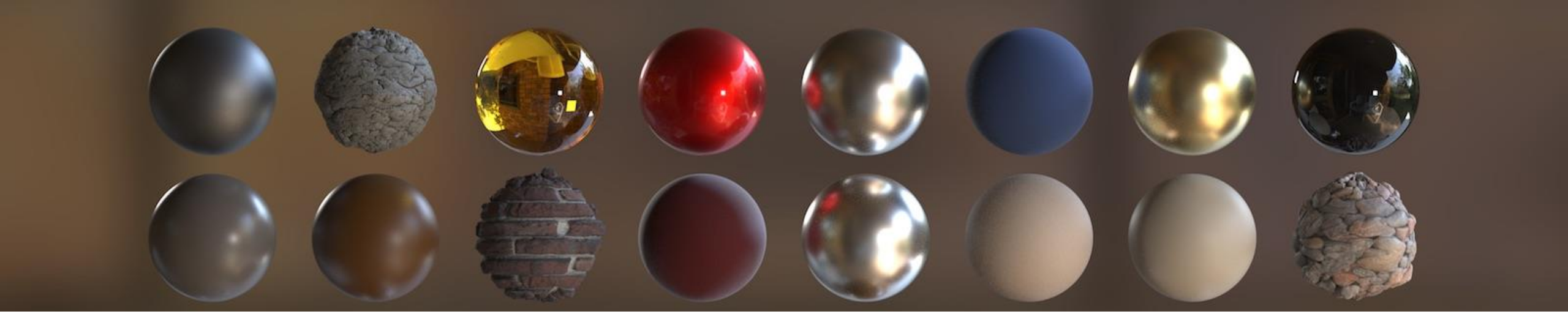
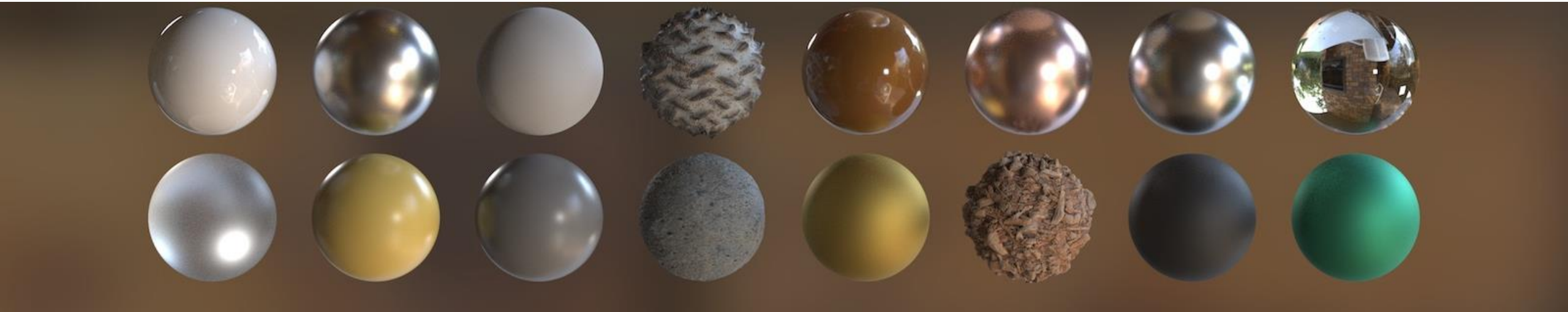
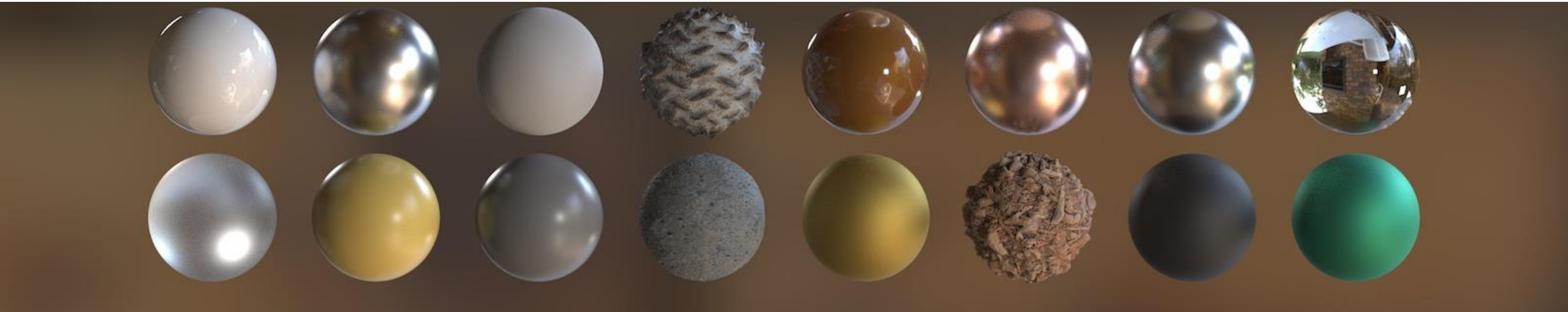


Shader Praktikum

FH Wedel



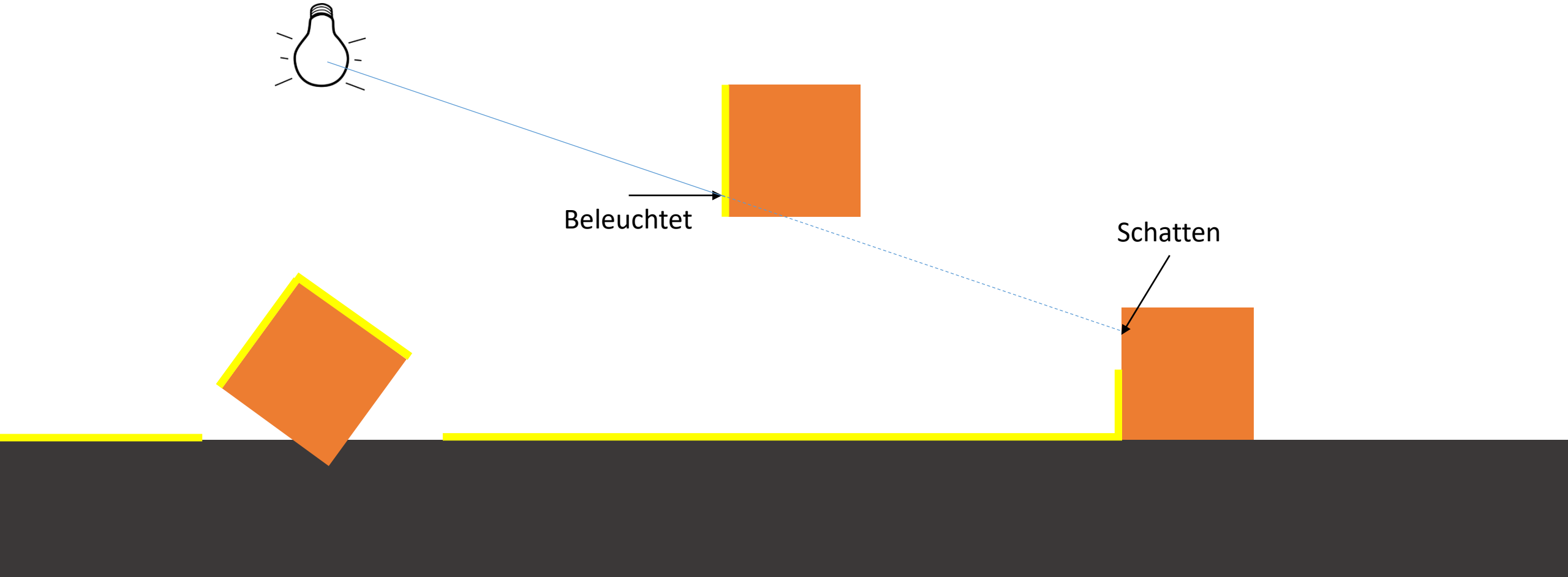
Shader – Aufgabe 4 (Zusatzinfos)



Themen

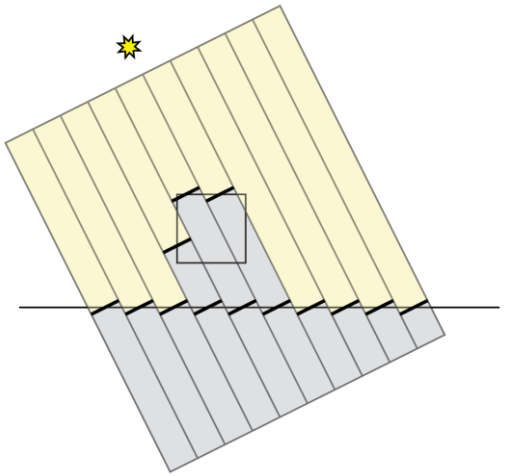
- Schatten
  - Punktschatten (Shadow-Maps)

Schatten



Shadow-Maps

- **Shadow-Mapping**
  - Rendern der Szene aus der Perspektive der Lichtquelle
- Algorithmus
  - Erzeugung einer **depth map** aus der aus der Perspektive der Lichtquelle
  - Samplen der depth map basierend auf der aktuellen Fragment Position
    - Fragmente aus dem Welt-Koordinatensystem auf die Shadow-Map projizieren
  - Vergleichen jedes Tiefenwertes mit dem gespeicherten Wert um zu schauen ob es sich im Schatten befindet oder nicht
    - Fragment „verschatten“, wenn der Tiefen-Wert der Shadow-Map geringer ist der Abstand der Lichtquelle zum aktuellen Fragment



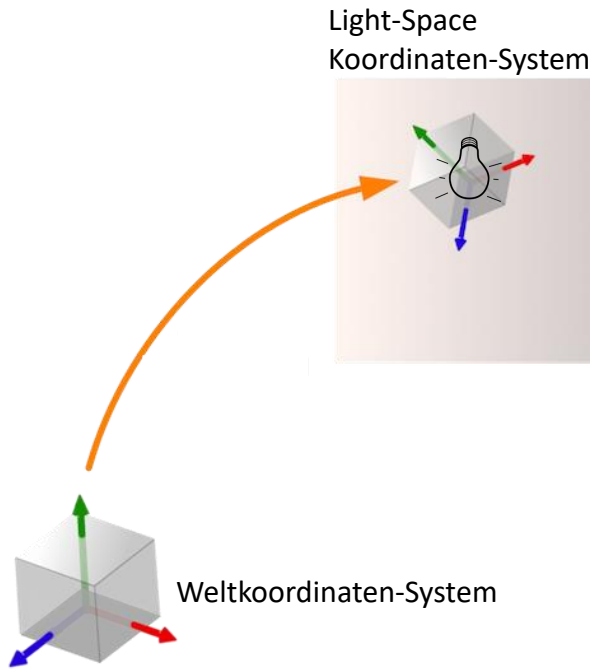
Shadow-Maps

- Beachte: Auflösung generell Quadratisch
  - Z.B. 1024x1024
  - glViewport muss angepasst werden, bei unterschiedlichen Auflösungen
    - Sonst wird die Shadow-Map zu klein sein oder nicht Komplett
- Es wird das Format *GL_DEPTH_COMPONENT* genutzt
- Als Interpolation (magnification function) wird GL_NEAREST genutzt
  - Das selbe gilt für die Textur minifying Funktion
- Da lediglich die Tiefenwerte gerendert werden, muss OpenGL explizit gesagt werden, dass nicht auf Farb-Buffer gerendert werden soll
  - *glDrawBuffer(GL_NONE)*
  - *glReadBuffer(GL_NONE)*

- Der Fragment Shader kann leer bleiben (alter. Setzten gl_FragDepth = gl_FragCoord.z)
- Vertex Shader: Transformation der Welt-Koordinanten in den Light-Space
  - *gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);*

Shadow-Maps

Transformations Matrix – Light Space Transformation



$T_{lightSpaceMatrix} = P_{lightProjectionMatrix} * V_{lightViewMatrix};$

Die Matrix $T_{lightSpaceMatrix}$ transformiert jeden Vektor im Welt-Koordinantensystem in den Lightspace.

Shadow-Maps

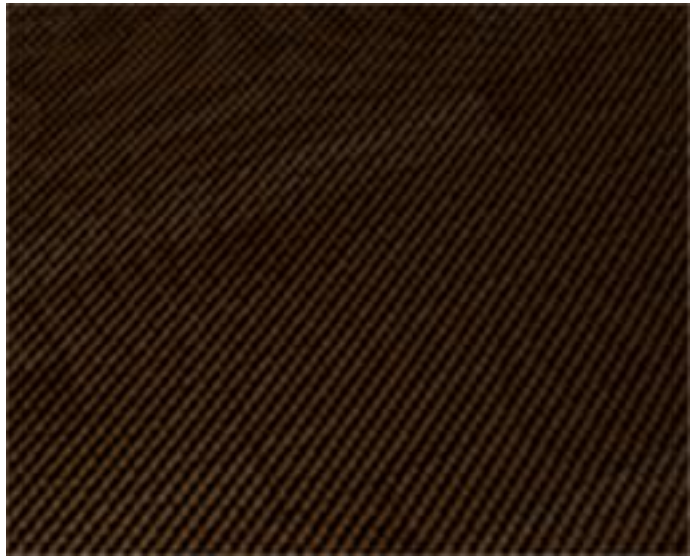
Rendering (2. Durchlauf)

- Im Fragment Shader
  - Vertex-Shader: konvertiert die FragPos *zusätzlich* in den LightSpace
- Schatten als Multiplikator in Licht-Berechnung
  - *lighting = … (1.0-shadowFactor)*(diffuse + specular) …;*
- Berechnen des Multiplikators input: vec4 fragPosLightSpace
  - Perspektivische Division (xzy / w) (bei perspektivischer Projection)
  - Transformation des Wertebereiches [-1,1] in den Bereich [0, 1] zum Sampeln
  - Sampeln aus der Shadow-Map mit xy-Koordinate (*texture(sM, coord.xz).r*)
  - Vergleich Sample mit aktueller Tiefe coord.z + **bias**
    - Wenn aktuelle Tiefe großer als gesampelte tiefe → Fragment im Schatten (1.0)
    - Sonst nicht im Schatten (0.0)

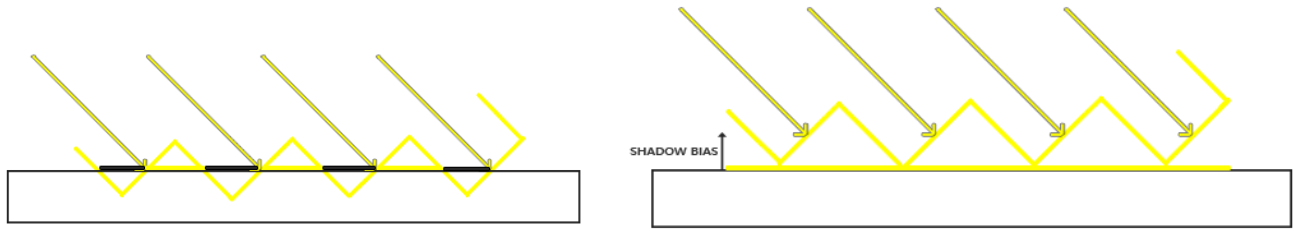
Shadow-Maps

Schatten Acne (Schattenartefakte)

- Linien aufgrund der (limitierten) Auflösung der Textur
  - Abweichung im Sampeln der Tiefe
- Abhängig vom Winkel
- Umgehen durch Addition eines Bias



Bias Abhängig vom Licht-Winkel
bias = max(maxBias * (1.0-dot(normal, lightDir)), minBias);



https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

Shadow-Maps

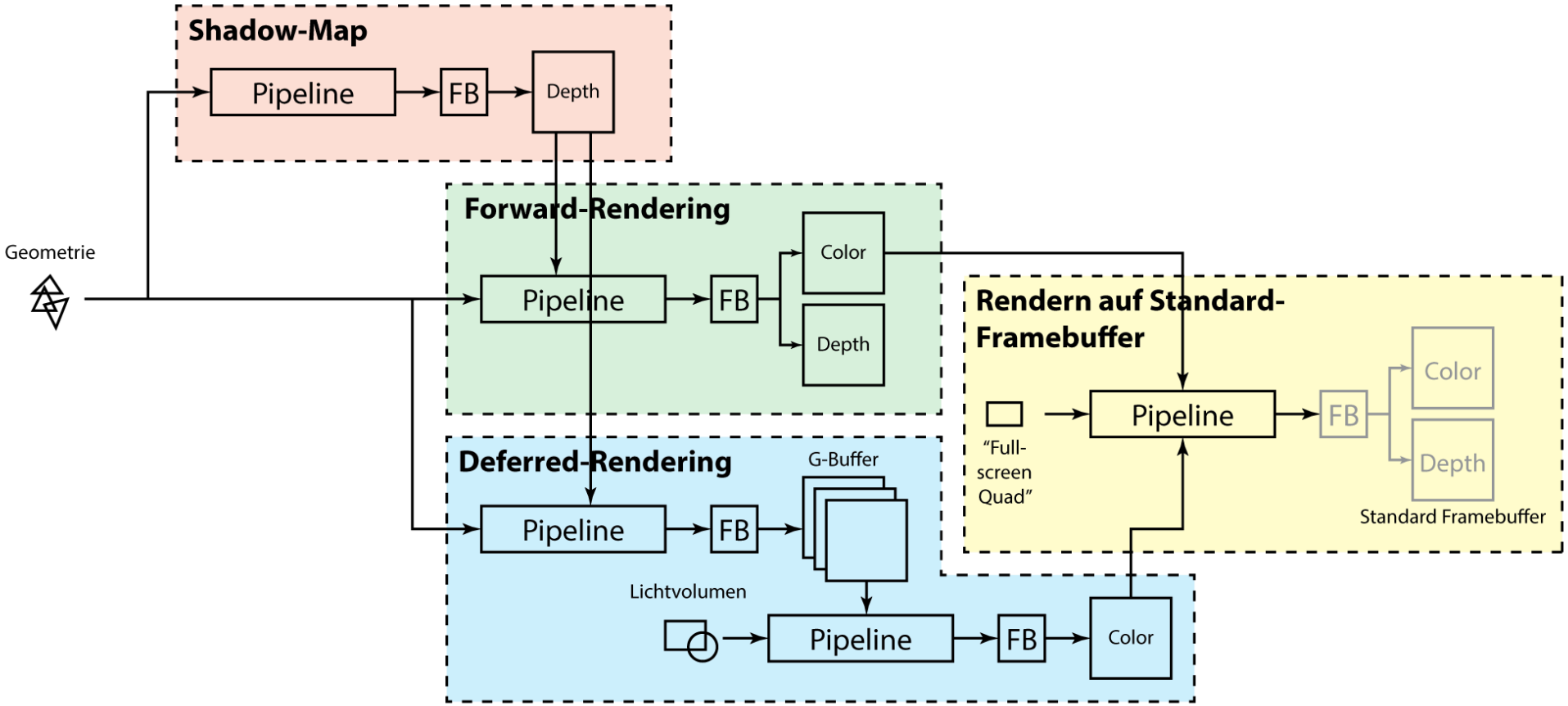
Percentage-closer filtering

- Artefakte (**zackige Kanten**)
  - Durch Shadow-Map Auflösung bedingt
- PCF → glätten der zackigen Kanten
  - Filterung: mehrere Samples
  - Samples um die Fragment Position
  - Durchschnitt aus Samples

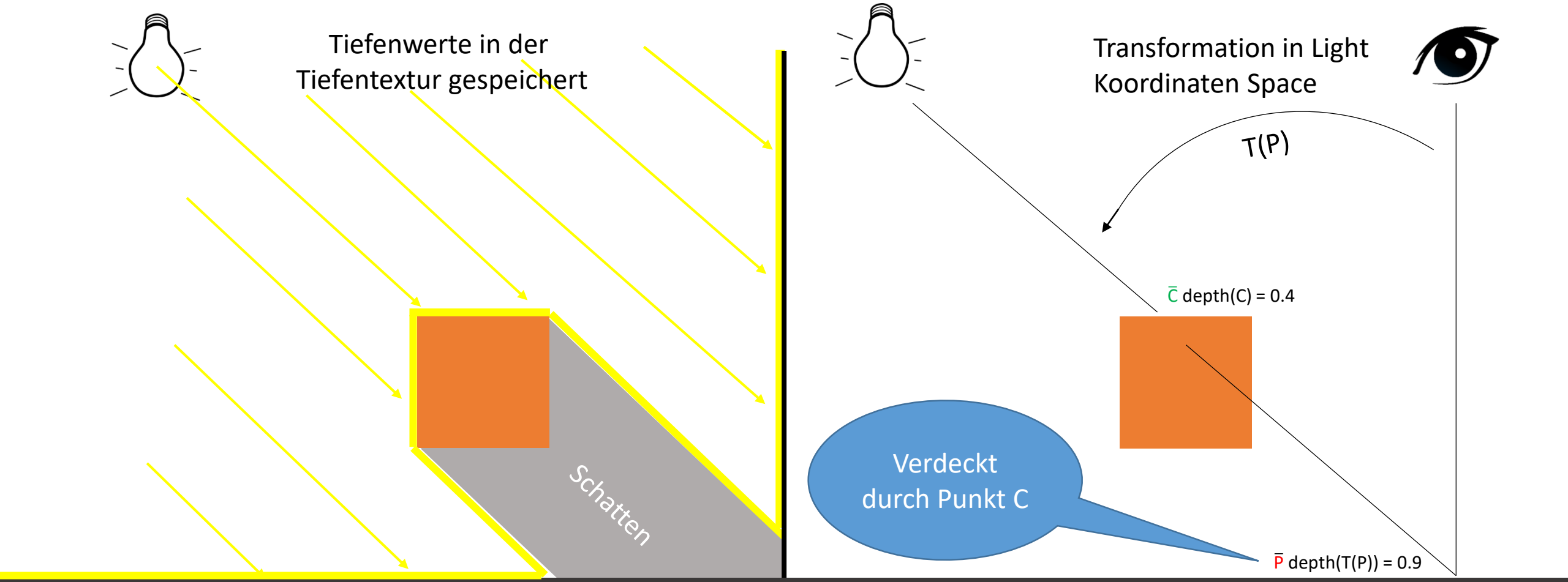
```
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
for( int x = -1; x <= 1; ++x) // 3x3 Sample Kernel
        for(int y = -1; y <= 1; ++y) {
                float Depth = texture(shadowMap, coord.xz +vec2(x, y) * texelSize).r; // Offset: vec2(x,y)
                shadow += currentDepth – bias ? 1.0 : 0.0;
        }
shadow = shadow / 9; // teilen durch Anzahl der Samples
```

https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

Shadow-Maps


Shadow-Map
Pipeline
FB
Depth
Forward-Rendering
Geometrie
Pipeline
FB
Color
Depth
Rendern auf Standard-Framebuffer
"Full-screen Quad"
Pipeline
FB
Color
Depth
Standard Framebuffer
Deferred-Rendering
G-Buffer
Pipeline
FB
Lichtvolumen
Pipeline
FB
Color

Schatten


Tiefenwerte in der
Tiefentextur gespeichert
Schatten
Transformation in Light
Koordinaten Space
T(P)
C̄ depth(C) = 0.4
Verdeckt
durch Punkt C
P̄ depth(T(P)) = 0.9

Directional Shadows

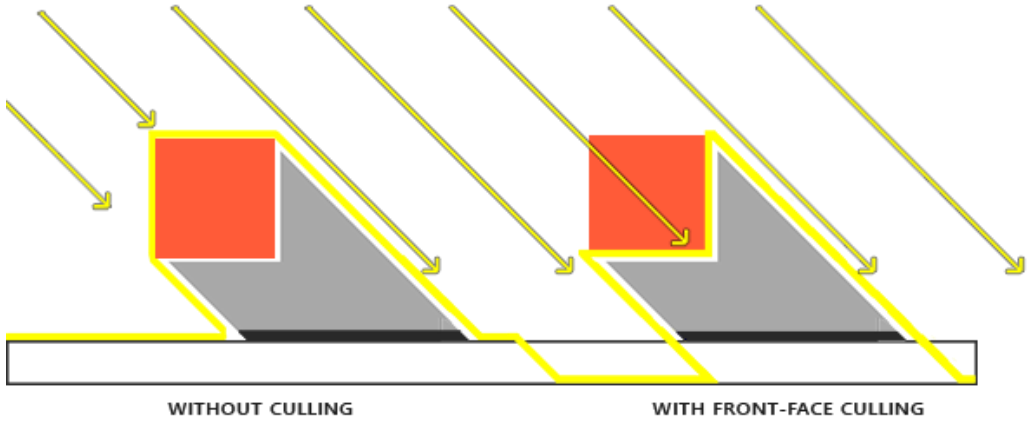
- Ein Directional Light hat keine Position
- Da Lichtstrahlen parallel sind
  - Orthografische Projektions-Matrix*
    - Achtung: Projektions-Matrix bestimmt was sichtbar ist und was geclippt wird
  - Bei einer Orthografischen Projektion, muss die perspektivische Division nicht durchgeführt werden verändert aber den Wert auch nicht

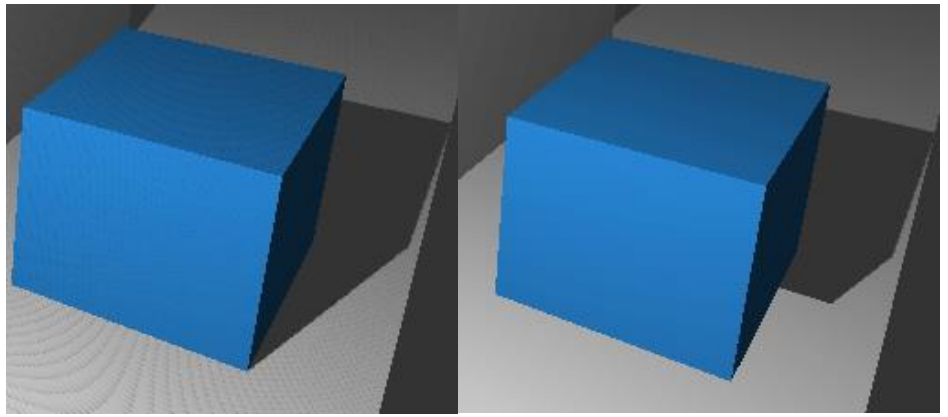
* bei Spot-Lights perspektivisch

Directional Shadows

Peter Panning

- Durch Bias erzeugter Offset
  - Verfälscht ggf. Tiefenwerte
- Lösung
  - Erstellen der Shadow-Map mit Frontface-Culling *(glCullFace(GL_FRONT))*





Peter Panning

https://hub.packtpub.com/global-illumination/
https://i.ytimg.com/vi/UvheNWRk5N4/hqdefault.jpg

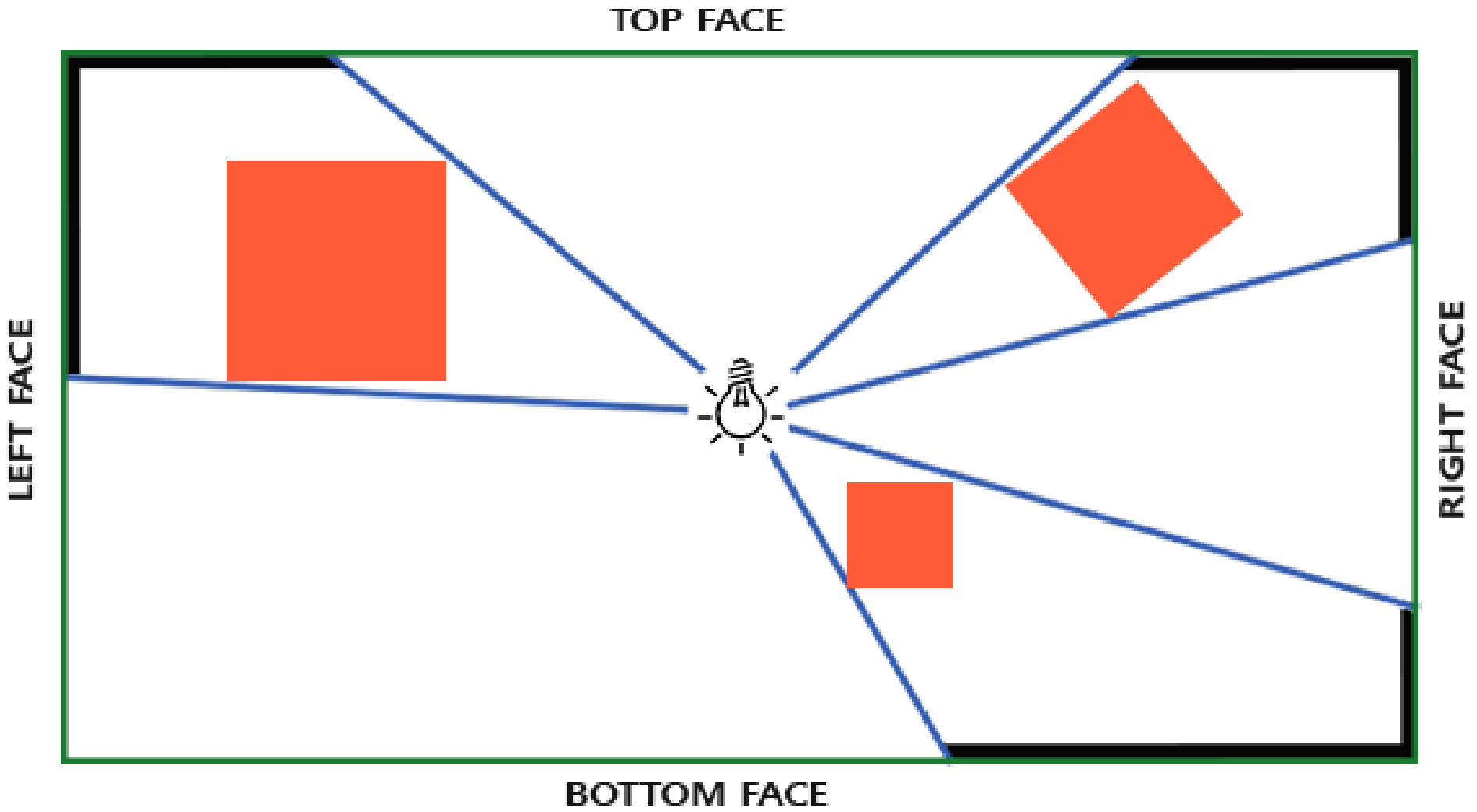
https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

Directional Shadows

- Umgehen von Übersampling (Abschneiden an Textur Kanten außerhalb des Light-Frustum)
  - GL_CLAMP_TO_BORDER
  - ***float*** *borderColor [] = {1.0f, 1.0f, 1.0f, 1.0f };*
  - *glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);*
  - Alle Bereiche außerhalb des Schatten-Map-Bereichs sind nun 1.0 (dadurch kein Schatten!)
- Außerhalb der Farplane:
  - Wenn die aktuelle Tiefe der Fragment-Position größer als 1.0 ist, wird kein Schatten angenommen

https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

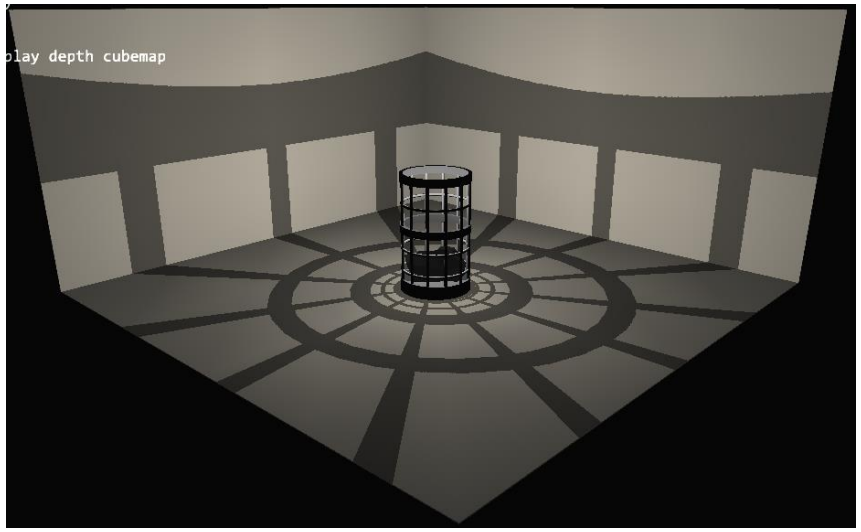
Point Shadows



https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows

Point Shadows

- dynamische Schatten für **Punktlichtquellen**
- **Omnidirectional Shadow Maps**
- Algorithmus bleibt weitestgehend gleich
- Nutzung einer **Cubemap** zum Rendern der Shadow-Map
  - 6x Rendern der Szene, für alle Faces der Cubemap
  - Als Schleife mit 6 View-Matrizen ggf. sehr performance-lastig
    - Nutzung eines Tricks mithilfe des Geometrie-Shaders
  - Textur Type: GL_TEXTURE_CUBE_MAP
  - *glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthCubemap, 0)*



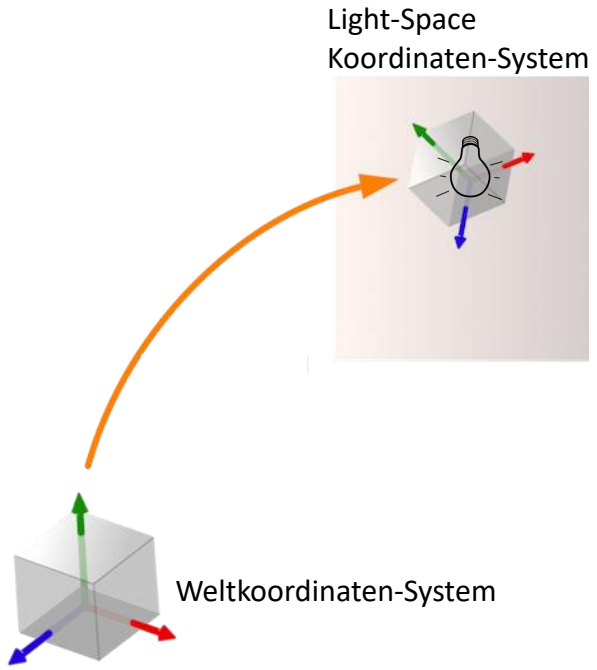
https://toutiao.io/posts/245778/app_preview

Point Shadows

Light space transform

- Light Space Transformations-Matrix **T**
  - Für **jede Fläche** des Würfels
  - Daher: (für jede Fläche)
    - eine **Projektion**-Matrix (gleich für alle Flächen) → **perspektivisch**
      - Field of view Parameter: 90 Grad, damit die Flächen korrekt aneinander liegen
    - eine **View**-Matrix (unterschiedlich)

Look-At Werte		
lightPos	lightPos+vec3(1,0,0)	vec3(0,-1,0)
lightPos	lightPos+vec3(-1,0,0)	vec3(0,-1,0)
lightPos	lightPos+vec3(0,1,0)	vec3(0,0,1)
lightPos	lightPos+vec3(0,-1,0)	vec3(0,0,-1)
lightPos	lightPos+vec3(0,0,1)	vec3(0,-1,0)
lightPos	lightPos+vec3(0,0,-1)	vec3(0,-1,0)



Point Shadows

Rendern Tiefenwerte in die Cubemap schreiben

Vertex-Shader

- Weitergabe der Vert. Poistion-Welt-Koordinante

Geometry-Shader

- Transformation der Vertices in die entsprechenden Light-Spaces
  - Array aus LightSpace-Matrizen (selbe Projektions-Matrix, vers. View-Matrizen)
- Nutzung *gl_Layer* um zu entscheiden, auf welches Cube_Map-Face wir rendern
  - Cube-Map → layered (faces) Framebuffer (gl_Layer → Fläche der Cube-Map)
  - z.B. in Verbindung mit *Invocations* effizienter (parallel)

Fragment-Shader

- Berechnung des (linearen) Abstandes der Lichtquelle zur Fragment-Position (world-space) & Mapping auf [0, 1] (Division durch far_plane)
  - Dadurch kann die Textur zu Debug-Zwecken einfach visualisiert werden
- schreiben auf build-in Variable: gl_FragDepth

Point Shadows

Schatten-Berechnung mithilfe der Cube-(Depth)-Map

Input:

- Fragment-Position *(World-Space)*
- Far clipping plane *(vgl. Frustum der Projektions-Matrix für Light-Space)*

Output:

- Float Wert (1.0 – Verschattet; 0.0 Beleuchtet)

Algorithmus:

- Berechnung Vektor zwischen Fragment Position & Licht Position
- Nutzung dieses (Richtungs-Vektors) um aus der Cubemap zu Samplen
- Transformation des float-Wertes aus dem linearem Wertebereich [0,1] auf richtige Tiefe (mit far_plane multiplizieren)
- Nutzung der GLSL *length* Funktion um den Betrag des Vektors zu errechnen
- Vergleich des gesampelten wertes *+ bias* mit echter Tiefe des aktuellen Fragments im Light Space (implizit durch berechnete Entfernung)

Point Shadows (PCF)

Percentage closer filtering

- Samplen aus einer Cube-Map mit einem **Richtungs-Vektor**
- Unmöglich diese (Sample-Kernel) optimal zu berechnen
- Nutzung eines **vorberechneten Arrays aus Offsets**
  - Abdeckung aller Umgebungsrichtungen
- Cube-Maps können auch mit nicht-normalisierten Vektoren gesamplet werden
- Einige Vektoren (Samples sind jedoch Redundant)

http://www.sunandblackcat.com/tipFullView.php?l=eng&topicid=36

https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows

Point Shadows (PCF)

Percentage-closer filtering

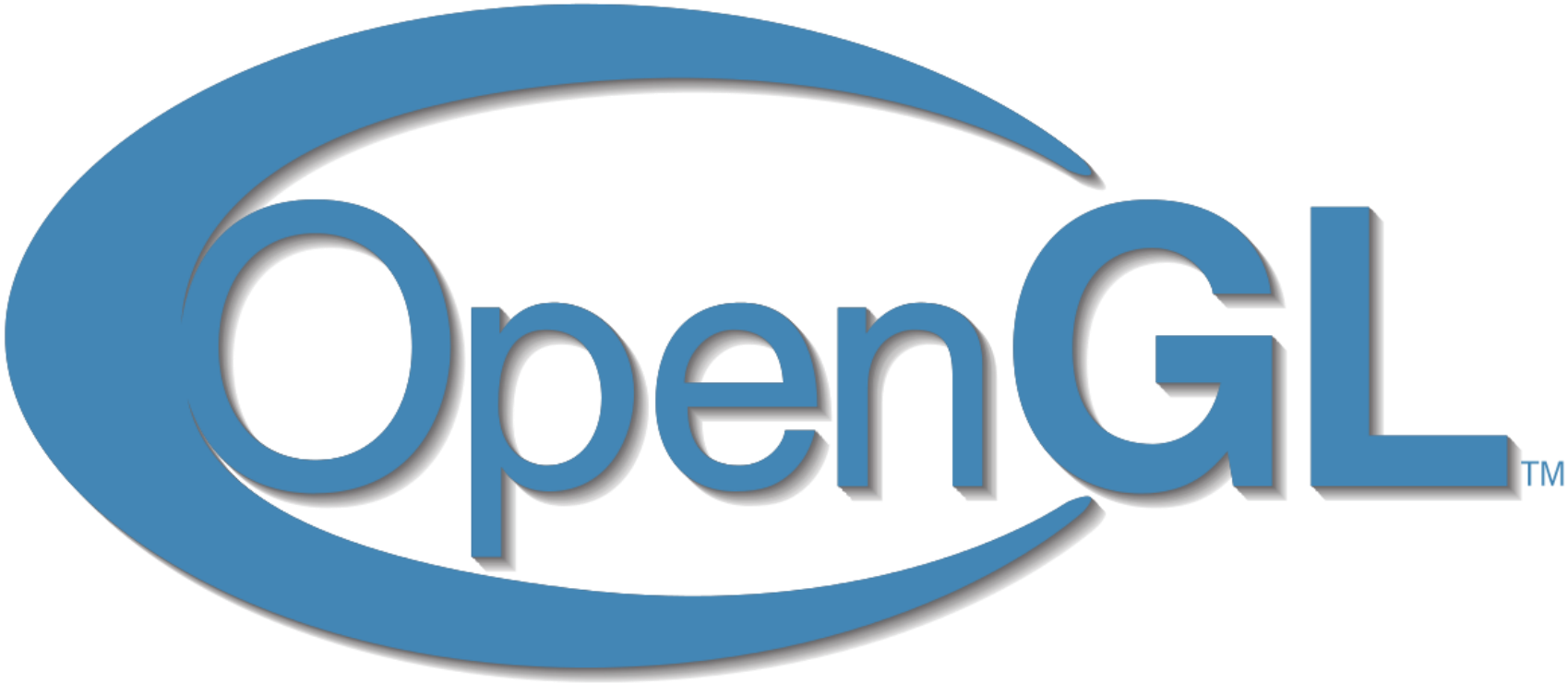
```
// Array aus Offset Richtungen für das Samplen, jedes zeigt in eine komplett andere Richtung (weniger Redundanz)
vec3 gridsamplingDisk[20] = vec3[]
(
   vec3(1, 1, 1),  vec3(1, -1, 1),   vec3(-1, -1, 1),  vec3(-1, 1, 1),
   vec3(1, 1, -1), vec3(1, -1, -1), vec3(-1, -1, -1), vec3(-1, 1, -1),
   vec3(1, 1, 0),  vec3(1, -1, 0),  vec3(-1, -1, 0),   vec3(-1, 1, 0),
   vec3(1, 0, 1),  vec3(-1, 0, 1),  vec3(1, 0, -1),    vec3(-1, 0, -1),
   vec3(0, 1, 1),  vec3(0, -1, 1),  vec3(0, -1, -1),   vec3(0, 1, -1)
);

//...
// sample mit i = 0; i < 20 ; i++
float closestDepth = texture(depthMap, fragToLight + sampleOffsetDirections[i] * diskRadius).r;
// Transformierung in den Wertebereich [0, far_plane] vergleich mit aktueller Tiefe
// erhöhen des Schatten-Wertes, nach Schleifendurchlauf, muss dieser durch die Anzahl der Samples geteilt
// werden (hier  shadow /= 20.0; )
```

Shadow Problem Behandlung

- Diverse Methoden zum Verbessern der Qualität
- Weiter Informationen
  - http://www.sunandblackcat.com/tipFullView.php?l=eng&topicid=35

Sonstiges



Random Noise Texture

Host

```
// 4x4 Array aus zufälligen Vektoren
MatVec3 noise[16];
for (unsigned int i = 0; i < 16; i++) {
        noise[i].x = randomFloat() * 2.0 – 1.0;
        noise[i].y = randomFloat() * 2.0 – 1.0;
        noise[i].z = 0.0f;                                  //  Rotierung um die z-Achse
}

unsigned int noiseTexture;
glGenTextures(1, &noiseTexture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, 4, 4, 0, GL_RGB, GL_FLOAT, noise);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT); // wiederholt sich über den ganzen
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT); // …. Bildschirm
```

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);
```

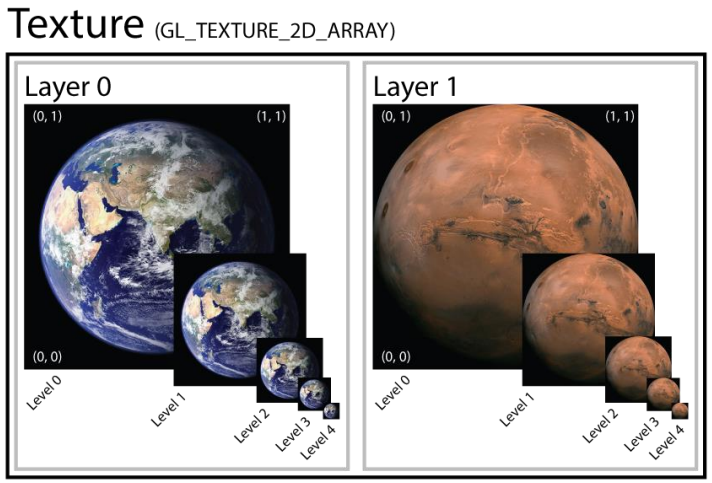
Device

```
// 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)

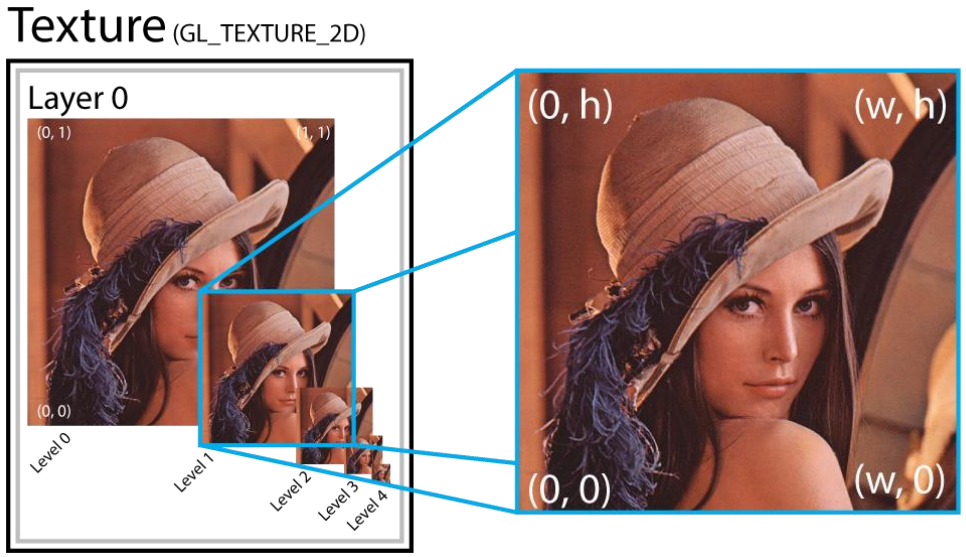


Image Unit und „image“

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

// …

// 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);
```

Due Date

16.01.