

## Beispiel: Hello\_GLSL



lighthouse\_tutorial/hello\_gsl\*

G. Zachmann Computer-Graphik 2 - SS 07 Shader 49

## Inspektion der Parameter eines GLSL-Programms

- **Attribut-Variablen:**
  - `glProgramiv()` : liefert die Anzahl aktiver "attribute"-Parameter
  - `glGetActiveAttrib()` : liefert Info über ein bestimmtes Attribut
  - `glGetAttribLocation()` : liefert einen Handle ein Attribut
- **Uniform-Variablen:**
  - `glProgramiv()` : liefert die Anzahl aktiver "uniform"-Parameter
  - `glGetActiveUniform()` : liefert Info zu einem Parameter
- Benötigt man vor allem zur Implementierung von sog. Shader-Editoren

G. Zachmann Computer-Graphik 2 - SS 07 Shader 50

## Setzen von "uniform"-Variablen

- Erst `glUseProgram()`
- Dann Handle auf Variable besorgen:
 

```
uint var_handle = glGetUniformLocation( progr_handle,
                                       "uniform_name" )
```
- Setzen einer uniform-Variablen:
  - Für Float:
 

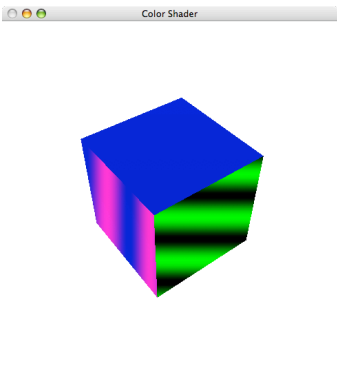
```
glUniform1f( var_handle, f )
```
  - Für Matrizen
 

```
glUniform4fv( var_handle, count, transpose, float * v )
```

analog gibt es `glUniform{2,3}fv`

G. Zachmann Computer-Graphik 2 - SS 07 Shader 51

## Beispiel für uniform-Variablen



G. Zachmann Computer-Graphik 2 - SS 07 Shader 52

### Die spezielle Funktion `ftransform`

- Tut genau das, was die fixed-function pipeline in der Vertex-Transformations-Stufe auch tut: einen Vertex von Model-Koordinaten in View-Koordinaten abbilden
- Idiom:
 

```
gl_Position = ftransform();
```
- Identisch:
 

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 53

### Beispiel für die Modifikation der Geometrie

- Wie man mit den Koordinaten (und sonstigen Attributen) eines Vertex im Vertex-Shader verfährt, ist völlig frei:



G. Zachmann Computer-Graphik 2 - SS 07 Shader 54

### Zustandsvariablen

- Zeigen den aktuellen Zustand von OpenGL an
- Sind als "*uniform*"-Variablen implementiert
- Die aktuellen Matrizen:

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat3 gl_NormalMatrix;
uniform mat4 gl_TextureMatrix[n];
uniform mat4 gl_*MatrixInverse;
```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 55

- Das aktuelle Material:

```
struct gl_MaterialParameters
{
    vec4 emission;
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    float shininess;
};
uniform gl_MaterialParameters gl_FrontMaterial;
```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 56

- Aktuelle Lichtquellen(-Parameter):

```


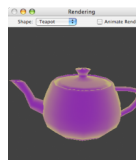
struct gl_LightSourceParameters
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec4 position;
    vec4 halfVector;
    vec3 spotDirection;
    float spotExponent;
    float spotCutoff;
    float spotCosCutoff;
    float constantAttenuation;
    float linearAttenuation;
    float quadraticAttenuation;
};
uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];

```

- Und viele weitere (z.B. zu Texturen, Clipping Planes,...)

G. Zachmann Computer-Graphik 2 - SS 07 Shader 57

- Beispiel für Verwendung von varying- und Zustands-Variablen

- Der "Toon-Shader":
  - Berechnet einen stark diskretisierten diffusen Farbanteil (typ. 3 Stufen)
- Der "Gooch-Shader":
  - Interpoliert zwischen 2 Farben, abhängig vom Winkel zwischen Normale und Lichtvektor
- Sind schon einfache Beispiele für "non-photorealistic rendering" (NPR)

G. Zachmann Computer-Graphik 2 - SS 07 Shader 58

## Attribute

- Vordefiniert:

```

attribute vec4 gl_Vertex;
attribute vec3 gl_Normal;
attribute vec4 gl_Color;
attribute vec4 gl_MultiTexCoord[n];
attribute vec4 gl_SecondaryColor;
attribute float gl_FogCoord;

```

- Man kann selbst Attribute definieren:
  - Im Vertex-Shader: `attribute vec3 myAttrib;`
  - Im C-Programm:

```

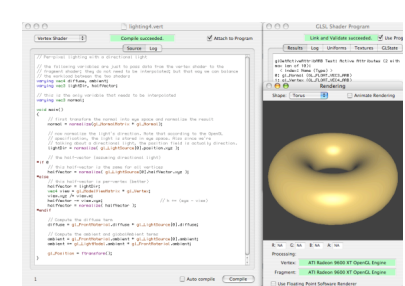
handle = glGetAttribLocation( prog_handle, "myAttrib" );
glVertexAttrib3f( handle, v1, v2, v3 );

```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 59

## Beispiel: Per-Pixel Lighting

1. Diffuse lighting per-vertex
2. Mit ambientem Licht
3. Mit spekularem Lichtanteil
4. Per-Pixel Lighting



GLSL\_editor/lighting[1-4].\*

G. Zachmann Computer-Graphik 2 - SS 07 Shader 60

### Achtung bei Subtraktion homogener Punkte

- Homogener Punkt = `vec4 (v.xyz, v.w)`
  - 3D-Äquivalent =  $v.xyz/v.w$
- Subtraktion zweier Punkte/Vektoren:
  - Homogen:  $v - e$
  - Als 3D-Äquivalent:
 
$$\frac{v.xyz}{v.w} - \frac{e.xyz}{e.w} = \frac{v.xyz \cdot e.w - e.xyz \cdot v.w}{v.w \cdot e.w}$$
- Normalisierung:
 
$$\text{normalize}(v.xyz/v.w) = \text{normalize}(v.xyz)$$
- Zusammen :
 
$$\text{normalize}(v-e) = \text{normalize}(v.xyz \cdot e.w - e.xyz \cdot v.w)$$

G. Zachmann Computer-Graphik 2 - SS 07 Shader 61

### Zugriff auf Texturen im Shader

- Deklariere Textur im Shader (Vertex oder Fragment):
 

```
uniform sampler2D myTex;
```
- Lade und binde Textur im C-Programm wie gehabt:
 

```
glBindTexture( GL_TEXTURE_2D, myTexture );
glTexImage2D(...);
```
- Verbinde beide:
 

```
uint mytex = glGetUniformLocation( prog, "myTex" );
glUniform1i( mytex, 0 ); // 0 = texture unit, not ID
```
- Zugriff im Fragment-Shader:
 

```
vec4 c = texture2D( myTex, gl_TexCoord[0].xy );
```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 62


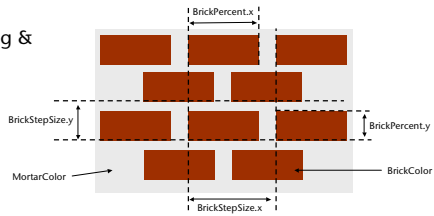
### Beispiel: eine einfache "Gloss-Textur"



GLSL\_editor/gloss.{frag,vert}

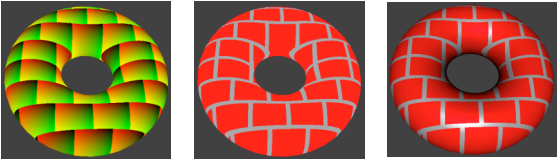
G. Zachmann Computer-Graphik 2 - SS 07 Shader 63

### Eine einfache prozedurale Textur

- Ziel: Ziegelstein-Textur
 
- Vereinfachung & Parameter:
 

G. Zachmann Computer-Graphik 2 - SS 07 Shader 64

- Generelle Funktionweise:
  - Vertex-Shader: normale Beleuchtungsrechnung
  - Fragment-Shader:
    - bestimme pro Fragment anhand der xy-Koordinaten des zugehörigen Punktes im Objektraum, ob der Punkt im Ziegel oder im Mörtel liegt
    - danach, entsprechende Farbe mit Beleuchtung multiplizieren
- Beispiele:
 



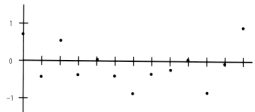
G. Zachmann Computer-Graphik 2 - SS 07 Shader 65

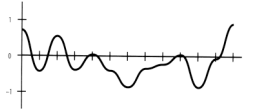
## Rauschen

- Die meisten prozeduralen Texturen sehen zu "clean" aus
- Idee: addiere Rauschen (Schmutz), für realistischeres Aussehen
- Gewünschte Eigenschaften einer Rausch-Funktion:
  - Stetig
  - Es reicht, wenn sie (nur) zufällig aussieht
    - Keine offensichtlichen Muster / Wiederholungen
  - Wiederholbar (gleiche Ausgabe bei gleichem Input)
  - Wertebereich  $[-1,1]$
  - Kann für 1–4 Dimensionen definiert werden
  - Isotrop (invariant unter Rotation)

G. Zachmann Computer-Graphik 2 - SS 07 Shader 66

- Einfache Idee, am 1-dimensionalen Beispiel:
  - Wähle zufällige y-Werte aus  $[-1,1]$  an den Integer-Stellen:
 



  - Interpoliere dazwischen, z.B. kubisch (linear reicht nicht):
 


- Diese Art Rauschfunktion heißt "*value noise*"


G. Zachmann Computer-Graphik 2 - SS 07 Shader 67

- Generiere mehrere Rausch-funktionen mit verschiedenen Frequenzen
 


Frequency = 1  
amplitude = 0.5




sum of 3 noises




Frequency = 5  
amplitude = 0.1




sum of 3 noises




Frequency = 10  
amplitude = 0.05




sum of 3 noises



Frequency = 100  
amplitude = 0.005

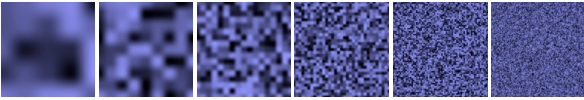
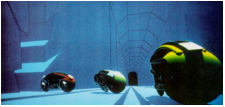




sum of 3 noises

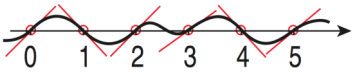
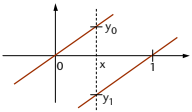

- Addiere alle diese zusammen
 

§ Ergibt Rauschen auf verschiedenen "Skalen"

G. Zachmann Computer-Graphik 2 - SS 07 Shader 68

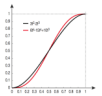
- Dasselbe in 2D:
 
- Ergebnis
 
- Lässt sich leicht verallgemeinern in höhere Dimensionen
- Heißt auch *Perlin noise*, *pink noise*, oder *fractal noise*
  - Ken Perlin; hat sich zuerst damit beschäftigt bei der Arbeit an TRON
 


G. Zachmann Computer-Graphik 2 - SS 07 Shader 69

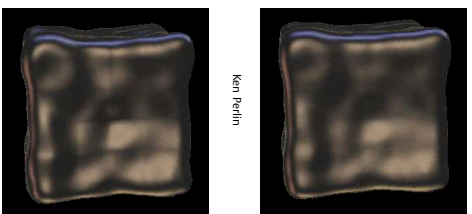
- **Gradient noise:**
  - Spezifiziert die **Gradienten** an den Integer-Stellen (statt den Werten):
 
  - Interpolation:
    - Berechne  $y_0$  und  $y_1$  als Wert der Geraden durch 0 und 1 mit den vorgegebenen (zufälligen) Gradienten
    - Interpoliere  $y_0$  und  $y_1$  mit einer Blending-Funktion, z.B.
 

oder

$$h(t) = 3t^2 - 2t^3$$

$$q(t) = 6t^5 - 15t^4 + 10t^3$$


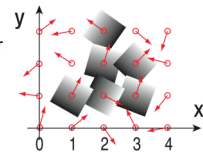
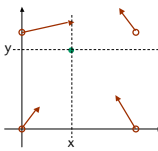
G. Zachmann Computer-Graphik 2 - SS 07 Shader 70

- Vorteil der quintischen Blending-Funktion: 2-te Ableitung bei  $t=0$  und  $t=1$  ist 0 → die gesamte Noise-Funktion ist  $C^2$ -stetig
  - Beispiel, wo man das sieht:
 

G. Zachmann Computer-Graphik 2 - SS 07 Shader 71

- **Gradient noise im 2D:**
  - Gebe an Integer-Gitterpunkten Gradienten vor (2D Vektoren, **nicht** notw.weise mit Länge 1)
  - Interpolation (wie im 1D):
    - O.B.d.A.  $P = (x,y)$  in  $[0,1] \times [0,1]$
    - Seien die Gradienten  $g_{00} = \text{Gradient an } (0,0)$ ,  $g_{01} = \text{Gradient an } (0,1)$ ,  $g_{10} = \text{Gradient an } (1,0)$ ,  $g_{11} = \text{Gradient an } (1,1)$
    - Berechne den Wert der "Gradienten-Rampen" am Punkt P:
 
$$z_{00} = g_{00} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \quad z_{01} = g_{01} \cdot \begin{pmatrix} x \\ y-1 \end{pmatrix}$$

$$z_{10} = g_{10} \cdot \begin{pmatrix} x-1 \\ y \end{pmatrix} \quad z_{11} = g_{11} \cdot \begin{pmatrix} x-1 \\ y-1 \end{pmatrix}$$

G. Zachmann Computer-Graphik 2 - SS 07 Shader 72

- Blending der 4 "z"-Werte durch bilineare Interpolation:

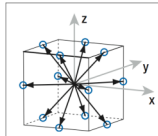
$$z_{x0} = (1 - a(x))z_{00} + a(x)z_{10}, \quad z_{x1} = (1 - a(x))z_{01} + a(x)z_{11}$$

$$z_{xy} = (1 - a(y))z_{x0} + a(y)z_{x1}$$

- Analog im 3D:
  - Spezifiziere Gradienten auf einem 3D-Gitter
  - Werte  $2^3=8$  "Gradienten-Rampen" aus
  - Interpoliere diese mit trilinearier Interpolation und der Blending-Fkt
- Und im  $d$ -dim. Raum? → Aufwand ist  $O(2^d)!$

G. Zachmann Computer-Graphik 2 - SS 07 Shader 73

- Ziel: **wiederholbare** Rauschfunktion
  - D.h.,  $f(x)$  liefert bei **gleichem**  $x$  immer **denselben** Wert
- Wähle feste Gradienten an den Gitterpunkten
- Beobachtung: es genügen einige wenige verschiedene
  - Z.B. für 3D genügen Gradienten aus dieser Menge:



$$g_0 = (0, 1, 1), g_1 = (0, 1, -1),$$

$$g_2 = (0, -1, 1), g_3 = (0, -1, -1),$$

$$g_4 = (1, 0, 1), g_5 = (1, 0, -1),$$

$$g_6 = (-1, 0, 1), g_7 = (-1, 0, -1),$$

$$g_8 = (1, 1, 0), g_9 = (1, -1, 0),$$

$$g_{10} = (-1, 1, 0), g_{11} = (-1, -1, 0)$$

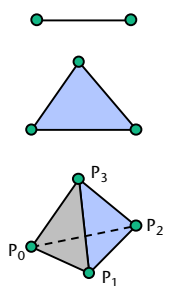
- Integer-Koordinaten der Gitterpunkte werden einfach gehasht → Index in eine Tabelle vordefinierter Gradienten

G. Zachmann Computer-Graphik 2 - SS 07 Shader 74

- **d-dimensionaler Simplex** :=
  - Verbindung von  $d + 1$  affin unabhängigen Punkten
- Beispiele:
  - 1D: Linie , 2D: Dreieck , 3D: Tetraeder
- Allgemein:
  - Punkte  $P_0, \dots, P_d$
  - Simplex = alle Punkte  $X$  mit

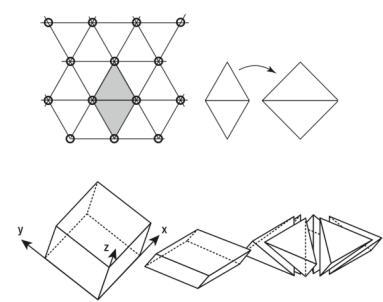
$$X = P_0 + \sum_{i=1}^d s_i \mathbf{u}_i$$

mit

$$\mathbf{u}_i = P_i - P_0, \quad s_i \geq 0, \quad \sum_{i=0}^d s_i \leq 1$$


G. Zachmann Computer-Graphik 2 - SS 07 Shader 75

- Mit **gleichseitigen**(!)  $d$ -dimensionalen Simplexes kann man den  $d$ -dim. Raum partitionieren (tessellieren):



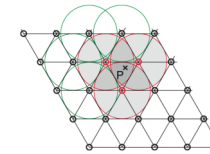
G. Zachmann Computer-Graphik 2 - SS 07 Shader 76

- Generell gilt:

- Ein  $d$ -dimensionaler Simplex hat  $d+1$  Ecken
- Mit gleichseitigen  $d$ -dimensionaler Simplices kann man einen Würfel partitionieren, der entlang seiner Diagonalen geeignet "gestaucht" wurde
- Solch ein  $d$ -dim. gestauchter Würfel enthält  $d!$  viele Simplices

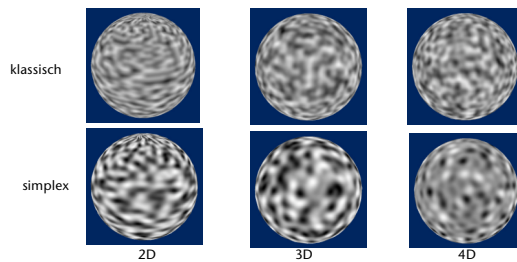
- Konstruktion der Noise-Funktion über einer Simplex-Partitionierung (daher "simplex noise"):

- Bestimme den Simplex, in dem ein Punkt P liegt
- Bestimme alle dessen Ecken und die Gradienten in den Ecken
- Bestimme (wie vorher) den Wert dieser "Gradienten-Rampen" in P
- Bilde eine gewichtete Summe dieser Werte
- Wähle dabei Gewichtungsfunktionen so, daß der "Einfluß" eines Simplex-Gitter-Punktes sich gerade nur die inzidenten Simplices erstreckt



- Ein großer Vorteil: nur noch Aufwand  $O(d)$

- Zu den Details siehe "Simplex noise demystified" (auf der Homepage der Vorlesung)
- Vergleich zwischen klassischem Perlin-Noise und Simplex-Noise:



- Im GLSL-Standard werden 4 noise-Funktionen definiert:  
`float noise1(gentype), vec2 noise2(gentype),  
vec3 noise3(gentype), vec4 noise4(gentype).`

- Aufruf einer Noise-Funktion:

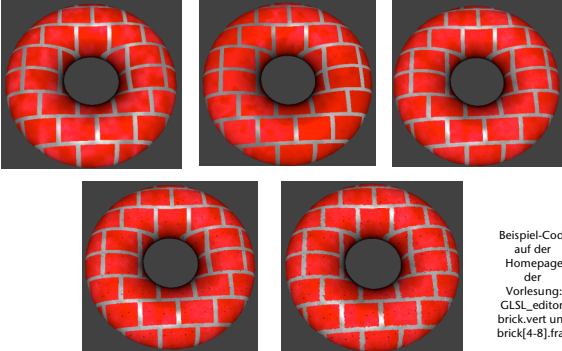
$$v = \text{noise2}(f*x + t, f*y + t)$$

- Mit  $f$  kann man die räumliche Frequenz steuern, mit  $t$  kann man eine Animation erzeugen ( $t$ ="Zeit").
- Analog für 1D- und 3D-Noise
- Achtung: Wertebereich ist  $[-1,+1]$ !
- Nachteile:
  - Sind nicht überall implementiert
  - Sind laaangsam ...



## Beispiel

- Unsere prozedurale Ziegelstein-Textur:



Beispiel-Code  
auf der  
Homepage  
der  
Vorlesung:  
GLSL\_editor/  
brick.vert und  
brick[4-8].frag

G. Zachmann Computer-Graphik 2 - SS 07 Shader 81