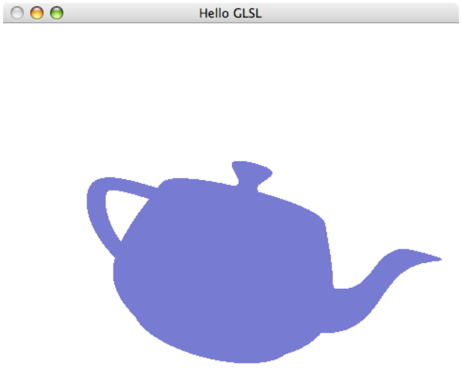
Beispiel: Hello_GLSL



lighthouse_tutorial/hello_glsl*

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

Setzen von "*uniform*"-Variablen

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

```
uint var_handle = glGetUniformLocation( progr_handle,
                                        "uniform_name" )
```

- Setzen einer uniform-Variable:
  - Für Float:

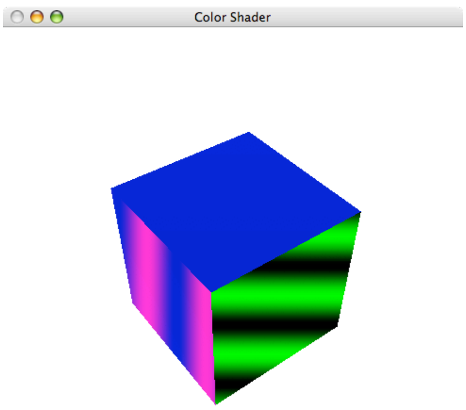
```
glUniform1f( var_handle, f )
```

  - Für Matrizen

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

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

Beispiel für uniform-Variable



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:

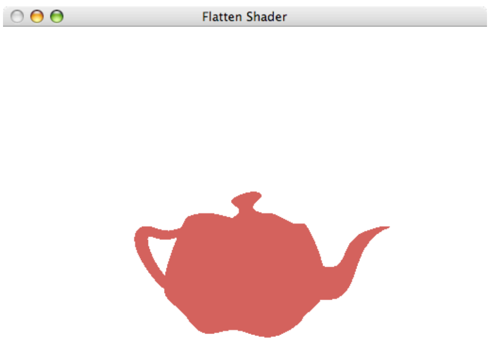
```
gl_Position = ftransform();
```

- Identisch:

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


Beispiel für die Modifikation der Geometrie

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



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;
```


- Das aktuelle Material:

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

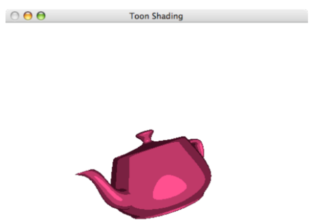
- Aktuelle Lichtquellen(-Parameter):

```
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];
```

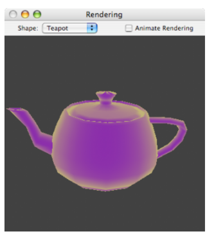
- Und viele weitere (z.B. zu Texturen, Clipping Planes,...)

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)

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:

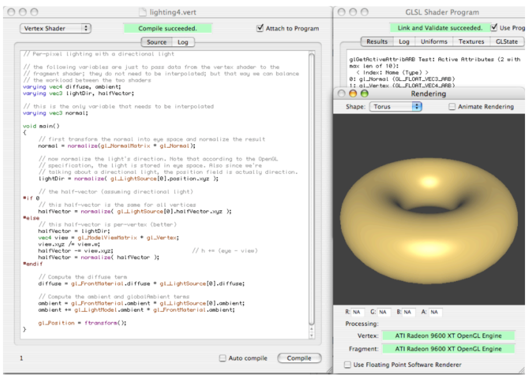
```
attribute vec3 myAttrib;
```

  - Im C-Programm:

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


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].*

Achtung bei Subtraktion homogener Punkte

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

$$\frac{\mathbf{v}.xyz}{\mathbf{v}.w} - \frac{\mathbf{e}.xyz}{\mathbf{e}.w} = \frac{\mathbf{v}.xyz \cdot \mathbf{e}.w - \mathbf{e}.xyz \cdot \mathbf{v}.w}{\mathbf{v}.w \cdot \mathbf{e}.w}$$

- Normalisierung:

```
normalize(V.xyz/V.w) = normalize(V.xyz)
```

- Zusammen :

```
normalize(v-e) = normalize(v.xyz*e.w - e.xyz*v.w)
```


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(...);
```

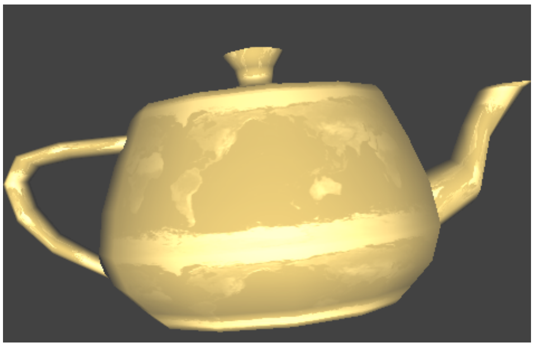
- Verbinde beide:

```
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 );
```

Beispiel: eine einfache "Gloss-Textur"

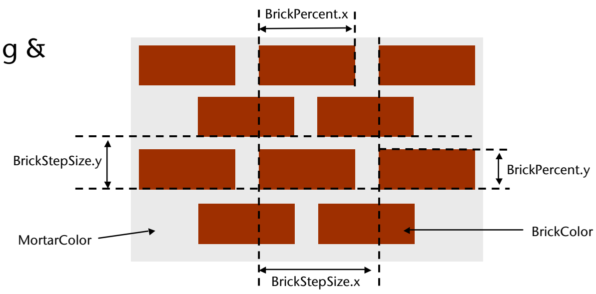


GLSL_editor/gloss.{frag,vert}

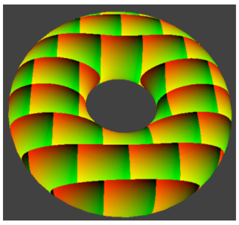
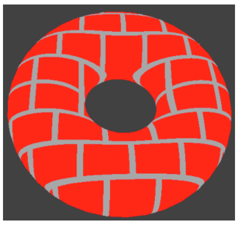
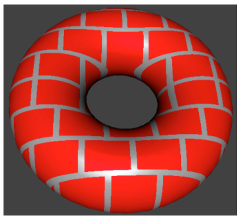
Eine einfache prozedurale Textur

- Ziel:
  Ziegelstein-Textur

- Vereinfachung & Parameter:



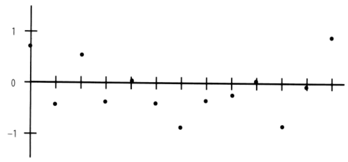
- Generelle Funktionweise:
  - Vertex-Shader: normale Beleuchtungsrechung
  - Fragment-Shader:
    - bestimme pro Fragment anhand der xy-Koordinaten des zugehörigen Punktes im Objektraum, ob der Punkt im Ziegel oder im Mörtel liegt
    - danach, entsprechende Farbe mit Beleuchtung multiplizieren
- Beispiele:

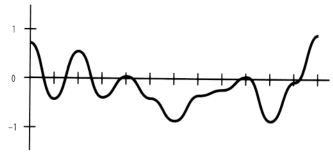
Rauschen

- 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)

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

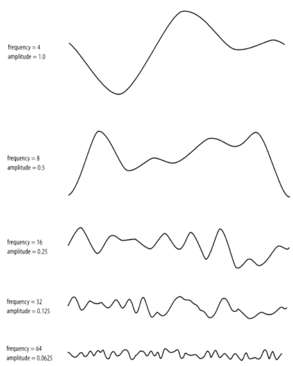


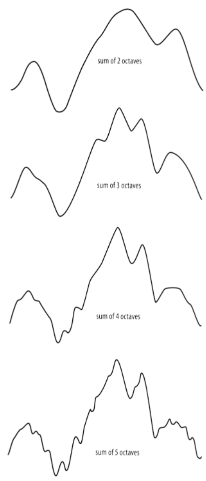
  2. Interpoliere dazwischen, z.B. kubisch (linear reicht nicht):



- Diese Art Rauschfunktion heißt *"value noise"*

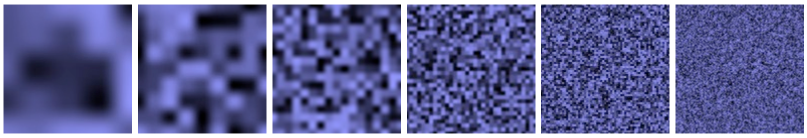
3. Generiere mehrere Rausch-funktionen mit verschiedenen Frequenzen

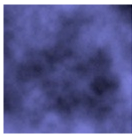




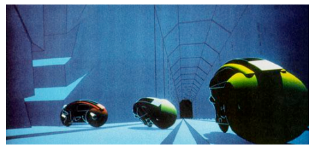
4. Addiere alle diese zusammen
   - Ergibt Rauschen auf verschiedenen "Skalen"

- Dasselbe in 2D:



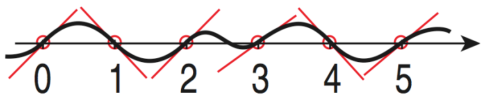


Ergebnis

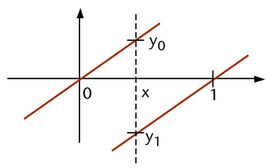


- Läßt sich leicht verallgemeinern in höhere Dimensionen
- Heißt auch *Perlin noise, pink noise,* oder *fractal noise*
  - Ken Perlin; hat sich zuerst damit beschäftigt bei der Arbeit an TRON

- *Gradient noise*:
  - Spezifiert die Gradienten an den Integer-Stellen (statt den Werten):



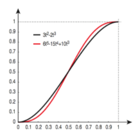
  - Interpolation:
    - Berechne y_0 und y_1 als Wert der Geraden durch 0 und 1 mit den vorgegebenen (zufälligen) Gradienten
    - Interpoliere y_0 und y_1 mit einer Blending-Funktion, z.B.



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

oder

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



- 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:


kubische Interpolation

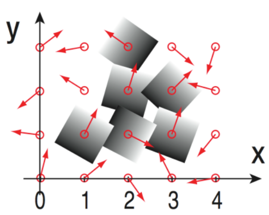
Ken Perlin

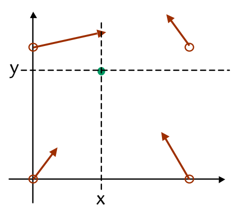

quintische Interpolation

- *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]x[0,1]
    - Seien die Gradienten
      g_{00} = Gradient an (0,0), g_{01} = Gradient an (0,1),
      g_{10} = Gradient an (1,0), g_{11} = Gradient an (1,1)
    - Berechne den Wert der "Gradienten-Rampen" am Punkt P:

$$z_{00} = g_{00} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \qquad z_{01} = g_{01} \cdot \begin{pmatrix} x \\ y-1 \end{pmatrix}$$

$$z_{00} = g_{00} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \qquad z_{01} = g_{01} \cdot \begin{pmatrix} x \\ y-1 \end{pmatrix}$$





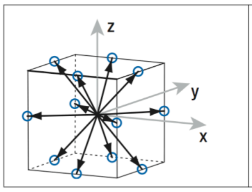
- Blending der 4 "z"-Werte durch bilineare Interpolation:

$$z_{x0} = (1 - q(x))z00 + q(x)z_{10}\,,\;\; z_{x1} = (1 - q(x))z01 + q(x)z_{11}$$

$$z_{xy} = (1 - q(y))zx0 + q(y)z_{x1}$$

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

- 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:



$$g_0 = (0, 1, 1),\ g_1 = (0, 1, -1),$$
$$g_2 = (0, -1, 1),\ g_3 = (0, -1, -1),$$
$$g_4 = (1, 0, 1),\ g_5 = (1, 0, -1),$$
$$g_6 = (-1, 0, 1),\ g_7 = (-1, 0, -1),$$
$$g_8 = (1, 1, 0),\ g_9 = (1, -1, 0),$$
$$g_{10} = (-1, 1, 0),\ g_{11} = (-1, -1, 0)$$

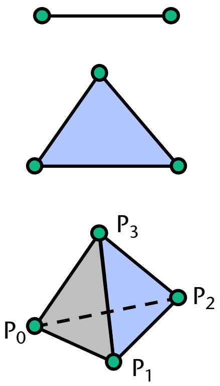
- Integer-Koordinaten der Gitterpunkte werden einfach gehasht → Index in eine Tabelle vordefinierter Gradienten

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

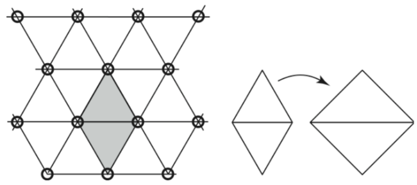
$$X = P_0 + \sum_{i=1}^{d} s_i \mathbf{u}_i$$

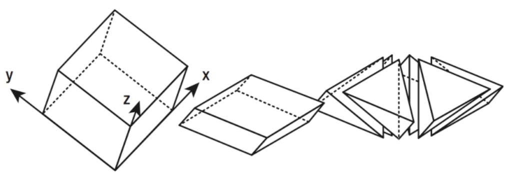
mit

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



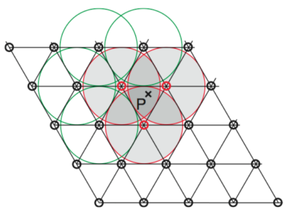
- Mit gleichseitigen(!) d-dimensionalen Simplices kann man den d-dim. Raum partitionieren (tesselieren):



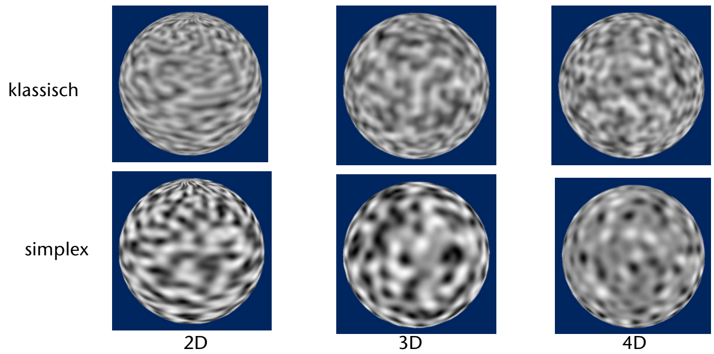


- Generell gilt:
  - Ein *d*-dimensionaler Simplex hat *d*+1 Ecken
  - Mit gleichseitigen *d*-dimensionaler Simplices kann man einen Würfel partitionieren, der entlang seiner Diagonalen geeignet "gestaucht" wurde
  - Solch ein *d*-dim. gestauchter Würfel enthält *d*! viele Simplices

- 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 Simplizes 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:



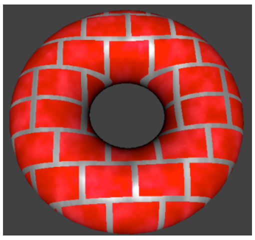
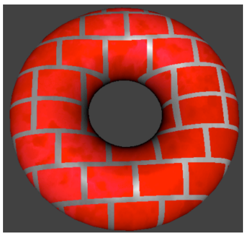
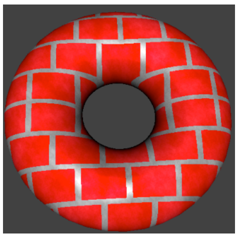
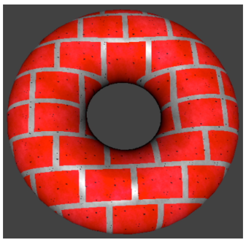
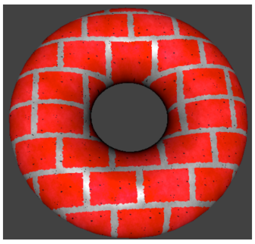
- Im GLSL-Standard werden 4 noise-Funktionen definiert: **`float noise1(gentype)`**, **`vec2 noise2(gentype)`**, **`vec3 noise3(gentype)`**, **`vec4 noise4(gentype)`**.
- Aufruf einer Noise-Funktion:

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

  - 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