C for Graphics

Benjamin Rommel

12.05.2009

Seminar 'Paralleles Rechnen auf Grafikkarten'

Sommersemester 2009

Betreuer: Julian M. Kunkel

Gliederung (1)

- Einführung
  - Was ist Cg?
  - Historische Entwicklung
  - Einordnung auf der Grafikkarte
- Grundlagen
  - Besonderheiten von Cg
    - Datentypen
    - Semantiken
  - Bestandteile einer Cg-Anwendung
    - Kontext
    - Programm
    - Profil
  - Beispiel: simples Vertex-Programm
  - Beispiel: simplex Fragment-Programm
  - *DEMO: sehr einfache Cg-Anwendung*

Gliederung (2)

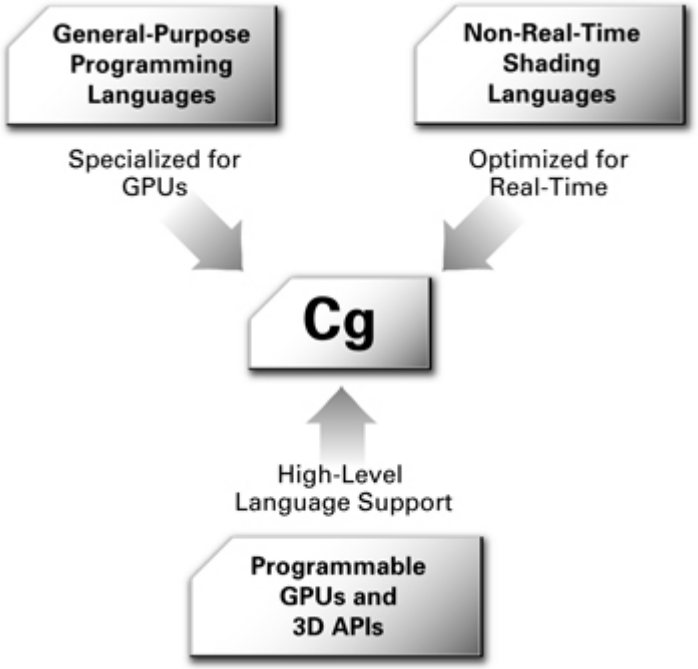
- Ausgewählte Themen
  - Texturen
    - Theorie: Texture Mapping
    - Texture Mapping in Cg
    - Double Vision
  - Beleuchtung
    - Motivation
    - Theorie: Das Lichtmodell
    - Vertex Shader Beleuchtung

Was ist Cg?

„Cg targets the ability to programmatically control the shape, appearance, and motion of objects rendered using graphics hardware. Broadly, this type of language is called a shading language. However, Cg can do more than just shading. For example, Cg programs can perform physical simulation, compositing, and other nonshading tasks.“
[KILGARD, FERNANDO 2003]

Historische Entwicklung

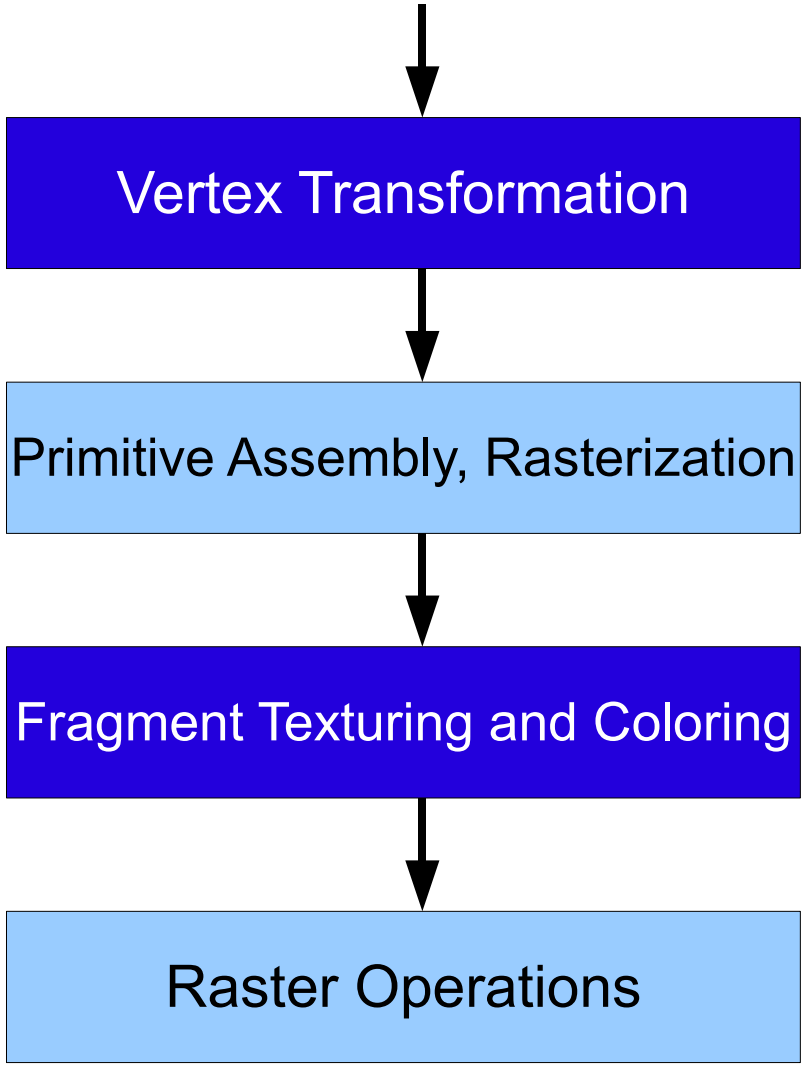
- Cg ist als Hochsprache für Grafikkartenprogrammierung konzipiert
- Enge Zusammenarbeit von NVIDIA und Microsoft => OpenGL und Direct3D



Quelle: [KILGARD, FERNANDO 2003]

- Syntax und Semantik basiert auf C
- Konzepte basieren auf Shading Languages
- Grafische Funktionen basieren auf OpenGL und DirectX

Einordnung auf der Grafikkarte



- Texturkoordinaten berechnen
- Farben berechnen (z.B. Licht)

- Primitive durch Rasterisierung erzeugen

- Fragmente überarbeiten

- Depth Test, Stencil Test, ...

Gliederung (1)

- Einführung
  - Was ist Cg?
  - Historische Entwicklung
  - Einordnung auf der Grafikkarte

- **Grundlagen**
  - Besonderheiten von Cg
    - Datentypen
    - Semantiken
  - Bestandteile einer Cg-Anwendung
    - Kontext
    - Programm
    - Profil
  - Beispiel: simples Vertex-Programm
  - Beispiel: simplex Fragment-Programm
  - *DEMO: sehr einfache Cg-Anwendung*

Besonderheiten von Cg (1)

- Datentypen
  - Keine Datentypen und Operationen für Strings
  - Vektor-Datentypen

```
float4 data = { 0.5, -2, 3, 3.14159 };
```

weitere Beispiele: `float2, float3`

`float4 (Cg)  ≠  float[4] (C/C++)`

  - Matrix-Datentypen

```
float2x3 matrix = { 1.0, 2.0,
                    3.0, 4.0,
                    5.0, 6.0 };
```

weitere Beispiele: `float4x4,  float2x4`

Besonderheiten von Cg (2)

- Semantiken
  - Weist einer Variablen ihre Bedeutung in der Grafik-Pipeline zu
  - Die „Verbindung“ zwischen Programm und Grafik-Pipeline
  - Mit Doppelpunkt vom Variablennamen getrennt
  - Beispiel (Variablendeklaration):

```
float3 position : POSITION;
```

  - Semantik unabhängig vom Variablennamen

Vorsicht: `float3 color : POSITION;`

Besonderheiten von Cg (3)

- Alle Semantiken in der Übersicht
  - Input Semantics

COLOR, COLOR0	Primärfarbe
COLOR1	Sekundärfarbe
TEX0-3	Textur 1-4
TEXCOORD0-3	Texturkoordinaten 1-4
FOGP	Nebelfarbe
FOG	Nebelfaktor

  - Output Semantics

COLOR, COLOR0	Farbe
DEPTH	Tiefenwert

Bestandteile einer Cg-Anwendung (1)

- Kontext
  - Container für alle Cg-Profile und Cg-Programme
  - Gleichzeitig kann nur ein Fragment- und Vertex-Programm aktiv sein
  - Kontext beinhaltet beliebig viele Fragment- und Vertex-Programme
  - Bildet den Ankerpunkt zwischen OpenGL-/Direct3D-Programm und Cg-Programm
  - Muss nur einmal erstellt werden, wird automatisch verwaltet

```
CGcontext cg_context = cgCreateContext();
```

Bestandteile einer Cg-Anwendung (2)

- Profile
  - Profile repräsentieren die Möglichkeiten mit einer GPU / Grafik API
  - Jedes Programm (Fragment und Vertex) nutzt ein Cg-Profil
  - Können automatisch von der Grafikkarte ausgelesen werden, z.B.

```
CGprofile  cg_vertex_profile  =
           cgGLGetLatestProfile(CG_GL_VERTEX);
```

  - Profil-Beispiele (Vertex):

Profile Name	Interface	Description
arbvp1	OpenGL	Basic multivendor vertex programmability
vs_1_1	DirectX 8	Basic multivendor vertex programmability

  - Profile-Beispiele (Fragment):

Profile Name	Interface	Description
ps_1_3	DirectX 8	Basic multivendor fragment programmability
fp20	OpenGL	Basic NVIDIA fragment programmability

Bestandteile einer Cg-Anwendung (3)

- Programme
  - Die eigentlichen Cg-Programme liegen als *.cg-Dateien vor
  - Wird zum Programmbeginn kompiliert und in den Kontext geladen

```
CGprogram cg_vertex_program =
     cgCreateProgramFromFile(cg_context,
     CG_SOURCE, "vertex.cg", cg_vertex_profile,
     "simple_vertex_shader", NULL);

cgGLLoadProgram(cg_vertex_program);
```

  - Wird zur Laufzeit auf die Grafikkarte geladen

```
cgGLBindProgram(cg_vertex_program);
```

Zusammenfassung: Einordnung im Programm

- Mit dem eigentlichen DirectX- / OpenGL-Programm werden die Cg-Programme compiliert und geladen und bei gewünschter Verwendung auf die Grafikkarte ausgelagert.
  - Dazu notwendig sind Profile und der Kontext
- Cg-Programme erhalten die Vertices, Fragmente, usw. durch die Grafikkartenpipeline und nicht durch das OGL-/DX-Programm
- Es können auch im OGL-/DX-Programm Werte berechnet und als Parameter an Cg-Programme übergeben werden.
  - Siehe dazu später: Texture Mapping

Beispiel: simples Vertex-Programm

Quellcode

```
struct Output
{
    float4 position : POSITION;
    float3 color    : COLOR;
};

Output simple_vertex_shader(float2 position : POSITION)
{
    Output OUT;

    OUT.position = float4(position, 0, 1);
    OUT.color    = float3(0.0, 1.0, 0.0);

    return OUT;
}
```

Beispiel: simples Fragment-Programm

Quellcode

```
struct Output
{
    float4 color  : COLOR;
};

Output simple_fragment_program(float4 color : COLOR)
{
    Output OUT;

    OUT.color = float4(1.0, color[1], 0.0, 1.0);

    return OUT;
}
```

DEMO: sehr einfache Cg-Anwendung (1)

- Aufgabe: Ein Dreieck soll mit Vertex- und Fragment-Buffer gefärbt werden
- Schritt 1: Dreieck in OpenGL (oder Direct3D) zeichnen

```
glBegin(GL_TRIANGLES);
    glVertex2f(-0.8, 0.8);
    glVertex2f(0.8, 0.8);
    glVertex2f(0.0, -0.8);
glEnd();
```

=> Vertices werden ohne Farbinformationen an die GPU gesendet

- Schritt 2: Vertices durchlaufen das Vertex-Programm auf der GPU
  - Die 2D-Koordinaten werden in homogene Koordinaten transformiert
  - Alle Vertices werden grün gefärbt

DEMO: sehr einfache Cg-Anwendung (2)

- (Rasterung: automatische Rasterung der Vertices zu einem Primitive)
- Schritt 3: Bearbeiten der Fragmente durch das Fragment-Programm
  - Alle Fragmente werden gelb gefärbt
- (Raster Operations: Stencil Test, Depth Test, ...)

=> Quelltext, Programm

Gliederung (2)

- Ausgewählte Themen
  - **Texturen**
    - Theorie: Texture Mapping
    - Texture Mapping in Cg
    - Double Vision
  - Beleuchtung
    - Motivation
    - Theorie: Das Lichtmodell
    - Vertex Shader Beleuchtung
    - Fragment Beleuchtung

Texturen – Texture Mapping (1)

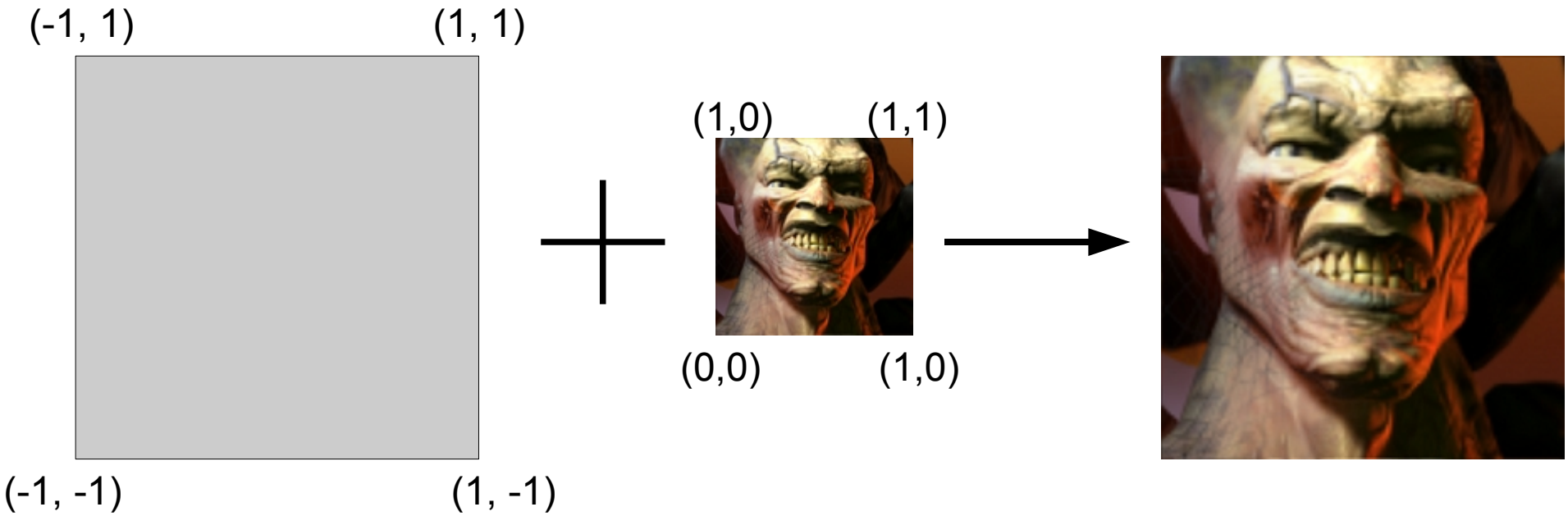
- Motivation
  - Um virtuelle Objekte realistisch wirken zu lassen, werden diese mit Farbinformationen aus einer Bilddatei, Textur genannt, versehen.
  - Die Objektoberfläche bekommt hierdurch die Optik der Bilddatei

- Funktionsweise
  - Jedem Vertex eines Polygons wird eine Texturkoordinate zugewiesen
  - Die Farbwerte der Textur werden anhand der Texturkoordinaten auf das Polygon abgebildet, ggf. ist Interpolation notwendig

- Warum in Cg?
  - Vertexkoordinaten könnten sich im Vertex-Shader ändern, damit auch Texturkoordinaten
  - Schnellere Texturierung, aufgrund der Parallelisierung und der in Cg implementierten Matrix-Funktionen

Texturen – Texture Mapping (2)

- Beispiel
  - Die Textur ist kleiner als das Quadrat => Interpolation notwendig



Texturen – Texture Mapping in Cg (1)

- Voraussetzungen
  - Textur bereits mit OpenGL/Direct3D geladen
  - Texturkoordinaten für das Primitive festgelegt

- Schritt 1: Cg-Parameter für die Textur in OpenGL-/Direct3D-Programm festlegen

- Schritt 2: Texturkoordinaten mit dem Vertex Shader weiterreichen

- Schritt 3: Textur im Fragment Programm auf Primitive abbilden

Texturen – Texture Mapping in Cg (2)

- Schritt 1: Cg-Parameter für die Textur in OpenGL-/Direct3D-Programm festlegen

Deklaration

```
static Cgparameter cg_parameter;
```

Initialisierung

```
cg_parameter =
  cgGetNamedParameter(cg_fragment_program, "decal");
cgGLSetTextureParameter(cg_parameter, 666);
```

Rendern

```
cgGLEnableTextureParameter(cg_parameter);
// ...
cgGLDisableTextureParameter(cg_parameter);
```

Texturen – Texture Mapping in Cg (3)

- Schritt 2: Texturkoordinaten mit dem Vertex Shader weiterreichen

```
struct Output
{
    float4 position  : POSITION;
    float3 color     : COLOR;
    float2 texCoord  : TEXCOORD0;
};


Output simple_vertex_shader(float2 position : POSITION, float3
                            color : COLOR, float2 texCoord : TEXCOORD0)
{
    Output OUT;

    OUT.position = float4(position, 0, 1);
    OUT.color    = color;
    OUT.texCoord = texCoord;

    return OUT;
}
```

Texturen – Texture Mapping in Cg (4)

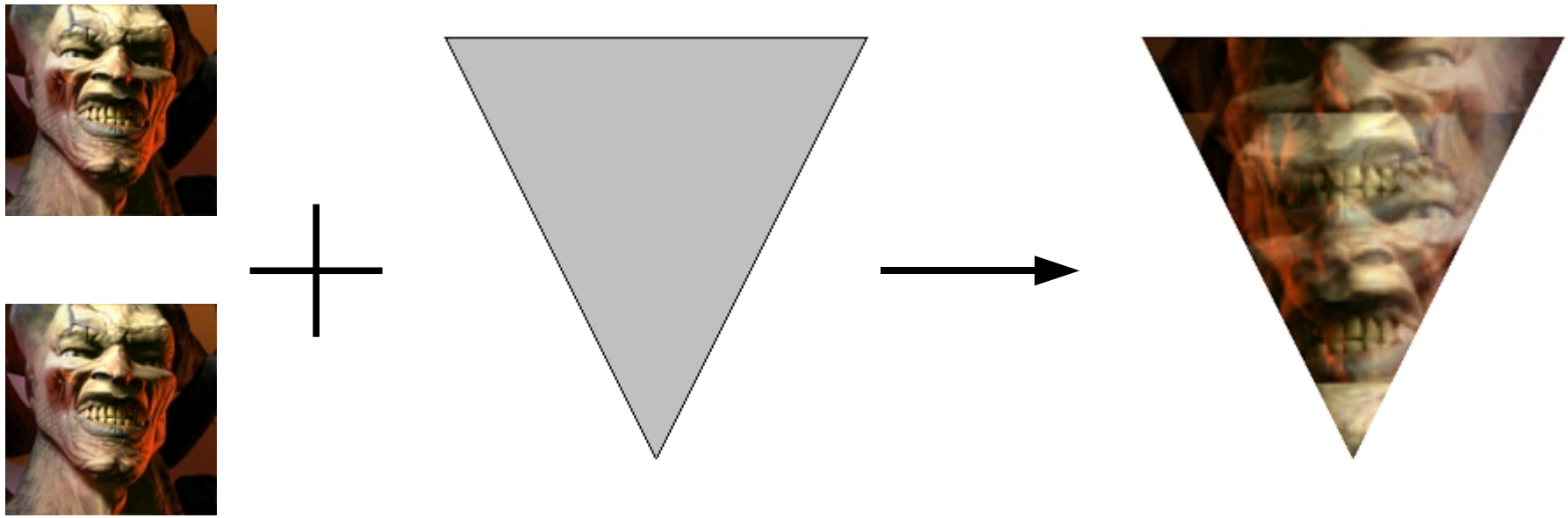
- Schritt 3: Textur im Fragment Programm auf Primitive abbilden

```
struct Output
{
    float4 color : COLOR;
};

Output simple_fragment_program(float2 texCoord :
 TEXCOORD0, uniform sampler2D decal : TEX0)
{
  Output OUT;

  OUT.color    = tex2D(decal,texCoord);

  return OUT;
}
```

=> Quelltext, Programm

Texturen – Double Vision (1)

- zwei Texturen werden versetzt übereinandergelegt und die Farben interpoliert
  - Vertex-Shader berechnet Texturkoordinaten für die Vertices
  - Fragment-Programm interpoliert Farbwerte mittels `lerp`



Texturen – Double Vision (2)

Quellcode

```
void vertex_shader( float2 position : POSITION,
                    float2 texCoord : TEXCOORD0,
                    out float4 oPosition : POSITION,
                  out float2 leftTexCoord : TEXCOORD0,
                  out float2 rightTexCoord : TEXCOORD1,
                     uniform float2 leftSeparation,
                     uniform float2 rightSeparation)
{
    oPosition     = float4(position, 0, 1);
    leftTexCoord  = texCoord + leftSeparation;
    rightTexCoord = texCoord + rightSeparation;
}
```

Texturen – Double Vision (3)

Quellcode

```
void fragment_program(float2 leftTexCoord : TEXCOORD0,
                      float2 rightTexCoord : TEXCOORD1,
                      out float4 color : COLOR,
                      uniform sampler2D decal)
{
    float4 leftColor  = tex2D(decal, leftTexCoord);
    float4 rightColor = tex2D(decal, rightTexCoord);
    color = lerp(leftColor, rightColor, 0.5);
}
```

Gliederung (2)

- Ausgewählte Themen
  - Texturen
    - Theorie: Texture Mapping
    - Texture Mapping in Cg
    - Double Vision
  - **Beleuchtung**
    - Motivation
    - Theorie: Das Lichtmodell
    - Vertex Shader Beleuchtung

Beleuchtung - Motivation

- Warum Beleuchtung?

  Beleuchtung verleiht einer virtuellen Szene mehr Realität, indem die Reflektion und Lichtbrechung an der Objektoberfläche simuliert wird.

- Wozu benötigt man ein Lichtmodell?

  Jeden Lichtstrahl, d.h. jede Wellenlänge an jedem Ort in einer Szene zu berechnen ist mit allen realen Einflüssen, wie Lichtbrechung, Reflektion und Interferierung nicht effizient zu berechnen. Lichtmodelle sind vereinfachte Modell der realen Lichtverhältnisse.

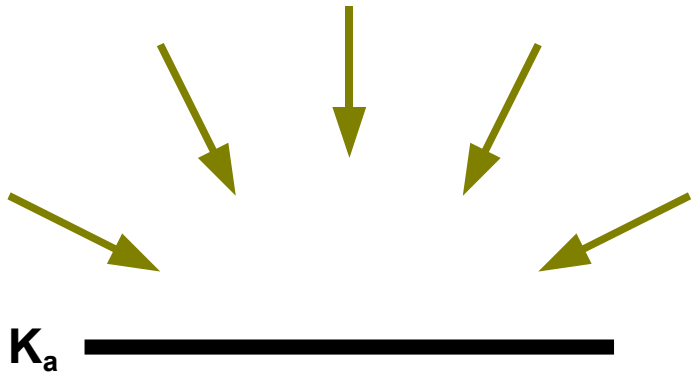
Beleuchtung - Theorie: Das Lichtmodell (1)

- Bestandteile des einfachen Lichtmodells
  - Umgebungslicht
  - Diffuses Licht
  - Glanzlicht

Umgebungslicht | Diffuses Licht | Glanzlicht

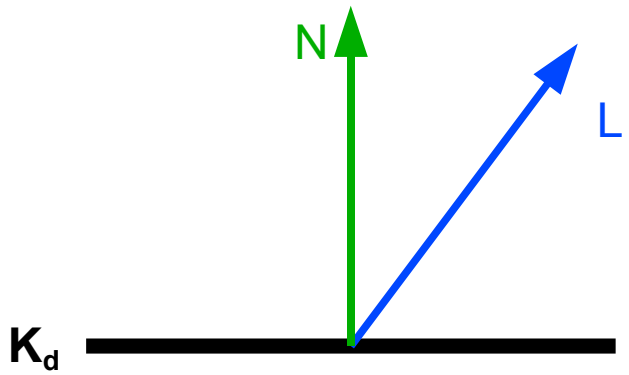
Beleuchtung - Theorie: Das Lichtmodell (2)

- Umgebungslicht
  - Keine spezifische Lichtquelle
  - Beleuchtung unabhängig von Oberflächenbeschaffenheit
  - Beispiel: Restlicht, Sonnenlicht
  - Formal: $\mathbf{I_a = K_a}$ **x AmbientColor**
    - K_a: Reflektionskoeffizient
    - AmbientColor: Farbe des Umgebungslichts



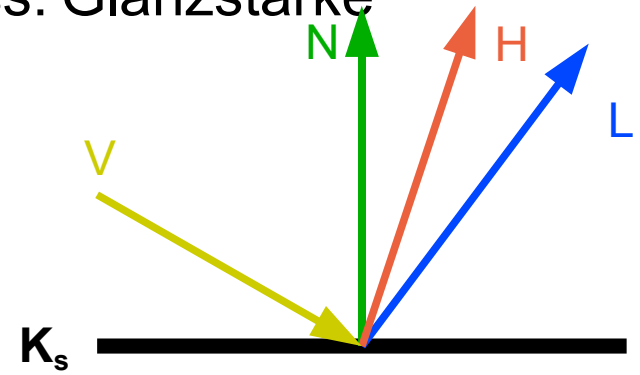
Beleuchtung - Theorie: Das Lichtmodell (3)

- Diffuses Licht
  - Eine gerichtete Lichtquelle, Reflexion in alle Richtungen
  - Je geringer der Winkel zw. Betrachtung und Lichtquelle, desto stärker ist das Licht
  - Beispiel: Taschenlampe
  - Formal: $\mathbf{I_d = K_d}$ **x DiffuseColor x max(N · L, 0)**
    - N: Oberflächennormale
    - L: normalisierter Vektor zur Lichtquelle



Beleuchtung - Theorie: Das Lichtmodell (4)

- Glanzlicht
  - Eine gerichtete Lichtquelle, Reflexion in alle Richtungen
  - Lichtintensität von Betrachterwinkel abhängig
  - Beispiel: Taschenlampe
  - Formal: $\mathbf{I_s = K_s \times SpecularColor \times facing \times (max(V \cdot H, 0))^{shininess}}$
    - V: Betrachtervektor
    - facing: 1, wenn $N \cdot L > 0$, sonst 0
    - H: halber Vektor zwischen N und L
    - Shininess: Glanzstärke



Beleuchtung – Vertex Shader Beleuchtung

```
...

   float3 eye_position = float3(0,0,5);
   float3 N = {0.0, 0.0, 1.0};
   float3 L = normalize(LightPosition - position);

   // Ambient Color
   float3 I_a = K[0] * AmbientColor;

   // Diffuse Color
   float3 I_d = K[1] * DiffuseColor * max(dot(N, L), 0);


   // final output
   float3 final_color = I_a + I_d;

...
```

Zusammenfassung

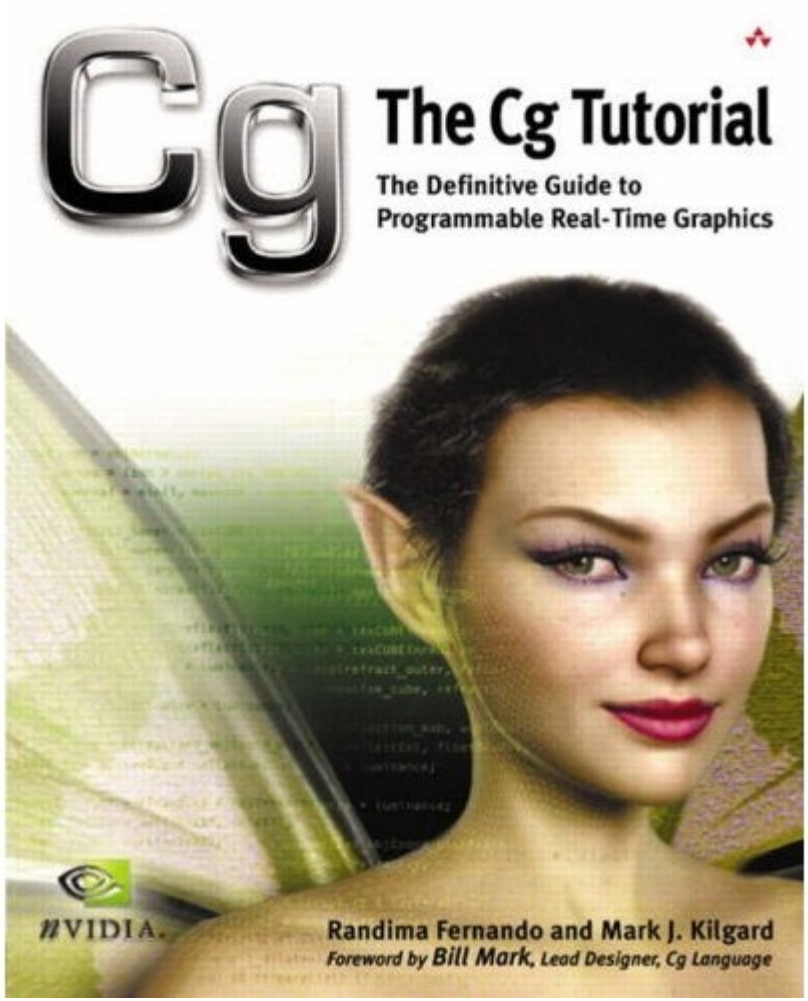
Fazit

- Cg ist sehr leicht zu erlernen, wenn man bereits OpenGL / DirectX kann
- Starke Effizienzsteigerung durch Auslagerung auf die Grafikkarte möglich
  - Paralleleigenschaften der Grafikkarte werden voll ausgenutzt
- CPU-Programm sehr leicht auf Cg zu portieren

Quellen

Quelle:

[KILGARD, FERNANDO 2003] Kilgard M.J., Fernando R. (2003):
The Cg Tutorial. Addison-Wesley Professional



C for Graphics

Benjamin Rommel

12.05.2009

Seminar 'Paralleles Rechnen auf Grafikkarten'
Sommersemester 2009
Betreuer: Julian M. Kunkel

Gliederung (1)

- Einführung
  - Was ist Cg?
  - Historische Entwicklung
  - Einordnung auf der Grafikkarte
- Grundlagen
  - Besonderheiten von Cg
    - Datentypen
    - Semantiken
  - Bestandteile einer Cg-Anwendung
    - Kontext
    - Programm
    - Profil
  - Beispiel: simples Vertex-Programm
  - Beispiel: simplex Fragment-Programm
  - *DEMO: sehr einfache Cg-Anwendung*

Inhaltsübersicht

Gliederung (2)

- Ausgewählte Themen
  - Texturen
    - Theorie: Texture Mapping
    - Texture Mapping in Cg
    - Double Vision
  - Beleuchtung
    - Motivation
    - Theorie: Das Lichtmodell
    - Vertex Shader Beleuchtung

3

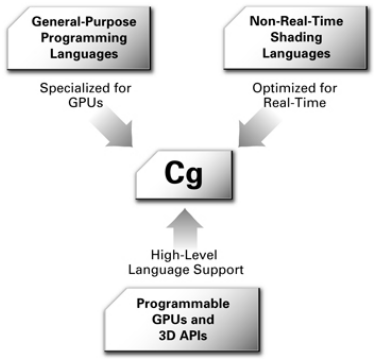
Diese beiden Themen habe ich gewählt, da sie Cg-Unkundigen problemlos zu erklären sind, optisch ansprechend sind und man sofort sehen kann, was im Quellcode geschieht. Desweiteren steckt hinter beiden ein mathematischer Hintergrund, der vollständig auf der Grafikkarte abgehandelt wird: Interpolation bei Texture Mapping und Normalenberechnung bei der Beleuchtung.

Was ist Cg?

„Cg targets the ability to programmatically control the shape, appearance, and motion of objects rendered using graphics hardware. Broadly, this type of language is called a shading language. However, Cg can do more than just shading. For example, Cg programs can perform physical simulation, compositing, and other nonshading tasks.“
[KILGARD, FERNANDO 2003]

Historische Entwicklung

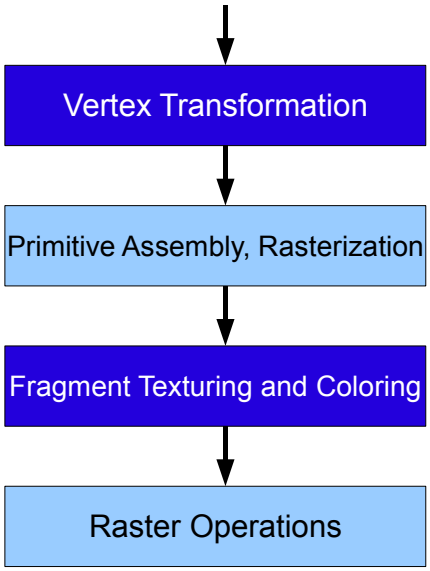
- Cg ist als Hochsprache für Grafikkartenprogrammierung konzipiert
- Enge Zusammenarbeit von NVIDIA und Microsoft => OpenGL und Direct3D



Quelle: [KILGARD, FERNANDO 2003]

- Syntax und Semantik basiert auf C
- Konzepte basieren auf Shading Languages
- Grafische Funktionen basieren auf OpenGL und DirectX

Einordnung auf der Grafikkarte



- Texturkoordinaten berechnen
- Farben berechnen (z.B. Licht)

- Primitive durch Rasterisierung erzeugen

- Fragmente überarbeiten

- Depth Test, Stencil Test, ...

Gliederung (1)

- Einführung
  - Was ist Cg?
  - Historische Entwicklung
  - Einordnung auf der Grafikkarte
- **Grundlagen**
  - Besonderheiten von Cg
    - Datentypen
    - Semantiken
  - Bestandteile einer Cg-Anwendung
    - Kontext
    - Programm
    - Profil
  - Beispiel: simples Vertex-Programm
  - Beispiel: simplex Fragment-Programm
  - *DEMO: sehr einfache Cg-Anwendung*

Besonderheiten von Cg (1)

- Datentypen
  - Keine Datentypen und Operationen für Strings
  - Vektor-Datentypen

```
float4 data = { 0.5, -2, 3, 3.14159 };
```

weitere Beispiele: `float2, float3`

```
float4 (Cg)  ≠  float[4] (C/C++)
```

  - Matrix-Datentypen

```
float2x3 matrix = { 1.0, 2.0,
                    3.0, 4.0,
                    5.0, 6.0 };
```

weitere Beispiele: `float4x4,  float2x4`

8

Dies sind Besonderheiten von Cg und gelten für die Vertex- und Fragment-Programme, nicht jedoch für die eigentliche OpenGL-/DirectX-Anwendung.

Grundlagen

Besonderheiten von Cg (2)

- Semantiken
  - Weist einer Variablen ihre Bedeutung in der Grafik-Pipeline zu
  - Die „Verbindung“ zwischen Programm und Grafik-Pipeline
  - Mit Doppelpunkt vom Variablennamen getrennt
  - Beispiel (Variablendeklaration):

```
float3 position : POSITION;
```

  - Semantik unabhängig vom Variablennamen

    Vorsicht: `float3 color : POSITION;`

9

Wenn die Daten über die Grafikpipelines weitergereicht werden, habe diese bestimmte Semantiken anhand derer man die Bedeutung der Daten erkennt.

In Cg-Programmen werden so eintreffende Daten erkannt und können weiterverarbeitet werden. Kommt z.B. ein Vektor mit der Semantik COLOR an, so weiß das Cg-Programm, dass es sich hierbei um einen Farbwert handelt.

Vorsicht ist jedoch bei der Verwendung geboten, da der Semantikname unabhängig vom Variablennamen ist und man daher als Entwickler selbst auf Konsistenz achten muss!

Besonderheiten von Cg (3)

- Alle Semantiken in der Übersicht
  - Input Semantics

COLOR, COLOR0	Primärfarbe
COLOR1	Sekundärfarbe
TEX0-3	Textur 1-4
TEXCOORD0-3	Texturkoordinaten 1-4
FOGP	Nebelfarbe
FOG	Nebelfaktor

  - Output Semantics

COLOR, COLOR0	Farbe
DEPTH	Tiefenwert

COLOR	Farbe des Vertex / Fragments
TEX0-3	Die verwendeten Texturen 1 bis 4
TEXCOORD0-3	Die Texturkoordinaten zu TEX0-3
FOGP	Die Farbe des Nebels
FOG	Faktor, wie stark der Nebel wird
DEPTH	z-Wert eines Vertex

Bestandteile einer Cg-Anwendung (1)

- Kontext
  - Container für alle Cg-Profile und Cg-Programme
  - Gleichzeitig kann nur ein Fragment- und Vertex-Programm aktiv sein
  - Kontext beinhaltet beliebig viele Fragment- und Vertex-Programme
  - Bildet den Ankerpunkt zwischen OpenGL-/Direct3D-Programm und Cg-Programm
  - Muss nur einmal erstellt werden, wird automatisch verwaltet

```
CGcontext cg_context = cgCreateContext();
```

Bestandteile einer Cg-Anwendung (2)

- Profile
  - Profile repräsentieren die Möglichkeiten mit einer GPU / Grafik API
  - Jedes Programm (Fragment und Vertex) nutzt ein Cg-Profil
  - Können automatisch von der Grafikkarte ausgelesen werden, z.B.

```
CGprofile cg_vertex_profile = cgGLGetLatestProfile(CG_GL_VERTEX);
```

  - Profil-Beispiele (Vertex):

Profile Name	Interface	Description
arbvp1	OpenGL	Basic multivendor vertex programmability
vs_1_1	DirectX 8	Basic multivendor vertex programmability

  - Profile-Beispiele (Fragment):

Profile Name	Interface	Description
ps_1_3	DirectX 8	Basic multivendor fragment programmability
fp20	OpenGL	Basic NVIDIA fragment programmability

Bestandteile einer Cg-Anwendung (3)

- Programme
  - Die eigentlichen Cg-Programme liegen als *.cg-Dateien vor
  - Wird zum Programmbeginn kompiliert und in den Kontext geladen

```
CGprogram cg_vertex_program =
     cgCreateProgramFromFile(cg_context,
     CG_SOURCE, "vertex.cg", cg_vertex_profile,
     "simple_vertex_shader", NULL);

cgGLLoadProgram(cg_vertex_program);
```

  - Wird zur Laufzeit auf die Grafikkarte geladen

```
cgGLBindProgram(cg_vertex_program);
```

Zusammenfassung: Einordnung im Programm

- Mit dem eigentlichen DirectX- / OpenGL-Programm werden die Cg-Programme compiliert und geladen und bei gewünschter Verwendung auf die Grafikkarte ausgelagert.
  - Dazu notwendig sind Profile und der Kontext
- Cg-Programme erhalten die Vertices, Fragmente, usw. durch die Grafikkartenpipeline und nicht durch das OGL-/DX-Programm
- Es können auch im OGL-/DX-Programm Werte berechnet und als Parameter an Cg-Programme übergeben werden.
  - Siehe dazu später: Texture Mapping

Beispiel: simples Vertex-Programm

Quellcode

```
struct Output
{
    float4 position : POSITION;
    float3 color    : COLOR;
};

Output simple_vertex_shader(float2 position : POSITION)
{
    Output OUT;

    OUT.position = float4(position, 0, 1);
    OUT.color    = float3(0.0, 1.0, 0.0);

    return OUT;
}
```


Dieses Cg-Vertex-Programm empfängt die zweidimensionale Position eines Vertices und wandelt die Position in homogene Koordiaten um:

OUT.position = float4(position, 0, 1);

Anschließend werden die bis dato ungefärbten Vertices grün gefärbt, d.h. der RGB-Wert (0,1,0) zugewiesen:

OUT.color = float3(0.0, 1.0, 0.0);

Grundlagen

Beispiel: simples Fragment-Programm

Quellcode

```
struct Output
{
    float4 color  : COLOR;
};

Output simple_fragment_program(float4 color : COLOR)
{
    Output OUT;

    OUT.color = float4(1.0, color[1], 0.0, 1.0);

    return OUT;
}
```

16

Wenn die Vertices mit den Polygon-Informationen (d.h. welche Form die Vertices annehmen sollen) durch die Rasterisierung gelaufen sind, liegen eine Menge an Fragmenten vor, welche, anschaulich gesprochen, das Polygon ausfüllen.

Diese Fragmente werden hier, in dem Fragment-Programm, nachträglich editiert, indem alle einkommenden Fragmente gelb gefärbt werden. Dabei wird der Rot-Wert auf 1 gesetzt und der Grün-Wert der eintreffenden Farbinformationen genommen. Es gibt sich die Farbe gelb, im RGB-Farbmodell (1,1,0).

DEMO: sehr einfache Cg-Anwendung (1)

- Aufgabe: Ein Dreieck soll mit Vertex- und Fragment-Buffer gefärbt werden
- Schritt 1: Dreieck in OpenGL (oder Direct3D) zeichnen

```
glBegin(GL_TRIANGLES);
    glVertex2f(-0.8, 0.8);
    glVertex2f(0.8, 0.8);
    glVertex2f(0.0, -0.8);
glEnd();
```

  => Vertices werden ohne Farbinformationen an die GPU gesendet

- Schritt 2: Vertices durchlaufen das Vertex-Programm auf der GPU
  - Die 2D-Koordinaten werden in homogene Koordinaten transformiert
  - Alle Vertices werden grün gefärbt

DEMO: sehr einfache Cg-Anwendung (2)

- (Rasterung: automatische Rasterung der Vertices zu einem Primitive)
- Schritt 3: Bearbeiten der Fragmente durch das Fragment-Programm
  - Alle Fragmente werden gelb gefärbt
- (Raster Operations: Stencil Test, Depth Test, ...)

=> Quelltext, Programm

Inhaltsübersicht

Gliederung (2)

- Ausgewählte Themen
  - **Texturen**
    - Theorie: Texture Mapping
    - Texture Mapping in Cg
    - Double Vision
  - Beleuchtung
    - Motivation
    - Theorie: Das Lichtmodell
    - Vertex Shader Beleuchtung
    - Fragment Beleuchtung

Diese beiden Themen habe ich gewählt, da sie Cg-Unkundigen problemlos zu erklären sind, optisch ansprechend sind und man sofort sehen kann, was im Quellcode geschieht. Desweiteren steckt hinter beiden ein mathematischer Hintergrund, der vollständig auf der Grafikkarte abgehandelt wird: Interpolation bei Texture Mapping und Normalenberechnung bei der Beleuchtung.

Texturen – Texture Mapping (1)

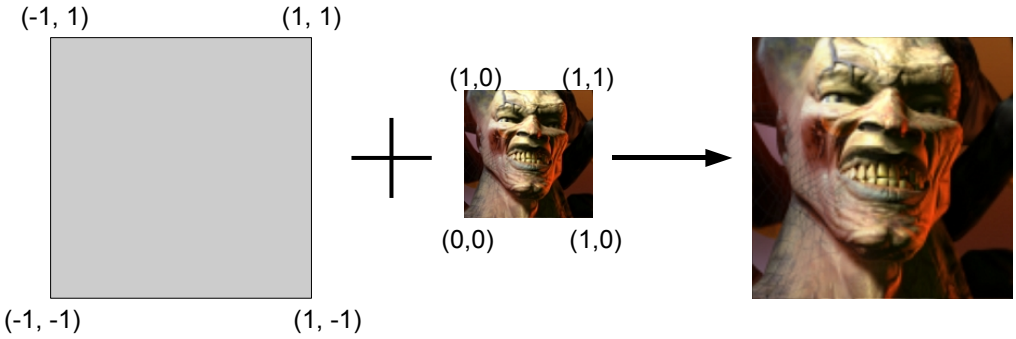
- Motivation
  - Um virtuelle Objekte realistisch wirken zu lassen, werden diese mit Farbinformationen aus einer Bilddatei, Textur genannt, versehen.
  - Die Objektoberfläche bekommt hierdurch die Optik der Bilddatei

- Funktionsweise
  - Jedem Vertex eines Polygons wird eine Texturkoordinate zugewiesen
  - Die Farbwerte der Textur werden anhand der Texturkoordinaten auf das Polygon abgebildet, ggf. ist Interpolation notwendig

- Warum in Cg?
  - Vertexkoordinaten könnten sich im Vertex-Shader ändern, damit auch Texturkoordinaten
  - Schnellere Texturierung, aufgrund der Parallelisierung und der in Cg implementierten Matrix-Funktionen

Texturen – Texture Mapping (2)

- Beispiel
  - Die Textur ist kleiner als das Quadrat => Interpolation notwendig



Texturen – Texture Mapping in Cg (1)

- Voraussetzungen
  - Textur bereits mit OpenGL/Direct3D geladen
  - Texturkoordinaten für das Primitive festgelegt
- Schritt 1: Cg-Parameter für die Textur in OpenGL-/Direct3D-Programm festlegen
- Schritt 2: Texturkoordinaten mit dem Vertex Shader weiterreichen
- Schritt 3: Textur im Fragment Programm auf Primitive abbilden

Texturen – Texture Mapping in Cg (2)

- Schritt 1: Cg-Parameter für die Textur in OpenGL-/Direct3D-Programm festlegen

Deklaration

```
static Cgparameter cg_parameter;
```

Initialisierung

```
cg_parameter =
  cgGetNamedParameter(cg_fragment_program, "decal");
cgGLSetTextureParameter(cg_parameter, 666);
```

Rendern

```
cgGLEnableTextureParameter(cg_parameter);
// ...
cgGLDisableTextureParameter(cg_parameter);
```

Texturen – Texture Mapping in Cg (3)

- Schritt 2: Texturkoordinaten mit dem Vertex Shader weiterreichen

```
struct Output
{
     float4 position  : POSITION;
     float3 color     : COLOR;
     float2 texCoord  : TEXCOORD0;
};


Output simple_vertex_shader(float2 position : POSITION, float3
                      color : COLOR, float2 texCoord : TEXCOORD0)
{
     Output OUT;

     OUT.position = float4(position, 0, 1);
     OUT.color    = color;
     OUT.texCoord = texCoord;

     return OUT;
}
```

Texturen – Texture Mapping in Cg (4)

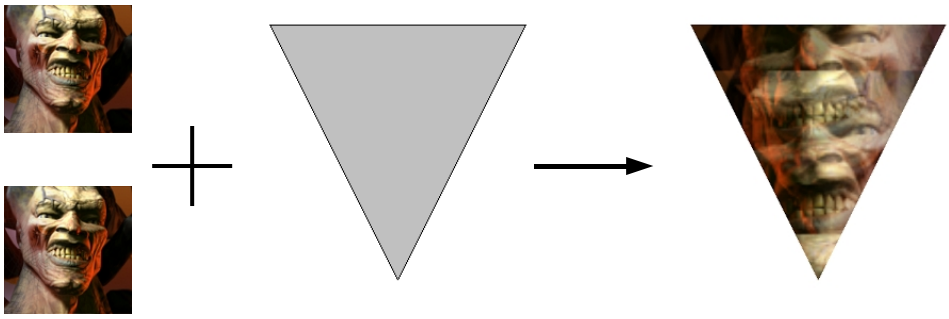
- Schritt 3: Textur im Fragment Programm auf Primitive abbilden

```
struct Output
{
    float4 color : COLOR;
};

Output simple_fragment_program(float2 texCoord :
 TEXCOORD0, uniform sampler2D decal : TEX0)
{
  Output OUT;

  OUT.color    = tex2D(decal,texCoord);

  return OUT;
}
```

=> Quelltext, Programm

Texturen – Double Vision (1)

- zwei Texturen werden versetzt übereinandergelegt und die Farben interpoliert
  - Vertex-Shader berechnet Texturkoordinaten für die Vertices
  - Fragment-Programm interpoliert Farbwerte mittels `lerp`



Texturen – Double Vision (2)

Quellcode

```
void vertex_shader( float2 position : POSITION,
                    float2 texCoord : TEXCOORD0,
                    out float4 oPosition : POSITION,
                  out float2 leftTexCoord : TEXCOORD0,
                  out float2 rightTexCoord : TEXCOORD1,
                    uniform float2 leftSeparation,
                    uniform float2 rightSeparation)
{
    oPosition     = float4(position, 0, 1);
    leftTexCoord  = texCoord + leftSeparation;
    rightTexCoord = texCoord + rightSeparation;
}
```

Texturen – Double Vision (3)

Quellcode

```
void fragment_program(float2 leftTexCoord : TEXCOORD0,
                      float2 rightTexCoord : TEXCOORD1,
                      out float4 color : COLOR,
                      uniform sampler2D decal)
{
    float4 leftColor  = tex2D(decal, leftTexCoord);
    float4 rightColor = tex2D(decal, rightTexCoord);
    color = lerp(leftColor, rightColor, 0.5);
}
```

Inhaltsübersicht

Gliederung (2)

- Ausgewählte Themen
  - Texturen
    - Theorie: Texture Mapping
    - Texture Mapping in Cg
    - Double Vision
  - **Beleuchtung**
    - Motivation
    - Theorie: Das Lichtmodell
    - Vertex Shader Beleuchtung

29

Diese beiden Themen habe ich gewählt, da sie Cg-Unkundigen problemlos zu erklären sind, optisch ansprechend sind und man sofort sehen kann, was im Quellcode geschieht. Desweiteren steckt hinter beiden ein mathematischer Hintergrund, der vollständig auf der Grafikkarte abgehandelt wird: Interpolation bei Texture Mapping und Normalenberechnung bei der Beleuchtung.

Beleuchtung - Motivation

- Warum Beleuchtung?

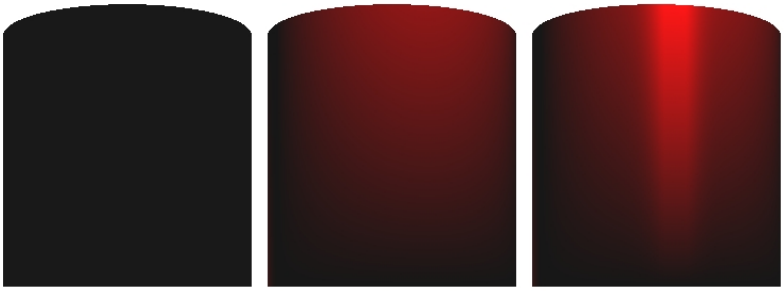
  Beleuchtung verleiht einer virtuellen Szene mehr Realität, indem die Reflektion und Lichtbrechung an der Objektoberfläche simuliert wird.

- Wozu benötigt man ein Lichtmodell?

  Jeden Lichtstrahl, d.h. jede Wellenlänge an jedem Ort in einer Szene zu berechnen ist mit allen realen Einflüssen, wie Lichtbrechung, Reflektion und Interferierung nicht effizient zu berechnen. Lichtmodelle sind vereinfachte Modell der realen Lichtverhältnisse.

Beleuchtung - Theorie: Das Lichtmodell (1)

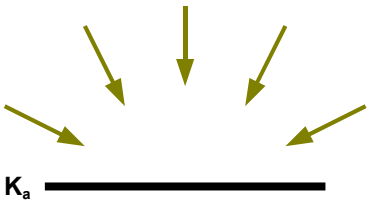
- Bestandteile des einfachen Lichtmodells
  - Umgebungslicht
  - Diffuses Licht
  - Glanzlicht



Umgebungslicht Diffuses Licht Glanzlicht

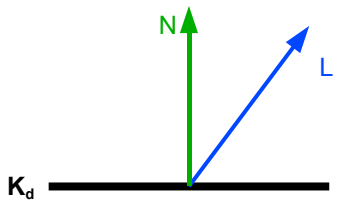
Beleuchtung - Theorie: Das Lichtmodell (2)

- Umgebungslicht
  - Keine spezifische Lichtquelle
  - Beleuchtung unabhängig von Oberflächenbeschaffenheit
  - Beispiel: Restlicht, Sonnenlicht
  - Formal: $\mathbf{I_a = K_a}$ **x AmbientColor**
    - K_a: Reflektionskoeffizient
    - AmbientColor: Farbe des Umgebungslichts



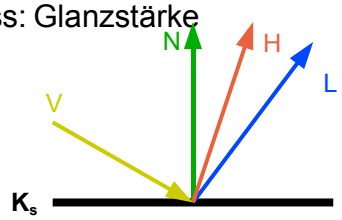
Beleuchtung - Theorie: Das Lichtmodell (3)

- Diffuses Licht
  - Eine gerichtete Lichtquelle, Reflexion in alle Richtungen
  - Je geringer der Winkel zw. Betrachtung und Lichtquelle, desto stärker ist das Licht
  - Beispiel: Taschenlampe
  - Formal: $\mathbf{I_d = K_d}$ **x DiffuseColor x max(N · L, 0)**
    - N: Oberflächennormale
    - L: normalisierter Vektor zur Lichtquelle



Beleuchtung - Theorie: Das Lichtmodell (4)

- Glanzlicht
  - Eine gerichtete Lichtquelle, Reflexion in alle Richtungen
  - Lichtintensität von Betrachterwinkel abhängig
  - Beispiel: Taschenlampe
  - Formal: $\mathbf{I_s = K_s}$ **x SpecularColor x facing x** $\mathbf{(max(V \cdot H, 0))^{shininess}}$
    - V: Betrachtervektor
    - facing: 1, wenn $N \cdot L > 0$, sonst 0
    - H: halber Vektor zwischen N und L
    - Shininess: Glanzstärke



Beleuchtung – Vertex Shader Beleuchtung

```
...

   float3 eye_position = float3(0,0,5);
   float3 N = {0.0, 0.0, 1.0};
   float3 L = normalize(LightPosition - position);

   // Ambient Color
   float3 I_a = K[0] * AmbientColor;

   // Diffuse Color
   float3 I_d = K[1] * DiffuseColor * max(dot(N, L), 0);


   // final output
   float3 final_color = I_a + I_d;

...
```

Zusammenfassung

Fazit

- Cg ist sehr leicht zu erlernen, wenn man bereits OpenGL / DirectX kann
- Starke Effizienzsteigerung durch Auslagerung auf die Grafikkarte möglich
  - Paralleleigenschaften der Grafikkarte werden voll ausgenutzt
- CPU-Programm sehr leicht auf Cg zu portieren

Quellen

Quelle:

[KILGARD, FERNANDO 2003] Kilgard M.J., Fernando R. (2003):
The Cg Tutorial. Addison-Wesley Professional