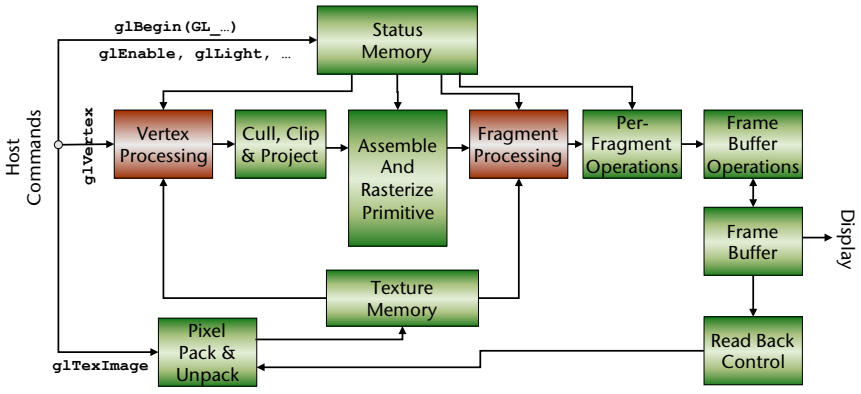
# Computer-Graphik II
## Advanced Shader Programming

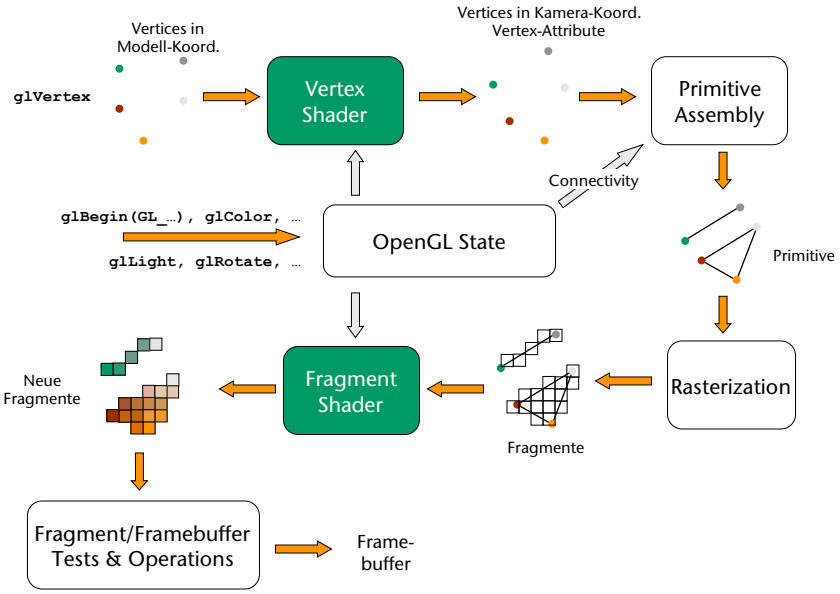
G. Zachmann
Clausthal University, Germany
cg.in.tu-clausthal.de

## Erinnerung

- Programmierbare *vertex und fragment processors*
  - Legen offen, was sowieso schon immer da war
- Texturspeicher = allgemeiner Speicher für beliebige Daten



## Abstraktere Übersicht der programmierbaren Pipeline






## 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(...);
```

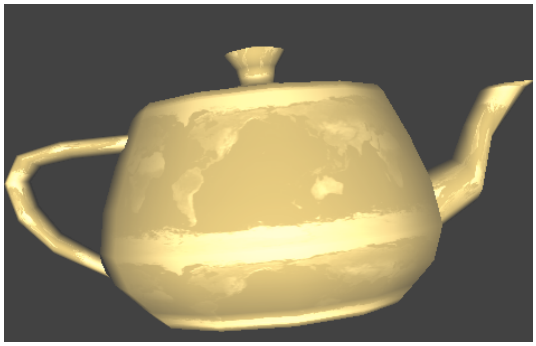
- Verbinde beide:

```
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"



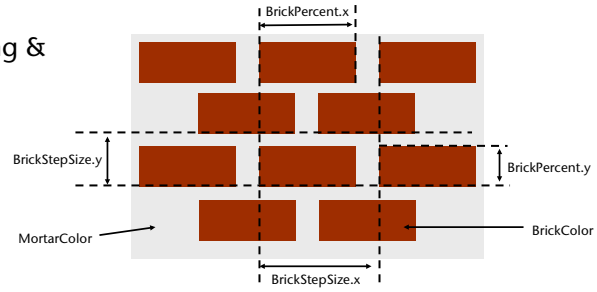
`vorlesung_demos/gloss.{frag,vert}`



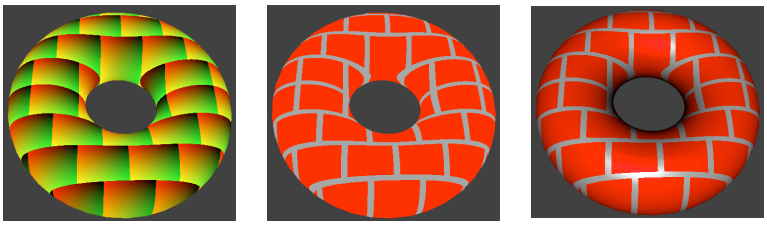
## Eine einfache prozedurale Textur

- 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:

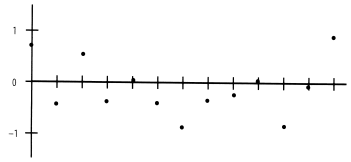




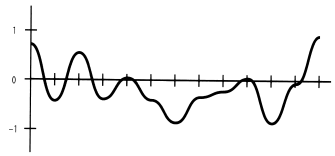
# Rauschen

- 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:



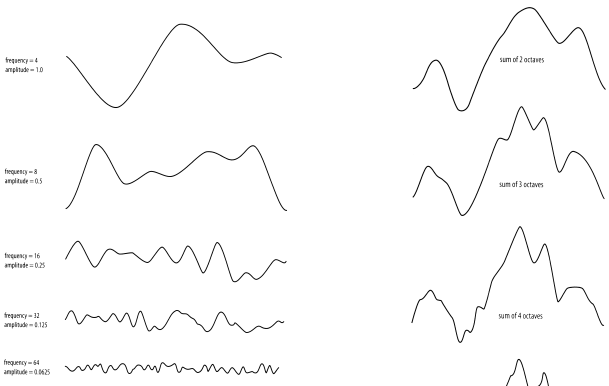

  2. Interpoliere dazwischen, z.B. kubisch (linear reicht nicht):




- Diese Art Rauschfunktion heißt *"value noise"*

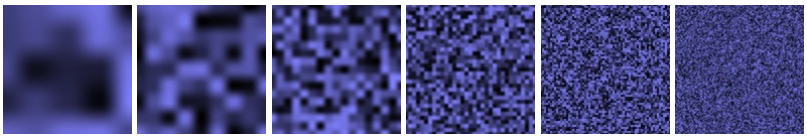


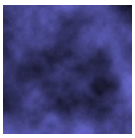
3. Generiere mehrere Rauschfunktionen mit verschiedenen Frequenzen



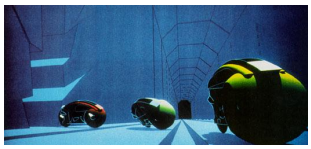

4. Addiere alle diese zusammen
   - Ergibt Rauschen auf verschiedenen "Skalen"

- Dasselbe in 2D:





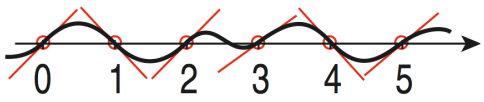
Ergebnis



- 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



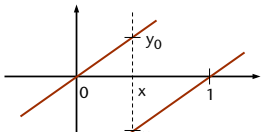

- *Gradient noise*:
  - Spezifiert die Gradienten an den Integer-Stellen (statt den Werten):

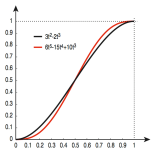



  - Interpolation:
    - Berechne $y_0$ und $y_1$ als Wert der Geraden durch 0 und 1 mit den vorgegebenen (zufälligen) Gradienten
    - Interpoliere $y_0$ und $y_1$ mit einer Blending-Funktion, z.B.

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

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



- 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:

Ken Perlin

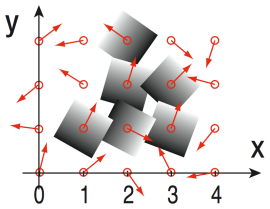
kubische Interpolation | quintische Interpolation

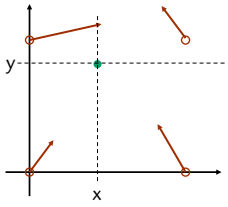


- *Gradient noise* im 2D:
  - Gebe an Integer-Gitterpunkten Gradienten vor (2D Vektoren, nicht notw.weise mit Länge 1)
  - Interpolation (wie im 1D):
    - O.B.d.A. P = $(x,y)$ in [0,1]x[0,1]
    - Seien die Gradienten
      $g_{00}$ = Gradient an (0,0), $g_{01}$ = Gradient an (0,1),
      $g_{10}$ =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

$$z_{00} = g_{00} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \qquad z_{10} = g_{10} \cdot \begin{pmatrix} x-1 \\ y \end{pmatrix}$$

$$z_{01} = g_{01} \cdot \begin{pmatrix} x \\ y-1 \end{pmatrix} \qquad z_{11} = g_{11} \cdot \begin{pmatrix} x-1 \\ y-1 \end{pmatrix}$$



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

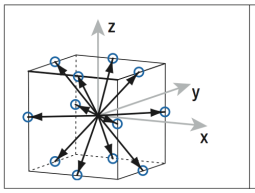
$$z_{x0} = (1 - q(x))z_{00} + q(x)z_{10}\ , \qquad z_{x1} = (1 - q(x))z_{01} + q(x)z_{11}$$

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

- 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:




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

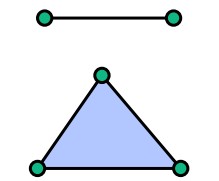
- Integer-Koordinaten der Gitterpunkte werden einfach gehasht → Index in eine Tabelle vordefinierter Gradienten

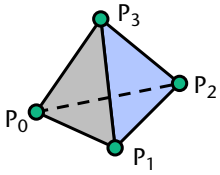
- d-dimensionaler Simplex :=
  - Verbindung von $d + 1$ affin unabhängigen Punkten
- Beispiele:
  - 1D: Linie , 2D: Dreieck , 3D: Tetraeder
- Allgemein:
  - Punkte $P_0, \ldots, P_d$
  - Simplex = alle Punkte $X$ mit

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

mit

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

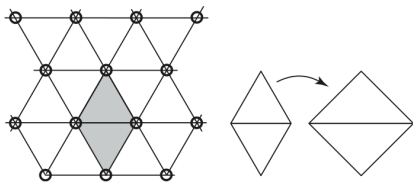


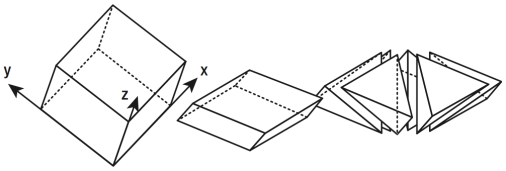







- Mit gleichseitigen(!) $d$-dimensionalen Simplices kann man den $d$-dim. Raum partitionieren (tesselieren):

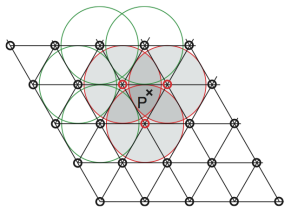




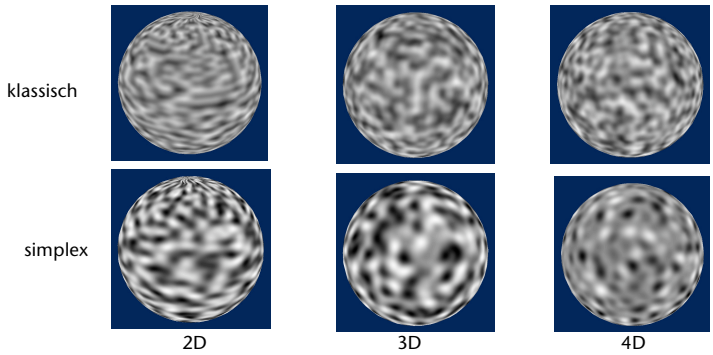
- Generell gilt:
  - Ein *d*-dimensionaler Simplex hat *d*+1 Ecken
  - Mit gleichseitigen *d*-dimensionaler Simplices kann man einen Würfel partitionieren, der entlang seiner Diagonalen geeignet "gestaucht" wurde
  - Solch ein *d*-dim. gestauchter Würfel enthält *d*! viele Simplices



- 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 auf 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:

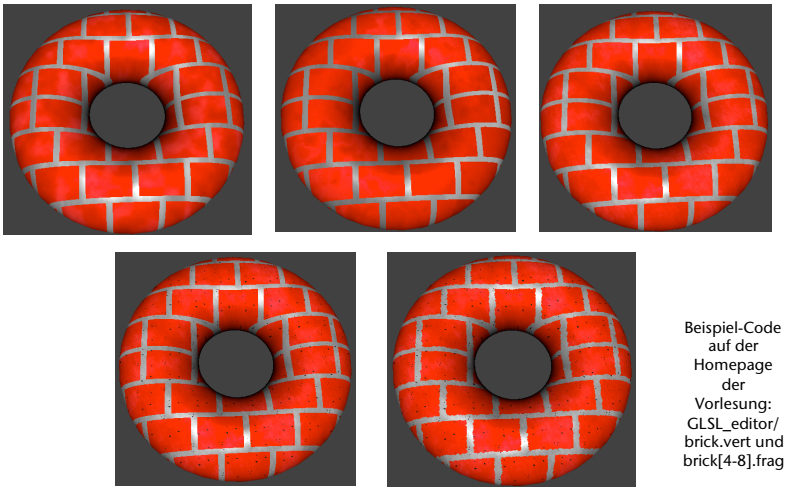





- Im GLSL-Standard werden 4 noise-Funktionen definiert:
  `float noise1(gentype), vec2 noise2(gentype), vec3 noise3(gentype), vec4 noise4(gentype).`
- Aufruf einer Noise-Funktion:
  $v$ = noise2( $f*x + t$, $f*y + t$ )
  - Mit $f$ kann man die räumliche Frequenz steuern, mit $t$ kann man eine Animation 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:

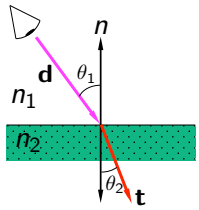


Beispiel-Code auf der Homepage der Vorlesung: GLSL_editor/ brick.vert und brick[4-8].frag

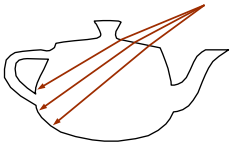


## Lichtbrechung

- Mit Shadern kann man Approximationen von einfachen "globalen" Effekten versuchen
- Beispiel: Lichtbrechung
- Was benötigt man, um den gebrochenen Strahl (*refracted ray*) zu berechnen?
  - Snell's Gesetz: $n_1 \sin\theta_1 = n_2 \sin\theta_2$
  - Benötigt werden: **n**, **d**, $n_1$, $n_2$
  - Ist alles im Fragment-Shader vorhanden
  - Man kann also **t** pro Pixel berechnen

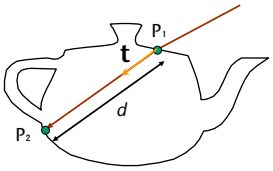



- Warum also ist Brechung so schwer?
  - Um den korrekten Schnittpunkt des gebrochenen Strahls zu berechnen, benötigt man die gesamte Geometrie!

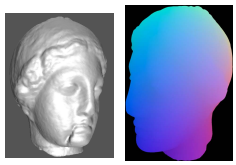


- Ziel: transparente Objekte mit 2 Schnittflächen approximieren
- Schritt 1: bestimme den nächsten Schnittpunkt

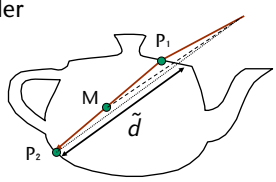
$$P_2 = P_1 + d\mathbf{t}$$




  - Idee: approximiere $d$
  - Rendere dazu 1x in einem Pass vorab eine Depth-Map der backfacing Polygone vom Viewpoint aus

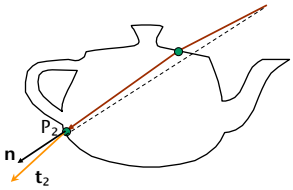


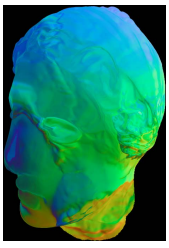
  - Suche mit Binärsuche (ca. 5 Iter.) darin nach der "richtigen" Tiefe:
    - Bestimme Midpoint
    - Projiziere Midpoint bzgl. Viewpoint nach 2D
    - Indiziere damit die Depth-Map






- Schritt 2: bestimme die Normale in $P_2$
  - Rendere dazu vorab eine Normal-Map aller backfacing Polygone vom Viewpoint aus
  - Projiziere $P_2$ bzgl Viewpoint nach 2D
  - Indiziere damit die Normal-Map
- Schritt 3:
  - Bestimme $\mathbf{t}_2$
  - Indiziere damit eine Environment-Map



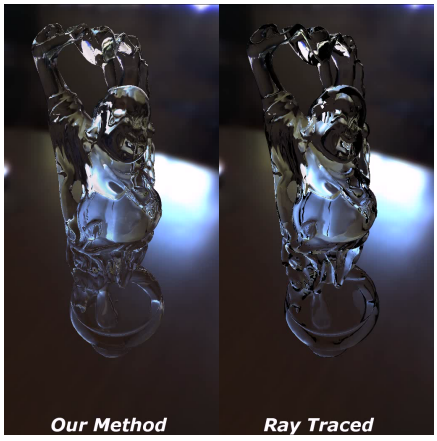



Normal-Map

- Viele offene Herausforderungen:
  - Bei *depth complexity* > 2:
    - Welche Normale / welcher Tiefenwert soll behalten werden?
  - Approximation der Distanz
  - Aliasing



## Beispiele





Mit innerer Reflexion