Deferred Shading – Beleuchtung als Post Processing

Schattenspiel

Mit *Deferred-shading*-Techniken zeichnen Sie Ihre 3D-Szenen zunächst völlig ohne Beleuchtungsberechnung. Diese übernimmt einmalig ein finaler Nachbearbeitungsschritt für jeden sichtbaren Pixel.

Carsten Dachsbacher

Das *Deferred Shading* konstruiert die Geometrie einer 3D-Szene zunächst ohne Beleuchtungsberechnung. Dabei zeichnen Sie nicht in den normalen sichtbaren Framebuffer, sondern in so genannte Fat-Buffers.
Der Name kommt daher, dass diese Buffer, verteilt auf mehrere gleichzeitig beschreibbare Rendertargets, verhältnismäßig viel Daten pro Pixel enthalten, wie z.B. die 3D-Position und die Normale der – in diesem Pixel – sichtbaren Oberfläche. Nach dem Zeichnen der Szene wird die Beleuchtungsberechnung für jeden Pixel durchgeführt: die dafür notwendige Information befindet sich in den Fat-Buffers. Für diese Technik benötigen Sie natürlich modernere, programmierbare Grafikkarten der DirectX9-Generation, um zum Einen die Rendertargets (dynamische Texturen) zu beschreiben und zum Anderen die Beleuchtungsberechnung in einem Pixel-Shader zu programmieren.

Deferred Shading: Sie berechnen die Beleuchtung nur einmal pro Pixel.

Vergleich

Unter den vielen Varianten Lighting/Shading für das Echtzeit-Rendering, greifen wir an dieser Stelle drei Varianten heraus, die mit dynamischen Lichtquellen und lokaler Beleuchtungsberechnung arbeiten. Das Single-Pass-Verfahren berechnet das Lighting direkt beim Rendern der Geometrie. Dieser Ansatz ist gut geeignet, um Szenen mit wenigen Lichtquellen darzustellen. Bei einer großen Anzahl von Lichtquellen wird die Organisation der Shader und der Lichtquellen, die für ein Objekt relevant sind, schwierig und der Vertex/Pixel Shader leicht zu komplex. Beim Multi Pass Lighting wird jeweils nur eine Lichtquelle auf ein Objekt angewendet und dieses gegebenenfalls mehrfach gezeichnet und in den Framebuffer geblendet. Das Problem hierbei ist der entstehende Aufwand bei der Verwaltung und dem Rendering von Lichtquellen und Objekten.
Beim Deferred Shading müssen Sie sich um die Zahl der endgültig angewendeten Lichtquellen beim Zeichnen der Objekte keine Gedanken machen. Für die Performance ist es auch nahezu egal, ob Sie viele klein- oder wenige großflächige Lichtquellen in Ihrer Szene verwenden.

Multiple Render Targets

In den Rendertargets, also dem Ergebnis des Geometrie-Renderings, benötigen Sie neben der 3D-Position jedes Pixels und seiner Normale noch Materialparameter. Diese können je nach verwendetem Beleuchtungsmodell variieren. Typischerweise umfassen die Parameter die diffuse Oberflächenfarbe, spekulare Reflektion und eventuell auch Parameter für Lichtemission und Subsurface-Scattering.
Prinzipiell sollten Sie die Datenmenge aber so gering wie möglich halten, wie das folgende Beispiel zeigt. Nehmen Sie an, Sie speichern die Position in einem *A32R32G32B32* Rendertarget (32 Bit IEEE Float für alle vier Komponenten), die Normale, diffuse Farbe und zusätzliche Materialparameter jeweils als *A8R8G8B8*-Rendertarget. Somit würden Sie pro Pixel bereits 224 Bits speichern, was sich bei einer Auflösung von 1024x768 auf 21 Megabyte summieren würde, ohne das Sie Anti-Aliasing verwenden könnten. Ein dabei verschwiegenes Problem ist, dass die momentane Grafikhardware es gar nicht erlaubt, unterschiedliche Bit-Tiefen bei multiplen Rendertargets zu verwenden.
In unserem Beispielprogramm verwenden Sie die folgende Konfiguration, wobei wir uns auf 32-Bit-Rendertargets beschränken wollen. Um trotzdem eine genügend hohe Genauigkeit zu erzielen, teilen Sie die 3D-Position auf zwei Rendertargets mit je zwei 16-Bit-Float-Werten auf (D3DFMT_G16R16F). Die Normale speichern Sie entweder in einem *A8R8G8B8* Target, d.h. mit drei 8-Bit-Komponenten und einem noch unbelegten Byte für weitere Daten oder, wenn Sie noch mehr Genauigkeit wünschen, in einem *A2R10G10B10* Rendertarget, also mit 10 Bit pro Komponente. Die Materialparameter beschränken sich in unserem Beispiel auf eine diffuse Farbe, die Sie in ein *A8R8G8B8* Target schreiben.

Implementation

Unser Beispielprogramm verwendet Direct3D9 und basiert auf dem Framework, das Sie vielleicht schon aus früheren Ausgaben kennen. Den vollständigen Quelltext finden Sie wie immer auf der Heft CD. Die Beschreibung hier konzentriert sich deshalb auf die relevanten Teile für die Deferred Shading Konzepte.
Die Rendertargets legen Sie mit der *D3DXCreateTexture*-Methode an. Wichtig ist, dass Sie bei dem Verwendungszweck der Textur (Usage-Flag) *D3DUSAGE_RENDERTARGET* angeben und das entsprechende Pixelformat wählen. Mit der *GetSurfaceLevel*-Methode des *IDIRECT3DTEXTURE9*-Interfaces (also Ihres Textur-Objektes) holen Sie sich einen Zeiger auf die erste Surface Ihrer Rendertarget-Textur.
In dem initialen Renderpass beschreiben Sie also die Rendertargets, deren Verwendung Sie Direct3D zunächst mitteilen müssen. Vorher holen Sie die Referenz auf den Backbuffer ein, auf den das später sichtbare Bild gerendert wird:

```
LPDIRECT3DSURFACE9 lpBackBuffer;
```

Anschließend können Sie schon beginnen, die Geometrie zu rendern. Um die multiplen Rendertargets beschreiben zu können, benötigen Sie einen Vertex und Pixel Shader, den das Beispielprogramm mit der Microsoft High Level Shader Language und einem Effect File definiert:

```
pEffect->SetTechnique( „InitialPass“ );
renderScene();
```

Der Vertex Shader übernimmt dabei die herkömmliche Transformation der Vertices für die Rasterisierung (matMVP Matrix) und die Transformation der Koordinaten in den World Space (matMV), um später die Beleuchtung zu berechnen. Diese werden – genauso, wie die Normale und die Textur-Koordinaten (für normales Textur-Mapping) – in den Textur-Koordinaten-Registern an die Rasterisierungseinheit übergeben:

```
FRAGMENT vsInitialPass( VERTEX vertex )
...  return f;
}
```

Der Pixel Shader nimmt diese Informationen, vom Rasterisierer für jeden Pixel interpoliert, entgegen, erledigt das normale Textur-Mapping und kodiert und verteilt die Information auf die Rendertargets.

```
struct FRAGRESULT
{
 float4 color[4] : COLOR;
};

FRAGRESULT psInitialPass( FRAGMENT
fragment )
{
  FRAGRESULT f;
```

Mit diesen Shadern rendern Sie Ihre komplette Geometrie. Für den zweiten und letzten Renderpass setzen Sie als Rendertarget wieder den ursprünglichen Backbuffer:

```
pD3DDevice->SetRenderTarget(1,NULL);
pD3DDevice->SetRenderTarget(2,NULL);
pD3DDevice->SetRenderTarget(3,NULL);
pD3DDevice-
>SetRenderTarget(0,lpBackBuffer);
```

Als Beispiel beleuchten Sie jetzt die Szene mit einer Lichtquelle. Dazu rendern Sie ein bildschirmfüllendes Rechteck, auf das die Rendertargets als Textur gespannt sind. Dazu verwenden Sie folgenden Code, wobei die Abbildungsmatrizen die Identitätsabbildung enthalten:

```
typedef struct
{
 float x, y, z, u, v;
}TEXTUREDVERTEX;

TEXTUREDVERTEX screenQuad[] =
{
 { -1, -1, 0, 0, 1 },
 { -1,  1, 0, 0, 0 },
 {  1, -1, 0, 1, 1 },
 {  1,  1, 0, 1, 0 },
};

pD3DDevice->SetFVF(
D3DFVF_XYZ|D3DFVF_TEX1 );
pD3DDevice->SetRenderState(
D3DRS_CULLMODE, D3DCULL_CCW );
pD3DDevice->DrawPrimitiveUP(
D3DPT_TRIANGLESTRIP, 2, screenQuad,
sizeof( TEXTUREDVERTEX ) );
```

Die Beleuchtungsberechnung übernimmt der folgende Pixel Shader, der ebenfalls im *Effect*-File definiert ist.

```
struct FRAGMENT_DEFERRED
```

World Space: Der genaue Ort der Oberfläche liegt im Raum.

Die Normale: Sie ist wichtig für die Beleuchtungsberechnung.

Zunächst lesen Sie die vier ehemaligen Rendertargets aus:

```
float4 posXY, posZ, normal, color;
```

Und rekonstruieren die Normale bzw. World Space Position:

```
normal = normalize(normal*2.0-1.0);
 float4 worldSpacePos =
float4(posXY.x,posXY.y,posZ.x,1.0);
```

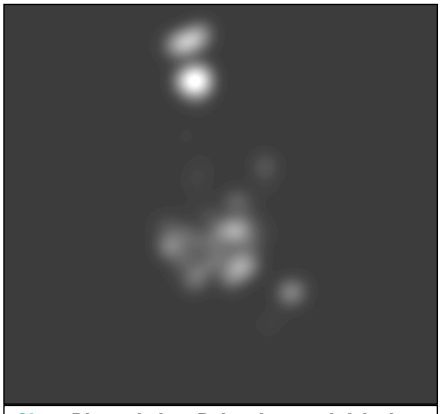
Anschließend führen Sie die Beleuchtungsberechnung aus und modulieren die diffuse Oberflächenfarbe und addieren die spekulare Beleuchtung. So erhalten Sie den endgültigen Farbwert, den Sie in den Framebuffer schreiben:

```
float4 eye, light, reflection, lit;
```

Mehr Licht

Wenn Sie die Szene mit weiteren Lichtquellen beleuchten wollen, müssen Sie den letzten Renderpass einfach nur wiederholen und dabei additives Blending im Framebuffer einstellen. Bei lokalen Lichtquellen, die nur einen Teil der Szene ausleuchten sollen, wie z.B. durch eine entfernungsabhängige Abschwächung, müssen Sie nicht jedes Mal den ganzen Bildschirm füllen. Stattdessen sparen Sie Rendering-Zeit, indem Sie nur den Teil des Bildschirms erneut rendern, der im Einflussgebiet der Lichtquelle liegt. Dazu erzeugen Sie für jede dieser Lichtquellen – als Vorberechnungsschritt – ein einfaches konvexes Dreiecksnetz, das den ausgeleuchteten Raum enthält. Dieses Dreiecksnetz rendern Sie mit dem entsprechenden Pixel Shader für die Beleuchtung. Der von diesem Netz bedeckte Bereich am Bildschirm ist der, den die Lichtquelle potentiell beeinflusst und für den Sie die Beleuchtungsberechnung durchführen müssen. Wichtig ist dabei, dass jeder Pixel nur einmalig behandelt wird. Bei konvexen Dreiecksnetzen können Sie das durch Backface Culling erwirken. Achten Sie dabei darauf, dass Sie nur die Vorderseiten rendern, wenn sich die Kamera außerhalb des Netzes befindet, ansonsten rendern Sie die Rückseiten.

Ein weiteres Problem ergibt sich, wenn das Netz die Near und/oder *Far Clipplane* schneidet. Diese Fälle müssen Sie speziell, z.B. durch *Clamping* des Volumens im Vertex Shader, behandeln. Um das Rendering zu beschleunigen, können Sie für das Zeichnen dieser Light Volumes Z-Buffering verwenden. Die notwendige Information haben Sie durch das Rendern im initialen Pass schon im Tiefenpuffer gespeichert. Je nachdem, ob Sie Vorder- oder Rückseiten zeichnen, verwenden Sie als Z-Buffer Test *D3DCMP_LESS* bzw. *D3DCMP_GREATER*.



Glow: Die spekulare Beleuchtung wird dank HDR übersteuert.

High Dynamic Range: Sie zeigt eine diffuse Beleuchtung und sehr helle spekulare Reflektion.

Frame Buffer Optimierungen

Der hohe Speicherbedarf der Rendertargets kann dazu führen, dass die Grafikhardware durch viel Speichertransfer ausgebremst wird. Um dies zu vermeiden, können Sie die Menge der gespeicherten Information reduzieren, wenn Sie dafür etwas mehr Rechenaufwand in Kauf nehmen. Die Frage, welche der im folgenden vorgestellten Optionen am schnellsten ist, hängt vom jeweiligen Einsatz, Beleuchtungsmodell und Grafikkarte ab und lässt sich im Vornherein nicht beantworten.

Den größten Teil der Daten nimmt das Speichern der World Space Position ein. Dabei ist durch die 2D-Position eines Pixels auf dem Bildschirm und die Kameraparameter ein Sichtstrahl durch jeden Pixel im Raum definiert. Statt der World Space Position speichern Sie die Entfernung zum ersten Oberflächenpunkt, den der Strahl schneidet. Dadurch können Sie die Position im Beleuchtungs-Renderpass berechnen. Diese Entfernung ist dabei nichts anderes als der Tiefenpuffer. Leider können Sie nicht performant auf den Tiefenpuffer der Grafikkarte zugreifen, aber Sie können die Information selbst berechnen und in einem Rendertarget speichern. Wenn Sie dafür einen 32-Bit-IEEE-Float verwenden, haben Sie die Information schon deutlich reduziert: In unserem Beispielprogramm würden Sie ein Rendertarget bzw. 32 Bit pro Pixel sparen.

Die Normale können Sie auch etwas sparsamer kodieren. Bei einer normalisierten Normale ist $x^2+y^2+z^2=1$. Wenn Sie nur zwei Komponenten speichern wie x und y können Sie die dritte im Pixel Shader berechnen: $z = sqrt(1-x^2-y^2)$. Eine dritte Option ist, dass Sie die Materialparameter nicht direkt in den Fat-Buffers speichern, sondern nur einen Index bzw. Verweis. Dieser Index wird im Beleuchtungs-Renderpass dazu verwendet, um die tatsächlichen Materialparameter aus einer Textur auszulesen.

High Dynamic Range (HDR)

Wenn Sie die Renderpasses für die Beleuchtung nicht direkt in den Framebuffer ausführen, sondern in weitere Rendertargets mit Floating-Point-Genauigkeit, können Sie den Wertebereich der erfassbaren Lichtintensität erhöhen. Allerdings müssen Sie sich um das additive Blending selbst bemühen. Das Rendering mit erhöhtem Wertebereich wird mit High Dynamic Range Rendering bezeichnet. Diese Information gilt es natürlich auf den normalen Helligkeitsbereich des Monitors bzw. Framebuffers abzubilden. Allerdings lassen sich Helligkeitsszenarien programmieren. Zudem können Sie Post-Processing-Effekte wie Glow anwenden.

Vor- und Nachteile

Die Vorteile von Deferred Shading ist die einfache Handhabung von sehr komplexen Szenen mit vielen Lichtquellen, komplexen Objekten und Post-Processing Effekten. Außerdem zeichnen Sie jedes Objekt nur einmalig und schattieren auch jeden Pixel nur einmal. Der Nachteil liegt im nicht vernünftig machbaren Alpha Blending, der hohen Speicherbandbreite und darin, dass Sie Hardware Multisampling nicht verwenden können. Und nicht zu vergessen: Sie benötigen Hardware, die Pixel-Shader unterstützt, denn alle Beleuchtungsberechnungen sind darauf angewiesen. : et

www.dachsbacher.de/pcu
www.ati.com/developer/