Computergraphik II

Winter 2011/2012

8 GPU Programmierung

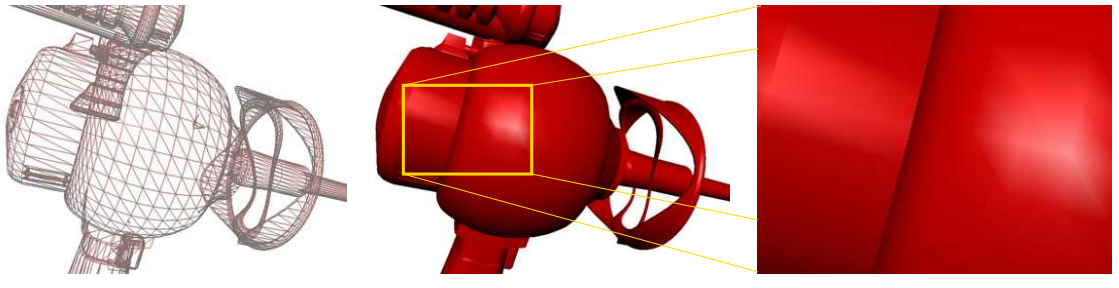
Versionsdatum: 14. November 2011

8 GPU Programmierung ...

Bisherige Themen CG-I: Raytracing und Rastergraphik (Graphik-Pipeline)

Einschränkungen bei der Nutzung der Rastergraphik

- „Kleinteilige“ Übertragung von Daten CPU→GPU (`glNormal()`, `glTexCoord()`, `glVertex()`; Lösung: Vertex-Arrays)
- Feste Verarbeitungsstruktur, z.B. per-Vertex Beleuchtung



Ziele: Grundlagen der GPU-Programmierung mit Anwendungsrichtungen

- flexiblere Datenverarbeitung, z.B. per-Fragment Beleuchtung
- Nutzung der Performance neuerer Graphikkarten

Literatur: Randi Rost: *OpenGL Shading Language*, Addison Wesley

8.1 Erinnerung Rastergraphik

Grundsätzlich Vorgehensweise bei der Rastergraphik

```
foreach Obj in SceneDatabase do {
  project Obj onto image plane
  foreach pixel (x,y) for Obj do {
    if ( Obj(x,y) closest so far )
      compute & set color for pixel (x,y) on Obj
  }
}
```

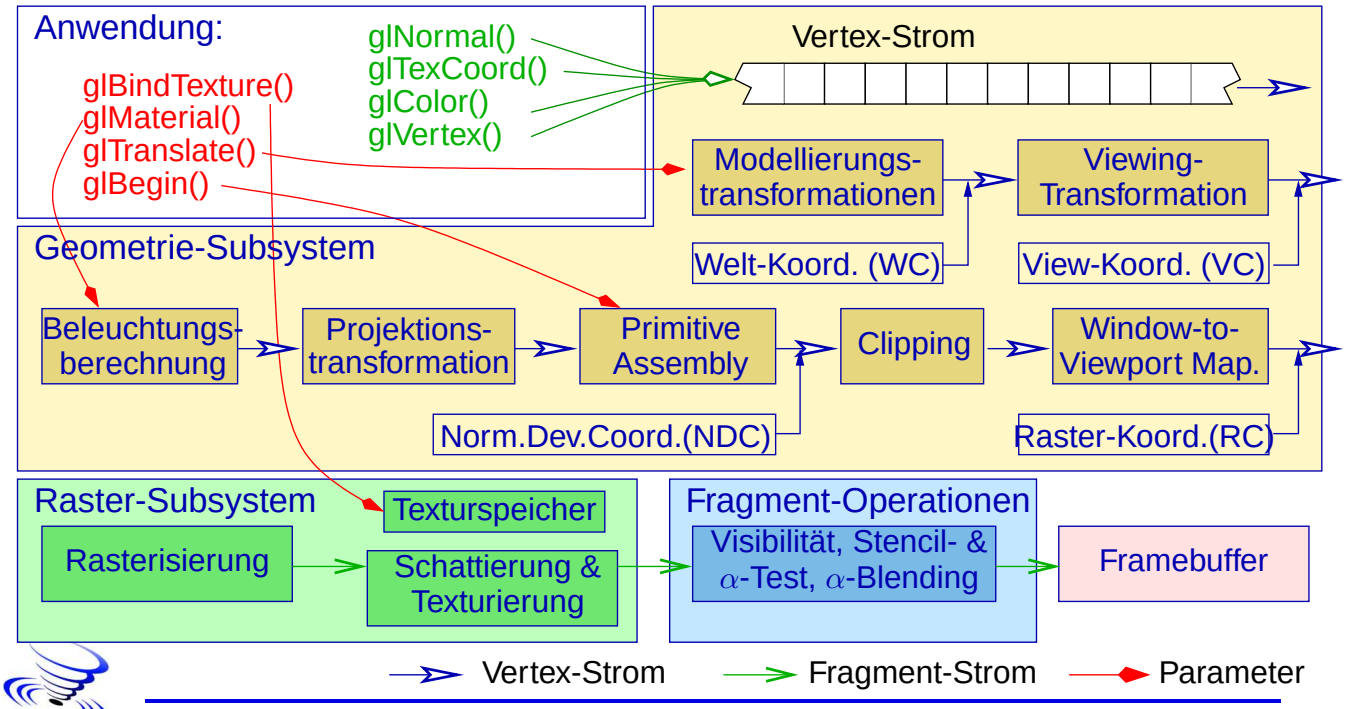
Subsysteme in der Rastergraphik

- Geometrie-Subsystem verarbeitet Punkte (Transform., Beleuchtung, etc.)
- Raster-Subsystem verarbeitet Fragmente, die späteren Pixel

Koordinatensysteme:
- Master-Koord. (MC): Hier werden Objekte erstellt
- Welt-Koordination (WC): Hier werden Objekt zueinander positioniert.
- View-Koordination (VC): Koordinaten des Beobachters
- Normalized Device Coordinates (NDC): $[-1,1]^3$
- Raster-Koordination (RC): Pixelkoordinaten in der Bildebene

8.1 Erinnerung Rastergraphik ...

- bestimmte OpenGL-Befehle ***parametrisieren*** die Graphik-Pipeline
- andere Befehle „schieben“ Daten in die Graphik-Pipeline (***Vertex-Stream***)



8.1.1 Rückblick: Geometrie-Subsystem

Ziel: Erinnerung an wichtige Bausteine des Geometrie-Subsystems

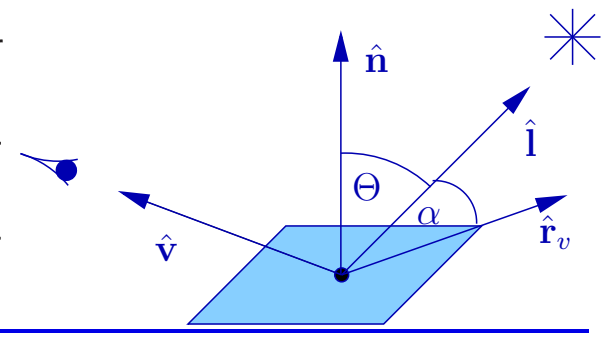
Model- und Viewing-Transformationen:

- Affine Model- (T_{model} : MC $\rightarrow$ WC) und Viewing-Transformation (T_{view} : WC $\rightarrow$ VC)
- OpenGL's Model-View-Matrix: $T_{\text{modelview}} = T_{view} \cdot T_{\text{model}}$

Per Vertex Beleuchtung nach dem Phong- (bzw. Blinn-Phong) Modell

$$\begin{aligned}\mathbf{I}_{\text{gesamt}} &= \mathbf{I}_a + \mathbf{I}_d + \mathbf{I}_s \\ &= \mathbf{k}_a * \mathbf{L}_a + \max\{\left(\hat{\mathbf{n}} \cdot \hat{\mathbf{l}}\right), 0\} \cdot \mathbf{k}_d * \mathbf{L}_d + \max\{\left(\hat{\mathbf{l}} \cdot \hat{\mathbf{r}}_v\right)^n, 0\} \cdot \mathbf{k}_s * \mathbf{L}_s\end{aligned}$$

$\mathbf{I}_{a,d,s}$	ambienter, diffuser, spekularer Beleuchtungsanteil
$\mathbf{L}_{a,d,s}$	amb., diff., spek. Licht der Lichtquelle
$\mathbf{k}_{a,d,s}$	amb., diff., spek. Reflexionskoeffizienten des Materials



8.1.1 Rückblick: Geometrie-Subsystem ...

Bemerkung: Transformation von Normalen

Vertex von MC nach VC: $\mathbf{P}^{VC} = T_{\text{modelview}} \cdot \mathbf{P}^{MC}$

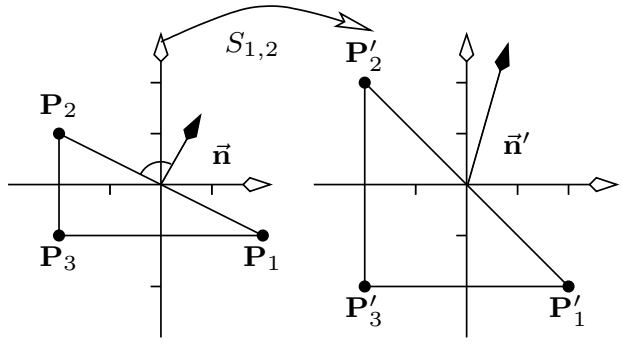
Was passiert mit Normalen, wenn man diese gleich transformiert?

2D Beispiel: Skalierung $S_{1,2}$

$$\mathbf{P}_1 = \begin{pmatrix}2\\-1\end{pmatrix},\ \mathbf{P}_2 = \begin{pmatrix}-2\\1\end{pmatrix},\ \vec{\mathbf{n}} = \begin{pmatrix}1\\2\end{pmatrix}$$

$$\mathbf{P}'_1 = \begin{pmatrix}2\\-2\end{pmatrix},\ \mathbf{P}'_2 = \begin{pmatrix}-2\\2\end{pmatrix},\ \vec{\mathbf{n}}' = \begin{pmatrix}1\\4\end{pmatrix}$$

$$(\vec{\mathbf{n}}' \cdot (\mathbf{P}'_2 - \mathbf{P}'_1)) = \left(\begin{pmatrix}1\\4\end{pmatrix} \cdot \begin{pmatrix}-4\\4\end{pmatrix}\right) = 12 \neq 0$$



Korrekt: $\mathbf{P}_i$ mit T transformieren und $\vec{\mathbf{n}}$ mit $(T^{-1})^T$, denn:

$$\begin{aligned}0 = (\vec{\mathbf{n}} \cdot (\mathbf{P}_2 - \mathbf{P}_1)) &= \vec{\mathbf{n}}^T \cdot (\mathbf{P}_2 - \mathbf{P}_1) = \vec{\mathbf{n}}^T \cdot (T^{-1} \cdot T) \cdot (\mathbf{P}_2 - \mathbf{P}_1) \\ &= (\vec{\mathbf{n}}^T \cdot T^{-1}) \cdot (T \cdot (\mathbf{P}_2 - \mathbf{P}_1)) = ((T^{-1})^T \cdot \vec{\mathbf{n}})^T \cdot (T \cdot (\mathbf{P}_2 - \mathbf{P}_1))\end{aligned}$$

Beachte: „·“ ist im 1. Ausdruck das Innere Produkt, sonst das ***Matrix-Produkt***

8.1.1 Rückblick: Geometrie-Subsystem ...

Projektions-Transformation:

- In der Regel perspektivische, selten orthographische Transformation
- Abbildung von VC in NDC
- OpenGL's Projection-Matrix: T_{proj}

Primitive Assemby: (das wurde in CG_I unterschlagen)

- Hier werden erstmals die Vertices zu Primitiven (entspr. `glBegin(`*Primitive-Typ*`)`) zusammengefügt
- zunächst können wir diese Stufe ignorieren

8.1.2 Rückblick: Raster-Subsystem

Ziel: Erinnerung an wichtige Bausteine des Raster-Subsystems

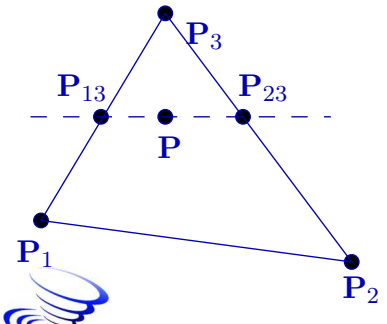
Input zum Raster-Subsystem sind Dreiecke mit

- Vertices $\mathbf{P}_i,\ i = 1, 2, 3$, wobei (p_x, p_y) Rasterkoordinaten und p_z die Tiefe ist
- per-Vertex Daten: Farbe $\mathbf{I}_i$, Textur-Koordinaten (s_i, t_i)

Schattierung & Texturierung basieren auf der ***Scanline-Rasterisierung***

- Interpolation der per-Vertex-Daten für jedes rasterisierte Pixel/Fragment
- Perspektivisch korrekte Interpolation der per-Vertex-Daten für Pixel

Erinnerung: Vereinfachte lineare Interpolation



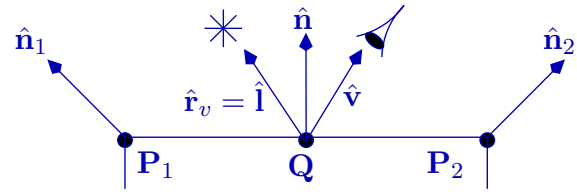
$$\mathbf{P}_{13} = (1-\alpha_{13})\mathbf{P}_1 + \alpha_{13}\mathbf{P}_3 \Rightarrow \mathbf{I}_{13} = (1-\alpha_{13})\mathbf{I}_1 + \alpha_{13}\mathbf{I}_3$$
$$\mathbf{P}_{23} = (1-\alpha_{23})\mathbf{P}_2 + \alpha_{23}\mathbf{P}_3 \Rightarrow \mathbf{I}_{23} = (1-\alpha_{23})\mathbf{I}_2 + \alpha_{23}\mathbf{I}_3$$
$$\mathbf{P} = (1-\alpha)\mathbf{P}_{13} + \alpha\mathbf{P}_{23} \Rightarrow \mathbf{I} = (1-\alpha)\mathbf{I}_{13} + \alpha\mathbf{I}_{23}$$

8.1.2 Rückblick: Raster-Subsystem ...

Probleme der Interpolation

Interpolation kann lokale Effekte nicht berücksichtigen, insbesondere spekulare Highlights innerhalb eines Dreiecks

Beispiel: Spekulare Highlights



Fehler bei spiegelnder Reflexion:
- bei $\mathbf{P}_1, \mathbf{P}_2$ keine spieg. Reflexion
- Gouraud: Kein spiegelnder Anteil bei $\mathbf{Q}$
- Phong bei $\mathbf{Q}$ liefert max. spieg. Reflexion

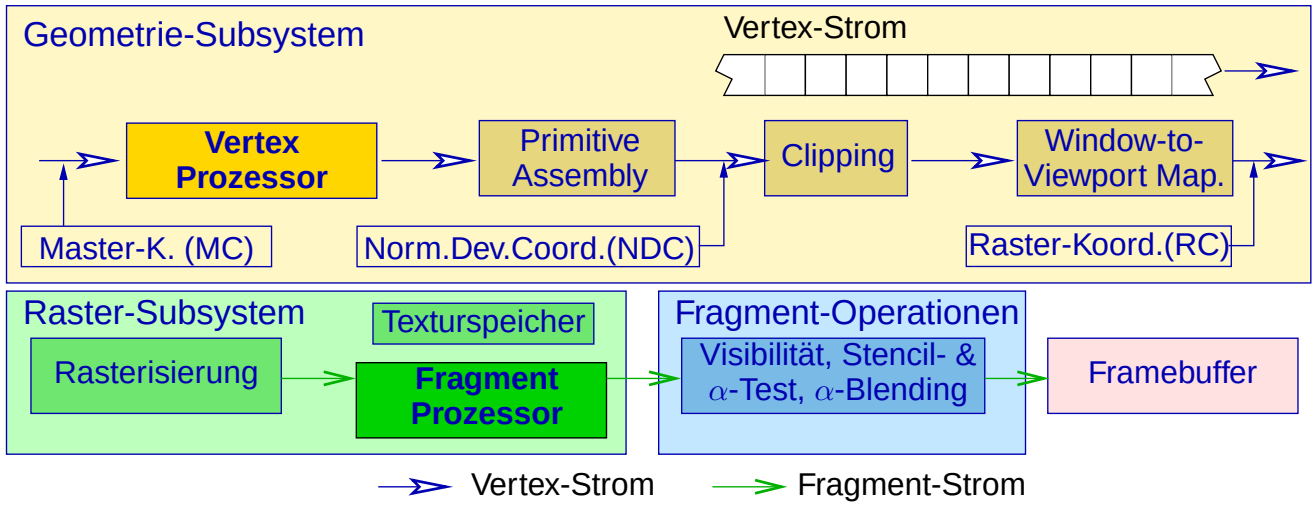
Fragment-Operationen

Visbilität: Wird durch z-Buffer-Algorithmus abgehandelt

α-Blending, Stencil-Test: siehe CG1-Abschnitt „Spezielle Aspekte der Computergraphik“

8.2 Programmierbare Graphikhardware

Grundidee: Workflow der Graphik-Pipeline bleibt erhalten, aber Teile des Geometrie-Subsystems (→ ***Vertex-Prozessor***) und des Raster-Subsystems (→ ***Fragment-Prozessor***) werden programmierbar



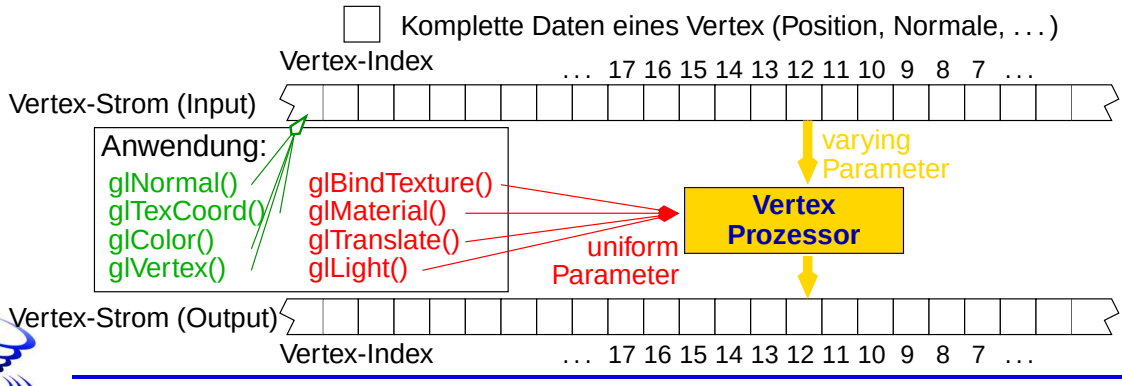
Stream-Prozessoren

GPU-Prozessoren sind ***Stream-Prozessoren***, d.h. sie

- lesen Daten von einem ***Input-Stream*** (read-only) (***Varying Variablen***),
- schreiben strukturgleiche Daten auf einen ***Output-Stream*** (write-only),
- sind programmierbar (→ ***Vertex-*** bzw. → ***Fragment-Programm***) und
- die Programme können parametrisiert werden (***Uniform Variablen***)

Wichtig: Lese- und Schreibposition in dem Stream sind fix!

Beispiel: Vertex-Prozessor / Vertex-Programm



Vertex-Prozessor

Aufgaben des Vertex-Prozessors/Vertex-Programms

- Vertex-Transformation, per-Vertex Beleuchtung
- Normalen-Transformation und Normalisierung der Normalen
- Generierung von Textur-Koordinaten

Varying (Stream-Input) Variablen: Position, Farbe, Normale, Textur-Koordinate

Fragment-Prozessor

Aufgaben: Verarbeitung der vom Rasterisierer interpolierten per-Vertex Daten (Farbe, Tiefe, Texture-Koordinaten) auf Fragment-Ebene

- Auslesen von Daten (Farben etc.) aus Texturen
- Textur-Funktion, also die Ermittlung der finalen Farbe aus Textur- und (interpolierter) Fragment-Intensität

Varying (Stream-Output) Variablen: Farbe, Tiefe

8.3 GPU-Hardware und Programmiersprachen

Historischer Abriß: Wesentliche Entwicklungsstufen

- 2001, GeForce 3: Erste programmierbare Vertex- und Fragmentprozessoren
- 2002, ATI Radeon 9700: Erste Schleifen und Fließkommazahlen
- 2006, Unified Shader Model: Generische Processing Units statt dedizierte Vertex- oder Fragment-Prozessoren
- 2007, NVIDIA CUDA (Compute Unified Device Architecture): Erste „Nicht-Graphik“ API für GPUs
- Aktuell: OpenCL (Open Compute Lang.), plattformübergreifend

Programmiersprachen: (nur die wichtigsten, „Graphik-lastigen“)

- NVIDIA’s „C for graphics“ (Cg), nur für NVIDIA Hardware
- Microsoft’s High Level Shading Language (HLSL), nur mit Direct3D
- ***OpenGL Shading Language (GLSL)***: Hardware- u. plattformübergreifend

Grundsätzlich:
- Alles sind C-ähnliche Sprachen mit sehr ähnlicher Syntax
- Im Folgenden: ***Shader*** als Synonym für GPU-Programm

8.3 GPU-Hardware und Programmiersprachen ...

Beispiel GPU-Hardware: NVIDIA GeForce GTX 280 (Juni 2008)

Anzahl Transistoren: 1.4 Millarden (1.400.000.000)

Gesamter Speicher: 1 GB

Taktraten: Core/Prozessor/Speicher 602 MHz/1296 MHz/2214 MHz

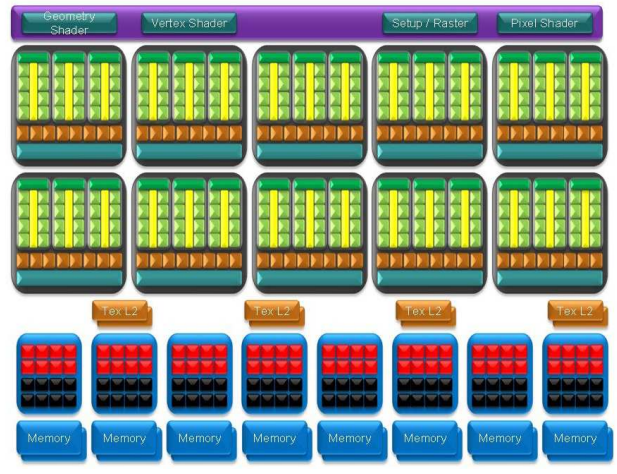
Pixel-Füllrate: 19,264 Giga-Pixel/sec (nur Rasterisierung)

Rechenleistung: 933 GFLOPS (Vergleich: Intel i7-965: 51.2 GFLOPS)

240 Prozessoren gruppiert in

- 10 Texture Processing Clusters (TPCs),
- jeder TPC hat 3 Streaming Multiprocessors (SMs) und
- jeder SM hat 8 Streaming Processors (SPs)

GeForce GTX 280 Graphics Processing Architecture



Performance Vorteil der GPU

Rechenleistung: Nominell ist die GPU 20-fach schneller als die CPU!

Ursache I: Massive Parallelität, konkret

- SIMD (Single Instructions Multiple Data): 4-fach Operationen pro Cycle:

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ w_1 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ w_2 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \\ z_1 + z_2 \\ w_1 + w_2 \end{bmatrix}$$

in einem Cycle (auch für `*`, `/`, `*=` etc.)!

- Viele, parallel arbeitende Prozessoren (s. NVIDIA GeForce GTX 280)

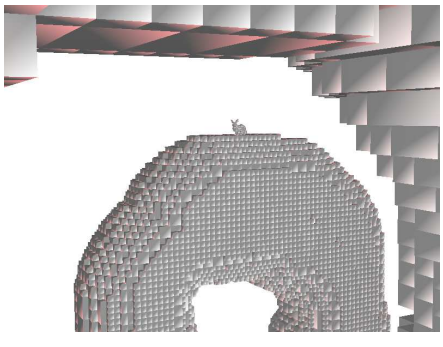
Ursache II: Restriktives Streamprozessor Programmiermodell, insbesondere fixe Lese- und Schreibposition im Stream

8.4 OpenGL Shading Language (GLSL)

Einschänkungen

- Hier werden nur die grundsätzlichen Funktionsweisen erläutert, zur Vertiefung wird auf die Literatur (insb. Rost: *OpenGL Shading Language*, Addison Wesley) verwiesen!
- Wir besprechen nur einfache Graphik-Beispiel, aber GPU-Programmierung ist ***viel mächtiger*** als Standard-Graphik-Programmierung!

Beispiel: Aktuelle Forschung am Siegener Graphik-Lehrstuhl



GPU-verwaltete, dynamische Voxelstruktur (M. Keller)

Mesh-Clustering auf der GPU (I. Chiosa)

Simulation von Gras inkl. Kollisionen (J. Orthmann)

OpenGL Extensions

OpenGL's GPU Funktionalität ist in vielen OpenGL-Extensions definiert bzw. realisiert

OpenGL-Extensions sind sehr dynamisch in ihrer Entwicklung

Je nach Hardware werden nicht alle Extensions unterstützt

Beispiel unter Linux mit Kommando `glxinfo` (Auszug)

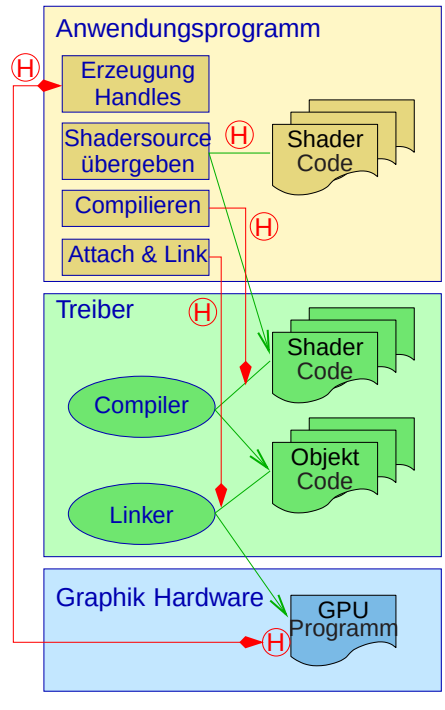
```
GLX version: 1.3
GLX extensions:
    GLX_EXT_visual_info, GLX_EXT_visual_rating, GLX_SGIX_fbconfig,
    GLX_SGIX_pbuffer, GLX_SGI_video_sync, GLX_SGI_swap_control,
    GLX_EXT_texture_from_pixmap, GLX_ARB_create_context, GLX_ARB_multisample,
OpenGL vendor string: NVIDIA Corporation
OpenGL renderer string: GeForce 7950 GX2/PCI/SSE2
OpenGL version string: 2.1.2 NVIDIA 180.44
OpenGL extensions:
    GL_ARB_color_buffer_float, GL_ARB_depth_texture, GL_ARB_draw_buffers,
    GL_ARB_fragment_program, GL_ARB_fragment_program_shadow,
    GL_ARB_fragment_shader, GL_ARB_half_float_pixel, GL_ARB_half_float_vertex,
```

GLEW-Bibliothek stellt die jeweils aktuellen Extensions zur Verfügung

Linux: `#include <GL/glew.h>`; Link: `-lGLEW`

Erzeugung und Aktivierung von GPU-Programmen

```
// Der Quellcode der GPU Programme
// VS = Vertex Shader, FS Fragment Shader
const char *vsCode = "...";
const char *fsCode = "...";

// Handles für GPU Programm, VS und FS
GLuint  pHandle, vsHandle, fsHandle;
pHandle  = glCreateProgram();
vsHandle = glCreateShader(GL_VERTEX_SHADER);
fsHandle = glCreateShader(GL_FRAGMENT_SHADER);

// Übergabe des Shader-Codes
glShaderSource(vsHandle, 1, &vsCode, NULL);
glShaderSource(fsHandle, 1, &fsCode, NULL);

// Übersetzen von Vertex- und Fragment Shaders
glCompileShader(vsHandle);
glCompileShader(fsHandle);

// Einhängen von VS und FS und linken
glAttachShader(pHandle, vsHandle);
glAttachShader(pHandle, fsHandle);
glLinkProgram(pHandle);

// Activiert GLSL Programm
glUseProgram(pHandle);
```



Varying Vertex-Shader (VS) Parameter

VS-Input-Stream: Einspeisen der Positions-, Normalen, TexCoord- und Color-Daten in den Vertexstream und Aufruf des aktiven VS

OpenGL-Befehl	`glVertex`	`glNormal`	`glColor`	`glTexCoord`
Varying VS-Param.	`gl_Vertex`	`gl_Normal`	`gl_Color`	`gl_MultiTexCoord`*X*
Datentyp	`vec4`	`vec3`	`vec4`	`vec4`

Textur-Koordinaten: Mehrere TexCoord per Vertex möglich (Standard `gl_MultiTexCoord0`)

VS-Output-Stream: Unterscheide ***built-in-*** und ***user-defined*** Parameter

- Built-In: `gl_Position` (vec4), `gl_FrontColor` (vec4), `gl_TexCoord[]` (vec4) in NDC
- User-defined: ***Identische Deklaration*** im Vertex- und Fragment-Shader

```
// Vertex Shader
varying vec3 normal; // per vertex normal
...
```

```
// Fragment Shader
varying vec3 normal; // interpol. normal
...
```

Varying Fragement-Shader (FS) Parameter

FS-Input-Stream: Varying output des VS werden interpoliert und sind varying input für den FS

- `gl_Position` (output VS) steht nicht zur Verfügung
- `gl_Color`,`gl_TexCoord[]`, und andere Varying-Var. (output VS) werden für Fragment (perspektivisch korrekt) interpoliert

FS-Output-Stream: `gl_FragColor` (vec4), `gl_Depth` (float)

Uniform Parameter

Parameter, die sich seltener ändern und nicht Teil des (Vertex-)Streams sind.

Built-in: Analog zu Stream-Parametern, sind aber für VS und FS zugreifbar

User-defined: Deklaration im Shader-Programm, ***location*** entspricht GPU-Speicheradresse, Wertzuweisung erfolgt in der Anwendung

```
// Shader (VS oder FS)
uniform vec2 myParam;
...
```

```
// C++-Code auf CPU
GLint loc = glGetUniformLocation(pHandle, "myParam");
glUniform2f(loc, 47.0, 11.0);
```

Überblick über built-in Uniform Parameter

Auszug von built-in Uniform Parameter:

`mat4 gl_ModelViewMatrix`	aktuelle $T_{\text{modelview}}$
`mat4 gl_ModelViewProjectionMatrix`	aktuelle $T_P \cdot T_{\text{modelview}}$
`mat4 gl_ProjectionMatrix`	aktuelle T_P
`mat4 gl_ModelViewMatrixInverseTranspose`	aktuelle $(T^{-1}_{\text{modelview}})^T$
`mat3 gl_NormalMatrix`	ebenfalls $(T^{-1}_{\text{modelview}})^T$
`vec4 gl_LightSource[i].ambient`	ambienter Anteil der i-ten Lichtquelle
`vec4 gl_LightSource[i].diffuse`	diffuserer Anteil der i-ten Lichtquelle
`vec4 gl_LightSource[i].specular`	spekularer Anteil der i-ten Lichtquelle
`vec4 gl_LightSource[i].position`	Positon der i-ten Lichtquelle
`vec3 gl_LightSource[i].spotDirection`	Spot-Richtung der i-ten Lichtquelle
`float gl_LightSource[i].spotExponent`	Spot-Exponent der i-ten Lichtquelle
`float gl_LightSource[i].spotCutoff`	Spot-Cutoff-Winkel der i-ten Lichtquelle
`vec4 gl_FrontMaterial.emission`	Eigenemission des Materials
`vec4 gl_FrontMaterial.ambient`	ambiente Materialkonstante
`vec4 gl_FrontMaterial.diffuse`	diffusee Materialkonstante
`vec4 gl_FrontMaterial.specular`	speculare Materialkonstante
`float gl_FrontMaterial.shininess`	Reflexionsexponent

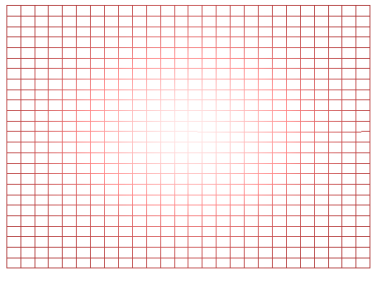
8.4.2 Geo-Morph

Ziel: Nutzung des VS zur „Mischung“ von Geometrie

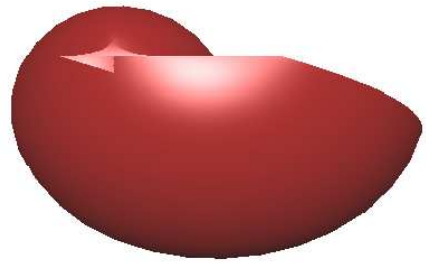
Konkretes Beispiel: Morphing zwischen einem Quadrat und einer Kugel

1. OpenGL-Anwendung erzeugt auf der CPU ein tesseliertes Quadrat
2. VS interpretiert Vertices als Parameter für Kugel-Parametrisierung
3. VS interpoliert zwischen den Punkten im Quadrat und auf der Kugel

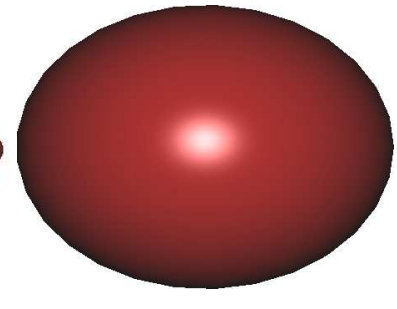
Ergebnis:



Ausgangsgitter



Morphing zwischen Ebene und Kugel



„fertige“ Kugel

Vertex-Shader Code

```
varying vec4 posVC; // vertex in VC
varying vec3 normVC;// normal in VC

uniform float time; // animation param.

const float M_PI = 3.14159;

// unit sphere: param in [-1,1]^2
vec4 sphere (vec2 param)
{
  vec4 result;
  float phi   = (param.x - 1.0) * M_PI;
  float theta = (param.y) * M_PI / 2.0;

  result.x = cos(phi)*cos(theta);
  result.y = sin(phi)*cos(theta);
  result.z = sin(theta);
  result.w = 1.0;

  return result;
}

void main()
{
  // sphere in MC (center in the origin)
  vec4 spherePos:
  vec3 sphereNorm;

  spherePos  = sphere(gl_Vertex.xy);
  sphereNorm = spherePos.xyz;

  // mix data (morphing); blend in [0,1]
  vec4 posMC;
  vec3 normMC;
  float blend;

  blend = abs(mod(time, 2.0) - 1.0);
  posMC =
    mix(gl_Vertex, spherePos, blend);

  normMC =
    mix(gl_Normal, sphereNorm, blend);
  normMC = normalize(normMC);

  // transform vertex and normal in VC
  normVC =
    normalize(gl_NormalMatrix * normMC);

  posVC =
    gl_ModelViewMatrix * posMC;

  // transform vertex in NDC (required)
  gl_Position =
    gl_ModelViewProjectionMatrix * posMC;
}
```

Geo-Morph Erläuterungen

User defined Varying: `posVC, normVC` für per-Fragment Beleuchtung (später!)

User defined Uniform: `time` zur Animation beim Morphing

Swizzling: Beliebiger Zugriff auf (eine oder mehrere) Komponenten

Beispiel: Sei `vec4 myVec = vec4(1,2,3,4)`, dann ist z.B.

```
myVec.x == float(1);          myVec.xy == vec2(1,2);
myVec.xyzw == vec3(1,2,3,4); myVec.wx == vec2(4,1);
myVec.yyxx == vec4(2,2,1,1); myVec.wwww == vec4(4,4,4,4);
```

`main`**-Funktion** als Einstiegspunkt (ohne Parameter!)

Unter-Funktionen wie in C/C++, allerdings keine Zeiger oder Referenzen

Funktionen arbeiten (fast) alle generisch auf `float, vec2, vec3, vec4`

- Standard mathematische Funktionen (`sin, cos, log, pow, exp ...`)
- Vektor-Funktionen: `dot` (Vektorprodukt), `cross` (Kreuzprodukt), `normalize` (Normalisierung), `reflect` (Reflexionsvektor)

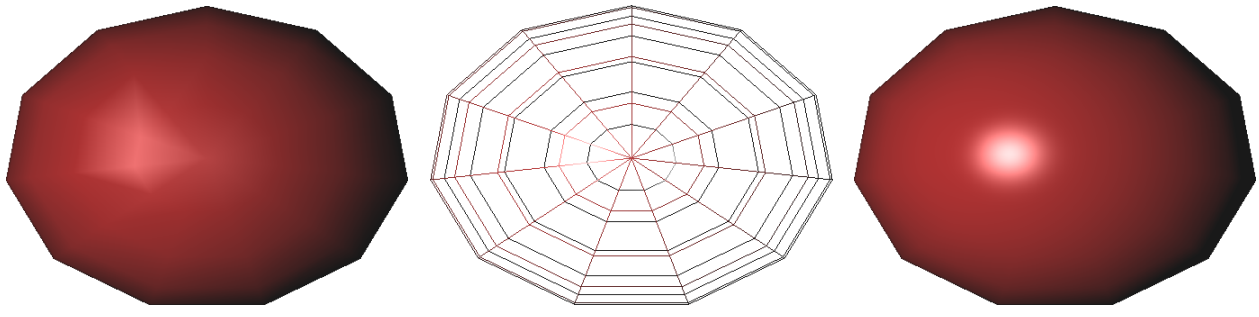
8.4.3 Per-Fragment Beleuchtung

Ziel: Ersetzen der per-Vertex Beleuchtung & Interpolation durch eine per-Fragment Beleuchtung

Vorgehen: Berechnung des Phong-Modells im FS

- ❍ Wesentliche Informationen liegen im FS schon vor (Licht, Material)
- ❍ Fehlt: Pro-Fragment Normale und -Position in View-Koordinaten (in NDC sind die Winkel durch die Projektion verändert!) Diese werden im VS erzeugt und an den FS als varying übergeben.

Ergebnis:



Per-Vertex Beleuchtung | Wireframe Ansicht | Per-Fragment Beleuchtung

8.4.3 Per-Fragment Beleuchtung ...

Fragment-Shader Code

```
varying vec4 posVC;       // fragment in VC
varying vec3 normVC;      // per fragment normal in VC

void main()
{
  // normal and other vectors in VC
  vec3 viewDir, reflectViewDir, lightDir;

  vec3 normal = normalize(normVC);  // may be distorted due to interpolation
  viewDir  = normalize(-posVC.xyz); // viewer is in origin!
  lightDir = normalize(gl_LightSource[0].position.xyz - posVC.xyz);
  reflectViewDir  = reflect(-viewDir, normal);

  // per fragment color according to the Phong model
  vec4 amb  = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
  vec4 diff = gl_FrontMaterial.diffuse *
    max(0.0, dot(normal, lightDir)) * gl_LightSource[0].diffuse;
  vec4 spec = gl_FrontMaterial.specular *
    pow(max(0.0, dot(reflectViewDir, lightDir)), gl_FrontMaterial.shininess) *
    gl_LightSource[0].specular;

  gl_FragColor = amb + diff + spec;  // fragment stream output
}
```

Hinweis: Das Geo-Morph Programm nutzt auch die per-Fragment Beleuchtung

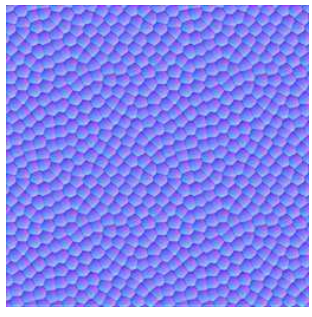
8.4.4 Normal Mapping

Idee: Interpretation von Texturdaten als Normalen im FS mit anschließender per-Fragment Beleuchtung

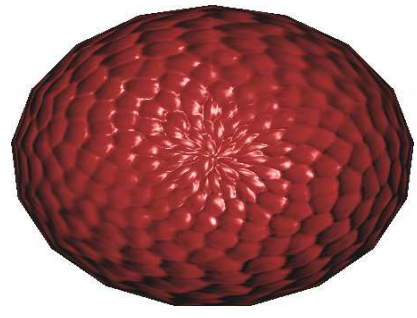
Vorgehen: 1. Speicherung der per-Fragment Normale in einer Textur
2. Auslesen der Textur im FS und entspr. „Verdrehen“ der Normalen
3. Einfache Nutzung der per-Fragment Beleuchtung

Beachte: Es werden keine Vertices verändert!

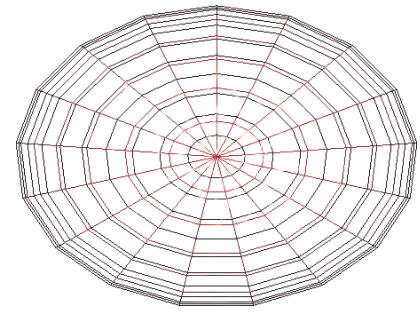
Ergebnis:



Normalen-Textur



Kugel mit Normal Mapping



Wireframe (keine Geometrieänderung)

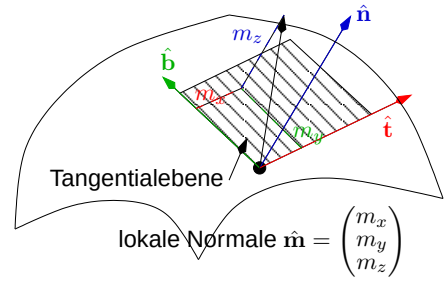
8.4.4 Normal Mapping ...

Fragment-Shader Code

```
varying vec4 posVC;       // fragment in VC
varying vec3 normVC;      // per fragment normal in VC
uniform sampler2D normalMap; // normal texture

void main()
{
  // normal and other vectors in VC
  vec3 viewDir, reflectViewDir, lightDir;

  // I. compute local coord. frame (tangent, binormal, normal)
  vec3 normal, tangent, binormal;

  normal   = normalize(normVC);
  tangent  = normalize(cross(normal, vec3(1,0,0)));
  binormal = cross(normal, tangent);

  // II. sample normal map and disturb normal in local coord. frame
  vec3 localNormal = texture2D(normalMap, gl_TexCoord[0].st).xyz;

  // transform values from [0,1] to [-1,1]
  localNormal = normalize((localNormal - 0.5) * 2.0); // SIMD calculus

  normal =
    localNormal[0]*tangent + localNormal[1]*binormal + localNormal[2]*normal;
  normal = normalize(normal);

  // standard phong
  (... skipped ...)

  gl_FragColor = amb + diff + spec;
}
```

Normal-Map Erläuterungen

Lokale Koordinaten: Normalen können nicht absolut in VC definiert sein, z.B. wegen wiederholter Textur-Koordinaten (`GL_REPEAT`)



Texturzugriff: ❍ Textur im Shader als `uniform sampler` deklarieren

❍ Anbinden der Textur im CPU-Programm

```
// activate 0-th texture unit and bind texture
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texId);

// connect 0th texture unit with sampler
GLint loc = glGetUniformLocation(g_pHandle, "normalMap");
glUniform1i(loc, 0);
```

❍ Auslesen der Texturdaten mit `texture2D(sampler, texCoord)`

Anpassung Wertebereich der Normalen-Textur: $[0,1]^2 \rightarrow [-1,1]^2$

`localNormal = normalize((localNormal - 0.5) * 2.0);` (SIMD!)

8.4.5 Weitere Beispiele

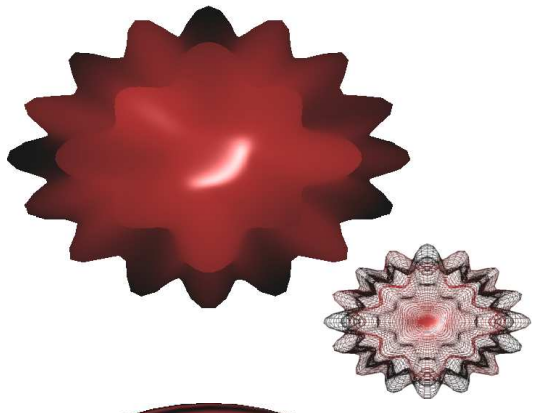
Animierte Geometrie

Idee: Veränderung der Geometrie im VS

Beispiel: Aussenden einer Sinus-Welle von einem Punkt im Raum

Geometrie-Anpassung sehr ähnlich zu Geo-Morph

Normalen-Anpassung ist schwieriger, da geometrische Nachbarschaftsinfo fehlt

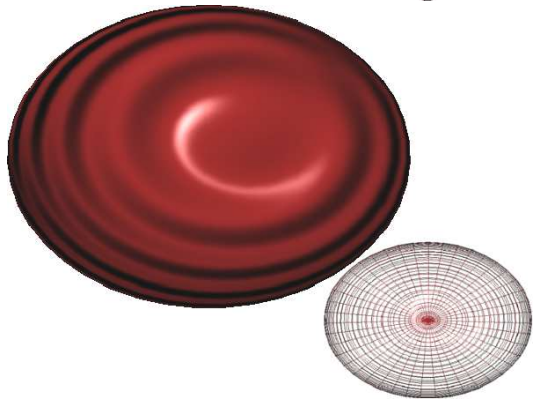


Prozedurales Bump-Mapping

Idee: (Dynamische) Änderung der Normalen per Prozedur statt per Textur im FS

Beispiel: Aussenden einer Sinus-Welle von einem Punkt im Raum

Normalen-Anpassung sehr ähnlich zu Normal Mapping



Framebuffer-Objects (FBO)

Bisher: Ergebnis des FS wird im Framebuffer gespeichert und angezeigt

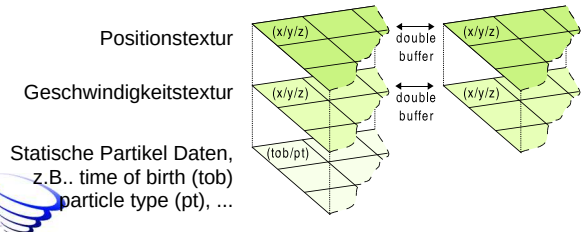
Jetzt: Speichern des Ergebnisses in einer Textur (***Framebuffer Object***)

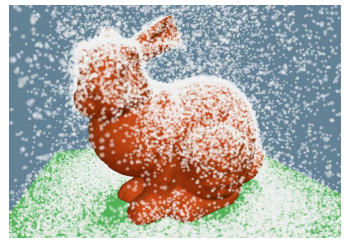
Dies führt zu einer ***erheblichen funktionalen Erweiterung***:

- „Render"-Ergebnis kann als beliebiges Datum interpretiert werden
- Erneute Nutzung der Daten in einem weiteren VS- oder FS-Pass

Beispiel Partikel-Systeme:

- Berechnung von Partikelparameter (Position, Geschwindigkeit, etc.) im FS und Speicherung in FBO
- Rendering der Partikel in einem zweiten Pass: VS liest Partikel-Textur und positioniert Partikel im Raum





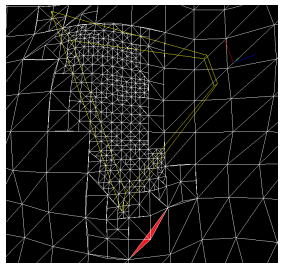
Geometry Prozessor und Geometry Shader

Bislang: Vertex- und Fragment-Prozessor verändern Stream-Struktur nicht

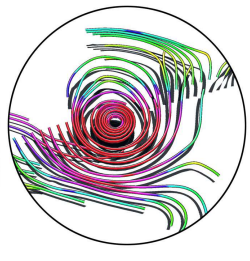
Jetzt: Geometry Prozessor kann Geometrie (konkret Vertices für das aktuelle Primitiv) ***erzeugen***

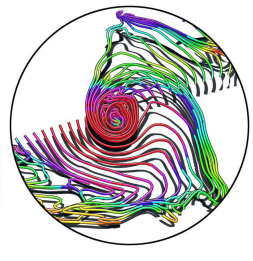
- Geometry Prozessor kommt nach dem Vertex Prozessor
- Geometry Prozessor kann Daten an Fragment Prozessor weiterleiten oder zurück in den Vertex Prozessor leiten (***Transform Feedback***)

Anwendungsbeispiel: Erzeugung adaptiver Geometrien z.B. für Terrain-Rendering oder für Subdivisionsflächen



Adaptives Terrainrendering





Adaptive Strömungslinien