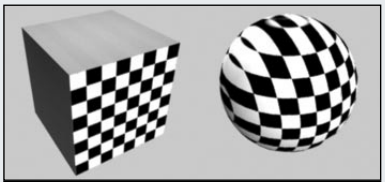
Direct3D 9 – Teil II

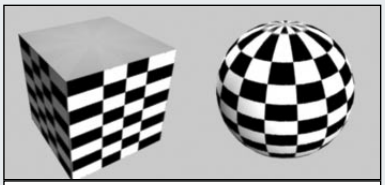
Kugel und Würfel im besten Licht

Machen Sie mehr aus simplen geometrischen Primitiven: Texturen und Beleuchtung verführen und verzaubern den Betrachter Ihrer virtuellen Welten.

Carsten Dachsbacher



Planare Projektion: So bestimmen Sie am leichtesten Textur-Koordinaten durch eine Projektion.



Thema mit Variationen: Eine zylindrische Projektion.

In der letzten Ausgabe von PC Underground haben Sie erfahren, wie Sie Direct3D initialisieren und geometrische Primitive mit Vertex Buffers rendern. Jetzt kümmern Sie sich darum, wie die Oberflächen erscheinen. Dazu benötigen Sie Texturen und Beleuchtungseffekte. Dieser Artikel führt Ihnen die notwendigen Schritte vor.

Die Verwendung von Texturen ist eine der am häufigsten eingesetzten Techniken bei der 3D-Grafik. Dabei wird ein, meist zweidimensionales, gegebenes Bild auf eine Oberfläche projiziert. Somit können verschiedene Punkte wie auf einem Dreieck unterschiedliche Farbwerte besitzen. Die Motivation ist einfach: Sie wollen eine hohe visuelle bei geringer geometrischer Komplexität erzielen. Die simple Abbildung einer Textur auf eine Oberfläche ist der einfachste Fall. Heutige Grafikkarten bieten enorm leistungsfähige Texturierungs-Features und eine breite Palette von und für Texturen, wie Bump- oder Gloss-Mapping, Toon-Shading und Shadow Maps.

Texturen können dabei entweder ein-, zwei- oder dreidimensionale Daten enthalten, statisch oder dynamisch sein. Die Art und Weise, wie Sie Texturdaten interpretieren bzw. auslesen, konfigurieren Sie über Render States, bedingt durch die Entwicklung der Grafik-Hardware und APIs. Auf neueren Grafikkarten programmieren Sie dies frei in den so genannten Pixel Shaders oder Fragment Programs.

Texturen in Direct3D

Um eine statische Textur in Direct3D zu laden und zu verwenden, nutzen Sie die Hilfsfunktionen aus der Direct3D-Bibliothek (D3DX). In Direct3D 9 greifen Sie über das *IDirect3DTexture9*-Interface auf Textur-Objekte zu und ändern so die Textur. Mit diesen Hilfsfunktionen lesen Sie mit nur einem Funktionsaufruf eine Textur aus einer Bilddatei im Format *.bmp, .dds, .dib, .jpg, .png* oder *.tga* aus, erzeugen ein Textur-Objekt und übergeben die Daten:

```
LPDIRECT3DDEVICE9 pD3DDevice;
...
LPDIRECT3DVERTEXBUFFER9 pMeshVB=
                                NULL;

D3DXCreateTextureFromFile(
 pD3DDevice, „bild.bmp“, &pTexture );
```

Dieser Aufruf variiert vereinfacht die Funktion *D3DXCreateTextureFromFileEx*, um eine statische 2D-Texture zu laden. Letztere bietet zusätzliche Parameter, um das Bild zu skalieren, die Textur zu verwenden, sie im Memory Pool zu platzieren und um Mip Maps zu generieren. Zusätzlich benötigen Sie Information darüber, wie die Textur auf Ihr 3D-Objekt, also bei-

spielsweise die Dreiecke, abgebildet wird. Dazu nutzen Sie Textur-Koordinaten: Jedem Vertex (Eckpunkt) Ihres Dreiecksnetzes weisen Sie eine Koordinate innerhalb der Textur zu. Diese Koordinate wird beim Rendering perspektivisch korrekt interpoliert und somit für jeden Pixel des Bildes der auszulesende Texel, der Bildpunkt der Textur, bestimmt. Die Textur-Koordinaten, die Sie explizit für jeden Vertex angeben müssen, bestimmen Sie auf verschiedene Weise. Sie erhalten diese aus einem Modellierungsprogramm, mit dem das 3D-Objekt angelegt wurde, wenn parametrische Flächen wie Spline Patches verwendet werden, oder Sie texturieren das Objekt von Hand. Dabei legen Sie selbst die Textur-Koordinaten für alle Vertices fest. Ganz allgemein können Sie noch für jede Art von Objekten die Textur-Koordinaten durch eine Projektion bestimmen. Häufig verwendete Projektionen zeigen die vier Bilder mit Würfel und Kugel im Schachbrettmuster.
Die Textur-Koordinaten geben Sie mit den anderen Daten im Vertex-Buffer an, indem Sie zunächst das Vertex-Format erweitern:

```
typedef struct
{
  float x, y, z;
  DWORD color;
  float s, t;
}MESHVERTEX;
```

Zudem passen Sie die flexible Vertex-Formatbeschreibung an:

```
#define FVF_MESHVERTEX
  ( D3DFVF_XYZ |
    D3DFVF_DIFFUSE |
    D3DFVF_TEX1 )

MESHVERTEX *pData = NULL;

pMeshVB->Lock( 0, 0,
  (void**)&pData, D3DLOCK_DISCARD );

for ( i = 0; i < nVertices; i++ )
{
  // alle Daten pro Vertex schreiben:
  // x, y, z, color, s, t
  pData->s = ...;
  pData->t = ...;
  ...

  pData ++;
}

pMeshVB->Unlock();
```

Um Ihr 3D-Objekt texturiert zu rendern, müssen Sie Direct3D mitteilen, welche Textur Sie verwenden wollen, wobei Sie mit heutiger 3D-Hardware mehrere Texturen gleichzeitig einsetzen können. Diese Texturierungs-Einheiten auf der Grafikkarte nennt man, auch wegen der Verknüpfungen miteinander, Texture Stages. Für jede Texture Stage legen Sie die verwendete Textur mit der *IDirect3DDevice9::SetTexture Methode* fest:

```
pD3DDevice->SetTexture(0, pTexture );
```

Die Anzahl der gleichzeitig von der Hardware unterstützten Texturen überprüfen Sie anhand der *Device Caps* in den Variablen *D3DCAPS.MaxSimultaneousTextureStages* und *D3DCAPS9.MaxTextureBlendingStages*.

Textur im Eigenbau

Für den Fall, dass Sie Texturen selbst in Ihrem Programm erzeugen und keine Bilddateien verwenden wollen, bietet die D3DX-Bibliothek passende Funktionen. Mit der *D3DXCreateTexture*-Methode legen Sie eine Textur mit beliebigen Auflösungen und im Texel-Format an. Dabei geben Sie den Speicherbereich an, in dem die Textur liegen soll. Weiterhin bestimmen Sie die Verwendung, indem Sie die Textur dynamisch (d.h. der Inhalt ändert sich) oder als Render Target deklarieren. Nur die Fähigkeiten der Grafikkarte limitieren diese Einstellungen, weshalb Sie die *Caps* und die Rückgabewerte der Methode beachten müssen.
Eine statische Textur, zunächst ohne Inhalt, mit 256x256 Texels, 32-Bit-RGBA-Format und Mip Maps legen Sie wie folgt an:

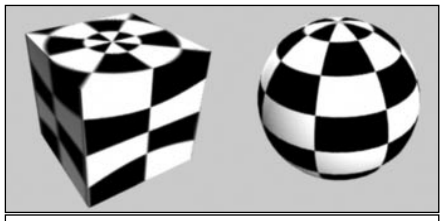
```
D3DXCreateTexture(
 pD3DDevice, 256, 256, 0, 0,
 D3DFMT_A8R8G8B8, D3DPOOL_MANAGED,
 pTexture );
```

Beachten Sie dabei, dass die Parameter der tatsächlich erzeugten Textur von den angegebenen abweichen können. Dies bedingt die *D3DXCheckTextureRequirements*-Funktion, mit der Sie Parameter auf ihre Validität überprüfen. Zudem können Sie ganz darauf verzichten, eine der D3DX-Methoden zu verwenden, um die Textur zu erzeugen. Dies kann z. B. wünschenswert sein, wenn Sie die D3DX-Bibliothek, die Sie statisch linken müssen, nicht in Ihrem Projekt verwenden wollen. So erhalten Sie kleinere ausführbare Dateien. In diesem Fall verwenden Sie die Methode *IDirect3DDevice9::CreateTexture*. Diese verwendet prinzipiell dieselben Parameter, passt die Parameter aber nicht an:

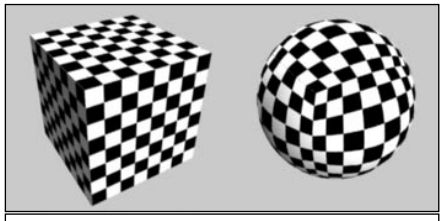
```
pD3DDevice->CreateTexture(
 256, 256, 0, 0, D3DFMT_A8R8G8B8,
 D3DPOOL_MANAGED, &pTexture, NULL );
```

Jetzt bleibt noch die Aufgabe, die Textur-Daten zu übergeben. Dazu müssen Sie die Textur per *Lock* verschließen, um auf die Daten zuzugreifen. Nach dem *Lock*-Befehl können Sie die Daten lesen und/oder schreiben (je nach Modus) und mit einem *Unlock*-Befehl die Änderungen wirksam werden lassen. Das Lock-Kommando des *IDirect3DTexture9* Interface füllt eine *D3DLOCKED_RECT*-Struktur aus, in der die notwendigen Daten für den Zugriff stehen:

```
typedef struct _D3DLOCKED_RECT {
    INT Pitch;
    void *pBits;
} D3DLOCKED_RECT;
```



Kugelnde Würfel: Eine sphärische Projektion mit breitem Muster.



Schach in 3D: Diese kubische Projektion mustert den Würfel wie ein Schachbrett.

Locking Flags für *LockRect*

Eintrag	Beschreibung
D3DLOCK_DISCARD	Write-Only-Zugriff auf die Textur, der gesamte *Rect*-Bereich wird überschrieben.
D3DLOCK_NO_DIRTY_UPDATE	*Dirty Regions* sind Bereiche in Texturen, für die ein Update zur Grafikkarte notwendig ist. Wenn Sie nicht wollen, dass der gerade bearbeitete Bereich als *dirty* markiert wird, wählen Sie dieses Flag.
D3DLOCK_NO_SYSLOCK	Während des Lockings wird das System gestoppt; dann können andere Tasks weiter arbeiten.
D3DLOCK_READONLY	Read-Only-Zugriff auf die Texture

```
HRESULT IDirect3DTexture9::LockRect(
  UINT Level,
  D3DLOCKED_RECT *pLockedRect,
  CONST RECT *pRect,
  DWORD Flags );
```

Dabei ist *pBits* ein Zeiger auf den Speicherbereich der Textur. *Pitch* gibt den Abstand zweier Zeilen der Textur im Speicher in Bytes an. Für das Locking müssen Sie angeben, was Sie mit der Textur anstellen wollen. Flags, die Sie miteinander kombinieren können, beschreibt die Tabelle. Um auf die gesamte Textur zuzugreifen, geben Sie für *pRect* den Wert *NULL* an:

```
D3DLOCKED_RECT lockedRect;

pTexture->LockRect(
  0, &lockedRect, NULL, 0 );
```

Anschließend füllen Sie die Textur mit Daten. Das Beispiel speichert sie im 32-Bit-RGBA-Format. Der Pitch-Wert, den Sie unbedingt beachten müssen, ist aber in Bytes angegeben. Die Quelldaten der Textur, im Array *texData* gegeben, kopieren Sie und geben sie am Ende wieder frei:

```
const int sizeX = 256;
const int sizeY = 256;
DWORD texData[ 256 * 256 ] = ...;

BYTE *pDest=(BYTE*)lockedRect.pBits;

DWORD *pSource = texData;

for ( y = 0; y < sizeY; y++ )
{

  DWORD *pLine = (DWORD*)
    &pDest[ y * lockedRect.Pitch ];

  for ( x = 0; x < sizeX; x++ )
        *(pLine++) = *(pSource++);
}

pTexture->UnlockRect( 0 );
```

Mip Mapping

Bisher haben wir Ihnen den ersten Parameter der *LockRect*-Methode und die Mip Mapping Parameter verschwiegen. Um das Mip Mapping zu erklären, folgen Sie uns zu einem Abstecher in die Sampling-Theorie. Das Texture Mapping bildet eine Textur zunächst per Rendering auf dem Bildschirm ab. Dabei wird für jeden Pixel, der im endgültigen Bild gesetzt wird, der dazugehörige Texel der Textur bestimmt und ausgelesen – die Textur also an verschiedenen Stellen abgetastet. Die Abstände der abgetasteten Texel hängen von der Abbildung der Textur auf dem Dreieck sowie von den Betrachter-Parametern ab und bestimmen somit die Abtastfrequenz. Der Inhalt der Textur lässt sich wiederum als ein bestimmtes Signal interpretieren. Die Sampling-Theorie besagt aber, dass die Abtastung eines Signals mit der doppelten Frequenz erfolgen muss, wie die höchste Frequenz in der Signalquelle (unsere Textur) schwingt.

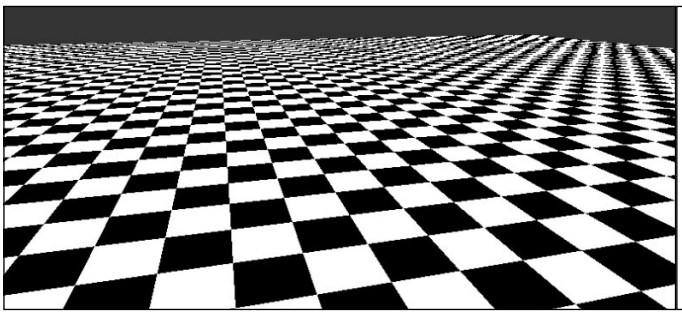
Daraus lässt sich folgern: Wird eine Textur verkleinert auf dem Bildschirm dargestellt und die Bildschirmauflösung ist nicht hoch genug, so treten Abtast-Artefakte bzw. -fehler auf. Frequenzen sind also sichtbar, die in der eigentlichen Textur nicht vorhanden sind, so genannte Aliasing Effekte.

Um diesem vorzubeugen, setzen Sie Mip Maps ein. Dabei handelt es sich um niedriger aufgelöste Varianten der ursprünglichen Textur, die die Grafikkarte selbstständig auswählt, um das Aliasing zu verwenden. Mip Maps können Sie entweder automatisch eine Textur erzeugen lassen, wenn Ihre Grafikkarte dies unterstützt (*D3DCAPS2_CANAUTOGENMIPMAP*), oder explizit eine Textur angeben. Das nächste Bild zeigt das Schachbrett-Muster aus dem vorigen Bild mit Mip Maps, wovon die Qualität der Darstellung deutlich profitiert. Im Folgenden sehen Sie dieselbe Situation mit unterschiedlich eingefärbten Mip Maps.

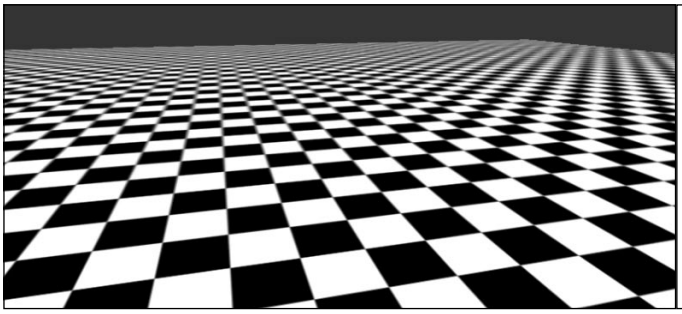
Wie eine Textur an einer Koordinate ausgelesen wird, bestimmen Sie mit der Methode *IDirect3DDevice9::SetSamplerState*. Dabei lässt sich die Adressierung und die Abtastung einstellen. Bei der Adressierung stellen Sie z.B. ein, ob eine Textur gekachelt oder ge-*clamped* wird, d.h. bei Überschreiten der Textur-Grenzen der letzte Texel wiederholt wird. Interessant im Kontext des Mip Mapping sind die so genannten *Magnification*, *Minification* und Mip Filter. Beim Fall von Magnification sehen Sie die Textur größer auf dem Bildschirm als sie ist. Beim Auslesen eines einzelnen Texels können Sie Farbwerte der umgebenden Texel interpolieren. Die häufigste Technik ist die *bilineare* Interpolation, welche die vier nächsten Texel heranzieht. Bei der verkleinerten Darstellung (Minification) sind ebenfalls verschiedene Abtastmodi wählbar.

Wie erwähnt, kann die Grafikkarte die entsprechende Mip Map Stufe der Textur selbstständig wählen. Es ist auch möglich, statt zwischen zwei Stufen hin und her zu schalten, zwischen diesen zu interpolieren: das so genannte *tri-lineare* Filtering.

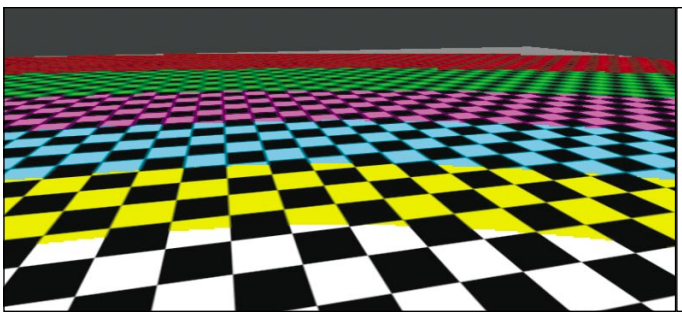
Diese Einstellungen finden Sie im Beispielprogramm, dessen Wirkungen Sie am Bildschirm nachvollziehen können.



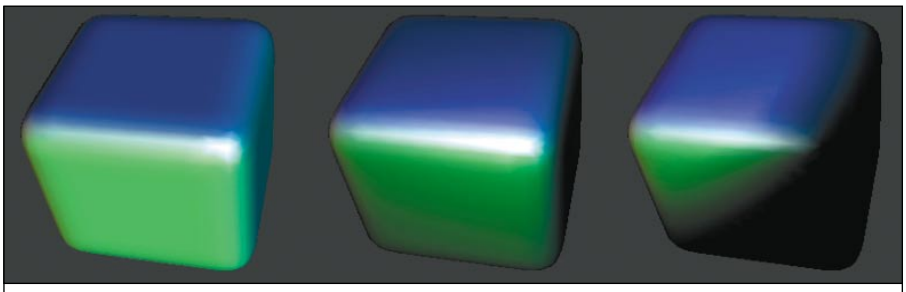
Aliasing: Abtast-Artefakte treten auf, wenn Sie ohne Mip Mapping arbeiten.



Mip Mapping: Wenn die Aliasing Artefakte verschwinden, wird die Darstellung glaubhafter.



Eingefärbt: Unterschiedlich eingefärbte Mip Maps tauchen die Landschaft in buntes Licht.



Beleuchtet: Unser Beispielprogramm zeigt den Würfel, den gleich drei Typen von Lichtquellen anstrahlen: Directional, Point mit Attenuation und Spot Light.

Info

- **www.microsoft.com/downloads/details.aspx?FamilyID=124552ff-8363-47fd-8f3b-36c226e04c85&DisplayLang=en**
- **www.dachsbacher.de/pcu**
- **www.ati.com**
- **www.nvidia.com**

Es werde Licht

Direct3D und heutige Grafikhardware erlaubt es, die Beleuchtung lokal zu berechnen. Lokal bedeutet, dass die Farbe einer Oberfläche (für jeden Vertex) anhand seiner Oberflächennormale, den angegebenen Lichtquellen und der Betrachterposition berechnet wird. In die Berechnung fließt kein reflektiertes Licht anderer Oberflächen der 3D-Szene ein. Die Auswertung des Beleuchtungsmodells erfolgt meist pro Vertex, und die berechnete Beleuchtung durch die Lichtquellen wird über ein Dreieck linear interpoliert. Das *Per-Pixel Lighting* berechnet die Beleuchtung für jedes Fragment (Pixel), das von der Grafikkarte gezeichnet wird. Die Berechnung findet dabei entweder mit Texturierungs-Einheiten oder in Pixel Shader bzw. Fragment Programm bezeichneten Teilen der Grafik-Pipeline statt. Der Aufwand beim Rendering für Effekte wie Bump Mapping ist deutlich höher. Solche Techniken werden Sie in den folgenden Direct3D-Teilen von PC Underground kennen lernen.

Zunächst widmen Sie sich den grundlegenden Verfahren. Wie erwähnt, benötigen Ihre Vertices für die Beleuchtungsberechnung einen weiteren Parameter: die Oberflächennormale. Diese erhalten Sie entweder vom Modeling Programm, bestimmen Sie bei parametrischen Flächen aus der Beschreibung oder Sie berechnen sie aus einem Dreiecksnetz. Die dazugehörige Flexible Vertex Format Konstante lautet *D3DFVF_NORMAL*. Um die Beleuchtungsberechnung durchzuführen, müssen Sie die Materialeigenschaften der Oberfläche angeben. Diese sind in der Struktur *D3DMATERIAL9* zusammengefasst, die je einen Farbwert für ambiente, diffuse und spekulare Reflexion und Lichtemission und einen Float Wert für das Phong Modell (Halfway Vektor Variante – siehe SDK) enthält. Die Farbwerte sind vom Typ *D3DCOLORVALUE*, der als Struktur aus vier Float-Werten definiert ist. Diese Struktur füllen Sie mit den gewünschten Werten aus. Die Methode *IDirect3DDevice9::SetMaterial* bestimmt die Materialeigenschaften. Folgendes Beispiel erzeugt ein Material mit rein diffusen Eigenschaften:

```
D3DMATERIAL9 mat;

ZeroMemory(
  &mat, sizeof( D3DMATERIAL9 ) );
mat.Diffuse.r =
mat.Diffuse.g =
mat.Diffuse.b =
mat.Diffuse.a = 1.0f;
pD3DDevice->SetMaterial( &mat );
```

Für die Beleuchtung müssen Sie noch sorgen. Dafür stehen Ihnen drei Typen von Lichtquellen zur Verfügung. Der einfachste Typ ist das *Directional Light* (D3DLIGHT_DIRECTIONAL). Das ist Licht mit parallelen Strahlen – vergleichbar mit einer unendlich weit entfernten Punkt-Lichtquelle. Für solche Lichtquellen ist die Beleuchtungsberechnung einfach, weil Sie keine Lichtquellen-Position berücksichtigen müssen. Beim zweiten Typ handelt es sich um die Punkt-Lichtquelle (*D3DLIGHT_POINT*). Diese Lichtquelle leuchtet von ihrer angegebenen Position radial in alle Richtungen. Die komplizierteste Lichtquelle in Direct3D ist ein Spot Light (*D3DLIGHT_SPOT*), die wie ein Scheinwerfer strahlt.

Die Abstrahlung ist vergleichbar mit der Punkt-Lichtquelle, ist aber auf einen kegelförmigen Bereich beschränkt, den Sie bestimmen können. Alle Lichtquellen-Parameter und -Definitionen fasst die *D3DLIGHT9*-Struktur zusammen. Je nach Lichtquellen-Typ benötigen Sie nur einen Teil oder alle Einträge dieser Struktur. Die Parameter umfassen die Farbe der Lichtquelle für ambientes, diffuses und spekulares Licht, die Position und Richtung, Parameter für den Spot Light Kegel und den maximalen Einflussbereich (ein Abstandswert).

Weiterhin ist die Abschwächung (Attenuation) des Lichts durch einen konstanten, linearen und quadratischen Koeffizienten (Attenuation0..2) einstellbar. Das bedeutet, die Lichtintensität wird in Abhängigkeit vom Abstand *d* eines Vertices zur Lichtquelle mit folgendem Wert multipliziert:

```
att = 1.0 / ( Attenuation0 +
              Attenuation1 * d +
             Attenuation2 * d² )
```

Nach dem Ausfüllen einer *D3DLIGHT9*-Struktur übergeben Sie diese an Direct3D mit der Methode *IDirect3DDevice9::SetLight(...)*. Dabei ist der erste Parameter ein Null basierter Index, mit dem die Lichtquelle referenziert wird. Als zweiten Parameter übergeben Sie die Adresse der *D3DLIGHT9*-Struktur.

Neu definierte Lichtquellen sind zunächst noch ausgeschaltet. Sie aktivieren Sie mit *LightEnable(...)*: der erste Parameter ist der Lichtquellen-Index, der zweite ein Bool-Wert, der angibt, ob die Lichtquelle an- oder ausgeschaltet ist. Die Anzahl der maximal gleichzeitig aktivierbaren Lichtquellen entnehmen Sie – wie alle anderen Device spezifischen Fähigkeiten – den Caps. Der Eintrag lautet *D3DCAPS9.MaxActiveLights*. Um die Parameter für eine bestehende Lichtquelle auszulesen verwenden Sie *GetLight(...)*. Um festzustellen, ob eine Lichtquelle an oder aus ist, steht Ihnen die *GetLightEnable(...)*-Methode zur Verfügung.

Folgendes Beispiel zeigt, wie Sie ein *Direction Light* definieren und anschalten:

```
D3DLIGHT9 light;
ZeroMemory(
  &light, sizeof( D3DLIGHT9 ) );
light.Type = D3DLIGHT_DIRECTIONAL;
light.Direction =
  D3DXVECTOR3( 0.0f, 0.0f, 1.0f );
light.Diffuse.r = 1.0f;
light.Diffuse.g = 1.0f;
light.Diffuse.b = 1.0f;
pD3DDevice->SetLight( 0, &light );
pD3DDevice->LightEnable( 0, TRUE );
```

Als letztes aktivieren Sie noch die Beleuchtungsberechnung (allgemein) für Direct3D:

```
pD3DDevice->SetRenderState(
            D3DRS_LIGHTING, FALSE );
```

Die spekulare Beleuchtung müssen Sie separat anschalten:

```
pD3DDevice->SetRenderState(
        D3DRS_SPECULARENABLE, TRUE );
```

: et