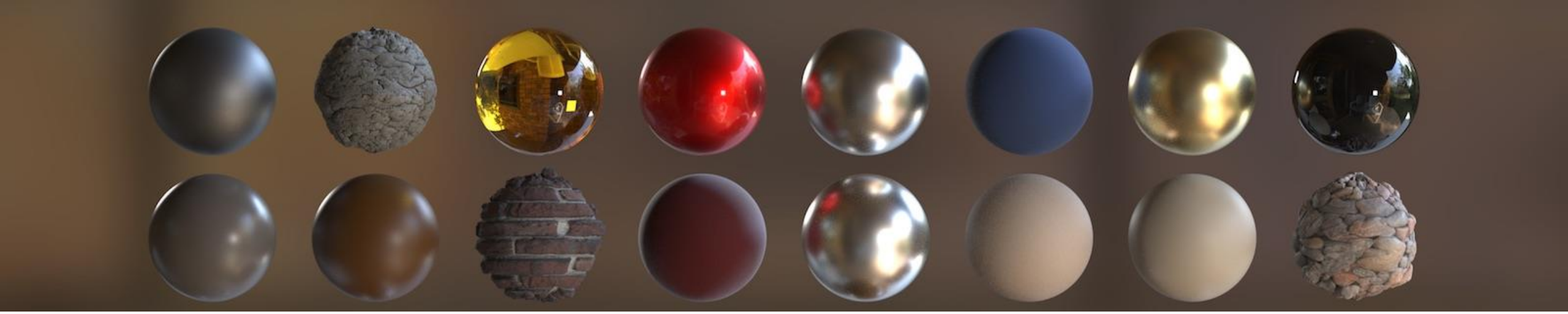
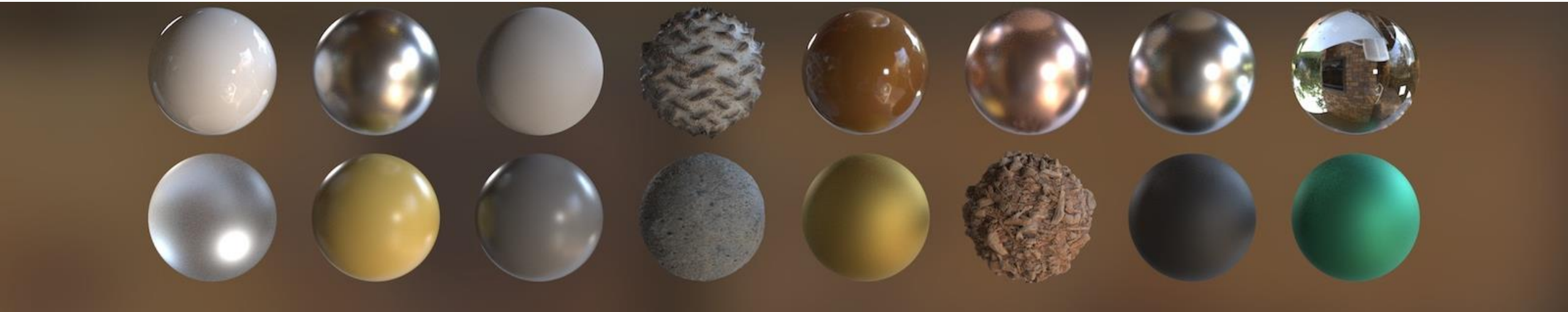
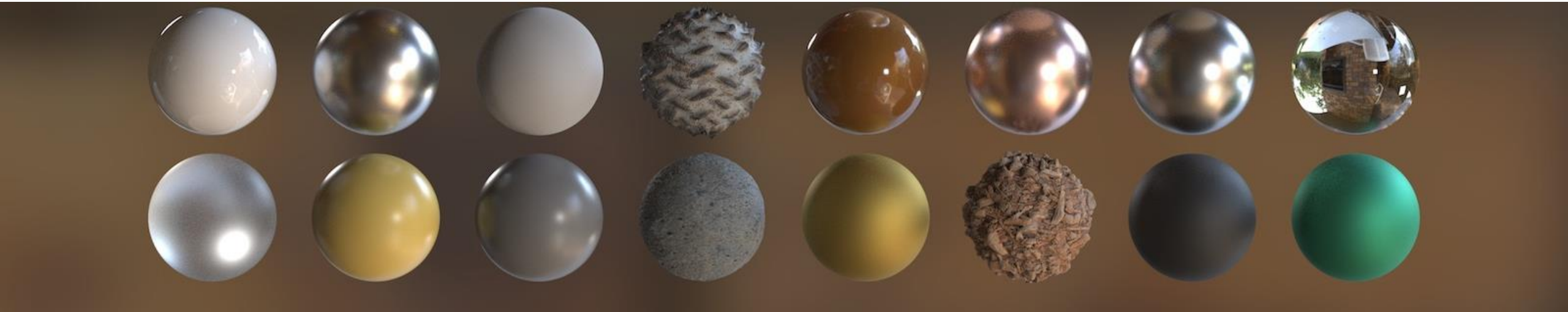


Shader Praktikum

FH Wedel



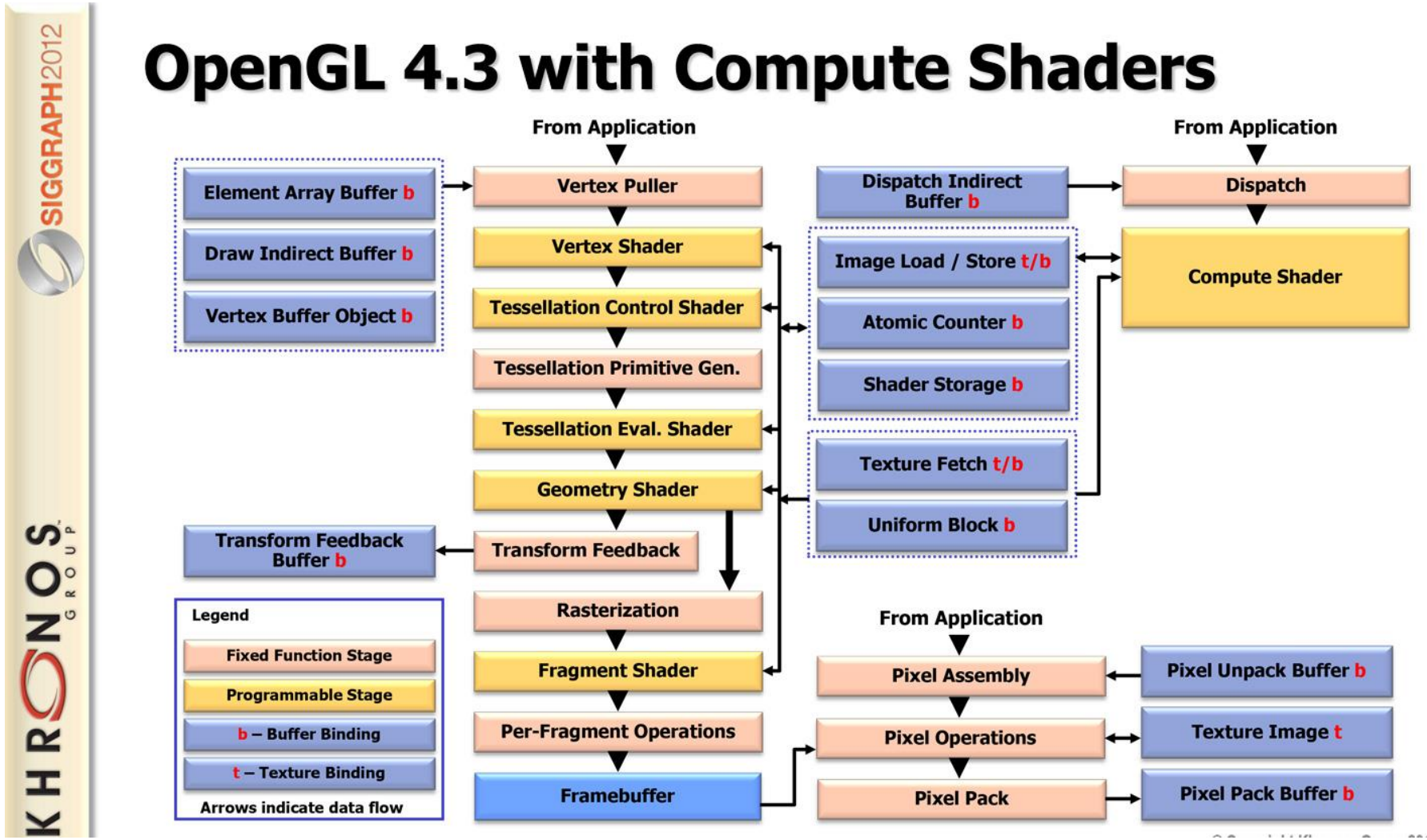
Shader – Aufgabe 3



Themen

- **Deferred Shading**
  - **Framebuffer**
  - **Stencil & Depth Testing**
  - **Light-Volumes**
- Special Effects (Post-Processing)
  - HDR
    - Tone Mapping
  - Gamma Korrektur
  - Bloom
  - Filterung und andere Effekte
- Sky Map

Framebuffer Object



https://www.khronos.org/files/opengl45-quick-reference-card.pdf

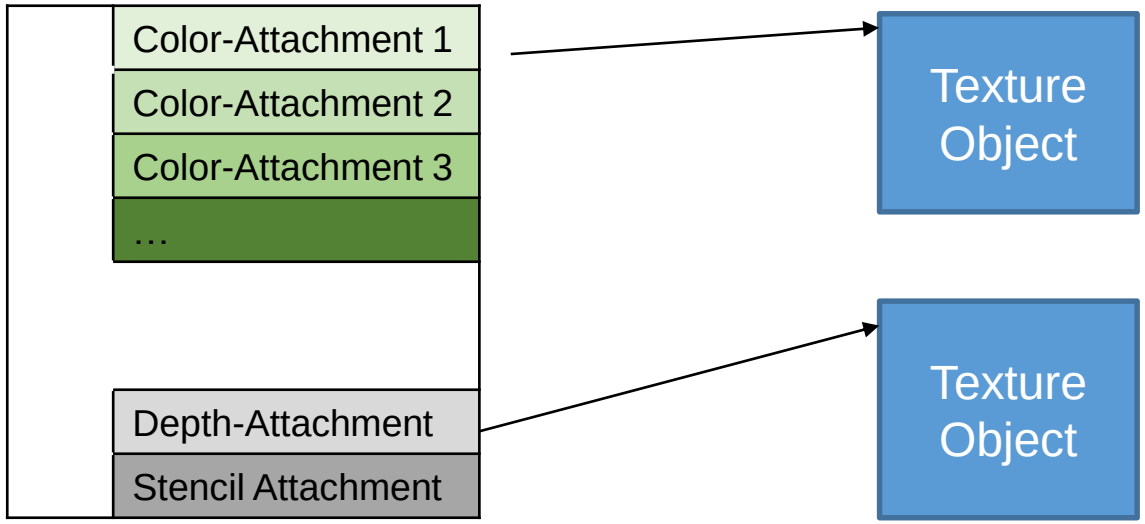
Framebuffer Object

- 2 Arten von **Framebuffern**
  - Vom Window-System erzeugt
    - Beim Erzeugen eines Fenster mit z.B. GLUT oder GLFW wird automatisch ein Framebuffer bereitgestellt. Dieser kann nach dem Erzeugen nicht weiter durch GL-Funktionen angepasst werden.
  - Vom Benutzer erzeugt
    - Der Benutzer kann **zusätzliche Framebuffer** erzeugen, um z.B. Zwischenergebnisse der Szene zu rendern. Dieser Framebuffer können zu jederzeit erzeugt und auch verändert werden.
    - **Framebuffer Objects** kapseln den **Zustand** dieser **Framebuffer**, so wie Texture Objects den Zustand einer Textur kapseln.

Framebuffer Object

Attachments

Pro **Framebuffer Object** kann jeweils eine Textur als Depth und Stencil Buffer genutzt werden. Mehrere Texturen können für benutzerdefinierte Ausgaben (**Color Attachments**) des Fragment Shaders genutzt werden.



Alle Texturen müssen die **gleiche Auflösung** haben.

(Wenn beispielsweise der Depth Buffer kleiner wäre als die Farb-Texturen (Color Attachments), könnte der Tiefentest nicht korrekt ausgeführt werden)

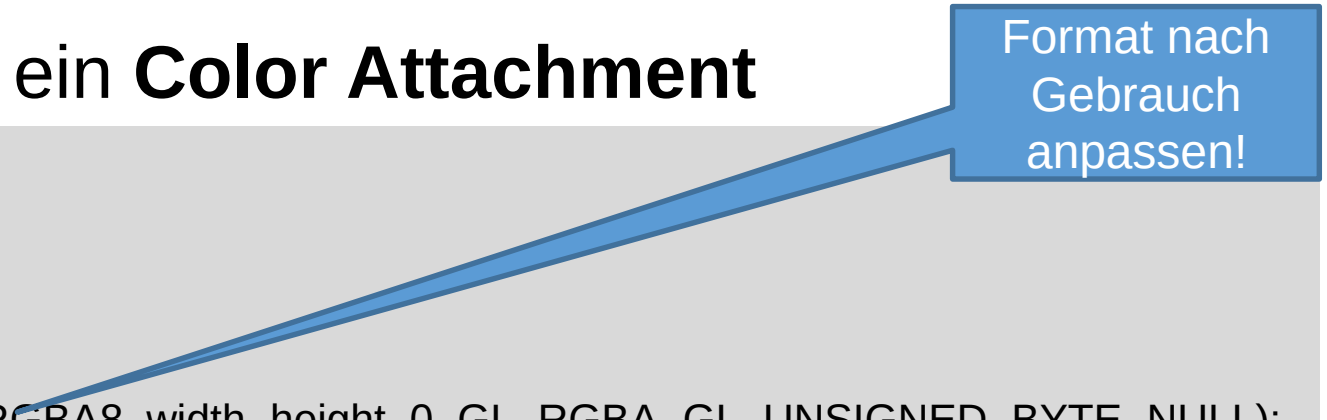
Framebuffer Object

Framebuffer Objects werden wie auch andere OpenGL-Objekte mit *glGen** erzeugt und mit *glBind** als aktuell markiert. Mit *glFramebufferTextur** können Texturen an das **Framebuffer Object** gebunden werden.

```
GLuint fbo;

glGenFramebuffer(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, colorTexture1, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, colorTexture2, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthTexture, 0);

if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
        // Fehlerbehandlung bzw. Fehlerausgabe
}
glBindFramebuffer(GL_FRAMEBUFFER, 0); // Bind Default Framebuffer
```

Framebuffer Object

Erstellen einer Textur für ein **Color Attachment**




```
GLuint id;

glGenTextures(1, &id);
glBindTextures(GL_TEXTURE_2D, id);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);

glTexParametri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParametri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
// Was könnte noch benötigt werden?
glBindTexture(GL_TEXTURE_2D, 0);
```

Für das **Depth Attachment** muss die Textur ein anderes Format haben

```
…
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, width, height, 0, GL_DEPTH_COMPONENT,
GL_UNSIGNED_BYTE, NULL);
…
```

Framebuffer Object

Texture vs. Renderbuffer

Für die Attachments eines **Framebuffer Objects** können **Texturen** oder **Renderbuffer** genutzt werden.

Der Zugriff auf letztere kann unter Umständen vom Treiber besser optimiert werden, da sie keine Möglichkeit zum Filtern (Zugriff aus dem Shader) bereitstellen. Bei vielen Algorithmen bietet es sich an, für den **Depth Buffer einen Renderbuffer** zu verwenden.

```
// Renderbuffer
glGenRenderbuffers(1, &rboDepth);
glBindRenderbuffer(GL_RENDERBUFFER, rboDepth);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, SCR_WIDTH, SCR_HEIGHT);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, rboDepth);
```

Framebuffer Object

Zum Rendern mit einem Framebuffer Object muss lediglich ein Aufruf an *glBindFramebuffer* stattfinden. Die darauf folgenden Draw Commands **speichern die produzierten Pixel dann in den Texturen des FBO**. (Für den Depth & Stencil Test müssen Werte aus den Texturen gelesen werden).

```
// Rendern auf den eigenen Framebuffer (GL_DRAW_FRAMEBUFFER zum Rendern)
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo);

// Viewport auf die Größe des eigenen Framebuffers einstellen (wenn unt. Auflösung)
glViewport(0,0, fboWidth, fboHeight);

// Normaler Rendering-Prozess
glUseProgram(program);
glDrawElements(GL_TRIANGLES, countIndices, GL_UNSIGNED_INT, 0);
glUseProgram(0);

// Rendern auf Standard-Framebuffer aktivieren
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glViewport(0,0, defaultFramebufferWidth, defaultFramebufferHeight);
```

Framebuffer Object

Targets für *glBindFramebuffer*

GL_DRAW_FRAMEBUFFER
	Ziel aller Draw-Commands
GL_READ_FRAMEBUFFER
	Lesen vom Framebuffer (z.B. glReadPixels, glBlitFramebuffer …)
GL_FRAMEBUFFER
	beide Targets gesetzt (lesen & schreiben)

(Texturen am Framebuffer-Objekt können anschließend wie andere Texturen gesamplet werden)

MRT

Multi Render Targets

- Ermöglicht das Rendern auf mehr als einen Output im Fragment Shader
  - Verschiedene Bilder Outputs
- Bindung mehrerer Color-Attachments glDrawBuffer

```
…
// Erstellung verschiedener Framebuffer Attachments, Texturen müssen gleiche Auflösung haben
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, colorBuffer, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, colorBuffer2, 0)
….

// openGL explizit sagen, welche Color-Attachments in welcher Reihenfolge genutzt werden sollen
GLuint attachments[2] = {
        GL_COLOR_ATTACHMENT0,
        GL_COLOR_ATTACHMENT1
};  // Array verknüpft locations im Shader mit den Attachments des FBO (locations = Array-Indizes)

glDrawBuffers(2, attachments);
```

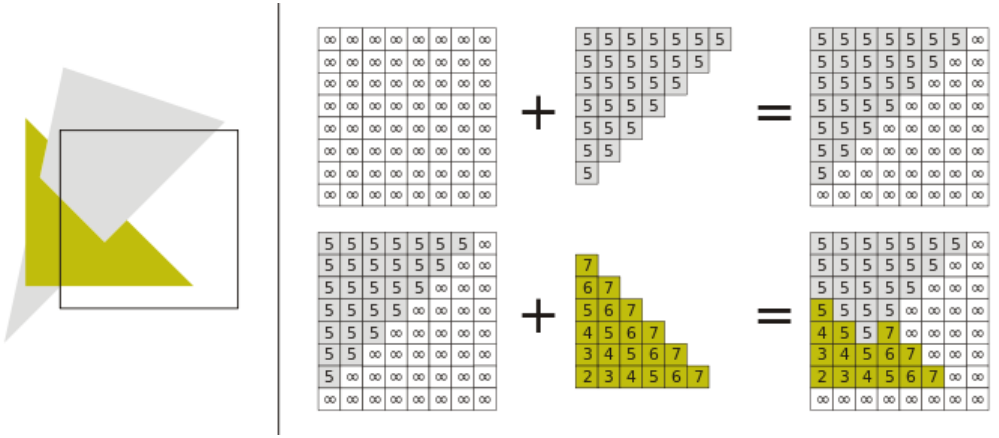
MRT

```
// Fragment Shader für MRT
layout (location = 0) out vec4 FragColor; // Texturformat ist RGBA
layout (location = 1) out vec3 FragNormal; // Textturformat ist RGB

…

void main() {
        FragColor = vec4 (1, 0, 0, 1);
        FragNormal = vec3 (0, 1, 0);
}
```

Bei diesem Beispiel ist FragColor mit GL_COLOR_ATTACHMENT0 verbunden, so dass dieser Wert auf die Textur colorBuffer geschrieben wird..
Beim Schreiben auf FragNormal wird auf GL_COLOR_ATTACHMENT1 (→ colorBuffer2) geschrieben

Depth Buffer (zBuffer)

- Speichert **Tiefenwerte** (das Fragments) [0.0, 1.0]
- Teil des FBO
  - Basis internes Format: *GL_DEPTH_COMPONENT*
- Zur **Verdeckungsberechnung**
- **Depth-Test** (in Screen-Space) in OpenGL: OpenGL testet, ob der Depth Wert des Fragments *(gl_FragCoord.z)* kleiner ist als der im Depth Buffer
  - Erfolgreich: Rendern des Fragments & Update des Depth Wertes
  - Nicht Erfolgreich: Fragment wird verworfen



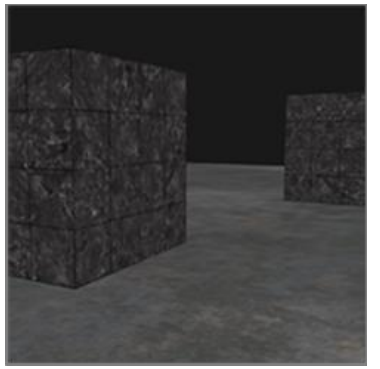
https://de.wikipedia.org/wiki/Z-Buffer#/media/File:Z-buffer.svg

Depth Buffer (zBuffer)

```
// Depth Testing ist per Default deaktiviert, daher: Aktivierung des Depth Testings
glEnable(GL_DEPTH_TEST);
…
// Löschen des depth buffers der letzten Iteration
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
…
// Read-only Depth-Buffer, lesen aus Depth-Buffer für Depth Test aber ohne Update der Depth-Buffer Wertes
glDepthMask(GL_FALSE);
…
// Depth Test Funktion
glDepthFunc(GL_LESS); // setzen eine Vergleichsoperators für den Depth Test
// GL_ALWAYS                Depth Test besteht immer
// GL_NEVER                 Depth Test besteht nie
// GL_LESS (Default!)       Depth Test besteht, wenn Frag. Depth Wert   <    Depth Buffer Wert
// GL_EQUAL                 Depth Test besteht, wenn Frag. Depth Wert   =    Depth Buffer Wert
// GL_LEQUAL                Depth Test besteht, wenn Frag. Depth Wert   <=   Depth Buffer Wert
// GL_GREATER               Depth Test besteht, wenn Frag. Depth Wert   >    Depth Buffer Wert
// GL_NOTEQUAL              Depth Test besteht, wenn Frag. Depth Wert   !=   Depth Buffer Wert
// GL_GEQUAL                Depth Test besteht, wenn Frag. Depth Wert   >=   Depth Buffer Wert
```

Stencil Buffer

- Teil des Framebuffers
  - Kann als eigener Buffer (*GL_STENCIL_INDEX*) oder mit dem Depth Buffer zusammen gespeichert werden (*GL_DEPTH_STENCIL*)
- Beinhaltet **Stencil(Schablone)-Wert** pro Pixel
- Aufgabe:
  - Übernahme von Fragmenten in den FB auf **Regionen** mit bestimmten Eigenschaften der **Stencilwerte einzuschränken** (verwerfen – vgl. Depth Buffer)
- Zu erfüllende Eigenschaft wird durch die **Stencilfunction** eingestellt.
- Art der Stencil-Wert Modifikation festgelegt durch **Stenciloperation**
  - Kann abhängig vom Stencil- & Z-Test unterschiedlich eingestellt werden
  - *glStencilOp* für Backface und Frontface gleich
  - *glStencilOpSeperate* unterschiedlich für Back- & Frontface



Color buffer

Stencil buffer

Nach stencil test

https://learnopengl.com/Advanced-OpenGL/Stencil-testing

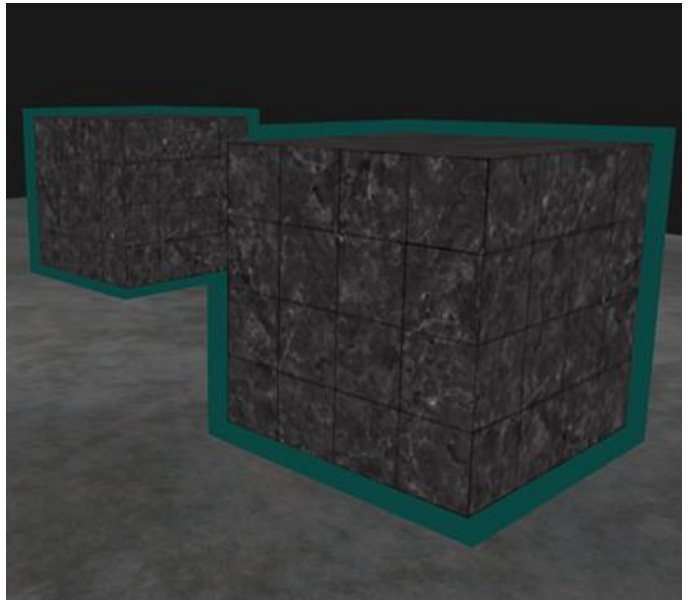
Stencil Buffer

```
// Stencil Testing ist per Default deaktiviert, daher: Aktivierung des Stencil Testings
glEnable(GL_STENCIL_TEST);
…
// Löschen des Stencil buffers der letzten Iteration
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
…
// Setzen der Stencil Maske (Bitmaske), diese wird per AND-Operation mit dem Stencil Wert auf den Buffer
// geschrieben, bei 0xFFwird  jedes Bit auf den Stencil Buffer geschrieben
glStencilMask(0xFF);                    // bei 0x00, jedes Bit wird zu 0 im Stencil Buffer (deaktivieren des Schreibens)
…
// Stencil Test Funktion (nur), um zu kontrollieren, wann der Test besteht
glStencilFunc(GL_EQUAL, 1, 0xFF); // Params: GLenum func, GLint ref, GLuint mask
// hier: wenn immer der Stencil Wert gleich dem Referenz-Wert 1 ist, besteht der Test und wird gezeichnet
..
// Stencil Operation bestimmt wie der Stencil Buffer upgedatet wird
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE); // Params: GLenum sfail, GLenum dpfail, GLenum dppass
// hier: wenn Stencil und Depth Test bestehen ersetzen des Stencil Wertes sonst belassen des Wertes
…
```

Stencil Buffer - Beispiel

Outlining von Objekten mittels Stencil Buffer

- Aktivieren des Stencil Testes
- Setzen der Stencil Funktion auf GL_ALWAYS, setzen der Stencil Operation auf:
  - GL_KEEP, wenn Stencil Test fehlschlägt
  - GL_KEEP, wenn Stencil Test besteht, aber Depth Test fehlschlägt
  - GL_REPLACE, wenn Stencil Test & Depth Test bestehen
- Rendern der Objekte
- Deaktivieren des Schreibens auf den Stencil Buffer (0x00) & Depth Test
- Setzen der Stencil Funktion auf die Bedingung: gespeicherte Wert ungleich 1 mit einer Maske für alle Bits
- Skalieren der Objekte (größer)
- Zeichnen der Objekte mit einem Shader, der die Fragmente in einer einzelne Farbe Shadet
- Zurücksetzen der Stencil und Depth Einstellung

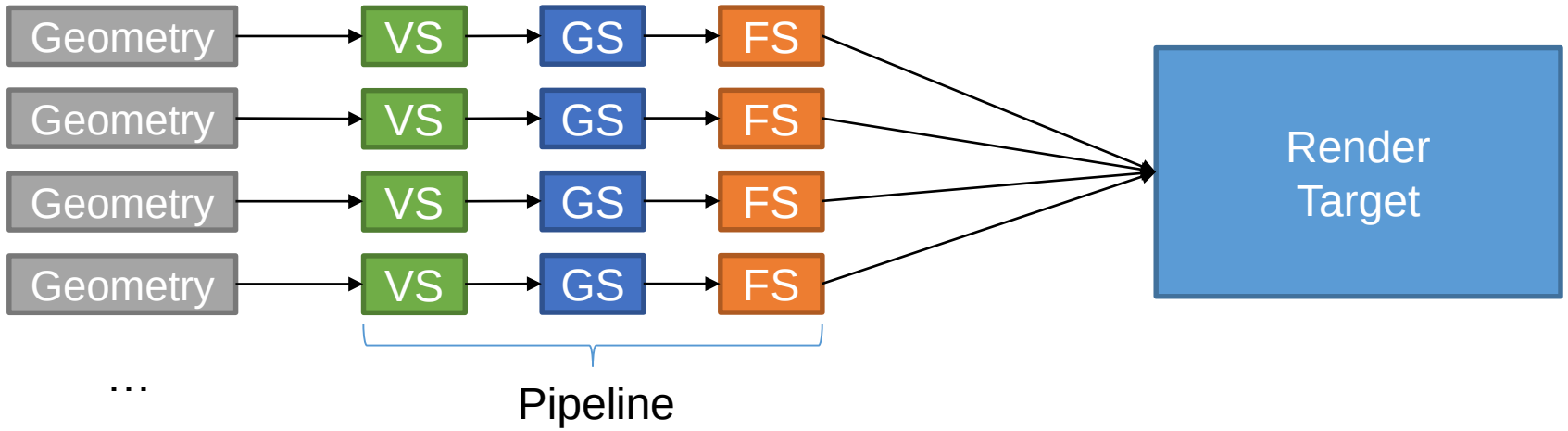


erstellen eines kleines farbigen
Randes für jedes Objekt

https://learnopengl.com/Advanced-OpenGL/Stencil-testing

Forward Rendering

- Standard (Out-of-the-box) Rendering Technik
  - Zur Verfügung stellen von Geometrie
  - Projektion -> Vertices -> Fragments -> Finales Rendering
- **Lineare Technik**
- Fixed-Function Pipeline in OpenGL
  - Max. 8 Lichtquellen (baked lights)
- Kostet viel Performance (insb. bei vielen Lichtquellen)



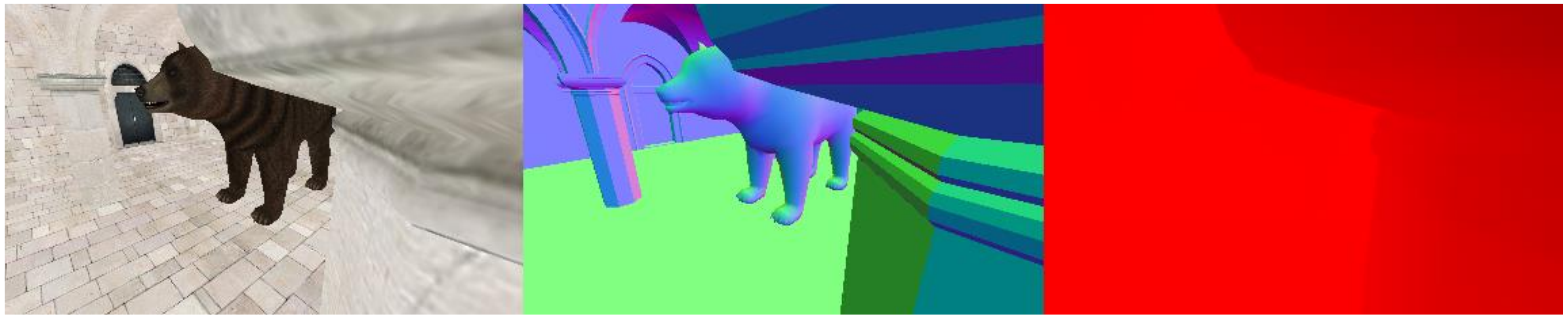
Deferred Shading

- **Aufgeschobene** (deferred) rechenintensive **Lichtberechnung** auf späteren Zeitpunkt
  - 2 Render Durchläufe
    - **geometry pass** – speichert alle geometrischen Informationen in einer Kollektion aus Texturen (**G-Buffer**)
    - **lighting pass** – komplexe Lichtberechnung
      - Lichtberechnung pro Fragment (auf dem finalen Bild), da der **Occlusiontest** bereits durchgeführt wurde
- Dadurch Möglichkeit vieler Lichtquellen in der komplexen Szene
  - Bei akzeptabler Framerate

Deferred Shading

Geometry Pass

- Rendern der **sichtbaren Geometrien** in den **G-Buffer**
  - Speicherbuffer (FBO) für verschiedene Texturen mit geometrischen Informationen
    - [Position, Diffuse Farbe, Normalen, Spekularer Anteil, ….]
- Nutzung von MRT (Multi-Render-Targets)
  - Rendern der Szene auf verschiedene Targets in einem Durchlauf



Deferred Shading

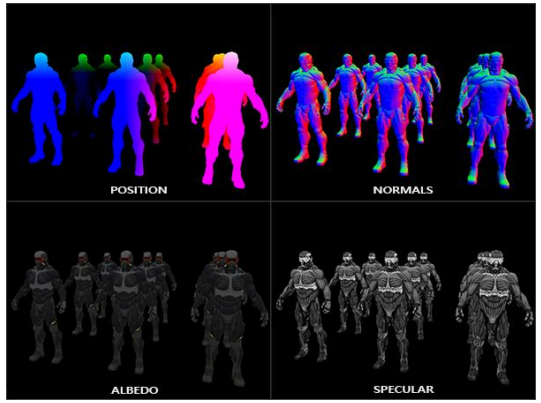
Lighting Pass

- Beleuchtungsberechnung abhängig von **geometrischen Daten** auf dem **G-Buffer** (genauer: Texturen verlinkt im FBO des G-Buffers)
- Beleuchtungsberechnung bleibt gleich
- Zur Steuerung des Prozesse:
  - Uniform Variablen

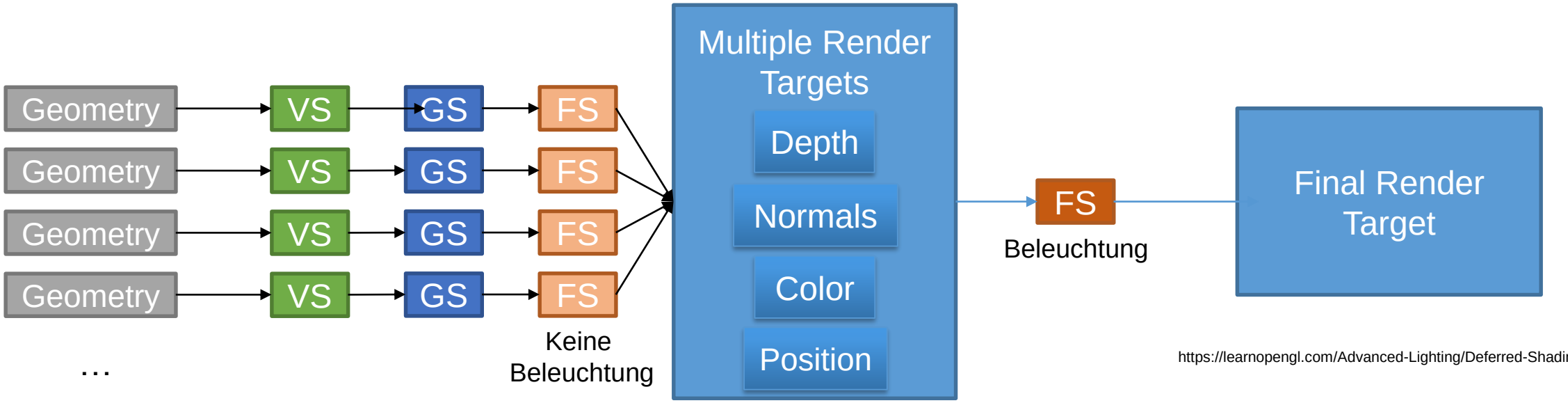
- Naiver Ansatz
  - Rendern eines Quads in Screen Größe
    - (Nutzen der Textur-Koordinaten für Sampling aus dem G-Buffer)

Deferred Shading

GEOMETRY



LIGHTING



https://learnopengl.com/Advanced-Lighting/Deferred-Shading

Deferred Shading

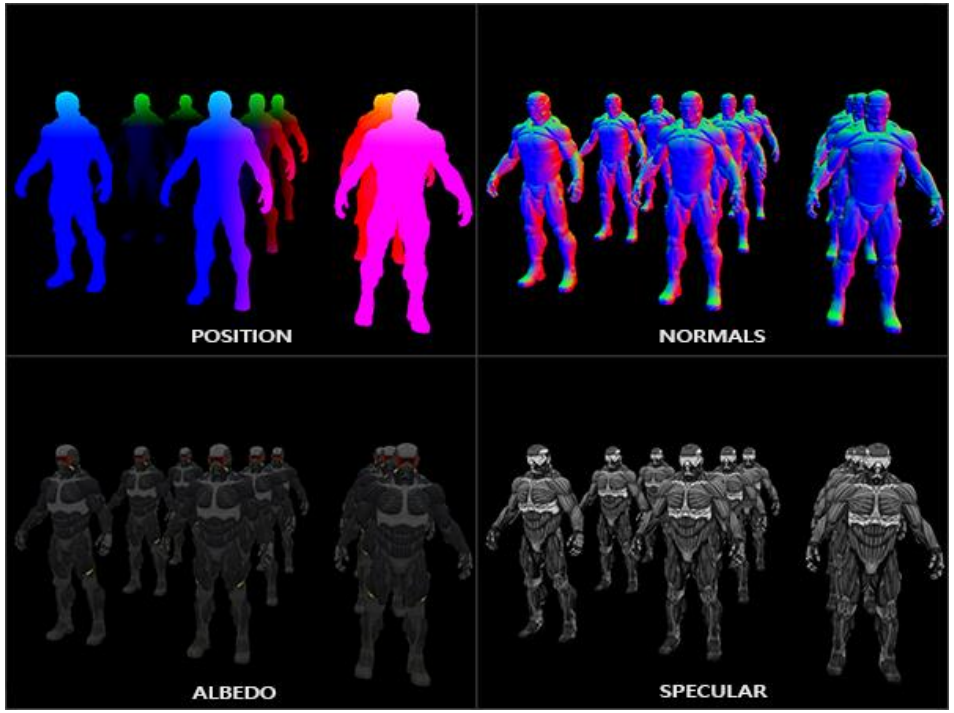
G-Buffer

Color-Attachments

- Position (high-precision 16/32 Bit [RGB16F])
- Normals (high-precision 16/32 Bit [RGB16F]))
- Albedo & Specular (kann low-precision [RGBA]) sein*)
  - Albedo in Texture.rgb
  - Specular in Texture.a
- Depth-Attachment
- Stencil-Attachment

  [GL_DEPTH32_STENCIL8]

* Wenn kein HDR genutzt wird sonst [RGBA16F])



Deferred Shading

Nachteile*:

- G-Buffer muss eine große **Menge** an **Daten** speichern (im Color-Attachment)
- **Blending** wird nicht mehr unterstützt
  - * Nutzung Depth-Peeling oder Forward + Deferred Shading
- **Multisampling** Anti-Aliasing (MSAA) wird nicht mehr unterstützt

* Es gibt verschiedene Workarounds

Deferred Shading

Kombination Deferred Rendering & Forward Rendering

- 2 Render Passes
  - Forward Rendering für Blending & andere spezielle Shader Effekte
- **Achtung**: der Depth Buffer des G-Buffer muss kopiert werden in das FBO, welches für das Forward Rendering genutzt wird

```
// vom gBuffer (FBO) lesen
glBindFramebuffer(GL_READ_FRAMEBUFFER, gBuffer);
// auf den default Frame Buffer schreiben
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glBlitFramebuffer( 0, 0, SCR_WIDTH, SCR_HEIGHT, 0, 0, SCR_WIDTH, SCR_HEIGHT, GL_DEPTH_BUFFER_BIT,
GL_NEAREST );
// default FBO binden als Lese- und Schreibe-Buffer
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// render Szene mit den Werten des Tiefenbuffers, der bei dem Rendern in den gBuffer gesetzt wurde
```

Deferred Shading

Light Volumes

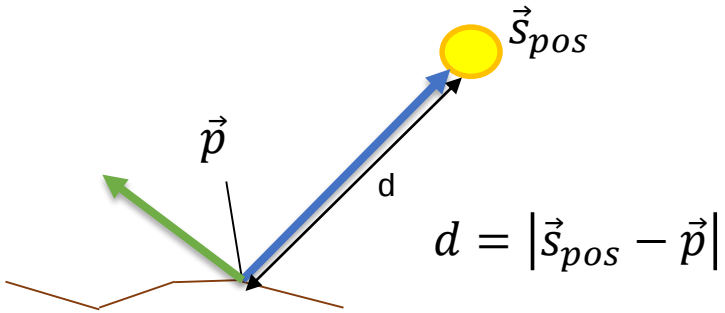
- **Beschränkung** des **Render-Bereich**s einer Lichtquelle auf den Bereich, den das Licht erreichen kann
  - **Abschwächungskoeffizienten** {konstant, linear, quadratisch}
- Dadurch: Berechnung des Fragments nur, wenn es sich im Bereich der Lichtquelle befindet
- Lichtquellen
  - Berechnung des maximalen Kugel-Radius einer Punkt-Lichtquelle
  - Berechnung des maximalen Kegels-Volumens einer Spot-Lichtquelle

Deferred Shading

Light Volumes

Berechnen der Abschwächungsfunktion (BRDF), wenn F_{light} nahe an 0.0 ist (komplett **dunkel**)

$$F_{light} = \frac{I}{s_c + s_l * d + s_q * d^2}$$



$$d = |\vec{s}_{pos} - \vec{p}|$$

$s_c = konstanter\ Term$
$s_l = linearer\ Term$
$s_q = quadratischer\ Term$
$I = Intensität$

$\vec{s}_{pos} = Position\ der\ Lichtquelle$
$\vec{p} = Fragmentposition\ bzw. Reflexionspunkt$
$d = Entfernung\ Fragment\ zur\ Lichtquelle$

$F_{light} = Intensität\ nach\ Abschwächung$

Deferred Shading

Light Volumes

$$F_{light} = \frac{I}{s_c + s_l * d + s_q * d^2}$$

Threshold nahe an 0.0, bei Default 8-bit FBO z.B. 5/256 (256=Anzahl Intensitäten pro Komponenten)

$$threshold = \frac{I_{max}}{Attentuation}$$

$$Attentuation = \frac{I_{max}}{threshold}$$

$$s_c + s_l * d + s_q * d^2 = \frac{I_{max}}{threshold}$$

Quadratische Gleichung:
$ax^2 + bx + c = 0$

$$s_q * d^2 + s_l * d + s_c - \frac{I_{max}}{threshold} = 0$$

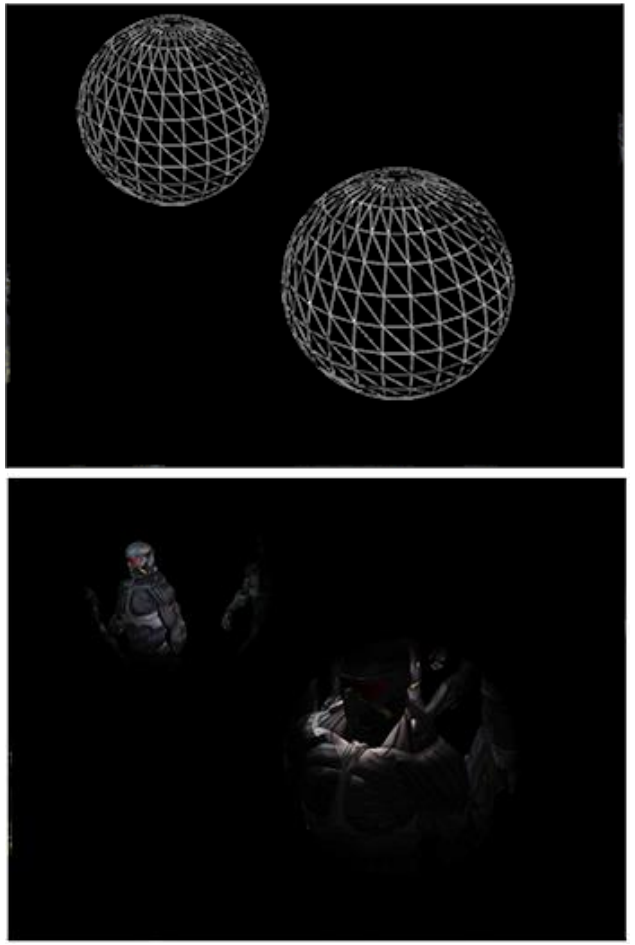
$$radius = \frac{-s_l + \sqrt{{s_l}^2 - 4 * s_q * (s_c - \frac{I_{max}}{threshold})}}{2 * s_q}$$

Deferred Shading

Light Volumes

Punktlichtquelle

- Rendern einer **Einheits-Kugel**
  - Zentrum bei der Position der Lichtquelle
  - Skalierung entsprechend des berechneten Radius (Abschwächung)
- Nutzung (quasi) desselben (Fragment-) Shaders, mit dem die Texturen auf ein Quad gerendert werden würden
  - die gerenderte Kugel erzeugt dieselben Fragment-Shader Aufrufe, die durch die Lichtquelle beeinflusst werden
- Dadurch werden nur die relevanten Pixel gerendert und alle weiteren werden übersprungen



https://learnopengl.com/Advanced-Lighting/Deferred-Shading

Deferred Shading

Light Volumes

- Für jede Lichtquelle in der Szene werden die resultierenden Fragmente **additiv übereinander geblendet** (res = 1 * src + 1 * dst)
- Reduzierung der Berechnungen
  - Von Anzahl_Objekte * Anzahl_Lichtquellen
  - Auf Anzahl_Objekte + Anzahl_Lichtquellen
- Aktivierung von Face-Culling (sonst wird jedes Licht 2-fach gerendert)

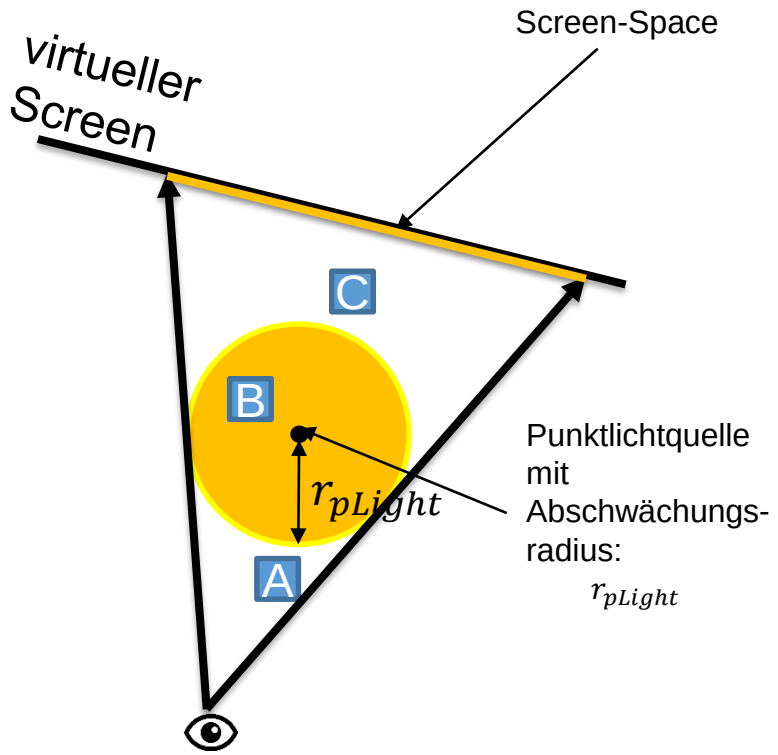
```
…
glDisable(GL_DEPTH_TEST);
glEnable(GL_BLEND);
glBlendEquation(GL_FUNC_ADD);
glBlendFunc(GL_ONE, GL_ONE);
….

for (unsigned int i = 0; i < POINT_LIGHTS_COUNT; i++) {
        // Set Uniforms & Bind G-Buffer Textures
        RenderUnitSphere();
}
```

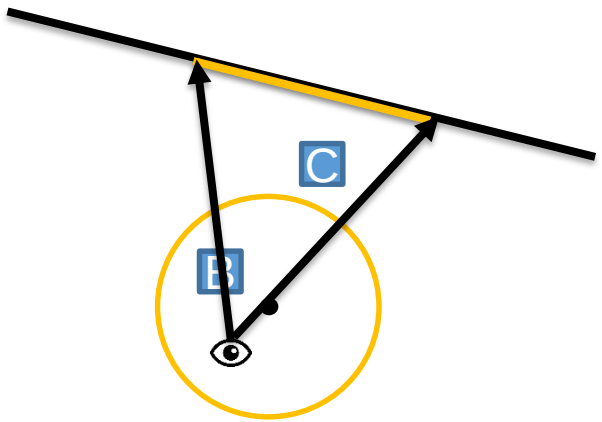
Deferred Shading

Light Volumes

ABER:

- Bei Back-Faceculling, sobald **Kamera in Lichtquelle** ist, wird die Kugel nicht mehr gerendert
- Bounding Kugel „bindet“ das Licht nicht und manchmal werden **Objekte außerhalb** auch gerendert, da die Kugel diese in Screen-Space beinhaltet

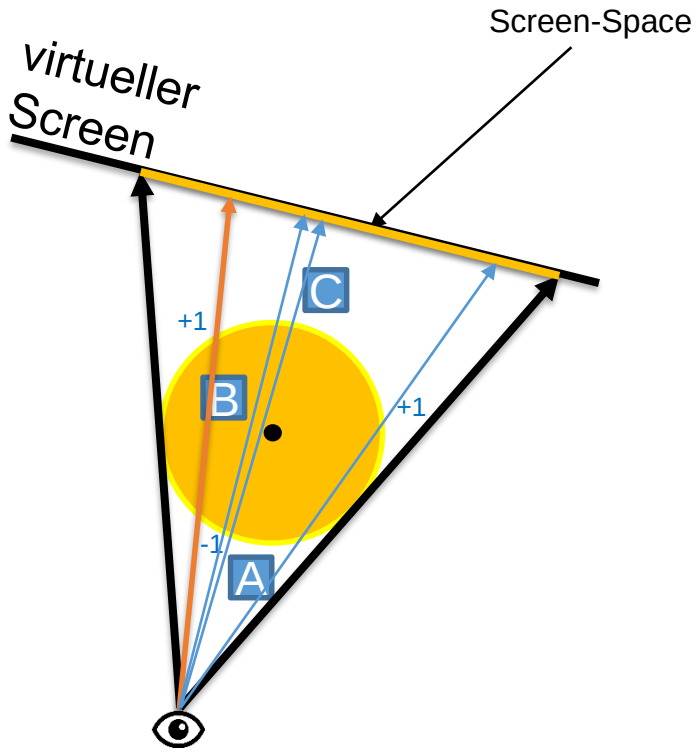




Deferred Shading

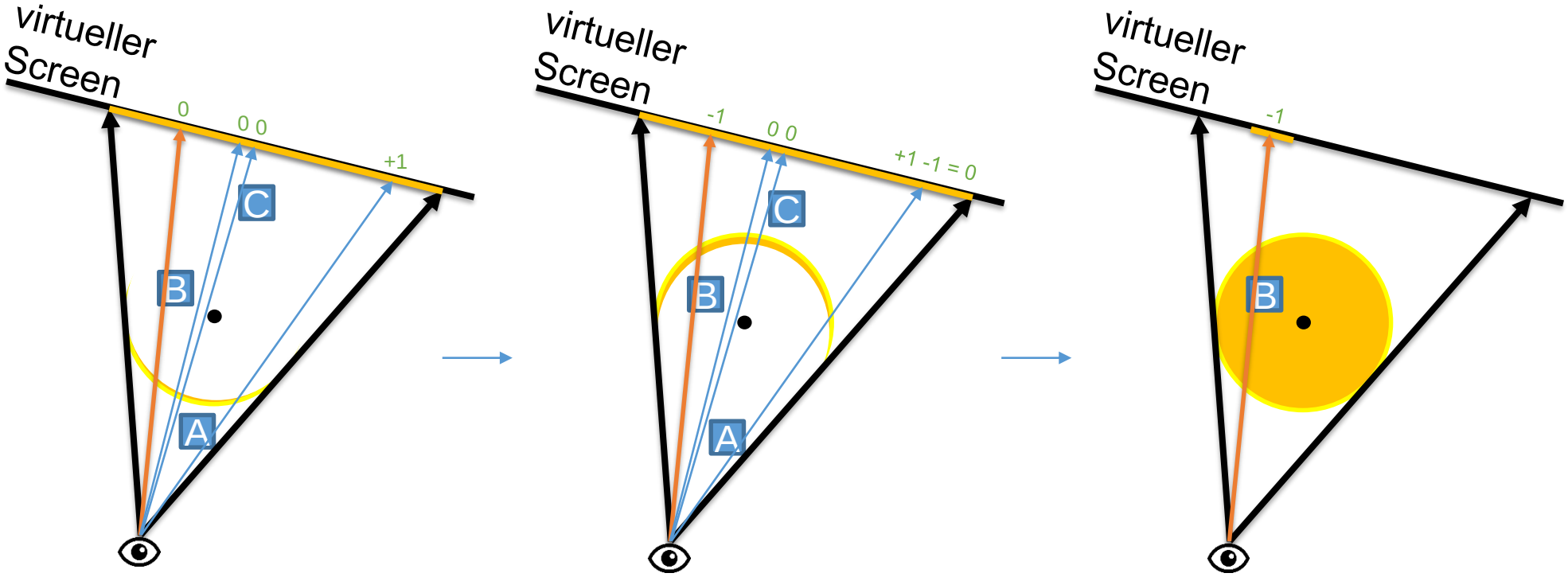
Light Volumes (Problembehandlung)

- Nutzung des **Stencil-Buffers** zur Limitierung der Berechnung der Fragmente, die von *Objekt B* eingenommen werden
- Rendere Szene in G-Buffer
- Deaktivierung des Schreibens auf den Depth-Buffer (*glDepthMask(GL_FALSE)*)
- Stencil Pass
  - Aktivierung Stencil Test, Stencil Function auf GL_ALWAYS (mit ref & maske = 0) Clear Stencil Buffer & setzen des StencilOperators auf:
    - GL_BACK → wenn Stencil T. besteht & Depth fehlschlägt: Stencil-Wert um 1 erhöhen
    - GL_FRONT → wenn Stencil T. besteht & Depth fehlschlägt: Stencil-Wert um 1 verringern
    - ALLE anderen Fälle: Stencil Wert behalten
  - Deaktivierung Face_Culling
  - Rendern der Lichtquellen, ohne auf ein Color-Attachment zu schreiben (NULLShader)
- Beim eigentlichen Rendern der Lichtvolumen auf das Color-Attachment
  - Stencil Funktion auf GL_NOT_EQUAL 0 mit der Maske 0xFF (alle Bits) setzen
  - Enable Front_Face Culling



Deferred Shading

Rendern der Kugel Geometrien mit dem NULLShader, die Tiefenwerte der Szene liegen im gebundenen G-Buf
Schreiben auf den Depth-Buffer ist deaktiviert. (Grün: Werte im Stencil Buffer für das Fragment)



1. Render der Kugel Geometrie Backface Culling

2. Render der Kugel Geometrie Frontface Culling

3. Nur Teile der Szene Rendern (mit PointLight-Shader), in denen der Stencil Buffer nicht 0 ist (sich ein Objekt ganz oder z.T. in der Kugel befindet)

Deferred Shading

Pseudo-Algorithmus

- Clear Buffers (für alle FBOs)
- Geometry-Pass
  - Bind G-Buffer für Geometry-Pass & entsprechenden Shader
  - Depth-Test aktivieren & *schreiben* auf Depth-Buffer aktivieren
  - Rendern der geometrischen Daten der Szene per MRT in den G-Buffer (setzen aller erforderlichen Uniforms)
  - *Schreiben* auf Depth Buffer deaktivieren
- Für alle Punktlichtquellen:
  - Stencil-Pass
    - Null-Shader binden & aktivieren (keinen Color-Buffer binden!)
    - Löschen der gespeicherten Werte im Stencil Buffer
    - Aktiviere Stencil Test & *glStencilFunc* auf pass GL_ALWAYS setzen (ref&mask =0) & *glStencilOpSeparate* auf:
      - GL_BACK → wenn Stencil T. besteht & Depth T. fehlschlägt: Stencil-Wert um 1 erhöhen
      - GL_FRONT → wenn Stencil T. besteht & Depth T. fehlschlägt: Stencil-Wert um 1 verringern
      - ALLE anderen Fälle: Stencil Wert behalten
    - Deaktiviere Face_Culling
    - Rendern der Punkt Lichtquellen (vorher setzen der entsprechenden Attribute)

Deferred Shading

Pseudo-Algorithmus (weiter)

- Für alle Punktlichtquellen:
  - …
  - Point-Pass
    - Bind Point-Light-Shader & aktivieren
    - Setze *glStencilFunc* auf GL_NOTEQUAL 0 mit der Maske für alle Bits
    - Deaktivieren des Depth-Test
    - Aktivieren des Blending unter Nutzung einer echten 1 zu 1 Addition
    - Aktivieren des Front-Face Cullings
    - Rendern der Punkt Lichtquellen (vorher setzen der entsprechenden Attribute, u.a. G-Buffer Texturen)
    - Face-Culling auf GL_BACK setzen
    - Deaktivieren von Blending
- Deaktiviere Stencil Test

Deferred Shading

Pseudo-Algorithmus (weiter)

…

- Directional Light-Pass
  - Aktivieren des Blending unter Nutzung einer echten 1 zu 1 Addition
  - Binden und aktivieren des Directional Light Shaders
  - Setzen von Uniform Variablen (u.a. G-Buffer Texturen)
  - Rendern eines Viewport Füllenden Quads
  - Deaktivieren von Blending
- FinalPass
  - Weiteres arbeiten auf dem Framebuffer Color-Attachment, auf das per additivem Blending gerendert wurde
- SwapBuffers

Deferred Shading

Normalmapping

- G-Buffer rendert in die Normal-Textur die ausgelesenen Normalen aus der Normal-Map

Paralaxmapping

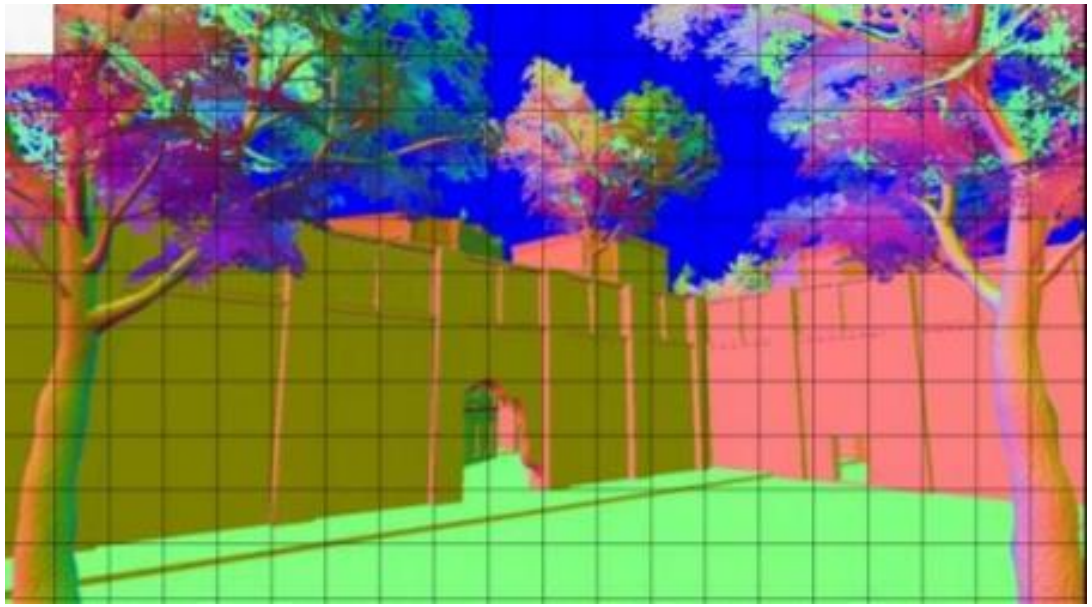
- (ebenfalls) in der Geometrie-Render-pass
  - Versetzen der Textur-Koordinate vor dem Sampling aus den Objekt-Texturen (Diffuse, Spec., Normal)

Beleuchtungsberechnung in der Lighting-Stage muss nicht angepasst werden!

Deferred Shading

Tile-based Deferred Shading

- (Normales *Deferred Shading*)
  - Trotz Culling Technik etc. immer noch viele G-Buffer Samples (viel Speicher)
- Teilt das Viewport-Frustum in viele kleinere Frustums
  - Diese sind erweitert entlang der Z-Achse im View-Space
- Führt das Licht Culling in demselben Abschnitt aus wie die Shader Berechnung
  - Entsprechend auf den kleineren Frustums
- Verhindert Übermalen & häufiges Sampling des G-Buffers
- idR Benötigt einen Compute-Shader



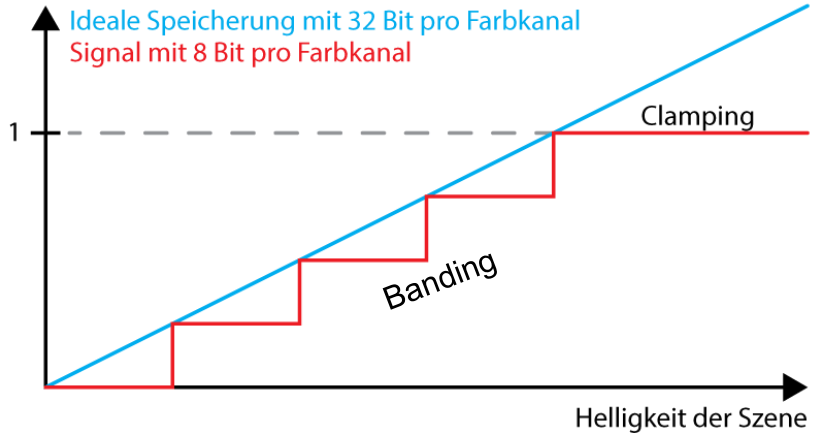
https://www.slideshare.net/DICEStudio/spubased-deferred-shading-in-battlefield-3-for-playstation-3

High Dynamic Range (HDR)

Typischerweise nutzt der Standard-Framebuffer 8 Bit pro Farbkanal. Dabei werden die Farbwerte von 0 bis 1 pro Farbkanal mit 8 Bit repräsentiert. (OpenGL-Texturformal *GL_RGB8 / GL_RGB*)

Entstehung zweier Probleme:

- **Banding**: Treppeneffekt, da nur 256 unterschiedliche Intensitäten pro Farbkanal zur Verfügung stehen
- **Clamping**: Werte oberhalb von 1 können nicht abgebildet werden (z.B. Strahlungsdichte) Clamped im Intervall [0,1]



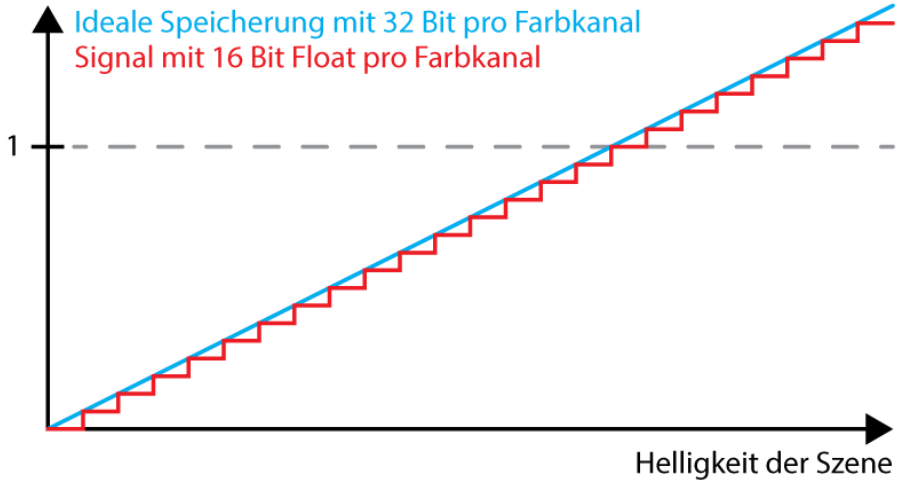
High Dynamic Range (HDR)

- Temporäres Erlauben von Werten über 1.0 Helligkeit (große Breite an Farben)
  - Dadurch Details in beiden Helligkeitsbereichen
    - Sehr helle Bereiche
    - Sehr dunkle Bereiche
  - Realistischeres Licht
- Rücktransformation im finalen Schritt zu LDR
  - Da Monitor die Helligkeit auf das besagte Intervall beschränkt
  - Tone Mapping Algorithmen (vers. Kurven oder Gleichungen)
- Fotographie
  - Erstellen mehrerer Bilder mit unterschiedlichem Exposure-Level (großer Bereich an Farbwerten)

High Dynamic Range (HDR)

Für HDR Framebuffer werden meistens 16 Bit (mög. sogar bis zu 32bit) Gleitkommawerte* pro Farbkanal verwendet.
(OpenGL-Texturformal ***GL_RGB16F*** *; GL_RGB16* ist kein Gleitkommaformat & bildet wie GL_RGB8 nur das Intervall 0 bis 1 ab.)

Mit diesem Texturformat kann das Originalsignal genauer und auch oberhalb von 1 repräsentiert werden.



*nur Floatingpoint-Formate können genutzt werden aufgrund ihres erweiterten Value-Umfanges (ausserhalb von [0,1])

Tone Mapping

Rücktransformation für Bildschirm (8 Bit pro Farbkanal)

- Oft Nutzung von **Exposure Parametern,** um besonders helle Bereiche oder dunkle Bereiche hervor zu heben
  - Menschliches Auge
    - Bei wenig Licht passt sich das Auge an, sodass dunklere Bereiche besser wahrgenommen werden können (andersherum bei hellen Bereichen)
- Ohne zu viel Detail zu verlieren
- Oft in Verbindung mit einer spezifischen Farbbalance

Tone Mapping

Reinhard Tone Mapping

- Bei diesem Tone Mapping Operator könne **keine weiteren Einstellungen** vorgenommen werden.

$$C_{LDR} = \frac{C_{HDR}}{C_{HDR} + 1}$$

Exposure Tone Mapping

- Dieser Tone Mapping Operator ermöglicht das Einstellen einer (*nicht physikalisch korrekten*) **Belichtungszeit** p:

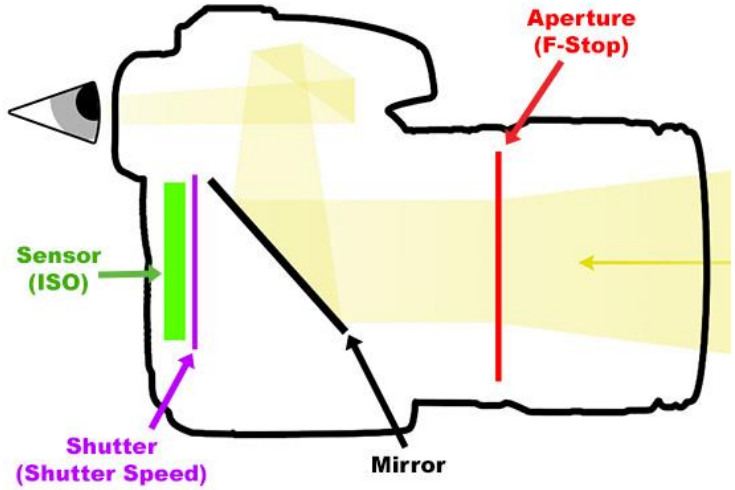
$$C_{LDR} = 1 - e^{-C_{HDR} * p}$$

Tone Mapping

In einer realen Kamera bestimmt die Blende (engl. **Aperture**), wie viel Licht (Breite der Strahlenbündel) bei der Aufnahme des Bildes auf den Sensor fällt.

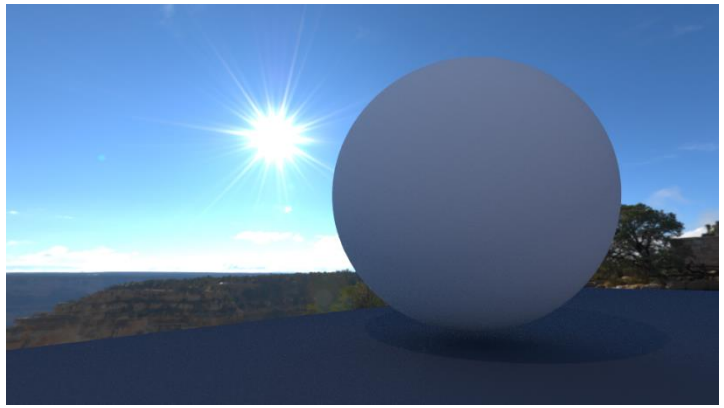
Damit wird die Tiefenschärfe (engl. Depth of Field) bestimmt.

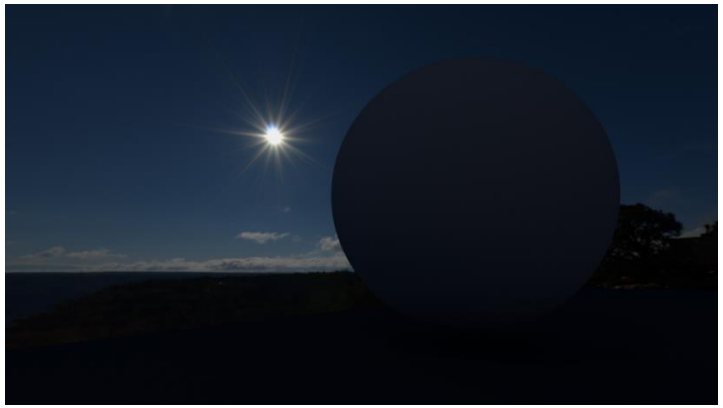
Bei der Aufnahme eines Bildes öffnet sich der Verschluss für einen Moment, der Belichtungszeit (engl. exposure time) genannt wird, damit die Strahlenbündel auf den Sensor fallen können. Je länger der Verschluss offen ist, desto heller wird das Bild. Außerdem entsteht auf diese Weise Bewegungsunschärfe (engl. Motion blur).

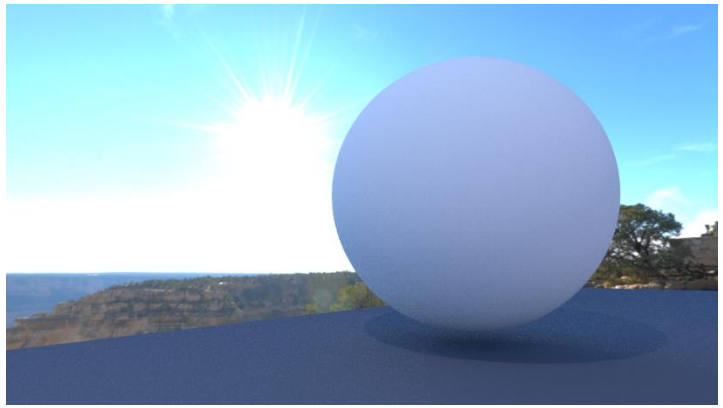


https://www.nickcarverphotography.com/blog/tag/exposure-triangle/

Tone Mapping







Mittlere Belichtungszeit

Unterbelichtet

Überbelichtet

https://www.promicabana.de/dumm-und-duemmer-sequel-jim-carrey/

Gamma-Korrektur

Mit der Gamma-Korrektur wird die **Nutzung der Bits für die menschliche Wahrnehmung optimiert**.

Da menschliche Helligkeitswahrnehmung ungefähr einer Potenzfunktion entspricht, bietet es sich an, die Pixelwerte entsprechend zu verteilen, damit keine Bits in Bereichen verschwendet werden, in denen Helligkeitsunterschiede sowieso schlechter wahrgenommen werden können.

Außerdem haben **CRT-Monitore eine nichtlineare Reaktion** auf die angelegte Spannung, was dazu führt, dass sich die abgegebene Helligkeit nicht verdoppelt, wenn die doppelte Spannung angelegt wird.

Typischerweise wird hierbei ein **Gamma-Wert von 2.2** angenommen und auch modernere Monitore sind entsprechend kalibriert. Einige Spiele bieten jedoch Möglichkeit zur Änderung dieses Wertes an, um das Bild besser anzupassen.

Bildbearbeitungsprogramme führen beim Exportieren von Bildern (Texturen) eine **Gamma-Korrektur** durch. Dadurch werden die Helligkeitswerte des Bildes entsprechend einer Exponentialfunktion verändert.

Gamma-Korrektur

- Beim **Einlesen** dieser Textur-Werte müssen diese **linearisiert** werden, da die folgenden Berechnungen sonst falsche Ergebnisse liefern. (Das Bild wird zu dunkel und die Farbsättigung zu stark.)
- Bei der **Ausgabe** auf den Standard Framebuffer muss erneut eine **Gamma-Korrektur** stattfinden.
- **HDR Texturen müssen nicht linearisiert werden**. Für diese wird keine Gamma-Korrektur ausgeführt.

Gamma-Korrektur

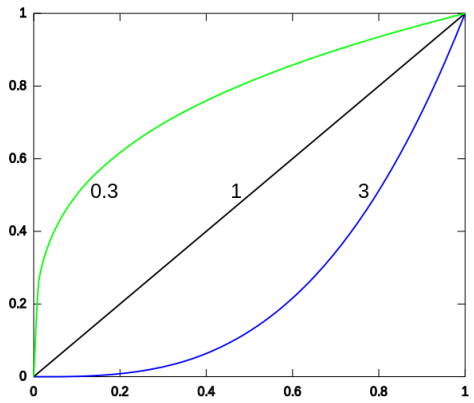
Beim Importieren von Texturen muss die **inverse Gamma-Korrektur** stattfinden:

$$C_{linear} = C_{gamma}{}^{\gamma}$$

(c_{gamma} kommt aus der Textur)

Vor dem Schreiben der Werte auf den Framebuffer muss folgende Gamma-Korrektur stattfinden:

$$C_{gamma} = C_{linear}{}^{\frac{1}{\gamma}}$$



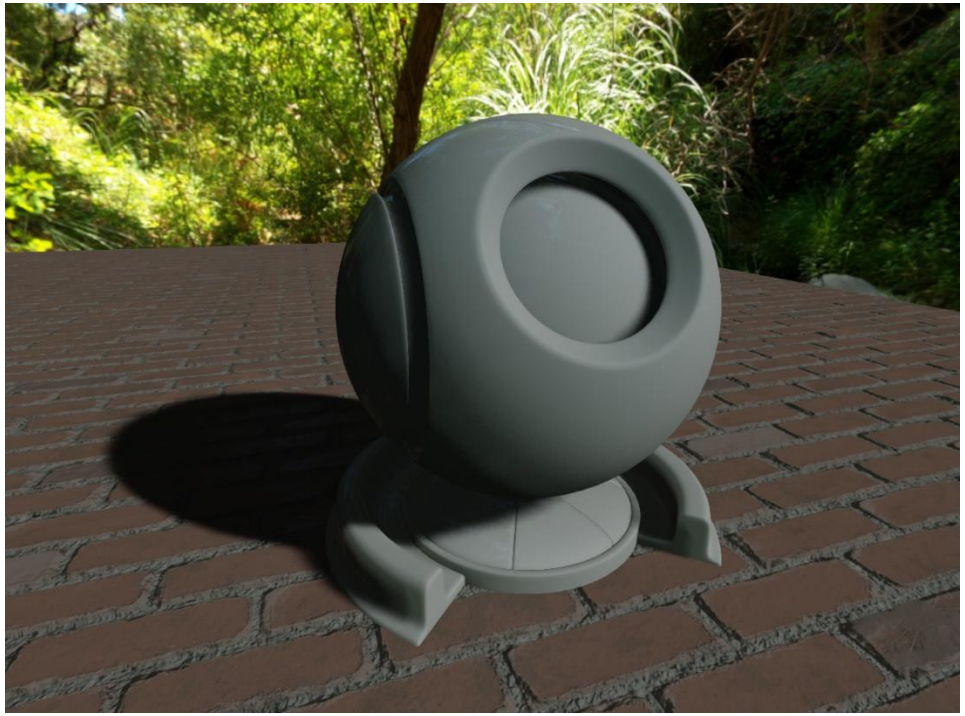
http://www.wikiwand.com/de/Gammakorrektur

Gamma-Korrektur

In OpenGL kann eine automatische Gamma-Korrektur aktiviert und das Farbprofil für betroffene Texturen auf sRGB umgestellt werden.

- *glEnable(GL_FRAMEBUFFER_SRGB)* // aktiviert die Konvertierung generell
- Für betroffene Texturen muss das Format *GL_SRGB8* (oder *GL_SRGB8_ALPHA8*) verwendet werden

Gamma-Korrektur



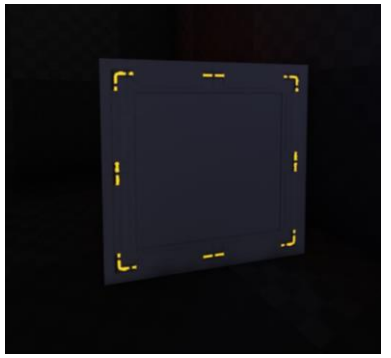
Keine Gamma-Korrektur

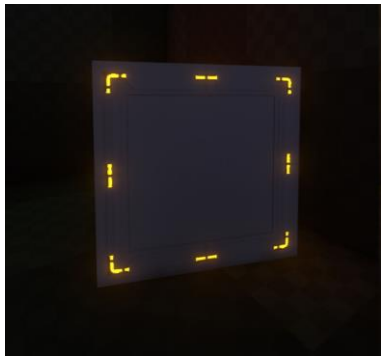
lineare Gamma-Korrektur

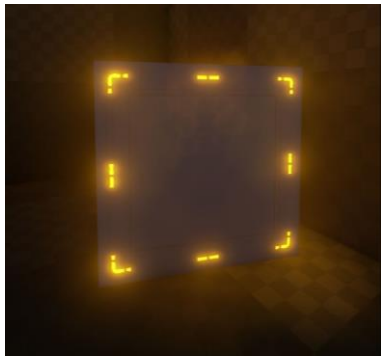
Bloom

Regionen „Glühen“ lassen

- Gibt Illusion, dass Flächen sehr hell sind
  - Verbessert Licht-Effekt in der Szene
- Bester Effekt mit HDR Rendering
  - Da Monitor limitiert ist
- Es ist ein Post Processing-Effekt



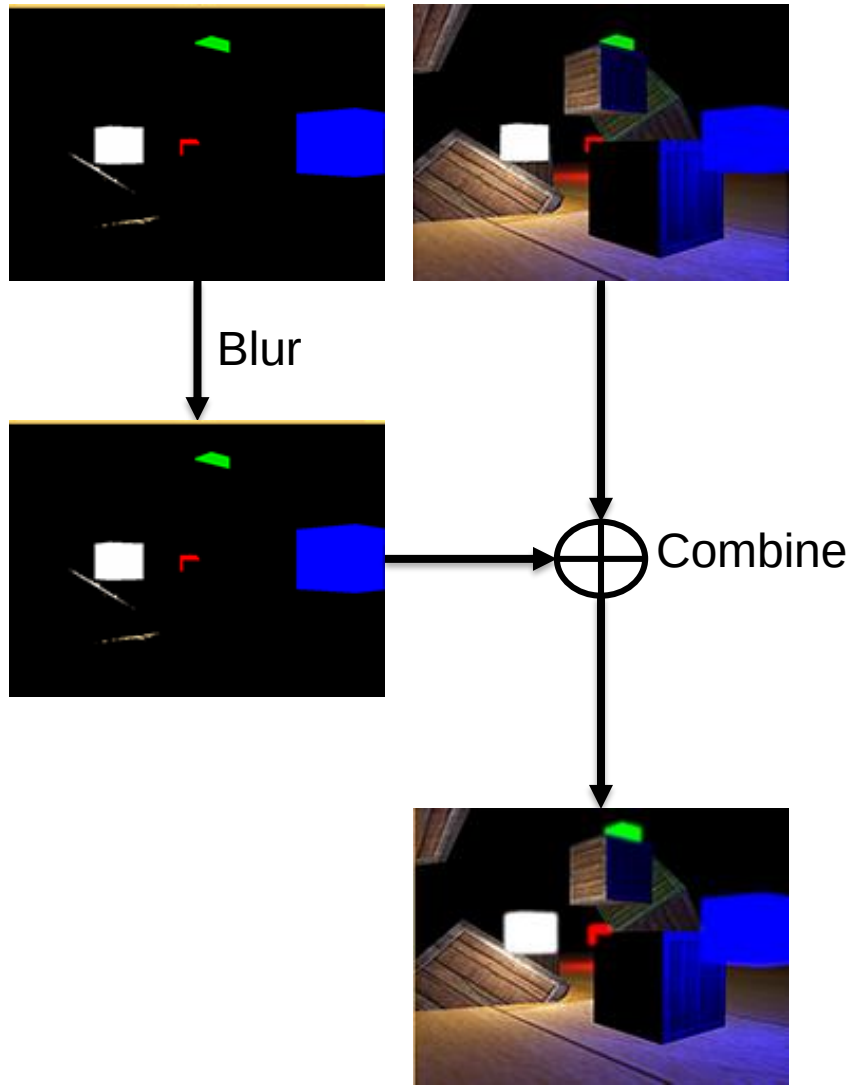




https://i.stack.imgur.com/09CkX.jpg

Bloom - Algorithmus

- Szene wie gewohnt rendern (MRT)
  - Extrahieren der Szene als HDR Colorbuffer
  - Extrahieren eines Bildes mit lediglich den sehr hellen Bereichen des Bildes
- Anwenden eines Blur-Filters auf das Bild (ggf. mehrfach) [vgl. Blur in Post-Processing]
- Addieren des gefilterten Bildes mit den hellen Bereichen auf die normale Szene (Additives Blending, addieren der Textur im FS)
- (im finalen Prozess: Transformation HDR→LDR Tone Mapping)



https://learnopengl.com/Advanced-Lighting/Bloom

Post-Processing

- Manipulierung der Texturdaten
- Vor finalem Schreiben auf den default FBO Color-Buffer

Inversion

Color = I_{max} – Color

Schwarz-Weiß

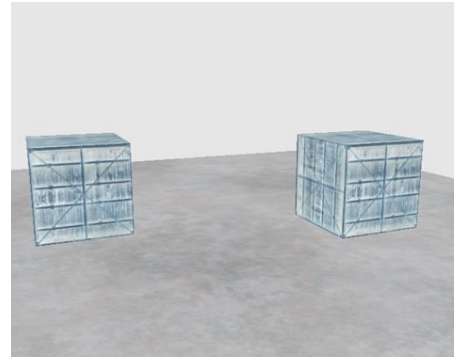
float *average = (0.2126*Color.r*
*+ 0.7152*Color.g*
*+ 0.0722 * Color.b);*

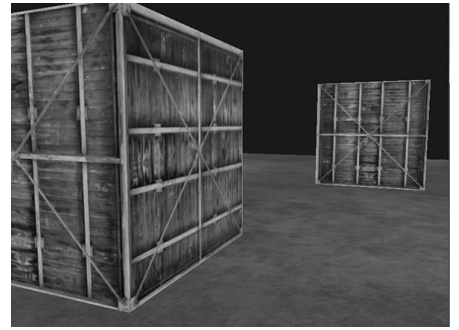
Color = (average, average, average)

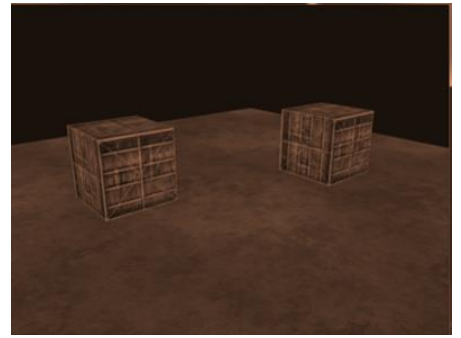
Farbton Anpassung

colorBalance = (1, 0,5, 0,5); //prozentualer Anteil

*Color = (colorBalance.r*color.r, colorBalance.g*color.g, colorBalance.b*color.b)*







https://learnopengl.com/Advanced-OpenGL/Framebuffers

Post-Processing

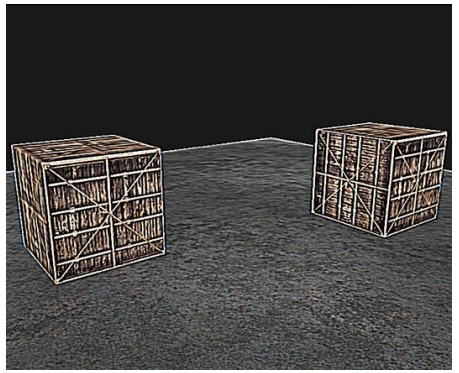
Kernel Effekte (wdh. BBA)

- Faltungsmatrizen
- Matrix, die umliegende Pixel entsprechend der Kernel-Werte multipliziert und addiert
  - Dadurch werden umliegende Pixel mit einbezogen

Post-Processing – Kernel Effekte

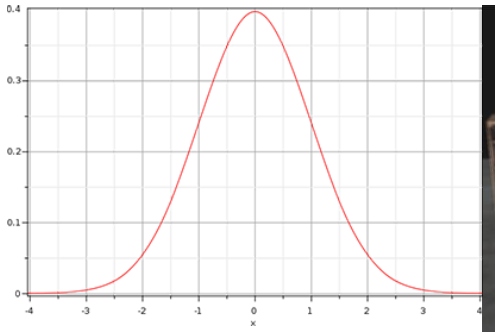
Scharfzeichnungsfilter

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$



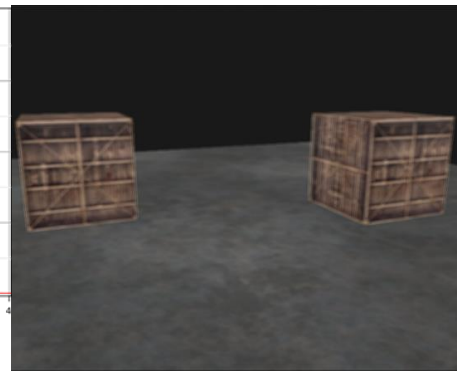
Blur

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} /16$$



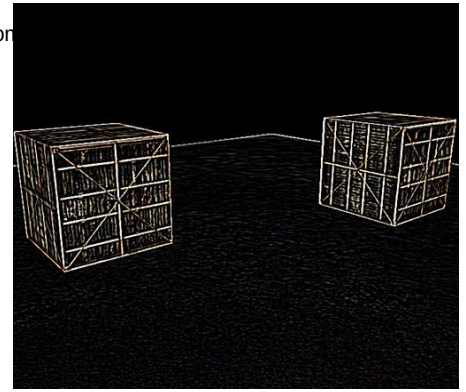
Gaussian Blur

https://learnopengl.com/Advanced-Lighting/Bloom



Edge detection

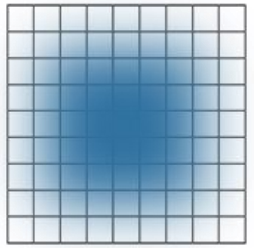
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$



Es gibt viele weitere Filter- & Post-Processing-Effekte

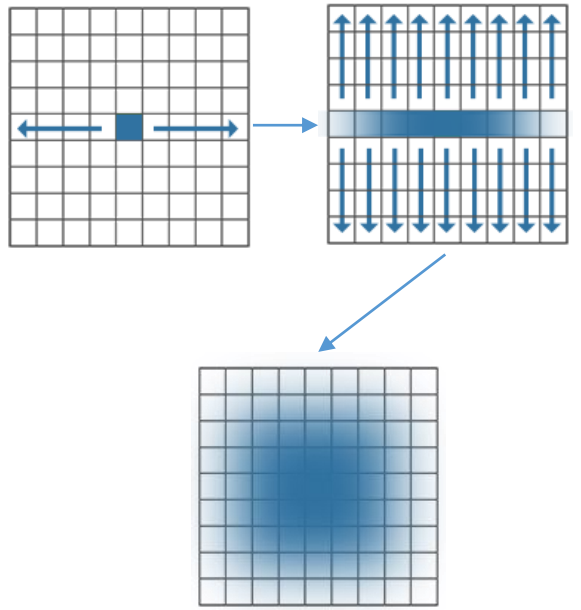
https://learnopengl.com/Advanced-OpenGL/Framebuffers

Post-Processing – Kernel Effekte



Blur – (Two-Pass Gaussian Blur)

- Bisher: Berechnen des **Blurs** auf Basis eines 2D Kernels
  - Ggf. extrem performancebeanspruchend (Sampling Texturen!)
- Jetzt: Aufteilung der 2D-Filterung auf **2x 1D-Filterung**
  - Horizontale Filterung
  - Vertikale Filterung
- **Stärkere Filterung, je öfter wiederholt (Ping-Pong Buffer)**
  - Bei 2-Pass Gaussian Blur, mind. 2x



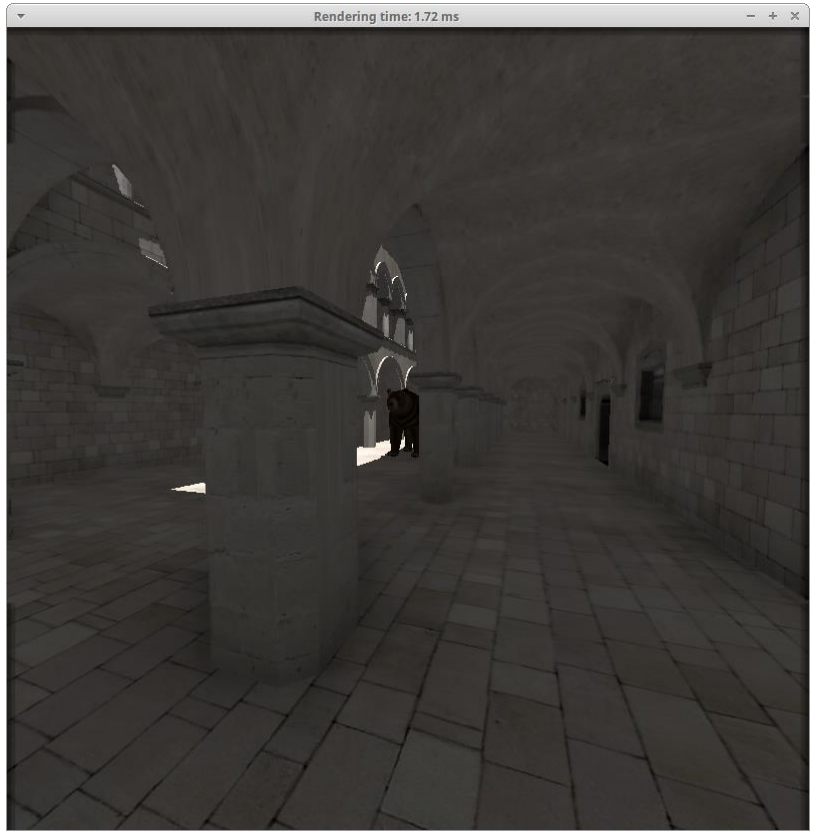
https://learnopengl.com/Advanced-Lighting/Bloom

Resultat:

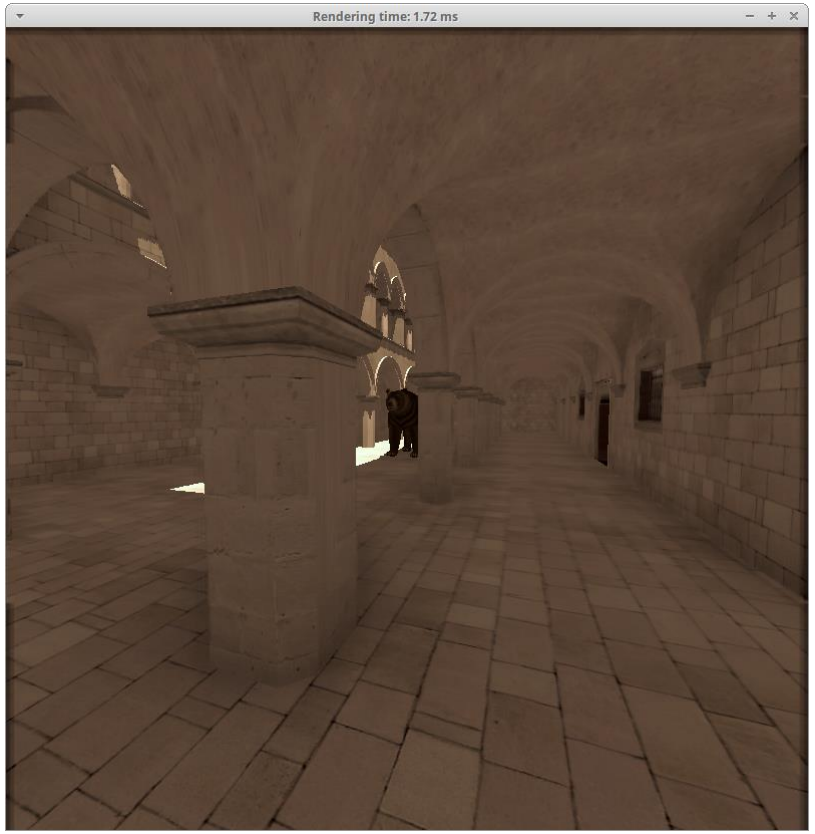
Anzahl der Texture Samples pro Filterung:

Kernel-Width*Kernel-Width → Kernel-Width + Kernel-Width

Color Grading (Color Correction)



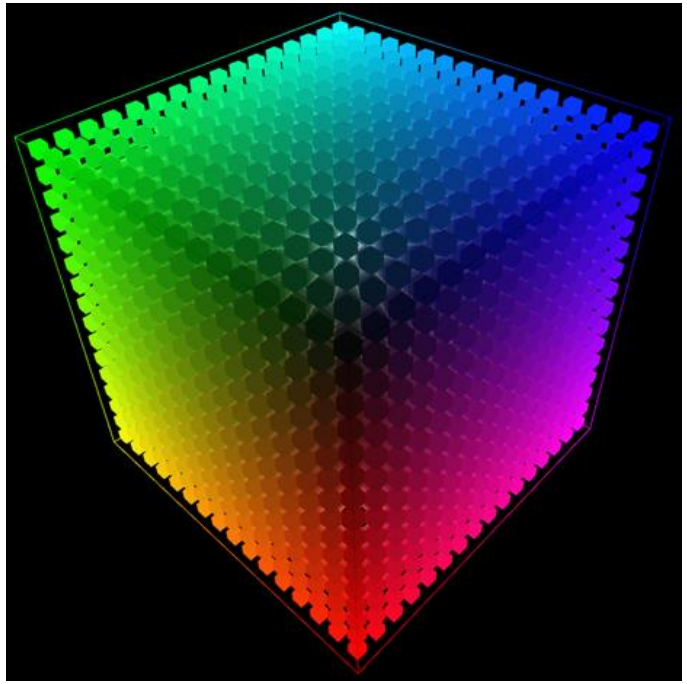
Neutral



Leichter Braunton

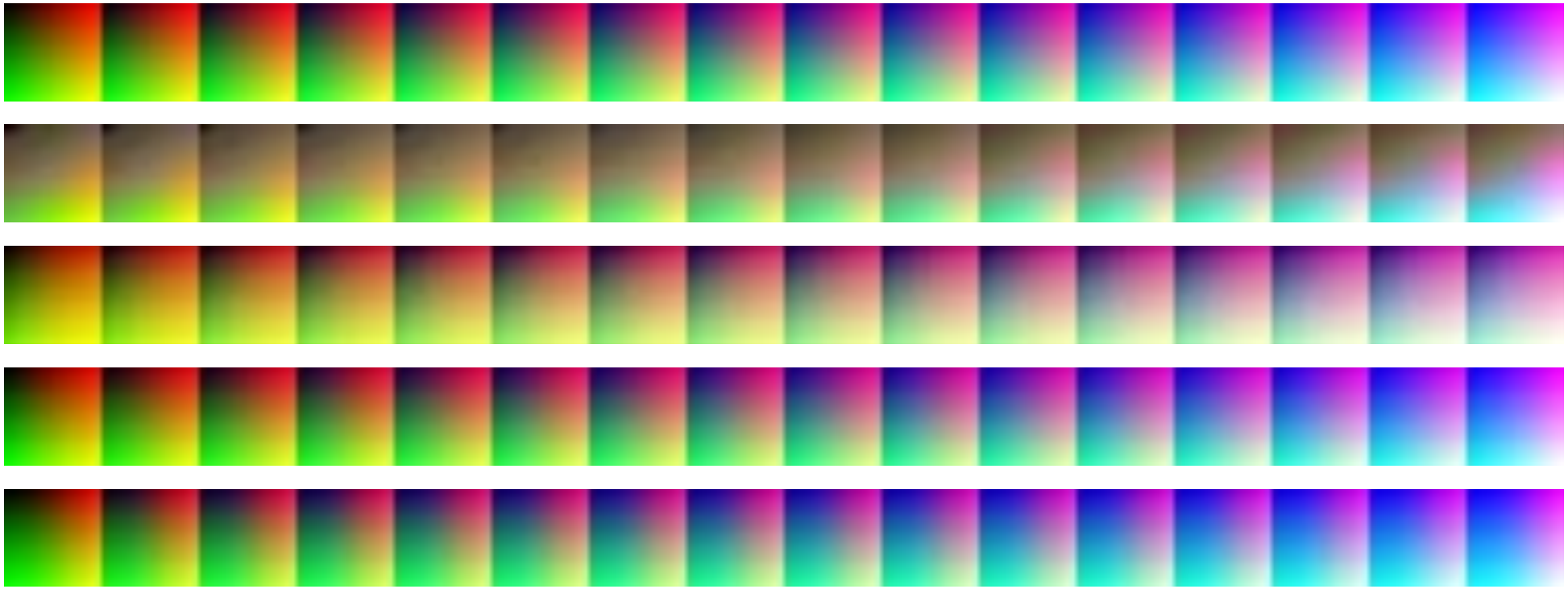
Color Grading (Color Correction)

- Farbe → Punkt im 3D-Raum
  - vec3(r,g,b)
- Idee
  - Speichern von Farben in 3D-Textur
- Identität durch Speichern der Texturkoordinate
- Effekt durch Änderung der Eintrage



Color Grading (Color Correction)

- Auflösung in 3D ist 16x16x16
- Ebenen der 3D-Textur werden als 2D-Textur gespeichert



Dithering

Auch bei korrektem HDR-Rendering mit Gamma-Korrektur und Tone Mapping kann es zu **Banding** kommen.

Dithering kann genutzt werden um die Treppeneffekte **durch Noise zu ersetzen**.

Die im Fragment Shader berechnete Farbe wird dabei genau um **1 Bit (pseudo-zufällig) verändert**.

Originaler Farbverlauf Banding Dithering hinzugefügt

Dithering

Dithering

- Die rote Funktion entspricht einer idealen Speicherung des Helligkeitswerts.
- Ein Helligkeitswert (roter Punkt) würde ohne Dithering durch die geringe Genauigkeit immer abgerundet werden (blauer Punkt auf blauer Funktion) und Banding würde entstehen.
- Durch Dithering wird erreich, dass einige Helligkeitswerte nicht ab- sondern aufgerundet werden.
- Die Wahrscheinlichkeit für das Aufrunden soll proportional zur Helligkeit auf einer Stufe sein.
  - Helligkeitswerte ganz rechts auf einer Stufe sollen eine Wahrscheinlichkeit von 100% haben, da die Helligkeit sozusagen an die nächste Stufe angepasst werden muss.
  - Helligkeitswerte ganz links auf einer Stufe sollen eine Wahrscheinlichkeit von 0% haben.
- Das Aufrunden mit der richtigen Wahrscheinlichkeit wird erreicht, indem ein Zufallswert, der maximal einem Bit entspricht, addiert wird. Damit liegen die Helligkeitswerte im grünen Bereich und alle Werte oberhalb der grünen Funktion werden (bzgl. Der roten Funktion) aufgerundet.
- Da die Helligkeitswerte im Shader zwischen 0 und 1 liegen, entspricht 1 Bit Änderung 1/256.

Ping-Pong Buffer

```
glGenFrameBuffers(2, pingPongFBO); // pingPongFBO ist ein GLuint[2]
glGenTextures(2, pingPongBuffer); // pingPongBuffer ist ein GLuint[2]
for (unsigned int i = 0; i < 2; i++) { //  erstellen der 2 Ping-Buffer Texturen
    glBindFramebuffer(GL_FRAMEBUFFER, pingpongFBO[i]);
    glBindTexture(GL_TEXTURE_2D, pingpongBuffer[i]);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_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);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, pingpongBuffer[i],
0);
}
```

```
for (unsigned int i = 0; i < WIEDERHOLUNGEN; i++) {
    glBindFramebuffer(GL_FRAMEBUFFER, pingpongFBO[i%2]);
    glUniform1i(utils_getUniformLocation(&filterShaderToRepeat, "BaseColorTexture"), 0);
    // …. setzen weiterer Uniforms für den Shader Run
    glBindTexture(GL_TEXTURE_2D, first_iteration ? inputFBOColor : pingPongBuffer[!(i%2)]);
    RenderQuad();
    if (first_iteration) { first_iteration = false; }
} // Nutzung des letzten pingPongBuffers als „final gefiltertes“ Bild
```

NULL-Shader

Shader für Stencil Writing

```
// Host
glDrawBuffer(GL_NONE);       // Kein Color-Attachment binden
```

```
layout (location = 0) in vec3 aPosition;

uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;

void main () {
        gl_Position = projectionMatrix * viewMatrix * modelMatrix * vec4(aPosition, 1.0);
}
```

```
void main () {
        // bleibt leer, da nichts auf einen Color Buffer gerendert wird!
}
```

Effizienz

Achtet auf **Effizienz**, da dies ab dieser Übung mit als wichtiges Kriterium für die Punktevergabe gilt!

- Shaderoptimierung für Branches und For-Loops ist nicht performance optimiert, lieber **mehr kleine Shader** als geschachtelter Code.
- Unnötiges Doppelt-Rendern von Fragmenten möglichst vermeiden
- Unnötig großen Datenfluss von der CPU→GPU (bsp. Nutzung **vec4**, wenn **vec3** reicht) vermeiden (sowie in der Shader-Pipeline)
- Wenn nicht anders benötigt aktivieren von Face-Culling (insb. Back-Face-Culling)
- Speicherfreigabe am Ende des Programmes

Due Date

16.12.