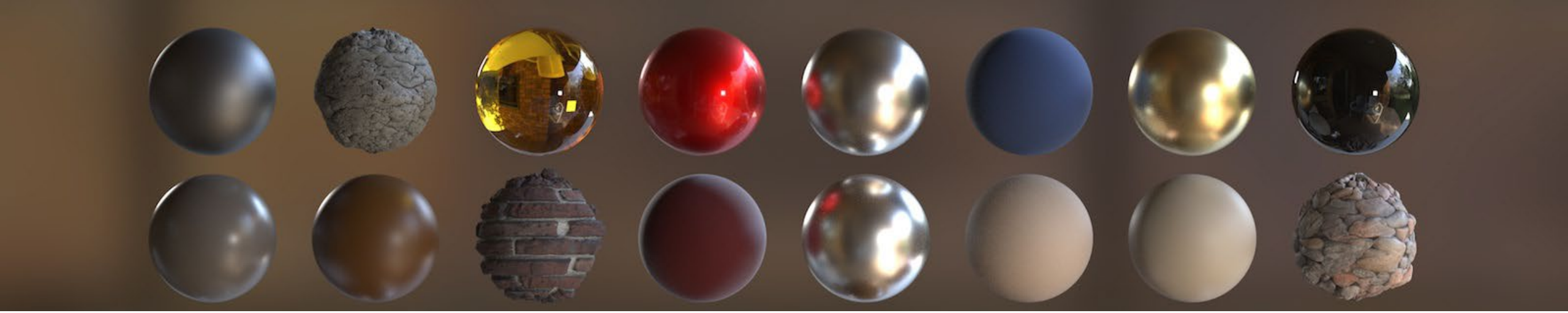
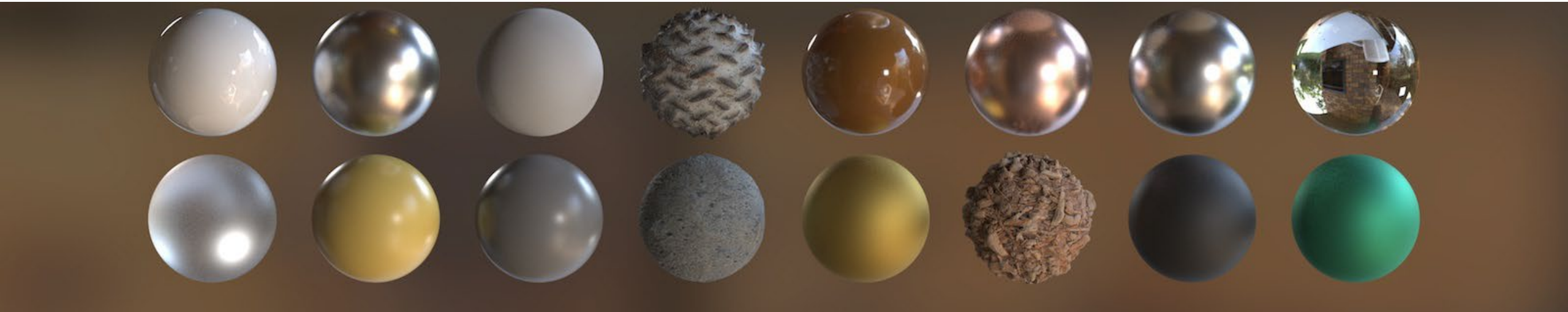
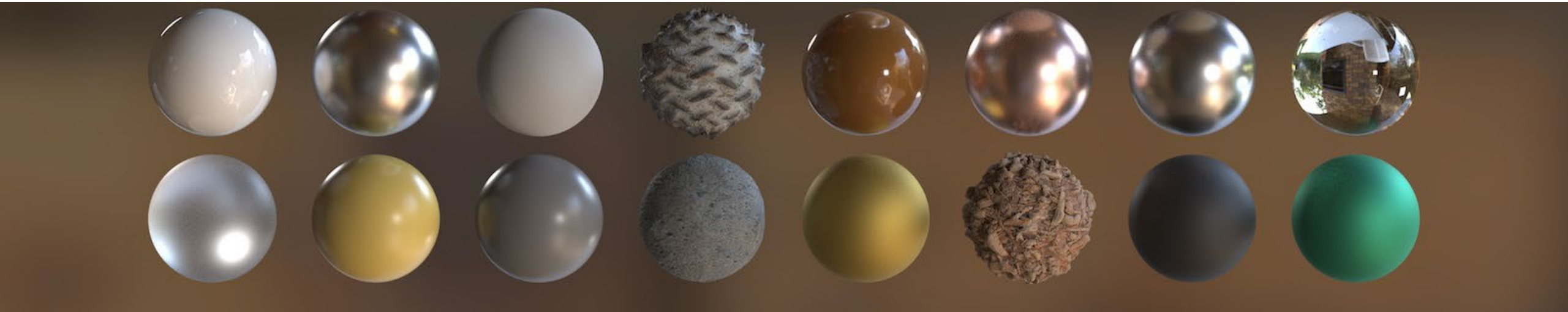


# Shader Praktikum

FH Wedel



# Shader – Aufgabe 5

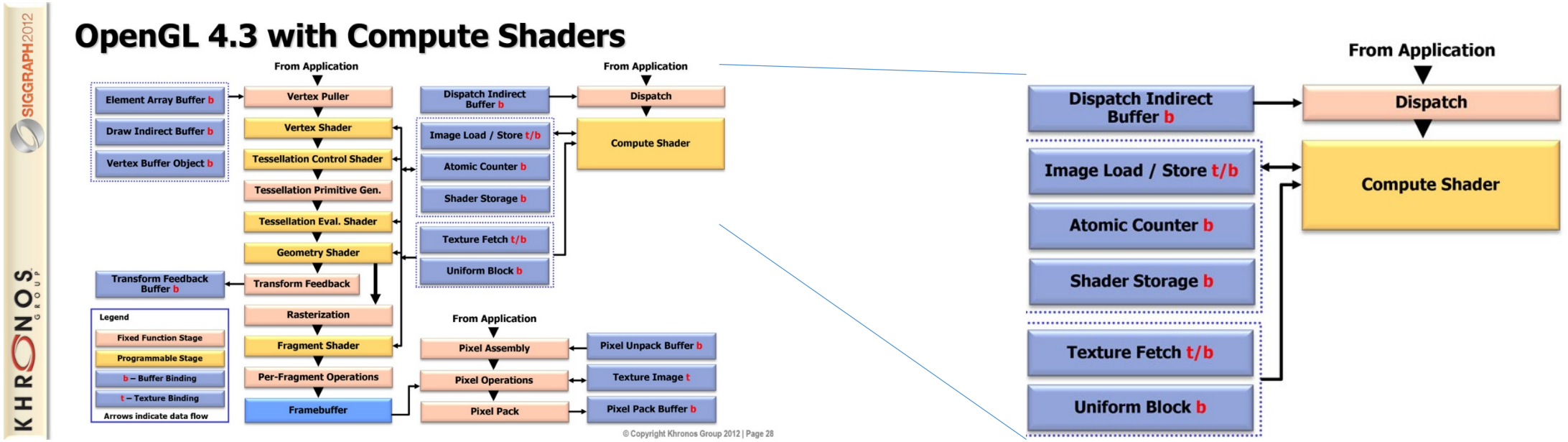


# Themen

- Compute Shader
- Simulation
  - Wassersimualtion
- Licht
  - Reflexion
  - Brechung
- Sonstiges

# Compute Shader

- Separate Stufe – kein Bestandteil der Grafikpipeline
- Kein vordefinierter Datenfluss (nur einige *built-in*-Variablen)
- Zusätzliche Eingabe z.B. über Textur- und Buffer-Objekte
- Ergebnis des Compute Shaders wird durch Seiteneffekte erzeugt



# Compute Shader

- Ab Version 4.3 (2012) unterstützt
- Arbeitet ähnlich wie ein **GPGPU-Kernel** (OpenCL/CUDA)
- Besitzt eine große Anzahl an identischen Threads auf Daten im Grafikspeicher
- Keine User-Definierten Inputs/Outputs
  - Input und Ouputs durch Schreiben & Lesen von
    - Texturen, Image Load/Store, Shader Storage Blocks
    - Zusätzlich: Uniform-Variablen

# Compute Shader

Zuordnung der Aufrufe in verschiedenen Shader-Stufen

| Stufe | Datenelement |
|---|---|
| Vertex Shader | pro Vertex |
| Tessellation Control Shader | pro Vertex (im Patch) |
| Tessellation Evaluation Shader | pro Vertex (im Patch) |
| Geometry Shader | pro Primitiv |
| Fragment Shader | pro Fragment |
| Compute Shader | pro „Work Item“ (abstrakt) |

# Compute Shader

## Logische Struktur

**Globale Workgroup**

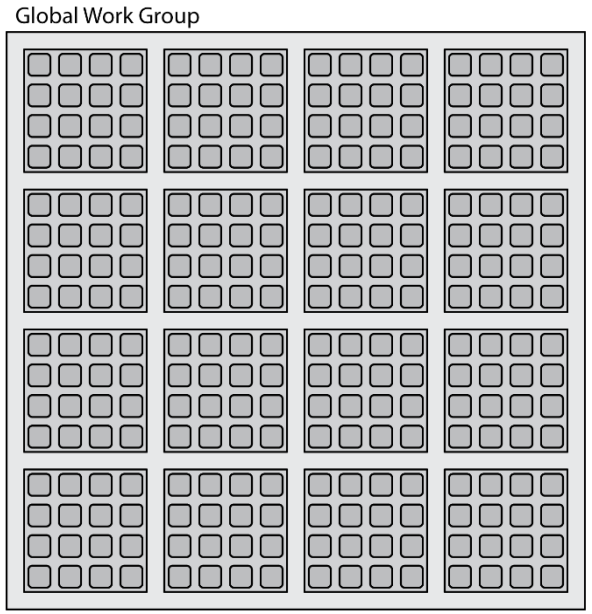
Die globale Arbeitsgruppe bildet einen 3-dimensionalen Raum, indem lokale Arbeitsgruppen gestartet werden.

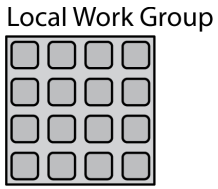
**Local Workgroup**

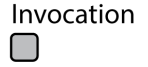
Die lokalen Arbeitsgruppen bilden wiederum einen 3-dimensionalen Raum, indem die einzelnen Aufrufe gestartet werden.

**Invocation**

Ein Aufruf (auf einem „Work Item") des Compute Shaders.



# Compute Shader

Position im Compute Space (in der globalen Arbeitsgruppe) wird mit *built-in*-Variable *gl_InvocationID* bestimmt.

1D Compute Space (Beispielprogramm)

```
// Position eines Partikels
vec2 position = positions[gl_GlobalInvocationID.x];
```

3D Compute Space (Beispielprogramm)

```
// Position eines Partikels
float density = imageLoad(VolumeTexture, ivec3(gl_GlobalInvocationID.xyz)).r;
```

Die weiteren *built-in*-Variablen sind für die Bearbeitung der Aufgabe irrelevant.

# Compute Shader

Shader Storage Buffer

- Textur-Objekte

**layout** (rgba8, binding = 0) **uniform** image2D MyImage;

  - Für bestimmte Daten gut geeignet
    - Ergebnis eines Rendering-Passes
    - Daten auf denen ein Sampler arbeitet
    - Uvm.

- Buffer-Objekte

**layout** (std430, binding = 0) buffer MyBuffer { … };

  - Buffer-Objekte können
    - Vertex-Daten
    - Matrizen für Instancing-Draw-Commands
    - Uvm.

# Compute Shader

## Shader Storage Buffer

Shader

```
#version 430 core

layout (std430, binding = 0) buffer BufferObject {
        int mode;
}
```

Host

```
// Buffer anlegen
Gluint buffer;
glGenBuffers(1, & buffer);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, buffer);
glBufferData(GL_SHADER_STORAGE_BUFFER, 4096, NULL, GL_DYNAMI_COPY);

// Buffer an das Shader Storage Buffer Target 0 binden
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, buffer);
```

# Compute Shader

**Speicherlayout**

- Uniform Buffer
  - Können verwendet werden, um **mehreren Shaderprogrammen** (nicht nur Compute Shader) den Zugriff auf die **gleichen Werte** für **Uniform-Variablen** (Uniform-Blöcke) zu ermöglichen
- Shader Storage Buffer
  - Die OpenGL-Spezifikation definiert auf PDF-Seite 145 (intern. Nummerierung 124) das **Speicherlayout std140** für den Uniform Buffer. Shader Storage Buffer mit dem Speicherlayout **std430** basieren auf dem Speicherlayout std140 der Uniform Buffer.

Ihr findet die OpenGL-Spezifikation für die Version 4.3 (Kernprofil) unter Material in der Aufgabenstellung oder unter folgendem Link:

https://www.opengl.org/registry/doc/glspec43.core.20120806.pdf

# Compute Shader

## Speicherlayout

Kein Padding bei einem Array mit **vec2** als Elementtyp

```
#version 430 core

layout (std430, binding = 0) buffer BufferObject {
        vec2 points[];
}
```

Anordnung der Elemente im Speicher

| v1.x | v1.y | v2.x | v2.y | v3.x | v3.y | v4.x | v4.y |
|---|---|---|---|---|---|---|---|

# Compute Shader

## Speicherlayout

Kein Padding bei einem Array mit **vec2** als Elementtyp

```
#version 430 core

layout (std430, binding = 0) buffer BufferObject {
        vec2 points[];
}
```

Anordnung der Elemente im Speicher


Ein float als Padding bei einem Array mit **vec3** als Elementtyp

```
#version 430 core

layout (std430, binding = 0) buffer BufferObject {
        vec3 points[];
}
```

# Compute Shader

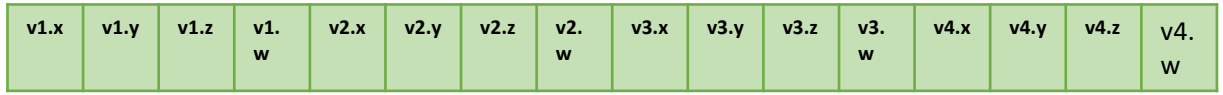
## Speicherlayout

Kein Padding bei einem Array mit vec4 als Elementtyp

```
#version 430 core

layout (std430, binding = 0) buffer BufferObject {
        vec4 points[];
}
```

Anordnung der Elemente im Speicher




Kein Padding bei einem Array vom Typ S als Elementtyp

```
#version 430 core

struct S {
        float x, y, z;
}


layout (std430, binding = 0) buffer BufferObject {
        S points[];
}
```

# Compute Shader

**Synchronisation**

Definition

**void** glMemoryBarrier(**GLbitfield** barriers);

Alle im Bitfeld spezifizierten Speicherzugriffe müssen ab diesem Kommando abgearbeitet sein.

**Beispiel**

```
// Compute Shader arbeitet auf dem Buffer „buffer“
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, buffer);
glDispatchCompute(3012, 1, 1);    // OpenGL wird soviel Aufrufe durchführen (wenn möglich parallel [keine fixe
                                  // Reihenfolge]) wie nötig

// Warten, bis die Schreiboperationen des Compute Shader abgeschlossen sind
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);

// Nutzt den Buffer „buffer“ als GL_ARRAY_BUFFER
glBindVertexArray(vertexArrayObject);
glDrawArray(GL_TRIANGLES, 0, 3012);
```

https://www.khronos.org/opengl/wiki/GLAPI/glMemoryBarrier

# Compute Shader

## GLSL Built-in Variablen

| | | |
|---|---|---|
| uvec3 | gl_NumWorkGroups | globale Workgroup Größe, die mittels glDispatchCompute() gesetzt wurde |
| uvec3 | gl_WorkGroupSize | lokale Workgroup Größe, die mittels layout gesetzt wurde |
| uvec3 | gl_WorkGroupID | Position des aktuellen Aufrufes in der globalen Workgroup |
| uvec3 | gl_LocalInvocationID | Position des aktuellen Aufrufes in der lokalen Workgroup |
| uvec3 | gl_GlobalInvocationID | eindeutiger Index des aktuellen Aufrufes in der globalen Workgroup |
| uint | gl_LocalInvocationIndex | gl_LocalInvocationID |

# Wassersimulation

# Simulation

## Newton‘s 2. Gesetz

$$\boldsymbol{f = ma}$$

$$a = f/m$$

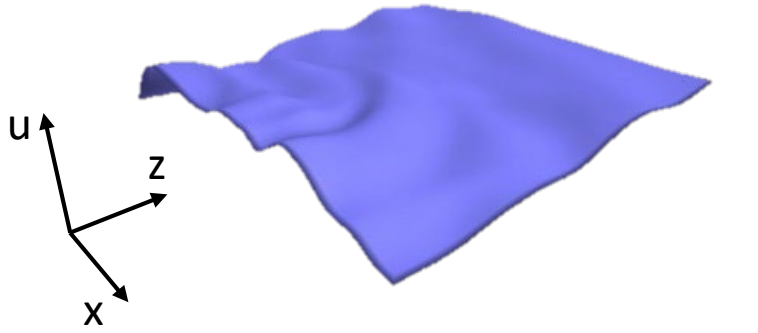
Geschwindigkeitsänderung pro Zeiteinheit = Kraft geteilt durch Masse

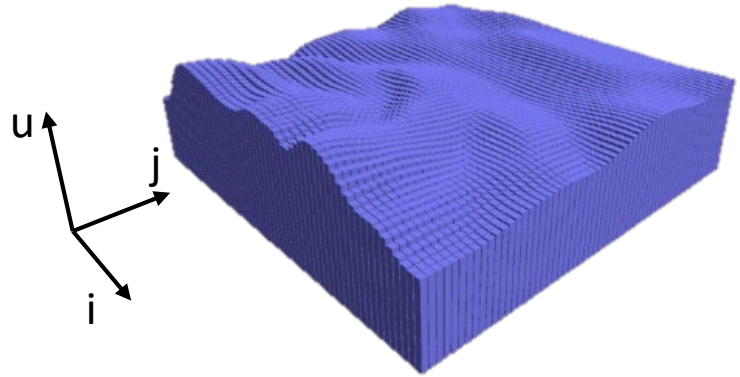
**Simulationsschleife**

- Berechnung der **Kraft** (Abhängig von aktuellen Positionen & Geschwindigkeiten)
- **Geschwindigkeit** += **Kraft/Masse** * **Δt** (Δt - Zeitschritt)
- Position += Geschwindigkeit * Δt

# Wassersimulation nach Müller-Fischer 2008

- Wasser-Volumen wir auf die **Oberfläche reduziert** & **diskretisiert** (Wasser als 2D-Funktion/Array)
- Einzelne **Wassersäulen** werden betrachtet und separat simuliert

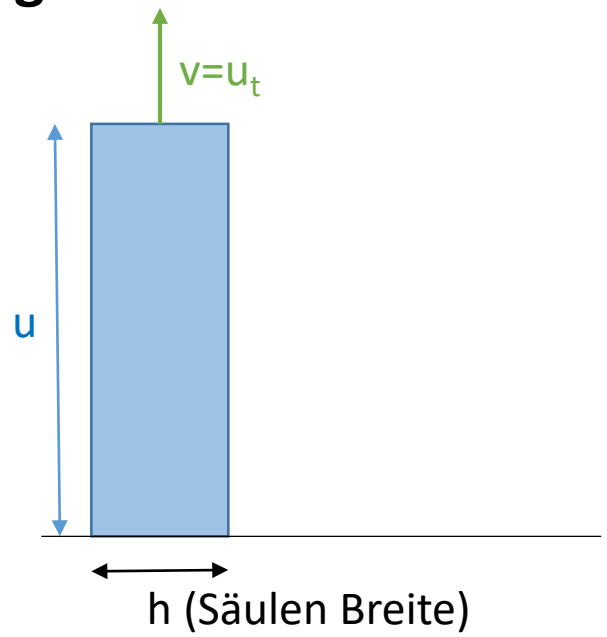



Quelle: http://matthias-mueller-fischer.ch/talks/GDC2008.pdf

# Wassersimulation nach Müller-Fischer 2008

- Pro **Wassersäule** wird die **Höhe** *u* und die vertikale **Geschwindigkeit** *v* gespeichert
- Berechnung der **Beschleunigung** der Wassersäule durch **umliegende Wassersäulen** und anschließende **Eulerintegration**




| Variable | Bedeutung |
|---|---|
| t | aktueller Zeitschritt |
| i,j | Position der Wassersäule in XZ-Ebene |
| c | Wellengeschwindigkeit |
| f | Beschleunigung der Wassersäule |
| h | Seitenlänge einer Wassersäule |
| u | Höhe der Wassersäule |
| v | Geschwindigkeit der Wassersäule |

# Wassersimulation nach Müller-Fischer 2008

Newton‘s Gesetz

$$u_{tt} = \frac{f}{m} = k * u_{tt}/\text{m}$$

Annahme: Masse *m* ist konstant, k/m kann durch c² ersetzt werden

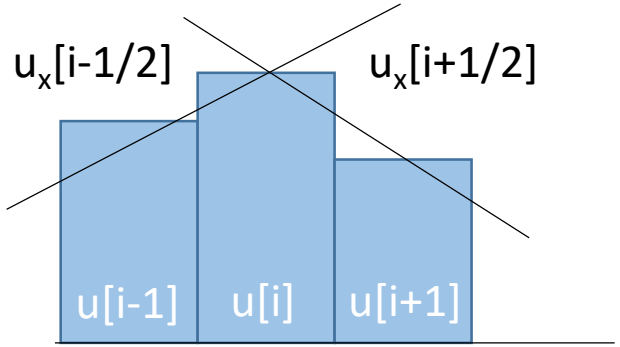
1d Wellengleichung

$$u_{tt} = c^2 * u_{xx}$$

Lösung:

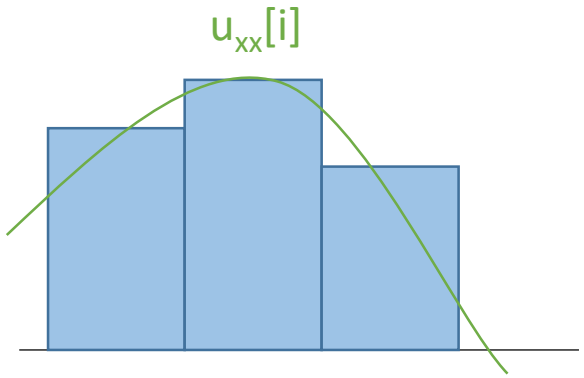
$$u(x,t) = f(x+ct) + g(x-ct)$$ Für alle f,g

# Wassersimulation nach Müller-Fischer 2008




$$u_x\left[i-\frac{1}{2}\right] = (u[i]-u[i-1])/h$$

$$u_x\left[i+\frac{1}{2}\right] = (u[i+1]-u[i])/h$$




$$u_{xx}[i] = (u_x\left[i+\frac{1}{2}\right] - u_x[i-1/2])/h$$

$$= (u_x[i+1]-2u[i]+u[i-1])/h^2$$

- 1d Krümmung

$$(u[i+1]+u[i-1]-2u[i])/h^2$$

- 2d Krümmung

$$(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1]-4u[i,j])/h^2$$

# Wassersimulation nach Müller-Fischer 2008

2D Simulation

## Beschleunigung durch umliegende Wassersäulen

$$f^t = \frac{c^2(u^t[i+1,j] + u^t[i-1,j] + u^t[i,j+1] + u^t[i,j-1] - 4u^t[i,j])}{h^2}$$

## Neue Geschwindigkeit der Wassersäule

$$v^{t+1}[i,j] = v^t[i,j] + \Delta t * f^t$$

## Neue Höhe der Wassersäule

$$u^{t+1}[i,j] = u^t[i,j] + \Delta t * v^t[i,j]$$

Zusätzlich muss eine Dämpfung (Multiplikation mit 0 < x < 1) der Geschwindigkeit stattfinden, um ein Aufschaukeln der Wasseroberfläche zu verhindern.

# Wassersimulation nach Müller-Fischer 2008

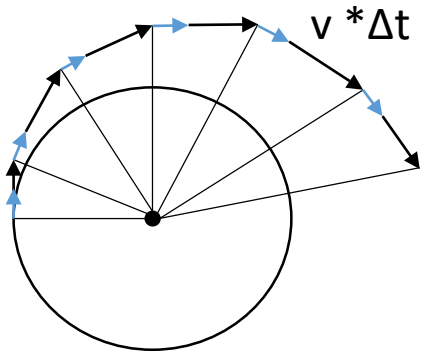
- Oberes Limit für die Wellengeschwindigkeit *c*
  - $c < h / \Delta t$
- *Oberes Limit für einen Zeitschritt Δt*
  - $\Delta t < h / c$ (Courant-Friedrichs-Lewy(CFL) Bedingung)

- Begrenzungsbedingung
  - Clampen ergibt die Wellenreflexion
  - Wrapen ergibt die periodische Ausbreitung

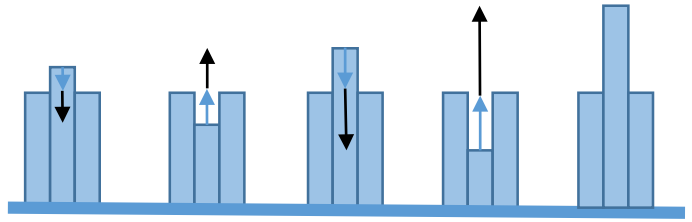
# Wassersimulation nach Müller-Fischer 2008

Numerische Explosion

- Explizierter Euler Schritt
  - $x \mathrel{+}= v * \Delta t$
- Annahme
  - Geschwindigkeit ist konstant während eines Zeitschrittes




Beispiel an einem Particle



Beispiel an Säulen
(overshooting)

# Wassersimulation nach Müller-Fischer 2008

- **Dämpfen** der Kraft (physikalisch „korrekt“)
  - $f = -k * v$
  - Problem
    - Wahl des Faktors $k$
    - Keine Garantie für Stabilität
- **Skalierung** (nicht physikalisch, stärkere direkte Kontrolle)
  - $v = s * v$
  - $s < 1$, je geringer der Zeitschritt, desto stärker der Effekt
- **Abschneiden** clamping (nicht physikalisch, direkte Kontrolle)
  - *offset = (u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1])/4 – u[i,j]*
  - *maxOffset = maxSlope * h // independence of resolution h:*

  - *if (offset > maxOffset) u[i,j] += offset – maxOffset*
  - *if (offset < -maxOffset) u[i,j] += offset + maxOffset*

# Wassersimulation nach Müller-Fischer 2008

- Eine **Wassersäule** entspricht einem **Vertex**.
- Die **Höhe** der Wassersäule ist die **Y-Komponente des Vertex**.
- Die **Geschwindigkeit** kann in der **W-Komponente** gespeichert werden. (Achtung: Beim Rendern darf die **W-Division nicht beeinflusst** werden.)
- Damit die **Wasseroberfläche** realistisch wirkt, müssen die **Wellengeschwindigkeit** *c* und **Dämpfung** gut aufeinander abgestimmt sein.

# Wassersimulation nach Müller-Fischer 2008

- Nachteil
  - Keine brechenden Wellen
    - Da Reduktion auf 2D Funktion, ein Wert pro (x,z)

- Objekt Interaktionen
  - Siehe Original Foliensatz
  - http://matthias-mueller-fischer.ch/talks/GDC2008.pdf

# Element Buffer

- Zum Rendern mit Indizes kann ein Buffer-Objekt anstelle eines Arrays auf der CPU verwendet werden.

```
// Erstellen des Buffers
GLuint* indices = …

GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, numIndices * sizeof(GLuint), indices, GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

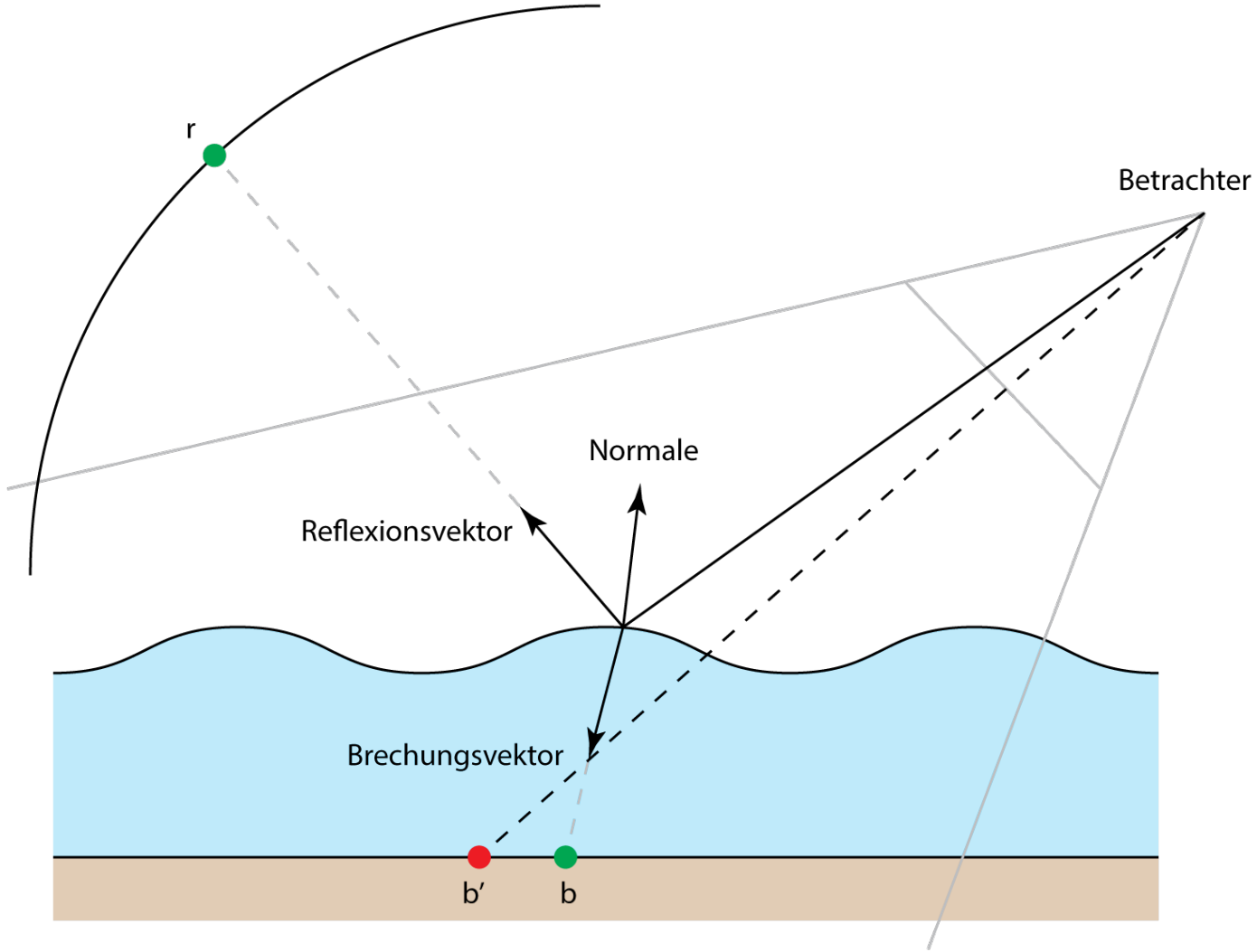
(Zum Erstellen des Buffers kann ein beliebiges Target verwendet werden.)

# Element Buffer

- Wenn ein Buffer-Objekt an das GL_ELEMENT_ARRAY_BUFFER-Target gebunden ist, werden die Indizes aus diesem Buffer gelesen.

```
// Rendern
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer);
glBindVertexArray(vertexArrayObject);
glDrawElements(GL_TRIANGLES, numIndices, GL_UNSIGNED_INT, NULL);
glBindBuffer(0, buffer);
```

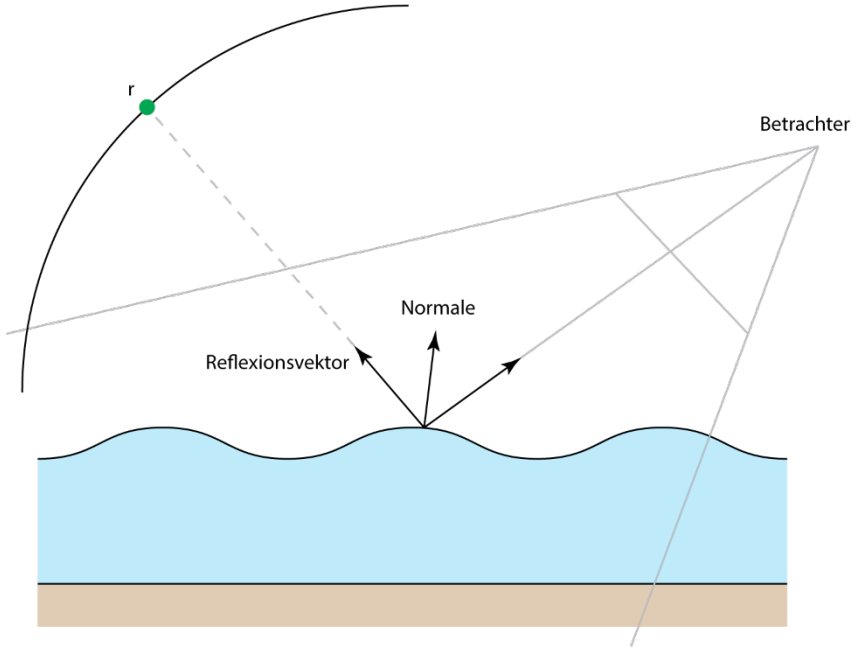
# Environment Mapping



# Reflexion & Brechung

**Reflexion** - Reflection

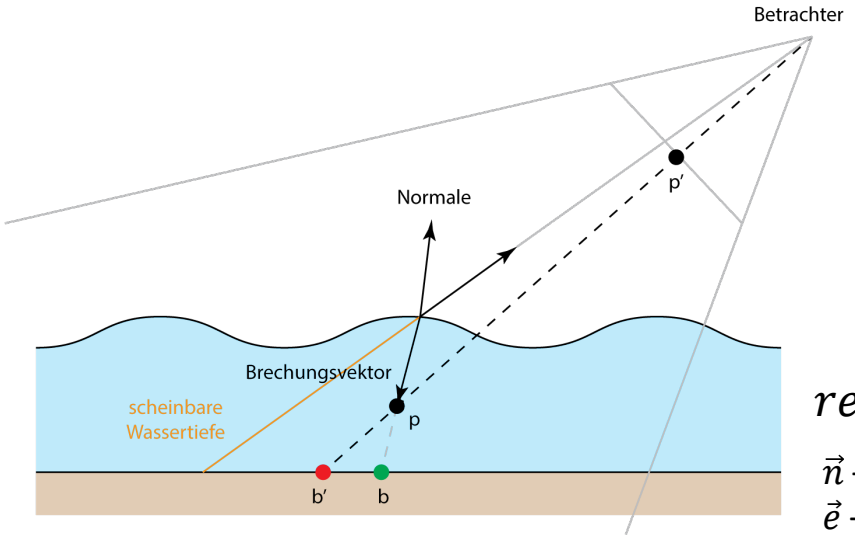
- Berechnung des **Reflexionsvektors**
  - Mit der GLSL-Funktion *reflect*(*viewVec*, normal) (oder eigener Implementierung)
- Zugriff auf die Plattkarte z.B. mit der Funktion *sampleEquirectangular*



# Reflexion & Brechung

(Licht-)**Brechung** - Refraction

- Berechnung des **Brechungsvektor** nach *Snellius*
  - Mit der GLSL-Funktion *refract(viewVec, normal, refractiveRatio)* (oder eigener Implementierung)
- Skalierung des Brechungsvektors mit der scheinbaren Wassertiefe (Skizze)
- Projektion des resultierenden Punktes → approximativer Schnittpunkt b‘ des Brechungsstrahls mit dem Untergrund (b wäre korrekt)




$$reflectionVec = \vec{e} - 2.0 * (\vec{n} \cdot \vec{e}) * \vec{n}$$

$\vec{n}$ – normalen Vektor an der Oberfläche
$\vec{e}$ – viewVector

# Reflexion & Brechung

$$\frac{\sin(\theta_2)}{\sin(\theta_1)} = \frac{v_2}{v_1} = \frac{n_1}{n_2}$$

## Snell's law

- Richtungsänderung der Ausbreitungsrichtung einer ebenen Welle beim **Übergang in ein anderes Medium**

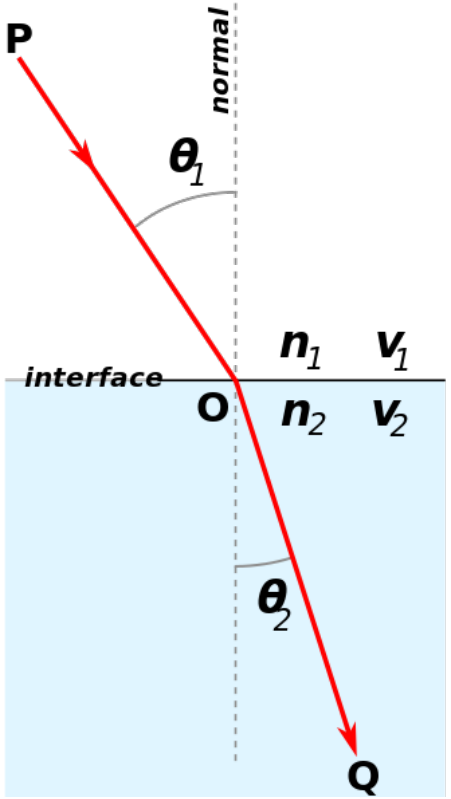
$$refractionVec = \left(\frac{n_1}{n_2}\right) * \vec{e} + \left(\frac{n_1}{n_2}\cos(\theta_1) - \cos(\theta_2)\right) * \vec{n}$$

$$refractionVec = \left(\frac{n_1}{n_2}\right) * \vec{e} + \left(\left(\frac{n_1}{n_2}\right) * (\vec{n} \cdot \vec{e}) - \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 (1 - (\vec{n} \cdot \vec{e})^2)}\right) * \vec{n}$$

ACHTUNG: **> 0.0**

$n_1$ – Brechungsindex Medium 1
$n_2$ – Brechungsindex Medium 2
$\theta_1$ - Einfallswinkel viewVektor
$\theta_2$ - Ausfallswinkel refractionVector

Medium 1



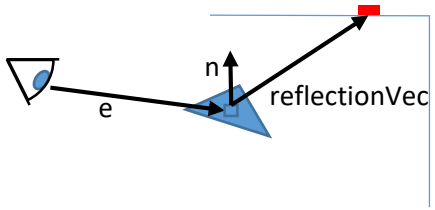

Medium 2

# Reflexion & Brechung

| Material | Brechungs-Index n |
|---|---|
| Luft | 1.00 |
| Wasser | 1.33 |
| Eis | 1.309 |
| Glass | 1.52 |
| Diamant | 2.42 |

# Reflexion & Brechung

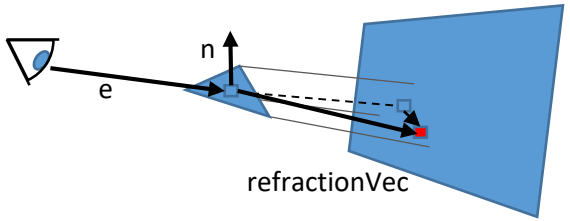
- Reflection




Environment Map
(z.B. cube map oder
Plattkarte)

Gaube Lake

- Refraction



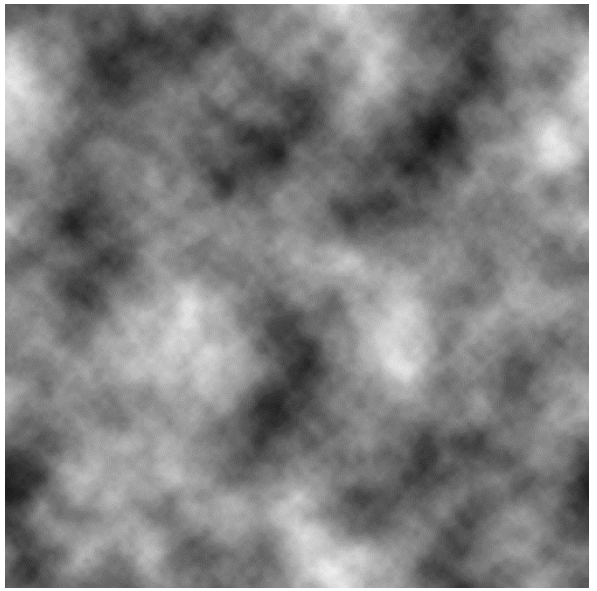

Szene unter der
Oberfläche. Vorher
auf eine Textur gerendert

- Kaustik
  - Schummeln, Nutzung einer Animierten Textur

https://www.touropia.com/lakes-in-france/

# Value Noise

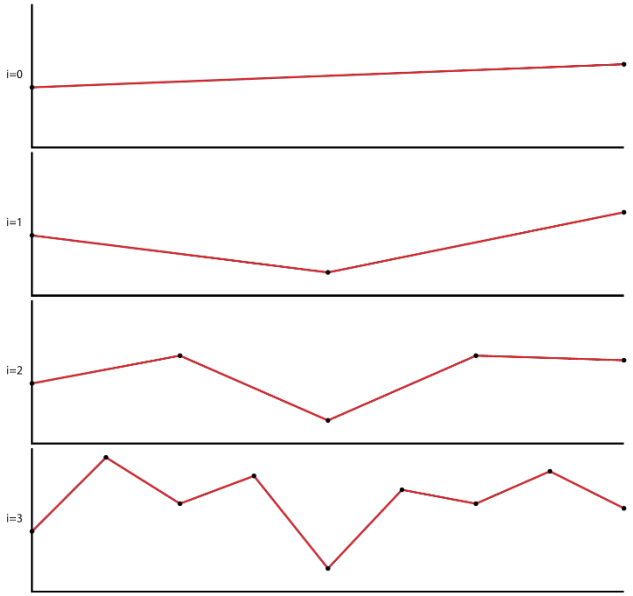
- Zum Nachbilden **natürlicher Strukturen** geeignet (z.B. Terrain)
- Auf beliebig viele Dimensionen erweiterbar
- Fraktale Struktur (selbstähnlich)
- Ist reproduzierbar (basiert auf Pseudo-Zufall)
- Alternativen: Perlin-Noise, Simplex-Noise, Wavelet-Noise



# Value Noise

Prinzip

- **Zufallswerte werden linear interpoliert** (in 2D bilinear)
- Mehrere Funktionen mit verschiedenen Amplituden und Frequenzen werden addiert (Oktaven)

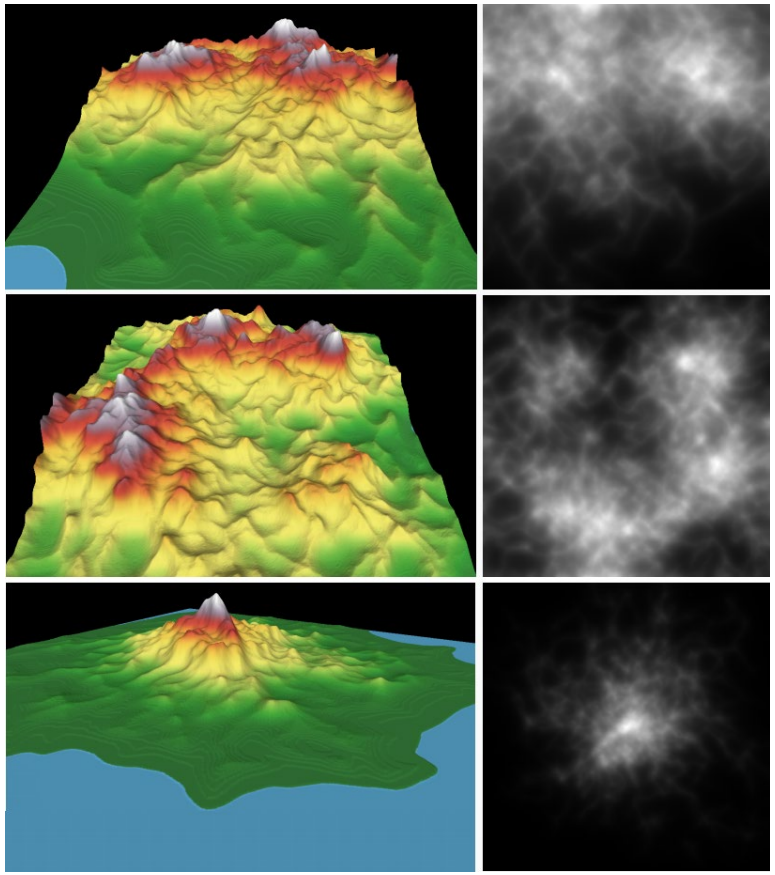



Frequenz = $2^i$
Amplitude = Persistenz$^i$

Persistenz wird typischerweise
Auf einen Wert 0 < x <= 1 gesetzt

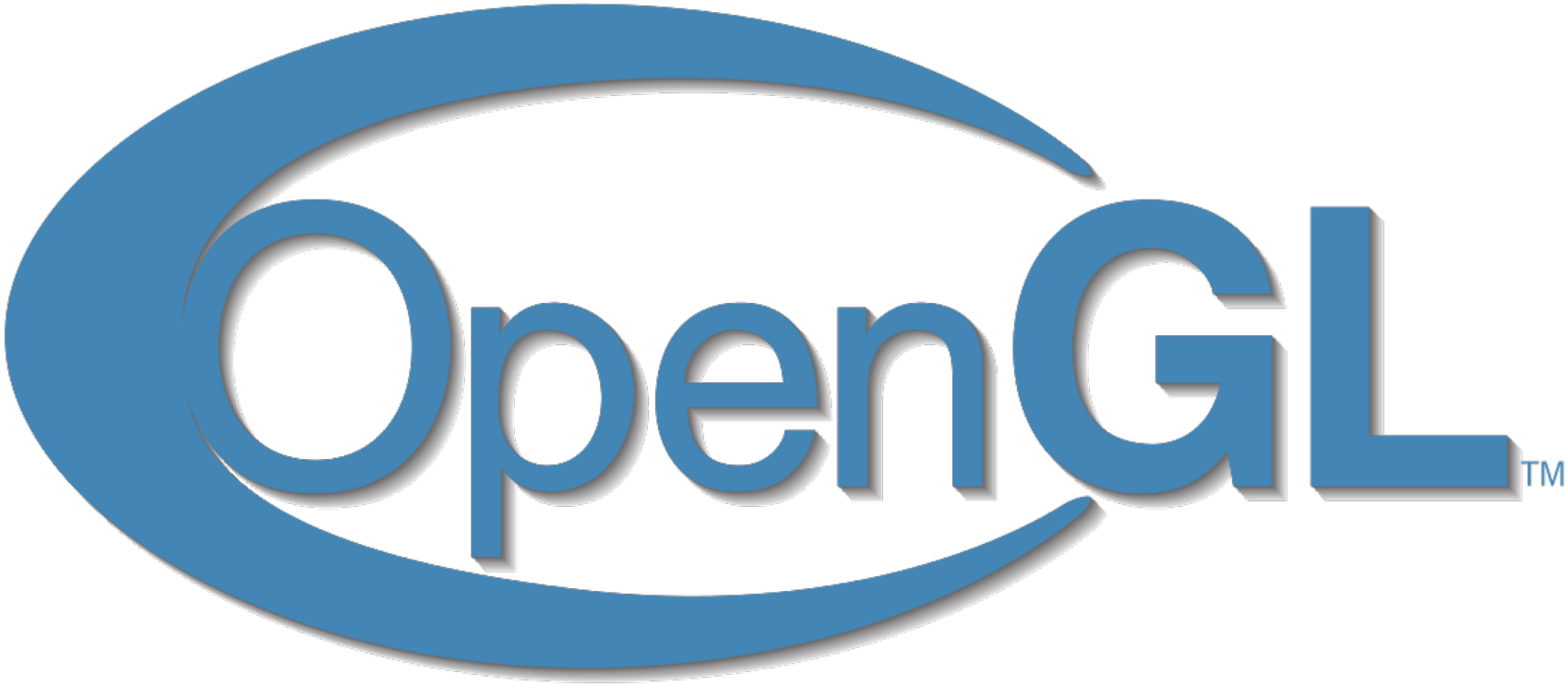
# Value Noise

## Interpretation als Höhe



http://1.bp.blogspot.com/-XJEtAk3nBvU/UtB_ov0IggI/AAAAAAAACjI/IneDgX4rpvI/s1600/123.png

# Sonstiges



# Explizite Uniform Locations

- *glGetUniformLocation* muss einen String-Vergleich ausführen, um die Location einer Uniform-Variable zu ermitteln (sollte in der Main-Loop vermieden werden)
- Locations können ab OpenGL 4.3 im Shader explizit festgelegt werden
- Ermöglicht Definition einer einheitlichen Schnittstelle für mehrere Shader
- Bei structs und Arrays werden fortlaufende Locations vergeben

Host

```
const int modelMatrixLocation = 0;
glUniformMatrix4f(modelMatrixLocation, 1, GL_FALSE, (GLfloat*)matrix);
```

Device

```
layout (location = 0) uniform mat4 ModelMatrix;
```

# Binding Points

- Können genutzt werden, um Texture- oder Image-Units festzulegen
- Keine Zuweisung per glUniform1i notwendig
- Ermöglicht Definition einer einheitlichen Schnittstelle für mehrere Shader

Host

```
// Zuweisung einer Textur auf die Texture-Unit 0
glActiveTexture(GL_TEXTURE0 + 0);
glBindTexture(GL_TEXTURE_2D, mainTexture);

// Zuweisung des Bildes (Level 0 der Textur mainTexture) auf die Image_Unit 0
glBindImageTexture(0, maintexture, 0, GL_FALSE, 0, GL_READ_WRITE, GL_RGBA8);
```

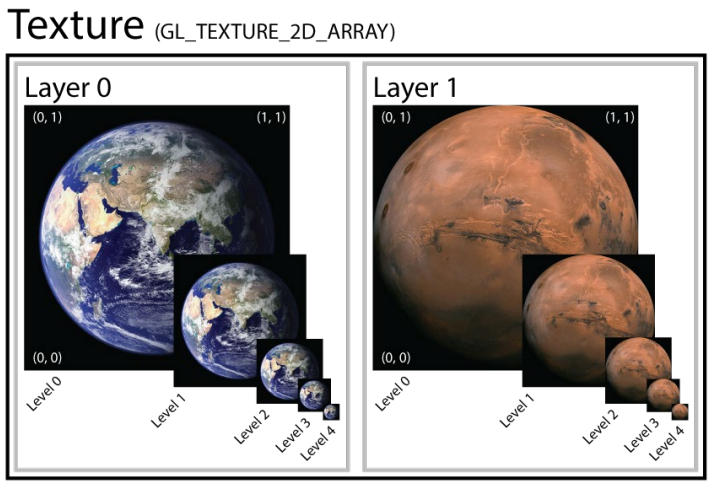
Device

```
// Sampler arbeitet auf Texture_Unit 0
layout (binding = 0) uniform sampler2D mainTexture;

// Zugriff auf Bild auf Image_Unit 0
layout (binding = 0) uniform image2D mainImage;
```

# Texture Units und „sampler“ (Wiederholung)

- Lesender Zugriff auf eine Textur aus dem Shader
- Einstellungen des Samples werden im Host vorgenommen
  - Können pro Textur/Sample Objekt vorgenommen werden.
- Zugriff auf die **komplette Textur** (alle Layer und Level)
- Textur wird an **Texture Unit** gebunden

# Texture Units und „sampler“ (Wiederholung)

Shader (beliebige Stufe)

```
// Deklaration der Sampler Variable (binding legt die Texture Unit fest)
layout (binding = 0) uniform sampler2D mySampler;

// …

// lesender Zugriff (potentiell auf ein beliebiges Mip Map Level)
vec4 color = texture(MySampler, fTexCoord);
```

Host

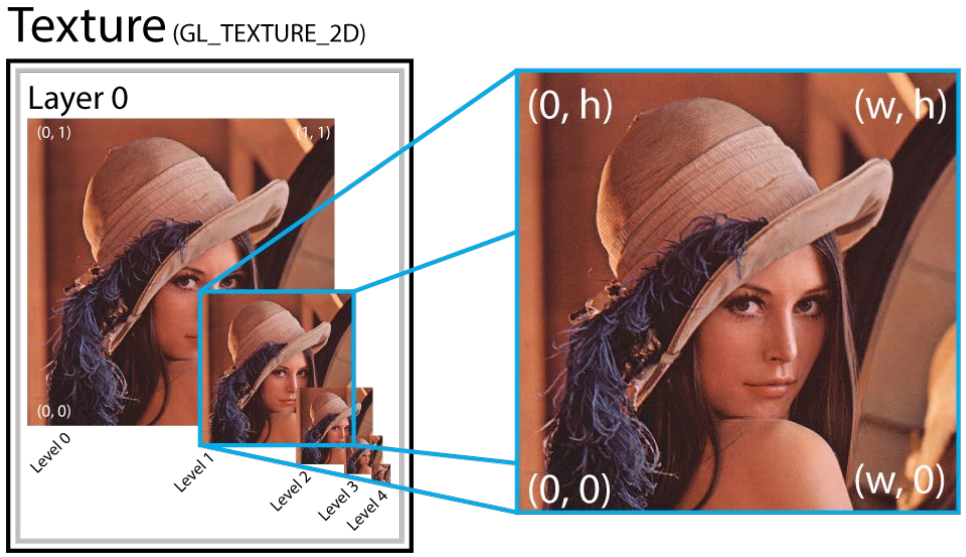
```
// Mit ´bindings´ ist kein Zuweisen der Texture Unit auf die Variablen notwending
glActiveTexture(GL_TEXTURE0 + 0);
glBindTexture(GL_TEXTURE_2D, texture);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

# Image Unit und „image“

- Lesender und schreibender Zugriff auf ein Bild (nicht Textur) aus dem Shader
- *Keine Einstellungen*, da Pixel-weises Lesen und Schreiben
- Nur ein **Level und Layer zugreifbar**
- Bild wird an **Image Unit zugreifbar**
- Notwendig für fortgeschrittene Technik (dann häufig mit Integer-Texturen)



# Image Unit und „image“

Host

```
// zuweisen eines einzelnen Layers und Levels
GLuint texture;
glGenTextures(1, &texture);

// …
// verwenden der Textur
GLuint binding = 0;
glBindImageTexture(binding, texture, level, layered, layer, access, format);
```

Shader (beliebige Stufe)

```
layout (rgba8, binding = 0) uniform image2D myImage; // Deklaration des Bildes ohne Sampler
// Achtung: hier muss das Bildformat definiert werden rgba8, rgba32f ….
// …

// speichern eines Vektors (Farbe) an der Position ´texelCoordinate1´
imageStore(myImage, texelCoordinate1, vec4(1, 1, 0, 1));
// lesen eines Vektors (Farbe) an der Position ´texelCoordinate2´
vec4 data = imageLoad(myImage, texelCoordinate2);
```

# Sky Map – Plattkarte

Sampling einer Plattkarte

- Winkel ϕ und θ müssen zu Textur-Koordinanten umgerechnet werden
- Mit *textureLod* kann das gesampelte Mipmap-Level bestimmt werden

```
vec4 sampleEquirectangular (sampler2D sampler, vec3 dir) {
        vec2 uv;
        uv.x = atan(dir.z, dir.x);
        uv.y = acos(dir.y);
        uv /= vec2(2 * PI, PI);

        return textureLod(sampler, uv, 0);
}
```

# Due Date

Termin 1: 23.01.

Termin 2: (bis zum)