Grundlagen der Programmierung von Multitasking-Betriebssystemen

Die praktische Anwendung ist der beste Weg, um komplexe Zusammenhänge wie beispielsweise mathematische Gleichungen nachvollziehen zu können. Ein theoretisch erklärter Grafikeffekt ist nutzlos, wenn dieser nicht in Form eines Programms in die Praxis umgesetzt werden kann. Bevor wir uns mit den Grundlagen der Grafikprogrammierung beschäftigen können, müssen wir jedoch einen Weg finden, ein geeignetes Umfeld für unsere Programme zu schaffen.

Am Beispiel des Betriebssystems WINDOWS wird im Folgenden erläutert, wie man ein Programmfenster öffnet und elementaren Zugriff auf die Hardware der Grafikkarte erhält. Der Schwerpunkt steht hierbei auf der Plattformunabhängigkeit – sämtliche Zugriffe auf das Betriebssystem erfolgen deshalb unter Verwendung einer Klasse namens `software_interface`, sodass erfahrene Nutzer von Betriebssystemen wie beispielsweise LINUX oder UNIX keine Probleme haben sollten, die Definition dieses Datentyps entsprechend anzupassen, um auf diese Weise die Quelltexte auf das gewünschten Betriebssystem portieren zu können.

C.1 Initialisierung der Softwaredarstellungen

Jedes unter WINDOWS auszuführende C++-Programm verfügt über eine Funktion namens `WinMain()`, welche die gleiche Aufgabe wie die bekannte Funktion `main()` erfüllt. Der wichtigste Unterschied zwischen den beiden besteht in der erweiterten Parameterkonfiguration:

```
int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int iCmdShow );
```

Vereinfacht ausgedrückt, handelt es sich bei dem ersten Parameter der Funktion `WinMain()` um einen ganzzahligen Wert, mit dessen Hilfe das Programm vom Betriebssystem identifiziert werden kann. Der zweite Parameter besitzt in der

Regel den Wert *NULL*. Die Variable `lpCmdLine` ist ein Zeiger auf einen String, welcher dem Programm eventuell übergebene Kommandozeilenparameter enthält. `iCmdShow` kann schließlich einen von mehreren vordefinierten Werten annehmen, welche das Aussehen des Programmfensters festlegen. Über diesen Parameter kann beispielsweise festgelegt werden, ob ein Fenster maximiert dargestellt werden soll.

Wie bereits erwähnt, wird in diesem Buch der Schwerpunkt auf die mathematischen Zusammenhänge gesetzt, welche die Grundlage der Computergrafik bilden. Da wir im gegebenen Rahmen nicht auf die Feinheiten der Windowsprogrammierung eingehen können, greifen die Programme dieses Buches lediglich auf den Parameter `hInstance` zurück.

Die Initialisierung der Softwaredarstellungen erfolgt unter Verwendung der vorgegebenen Klasse `software_interface`, die eine einzige Instanz namens `screen_interface` besitzt. Die Aufgabe dieser Klasse besteht darin, ein Programmfenster zu öffnen und einen Zeiger auf den Beginn des Videospeichers zu generieren.

Diese Klasse wird auch eingesetzt, um die horizontale und vertikale Auflösung des Bildschirms, in Pixel angegeben, zu verwalten. Auf diese wichtigen Konstanten wird mit Hilfe der Makros `x_res` und `y_res` zugegriffen, um eine ungewollte Veränderung auszuschließen:

```
#define x_res screen_interface.get_xr()
#define y_res screen_interface.get_yr()

class software_interface
{
  private:
  long x_resolution, y_resolution;

    LPDIRECTDRAW main_dd_object;
    LPDIRECTDRAWSURFACE primary_surface;
    DDSURFACEDESC surface_description;
    HWND main_window_handle;

    void initialise_platform
      ( long x, long y, long bit_depth );

  public:
    long get_xr( void ) {  return x_resolution;  }
```

```
  long get_yr( void ) {  return y_resolution;  }

  void open_window
 (
    HINSTANCE hInstance,
     long x, long y, long bit_depth
  );
  void close_window( void );

  void *get_screen_pointer( void );
  void release_screen_pointer( void )
  {  primary_surface->Unlock
    ( surface_description.lpSurface );
  }

  software_interface( void )
  {
    x_resolution = y_resolution = 0;
    main_dd_object = NULL;  primary_surface = NULL;
    main_window_handle = NULL;
    memset( &surface_description, 0,
            sizeof( surface_description ) );
  }
 ~software_interface( void ) {  close_window();  }
} screen_interface;
```

C.1.1 Erstellung des Programmfensters

Um die Quelltexte möglichst übersichtlich gestalten zu können, werden unsere Softwaredarstellungen stets über ein einziges Fenster verfügen, das sich über die gesamte Oberfläche des Bildschirms erstreckt. Dieses Fenster wird während der Ausführung von `screen_interface.open_window()` generiert:

```
void software_interface :: open_window
( HINSTANCE hInstance, long x, long y, long bit_depth )
{
  x_resolution = x;  y_resolution = y;
```

```
WNDCLASS winclass;

winclass.style = CS_OWNDC;
winclass.lpfnWndProc = main_window_procedure;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance = hInstance;
winclass.hIcon = LoadIcon( NULL, IDI_APPLICATION );
winclass.hCursor = LoadCursor( NULL, IDC_ARROW );
winclass.hbrBackground =
  (HBRUSH) GetStockObject( BLACK_BRUSH );
winclass.lpszMenuName = NULL;
winclass.lpszClassName = "Main Window";
if( RegisterClass( &winclass ) == 0 )
  exit_error( "Fehler während der Registrierung des \
            Programmfensters.\n" );

char window_title[] = "3D-Grafik Programmierung";

main_window_handle = CreateWindowEx
(
  WS_EX_TOPMOST,
  "Main Window",
  window_title,
  WS_VISIBLE | WS_POPUP,
  0, 0,
  GetSystemMetrics( SM_CXSCREEN ),
  GetSystemMetrics( SM_CYSCREEN ),
  NULL, NULL, hInstance, NULL
);

if( main_window_handle == 0 )
  exit_error( "Fehler beim Öffnen des \
            Programmfensters.\n" );

ShowCursor( 0 );
```

```
  initialise_platform
  ( x_resolution, y_resolution, bit_depth );
}
```

Um ein Fenster generieren zu können, muss dem Betriebssystem eine entsprechende Beschreibung übermittelt werden. In der WINDOWS-Programmierung wird hierzu die Struktur `WNDCLASS` eingesetzt; die wenigen für uns relevanten Komponenten dieser Struktur sind im gegebenen Quelltext hervorgehoben dargestellt.

Beim Start jedes WINDOWS-Programms übergibt das Betriebssystem der entsprechenden `WinMain()`-Funktion einen ganzzahligen Wert, der innerhalb des Parameters `hInstance` gespeichert wird. Die Verbindung zwischen dem Programm und seinem Fenster wird dadurch hergestellt, dass die Komponente `WNDCLASS::hInstance` mit dem Wert dieses Parameters initialisiert wird.

Ein Multitasking-Programm kann durchaus über mehrere Fenster verfügen. Um diese besser voneinander unterscheiden zu können, wird jedem Fenster einen Namen zugeordnet, welcher der Komponenten `WNDCLASS::lpszClassName` zugewiesen werden muss. Der Name unseres einzigen Fensters lautet `"Main_Window"`.

Auf die Bedeutung der Komponente `WNDCLASS::lpfnWndProc` werden wir erst im Zusammenhang mit der Kommunikation zwischen Programm und Betriebssystem näher eingehen.

Um dem Betriebssystem die Informationen zu übermitteln, die innerhalb von `winclass` gespeichert worden sind, ist der Aufruf von `RegisterClass()` erforderlich. Der Aufruf dieser Funktion kann durchaus fehlschlagen, beispielsweise dann, wenn ein Programm versucht, unterschiedliche Fenster mit demselben Namen zu generieren.

Nachdem ein Fenster registriert worden ist, bleibt es zunächst für den Benutzer unsichtbar. Um das Fenster auf dem Bildschirm anzuzeigen, ist der Aufruf einer Funktion wie `CreateWindowEx()` erforderlich. Die Parameter dieser Funktion legen die Art der Darstellung des Fensters fest.

Das Fenster, für das diese Funktion aufgerufen wird, wird unter Verwendung des vorher festgelegten Namens identifiziert, in unserem Fall `"Main_Window"`. Der Parameter `WS_EX_TOPMOST` gibt an, dass unser Fenster allen bereits geöffneten übergeordnet ist – es darf von keinem anderen verdeckt werden und nimmt sämtliche Eingaben des Benutzers entgegen. Der String `window_title[]` legt die Bezeichnung fest, die auf der Titelleiste des Fensters erscheinen soll.

Die beiden Parameter, die den Wert `0` erhalten, legen die x- und y-Koordinate der oberen linken Ecke des Fensters fest. Die zwei darauffolgenden Parameter geben die Breite und Höhe des Fensters in Pixel an. Unser Fenster soll sich über den gesamten Bildschirm erstrecken – aus diesem Grund werden diese mit Hilfe der

Funktion `GetSystemMetrics()` initialisiert, die für die Konstanten `SM_CXSCREEN` und `SM_CYSCREEN` die aktuelle horizontale bzw. vertikale Auflösung angibt.

Jedes geöffnete Fenster wird vom Betriebssystem mit Hilfe eines ganzzahligen Wertes ungleich `0` identifiziert, der als *Window Handle* bezeichnet wird. Operationen wie die Erweiterung eines Fensters um *High-Level-API*-Komponenten oder das Schließen des Fensters erfolgen unter Verwendung dieses Wertes. Die Kennzahl unseres Programmfensters wird von der Funktion `CreateWindowEx()` erzeugt und in der Variable `main_window_handle` gespeichert.

C.1.2 Kommunikation zwischen Programm und Betriebssystem

Während der Ausführung eines Programms sind sämtliche Eingaben des Benutzers an die Fenster gerichtet, die von der jeweiligen Anwendung geöffnet worden sind. Das Problem hierbei besteht darin, dass dieselbe Eingabe von verschiedenen Fenstern auf unterschiedliche Weise interpretiert werden kann.

Wird in einem Fenster ein Polyeder angezeigt, kann das Drücken der Taste *D* beispielsweise eine Rotation um die y-Achse einleiten, genau wie in den Programmen des 5. Kapitels. Dieselbe Eingabe, auf einem Fenster mit einem Textfeld bezogen, kann aber auch die Aufnahme des Zeichens `'d'` innerhalb eines Strings bedeuten.

Um diese Funktionalität gewährleisten zu können muss jedem Fenster eine eigene Funktion zur Verarbeitung der Benutzereingaben zugeordnet werden. Während der Laufzeit des Programms wird diese Funktion automatisch vom Betriebssystem aufgerufen, sobald der Benutzer eine Eingabe an das jeweilige Fenster richtet.

Unsere Programme besitzen ein einziges Fenster, das über keine besondern Merkmale wie Schaltflächen zum Minimieren oder Maximieren verfügt. Der Aufbau der Funktion `main_window_procedure()` kann aus diesem Grund besonders einfach gestaltet werden:

```
LRESULT CALLBACK main_window_procedure
(
  HWND main_window_handle, UINT message,
  WPARAM wparam, LPARAM lparam
)
{
  if( message == WM_CLOSE )
  {
    PostQuitMessage( 0 ); return 0;
  }
```

```
  return DefWindowProc( main_window_handle, message,
                    wparam, lparam );
}
```

Die Funktion eines Programmfensters besitzt eine fest vorgegebene Parameterkonfiguration. Bei dem ersten Parameter handelt es sich um den Integerwert, mit dessen Hilfe das jeweilige Fenster von Betriebssystem identifiziert wird. Die Variable `message` kann einen von mehreren vordefinieren Werten annehmen: Wenn der Benutzer eine beliebige Taste drückt, nimmt `message` beispielsweise den Wert `WM_KEYDOWN` ein. `wparam` und `lparam` sind schließlich zwei Variablen benutzerdefinierten Typs, die zusätzliche Informationen bezüglich der empfangenen Nachricht enthalten. Wenn der Benutzer beispielsweise Befehle unter Verwendung der Maus erteilt, enthalten diese beiden Variablen die Position des Cursors und den Zustand der Maustasten.

Aufgrund der vielen verschiedenen Werte, die `message` annehmen kann, ist das explizite Eingehen auf jeden möglichen Wert des Parameters `message` nicht unbedingt erforderlich; diese Aufgabe übernimmt die Funktion `DefWindowProc()`, welche mit denselben Parametern wie die Fensterfunktion aufgerufen wird.

Der einzige Fall, der explizit vom Programmierer behandelt werden muss, ist die Verarbeitung der Tastenkombination Alt+F4, welche zum Beenden des Programms eingesetzt wird. In diesem Spezialfall, in dem die Variable `message` den Wert `WM_CLOSE` besitzt, muss die Funktion `PostQuitMessage()` mit dem Parameter `0` aufgerufen werden.

Dieser Aufruf sendet dem Programm die Nachricht `WM_QUIT`, um auf diese Weise das Beenden der Anwendung hervorzurufen. Das Auftreten dieser Nachricht kann durch den Aufruf von `PeekMessage()` festgestellt werden; diese Funktion wird im 5. Kapitel ausführlich besprochen.

Die bloße Definition einer Funktion mit der gegebenen Parameterkonfiguration besitzt zunächst keinen Einfluss auf die Fenster eines Programms; um festzulegen, dass `main_window_procedure()` die Kommunikation zwischen Betriebssystem und unserem Programmfenster übernehmen soll, ist folgende Anweisung erforderlich:

```
winclass.lpfnWndProc = main_window_procedure;
```

Die Variable `winclass` wird während der Ausführung von `screen_interface.open_window()` definiert und enthält die Beschreibung des Programmfensters.

C.1.3 Der Zeiger auf den Anfang des Videospeichers

Sämtliche Zugriffe auf die Grafikhardware erfolgen während der Ausführung unserer Softwaredarstellungen ausschließlich über einen Zeiger namens `*screen`, der auf den Anfang des Videospeichers verweist. Aufgrund des komplexen Aufbaus moderner Betriebssysteme und Grafikkarten erfolgt die Initialisierung dieses Zeigers unter Verwendung der High Level Funktionen der Grafikbibliothek DIRECTDRAW. Um auf diese Funktionen zugreifen zu können, ist eine entsprechend initialisierte Variable vom Typ `LPDIRECTDRAW`, in unserem Fall `main_dd_object`, erforderlich:

```
void software_interface :: initialise_platform
( long x, long y, long bit_depth )
{
  if
  (
    DirectDrawCreate( NULL, &main_dd_object, NULL )
    != DD_OK
  )
  exit_error( "Fehler während der Ausführung von \
              DirectDrawCreate().\n" );

  if
  (
    main_dd_object->SetCooperativeLevel
    (
      main_window_handle,
      DDSCL_ALLOWREBOOT | DDSCL_EXCLUSIVE |
      DDSCL_FULLSCREEN | DDSCL_ALLOWMODEX
    )
    != DD_OK
  )
  exit_error( "Fehler während der Ausführung von \
              SetCooperativeLevel().\n" );

  if
  (
    main_dd_object->SetDisplayMode( x, y, bit_depth )
```

```
    != DD_OK
  )
  exit_error( "Fehler beim Einstellen der gewünschten \
              Auflösung.\n" );

  memset( &surface_description, 0,
          sizeof( surface_description ) );
  surface_description.dwSize =
    sizeof( surface_description );
  surface_description.dwFlags = DDSD_CAPS;
  surface_description.ddsCaps.dwCaps =
    DDSCAPS_PRIMARYSURFACE;

  if
  (
     main_dd_object->CreateSurface
     ( &surface_description, &primary_surface, NULL )
     != DD_OK
   )
    exit_error( "Ausführung von CreateSurface() ergibt \
                FALSE.\n" );
}
```

Im ersten Schritt der Funktion `initialise_platform()` wird die Variable `main_dd_object` durch den Aufruf von `DirectDrawCreate()` initialisiert. Anschließend muss diese mit unserem Fenster, identifiziert durch die ganzzahlige Variable `main_window_handle`, verknüpft werden – dies erfolgt unter Verwendung der Funktion `SetCooperativeLevel()`. Die restlichen Parameter dieser Funktion legen folgende erweiterte Eigenschaften des Programmfensters fest:

Flag	Bedeutung
`DDSCL_ALLOWREBOOT`	ermöglicht ein Abbruch des Programms unter Verwendung der Tastenkombination Strg+Alt+Entf
`DDSCL_FULLSCREEN`	Programmfenster muss sich über die gesamte Oberfläche des Bildschirms erstrecken
`DDSCL_EXCLUSIVE`	muss in Verbindung mit `DDSCL_FULLSCREEN` verwendet werden
`DDSCL_ALLOWMODEX`	Unterstützt zusätzliche Auflösungen wie beispielsweise **320 x 400** Pixel bei einer Farbtiefe von **8** Bit

Die Initialisierung des Zeigers `*screen` mit der Anfangsadresse des Videospeichers erfolgt schließlich durch den Aufruf von `get_screen_pointer()`:

```
void *software_interface::get_screen_pointer( void )
{
  primary_surface->Lock( NULL, &surface_description,
    DDLOCK_SURFACEMEMORYPTR, NULL );

  return surface_description.lpSurface;
}
```

Hirbei kommt eine Variable vom Typ `LPDIRECTDRAWSURFACE` namens `primary_surface` zum Einsatz. Die Initialisierung dieser Variable erfolgt im letzten Schritt der Funktion `initialise_platform()` auf der Grundlage von `main_dd_object` mittels `CreateSurface()`.

Zusätzlich zur Initialisierung von `*screen` bietet `main_dd_object` eine sehr einfache Möglichkeit, die Auflösung und Farbtiefe des Bildschirms temporär während der Laufzeit eines Programms zu verändern. Hierzu muss lediglich die Funktion `SetDisplayMode()` mit benutzerdefinierten Parametern aufgerufen werden. Dieser Aufruf erfolgt im gegebenen Quelltext direkt nach der erfolgreichen Ausführung von `SetCooperativeLevel()`.

C.2 Initialisierung der OpenGL-Darstellungen

Die Initialisierung der Programme, die auf Funktionen der OpenGL-Grafikbibliothek zugreifen, erfolgt unter Verwendung einer Klasse namens `hardware_interface`, die nach demselben Prinzip aufgebaut ist wie die bereits vorgestellten Klasse `software_interface`: Im ersten Schritt wird ein Programmfenster auf der Grundlage des Datentyps `WNDCLASS` geöffnet, anschließend erfolgt die Initialisierung einer Variable vom Typ `HGLRC` namens `rendering_context`, welche mit dem bereits bekannten `main_dd_object` vergleichbar ist.

Bei der Initialisierung von `rendering_context` ist die Hilfsvariable `device_context` beteiligt, die ebenfalls eine Komponente von `hardware_interface` ist. Aufgrund der großen Ähnlichkeit zwischen dieser Klasse und `software_interface` werden lediglich die wichtigsten Unterschiede beschrieben; die vollständige Definition von `hardware_interface` befindet sich in der Datei `"screen_interface.h"`, die Teil des Quelltextes jedes OpenGL-Programms ist.

```
void hardware_interface :: initialise_platform( void )
{
  device_context = GetDC( main_window_handle );

  PIXELFORMATDESCRIPTOR pfd;

  memset( &pfd, 0, sizeof( pfd ) );
  pfd.nSize = sizeof( pfd );
  pfd.nVersion = 1;
  pfd.dwFlags = PFD_DRAW_TO_WINDOW |
              PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
  pfd.iPixelType = PFD_TYPE_RGBA;
  pfd.cColorBits = 32;
  pfd.cDepthBits = 32;
  pfd.iLayerType = PFD_MAIN_PLANE;

  int format = ChoosePixelFormat(device_context, &pfd);
  SetPixelFormat( device_context, format, &pfd );

  rendering_context = wglCreateContext(device_context);
  wglMakeCurrent( device_context, rendering_context );
}
```

Der Initialisierung von `rendering_context` liegt eine Variable namens `pfd` zugrunde, deren wichtigste Komponenten hervorgehoben sind. Wichtig ist, dass OpenGL die Generierung von Zeiger auf den Videospeicher nicht unterstützt – sämtliche Zugriffe auf die Grafikhardware erfolgen unter Verwendung von High Level Funktionen wie `glColor3ub()`, `glVertex3d()`, usw. Die Programmierung mit OpenGL wird ab dem zweiten Kapitel ausführlich behandelt.

C.3 Initialisierung der DirectX-Darstellungen

Die Programme, die auf DirectX 9 zugreifen, werden grundsätzlich nach dem bereits vorgestellten Prinzip unter Verwendung einer dritten Klasse namens `directx_interface` initialisiert. Die wichtigste Komponente dieser Klasse ist eine Variable vom Typ `LPDIRECT3DDEVICE9` namens `rendering_device`, die alle Zugriffe auf die Grafikhardware steuert. Daneben gibt es noch einen Zeiger `vertex_buffer` auf einem Array aus Vertices sowie eine Hilfsvariable `hardware_device` vom Typ `LPDIRECT3D9`.

Die Initialisierung dieser Komponenten erfolgt während der Ausführung von `initialise_platform()`; auch in diesem Zusammenhang spielt die ganzzahlige Variable `main_window_handle` eine wichtige Rolle, sodass diese Funktion wie gehabt erst nach dem Öffnen des Programmfensters aufgerufen werden darf:

```
void directx_interface::initialise_platform( void )
{
  if( (hardware_device = Direct3DCreate9
     ( D3D_SDK_VERSION )) == NULL )
    exit_error( "DirectX 9.0 wurde auf dem System \
                nicht gefunden.\n" );

  D3DPRESENT_PARAMETERS parameters;

  memset( &parameters, 0,
         sizeof( D3DPRESENT_PARAMETERS ) );
  parameters.Windowed = TRUE;
  parameters.SwapEffect = D3DSWAPEFFECT_DISCARD;
  parameters.BackBufferFormat = D3DFMT_UNKNOWN;
  parameters.EnableAutoDepthStencil = TRUE;
  parameters.AutoDepthStencilFormat = D3DFMT_D16;

  if
  (
    FAILED
    (
      hardware_device->CreateDevice
      (
        D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
        main_window_handle,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &parameters, &rendering_device
      )
    )
  )
  exit_error( "Fehler während der Ausführung von \
            Ausführung von CreateDevice().\n" );
```

```
  if
  (
    FAILED
    (
      rendering_device->CreateVertexBuffer
      (
        largest_point_count * sizeof( hvertex ),
        0, render_mode,
        D3DPOOL_DEFAULT, &vertex_buffer, NULL
      )
    )
  )
  exit_error( "Ausführung von CreateVertexBuffer() ist \
              fehlgeschlagen.\n" );

  rendering_device->SetRenderState
    ( D3DRS_CULLMODE, D3DCULL_NONE );
  rendering_device->SetRenderState
    ( D3DRS_LIGHTING, FALSE );
  rendering_device->SetRenderState
    ( D3DRS_ZENABLE, TRUE );
}
```

Der genaue Zusammenhang zwischen den Komponenten `rendering_device` und `vertex_buffer` bei der Darstellung beliebiger Gegenstände wird ab dem 3. Kapitel besprochen.

Kompilierung der Quelltexte

Auf der beiliegenden CD befinden sich die vollständigen Quelltexte und die lauffähigen Versionen der im Buch vorgestellten Programme. Um eine höhere Übersicht gewährleisten zu können, sind die einzelnen Anwendungen nach Kapitel gruppiert, wobei Dateien mit den Quelltexten desselben Programms zusätzlich in eigenen Verzeichnissen gespeichert sind. Neben den Quelltexten wird das unter Windows arbeitende Freeware-Compilersystem CYGWIN mitgeliefert.

Die in diesem Buch vorgestellten Beispielprogramme greifen auf Header-Dateien und Bibliotheken zu, die zwar im Lieferumfang der meisten gängigen Compilersysteme enthalten sind, aber nicht standardmäßig bei der Erstellung eines Programms eingebunden werden:

Art der Darstellung	Header	Libraries
Software	`ddraw.h`	`ddraw.lib`
OPENGL	`gl\gl.h; gl\glu.h`	`opengl32.lib; glu32.lib`
DIRECTX 9	`D3dx9.h`	`d3d9.lib; d3dx9.lib`

Diese Dateien sind Teil des OPENGL SDK bzw. DIRECTX SDK, die im Internet zum freien Download zur Verfügung gestellt werden. Der Zugriff auf diese Dateien während des Kompilierungsvorganges muss vom Programmierer explizit eingestellt werden.

Die Art, wie dieses Einbinden erfolgt, variiert je nach eingesetztem Compilersystem. Im Folgenden wird die Übersetzung der Beispielprogramme mit dem CYGWIN – Compiler sowie zwei weit verbreiteten Compilersystemen von MICROSOFT beschrieben.

D.1 Das Cygwin - Compilersystem

CYGWIN ist ein sehr leistungsfähiges und hochwertiges Freeware-Compilersystem für WINDOWS, das im Internet auf der Seite `www.cygwin.com` vorgestellt wird. Die Version dieses Compilers, die für die Übersetzung der Beispielprogramme eingesetzt worden ist, befindet sich auf der beiliegenden CD.

D.1.1 Installation des Cygwin - Compilers

Die Benutzung des Compilersystems setzt eine zweistufige Installation voraus. Im ersten Schritt muss das Verzeichnis `DXSDK` von der CD vollständig und unverändert auf die Festplatte kopiert werden. Die Klasse `software_interface`, welche die Grundlage sämtlicher Softwaredarstellungen darstellt, verwendet die in diesem Verzeichnis enthaltenen Bibliotheken. Sollte bereits ein gleichnamiges Verzeichnis existieren, kann auch ein anderer Name angegeben werden.

Die im zweiten Schritt erfolgende Hauptinstallation wird durch den Aufruf der Datei `setup.exe` im Verzeichnis `COMPILER` auf der CD gestartet. Nach dem Begrüßungsbildschirm muss die Installationsart festgelegt werden. Um den Compiler von der CD des Buches zu installieren, muss:

```
Install from Local Directory
```

ausgewählt werden. Anschließend sind die Quell- und Zielverzeichnisse anzugeben; in der Regel reicht jedoch die Bestätigung der bereits vorhandenen Einträge aus. Bei der Angabe zusätzlicher Installationsinformationen müssen unter Windows:

```
Default Text File Type: DOS  und
Install For: Just Me
```

aktiviert sein. Bei der Auswahl der Packages muss lediglich die Schaltfläche `Next` gedrückt werden. Nach der Installation lässt sich der Compiler durch die Aktivierung von:

```
Create Desktop Icon
Add to Start Menu
```

bequem starten.

D.1.2 Erstellung lauffähiger Dateien

Nach der Installation des CYGWIN – Compilersystems lässt sich dieses unter Verwendung der entsprechenden Desktop-Verknüpfung oder mit `Start -> Programme -> Cygnus Solutions -> Cygnus Bash Shell` starten.

Die Kompilierung der Softwaredarstellungen wird am Beispiel des Programms `a6_11` vorgestellt. Zunächst werden die Quelltextdateien von der CD in ein gleichnamiges Festplattenverzeichnis kopiert. Anschließend muss in der Shell des Compilersystems unter Verwendung des Befehls:

```
cd c:/a6_11
```

in das betreffende Verzeichnis gewechselt werden. Hierbei ist es von großer Wichtigkeit, dass anstelle des in WINDOWS üblichen Backslash '\' das Trennzeichen '/' eingesetzt wird. Das jeweils aktuelle Verzeichnis ist rechts über der Position des Cursors dargestellt. Der Kompiliervorgang wird durch folgende Anweisung gestartet:

```
g++ application.cpp -o application.exe -Ic:/dxsdk/include -Lc:/dxsdk/lib
-fvtable-thunks -mwindows -e _mainCRTStartup -ffreestanding -lddraw
```

Ungeachtet der Tatsache, dass dieser Befehl sich über mehrere Zeilen erstreckt, handelt es sich hierbei um eine einzige Anweisung, die ohne Rücksicht auf den Zeilenumbruch einzugeben ist. Das Eintippen dieser Eingabe ist jedoch nur ein einziges Mal erforderlich, da sie beliebig oft mit der Pfeil o – Taste wiederholt werden kann. Nach dem Beenden der Shell mit:

```
exit
```

werden die letzten Eingaben dauerhaft gespeichert.

Die Schalter:

```
-Ic:/dxsdk/include
```

und

```
-Lc:/dxsdk/lib
```

geben hierbei an, dass für die Übersetzung von a6_11 zusätzliche Header und Bibliotheken eingesetzt werden, die im Verzeichnis c:\dxsdk\include bzw. c:\dxsdk\lib zu finden sind. Bei c:\dxsdk handelt es sich um das Verzeichnis, das im ersten Schritt der Installation von der CD auf die Festplatte kopiert worden ist.

Die fertige Anwendung a6_11.exe kann entweder in einem Explorer – Fenster oder direkt in der Shell mit Hilfe des Befehls:

```
./a6_11
```

ausgeführt werden. Hierzu muss sich die Datei cygwin1.dll, die im Lieferumfang des Compilers enthalten ist, innerhalb des Festplattenverzeichnisses mit der Datei a6_11.cpp befinden.

Die Kompilierung der OpenGL-Programme erfolgt nach demselben Prinzip, unter Verwendung der folgenden Anweisung:

```
g++ hardware.cpp -o hardware.exe -lglu32 -lopengl32 -fvtable-thunks -
mwindows -e _mainCRTStartup -ffreestanding
```

Auch hierbei handelt es sich um eine einzige Anweisung, die ohne Beachtung des Zeilenumbruches einzutippen ist. Die auf diese Weise erstellte ausführbare Datei `application.exe` befindet sich anschließend im gleichnamigen Verzeichnis und lässt sich wie oben beschrieben ausführen.

Der Cygwin-Compiler ist mit dem von Microsoft veröffentlichten DirectX 9.0 SDK nicht kompatibel – aus diesem Grund lassen sich die in diesem Buch vorgestellten DirectX-Programme nicht mit Cygwin kompilieren. Für die Übersetzung dieser Programme muss auf ein anderes Compilersystem zurückgegriffen werden.

D.2 Microsoft Visual C++ 6.0

Die Übersetzung der Quelltexte mit dem Compiler Microsoft Visual C++ 6.0 erfordert im ersten Schritt die Einrichtung eines neuen Projektes. Die hierzu notwendigen Schritte werden im Folgenden anhand der Erstellung eines Projektes für die Softwaredarstellungen beschrieben. Auf die Übersetzung der OpenGL- und DirectX-Programme wird im Anschluss eingegangen.

Die Einrichtung des Projektes erfolgt nach dem Start des Compilers durch `Datei -> Neu`. Auf der linken Seite des darauffolgenden Menüs muss man **`Win32-Anwendung`** auswählen, auf der rechten Seite im Feld Projektname beispielsweise die Bezeichnung **`application`** eingeben. Im unteren Textfeld `Pfad` sollte automatisch der Entrag **`c:\application`** erscheinen – weiter mit `OK`. Im darauffolgenden Menüfeld muss der Eintrag **`Ein leeres Projekt`** ausgewählt, anschließend `Fertigstellen` gedrückt und das letzte Menü mit `OK` geschlossen werden.

Auf der Festplatte sollte jetzt ein neues Verzeichnis namens `c:\application` vorhanden sein, das ein Unterverzeichnis `c:\application\debug` sowie vier automatisch generierte Dateien enthält. Neben diesen müssen sämtliche Quelltextdateien aus dem Verzeichnis `a2_1` auf der CD in das neue Verzeichnis `c:\application` kopiert werden; dieser Vorgang kann über den Datei Manager erfolgen, während das Fenster des Compilers minimiert ist.

In der Entwicklungsumgebung wird anschließend `Projekt -> Dem Projekt hinzufügen -> Dateien` ausgewählt. Im Eintrag `Dateiname` des darauffolgenden Menüs muss **`c:\application\application.cpp`** eingegeben und mit `OK` bestätigt werden.

Danach muss `Erstellen -> Aktive Konfiguration festlegen -> **application-Win32 Release**` ausgewählt, und schließlich das Menüfeld `Projekt -> Einstellungen` aufgerufen werden. Im Textfeld links oben mit dem Titel `Einstellungen für` ist der Eintrag **`Alle Konfigurationen`** festzulegen. In der Kartei `Linker` muss im Textfeld `Objekt-/Bibliothek-Module` zusätzlich zu den bereits bestehenden Einträgen der Dateiname **`ddraw.lib`** eingefügt, von den anderen durch ein Leerzeichen getrennt, und das Menü mit `OK` geschlossen werden.

Die Kompilierung des Quelltextes erfolgt durch `Erstellen -> application.exe erstellen`, die Ausführung des Programms durch `Erstellen -> Ausführen von application.exe`.

Beim Schließen der Entwicklungsumgebung wird gefragt, ob die Änderungen am Arbeitsbereich `application` gespeichert werden sollen. Diese Frage ist mit `Ja` zu beantworten.

Möchte man ein anderes Programm kompilieren, ist eine erneute Ausführung der hier beschriebenen Vorgehensweise nicht erforderlich. Die Hauptdatei sämtlicher Softwaredarstellungen besitzt stets den Namen `application.cpp`; die automatisch erstellten Dateien `application.dsp`, `application.dsw`, `application.opt` und `application.ncb` brauchen nur in das Verzeichnis mit den neuen Quelltexten kopiert zu werden. Der Name dieses Verzeichnisses lässt sich beliebig auswählen.

Nach dem Start des Compilers durch das Doppelklicken auf die Datei `application.dsw` lässt sich das neue Programm wie gehabt über das Menüfeld `Erstellen` übersetzen und ausführen.

Die OpenGL-Programme werden nach demselben Prinzip kompiliert; der einzige Unterschied ist, dass im Textfeld `Linker -> Objekt-/Bibliothek-Module` neben den Standardbibliotheken die Dateinamen **`opengl32.lib`** und **`glu32.lib`** eingetragen werden müssen. Diese Bibliotheken sind Teil dieses Microsoft-Compilersystems.

D.2.1 Übersetzung der DirectX-Programme

Im vorherigen Abschnitt wird davon ausgegangen, dass die Bibliotheken der Software- und OpenGL-Darstellungen im Lieferumfang des Compilers enthalten sind. Für die DirectX 9 Programme gilt diese Voraussetzung im Allgemeinen nicht; wenn die hierfür benötigten Header und Bibliotheken auf der Festplatte beispielsweise in einem Verzeichnis namens `c:\dx9sdk` enthalten sind, muss dieser Pfad dem Compiler bei der Erstellung des Projektes explizit übermittelt werden.

Im ersten Schritt müssen die Dateien `d3d9.lib` und `d3dx9.lib` mit Hilfe der im letzten Abschnitt beschriebenen Vorgehensweise in das Projekt eingebunden werden. Anschließend ist aus dem Hauptmenü des Compilers der Eintrag `Extras ->`

`Optionen -> Verzeichnisse` auszuwählen. Im Feld `Verzeichnisse anzeigen für` muss der Eintrag **`Include-Dateien`** markiert werden.

Die bereits bestehenden Einträge des großen Textfeldes `Verzeichnisse` sind anschließend um **`c:\dx9sdk\include`** zu erweitern. Sollte sich die Datei `d3dx9.h` in einem anderen Verzeichnis befinden, lässt sich dieses mit Hilfe der Möglichkeit zum Durchsuchen auswählen.

Um das Verzeichnis der Bibliotheken festzulegen, muss im Feld `Verzeichnisse anzeigen` zunächst der Eintrag `Bibliothekdateien` markiert werden. In das dazugehörige Textfeld muss schließlich der Pfad des Verzeichnisses mit den Dateien `d3d9.lib` und `d3dx9.lib` eingetragen werden; dieser Pfad lautet beispielsweise **`c:\dx9sdk\lib\x86`**.

Nach dem Erstellen des Projektes lassen sich die DIRECTX Programme genau wie die Software- und OPENGL-Darstellungen kompilieren und ausführen.

D.3 Microsoft Visual Studio NET / 2005 / 2008

Die Übersetzung der Quelltexte mit diesem dritten Compilersystem erfolgt ebenfalls nach Erstellung eines entsprechenden Projektes. Um das erste Beispielprogramm `a2_1` zu kompilieren, muss zunächst VISUAL STUDIO gestartet, anschließend der Eintrag `File -> New -> Project -> Visual C++ Projects -> Win32 -> Win32 Project` ausgewählt werden.

Als Projektname kann beispielsweise **`application`** eingegeben (der Pfad sollte **`c:\`** sein) und mit `OK` bestätigt werden. In der Registrierkarte `Application Settings` ist der Eintrag `Empty project` anzukreuzen (der Eintrag `Windows application` müsste bereits ausgewählt sein) und das Menü mit `Finish` zu schließen.

Der Compiler generiert automatisch ein neues Festplattenverzeichnis `c:\application` mit mehreren Dateien. Zusätzlich zu diesen Dateien müssen die Quelltexte aus dem Verzeichnis `a2_1` auf der CD mit Hilfe des Datei Manager nach `c:\application` kopiert werden.

Während dieses Kopiervorganges sollte das Hauptfenster des Compilers minimiert sein. Anschließend müssen die Quelltexte mit dem Projekt verbunden werden. Dies erfolgt über `Project -> Add Existing Item`. Bei der Pfadangabe muss **`c:\application\application.cpp`** eingeben werden.

Das Einbinden der Bibliothek `ddraw.lib` erfolgt unter Verwendung der folgenden Vorgehensweise: Hauptmenü MICROSOFT VISUAL STUDIO -> `Project -> Properties` -> Verzeichnis `Configuration Properties` -> Schaltfläche `Configuration Manager -> Active solution configuration` -> Menüpunkt **`Release`** auswählen -> `Close`.

Verzeichnis `Configuration Properties` -> Verzeichnis `Linker` -> `Input` -> `Additional Dependencies` -> **`ddraw.lib`** eintragen.

Das Fenster `Properties` kann anschließend mit `OK` geschlossen werden. Die Übersetzung des Programms erfolgt über `Build` -> `Rebuild Solution`. Die ausführbare Datei `application.exe` befindet sich im Verzeichnis `c:\application\release` und lässt sich kann durch Doppelklick ausführen.

Die OpenGL-Programme werden unter Verwendung derselben Vorgehensweise übersetzt; anstelle von `ddraw.lib` müssen jedoch die Bibliotheken **`opengl32.lib`** und **`glu32.lib`** eingebunden werden.

Bei der Kompilierung der DirectX 9 Programme müssen zusätzlich zum beschriebenen Aufbau eines Projektes die Verzeichnisse mit den Dateien `d3dx9.h` sowie `d3d9.lib` und `d3dx9.lib` angegeben werden. Für den Header erfolgt dies über Hauptmenü -> `Project` -> `Properties` -> Verzeichnis `Configuration Properties` -> Verzeichnis `C/C++` -> `General` -> `Additional Include Directories`. Für die beiden Bibliotheken: Hauptmenü -> `Project` -> `Properties` -> Verzeichnis `Configuration Properties` -> Verzeichnis `Linker` -> `Input` -> `Additional Directories`.