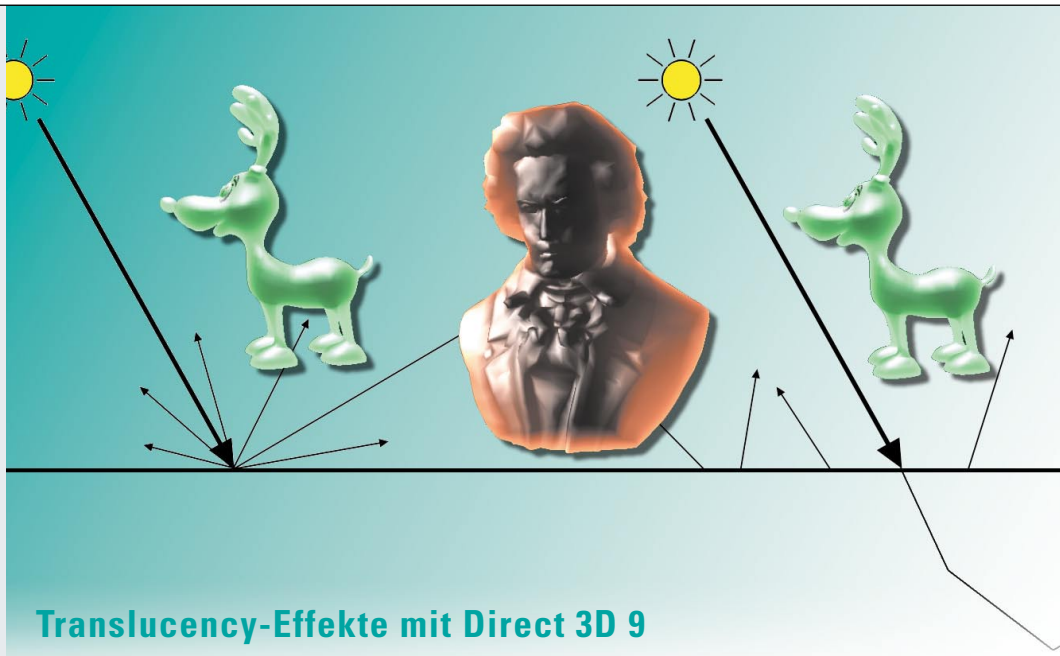


Translucency-Effekte mit Direct 3D 9

Von Schein und Sein

Mit Direct3D und den richtigen Vertex und Pixel Shaders erzielen Sie Transparenzeffekte. Mit Multi-Pass Rendering rechnen Sie bis auf Pixel-Ebene genau.

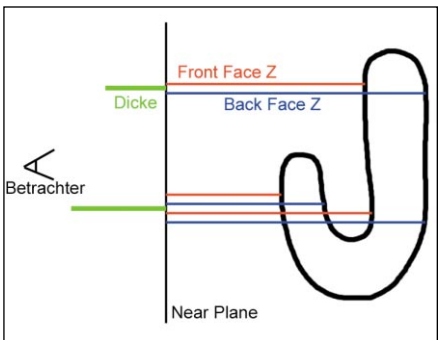
Carsten Dachsbacher

Translucency: **Beethoven stellen Sie eindrucksvoll als durchscheinendes Objekt in Echtzeit dar.**

Nach den letzten Ausgaben der PC-Underground-Reihe verfügen Sie mit den Grundlagen der Direct3D-Programmierung nun über alle Werkzeuge, um eindrucksvolle Grafiken zu gestalten. Diese Ausgabe stellt Ihnen die Techniken vor, die Sie immer wieder für Spezialeffekte benötigen. Damit implementieren Sie eine Pixel genaue Darstellung von transparenten 3D-Modellen.

Das Aussehen vieler realer Materialien wie Marmor, Milch oder menschliche Haut hängt nicht nur von dem an der Oberfläche reflektierten Licht ab. Ein Teil des Lichts dringt an einem bestimmten Punkt in das Material ein, wird dort mehrfach gestreut und reflektiert und kann das Material an einer anderen Stelle wieder verlassen. Diese Prozesse werden unter der Bezeichnung *Subsurface Scattering* zusammengefasst. Solche Materialeigenschaften rendern Sie nicht allein mit einem lokalen Beleuchtungsmodell, mit dem Programmierer in der Echtzeit-Computergrafik kämpfen. An dieser Technik wird gegenwärtig intensiv geforscht.

Wenn Sie lediglich optische Spezialeffekte gestalten wollen, können Sie durch eine starke Vereinfachung des Sachverhalts in Echtzeit durchscheinende (translucent) Objekte darstellen.



Zurückgelegte Strecke: Mit der Grafikhardware können Sie die Dicke des Materials bestimmen.

Ein einfaches Modell

Dabei gehen Sie zunächst davon aus, dass eine unendliche Flächenlichtquelle das Objekt von hinten beleuchtet. Nehmen Sie nun an, dass das aus parallelen Strahlen bestehende eindringende Licht nicht gestreut oder reflektiert, sondern lediglich absorbiert wird. Die Lichtmenge, die durch das Objekt hindurchscheint, lässt sich dann durch eine exponenzielle Funktion beschreiben. Deren Parameter gibt die Strecke vor, die das Licht durch das Material zurückgelegt hat. Für den Rest der 3D-Welt nehmen Sie ein Vakuum an, welches die Lichtstrahlen nicht beeinflusst.

Für diese Berechnung – im Detail später – müssen Sie also für jeden Lichtstrahl, der je-

weils einem Pixel im Bild entspricht, die Strecke durch das 3D-Objekt bestimmen. Dieses können Sie analytisch per CPU rechnen lassen, was allerdings keine Darstellung in Echtzeit ergibt. Stattdessen zweckentfremden Sie dafür Ihre Grafikhardware. Das Prinzip zeigt das Bild (links): Die Strecke, die ein Lichtstrahl durch das Material zurücklegt, entspricht der Differenz des Abstands der Vorder- (*fi*) bzw. Rückseiten (*bi*) des Objektes vom Betrachter. Wenn der Strahl mehrfach in das Objekt eindringt und es wieder verlässt, müssen Sie die Summe der Differenzen berechnen:

$$(b_1-f_1) + (b_2-f_2) + \ldots$$

Durch eine andere Klammerung dieser Summe erhalten Sie:

$$\Sigma\, b_i - \Sigma\, f_i$$

Das bedeutet, Sie können zuerst die Tiefenwerte aller Rückseiten (Back Faces) bzw. Vorderseiten (Front Faces) summieren und anschließend die Differenz bilden. Dieser Ansatz ist schon durchaus tauglich für eine Umsetzung mittels Grafikhardware. Es bleibt lediglich zu klären, wie Sie die Tiefenwerte aufsummieren. Deren Berechnung können Sie leicht in einem Vertex Shader erledigen. Das Problem ist das Aufsummieren selbst: Ein Framebuffer hat eine Genauigkeit von 8 Bit pro Farbkomponente. Bei der Akkumulation mehrerer Werte und anschließender Differenzbildung bleibt Ihnen eine Genauigkeit von wenigen Bits für die Repräsentation der zurückgelegten Strecke. Somit ist dieser Ansatz nicht zu gebrauchen.

Auch die Framebuffers mit Floating-Point-Genauigkeit, die von den neuesten Grafikkarten unterstützt werden, können Sie nicht verwenden. Der Grund ist, dass Sie Werte aufsummieren wollen, was zunächst nichts anderes ist als additives Blending. Leider unterstützt bislang keine Grafikkarte solche Operationen bei Floating Point Rendertargets. Aber Sie können sich mit einem Trick weiterhelfen, der es Ihnen gestattet, einen Wert mit hoher Genauigkeit in einem herkömmlichen RGBA-32-Bit-Rendertarget zu speichern. Dabei wird ein Float-Wert *D* (aus dem Intervall *[0;1]*) auf alle vier Komponenten *R, G, B, A* verteilt:

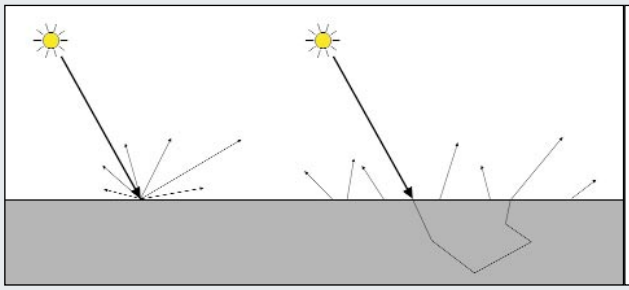
```
R = frac( D * 1.0 )
G = frac( D * 2^L)
B = frac( D * 2^2L )
A = frac( D * 2^3L )
```

Die Funktion *frac(..)* liefert dabei jeweils den Nachkomma-Anteil des Arguments zurück. Die Konstante *L* bestimmt, wie sich die Floating-Point-Zahl auf die vier Komponenten aufteilt. Jede dieser vier Komponenten wird für die Verwendung im Framebuffer auf 8 Bit quantisiert. Eben wegen dieses herkömmlichen Framebuffers können Sie mit einer derartigen Repräsentation alle Blending-Modi der Grafikkarte nutzen. Wenn Sie für *L* den Wert 8 verwenden, schöpfen Sie die Genauigkeit des Framebuffers voll aus. Wollen Sie allerdings mehrere solcher kodierten Werte aufsummieren, müssen Sie kleinere Werte wählen. Sie können dann genau $2^{(8-L)}$ Werte aufsummieren. Diese Werte können Sie etwa mit einem Pixel Shader in nur einem Befehl dekodieren! Bei der benötigten Operation handelt es sich um ein einfaches Skalarprodukt, wobei jede Komponente mit einer Konstanten multipliziert und anschließend aufsummiert wird:

$$D = R * 1.0 + G * 2^{-L} + B * 2^{-2L} + A * 2^{-3L} = (R,G,B,A)^T \text{ dot } (1.0,\ 2^{-L},\ 2^{-2L}, 2^{-3L})^T$$

Diese Operationen müssen Sie allerdings jeweils mit Floating-Point-Genauigkeit ausfüh-

Subsurface Scattering



Unterschied: Die *BRDF*s (links) können nur lokale Beleuchtungseffekte beschreiben. Die *BSSRDF* (rechts) erfassen den Lichttransport durch das Material.

Mit dem Rendering von Subsurface Scattering, also von allen Lichtreflexionsprozessen innerhalb eines Körpers, in Echtzeit bzw. mit interaktiven Frame Rates beschäftigt sich derzeit auch die Forschung. Ein kurzer Einblick in die Theorie: Ein allgemeines Modell für die lokale Beleuchtungsberechnung sind die so genannten Bidirectional Reflectance Distribution Functions (BRDFs). Diese Funktionen bestimmen für einen Oberflächenpunkt, den Teil des auftreffenden Lichtes, der aus einer in eine andere Richtung reflektiert wird. Es handelt sich dabei also um vierdimensionale Funktionen – zwei Dimensionen pro Richtung. Diese Funktionen können natürlich nicht den Lichttransport durch das Material modellieren. Deshalb gibt es eine weiter verallgemeinerte Klasse von Funktionen, die *Bidirectional Surface Scattering Reflectance Distribution Function* (BSSRDF). Sie beschreiben den Lichttransport zwischen zwei Lichtstrahlen, die auf der Oberfläche auftreffen oder abgehen, und sind daher achtdimensional. Aufgrund der Komplexität sind diese Funktionen nicht für Echtzeit-Rendering zu gebrauchen, können aber durch geeignete Verfahren und bestimmte Materialen gut approximiert werden.

Das Prinzip dahinter: Die BSSRDF können Sie für stark lichtstreuende Materialtypen sehr wirklichkeitsgetreu gestalten, indem nur Mehrfachstreuung angenommen wird. Strahlen, die nur einmal gestreut wurden und das Material gleich wieder verlassen, sind dabei selten.

- In einem ersten Abschnitt wird das eintreffende Licht auf der Oberfläche berechnet, etwa an vielen Punkten, die zufällig auf der Oberfläche verteilt sind. Die Menge des Lichtes, das jeweils reflektiert wird bzw. in das Material eindringt, hängt vom so genannten *Fresnel Term* ab.
- Im zweiten Abschnitt wird anschließend der Lichttransport durch das Material berechnet. Für jeden Oberflächenpunkt bestimmen Sie die Menge des Lichts, das von allen anderen Oberflächenpunkten zu einem Punkt dringt. Dieser Term lässt sich berechnen: Er hängt von der Transportrichtung und -strecke des Lichts und der Normalen am Eintrittspunkt eines Lichtstrahls ab. Doch selbst mit diesem Verfahren benötigen Sie noch einige Sekunden pro Bild! Doch mit moderner Grafikhardware sind interaktive Frame Rates zu erreichen, wie die Literatur angibt.

www.dachsbacher.de/pcu
www.ati.com
www.nvidia.com
http://graphics.stanford.edu/papers/bssrdf/
http://www9.informatik.uni-erlangen.de/Research/Rendering/TSM

ren, benötigen aber dafür keine Grafikkarte der neuesten Generation. Es genügt bereits eine GeForce-3-Karte, um diese Effekte darzustellen.

Rendertargets anlegen

Für diesen Spezialeffekt benötigen Sie mehrere Renderpasses, deren Resultate erst später mit dem endgültigen Bild kombiniert werden. Mit Direct3D ist es einfach, in eine Textur statt in den Framebuffer zu zeichnen. Sie müssen das lediglich beim Anlegen der Textur berücksichtigen. Das erledigen Sie während der Initialisierung. Beachten Sie dabei, dass eine Textur, die als Rendertarget verwendet wird, auch einen eigenen Z-Buffer besitzt:

```
LPDIRECT3DTEXTURE9 pDynamicTexture;
LPD3DXRENDERTOSURFACE pRenderSurface;
LPDIRECT3DSURFACE9   pTextureSurface;

// dynamische Textur erzeugen
D3DXCreateTexture (....);

// Off-Screen Surface anlegen
D3DSURFACE_DESC desc;
pDynamicTexture->GetSurfaceLevel(
  0, &pTextureSurface );
pTextureSurface->GetDesc( &desc );
D3DXCreateRenderToSurface( ....);
```

Wenn Sie nun diese Textur als Rendertarget statt des normalen Framebuffers verwenden wollen, modifizieren Sie die *BeginScene/EndScene*-Aufrufe:

```
pRenderSurface->BeginScene(
  pTextureSurface, NULL );
pD3DDevice->Clear( ... );
renderScene();
pRenderSurface->EndScene( 0 );
```

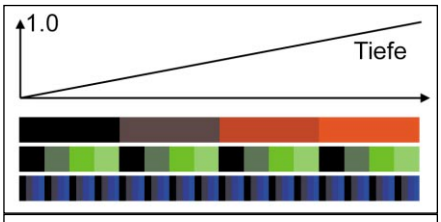


Abbildung: So wird die Tiefe in den RGB-Werten kodiert.

Verlauf: Mit dieser Textur werden die Farbwerte den Strecken durch das Material zugewiesen.

Den Inhalt der Textur können Sie wie jede andere für ein anderes Rendertarget mit

```
pD3DDevice->SetTexture
( 0, pDynamicTexture )
```

verwenden.

Codierung und Akkumulation der Tiefenwerte

Im zweiten Schritt müssen Sie die Tiefenwerte jeweils für die Front und Back Faces getrennt aufsummieren. Zunächst müssen Sie diese aber für jeden Pixel berechnen. Die Berechnung beginnt in einem Vertex Shader. Dieser transformiert natürlich die Vertex Koordinaten entsprechend der *konkatenierten* Projektion und *World Matrix* (gespeichert in den Konstanten *c0* bis *c3*). Als Tiefenwert verwenden Sie die vierte Komponente (homogene Koordinaten) dieses Resultats, die Sie noch abhängig von den *Near* und *Far Planes* auf das Intervall *[0;1]* abbilden:

```
vs.1.0
...
; Tiefenwert berechnen
dp4   r0.w, v0, c3

; Abbilden auf [0;1]
mad r0, r0.w, c10.xxxz, c10.yyyw
```

Die Konstante *c10* enthält dabei:

```
c10.x = 1.0 / (zFar – zNear)
c10.y = -zNear / (zFar – zNear)
c10.z = 0.0
c10.w = 1.0
```

Die Kodierung des Tiefenwerts muss nun in der Fragment-Stufe über die Bühne gehen. Wenn Sie sich auf eine Kodierung in drei der Farbkomponenten beschränken, was immer noch eine genaue Darstellung ist, können Sie das durch eine 3D-Textur erledigen. Dazu berechnen Sie im Vertex Shader lediglich die Textur-Koordinaten:

```
mul oT0, r0, c20
```

Wobei die Konstante *c20* den Vektor *(1.0, 2L, 22L, 0.0)T* enthält. Unser Beispielprogramm wählt für *L* den Wert *4*, womit Sie mindestens 16 Tiefenwerte fehlerfrei akkumulieren können. Die 3D-Textur muss also *16x16x16* Werte enthalten, wobei Sie den Farbwert eines Texels (32 Bit ARGB) bestimmen. Der Wertebereich pro Komponente liegt von *0-255*:

```
// pRampTexture
for ( z = 0..16 )
 for ( y = 0..16 )
  for ( x = 0..16 )
   tex3d( x, y, z ) =
  (x<<16) | (y<<8) | x;
```

Die Abbildung(links unten) verdeutlicht, wie sich Tiefenwerte auf den RGB-Tripel darstellen. Gehen Sie beim Rendering (pro Frame) zunächst wie folgt vor: Im Rendertarget *#1* akkumulieren Sie die Tiefenwerte der Back Faces. Dazu schalten Sie den Z-Buffer-Test aus, denn jede Fläche soll gezeichnet werden:

```
pD3DDevice->SetRenderState(
  D3DRS_ZENABLE, false );
pD3DDevice->SetRenderState(
  D3DRS_ZWRITEENABLE, false );
```

Das Akkumulieren erledigen Sie mittels additivem Blending:

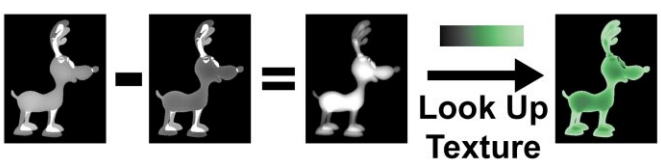
```
pD3DDevice->SetRenderState(
  D3DRS_ALPHABLENDENABLE, true );
pD3DDevice->SetRenderState(
  D3DRS_SRCBLEND,  D3DBLEND_ONE );
pD3DDevice->SetRenderState(
  D3DRS_DESTBLEND, D3DBLEND_ONE );
pD3DDevice->SetRenderState(
  D3DRS_BLENDOP,   D3DBLENDOP_ADD );
```

Anschließend aktivieren Sie den obigen Vertex Shader, setzen die Abbildungsmatrizen und zeichnen die Back Faces der durchscheinenden Objekte:

```
pD3DDevice->SetTexture( ...);
pD3DDevice->SetSamplerState(...);
pD3DDevice->SetSamplerState( ...);

// Render Back Faces
pD3DDevice->SetRenderState(...);
obj->drawModel( pD3DDevice );
```

Im zweiten Rendertarget gehen Sie analog vor, abgesehen davon, dass Sie mit *D3DCULL_CCW* die Front Faces zeichnen.



Farbberechnung: Tiefenwerte für Vorder- und Rückseiten werden aufsummiert, die Differenz ist die Farbe.

Differenz und Dekodieren

Für die bisherigen Operationen brauchen Sie keine Unterscheidung, ob Sie eine neue oder ältere Grafikkarte programmieren. Die folgenden Teilaufgaben lassen sich mit einer Grafik-

karte, die Pixel Shader der Version 2 unterstützt, einfacher ausführen. Deshalb betrachten Sie zunächst diese Variante.
Zunächst lesen und subtrahieren Sie die kodierten Tiefenwerte der beiden Rendertargets. Nach der Dekodierung per Skalarprodukt wandeln Sie die resultierende Tiefendifferenz in einen Farbwert um. Dazu verwenden Sie eine Textur mit einem entsprechenden Farbverlauf, den Sie dann auslesen.
Sie beginnen, ein Rechteck über den gesamten sichtbaren Bereich zu zeichnen. Auf dieses Rechteck bilden Sie die obigen Rendertargets als Textur *#0* und *#1* ab. Auf die dritte Textur-Stage legen Sie die Textur mit dem Farbverlauf, die Textur-Stage *#2* halten Sie für einen zusätzlichen Effekt zunächst frei. Für das Zeichnen dieses Rechtecks verwenden Sie den folgenden Pixel Shader (Version 2.0), bei dem die Deklaration der Textur-Stages entfällt.
Als Erstes lesen Sie die Texturen mit den Tiefenwerten aus, subtrahieren diese und berechnen das Skalarprodukt zum Dekodieren:

```
ps.2.0
...
texld    r8, t0, s0
texld    r9, t1, s1

sub      r0, r8, r9
dp4_sat  r0, r0, c20
```

Die Konstante *c20* enthält den bereits erwähnten Vektor *(1.0, 2-L, 2-2L, 0.0)T*, den Sie aber mit einer beliebigen Konstante multiplizieren können, um eine variable Dichte des Materials zu modellieren.
Mit der berechneten Tiefendifferenz greifen Sie auf die Textur mit dem Farbverlauf zu und verwenden das Resultat als endgültigen Farbwert:

```
texld    r0, r0, s3
mov      oC0, r0
```

Der oben erwähnte Spezialeffekt ist eine zusätzliche lokale Beleuchtungsberechnung, um etwa Glanzlichter auf den durchscheinenden Objekten darzustellen. Sehen Sie, wie zwei Bestandteile das endgültige Bild ergeben.
Diese lokale Beleuchtungsberechnung rendern Sie, bevor Sie das Bild endgültig zeichnen, in ein weiteres Rendertarget.
Diese Textur können Sie analog auslesen und einfach zu dem Farbwert addieren:

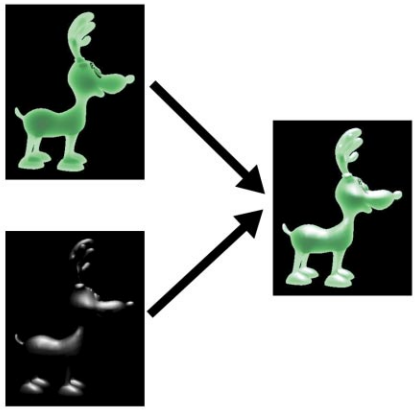
```
texld r10, t2, s2
add r0, r0, r10
mov oC0, r0
```

Etwas komplizierter fällt diese Berechnung für ältere Grafikkarten aus. Denn dort besteht das Problem, dass Sie nicht in einem Renderpass Texturkoordinaten beliebig berechnen (in diesem Fall die dekodierte Tiefe) und gleich damit auf eine Textur zugreifen können. Deshalb müssen Sie die Auswertung in zwei Renderpasses aufteilen.
Im ersten Pass führen Sie lediglich die Subtraktion durch. Dazu legen Sie die beiden Rendertargets mit den akkumulierten Tiefenwerten auf die ersten beiden Textur-Stages und zeichnen wieder ein Rechteck über den Bildschirm. Dabei verwenden Sie den folgenden Pixel Shader, um die Werte zu subtrahieren und anschließend *0.5* auf jede Komponente zu addieren (Konstante *c3*):

```
ps.1.0...
tex  t0  tex  t1
add  r0, t1, -t0  add  r0, r0, c3
```

In diesem Renderpass zeichnen Sie noch nicht in den später sichtbaren Back Buffer, sondern nochmals in ein extra Rendertarget.
Diese verwenden Sie nun im endgültig letzten Renderpass als Textur auf Stage *#0*.
Der Pixel Shader für das Dekodieren sieht wie folgt aus:

```
ps.1.0
tex  t0
```



Optische Aufbesserung: Die zusätzliche lokale Beleuchtung fügen Sie durch Addition der Farbwerte hinzu.

```
; Skalarprodukt: t1 dot (2*(t0-0.5))
texm3x2pad  t1, t0_bx2
texm3x2tex  t2, t0_bx2
tex  t3 ; lokale Beleuchtung auslesen
add  r0, t2, t3 ; Finaler Farbwert
```

Die zwei Textur-Operationen zu Beginn des Shaders müssen zusammen auftreten, können also nicht einzeln verwendet werden.
Sie berechnen das Skalarprodukt von dem ausgelesenen Farbwert der Textur *#0* (Register *t0*) und dem Wert in der Textur Koordinate *t1*. Der Register Modifier *_bx2* bewirkt, dass von den RGBA-Werten aus der Textur *#0* zunächst *0.5* abgezogen und anschließend mit *2.0* multipliziert wird. Vielleicht fragen Sie sich, was es nun mit dieser Befehlskonstellation auf sich hat?
Wenn Sie einen kleinen zusätzlichen Vertex Shader für diesen Renderpass verwenden, der in die Textur-Koordinate *t1* jeweils den Vektor *(1.0, 2-L, 2-2L, 0.0)T* schreibt, führen Sie so das Dekodieren der Tiefe und das Auslesen der Farbverlauf-Textur (auf Stage *#2*) aus. Der Rest des Pixel Shaders liest lediglich wieder die lokale Beleuchtung aus und addiert diese zu dem vorher bestimmten Farbwert. : et