AUF CD

Die Quelltexte sowie die fertig übersetzten Routinen finden Sie im Verzeichnis *Heft Add-ons/Programmierung/PC Underground.*

High Performance Rendering

Ein Bild wie der Blitz

Die Leistung moderner Grafikkarten ist beeindruckend – wird aber kaum genutzt. Bringen Sie Ihre nVidia- oder ATI-Grafikkarte und OpenGL ans **theoretische Leistungsmaximum**.

CARSTEN DACHSBACHER

Die Leistung moderner Grafikkarten ist in den letzten Jahren rapide gestiegen. Der Trend wird sich voraussichtlich fortsetzen. Mit den neuen Features und zunehmender Leistung steigen auch die Anforderungen an die Programmierer, diese auszunutzen. In dieser Ausgabe lernen Sie, die Performance-Engpässe eines OpenGL-Programms zu identifizieren und der Graphic Processing Unit (GPU) die Geometriedaten optimal bereitzustellen.

Die Grafik-Pipeline

In der Grafik-Pipeline sind schematisch die benötigten Operationen für 3D-Grafik aufgezeigt. Sie lässt sich in drei große Bereiche aufteilen:

- die Geometrieverarbeitung (Geometry Processing), auch als Transform and Lighting bezeichnet, ist für die Koordinaten-Transformation, die Beleuchtungsberechnung und das Clipping zuständig und kann durch Vertex-Shader ersetzt werden.
- In der zweiten Stufe namens Rasterisierung (Rasterization) werden Dreiecke, Linien und Punkte gezeichnet.
- Den letzten Teil stellen die Fragment-Operationen dar, bei denen es sich unter anderem um Alpha-, Stencil- und Z-Buffer-Tests handelt.

Um die Geschwindigkeit eines OpenGL-Programms zu optimieren, mass man die möglichen Schwachstellen kennen. Dazu betrachten Sie den Weg der Grafikdaten durch die Grafik-Pipeline. Gehen Sie zunächst von einem Dreiecksnetz in der Shared-Vertex-Struktur aus: Darin bestehen Ihre Grafikdaten zum einen aus der Geometrieinformation, den Vertices, eventuell mit Normalen, Texturkoordinaten usw. und zum anderen aus einer Indexliste, in der für jedes Dreieck die drei Indizes der Eckpunkte gespeichert sind (Topologie-Information).

Fürs Rendering verarbeiten Sie zunächst die Geometriedaten in der ersten Stufe der Pipeline. Die transformierten, beleuchteten Koordinaten werden zusammen mit der Topologie-Information für das Rasterizer Setup (dem Vorbereiten des Rasterisierens) benötigt. Jetzt werden die Dreiecke gezeichnet, und jeder Pixel durchläuft die letzte Stufe der Pipeline. Statt *Pixel* finden Sie in der Expertenliteratur häufiger die Bezeichnung *Fragment*.

Ein weiterer Engpass liegt in der Übertragung der Geometriedaten zur GPU selbst. Wenn die Geometriedaten sich im Hauptspeicher des Rechners befinden, müssen sie jedes Mal über den AGP-Bus transferiert werden. Angenommen, Sie verwenden pro Vertex Daten mit Koordinate, Normal und zwei Texturkoordinaten-Paaren (jeweils 32-Bit-Float-Werte). Sie benötigen 40 Byte pro Vertex. Selbst bei der theoretischen maximalen Transferleistung, beispielsweise des AGP-4x-Busses mit 1066 MByte/s, reicht diese Transferleistung nicht aus, um die Geometrieverarbeitungsgeschwindigkeit einer GeForce 4 auszunutzen. Sie können $1066*1024^2/40$,

DIE GRAFIK-PIPELINE zeigt die anfallenden Aufgaben beim Rendering.

Die GPUs bieten nur begrenzte Rechenleistung für die Geometrieverarbeitung. Diese hängt von der Taktfrequenz der GPUs und dem Modell der GPU ab. Die Anzahl der Vertices, die verarbeitet werden kann, sinkt auch, wenn komplexe Beleuchtungsberechnungen durchgeführt werden oder die Anzahl der Lichtquellen zunimmt.

Als Anhaltspunkt für die reine Transformationsleistung (ohne Beleuchtung) können Sie für eine GeForce-3-GPU mit 200 MHz etwa 40 Millionen, für eine GeForce-4-Ti4200 etwa 95 Millionen Vertices pro Sekunde ansetzen. Eine ATI Radeon 8500 mit 250 MHz erreicht etwa bis zu 40 Millionen Vertices pro Sekunde.

also etwa 28 Millionen Vertices transferieren (theoritisches Maximum). Da es sich bei diesem Punkt um eine kritische Stelle der Grafik-Pipeline handelt, existieren OpenGL-Erweiterungen, die diesen Flaschenhals umgehen. Damit greifen Sie auf die Geometriedaten direkt im Speicher der Grafikkarte oder in einem Speicherbereich zu, den Sie mittels Direct Memory Access (DMA) Transfer, also an der CPU vorbei, manipulieren können.

Beim letzten wichtigen Punkt handelt es sich um die begrenzte Rasterisierungsleistung, oft als Fill Rate bezeichnet. Diese ist ein Engpass auf Grund begrenzter Speicherbandbreite und GPU -Geschwindigkeit. Diese interne Spei-

cherbandbreite, die bei heutigen General-Purpose-Grafikkarten im Bereich von 7 GByte/s bis 12 GByte/s liegt, ist von Bedeutung, da beim Rendering auf Texturen, Frame und Z-Buffer usw. zugegriffen werden muss.

Die GPU-Geschwindigkeit kommt beim Verarbeiten dieser Informationen zum Tragen. Beispielsweise sind verschiedene Textur-Mapping-Techniken unterschiedlich schnell. Ein- oder zweidimensionale Texturen und Cube Maps sind schnell, die Passthrough- oder Pixel- Kill-Operationen der Texture-Shader (nVidia) bzw. Pixel-Shader sind schon langsamer. Noch aufwändiger sind die Dependent Lookups oder die Dot-Product-Operationen.

Die Geometrie

Wenn Sie wissen, wo sich die Performance-Fallen verbergen, versuchen Sie, diese zu umgehen, bzw in einer bestehenden Implementation zu identifizieren. Zunächst lernen Sie die Methoden kennen, um die Geometriedaten in einem geeigneten Speicherbereich abzulegen und somit eine weit höhere Leistung zu erreichen, als dies mit Compiled Vertex Arrays (CVA) möglich ist. Für das spätere Rendering verwenden Sie im Folgenden wie bei CVAs jeweils Daten-Streams: ganze Arrays von Vertexdaten und -attributen bzw. Indexlisten. Diese können Sie mit dem *glDrawElements*-Befehl von OpenGL rendern.

Wie nicht anders zu erwarten war, haben Sie es, je nach Grafikkartenhersteller, mit unterschiedlichen Extensions zu tun. Hier stellen wir Ihnen beide vor, beginnend mit der nVidia Vertex Array Range Extension (VAR). Diese bietet eine Funktion an, mit der Sie Speicher für Geometriedaten allokieren können, der entweder auf dem Grafikkarten-Speicher liegt oder für die Grafikkarte per DMA-Zugriff erreichbar ist.

Der Name der Erweiterung im OpenGL Extension String lautet *GL_NV_vertex_array_range*. Von den benötigten Funktionen fordern Sie die Adressen an:

```
void *wglAllocateMemoryNV(
  GLsizei size,
  GLfloat readFrequency,
  GLfloat writeFrequency,
  GLfloat priority );

void wglFreeMemoryNV
           ( void *ptr );

void glVertexArrayRangeNV(
  GLsizei size,
  const GLvoid *pointer );
```

Wenn Sie unter Linux arbeiten, lautet das *Namesprefix* der obigen Befehle nicht *wgl*, sondern *glX*. Mit der ersten Funktion können Sie den Speicher mit der Größe *size* (in Bytes) allokieren. Mit *readFrequency*, *writeFrequency* und *priority* können Sie die Zugriffscharakteristik und die Priorität des Speicherbereichs festlegen.

Allerdings sind lediglich zwei Parameterkombinationen praxisrelevant. Wenn Sie Speicher für DMA-/AGP-Zugriffe allokieren wollen, verwenden Sie *0.2 / 0.2 / 0.5* und für Videospeicher *0.2 / 0.2 / 1.0*. Wenn der Speicher in der gewünschten Größe verfügbar ist, erhalten Sie als Rückgabewert dessen Speicheradresse, sonst Null.

DIESES 3D-MODELL besteht aus 1 087 716 Dreiecken. Eine GeForce-3 (200 MHz GPU) schafft mit den Optimierungen dieses Artikels etwa 17 Bilder pro Sekunde.

Anschließend teilen Sie dem Grafikkartentreiber mit, dass Sie diesen Speicherbereich mit den *Vertex Array Range Extensions* nutzen wollen. Dies geschieht mit dem Befehl *glVertexArrayRangeNV*, wobei die Größe und der Zeiger auf den Speicherbereich die Parameter der Funktion sind. Wichtig für die Performance ist, dass Sie nur einen Speicherbereich so allokieren.

Wenn Sie mehrere Arrays benötigen, sollten Sie unbedingt diese in einen Speicherbereich zusammenkopieren und gegebenenfalls einen kleinen Speichermanager schreiben. Mit der folgenden Methode können Sie einen Speicherbereich nach Wunsch ansprechen. Kann kein Videospeicher allokiert werden, wird jeweils der nächst langsamere Speichertyp angefordert, bis dies gelingt. Der Speicherbereich sollte auf 64-Byte-Grenzen *aligned* werden, weil dies für manche Vertex-Datenformate wichtig ist:

```
void *allocateMemory(U32 size)
{
  size += 64;

  void *varMemory = NULL;

  // Extension unterstützt ?
  if ( supportVAR )
  {
if ( memoryType == VIDEOMEMORY )
    {
varMemory = wglAllocateMemoryNV
          ( size, .2, .2, 1 );
      if ( varMemory == NULL )
        memoryType = AGPMEMORY;
    }

  if ( memoryType == AGPMEMORY )
varMemory = wglAllocateMemoryNV
          ( size, .2, .2, .5 );

  if ( varMemory == NULL )
  {
    memoryType = SYSTEMMEMORY;
varMemory=(void*)new char[size];
  } else
    glVertexArrayRangeNV(size,
                  varMemory);

  // Alignment auf 64 Byte
  varMemory =
 (void*)
 (((int)varMemory+64)&~63);

  return varMemory;
}
```

Jetzt können Sie Ihre Vertex- und Attribut-Arrays erzeugen und in den gerade allokierten Speicherbereich kopieren. Diesen Speicher können Sie nutzen wie jeden anderen. Sie müssen nur, wenn Sie die Daten darin modifizieren, daran denken, dass auch die Grafikkarte diesen Speicherbereich liest; das heißt, für dynamische Vertexdaten sind Synchronisationsmechanismen notwendig. Dazu steht die Erweiterung *GL_NV_fence* zur Verfügung. Der folgende Code zeigt exemplarisch die Erzeugung der Arrays für Vertices und Normale:

```
// 24 Byte pro Vertex
// Koordinate+Normale à 3Floats
VERTEX3D *memory = (VERTEX3D*)
allocateMemory(nVertices*24 );

VERTEX3D *varVertex, *varNormal;

varVertex = &varMemory[ 0 ];
varNormal =
       &varMemory[nVertices];

for ( i = 0; i < nVertices; i++)
{
```

```
  varVertex[ i ] = ...;
  varNormal[ i ] = ...;
}
```

Das Rendering selbst erfolgt genauso, wie Sie es von den Arrays von OpenGL her kennen; abgesehen davon, dass Sie die Erweiterung *VAR* zuvor aktivieren:

```
glEnableClientState
  ( GL_VERTEX_ARRAY_RANGE_NV );

glVertexPointer( 3, GL_FLOAT,
                 0, varVertex );
glNormalPointer( GL_FLOAT, 0,
                 varNormal );

glEnable( GL_VERTEX_ARRAY );
glEnable( GL_NORMAL_ARRAY );

// Indexliste pIndexList mit 3
// Indizes für 'nFaces' Dreiecke
glDrawElements(
  GL_TRIANGLES, nFaces * 3,
  GL_UNSIGNED_INT, pIndexList );

glDisableClientState
  ( GL_VERTEX_ARRAY );

glDisableClientState
  ( GL_NORMAL_ARRAY );

glDisableClientState
  ( GL_VERTEX_ARRAY_RANGE_NV );
```

Am Ende Ihres Programms müssen Sie nur noch den Speicher freigeben. Sollten Sie ihn mit dem *VAR*-Befehlen allokiert haben, verwenden Sie *wglFreeMemoryNV*.

Der Grafikkartenhersteller ATI bietet für seine Radeon-GPUs die *ATI_vertex_array_object*-Extension an, die auch der Optimierung dient, aber eine andere Syntax und Semantik besitzt. Diese erlaubt es zunächst, so genannte *Array Objects* zu allokieren. Dabei handelt es sich um Speicherbereiche, in denen die Arrays für die Vertex- oder Normalendaten liegen. Wenn zusätzlich die *ATI_element_array*-Extension unterstützt wird, lassen sich die Indexlisten auch in einem *Array Object* ablegen. Dieses erzeugen Sie mit folgender Funktion. Zuvor müssen Sie deren Adresse, wie die anderen OpenGL-Extension-Funktionen laden:

```
GLuint glNewObjectBufferATI(
  GLsizei size,
  const GLvoid *pointer,
  GLenum usage );
```

Dabei ist *size* die Größe des Speicherbereichs, *pointer* der Zeiger auf Ihre Daten im Speicher und *usage* ist entweder *GL_STATIC_ATI* oder *GL_DYNAMIC_ATI* für eher statische oder dynamische Daten. Auch statische Daten können Sie im Nachhinein modifizieren, aber dabei an Performance verlieren. Der Rückgabewert ist entweder Null, wenn der Aufruf fehlgeschlagen ist, oder ein Integer als Identifier, den Sie für den späteren Gebrauch speichern müssen.

Erzeugen Sie für all Ihre Daten, und wenn die *ATI_element_array*-Extension unterstützt wird, auch für die Indexliste solche *Array Objects*. Jetzt sind Sie schon an der Stelle angelangt, an der es zum Rendering geht. Um die Array Objects an OpenGL als Daten-Arrays zu übergeben, gibt es folgenden Befehl:

```
void glArrayObjectATI(
  GLenum array, GLint size,
  GLenum type, GLsizei stride,
  GLuint buffer, GLuint offset );
```

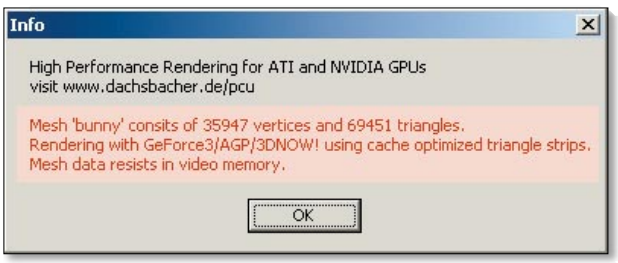
Array gibt an, welchem OpenGL-Array ein Array Object zugewiesen werden soll. Parameter ist beispielsweise *GL_VERTEX_ARRAY*. Die Größe eines Elements übergeben Sie in *size*, das Datenformat in *type*. Der *stride*-Wert ist Null, wenn die Daten dicht gepackt im Speicher liegen. Wenn Sie z.B. jeweils pro Vertex alle Attribute in Folge speichern, gibt der *Stride*-Wert die Größe der Datenstruktur an. Der *buffer*-Parameter enthält den Identifier des Array-Object, und der *offset*-Wert gibt an, wo die entsprechenden Daten in diesem Buffer starten. Das folgende Beispiel verdeutlicht die Aufrufe:

```
// Daten pro Vertex: 24 Byte
// typedef struct {
//   // Koordinate
//   float x, y, z;
//   // Normale
//   float nx, ny, nz; };

glEnableClientState
  ( GL_VERTEX_ARRAY );
glEnableClientState
  ( GL_NORMAL_ARRAY );

// Array Objects
glArrayObjectATI(
GL_NORMAL_ARRAY, 3, GL_FLOAT,24,
  atiVertexObject,
                sizeof(float)*3);

glArrayObjectATI(
  GL_VERTEX_ARRAY, 3,GL_FLOAT,24,
  atiVertexObject, 0 );

glDrawElements
        (GL_TRIANGLES,nFaces*3,
  GL_UNSIGNED_INT, pIndexList );

glDisableClientState
              (GL_VERTEX_ARRAY);
glDisableClientState
              (GL_NORMAL_ARRAY);
```

Wenn Sie die Indexliste ebenfalls in einem Array Object gespeichert haben, fügen Sie die folgenden Befehle hinzu und ersetzen den *glDrawElements*-Aufruf durch eine neue Funktion der ATI-Erweiterung:

```
glEnableClientState
  ( GL_ELEMENT_ARRAY_ATI );
...
glArrayObjectATI(
  GL_ELEMENT_ARRAY_ATI, 1,
  GL_UNSIGNED_INT, 0,
  atiArrayElement, 0 );
...
glDrawElementArrayATI
  ( GL_TRIANGLES, nFaces * 3 );
...
glDisableClientState
  ( GL_ELEMENT_ARRAY_ATI );
```

Damit haben Sie endgültig die Probleme der Geometrie-Speicherbandbreite gelöst. Das ist die Voraussetzung für hochperformantes Rendering mit vielen bzw. komplexen 3D-Objekten.



DIESE DIALOG-BOX des Beispielprogramms gibt Auskunft über den Renderer, die Daten und deren Lage im Speicher.

Triangle Strips und Cache-Optimierung

Verwenden Sie Triangle Strips ausgiebig. Wenn Sie Triangle Strips aus beliebigen 3D-Objekten anlegen, gibt es einiges zu beachten. Moderne GPUs verfügen unter anderem über zwei Caches: Der eine speichert untransformierte Vertexdaten, um die Geometriebandbreite zu schonen.

Diese hat eine Größe von mehreren Kilobyte. Viel kritischer für die Performance ist aber der Cache für bereits transformierte und beleuchtete Vertices. Er fasst bei GeForce-1/2-Karten beispielsweise 16, für GeForce-3 schon 24 Vertices. Triangle Strips können Sie so anlegen, dass möglichst Vertices, die bereits im Cache liegen, zur Fortführung des Strips verwendet werden. Glücklicherweise bietet nVidia die *NvTriStrip Library* (inklusive Quelltext) zum Download an, die die Aufgabe der Triangle-Strip-Generierung übernimmt. Im Sourcecode zu dieser Ausgabe befindet sich eine leicht modifizierte Variante (um 32 Bit Indizes verwenden zu können), deren Benutzung Ihnen die folgenden Quellcode-Fragmente verdeutlichen.

Fürs Stripping benötigen Sie lediglich die Indexliste der Shared-Vertex- Struktur. Damit füllen Sie die folgende Struktur aus:

```
#include „nvtristrip.h“

PrimitiveGroup triangles;

triangles.type = PT_LIST;
triangles.numIndices = nFaces*3;
triangles.indices =
  new unsigned int[ nFaces*3 ];

// Indizes für jedes Dreieck

for ( int i = 0; i < nFaces;i++)
{
triangles.indices[ i*3+0 ] =...;
triangles.indices[ i*3+1 ] =...;
triangles.indices[ i*3+2 ] =...;
}
```

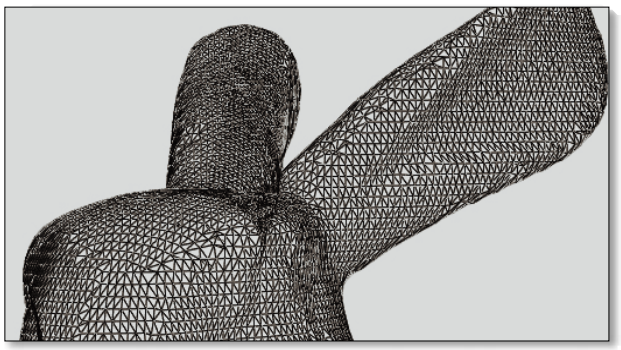
Jetzt legen Sie die Cache-Größe fest, die bei der Generierung berücksichtigt werden soll, und teilen mit, dass Sie einen großen Triangle Strip (und nicht mehrere) wollen:

```
SetCacheSize
     ( CACHESIZE_GEFORCE3 );
SetStitchStrips( true );
```

Damit können Sie die Strips erzeugen lassen, wobei der *nGroups*-Parameter *1* sein wird, weil nur ein Strip erzeugt wird:

```
PrimitiveGroup *strip;

strip = new PrimitiveGroup;
unsigned short nGroups;

GenerateStrips(
  triangles.indices,
  triangles.numIndices,
  &strip, &nGroups );
```

Zugriff auf die Indizes erhalten Sie mit *strip->indices*, wobei es sich um *strip->numIndices* handelt. Der Vorteil der Triangle Strips ist, dass Sie es meist mit weniger Indizes als bei der Shared-Vertex-Darstellung zu tun haben. Vor allem werden die Caches ausgenutzt und der Clipping-Aufwand reduziert.



DIESE BILD VERDEUTLICH wie fein aufgelöst die Beispiels-Dreiecksnetze sind.

Diese Funktionalen, die Sie in Ihre eigenen Programme zur Beschleunigung einbauen können, finden Sie in unseren Beispielprogrammen zu dieser Ausgabe.

Analyse weiterer Engpässe

Wenn Sie trotz der obigen Optimierungen mit Ihren Programmen noch nicht nahe an die maximale theoretische Leistungsfähigkeit Ihrer Grafikkarte stoßen, stellen Sie mit einfachen Tests fest, ob und wo der begrenzende Faktor in der Grafik-Pipeline liegt. Denken Sie daran, dass die Render-Performance der meisten Spiele und Demos durch die CPU (bzw. eine nicht optimale Umsetzung der Renderloops) oder so genannte *Stalls* (erzwungene Synchronisationen zwischen CPU und Grafikkarte) beschränkt ist.

Ob die Performance durch die Transform-and-Lighting-Berechnung begrenzt ist, können Sie einfach feststellen, indem Sie die Anzahl der Lichtquellen erhöhen oder reduzieren. Sollte sich die Geschwindigkeit beim Rendering ändern, ist das ein Indiz dafür. Ähnlich verhält es sich, wenn Sie Vertex-Shader verwenden. Deren Ausführungszeit ist proportional zu ihrer Länge. Durch Hinzufügen oder Entfernen von Instruktionen können Sie feststellen, ob es sich hierbei um einen Flaschenhals handelt. Aber beachten Sie, dass offensichtlich unnötige Operationen in Vertex-Shadern meist automatisch eliminiert werden. Für die künstliche Verlängerung der Shader addieren Sie am besten eine Null aus dem Konstantenspeicher auf ein Ausgaberegister.

Die Geometrie-Bandbreite können Sie testen, indem Sie unbenutzte Attribute wie weitere Texturkoordinaten mitübertragen. Wenn die Geschwindigkeit sinkt, befinden Sie sich an der Grenze dieser Bandbreite.

Die Füllrate können Sie in vielerlei Hinsicht untersuchen. Zum einen ist unterschiedliche Performance bei geänderter Bildschirmauflösung bzw. Fenstergröße ein Indiz. Zum anderen sollten Sie aufwändige Blending-Operationen und Multitexturing-Teile Ihres Programmes untersuchen. ET

Infos zu Grafikkarten und zum Artikel finden Sie unter folgenden Web-Adressen:
www.nvidia.com
www.ati.org
www.dachsbacher.de/pcu