PC: Intel PentiumII 233/ 64 MB RAM/ 2 GB Festplatte

Pocket PC: Yakumo Delta/ Intel PXA250 Prozessor/
36 MB Speicher (Festplatte + RAM)/Farb-LCD 3,5"

Software (DirectX):
Microsoft Visual C++ 6.0 Introductory Edition
Microsoft DirectX SDK 8.0

Software (GameAPI):
Microsoft Embedded Visual C++ 3.0
GameAPI 1.2

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Denny Bertram
Leipzig, 13.Juni 2003