Projektarbeit: GPU Programmierung

Martin Christen, martin.christen@stud.fhbb.ch
Tobias Egartner, t.egartner@bluewin.ch
Pietro Zanoni, zanonip@gmx.ch

Betreuer: M. Hudritsch

Inhaltsverzeichnis

1. Ziele dieser Projektarbeit

1. Grundlagen

- Einarbeitung in die verschiedenen High Level Shader Sprachen mit Schwerpunkt OGLSL.
- Wir entwicklen unsere eigene Umgebung für die Enwicklung von GPU Programmen, welche auch für nichtgraphische Zwecke eingesetzt werden kann. Wir unterstützen dabei die OGLSL, aber auch die ARB_vertex_program/ARB_fragment_program Extensions für die Assemblerprogrammierung.
- Laden und Verwenden von GPU Programmen, welche wir in einer High Level Programmiersprache schreiben. Dazu verwenden wir in erster Linie OpenGL Shading Language. Wir schreiben C++ Klassen, welche die wesentlichen Operationen und Hilfsfunktionen für die GPU Programmierung mit OGLSL enthält.
    - Laden eines Vertex bzw. Fragment Programmes. (Programme sind Objekte welche mit Konstruktoren und Destruktoren geladen (und compiliert und gelinkt) resp. freigegeben werden.)*[zu beachten ist, dass ein Programm Objekt aus mehreren Programmen bestehen kann (z.B. Vertex+Fragment Programm)]*
    - Aktivieren/deaktivieren eines GPU Programmes (program.begin(); und program.end();)
    - Parameterübergabe vom C++ Programm zum Shader Programm.
    - Laden von OpenGL Extensions und erhalten von OpenGL Informationen.
- Erstellen einer IDE für die OpenGL Shading Language Programmierung.
- Sammlung einfacher Vertex Programme.
- Erweiterung des Szenengraphen

2. Beleuchtung

- Integration und Kompatibiliät zu OpenGL.
- Per Pixel Blinn und Phong Beleuchtung.
- Bumpmapping
- Cook Torrance Beleuchtungsmodell

3. Prozedurales Texturing und Modeling mit OpenGL Shading Language

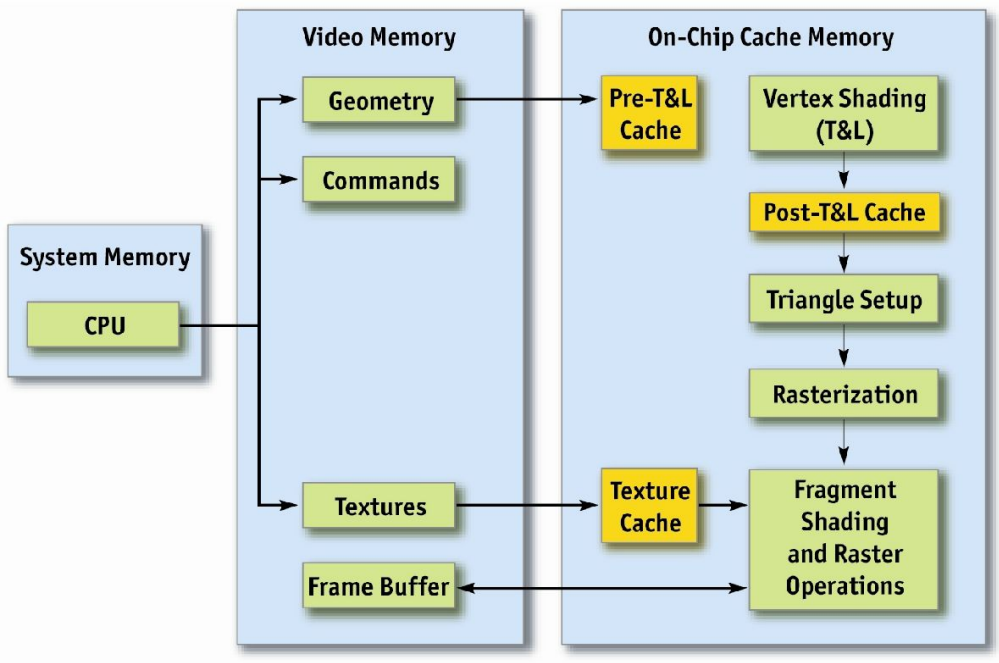
- Generation von Objekten unter Verwendung physikalischer Modelle
- Generation von Texturen.
- Volumentexturen (Noise)

4. Nichtgraphische Anwendungen

- GPU Programme für die Bildverarbeitung
- Nichtgraphische GPU Programme

2. Grundlagen der Shaderprogrammierung

2.1. Einführung: Wie läuft ein GPU Programm ab ?



Speicherlayout CPU – GPU *[20]*

1. Texturen werden an die Grafikkarte gesendet.

2. Vertex Daten (Position, Normalen, Farbe etc.) werden übermittelt.

3. Der Vertex Shader, den wir programmieren können, erhält diese Daten. Dort können wir z.B. Vertices von Objektkoordinaten in Welt-Koordinaten transformieren.

4. Der Fragment Shader, den wir auch programmieren können, berechnet Farbwerte jedes Pixels der Dreiecke. Wie die Farbwerte berechnet werden, ist uns überlassen!

Zu jedem Vertex Shader (Geometrie der Ecken) gehört ein entsprechender Fragment Shader (Farbe pro Pixel).

Aufgaben Vertex Shader	*Aufgaben Fragment Shader*
• **Kontrolle über die Position von Vertices** • Benutzerdefiniertes Lighting (per Vertex) • Benutzerdefiniertes Skinning/Blending • Beliebige Vertexberechnung (z.B. Fischauge) • Beschleuningung komplexer Vertex Operationen, Partikelsysteme.	• **Farbe pro Pixel berechnen** • Texturierung • Benutzerdefiniertes Lighting (per Fragment) • Nebel • u.v.m.

2.2. Minimales Vertex Programm

Eingabe: Beliebige Vertex Attribute, ev. "globale" Variablen
Ausgabe: Transformierte Vertex Attribute:

- **erforderlich:** Position in Clip Koordinaten (homogen)
- Farbe (front/back, primary/secondary)
- Fog Koordinaten
- Textur Koordinaten
- Point Size

Ein Vertex Programm kann keine Vertices generieren oder zerstören!
Standardmässig sind keine topologischen Informationen vorhanden! (Kanten, Nachbar...)

Beispiel: Minimales Vertex Programm GLSL
Ein Vertex wird von Objekt Koordinaten in Clip Koordinaten transformiert.

```
void main(void)
{
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

2.3. Minimales Fragment Programm

Eingabe: Interpolierte Daten vom Vertex Shader, ev. "globale" Variablen
Ausgabe: Farbwert Pixel

Beispiel: Minimales Fragment Programm GLSL
Jeder Pixel wird rot.

```
void main(void)
{
        gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

2.4. Vertex Shader Version 1.x und 2.x - Unterschiede

Version 1: DirectX 8.1 und GL_ARB_vertex_program / GL_ARB_fragment_program
Version 2: DirectX 9 und OpenGL Shading Language

- Flusskontrolle mit Konstanten und nicht per-vertex Attributen!
- Vergleiche (if, else-Verzweigungen)
- Repetition (do...while),
  Es gibt jedoch noch kein echtes while auf der Hardware, sondern vom Compiler durch Repetition des Codes erzeugt, was natürlich mit der begrenzten Anzahl Instruktionen Konflikte gibt.
- Unterprogramme

2.5. Shader Version 3.0

Grafikkarten mit Shader Version 3.0 Support waren zur Zeit unserer Projektarbeit für uns noch nicht verfügbar (die ersten Karten kamen Anfang Juni auf den amerikanischen Markt).
Die Karten dieser Generation werden echte Repetition und if Statements unterstützen. NVidia hat in ihrem GPU Entwickler Dokument *[20]* im Juli 2004 das Shader Model 3.0 aufgenommen:

Pixel Shader

Pixel Shader Feature	*Shader 2.0*	*Shader 3.0*	*Beschreibung*
Shader Length	96	65535+	Erlaubt komplexeres Shading, Lighting und Prozedurale Materialien.
Dynamic Branching	No	Yes	Erhöht Performance durch auslassen von irrelevanten Pixeln bei komplexem Shading.
Shader Antialiasing	not supported	Built-in derivate instructions	Entwickler können auf die Farbunterschiede im Screenspace zugreifen.
Back-face register	No	Yes	Erlaubt beidseitige Beleuchgung (Front/Back) in einem einzigen Pass.
Interpolated Color Format	8 bit integer minimum	32-bit floating point minimum	Grösserer Bereich und Farb-Präzision ermöglicht HDR Beleuchtung auf Vertex Level.
Multiple Render Targets	optionel	4 required	Filter und Vertex Arbeit der Algorithmen von erweiterten Beleuchtungsmodellen wird optimiert.
Fog and specular	8-Bit fixed function minimum	Custom fp16-fp32 shader program	Das Shader Modell 3.0 ermöglicht volle und präzisere Kontrolle über Specular und Fog Berechnungen.
Texture Coordinate Count	8	10	Mehr per-pixel input!

Vertex Shader

Vertex Shader Feature	*Shader 2.0*	*Shader 3.0*	*Beschreibung*
Shader Length	256	65535	Mehr Instruktionen im Vertex Shader ermöglicht z.B. detailiertere Beleuchtung.
Dynamic Branching	No	Yes	Auslassen von irrelevanten Vertex Berechnungen erhöhen Performance!
Vertex Texture	No	Unbegrenzte Anzahl Lookups von bis zu 4 Texturen!	Displacement Mapping und Partikel Effekte.
Instancing support	No	required	Ein Objekt kann mehrmals (an anderer Position) gezeichnen werden, jedoch mit unterschiedlicherr Vertex Positionen/Farben etc.

2.6. Shading- und GPU Programmiersprachen

RenderMan
Pixar, 1988
RenderMan werden wir mit der frei erhältichen OpenSource Lösungen 'Aqsis' und 'K3D' betrachten. Es handelt sich dabei ein zu RenderMan kompatibles Paket.
http://aqsis.sourceforge.net
http://k3d.sourceforge.net

PixelFlow Shading Language
UNC, 1998

Stanford Real-Time Shading Language
Stanford, 2001

Cg
C for Graphics von NVIDIA.
Cg kann auch für OpenGL Anwendungen verwendet werden, allerdings nur ARB 1.x Shader und nicht fortgeschrittene (2.x) Shader. Diese werden zwar auch unterstützt, funktionieren jedoch nur auf NVIDIA Karten (z.B. GeForce FX).
Unter DirectX 9 können auch Karten anderer Hersteller verwendet werden.

HLSL
Die High Level Shader Language von Microsoft ist ein Teil von DirectX 9. Der Nachteil der HLSL ist die doppelte Abhängigkeit: Platform (Windows) und Grafik-Bibliothek (DirectX) können nicht geändert werden. Diese Einschränkung ist für unsere Zwecke weniger geeignet, da wir möglichst Platformunabhängig bleiben wollen. Die HLSL Syntax ist im wesentlichen identisch mit Cg.

OpenGL Shading Language ("OGLSL")
GLSL ist der offizielle Standard für Shader Programmierung unter OpenGL.
Implementation sind z.Z erhältlich von 3D Labs und ATI.
Wir werden die meisten Programme mit der OGLSL schreiben, da wir die OGLSL als die High Level Shading Language mit dem grössten Potential sehen.

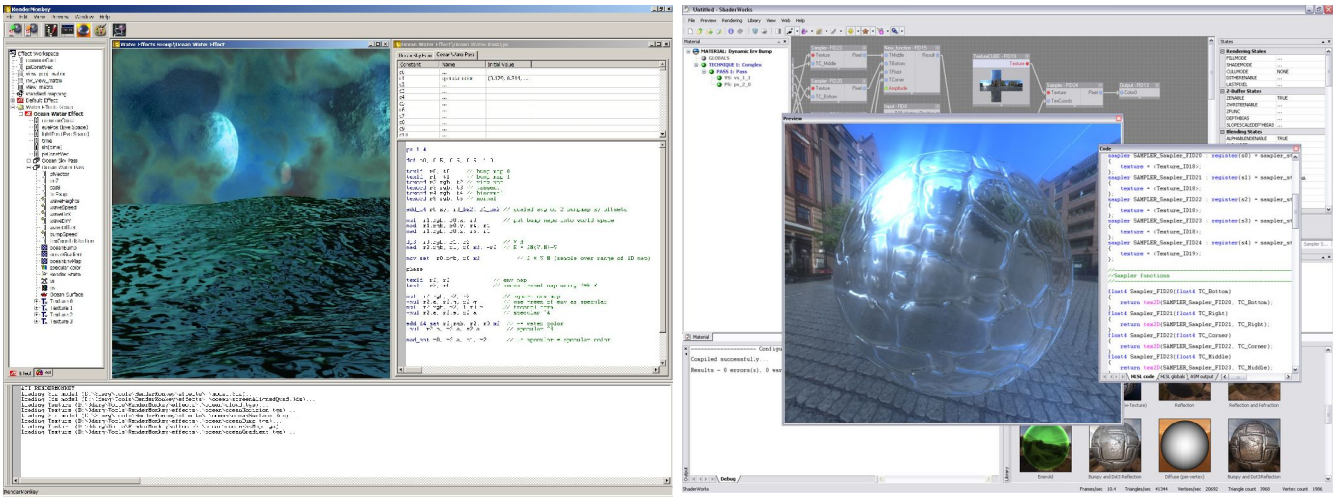
BrookGPU
BrookGPU ist eine Compiler und Runtime Implementation der 'Brook stream programming langugage', welche 'General Purpose GPU Programmierung' anstrebt. Brook GPU wird am Stanford University Graphics Lab entwickelt. BrookGPU verwendet intern Cg und HLSL.
http://graphics.stanford.edu/projects/brookgpu/

2.7. Tools für die Shaderprogrammierung

Mittlerweile gibt es neben den High Level-Programmiersprachen auch Entwicklungsumgebungen für Shader. Eines davon ist RenderMonkey von ATI, wir werden dieses Tool noch genauer betrachten.

Ein anderes Werkzeug für die Shader-Entwicklung ist "ShaderWorks" (http://www.shaderworks.com). Es ist ein kommerzielles Tool, welches vor allem für Spielentwickler gedacht ist. Im Dezember erhielten wir eine Alpha Version zum Testen.

Auch 3D Programme wie Maya und 3D Studio verwenden mittlerweile Schnittstellen zur Shaderprogrammierung.
Erwähnenswert ist an dieser Stelle auch ASHLI (Advanced Shading Language Interface), ein Tool von ATI welches RenderMan Shader (*.sl) in Vertex/Fragment Programme (HLSL oder OpenGL Shading Language) übersetzt.



RenderMonkey *(links)* und ShaderWorks *(rechts)*

NVidia hat "FX Composer" veröffentlicht, ein weiteres Tool für die Shaderprogrammierung. Es basiert auf Cg / HLSL und kann nicht für die OpenGL Shading Language eingesetzt werden.

Wir haben eine Beta Version von Render Monkey Version 1.5 erhalten, welches die OpenGL Shading Language und die Microsoft High Level Shading Language unterstützt. RenderMonkey ist sehr gut geeignet um in die Welt der Shaderprogrammierung einzusteigen.

3. Ziele dieser Projektarbeit

Teil I: Grundlagen

- Einarbeitung in die verschiedenen High Level Shader Sprachen mit Schwerpunkt GLSL.
- Wir entwicklen unsere eigene Umgebung für die Enwicklung von GPU Programmen, welche auch für nichtgraphische Zwecke eingesetzt werden kann. Wir unterstützen dabei die GLSL, aber auch die ARB_vertex_program/ARB_fragment_program Extensions für die Assemblerprogrammierung.
- Laden und Verwenden von GPU Programmen, welche wir in einer High Level Programmiersprache schreiben. Dazu verwenden wir in erster Linie OpenGL Shading Language. Wir schreiben C++ Klassen, welche die wesentlichen Operationen und Hilfsfunktionen für die GPU Programmierung mit GLSL enthält.
    - Laden eines Vertex bzw. Fragment Programmes. (Programme sind Objekte welche mit Konstruktoren und Destruktoren geladen (und compiliert und gelinkt) resp. freigegeben werden.)*[zu beachten ist, dass ein Programm Objekt aus mehreren Programmen bestehen kann (z.B. Vertex+Fragment Programm)]*
    - Aktivieren/deaktivieren eines GPU Programmes (program.begin(); und program.end();)
    - Parameterübergabe vom C++ Programm zum Shader Programm.
    - Laden von OpenGL Extensions und erhalten von OpenGL Informationen.
- Erstellen einer IDE für die OpenGL Shading Language Programmierung.
- Sammlung einfacher Vertex Programme.
- Erweiterung des Szenengraphen

Teil II: Beleuchtung

- Volle Integration und Kompatibiliät zu OpenGL.
- Per Pixel Blinn und Phong Beleuchtung.
- Bumpmapping
- Cook Torrance Beleuchtungsmodell

Teil III: Prozedurales Texturing und Modeling mit OpenGL Shading Language

- Generation von Objekten unter Verwendung physikalischer Modelle
- Generation von Texturen.
    - RenderMan -> OpenGL Shading Language
- Volumentexturen (Noise)

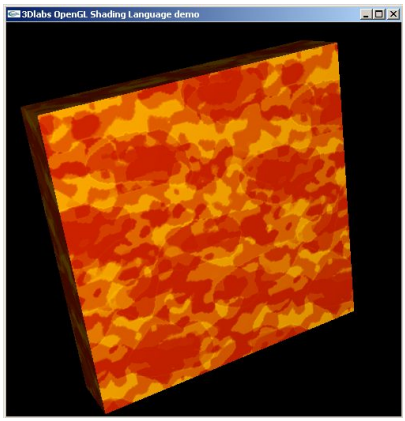
Teil IV: Nichtgraphische Anwendungen

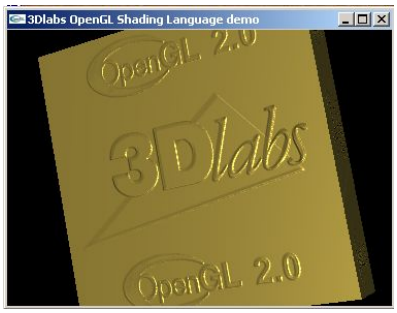
- GPU Programme für die Bildverarbeitung
- Nichtgraphische GPU Programme

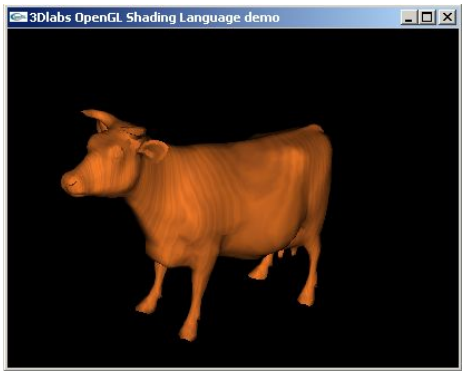
4. OpenGL Shading Language

4.1. Aktueller Stand von GLSL

Die OpenGL Shading Language ist offizieller ARB Standard. 3DLabs, ATI und NVidia hatten uns während unserer Arbeit zahlreiche Beta Versionen ihrer Treiber mit GLSL Support zur Verfügung gestellt.







(Lava Effekte mit Alpha-Blending, Bump Mapping und generierte Holz Textur.)

Beispiel: ProzeduraleHolz Textur

Vertex Programm:

```
varying float lightIntensity;
varying vec3 Position;
uniform vec3 LightPosition;
uniform float Scale;

void main(void)
{
 vec4 pos = gl_ModelViewMatrix * gl_Vertex;
 Position = vec3(gl_Vertex) * Scale;
 vec3 tnorm = normalize(gl_NormalMatrix * gl_Normal);
 lightIntensity = max(dot(normalize(LightPosition - vec3(pos)),
tnorm), 0.0) * 1.5;

 gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment Programm:

```
void main (void)
{
    vec3 location = Position;

    vec3 floorvec = vec3(floor(Position.x * 10.0), 0.0, floor
(Position.z * 10.0));
    vec3 noise = Position * 10.0 - floorvec - 0.5;
    noise *= noise;
    location += noise * 0.12;

    float dist = location.x * location.x + location.z *
location.z;
    float grain = dist * GrainSizeRecip;

    float brightness = fract(grain);
    if (brightness > 0.5)
        brightness = (1.0 - brightness);
    vec3 color = DarkColor + brightness * (colorSpread);

    brightness = fract(grain*7.0);
    if (brightness > 0.5)
        brightness = 1.0 - brightness;
    color -= brightness * colorSpread;

    color = clamp(color * lightIntensity, 0.0, 1.0);

    gl_FragColor = vec4(color, 1.0);
}
```

4.2. Ziele der OpenGL Shading Language

Das ARB hatte folgende Vorstellungen bei der Definition von GLSL, hier die wichtigsten Punkte:

1. OGLSL soll eine Sprache sein, welche an OpenGL angepasst ist:
   - Totale Kontrolle über die Pipeline
   - Innerhalb eines OGLSL Programmes soll es möglich sein die GL Zustände abzufragen
2. Hardwareprogrammierbarkeit
3. Hardwareunabhängigkeit
   - Portabilität soll gewährleistet werden (verschiedene Platformen + Grafikchips)
4. Die Sprache soll einfach sein.
   - Vertex und Fragment Programme benutzen die gleiche Sprache
   - Auf C basierend mit Vektor und Matrizen-Typen
   - Einige RenderMan (Pixar) Funktionen.
5. Die Sprache soll langfristig einsetzbar sein
   - **Programme die heute geschrieben werden, sollen auch in 10 Jahren noch laufen.**

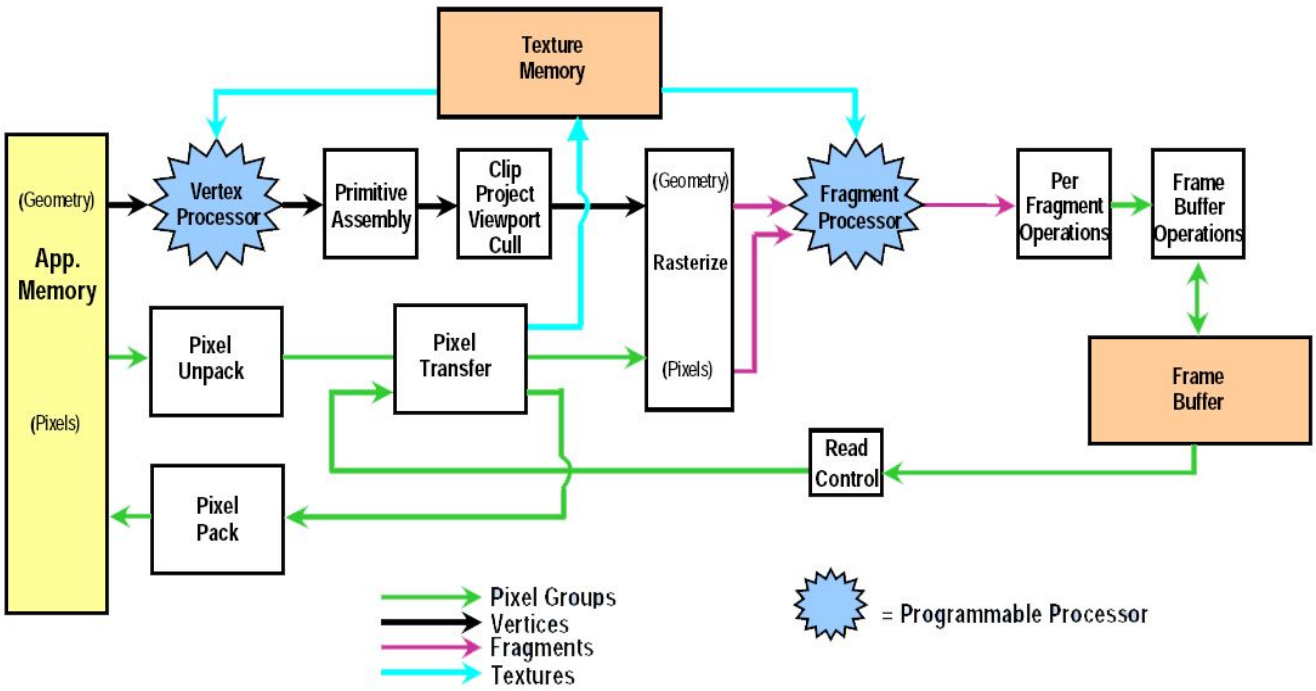
4.3. Unterschied zu DirectX HLSL

Bei HLSL compiliert man den C-ähnlichen Source ausserhalb von DirectX. Während die Applikation mit Direct3D läuft, wird Assemblercode übergeben (gleicher Assemblercode für verschiedene Hardware!!) Dies führt langfristig zu Inkompatibilität. Bei der CPU Programmierung sieht man es schön: Der Hochsprachen-Quellcode und nicht der Assemblercode hat verschiedede CPU Generationen überlebt.

Bei der OGLSL wird nur der C-ähnliche Source übergeben. **Der Compiler wird von dem Hersteller der Grafikkarte entwicklelt**. Schliesslich kennt dieser auch die Grafikkarte am besten... Dies funktioniert natürlich nur, wenn die Grafikkarten Hersteller kooperieren, und dies ist zur Zeit bei den meisten Herstellern der Fall (z.B. ATI, NVidia, 3DLabs).

Ein weiteres Problem ist, wenn man mit DirectX programmiert, so ist man abhängig von der Vision von Microsoft, im Gegensatz zu OpenGL, bei der das Architectural Review Board dahinter steht und somit nicht die Interessen einer einzelnen Firma vertreten kann.

4.4. GLSL Vertex und Fragment Processors



(Quelle: [15])

4.4.1. Vertex Processor

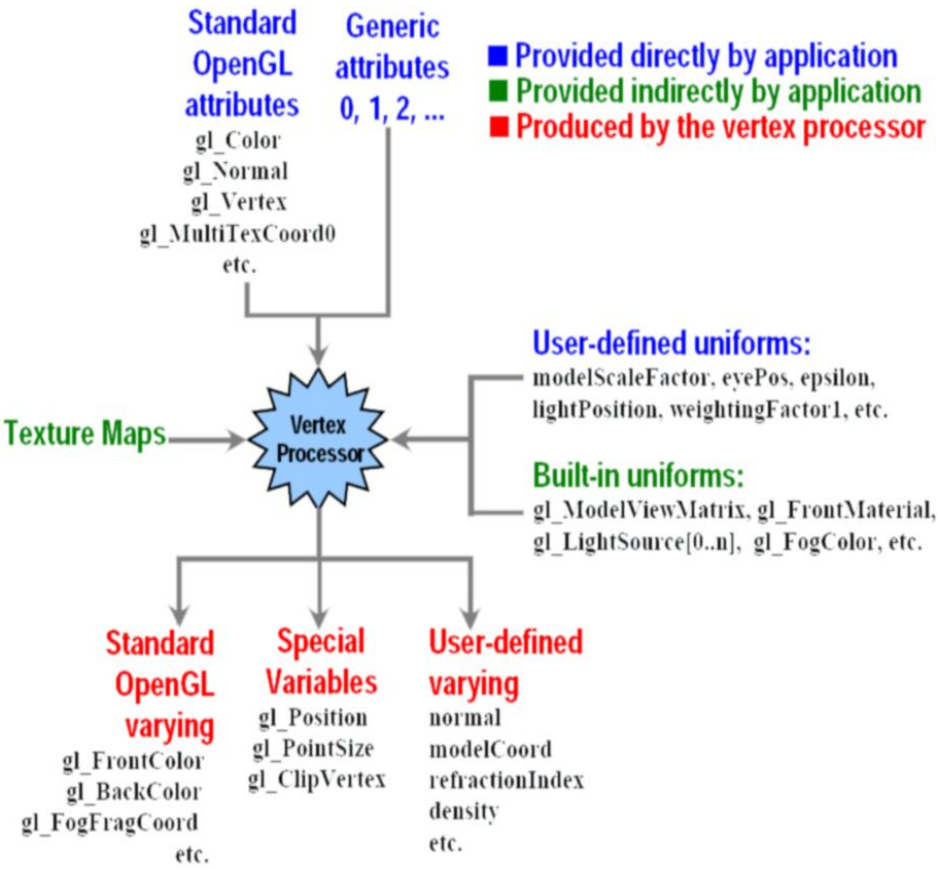
Flexibilität bei: Beleuchtung, Material und Geometrie

Der Vertex Processor ersetzt:

Vertex Transformation
Normalen Transformation, Normalisierung und Rescaling
Beleuchtung
Anwendung des Color Material
Clamping von Farben
Generierung von Textur-Koordinaten
Transformation von Textur-Koordinaten

Der Vertex Processor ersetzt nicht:

Perspektivische Division
Frustum um User Clipping
Backface Culling
Primitive Assembly
Two-Sided lighting selection
Polygon offset
Polygon mode

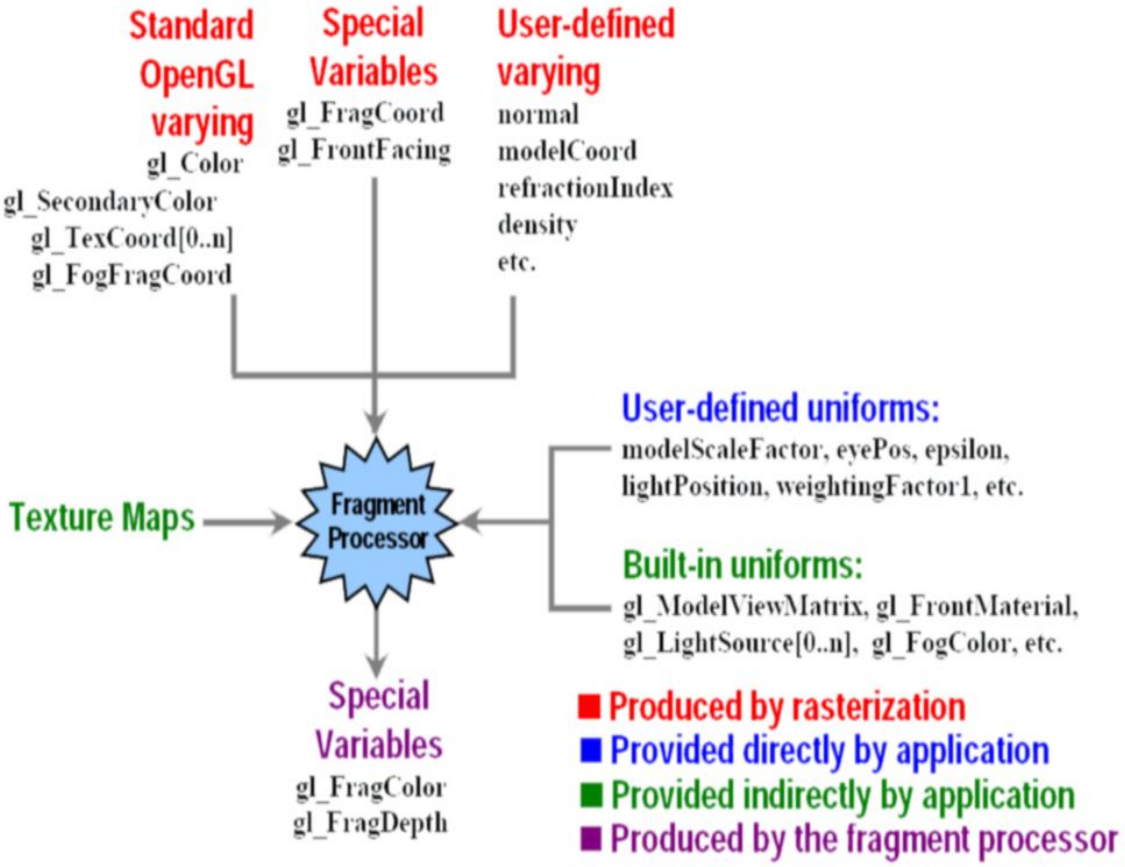


4.4.2. Fragment Processor

Flexibilität bei: Texturierung und per-Pixel Operationen
Der Fragment Processor kann verwendet werden für:
Color Matrix, Skalierung, Funktionstablle: f(x,y) auf Textur.

Der Fragment Processor ersetzt:

Operationen auf interpolierte Werte
Zugang zu Textur
Textur-Anwendung
Fog
Color sum



4.5. GLSL – Ein Überblick

4.5.1. Datentypen

void	für Funktionen, welche keinen Wert zurückgeben
bool	Vergleichs-Typ, kann 'true' oder 'false' sein.
int	Integer mit Vorzeichen
float	floating point Zahl
vec2, vec3, vec4	Vektor-Typ mit 2,3 oder 4 float Komponenten
bvec2, bvec3, bvec4	Vektor-Typ mit 2,3 oder 4 bool Komponenten
ivec2, ivec3, ivec4	Vektor-Typ mit 2,3 oder 4 integer Komponenten
mat2, mat3, mat4	2x2, 3x3 oder 4x4 Matrix (float)
sampler1D, sampler2D, sampler3D	Handle um auf eine 1D, 2D oder 3D Textur zuzugreifen
samplerCube	Handle, um auf eine Textur für Cube-Mapping zuzugreifen
sampler1DShadow, sampler2DShadow	Handle um auf eine 1D oder 2D Depth Textur mit Vergleich zuzugreifen

4.5.2. Strukturen

Strukturen werden analog wie in C definiert:

```
struct myStruct
{
        float a;
        vec3 b;
};
```

4.5.3. Arrays

Arrays können nicht direkt initialisiert werden.

Beispiel:

```
float test[3];
```

4.5.4. Typen Qualifizierer

const

Const Variablen sind für konstante Werte und müssen bei der Deklaration initialisiert werden.

attribute

Attribute werden für die (einseitige) per-Vertex Kommunikation eingesetzt, jedem Vertex kann ein anderer Wert zugewiesen werden. Der Wert kann innerhalb eines glBegin(...) und glEnd() ändern.

- Zugriff ist Read-Only
- Kann vordefiniert sein: Standard OpenGL Vertex Attribute sind z.B. gl_Color, gl_Normal, gl_Vertex, gl_TexCoord.
- Kann benutzerdefiniert sein

Beispiel:

```
attribute vec3        myCoolAttribute;
```

uniform

Uniform Variablen werden für die einseitige Kommunikation zwischen C++ und Shaderprogramm verwendet. Der Wert kann innerhalb eines OpenGL glBegin(...) und glEnd() nicht verändert werden.

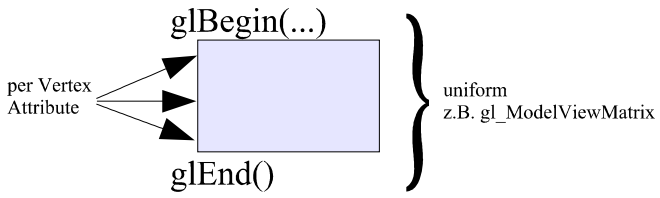
- Zugriff ist Read-Only
- Globale Variable für das Shaderprogramm
- Kann vordefiniert sein: gl_ModelViewProjectionMatrix, gl_FogColor, ...
- Kann benutzerdefiniert, z.B. Zeit, Grundfarbe, ...

Beispiel:

```
uniform float time;
```

varying

- Zugriff ist Read/Write
- Varying Variablen werden entlang einer Primitive korrekt interpoliert.
- Kann vordefiniert sein: gl_TexCoord[0], gl_FrontColor
- Kann beutzerdefiniert sein: z.B. Normale, Lichtintensität, ...



4.6. Vordefinierte Variablen

4.6.1. Matrizen

gl_ModelViewMatrix
Die ModelviewMatrix fasst die Model- und Transformationsmatrix zusammen.
Die gl_ModelViewMatrix entspricht der **aktuellen** Modelview Matrix, welche in OpenGL mit glGetFloatv(GL_MODELVIEW_MATRIX, m) ausgelesen werden kann.
Ein Vektor mit gl_ModelViewMatrix multplizieren entspricht der Transformation von Objekt-Koordinaten zu Kamera-Koordinaten:

```
v_camera = gl_ModelViewMatrix * v_object;
```

gl_ProjectionMatrix
Diese Matrix enspricht der aktuellen Projektions Matrix, welche in OpenGL mit glGetFloatv (GL_PROJECTION_MATRIX, p) ausgelesen werden kann.

Ein Vektor mit gl_ProjectionMatrix multiplizieren entspricht der Transformation von Kamera-Koordinaten zu Clip-Koordinaten:

```
v_clip =  gl_ProjectionMatrix * v_camera;
```

gl_ModelViewProjectionMatrix
Die ModelViewProjetions-Matrix ist die Matrix, welche aus der Multiplikation von der ModelView mit der Projektions Matrix ensteht.

```
gl_ModelViewProjectionMatrix = gl_ProjectionMatrix * gl_ModelViewMatrix;
```

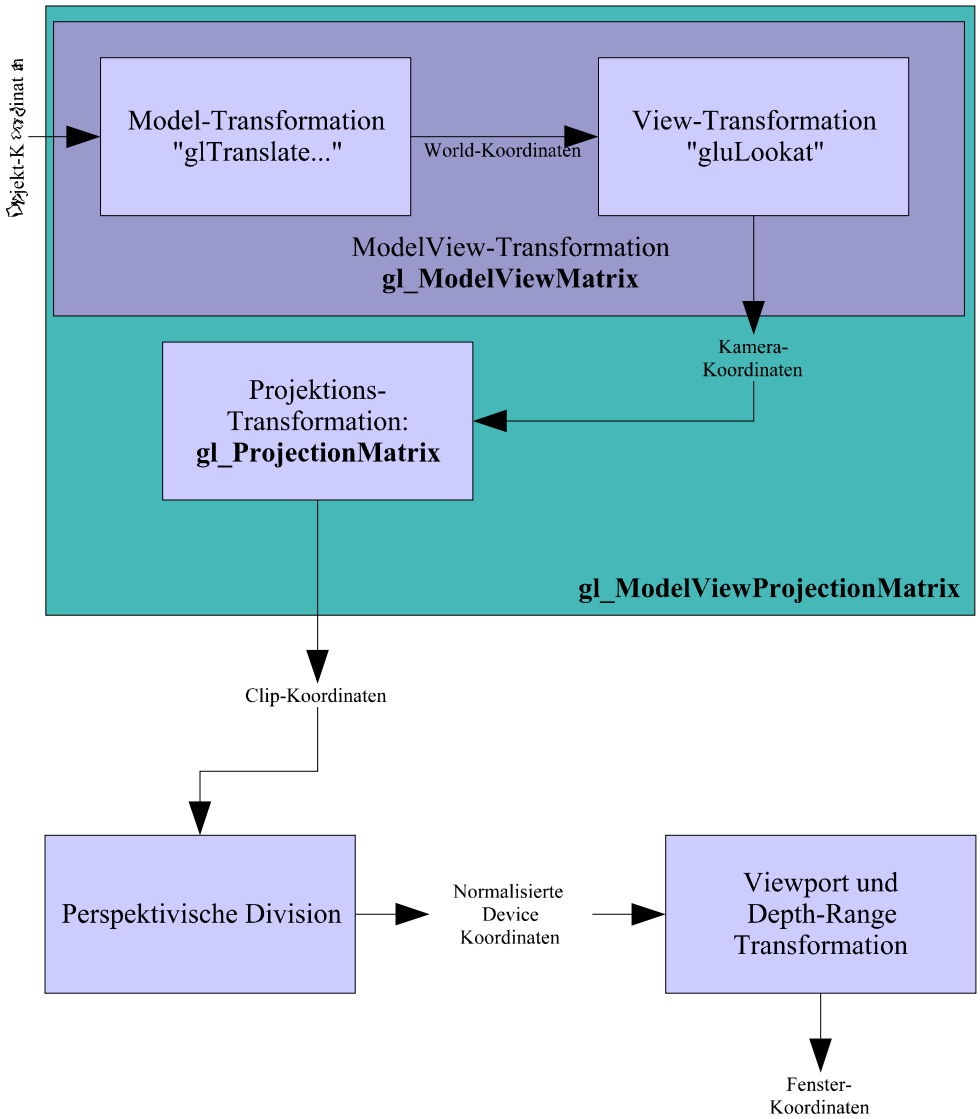
Ein Vektor mit gl_ModelViewProjectionMatrix mulitplizieren entspricht der Transformation von Objekt-Koordinaten zu Clip-Koordinaten:

```
v_clip =  gl_ModelViewProjectionMatrix * v_object;
```

gl_NormalMatrix
Die gl_NormalMatrix ist der obere 3x3 Teil der inversen ModelView Matrix. Werden Normalen mit dieser multipliziert, so enspricht das glEnable(GL_NORMALIZE).

gl_TextureMatrix[i]
Die gl_Texture Matrix, ist die Texturmatrix, welche verwendet wird, um Texturkoordinaten zu transformieren. Es gibt maximal gl_MaxTextureCoordsARB Textur-Matrizen.

Koordinatensysteme und Transformationen beim Vertex-Processing:



Objekt-Koordinaten: gl_Vertex := $\begin{pmatrix} x_{object} \\ y_{object} \\ z_{object} \\ 1.0 \end{pmatrix}$

Clip-Koordinaten: $\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix}$ = gl_ModelViewProjectionMatrix * gl_Vertex

Transformation von Normalen

Gegeben:
Ebene E : $\vec{t} \cdot \vec{n} + d = 0$ mit $\vec{n} = (n_x, n_y, n_z, n_w)^T$ und $\vec{t} = (x, y, z, w)^T$ und $d \in \mathbb{R}$
ModelView Matrix M wobei $M = M^T_{OpenGL}$

Behauptung:
Normalenvektoren werden nicht mit der ModelView Matrix M transformiert, sondern mit der Transponierten der Inversen ModelView Matrix: $(M^{-1})^T$.

Beweis:

$$\vec{t} \cdot \vec{n} \;=\; \vec{t} \cdot I \cdot \vec{n} \;=\; \vec{t} \cdot (M \cdot M^{-1}) \cdot \vec{n} \;=\; (\vec{t} \cdot M) \cdot (M^{-1} \cdot \vec{n}) \;=\; \vec{t}\,' \cdot (\vec{n} \cdot (M^{-1})^T)$$

4.6.2. Vertex Shader

Spezielle Variablen

Position des Vertex, **muss** geschrieben werden, also irgendwo im Vertex Shader müssen dieser Variable Werte zugewiesen werden.

```
vec4  gl_Position
```

Grösse des Punktes (in Pixeln) des zu rasterisierenden Punktes (optional)

```
float  gl_PointSize
```

Benutzerdefiniertes Clipping (optional)

```
vec4 gl_ClipVertex
```

Vordefinierte Attribute:

```
attribute vec4 gl_Color;
attribute vec4 gl_SecondaryColor;
attribute vec3 gl_Normal;
attribute vec4 gl_Vertex;
attribute vec4 gl_MultiTexCoord0;
attribute vec4 gl_MultiTexCoord1;
attribute vec4 gl_MultiTexCoord2;
attribute vec4 gl_MultiTexCoord3;
attribute vec4 gl_MultiTexCoord4;
attribute vec4 gl_MultiTexCoord5;
attribute vec4 gl_MultiTexCoord6;
attribute vec4 gl_MultiTexCoord7;
attribute float gl_FogCoord;
```

Vordefinierte varying:

```
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;
varying vec4 gl_FrontSecondaryColor;
varying vec4 gl_BackSecondaryColor;
varying vec4 gl_TexCoord[]; // at most will be gl_MaxTextureCoordsARB
varying float gl_FogFragCoord;
```

Lesbar in Fragment Shader sind:

```
varying vec4 gl_Color;
varying vec4 gl_SecondaryColor;
varying vec4 gl_TexCoord[]; // at most will be gl_MaxTextureCoordsARB
varying float gl_FogFragCoord;
```

4.6.3. Fragment Shader

Die (Fenster-Relativen) Werte x,y,z,1/w des Fragments (read-only)

```
vec4 gl_FragCoord
```

True, wenn das Fragment Front-Facing ist (read-only)

```
bool gl_FrontFacing
```

Die Farbe des Fragments, **muss** geschrieben werden (write)

```
vec4 gl_FragColor
```

Falls Depth Buffering aktiviert ist, so kann optional der Tiefenwert manuell gesetzt werden.

```
float gl_FragDepth
```

4.6.4. Variablen für Vertex und Fragment Shader

Vordefinierte Konstanten:
(Werte, welche hier verwendet werden entsprech der Minimal-Spezifikation)

```
const int gl_MaxLights = 8;                          // GL 1.0
const int gl_MaxClipPlanes = 6;                      // GL 1.0
const int gl_MaxTextureUnits = 2;                    // GL 1.2
const int gl_MaxTextureCoordsARB = 2;                // ARB_fragment_program
const int gl_MaxVertexAttributesGL2 = 16;            // GL2_vertex_shader
const int gl_MaxVertexUniformFloatsGL2 = 512;        // GL2_vertex_shader
const int gl_MaxVaryingFloatsGL2 = 32;               // GL2_vertex_shader
const int gl_MaxVertexTextureUnitsGL2 = 1;           // GL2_vertex_shader
const int gl_MaxFragmentTextureUnitsGL2 = 2;         // GL2_fragment_shader
const int gl_MaxFragmentUniformFloatsGL2 = 64;       // GL2_fragment_shader
```

Vordefinierte Uniforms:

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat3 gl_NormalMatrix; // derived
uniform mat4 gl_TextureMatrix[gl_MaxTextureCoordsARB];
```

```
uniform float gl_NormalScale;
```

```
struct gl_DepthRangeParameters
{
float near;      // n
float far;       // f
float diff;      // f - n
};
uniform gl_DepthRangeParameters gl_DepthRange;
```

```
uniform vec4 gl_ClipPlane[gl_MaxClipPlanes];
```

```
struct gl_PointParameters {
float size;
float sizeMin;
float sizeMax;
float fadeThresholdSize;
float distanceConstantAttenuation;
float distanceLinearAttenuation;
float distanceQuadraticAttenuation;
};
uniform gl_PointParameters gl_Point;
```

```
struct gl_MaterialParameters {
vec4 emission;   // Ecm
vec4 ambient;    // Acm
vec4 diffuse;    // Dcm
vec4 specular;   // Scm
float shininess;         // Srm
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

```
struct gl_LightSourceParameters {
vec4 ambient;                    // Acli
vec4 diffuse;                    // Dcli
vec4 specular;                   // Scli
vec4 position;                   // Ppli
vec4 halfVector;                 // Derived: Hi
vec3 spotDirection;              // Sdli
float spotExponent;              // Srli
float spotCutoff;                // Crli
                                 // (range: [0.0,90.0], 180.0)
float spotCosCutoff;             // Derived: cos(Crli)
                                 // (range: [1.0,0.0],-1.0)
float constantAttenuation;       // K0
float linearAttenuation;         // K1
float quadraticAttenuation;      // K2
};
uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
```

```
struct gl_LightModelParameters {
vec4 ambient; // Acs
};
uniform gl_LightModelParameters gl_LightModel;
```

```
struct gl_LightModelProducts {
vec4 sceneColor; // Derived. Ecm + Acm * Acs
};
uniform gl_LightModelProducts gl_FrontLightModelProduct;
uniform gl_LightModelProducts gl_BackLightModelProduct;
```

```
struct gl_LightProducts {
vec4 ambient; // Acm * Acli
vec4 diffuse; // Dcm * Dcli
vec4 specular; // Scm * Scli
};
uniform gl_LightProducts gl_FrontLightProduct[gl_MaxLights];
uniform gl_LightProducts gl_BackLightProduct[gl_MaxLights];
```

```
uniform vec4 gl_TextureEnvColor[gl_MaxFragmentTextureUnitsGL2];
uniform vec4 gl_EyePlaneS[gl_MaxTextureCoordsARB];
uniform vec4 gl_EyePlaneT[gl_MaxTextureCoordsARB];
uniform vec4 gl_EyePlaneR[gl_MaxTextureCoordsARB];
uniform vec4 gl_EyePlaneQ[gl_MaxTextureCoordsARB];
uniform vec4 gl_ObjectPlaneS[gl_MaxTextureCoordsARB];
uniform vec4 gl_ObjectPlaneT[gl_MaxTextureCoordsARB];
uniform vec4 gl_ObjectPlaneR[gl_MaxTextureCoordsARB];
uniform vec4 gl_ObjectPlaneQ[gl_MaxTextureCoordsARB];
```

```
struct gl_FogParameters {
vec4 color;
float density;
float start;
float end;
float scale; // 1 / (gl_FogEnd - gl_FogStart)
};
uniform gl_FogParameters gl_Fog;
```

4.6.5. Eingebaute Funktionen

Winkel und Trigonometrische Funktionen

genType **radians** (genType *degrees*)	Converts *degrees* to radians and returns the result, i.e., result = ? /180 ?*degrees*.
genType **degrees** (genType *radians*)	Converts *radians* to degrees and returns the result, i.e., result = 180/? ?*radians*.
genType **sin** (genType *angle*)	The standard trigonometric sine function.
genType **cos** (genType *angle*)	The standard trigonometric cosine function.
genType **tan** (genType *angle*)	The standard trigonometric tangent.
genType **asin** (genType *x*)	Arc sine. Returns an angle whose sine is *x*. The range of values returned by this function is [–?/2, ?/2]. Results are undefined if $\|x\| > 1$.
genType **acos** (genType *x*) Arc cosine.	Returns an angle whose cosine is *x*. The range of values returned by this function is [0, ?]. Results are undefined if $\|x\| > 1$.
genType **atan** (genType *y*, genType *x*)	Arc tangent. Returns an angle whose tangent is *y/x*. The signs of *x* and *y* are used to determine what quadrant the angle is in. The range of values returned by this function is [–?, ?]. Results are undefined if *x* and *y* are both 0.
genType **atan** (genType *y_over_x*)	Arc tangent. Returns an angle whose tangent is *y_over_x*. The range of values returned by this function is [–?/2, ?/2].

Exponentialfunktionen

genType **pow** (genType *x*, genType *y*)	Returns *x* raised to the *y* power, i.e., *xy*
genType **exp2** (genType *x*)	Returns 2 raised to the *x* power, i.e., 2*x*
genType **log2** (genType *x*)	Returns the base 2 log of *x*, i.e., returns the value *y* which satisfies the equation $x = 2y$
genType **sqrt** (genType *x*)	Returns the positive square root of *x*
genType **inversesqrt** (genType *x*)	Returns the reciprocal of the positive square root of *x*

Standardfunktionen

genType **abs** (genType *x*)	Returns *x* if $x >= 0$, otherwise it returns –*x*
genType **sign** (genType *x*)	Returns 1.0 if $x > 0$, 0.0 if $x = 0$, or –1.0 if $x < 0$
genType **floor** (genType *x*)	Returns a value equal to the nearest integer that is less than or equal to *x*
genType **ceil** (genType *x*)	Returns a value equal to the nearest integer that is greater than or equal to *x*
genType **fract** (genType *x*)	Returns x – **floor** (*x*)
genType **mod** (genType *x*, float *y*) genType **mod** (genType *x*, genType *y*)	Modulus. Returns $x - y * $ **floor** (*x*/*y*)
genType **min** (genType *x*, genType *y*) genType **min** (genType *x*, float *y)*	Returns *y* if $y < x$, otherwise it returns *x*
genType **max** (genType *x*, genType *y*) genType **max** (genType *x*, float *y*)	Returns *y* if $x < y$, otherwise it returns *x*
genType **clamp** (genType *x*, genType *minVal*, genType *maxVal*) genType **clamp** (genType *x*, float *minVal*, float *maxVal*)	Returns **min** (**max** (*x*, *minVal*), *maxVal*) Note that colors and depths written by fragment shaders will be clamped by the implementation after the fragment shader runs.
genType **mix** (genType *x*,genType *y*,genType *a*) genType **mix** (genType *x*, genType *y*, float *a*)	Returns $x * (1 - a) + y * a$, i.e., the linear blend of *x* and *y*
genType **step** (genType *edge*, genType *x*) genType **step** (float *edge*, genType *x*)	Returns 0.0 if $x <= edge$, otherwise it returns 1.0
genType **smoothstep** (genType *edge0*,genType *edge1*,genType *x*)	
genType **smoothstep** (float *edge0*,float *edge1*,genType *x*)	Returns 0.0 if $x <= edge0$ and 1.0 if $x >= edge1$ and performs smooth Hermite interpolation between 0 and 1 when $edge0 < x < edge1$. This is useful in cases where you would want a threshold function with a smooth transition. This is equivalent to: genType t; t = clamp ((x – edge0) / (edge1 – edge0), 0, 1); return t * t * (3 – 2 * t);

Geometrische Funktionen

float **length** (genType *x*)	Returns the length of vector *x*, i.e., **sqrt** (*x*[0] ∗ *x*[0] + *x*[1] ∗ *x*[1] + ...)
float **distance** (genType *p0*, genType *p1*)	Returns the distance between *p0* and *p1*, i.e. **length** (p0 – p1)
float **dot** (genType *x*, genType *y*)	Returns the dot product of *x* and *y*, i.e., result = *x*[0] ∗ *y*[0] + *x*[1] ∗ *y*[1] + ...
vec3 **cross** (vec3 *x*, vec3 *y*)	Returns the cross product of x and y, i.e. result.0 = *x*[1] ∗ *y*[2] - *y*[1] ∗ *x*[2] result.1 = *x*[2] ∗ *y*[0] - *y*[2] ∗ *x*[0] result.2 = *x*[0] ∗ *y*[1] - *y*[0] ∗ *x*[1]
genType **normalize** (genType *x*)	Returns a vector in the same direction as *x* but with a length of 1.
vec4 **ftransform**()	For vertex shaders only. This function will ensure that the incoming vertex value will be transformed in a way that produces exactly the same result as would be produced by OpenGL's fixed functionality transform. It is intended to be used to compute gl_Position, e.g.,
gl_Position = **ftransform**()	This function should be used, for example, when an application is rendering the same geometry in separate passes, and one pass uses the fixed functionality path to render and another pass uses programmable shaders.
genType **faceforward** (genType *N*, genType *I*, genType *Nref*)	If **dot** (*Nref*, *I*) < 0 return *N* otherwise return –*N*
genType **reflect** (genType *I*, genType *N*)	For the incident vector *I* and surface orientation *N*, returns the reflection direction: result = *I* – 2 ∗ **dot** (*N*, *I*) ∗ *N* *N should be normalized in order to achieve the desired result.*

Matrix Funktionen

mat **matrixCompMult** (mat *x*, mat *y*)	Multiply matrix *x* by matrix *y* component-wise, i.e., result[i][j] is the scalar product of *x*.[i][j] and *y*.[i][j]. Note: to get linear algebraic matrix multiplication, use the multiply operator (*).

Vektor Vergleichsfunktionen

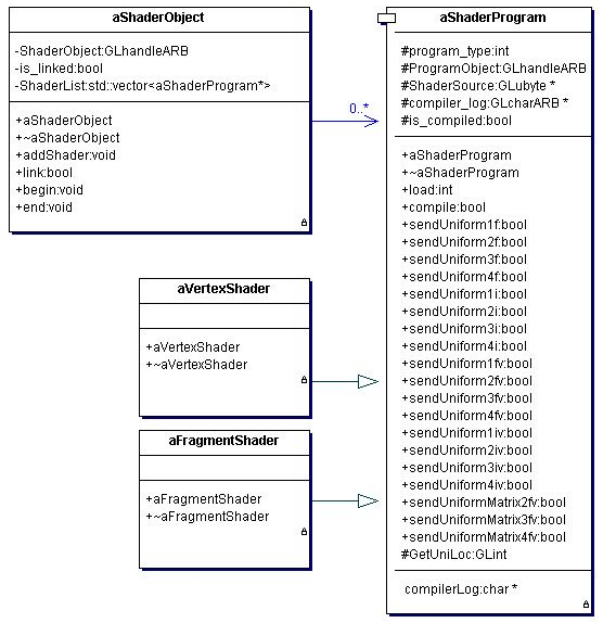
bvec **lessThan**(vec x, vec y) bvec **lessThan**(ivec x, ivec y)	Returns the component-wise compare of *x* < *y*.
bvec **lessThanEqual**(vec x, vec y) bvec **lessThanEqual**(ivec x, ivec y)	Returns the component-wise compare of *x* <= *y*.
bvec **greaterThan**(vec x, vec y) bvec **greaterThan**(ivec x, ivec y)	Returns the component-wise compare of *x* > *y*.
bvec **greaterThanEqual**(vec x, vec y) bvec **greaterThanEqual**(ivec x, ivec y)	Returns the component-wise compare of *x* >= *y*.
bvec **equal**(vec x, vec y) bvec **equal**(ivec x, ivec y) bvec **equal**(bvec x, bvec y)	Returns the component-wise compare of *x* == *y*.
bvec **notEqual**(vec x, vec y) bvec **notEqual**(ivec x, ivec y) bvec **notEqual**(bvec x, bvec y)	Returns the component-wise compare of *x* != *y*.
bool **any**(bvec x)	Returns true if any component of *x* is **true**.
bool **all**(bvec x)	Returns true only if all components of *x* are **true**.
bvec **not**(bvec x)	Returns the component-wise logical complement of *x*.

4.7. C++ Framework für die OpenGL Shading Language

Im Rahmen dieser Arbeit haben wir ein Framework geschrieben, welche die Benutzung der Shading Language mit C++ wesentlich vereinfacht. So ist es möglich Laden, Compilieren, die Datenübergabe und Aktivierung von verschiedenen Shadern und vieles mehr in wenigen Zeilen Code zu nutzen.

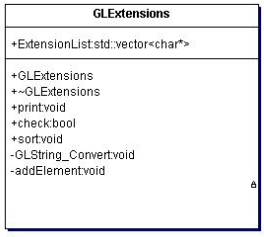
Auch wichtige Extensions wie GL_ARB_multitexturing und WGL_ARB_pbuffer werden von dem Framework verwaltet, weiterhin wird folgendes unterstützt:

- Laden von Texturen
- Einfache Lichteinstellungen
- Verwalten von OpenGL Extensions



Beispiel-Code:

```
// Variablen definieren (hier global)
aShaderObject* myShader;

// Source laden, compilieren und linken

myShader = DefaultShaderSetup("test.vert","test.frag");


// Programm verwenden (während Display-Loop)

myShader->begin();
 glutSolidTeapot(1.5f);
myShader->end();

// Speicher wird Ende am Ende automatisch freigegeben
```



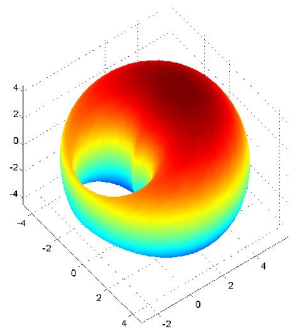
Hilfsklasse: OpenGL Extensions überprüfen.

Eine Sammlung von einfachen Beispielen zu diesem Framework (und auch die aktualle Version des Frameworks) kann man auf http://www.clockworkcoders.com/oglsl herunterladen.

4.8. Beispiele für Vertex Shader

4.8.1. Generieren von Oberfächenobjekten aus einem Mesh

In Matlab werden Oberflächen-Objekte aus einem Meshgrid erstellt. Am Beispiel eines Torus (und zwar kein symmetrischer Kreistorus) sieht dies folgendermassen aus:



```
[u,v] = meshgrid(0:0.1:2*pi+0.1, 0:0.1:2*pi+0.1);<

% Parameter des Torus:
R = 3; r=1.5;

% Torus
X = (R + r .* cos(v) .* (1+sin(u))) .* cos(u);
Y = (R + r .* cos(v) .* (1+sin(u))) .* sin(u);
Z = r .* sin(v) .* (2+sin(u));

% Plot
surf(X,Y,Z)
shading INTERP
axis equal
```

Die Idee dieses Vertex-Shaders ist es, ein Grid, bestehend aus einer vordefinierten Anzahl von Quads durch einen Vertex-Shader laufen zu lassen, um ein Oberflächenobjekt zu erzeugen.
Das Grid wird dabei in einem glBegin()...glEnd() Block definiert. Für bestmögliche Performance kann man das Grid auch als Vertex-Buffer-Objekt definieren.

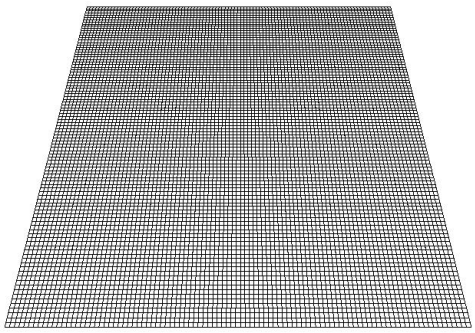
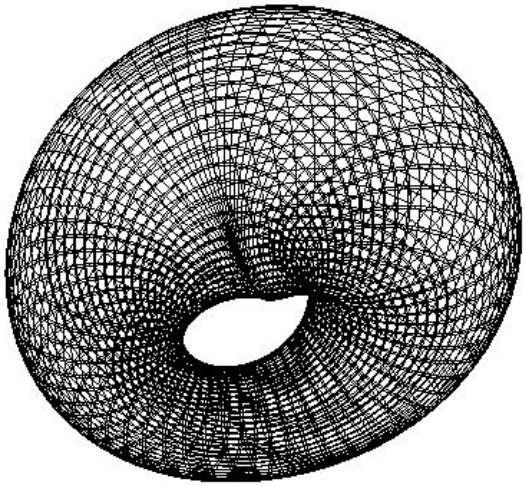


Bild: Grid mit 10'000 Quads



Screenshot: animierter asymmetrischer Torus durch Vertex Shader

Der asymmetrische Torus entsteht durch folgende Formel:

$$V(u,v)=\begin{pmatrix} -(3+\cos(u)\cdot(2+\sin(v+time)))\cdot\sin(v); \\ (3+\cos(u)\cdot(2+\sin(v+time)))\cdot\cos(v); \\ \sin(u)\cdot(2+\sin(v+time)); \end{pmatrix}$$

'time' ist eine Variable welche sich pro Millisekunde um $\frac{1}{1000}$ erhöht. Die Variable wird im C++ Programm verwaltet und dem Vertex Shader durch eine uniforme Variable übergeben. Somit ist eine zeitbasierte Animation gewährleistet.

Vertex Shader Source Code

```
uniform float time;
uniform float k;

varying vec4 color;


void main(void)
{
  vec4 V;
  float u = gl_Vertex.x;
  float v = gl_Vertex.y;

  V.x = -(3.0 + 1.0 * cos(u) * (2.0+sin(v+time))) * sin(v);
  V.y = (3.0 + 1.0 * cos(u) * (2.0+sin(v+time))) * cos(v);
  V.z = 1.0 * sin(u) * (2.0+sin(v+time));
  V.w = 1.0;

  color = vec4(0.0,0.0,0.0,1.0);

  gl_Position = gl_ModelViewProjectionMatrix * V;
}
```

Anhand dieses einfachen Beispiels für den Torus, sieht man, dass man ohne grossen Aufwand beliebige Oberflächenobjekte auf der GPU berechnen kann.

Einige weitere Beispiele interessanter Funktionen:

Elliptische Spirale um die x-Achse: $\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} t/r \\ 2\cdot\cos(t) \\ \sin(t) \end{pmatrix}$

Kreistorus: $\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -(R+r\cdot\cos(u))\cdot\sin(v) \\ (R+r\cdot\cos(u))\cdot\cos(v) \\ r\cdot\sin(u) \end{pmatrix}$

Kugel: $\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r\cdot\cos(u)\cdot\cos(v) \\ r\cdot\sin(u)\cdot\cos(v) \\ r\cdot\sin(v) \end{pmatrix}$

Erweiterung: Bewegen des Objektes

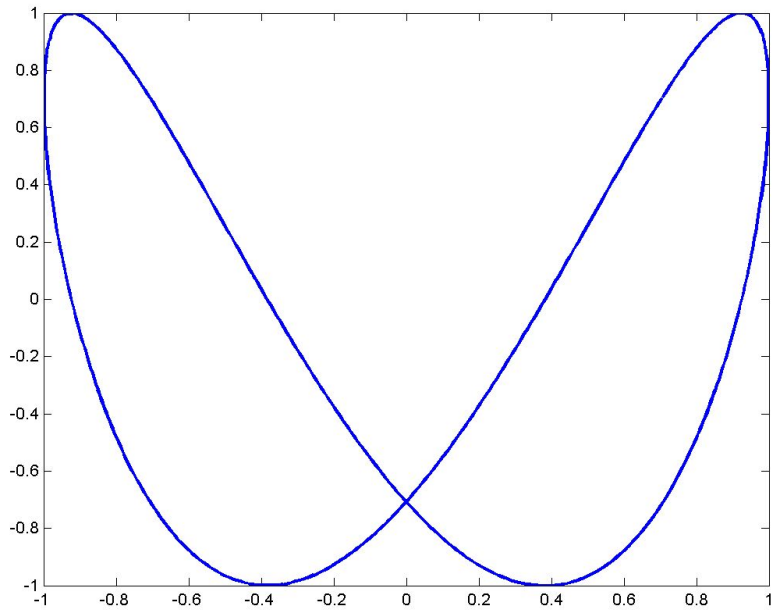
Das Vertex Programm soll nun erweitert werden, so dass sich der Torus entlang einer Lissajous Figur bewegt.

Eine allgemeine Formel für Lissajous Figuren:

$$\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} a_1 \cdot \cos(w_1 \cdot t - d_1) + b_1 \cdot \cos(w_2 \cdot t - d_2) \\ a_2 \cdot \cos(w_1 \cdot t - d_1) + b_2 \cdot \cos(w_2 \cdot t - d_2) \end{pmatrix}$$

Verwendet werden die Werte

$$a_1 = 1 ; a_2 = 0 ; b_1 = 0 ; b_2 = 1 ; w_1 = 1 ; w_2 = 2 ; d_1 = \frac{\pi}{4} ; d_2 = \frac{\pi}{4}$$



Vertex Shader Source Code

```
// Bewegen entlang einer Lissajous-Figur:
float t = mod(time,6.282);
V.x = V.x + cos(t - 0.785);
V.y = V.y + cos(2.0*t - 0.785);
```

Fazit

Animierte Oberflächenobjekte können problemlos auf der GPU erstellt werden, während die CPU nur mit einem einfachen Mesh beschäftigt ist. Die Bewegung entlang einer Kurve muss für jeden Vertex einzeln berechnet werden, was einen deutlichen Mehraufwand darstellt. Dieser Teil sollte nach wie vor in OpenGL als Transformation verwendet werden.

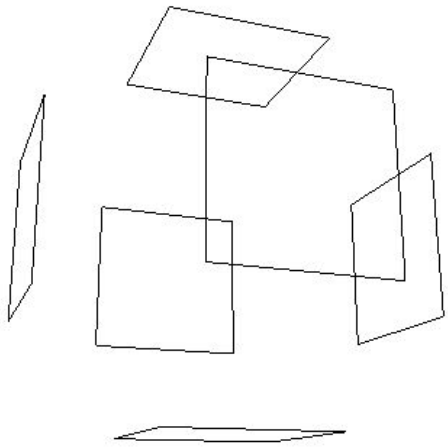
4.8.2. Deformation entlang von Normalen

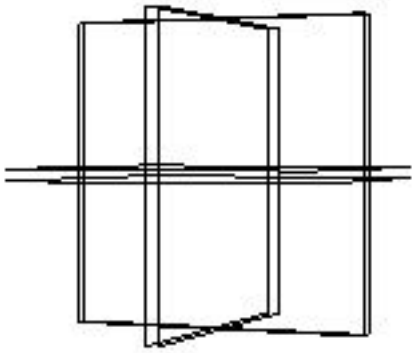
Ein grosser Vorteil von Vertex Shadern ist, dass man auch nach dem glBegin/End Block über alle Vertex Attribute des Objektes frei verfügen kann. So kann man zum Beispiel Vertices entlang der Normalen verschieben.

In diesem Beispiel soll gezeigt werden, wie man einfache Deformationen entlang von Normalen in der OpenGL Shading Language realisieren kann.

Extrude und Intrude

Beim extrudieren werden Polygone entlang der Normalen verschoben. Dies ergibt den Effekt des 'zerfallens' eines Objektes. Beim intrudieren wird entlang der negativen Normalen (also nach innen) verschoben.





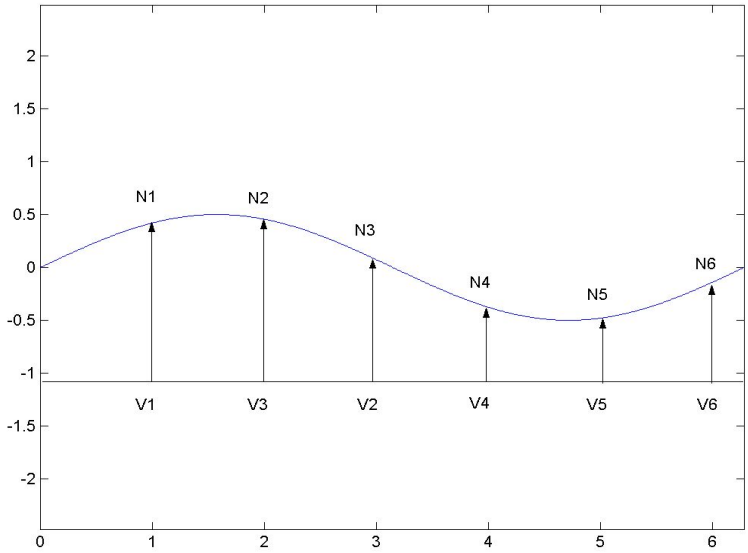
Screenshot: Extrude Intrude

Ein Vertex Shader kennt die Topologie der Polygone nicht. Wir transformieren einen einkommenden Vertex in Objekt-Koordinaten zu einen Vertex in clip-Koordinaten.

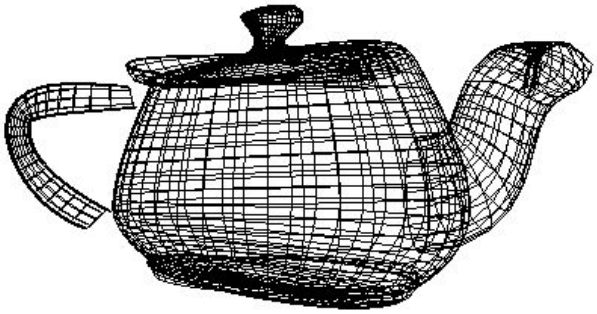
Der Variablen 'a' enthält irgendein Vertex vom jeweiligen Objekt. Die Vektoraddition a + d*N verschiebt den Vertex entlang der Normalen. Dies führt zu einem animierten Intrude resp. Extrude. Das C++ Programm hat die Kontrolle über den Wert 'd', in unserem Fall pendelt es zwischen 0 bis π hin-und her.

```
uniform float Position;
varying vec4 color;

void main(void)
{
      vec4 N = Position* vec4(normalize(gl_Normal),0.0);
      vec4 a = gl_Vertex;

      //Intrude:
      //float d = -Position;

      //Extrude:
      float d = Position;

      a = a + d*N;

      gl_Position = gl_ModelViewProjectionMatrix * a;

      color = gl_Color;
}
```

Die Punkte V werden entlang der Normalen N verschoben, auf der definierten Sinuskurve.



"Warp-Effekt"



In vec4 a wird ein Vertex von dem jeweiligen Objekt übergeben.
Position ist eine Variable dessen Wert von 0 bis π hin- und her pendelt.
Die Funktion, die die Varibale d beschreibt, ist abhängig von dem x-Wert und dem pendeltem Wert.

```
uniform float Position;
varying vec4 color;

void main(void)
{
       vec4 N = Position* vec4(normalize(gl_Normal),0.0);
       vec4 a = gl_Vertex;

       //Warp-Effekt
        float d = 0.5*sin(a.x * Position );

       a = a + d*N;

       gl_Position = gl_ModelViewProjectionMatrix * a;

       color = gl_Color;


}
```

4.8.3. Animation einer Flagge

Die Bedingungen einer richtigen Flagge sind , dass diese sich an einen Fahnenmast befestigen lässt und die Wind stärke zu- und abnehmen kann. Eine mathematische Funktion ist also grundlegend für das Verständnis der Bewegung einer Flagge. Es ergibt sich eine komplexe Sinusformel die in den Vertex Shader ausgelagert wird und damit den Prozessor entlastet.

Erzeugen einer Sinuskurve mit MATLAB

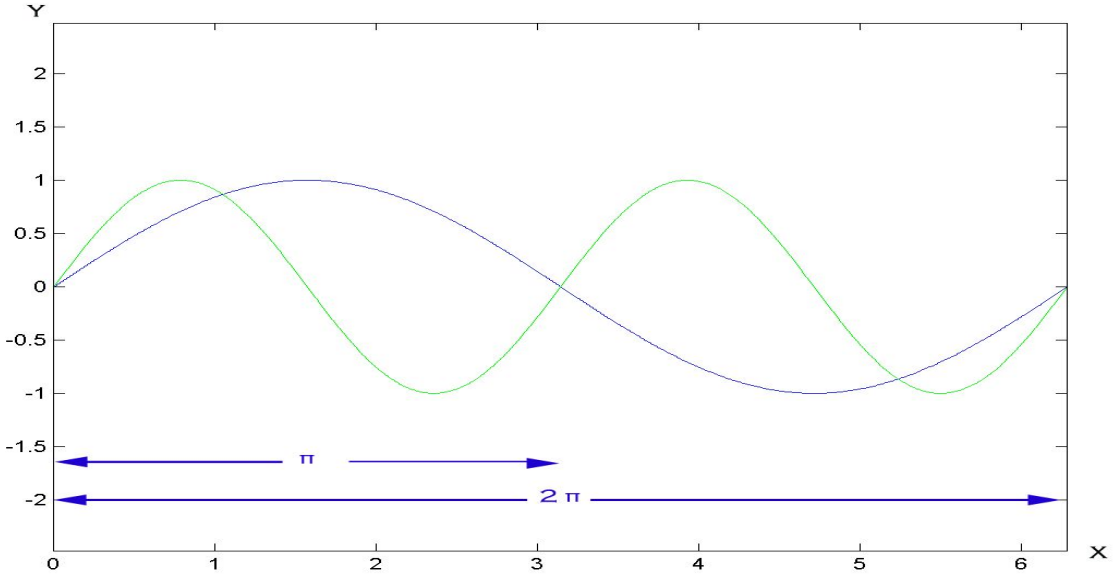
Um die Bewegung der Flagge zu verstehen beginnen wir mit einer einfachen Sinuskurve.

Matlab File die Stauchung einer Sinuskurve: test1.m

```
x = 0:0.01:2*pi;
z = sin(x);

plot(x,z,'b');
axis equal
hold on

%location = -0.5;
%z = sin(x+location);
%plot(x,z,'r');

wind = 2;
z = sin(wind*x);
plot(x,z,'g')
```

Betrachten wir die Darstellung der 2 Sinuskurven erkennen wir, dass eine Veränderung der Variablen wind, eine Stauchung oder eine Streckung der Wellenlänge. In diesem fall wird die Wellenlänge halbiert.

Gestauchte Sinuskurve:



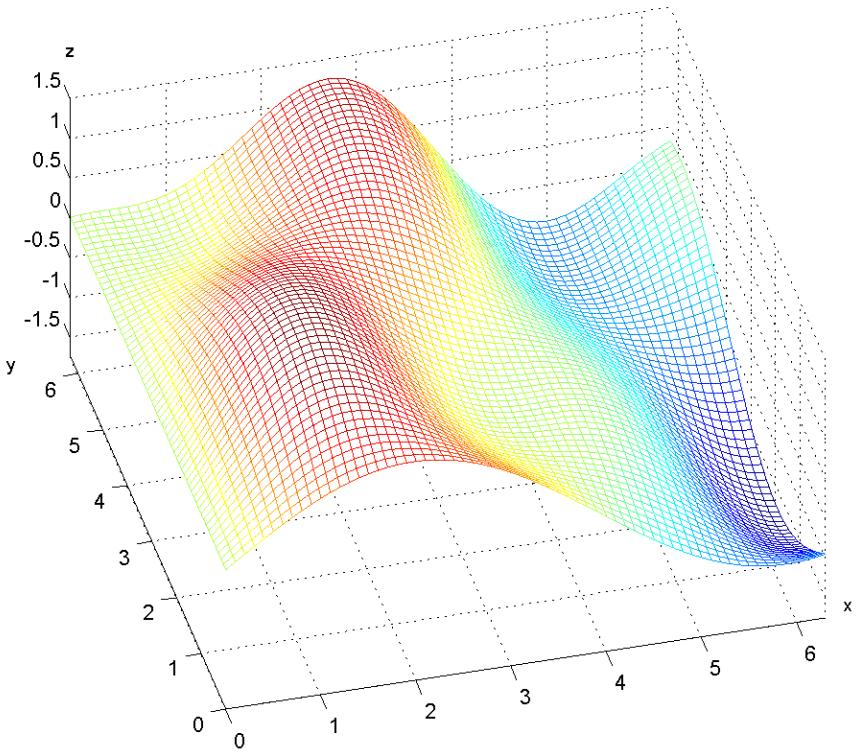
Diese einfache Sinuskurve reicht noch nicht aus um eine natürliche Flaggen Bewegung zu simulieren. Um eine bessere Bewegung zu erreichen, bestimmen wir eine zweite Variable location. Diese änderbare Variable veranlasst den Ausschlag der Flagge.

Für die eigentliche Funktion wird eine mehrfache Sinus Multiplikation und Addition verwendet.

Matlab File ein Flaggen ähnliche Sinusmesh: flagge.m

```
[u,v] = meshgrid(0:0.1:2*pi+0.1, 0:0.1:2*pi+0.1);

wind = 0.8;
location = 1;

X = u;
Y = v;
Z = sin(wind.*u).*sin(wind.*v) .* sin(location+u)+sin(wind .* u);

% Plot
mesh(X,Y,Z)
shading INTERP
axis equal
```

Darstellung der oben berechneten Funkion.

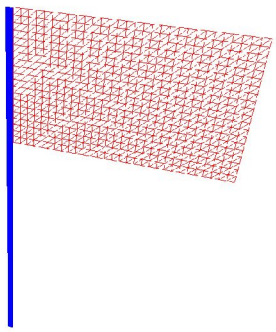


Darstellen einer Flagge mit OpenGL

Die Flagge wird aus einem rechteck mit F_Läge und F_Breite gebildet. Dieses Rechteck wird in viele kleine Quadrate unterteilt welche wiederum in Dreiecke halbiert werden. Die Quadrate bestehen aus den Eckpunkten a, b, c, d. Dieses Quadrat verschiebt sich in der Breite und Länge bis diese die Fläche der Flagge erstellt hat. Mit glVertex3f senden wir die einzelnen Vertexknoten dem Vertex Shader.

```
int F_Breite, F_Laenge;                    //Fahnen Breite und Länge
 float ax, ay, bx, by, cx, cy, dx, dy;    //Vertex Punkte eines 2D Quad
 float r=0.2;                              //Quad Seitenlänge

 F_Laenge = 30;
 F_Breite = 20;


 for (int i=0; i<F_Laenge; i++)                 //Meshzeichnen
 {
       for (int j=0; j<F_Breite; j++)
       {
             ax = i*r;                     //Algorithmus für die Verschiebung der
             ay = j*r;                     // einzelnen Vertex Punkte des Quad
             bx = (i+1)*r;
             by = ay;
             cx = (i+1)*r;
             cy = (j+1)*r;
             dx = ax;
             dy = (j+1)*r;


           glColor3f(1.0,0.0,0.0);        //Flagge wird Rot gefärbt
            glBegin(GL_TRIANGLES);         //Die einzelnen Quads werden als
                                                 //Dreiecke aufgebaut
                   glVertex3f(ax, ay, 0.0f);
                   glVertex3f(bx, by, 0.0f);
                   glVertex3f(cx, cy, 0.0f);

                   glVertex3f(cx, cy, 0.0f);
                   glVertex3f(dx, dy, 0.0f);
                   glVertex3f(ax, ay, 0.0f);

             glEnd();
       }}
```

Darstellung der Flagge ohne Vertex Shader.



Verwenden des Vertex Shaders für die Flagge.

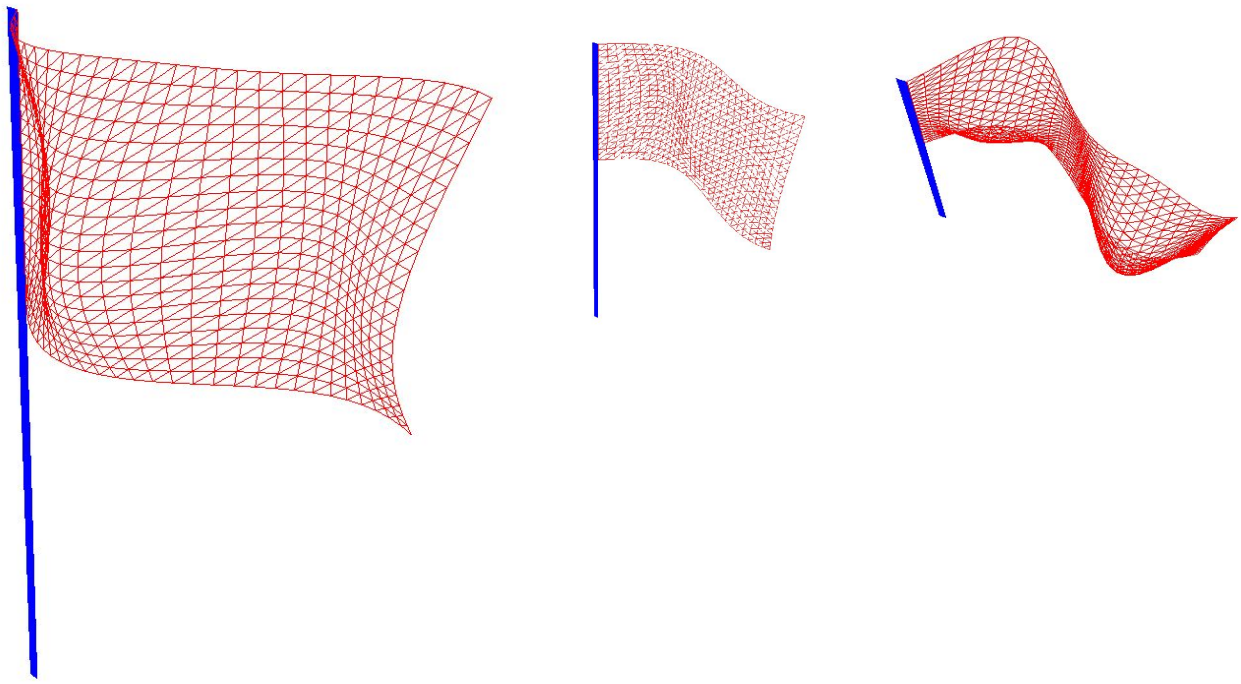
Hier erkennern wird den Grund weshalb Matlab als Einführung verwendet wurde, denn ohne diese ist es schwer zu erkennen was diese Sinusfunktion macht. Die erste, hier auskommentierte Sinusfunktion bewirkt die Stauchung der Fahne. Die zweite, etwas komplexere Funktion unterstützt die natürliche Flaggen bewegung.

Source code: simple.vert

```
uniform float location;
uniform float wind;
varying vec4 color;

void main(void)
{
      vec4 a = gl_Vertex;  //  a ist eine schreibbare kopie von gl_vertex

      //a.z = sin(wind*a.x)*sin(wind*a.x + location);      //Fahne stauchen

      a.z =sin(wind*a.x)*sin(wind*a.y)*sin(location+a.x)+sin(wind*a.x);
                  //Fahne weht und flattert

      gl_Position = gl_ModelViewProjectionMatrix * a;
      color = gl_Color;
}
```

Das Ergebnis der einfachen Sinuskurve sieht noch etwas künstlich aus.

Was sich jedoch mit der überlagerten Sinusfunktion in ein anschauliches Ergebnis wandelt.

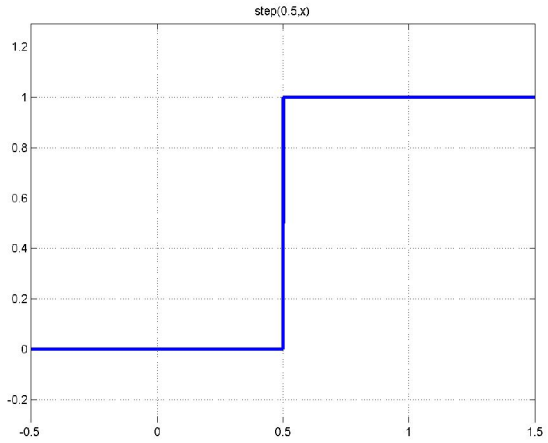


5. Mathematik des Shading

5.1. Grundfunktionen

Für das Shading gibt es einige Grundfunktionen, welche in den meisten Shadersprachen vordefiniert sind, es folgt hier eine kurze Betrachtung.

5.1.1. Step



$$step(e,x) := \begin{cases} 0 & falls\ x < e \\ 1 & sonst \end{cases}$$

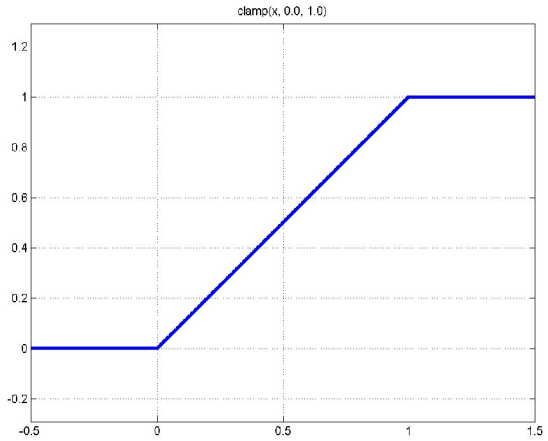
Matlab

```
function r = step(e,x)
r=(sign(x-e)+1)/2;
```

C++

```
float step(float e, float x)
{
     return (float) (x>=a);
}
```

5.1.2. Clamp



Matlab

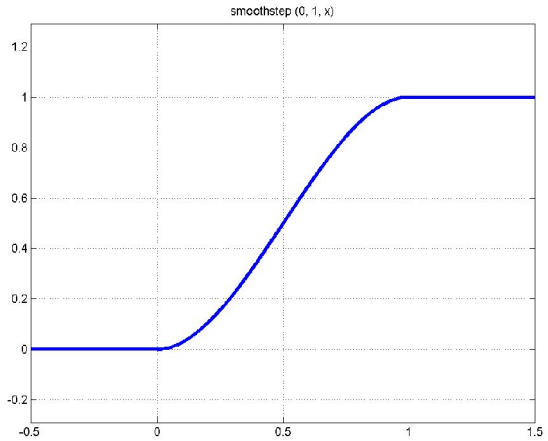
```
function r = clamp(x, xmin , xmax)
r = min(max(x,xmin),xmax);
```

C++

```
float clamp(float x, float a, float b)
{
return (x < a ? a : (x>b ? b : x ));
}
```

```
float min(float a, float b)
{
   return (a < b ? a : b);
}

float max(float a, float b)
{
   return (a < b ? b : a)
}
```

5.1.3. Smoothstep



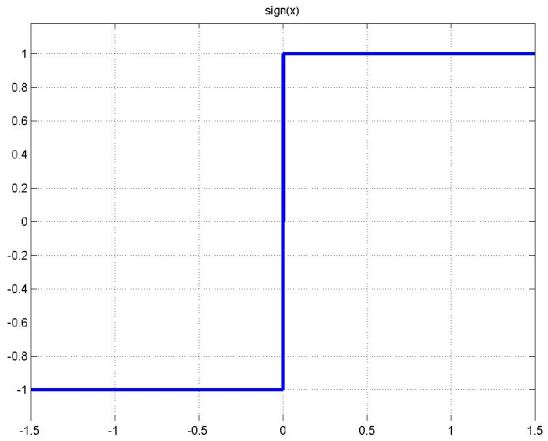
$$t = clamp\left(\frac{(x-e_0)}{(e_1-e_0)}, 0, 1\right)$$

$$smoothstep(e_0, e_1, x) := 3t^2 - 2t$$

Matlab

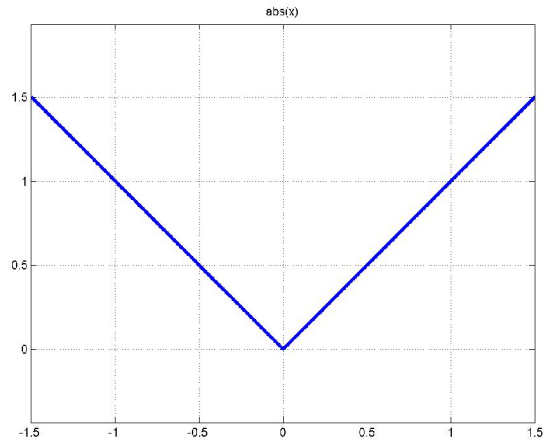
```
function r = smoothstep(e0,e1,x)
r = clamp((x-e0)/(e1-e0),0,1);
r = r.*r.*(3-2*r);
```

5.1.4. Sign



$$sign(x) := \begin{cases} 1 & falls\ x > 0 \\ -1 & falls\ x < 0 \\ 0 & falls\ x = 0 \end{cases}$$

5.1.5. Abs



$$abs(x) := \begin{cases} x & falls\ x \geqslant 0 \\ -x & falls\ x < 0 \end{cases} = |x|$$

```
float abs(float x)
{
    return (x < 0 ? -x : x);
}
```

5.1.6. Fract und Pulse

nur gebrochener Teil einer Zahl.

$$0 \leqslant fract(x) < 1$$

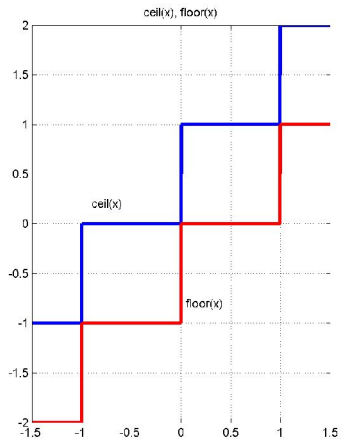
```
function r = fract(x)
r=x-floor(x);
```

Eine Puls Funktion lässt sich folgendermassen erzeugen:

C++

```
#define pulse(a,b,x) (step((a),(x)) - step((b),(x)))
```

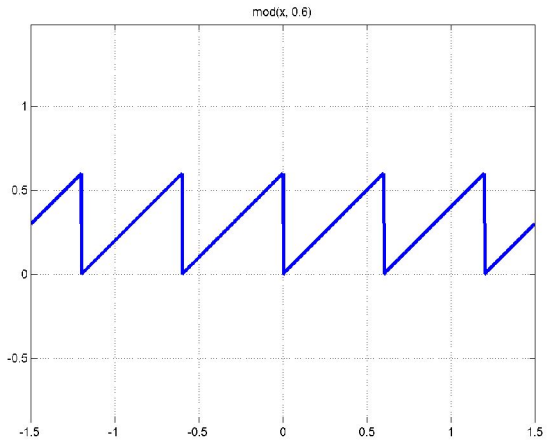
5.1.7. Ceil und Floor



C++

```
#define floor(x)  ((int)(x) - ((x) < 0 && (x) != (int)(x)))
#define ceil(x)   ((int)(x) + ((x) > 0 && (x) != (int)(x)))
```

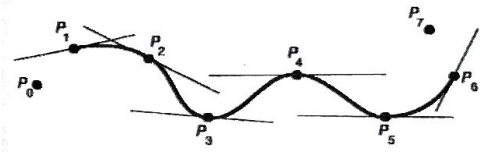
5.1.8. Modulo



$$mod(x, y) := x - y \cdot floor\left(\frac{x}{y}\right)$$

5.2. Spline

Die RenderMan Shading Language verfügt über eine spline Funktion, welche unter anderem ein eindimensionales Catmull-Rom Spline, das durch eine Menge von Knoten ("knot values") interpoliert wird, berechnet. Wir wollen diese Spline Funktion auf der GPU benutzen können. Beim Catmull-Rom Spline geht die Kurve durch die angegebenen Punkte (ohne Start- und Endpunkt) und die Tangente ist durch die jeweiligen Nachbarpunkte gegeben:



$$p(u)=[1,\ u,\ u^2,\ u^3]\begin{bmatrix} 0 & 1 & 0 & 0 \\ -s & 0 & s & 0 \\ 2s & s-3 & 3-2s & -s \\ -s & 2-s & s-2 & s \end{bmatrix}\begin{bmatrix} p_{i-3} \\ p_{i-2} \\ p_{i-1} \\ p_i \end{bmatrix}$$

Wobei der Spannungsparameter s typischerweise 0.5 ist.

Catmull-Rom Spline Funktionen werden vor allem für folgende Anwendungen benötigt:

- Glatte Interpolation von Punktfolgen
- Keyframe Animation, Kamerafahrt etc.
- Kontrollpunkte äquidistant in der Zeit

Weshalb reicht eindimensional aus ?

$$spline\left(\begin{pmatrix} X \\ Y \\ Z \end{pmatrix}, \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}, \ldots, \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix}\right) \equiv (spline(X, x_1, \ldots, x_n), spline(Y, y_1, \ldots, y_n), spline(Z, z_1, \ldots, z_n))^T$$

Implementation: Eindimensionale Spline Funktion mit 4 Knoten:

Die Parameter sind float Werte.

C++ / GLSL

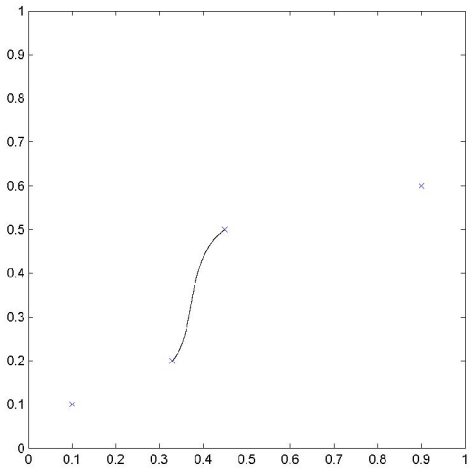
```
float spline(float x, float k0, float k1, float k2, float k3)
{
float c0, c1, c2, c3; // Koeffizienten
x = clamp(x,0,1);

c3 = -0.5*k0 + 1.5*k1 - 1.5*k2 + 0.5*k3;  // Spannungsparameter 0.5 ist hier fest vorgegeben
c2 =  1.0*k0 - 2.5*k1 + 2.0*k2 - 0.5*k3;
c1 = -0.5*k0 + 0.0*k1 + 0.5*k2 + 0.0*k3;
c0 =  0.0*k0 + 1.0*k1 + 0.0*k2 + 0.0*k3;

return ((c3*x + c2)*x + c1)*x + c0;
}
```

Matlab

```
function r = catmullspline(x, k0, k1, k2, k3);
s = 0.5 % Spannungsparameter

M = [0 1 0 0; -s 0 s 0; 2*s s-3 3-2*s -s; -s 2-s s-2 s];
v = [1 x x^2 x^3];
w =[k0; k1; k2; k3];

r = v*M*w;
```

Demo-Script in Matlab: 2-dimensionales Spline zeichnen:

```
k0 = [0.1 0.1]; k1 = [0.33 0.2]; k2 = [0.45 0.5]; k3 = [0.9 0.6];

plot(k0(1), k0(2), 'bx'); hold on; axis equal
plot(k1(1), k1(2), 'bx');
plot(k2(1), k2(2), 'bx');
plot(k3(1), k3(2), 'bx');

axis([0 1 0 1])

for i=0:0.01:1;
   x1 = catmullspline(i, k0(1), k1(1), k2(1), k3(1));
   y1 = catmullspline(i, k0(2), k1(2), k2(2), k3(2));
end
```



Die Spline Funktion auf der GPU kann nun für Bewegungsabläufe und Farbverläufe verwendet werden.

5.3. Noise

Noise Funktionen sind in Fragment und in Vertex Shader vorhanden. Sie sind stochastische Funktionen welche verwendet werden können, um die visuelle Komplexität zu erhöhen. Die Funktionen scheinen zufällig zu sein, sind es aber nicht:

Rückgabewerte der noise Funktion liegen in [-1,1]
Rückgabewerte haben den Durchschnitt von 0.0
Werte sind wiederholbar: noise mit dem selben Eingabeparameter gibt den selben Wert zurück
Statistisch unveränderlich bei Rotation (Isotrop)
Statistisch unveränderlich bei Translation (Stationär)
Noise hat eine maximale Band-Frequenz von ungefähr 1

Wir verwenden die noise(x,y,z) Funktion Ken Perlin wie in seinem Artikel "Noise, Hypertexture, Antialiasing and Gesture" in "Texturing&Modeling – A procedural approach" beschrieben ist.

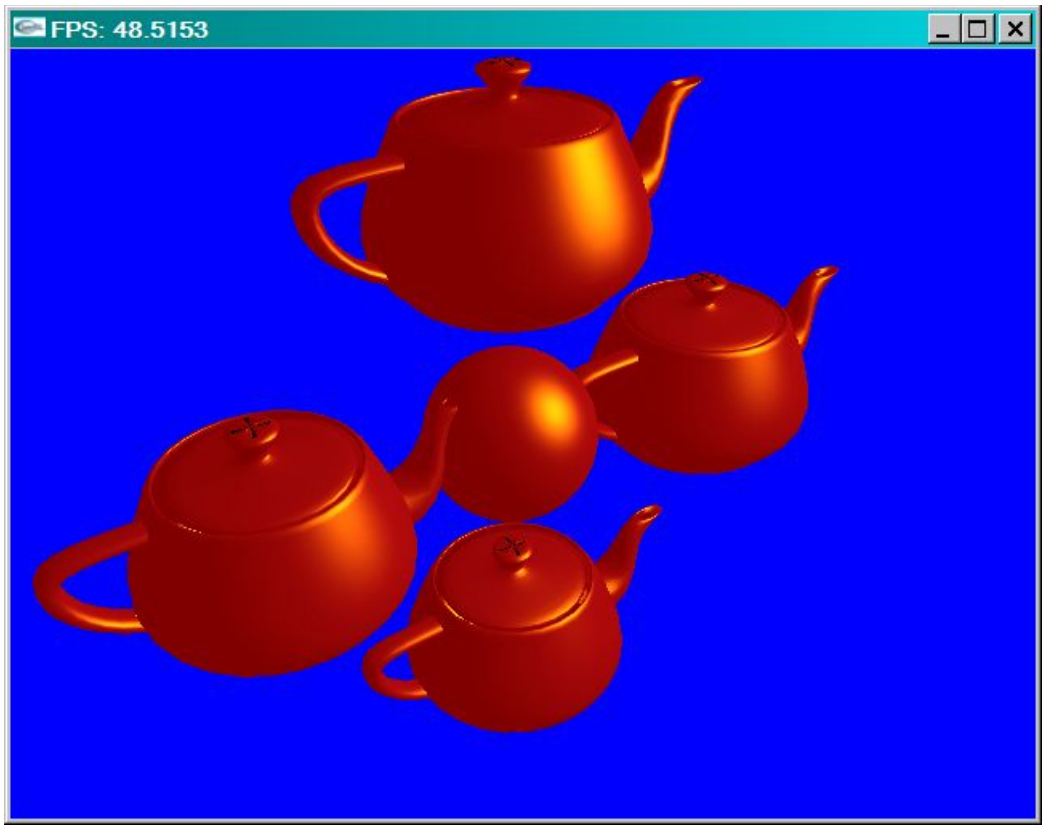
Die OpenGL Shading Language verfügt über eingebaute noise Funktionen, welche jedoch noch auf keiner Implementation vorhanden sind. NVidia/ATI und 3DLabs empfehlen die Verwendung von 3D Noise Texturen, d.h. Perlin Noise in einer Textur encodiert.

5.4. Beispiel

Eine 3D Textur wurde mit der Perlin Noise Funktion erstellt und auf eine Oberfläche eines Objektes abgebildet.

6. Beleuchtung

6.1. Per Pixel Phong Shading



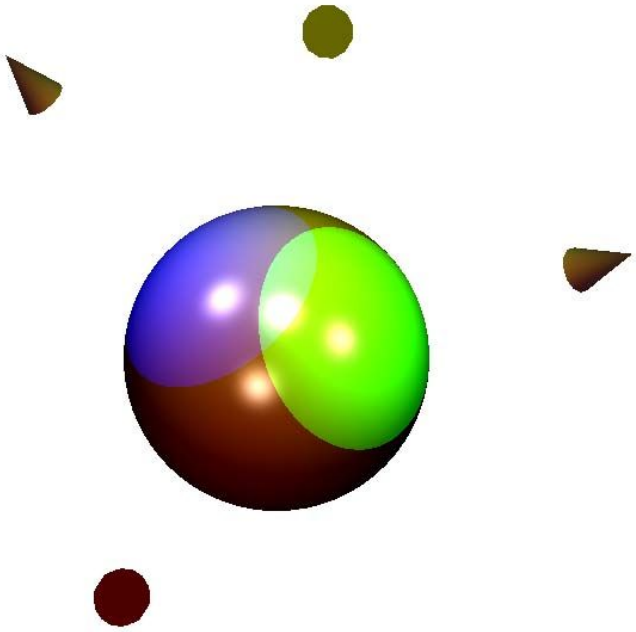
Die Formel für die per Pixel Phong Beleuchtung, wie sie in OpenGL verwendet wird, kann man der OpenGL Referenz entnehmen und in ein GLSL Programm konvertieren.

Fragment Shader:

```
varying vec3 L;
varying vec3 N;
varying vec3 P;

void main (void)
{
  vec3 E = normalize(-P);
  vec3 R = normalize(-reflect(L,N));

  vec4 Iamb = gl_FrontLightProduct[0].ambient;
  vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);
  vec4 Ispec = gl_FrontLightProduct[0].specular * pow(max(dot(R,E),0.0),0.3 *
gl_FrontMaterial.shininess);

  gl_FragColor =  gl_FrontLightModelProduct.sceneColor + Iamb + Idiff + Ispec;
}
```

6.2. Mehrere Lichtquellen

Die GeForce FX Generation von NVidia kann **maximal nur 4 Lichtquellen verwenden**, da sonst die maximal zulässige Anzahl von Instruktionen überschritten würde. Auf der ATI Karte konnten wir nur maximal 2 Lichtquellen verwenden. Wie bereits früher erwähnt, wird diese Restriktion in Pixel Shader 3.0 kompatiblen Grafikkarten nicht mehr vorhanden sein.



6.2.1. Licht Shader Programmierung

Wir verwenden hier vier Lichter die auf eine Kugel gerichtet sind. Die Deklaration und die Einbindung der beiden verwendeten Shader steht in der FragmentLightingShader2 Klasse. Die Ausgabe findet mit Hilfe eines Qt Widget statt, d.h. auch die Übergabe der beiden Parameter int argc, char** argv, bei main(). Die Lichter werden einzeln erzeugt und im Fragmet Shader einzeln abgefragt. Es muss eine Instanz der FragmentLightingShader2 Klasse erzeugt werden.

Übergabe der Parameter an den Shader innerhalb des main.

```
scene->addNode(new SLAppearance(SLCol4f(1.0,1.0,1.0,1),SLCol4f(1,1,1,1),100,0, &Flightshader2));

  scene->addNode(sphere);
```

Beenden des Shader

```

  scene->addNode(new SLAppearance(SLCol4f(), SLCol4f(), 0, 0, 0));//stop shader
```

Licht Shader – Vertex Shader Programmierung

Dies ist eigentlich in diesem Beispiel nicht relevant, da ein Per Fragmentlighting um einiges besser dargestellt wird. Mittels Per Framgmentlighting werden auch Fragmente belichtet die innerhalb der verschiedenen Vertex Punkte liegen. Es werden hier keine wichtigen Berechnungen durchgeführt. Der Vertex Shader ist nur für die Ansichtsmatrix zuständig und für die Übergabe der einzelnen Punkte P (Vertex), n ist die Normalisierte der Normalen N. d.h. haben wir hier auch das kleine n verwendet. Anhand der varying Variabeln können wir die beiden berechneten Werte P und n dem Fragment Shader übergeben.

Unten ist der ganze Inhalt des Vertex Shader Programms aufgelistet.

F_lighting2.vert

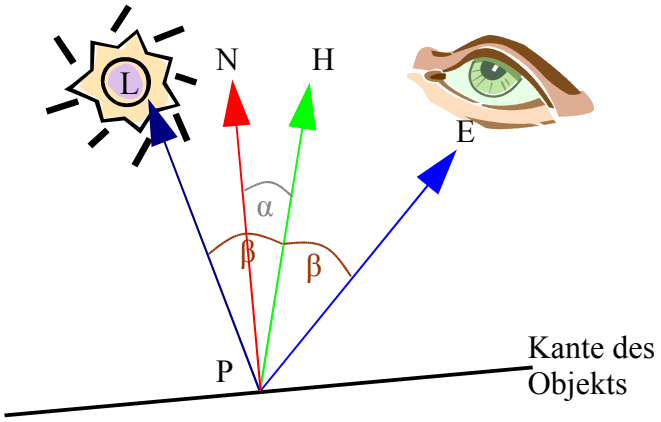
```
varying vec3 n;
varying vec3 P;

void main(void)
{

  P = vec3(gl_ModelViewMatrix * gl_Vertex);
  n = normalize(gl_NormalMatrix * gl_Normal);

  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

6.2.2. Licht Shader – Fragment Shader Programmierung

Im Fragment Shader F_lighting2.frag wird die eigentliche Beleuchtung berechnet. Hier ist wichtig, dass man die Vektormathematik verstanden hat.

6.2.3. Einführung in die Mathematik der vektorbasierten Licht- und Reflektionsgesetz

Wenn ein Lichtstrahl L auf eine Oberfläche fällt, wird dieser zu unserem Auge E reflektiert. Halbiert man den Winkel zwischen dem Lichtstrahl L und dem Eye Vektor findet man die Winkelhabierende H. Die Normale N steht senkrecht auf der Oberfläche unseres beleuchteten Objekts. Aus der Physik wissen wir, dass der Lichteinfalls Winkel mit dem gleichen Winkel an der Normalen gespiegelt wird, diesen Reflektierte Lichtstrahl nennen wir R. R ist in der unteren Grafik nicht eingezeichnet.



P: Punkt auf dem Objekt
E: Eye Vektor (Vektor zum Auge)
L: Licht Vektor
H: Half Vektor (Winkelhalbierende)
N: Normale zum Objekt

Im Fragment Shader F_lighting2.frag werden zu erst alle Variabeln deklariert. Danach werden für die vier Lichtquellen die Vektoren berechnet.

Normale N = normalisierte von n (n wurde im Vertex Shader berechnet).

Licht Vektor L = normalisierter Abstand des Lichtes zum Punkt P.
Die einzelnen Lichtpos. und der Punkt P sind bekannt und somit mit L = Lichtpos – P zu berechnen.

Eye Vektor E = normalisierter und invertierter Vektor von P. Die Koordinaten der beiden Punkte E und P sind gegeben. Der Vektor muss invertiert werden damit er von P nach E zeigt.

R = normalisierter Licht Vektor L an der Normalen N gespiegelt. L und N sind gegeben, d.h. wird R mit der Funktion reflect() berechnet. R muss invertiert werden damit die Richtung des Vektors stimmt.

H = normalisierter Half Vektor von den Vektoren E+L.

Wichtig für die Darstellung ist die Lichtstärke, die mit zunehmender Distanz zum Objekt abnimmt. Diese abnahm der Lichtstärke wird als attenuation beschrieben. Es wird die konstante, lineare und quadratische Abnahme berücksichtig. C sind die konstante und d steht für die Distanz.

$$f_{attenuation} = \left(\frac{1}{c_1 + c_2 d + c_3 d^2}, 1\right)$$

Danach wird mit Spotdot kontrolliert, ob der Vertex Punkt P innerhalb des Lichtkegels liegt. Dies wird mit dem Punktprodukt(dot) von L und der Spotrichtung der einzelnen Lichtquelle berechnet. Falls der Punkt P ausserhalb des Lichtkegels liegt, muss dieser nicht berücksichtigt werden, andernfalls muss er berechnet werden.

Danach wird mit Spotdot kontrolliert, ob der Vertex Punkt P innerhalb des Lichtkegels liegt. Dies wird mit dem Punktprodukt(dot) von L und der Spotrichtung der einzelnen Lichtquelle berechnet. Falls der Punkt P ausserhalb des Lichtkegels liegt, muss dieser nicht berücksichtigt werden, andernfalls muss er berechnet werden.

Es werden noch die verschiedenen Lichtreflexionen, wie das Ambiente-, Diffuse- und Speculärelicht benötigt.

Das Ambientelicht wird mit der Formel berechnet: $I_{ambient} = I_a * k_a$;

mit I_a als konstante, ambiente Lichtintensität und mit k_a als ambiente Reflexionskoeffizienten.

Das Diffuselicht wird mit der Formel berechnet: $I_{diffus} = I_d * k_d * \max(dot(N, L), 0)$

mit I_d als konstante, diffuse Lichtintensität, mit k_d als diffuse Reflexionskoeffizienten (zwischen 0 und 1) und der Berechnung des Puntproduktes von N und L.

as Speculärelicht (spiegelndes Licht) wird mit der Formel berechnet:

$$I_{speculär} = I_s * k_s * \max(dot(R, E), 0)$$

Diese Formel stammt von Bui-Thong Phong.

mit I_s als konstante, speculäre Lichtintensität, mit als speculäre Reflexionskoeffizienten (zwischen 0 und 1) und der Berechnung des Puntproduktes von R und E.

F_lighting2.frag

```
varying vec3 n;
varying vec3 P;


void main (void)
{
      int i; vec3 L; vec3 E; vec3 R; vec3 H; vec3 N;
```

```
      float distance;
      float SpotEffect;
      float attenuation;
      vec4 Iamb ;
      vec4 Idiff;
      vec4 Ispec;
      float SpotDot;


      for( i=0; i < 4; i++)
      {

   N = normalize(n);
   L = normalize(gl_LightSource[i].position.xyz-P);
   E = normalize(-P);
   R = normalize(-reflect(L,N));
   H = normalize(E+L);

   distance = length(L);      //Distanz des Lichtes auf den Vertex
   SpotEffect;

//abnahme
   attenuation= 1.0 / (gl_LightSource[i].constantAttenuation +
                  gl_LightSource[i].linearAttenuation *distance +
                  gl_LightSource[i].quadraticAttenuation *distance *distance);
//Spott berechnung ob p innerhalb oder ausserhalb Lichtkegel
      SpotDot = dot(-L, gl_LightSource[i].spotDirection);
      if (SpotDot < gl_LightSource[i].spotCosCutoff)
            SpotEffect = 0.0;
      else
            SpotEffect = pow(SpotDot, gl_LightSource[i].spotExponent);

   Iamb  =  gl_FrontLightProduct[i].ambient;
   Idiff =  gl_FrontLightProduct[i].diffuse * max(dot(N,L), 0.0);
   Ispec =  gl_FrontLightProduct[i].specular * pow(max(dot(R,E),0.0),0.3 *
            gl_FrontMaterial.shininess);


  gl_FragColor += Iamb + attenuation * SpotEffect * (Idiff + Ispec);

      }
}
```

Ispec mit Blinn Shader

```
vec4 Ispec = gl_FrontLightProduct[i].specular * pow(max(dot(N,H), 0.0),
                  gl_FrontMaterial.shininess);
```

.cpp

```
#include <qapplication.h>
#include <qimage.h>
#include "include/QSceneWindow.h"

#include "include/SLScene.h"
#include "include/SLLight.h"
#include "include/SLGroup.h"
#include "include/SLSphere.h"
#include "include/SLCylinder.h"
#include "include/SLMaterial.h"
#include "include/SLTextureGL.h"
#include "include/SLAppearance.h"
#include "include/SLImageRGB.h"
#include "include/SLPolygon.h"
#include "include/SLBox.h"

#include "../../Global/SLImage.h"

//-----------------------------------------------------------------------------


class FragmentLightingShader2 : public SLShader
{
public:
   FragmentLightingShader2() : SLShader
("../bin/shader/F_lighting2.vert","../bin/shader/F_lighting2.frag")
   {
   }

   void    update(void)
   {
      //myShader->sendUniform1i("numlight",2);
   }
};



//-----------------------------------------------------------------------------
int main( int argc, char** argv )
{

   QApplication a(argc, argv);
   QSceneWindow w;
   SLImage i;


   if (!QGLFormat::hasOpenGL()) qFatal("System has no OpenGL support!");
   a.setMainWidget(&w);
   w.resize(550, 350);
   w.setCaption("Image Demo");

   if (!i.load("../Textures/marsAlphaHeight.tif")) qFatal("Could not load texture
image.");
   i.convertBitsPerPixel(32);  // don't use convertDepth() anymore.

   i.mirror();
```

```
   // create texture

   SLTextureGL tex(  i.getData()
                    ,i.width()
                    ,i.height()
                    ,false
                    ,GL_NEAREST
                    ,GL_NEAREST
                    ,GL_REPEAT
                    ,GL_REPEAT
                    ,GL_MODULATE
                    ,i.getFormat()
                    ,false);


      //Lighting 2

   SLLight* light0 = new SLLight();
   light0->translate(0,2,2);
   light0->diffuse(SLCol4f(0.4,0.4,0));

   SLLight* light1 = new SLLight();
   light1->diffuse(SLCol4f(0.3,0,0));
   light1->translate(0,0,3);

   SLLight* light2 = new SLLight(0.3f);
   light2->lightAt(1.5, 1.5, 1.5);
   light2->spotCutoff(20);
   light2->diffuse(SLCol4f(0,1,0));

   SLLight* light3 = new SLLight(0.3f);
   light3->lightAt(-1.5, 1.5, 1.5);
   light3->spotCutoff(20);
   light3->diffuse(SLCol4f(0,0,1));

      SLGroup* scene = new SLGroup;
   scene->addNode(light0);
   scene->addNode(light1);
   scene->addNode(light2);
   scene->addNode(light3);
      SLSphere* sphere = new SLSphere(1.0,512,512);
   //scene->addNode(new SLSphere(1.0,128,128));


   FragmentLightingShader2               Flightshader2;

   scene->addNode(new SLAppearance(SLCol4f(1.0,1.0,1.0,1), SLCol4f(1,1,1,1),
100,0, &Flightshader2));
   scene->addNode(sphere);
   scene->addNode(new SLAppearance(SLCol4f(), SLCol4f(), 0, 0, 0)); // stop
shader

   w.root(scene);

   SLCamera* cam = new SLCamera;
   cam->lookAt(0,0,6);
   w.backColor(SLCol4f::GRAY);
   w.camera(cam);
   w.show();
   return a.exec();
```

```
}
```

6.3. Cook Torrance Beleuchungsmodell

Das Cook/Torrance Beleuchtungsmodell strebt physikalische Korrektheit an. Energieerhaltung, komplexe Substrikturen, Vorhersagbares Resultat. Das Cook/Torrance Modell benutzt parameter mit physikalischen Analogien.

6.3.1. Der Fresnel Term

Der Fresnel Term („F-Term“) kann durch die Schlick Approximation ersetzt werden.

Schlick-Approximation

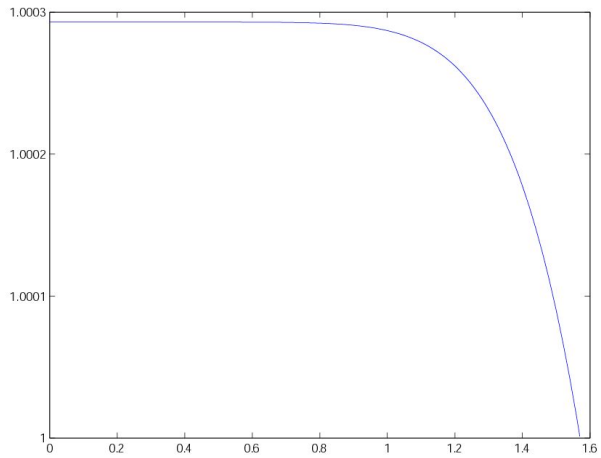
$$F_r(\theta) = F_0 + (1 - F_0)(1 - \cos\theta)^5 = F_0 + (1 - F_0)(1 - N \cdot E)^5$$

F_0: Brechungsindex [Engl. Refraction Index] (Reflexionskoeffizient bei senkrechter Einfallsrichtung)

Vakuum:	1.0
Luft:	1.000293
Wasser:	1.33333
Diamant:	2.417

θ : Winkel zwischen Normalen und Einfallsrichtung

```
float f = 2.417;       // Diamant
float F = f + (1-f) * pow(1.0 - dot(N,E), 5.0);
```



Beispiel: Schlick Approximation für Wasser. Einfallswinkel 0 bis pi/2

6.3.2. Cook Torrance Reflektions Modell

Blinn Mikrofacetten Verteilungs-Funktion

$$D = ce^{-\left(\frac{\alpha}{m}\right)^2}$$

α : Winkel zwischen N und H

Blinn hat auch zwei weitere Verteilungsfunktionen beschrieben, auf die hier nicht eingegangen wird.

Beckmann Mikrofacetten Verteilungs-Funktion (für rauhe Oberflächen)

$$D = \frac{1}{m^2 \cos^4 \alpha} e^{-\frac{\tan^2 \alpha}{m^2}}$$

m: Rauheit Oberfläche
Beispiele: Kohle=0.4, Gummi=0.3, Obsidian=0.15

Für die Verwendung im Shader, kann die Gleichung folgendermassen umgeformt werden:

$$\cos\alpha = N \cdot H$$

$$\tan^2 \alpha = \frac{\sin^2 \alpha}{\cos^2 \alpha} = \frac{1 - \cos^2 \alpha}{\cos^2 \alpha} = \frac{1 - (N \cdot H)^2}{(N \cdot H)^2}$$

$$\cos^4 \alpha = (N \cdot H)^4$$

Daraus folgt die Beckmann Mikrofacetten Verteilungsfunktion mit den bekannten Grössen N und H.

$$D = \frac{1}{m^2 (N \cdot H)^4} e^{-\frac{1-(N \cdot H)^2}{(N \cdot H)^2 m^2}}$$

6.3.3. Geometrische Abschwächung

Die Geometrische Abschwächung G ist die Stärke der Selbst-Schattierung der Mikrofacetten.

$$G = \min(1, \frac{2(N \cdot H)(N \cdot E)}{(E \cdot H)}, \frac{2(N \cdot H)(N \cdot L)}{(E \cdot H)})$$

In GLSL kann dies folgendermassen besschrieben werden:

```
float NH = dot(N,H);
float NE = dot(N,E);
float NL = dot(N,L);
float EH = dot(E,H);
float G = min(1.0, min((2.0 * NH * NL) / EH, (2.0 * NH * NE) / EH));
```

6.3.4. Spektrale Zusammensetzung des reflektieren Lichtes

Ambiente, Diffuse und Spekuläre Reflektionen hängen alle von der Wellenlänge ab. Ra, Rd und der F Term von Rs können von der ensprechenden Reflexionsspektrum des Materials erhalten werden. Reflektionsspektren wurden von Tausenden von Materialien gemessen und gesammelt.
Die real gemessenen Daten können z.B. in einer Texturmap als Funktion gespeichert werden.

Es kann jedoch auch die Schlick Approximation der Fresnelgleichung verwendet werden.

R_a, R_d hängen nicht von der Position des Betrachters ab, da sie das Licht in alle Richtungen gleichmässig reflektieren.

$$R_s = \frac{F}{\pi} \frac{D}{(N \cdot L)} \frac{G}{(N \cdot E)}$$

6.3.5. Implementation Cook-Torrance Lighting Gleichung

$$I = k_a + k_d \rho (L \cdot N) + k_s \frac{DGF_\lambda(\theta_i)}{\pi (E \cdot N)}$$

$$\rho = k_s f_{att} \cdot \frac{DGF_\lambda(\theta_i)}{\pi (L \cdot N)(E \cdot N)}$$

$$D = \frac{1}{m^2 (N \cdot H)^4} e^{-\frac{1-(N \cdot H)^2}{(N \cdot H)^2 m^2}}$$

Fragment Shader

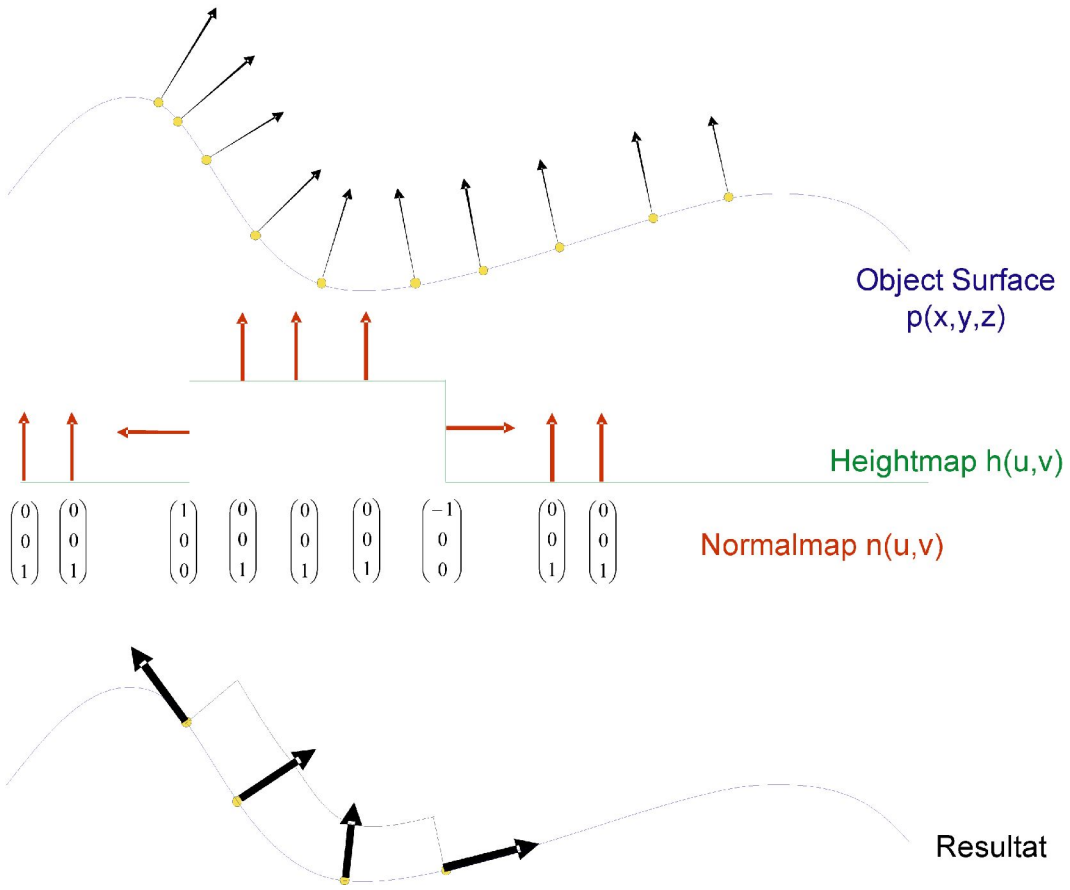
```
varying vec3 n;
varying vec3 P;
varying vec3 vOP;   // Vertex in Object Coordinates
varying vec3 T;     // Tangent

uniform sampler2D texture0;   // diffusemap

// Material:
uniform float m;
uniform float s;
uniform float ri;

void main (void)
{
   float d = 1.0-s;
   float a = 0.1;                    //ambient intensity
   vec4 Ks = vec4(0.6,0.6,0.6,1.0); //gl_FrontLightProduct[0].diffuse;
   vec4 Ka = vec4(0.1,0.1,0.5,1.0); //ambient color
   float f = ri;
   float dw = 1.5;                  // unit solid angle

   // Textur:
   vec4 tex0 = texture2D(texture0, vec2(gl_TexCoord[0]));
   //vec4 tex0 = vec4(0.0,0.0,0.0,0.0);
```

```
    // Standard-Vektoren
    vec3 N = normalize(n);
    vec3 L = normalize(gl_LightSource[0].position.xyz-P);
    vec3 E = normalize(-P);
    //vec3 R = normalize(-reflect(L,N));
    vec3 H = normalize(E+L);

    float NH = dot(N,H);
    float NE = dot(N,E);
    float NL = dot(N,L);
    float EH = dot(E,H);

    // Fresnel Term: Schlick Approximation
    float F = f + (1.0-f) * pow(1.0 - NE, 5.0);

    // Geometrische Abschwächung
    float G = min(1.0, min((2.0 * NH * NL) / EH, (2.0 * NH * NE) / EH));

    // Beckmann Mikrofacetten Verteilungsfunktion
    float mq=m*m;
    float NHq = NH*NH;
    float D = (1.0 / mq * NHq * NHq) * (pow(2.7, -((1.0 - NHq) / (mq * NHq))) / (mq * NHq * NHq));

    //Alternative: Blinn Mikrofacetten Verteilungs-Funktion:
    //float c = 1.0;
    //float bla = acos(NH)/m;
    //float D = c * pow(2.7,-bla*bla);

    vec4 Rd = tex0;  // Diffuse-Wert = Texturwert

    // Spekulärer Term:
    vec4 Rs = F*D*G/(3.1415926*NE*NL) * Ks;

    // Farbe
    gl_FragColor = a * Ka + NL * dw * (s * Rs + d * Rd);   // * 4.0 * clamp(NL, 0.0, 0.25);

}
```

Screenshots: Cook Torrance Beleuchtunsmodell

Materialwerte: m=0.15 (links) und m=0.40 (rechts)

6.4. Bump Mapping



Vertex Shader

```
varying vec3 L;         // interpolated surface local coordinate light direction
varying vec3 E;     // interpolated surface local coordinate view direction
varying vec3 N;

void main(void)
{
   gl_Position  = gl_ModelViewProjectionMatrix * gl_Vertex;
   gl_TexCoord[0] = gl_MultiTexCoord0;                    // uv

   mat3 TBN;

   TBN[0] = gl_NormalMatrix * vec3 (gl_MultiTexCoord1);  // Tangente
   TBN[1] = gl_NormalMatrix * vec3 (gl_MultiTexCoord2);  // Binormale
   TBN[2] = gl_NormalMatrix * gl_Normal;                 // Normale

   vec4 P = gl_ModelViewMatrix * gl_Vertex;

   E = normalize(TBN * vec3(-P));
   L = normalize(TBN * vec3(gl_LightSource[0].position - P));

}
```

Fragment Shader

```
uniform sampler2D myTexture1;
uniform sampler2D myTexture2;
varying vec3 L;         // interpolated surface local coordinate light direction
varying vec3 E;     // interpolated surface local coordinate view direction
varying vec3 N;


void main (void)
{
    vec3 N2;

    // Fetch normal from normal map
    vec4 textur = texture2D(myTexture1, vec2 (gl_TexCoord[0]));

    N2 = vec3(texture2D(myTexture2, vec2 (gl_TexCoord[0])));
    N2 = (N2 - 0.5) * 2.0;
    N2.y = -N2.y;

    N2 = normalize(N2);

    vec3 Ln = normalize(L);
    vec3 En = normalize(E);

    float e = 8.0 * clamp(L[2],0.0,0.125);  // N is (0 0 1) => N dot L is L[2]

    vec3 R = normalize(-reflect(Ln,N2));

    vec4 Iamb = gl_FrontLightProduct[0].ambient;
    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N2,Ln), 0.0);
    vec4 Ispec = gl_FrontLightProduct[0].specular * pow(max(dot(R,En),0.0),
gl_FrontMaterial.shininess);

    gl_FragColor =  textur * ( Iamb + e* (Idiff + Ispec));
}
```

7. Ausblick

GPU Programmierung ist mit der heutigen Generation (GeForce FX, Radeon 9500-9800 etc.) von Grafikkarten immer noch **sehr limitiert**.

Die **Anzahl Instruktionen** muss erhöht werden und die Instruktionsarten um ein generisches "while" erweitert werden, was in Pixel Shader 3.0 kompatiblen Karten der Fall sein wird. Die **Busgeschwindigkeit** um Daten vom VRAM in das "normale" RAM zu kopieren ist zu langsam.

PCI Express und die neuen Generationen von Grafikkarten (ATI X800 und NVidia 6800) werden die Anzahl Instruktionen erhöhen und auch erlauben generischer zu Programmieren. Die NVidia 6800 kann schon 65535 Instruktionen gegenüber 96 Instruktionen auf den Grafikkarten der FX Generation.

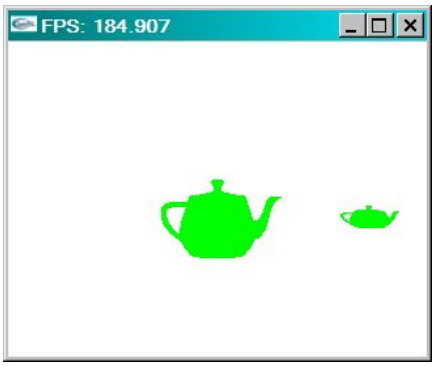
NVidia hat Ende Juni 2004 das **SLI / Multi-GPU Modell** mit PCI-Express angekündigt, welche durch Verwendung von zwei Grafikkarten die Performance deutlich steigern wird.

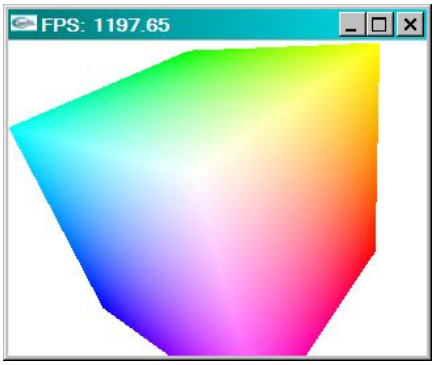
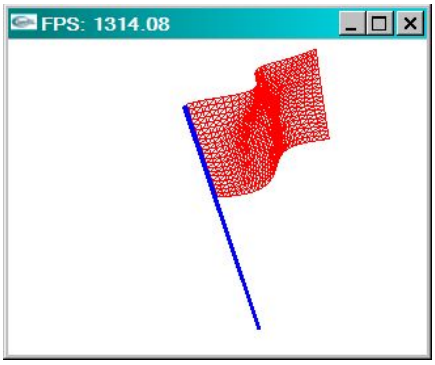
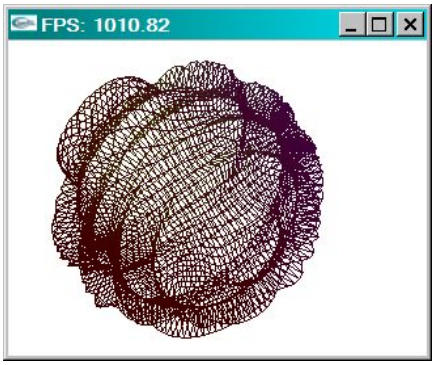
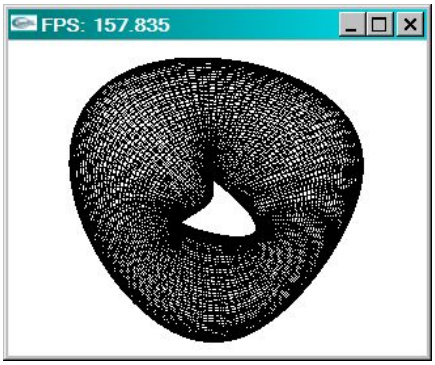
Weiterhin sind **OpenGL Extensions** geplant, welche die Kommunikation mit der Grafikkarte erleichtern sollen.

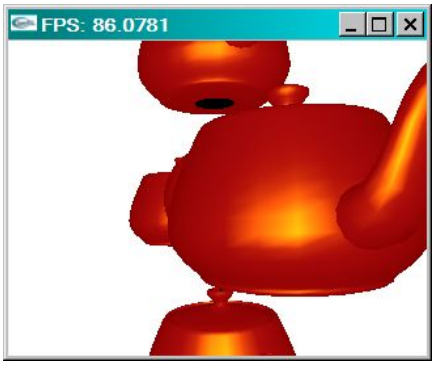
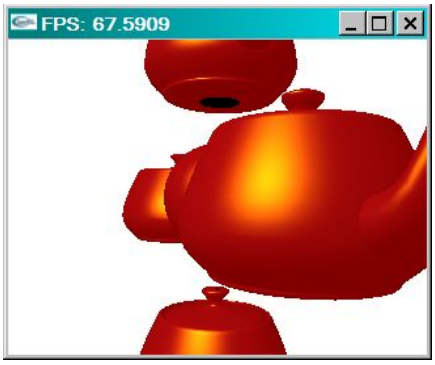
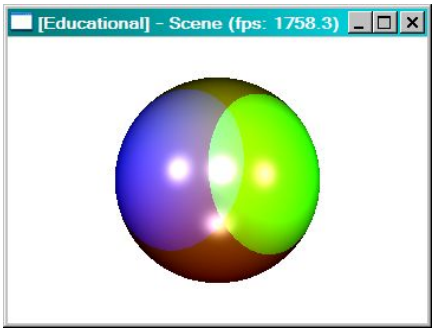
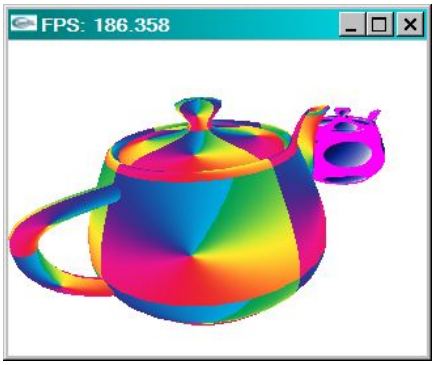
Wir denken die GPU Programmierung wird in Zukunft – auch für nichtgraphische Anwendungen - sehr grosse Bedeutung erhalten.

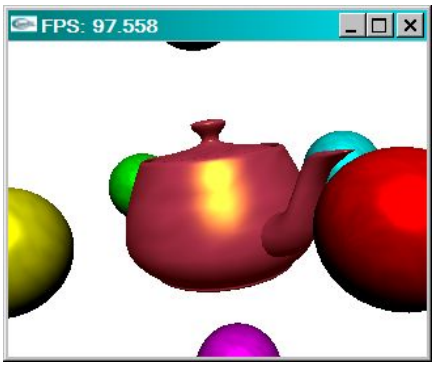
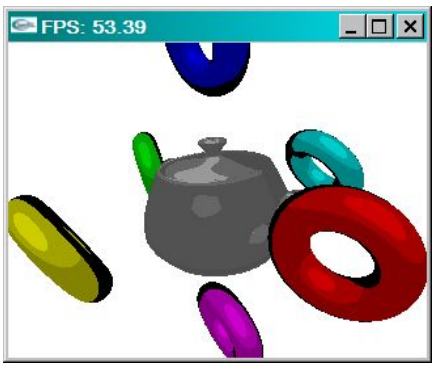
8. Liste unserer Shader Programme

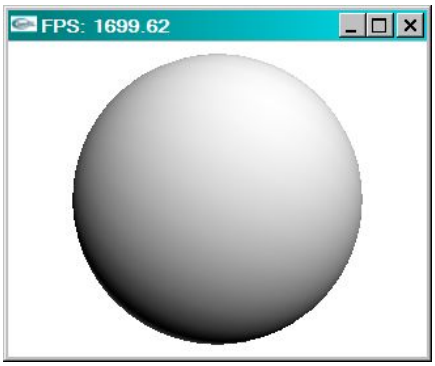
Die Programme sind als .exe Files + dem Shader Source Code vorhanden, jedoch benötigt man eine Grafikkarte mit OpenGL Shading Language Unterstützung, um sie ausführen zu können.

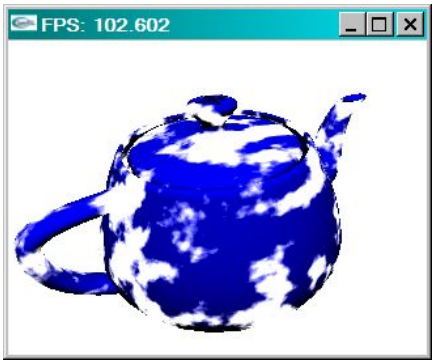
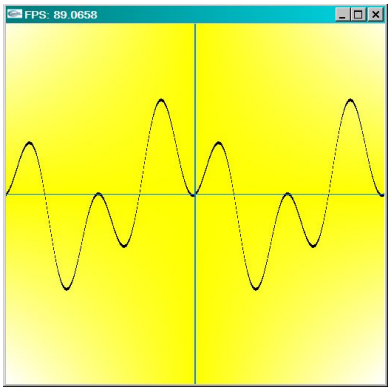
Screenshot	*Zweck*	*Filenamen*
	**Einführungs-Beispiel** Beispiel: Skalierung eines Objektes (Vertex Shader) und Zuweisung einer Farbe (Fragment Shader)	Project 1.01 – Load, Compile **Shader:** tutorial1.frag tutorial1.vert **Ausführbare Datei:** Tutorial_1_01.exe
	**Uniforme Variablen im Vertex Shader** Beispiel, welches den Umgang mit Uniformen Variablen im Vertex Shader zeigt. Eine uniforme Variable wird vom C++ Progeamm an den Shader "übergeben". Der Wert variiert zwischen 0 und 1 und bewirkt eine Skalierung entlang der z-Achse.	Project 1.02 – Uniforms (Vertex) **Shader:** tutorial2.frag tutorial2.vert **Ausführbare Datei:** Tutorial_1_02.exe
	**Uniforme Variablen im Fragment Shader** Beispiel, welches Uniforme Variablen im Fragment Shader zeigt. Eine uniforme Variable steuert die Farbe im Fragment shader.	Project 1.03 – Uniforms (Fragment) **Shader:** tutorial3.frag tutorial3.vert **Ausführbare Datei:** Tutorial_1_03.exe

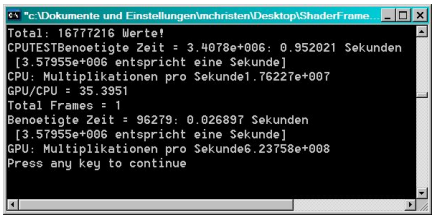
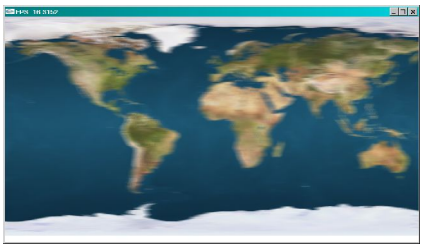
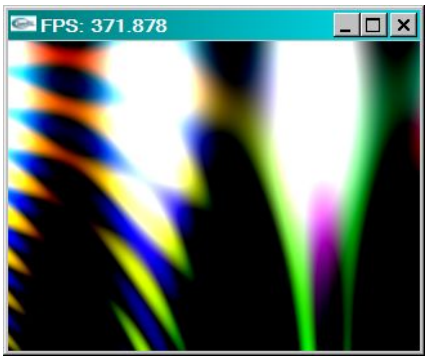
Screenshot	*Zweck*	*Filenamen*
	**Varying Variablen** Dieser Shader zeigt den Umgang mit varying variablen. Farbwerte werden entlang des Würfels interpoliert, was zu dem Farb Würfel führt.	Project 1.04 – Varying **Shader:** tutorial4.frag tutorial4.vert **Ausführbare Datei:** Tutorial_1_04.exe
	**Flag Shader** Eine Flagge wird – mittels Vertex Shader – animiert.	Project – Flagshader **Shader:** flag.frag flag.vert **Ausführbare Datei:** flag.exe
	**Warp** Die Vertices werden so verändert, dass ein "Warp" Effekt (animiert) entsteht.	Projekt – Warp **Shader:** warp.frag warp.vert **Ausführbare Datei:** warp.exe
	**Torus** Ein Torus wird aus einem Grid durch entsprechende Vertex Transformation im Shader erstelllt. Der Torus kann sich (optional) entlang einer Lissajous Figur bewegen.	Projekt – Torus **Shader:** torus.frag torus.vert **Ausführbare Datei:** torus.exe

Screenshot	*Zweck*	*Filenamen*
	**Per Vertex Lighting** OpenGL Lighting (mit einer Lichtquelle) soll simuliert werden. Licht wird nur Pro Vertex berechnet und Linear interpoliert.	Project 1.05a – Per Vertex Light **Shader:** tutorial5a.frag tutorial5a.vert **Ausführbare Datei:** Tutorial_1_05a.exe
	**Per Fragment Lighting** Die per Vertex Lighting implementation wird erweitert, so dass nun Punktlichter auf Pixelebene berechnet werden.	Project 1.05b– Per Fragment Light **Shader:** tutorial5b.frag tutorial5b.vert **Ausführbare Datei:** Tutorial_1_05b.exe
	**Multi Light Per Fragment Lighting** Spotlight und Punktlicht mit bis zu 4 Lichtquellen (mehr als 4 Lichtquellen sind ab NVidia 6800 Karten möglich)	Project Lighting Advanced **Shader:** tutorial5b.frag tutorial5b.vert **Ausführbare Datei:** lighting_qt.exe
	**Texturen** Dieses Programm zeigt, wie man Objekte Texturiert. Es werden die Texturkoordinaten, welche im OpenGL Programm definiert wurden verwendet.	Project 1.06– Textures **Shader:** tutorial6.frag tutorial6.vert **Ausführbare Datei:** Tutorial_1_06.exe

Screenshot	*Zweck*	*Filenamen*
	**Color Key** In diesem Beispiel wird gezeigt, wie man mit dem "discard" Befehle Fragmente auslässt.	Projekt 1.07 – Color Key **Shader:** tutorial7.frag tutorial7.vert **Ausführbare Datei:** Tutorial_1_07.exe
	**Multitexturing** Zwei Texturen werden "übereinandergelegt", jeweils 50% transparent. Dieses Shaderprogramm zeigt den Umgang mit mehreren Texturen.	Projekt 1.07 – Multitexturing **Shader:** tutorial8.frag tutorial8.vert **Ausführbare Datei:** Tutorial_1_08.exe
	**Bumpy Effect** Dieser Shader legt ein Störungs-Muster über ein vorhandenes Objekt, so dass der Eindruck entsteht, das Objekt sei uneben.	Projekt 1.09 – Bumpy Effect **Shader:** tutorial9.frag tutorial9.vert **Ausführbare Datei:** Tutorial_1_09.exe
	**Toon Shading** NPR – Non Photorealistic Rendering, in diesem Fall ein Ein-Pass Toon-Shading wird mit diesem Shader implementiert.	Projekt 1.10 – Simple Toon Shading **Shader:** tutorial10.frag tutorial10.vert **Ausführbare Datei:** Tutorial_1_10.exe

Screenshot	*Zweck*	*Filenamen*
	**Cook Torrance Beleuchung** Dieser Shader implementiert das Beleuchtungsmodell, welches 1981 von Cook/Torrance vorgestellt wurde.	Projekt 1.13 – Cook-Torr **Shader:** tutorial13.frag tutorial13.vert **Ausführbare Datei:** Tutorial_1_13.exe
	**Bump Mapping** Implementation von Bump Mapping mit Normalmap.	Projekt 1.15 – Bumpmap Demo **Shader:** tutorial15.frag tutorial15.vert **Ausführbare Datei:** Tutorial_1_15.exe
	**Funktionsübergabe über Textur** Eine Funktion [hier: abs(x)] wird als 1D Textur dem Shader übergeben und als Grauwert auf eine Kugel (entlang der x-Achse) projiziert.	Projekt 2.01 – Texture Communication **Shader:** example-2-01.frag example-2-01.vert **Ausführbare Datei:** Tutorial_2_01.exe

Screenshot	*Zweck*	*Filenamen*
  	**Prozedurale Texturen mit Noise** Eine 3D Textur wird mit Werten der Perlin Noise Funktion aufgefüllt. Der Shader erhält darauf Zugriff und implementiert eine Turbulenz und Cloud Funktion (Summen von Noise) welche zu Unterschiedlichsten Effekten führen können.	Projekt 2.02 – Noise **Shader:** texture3d.frag texture3d.vert **Ausführbare Datei:** Tutorial_2_02.exe
	**Funktionsplotter** Ein Quad Array wird dem Shader übergeben. In dieses zeichnet er (implizit) eine Funktion. Die Ausgabe erfolgt nicht auf den Bildschirm, sondern in eine Textur.	Projekt 2.03 – Function Plotter **Shader:** pbuffer.frag pbuffer.vert **Ausführbare Datei:** Tutorial_2_03.exe

Screenshot	*Zweck*	*Filenamen*
	**Buffer Communication** Eine Funktion wird auf der GPU in einem bestimmten Bereich ausgewertet und in einer 32 Bit Textur geschrieben. Die Daten werden zurück an die CPU gesendet und können dort weiter verwendet werden.	Projekt 2.04 – Buffer Communication **Shader:** calc.frag calc.vert **Ausführbare Datei:** Tutorial_2_04.exe
	**Image Processing** Ein Bild erhält eine Verwischung (Blur). Eine Textur wird als Eingabe verwendet und es wird in eine Textur zurückgeschrieben.	Projekt 2.05 – Image Operations **Shader:** image.frag image.vert image2.frag image2.vert **Ausführbare Datei:** Tutorial_2_05.exe
	**Plot2D**	Projekt 2.06 – PlotXY **Shader:** plotxy.frag plotxy.vert **Ausführbare Datei:** Tutorial_2_06.exe

9. Quellen

Nummer	*Titel*	*Autor(en) und Jahr*	*Link/Datei*
[01]	The Cg Tutorial Book	Randima Fernano, Mark J. Kilgard, 2003	
[02]	Cg Toolkit User's Manual – A Developer's Guide to Programmable Graphics, Version 1.1	Nvidia Corporation, 2003	C:\Programme\Nvidia Corporation\Cg\docs\Cg_Toolkit.pdf
[06]	A Reflectance Model for Computer Graphics	Robert L. Cook, Kenneth R. Torrance, 1981	
[07]	A Practical and Robust Bump-mapping Technique for Today's GPUs	Mark J. Kilgard, 2000	PracticalBumpMap.pdf
[08]	NVIDIA Cg Runtime	Nvidia Corporation	cgRuntime.chm
[09]	OpenGL ARB Vertex Program, OpenGL ARB Fragment Program	Cass Everitt, 2003	
[10]	OpenGL Shading Language Tutorials	Martin Christen, 2003/04	http://www.clockworkcoders.com/oglsl
[11]	ATI RenderMonkey IDE	ATI, 2003	RenderMonkey_Documentation.pdf
[12]	Direct3D® High Level Shader Language Programming using RenderMonkey™ IDE	Natasha Tatarchuk, 2003	HLSL_Programming_with_RenderMonkey-GDC_2003.pdf
[13]	OpenGL Shading Language	Randi J. Rost, 2004	
[15]	The OpenGL Shading Language, Siggraph 2003, Course 22	Randi Rost, Bill Licea-Kane, 2003	
[16]	The OpenGL Shading Language 1.051	John Kessenich, Dave Baldwin, Randi Rost, 2003	
[17]	A Non-Photorealistic Fragment Shader in OpenGL 2.0	Bert Freudenberg	
[18]	Texturing & Modeling – A Procedurl Approach, 3rd Edition	David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, 2003	
[19]	A Class of Local Interpolating Splines. In Computer Aided Geometric Design, R.E. Barnhill and R.F. Reisenfeld, Eds. Academic Press, New Yark, 1974, pp. 317-326	E. Catmull, R. Rom, 1974	
[20]	NVidia GPU Programming Guide Version 2.0.3 – July 2004.	NVIDIA, 2004	