Die Lösungen für diese Übung sind abzugeben bis Sonntag den 02.06.2023 um 18:00h.

Deferred Rendering

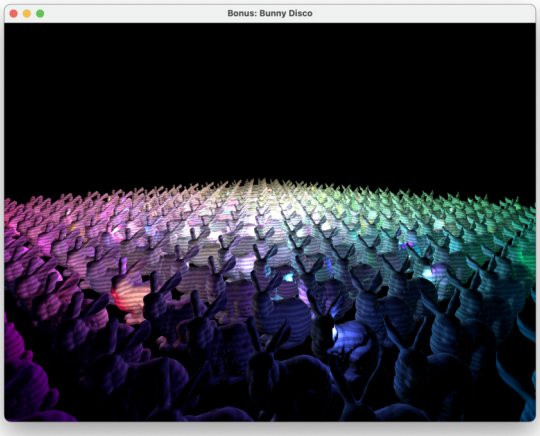
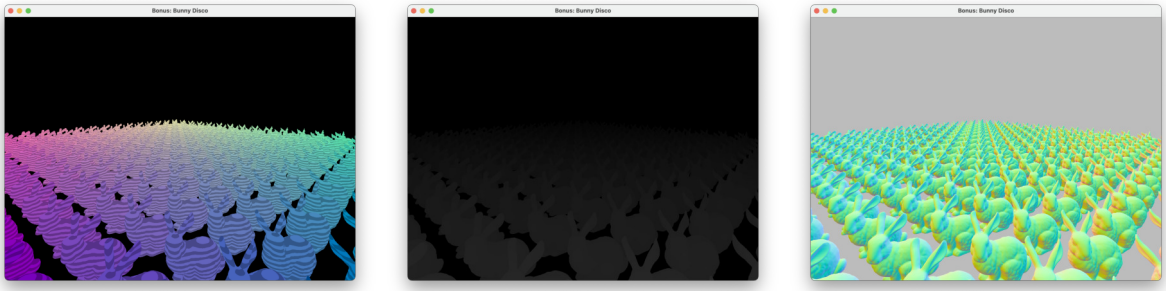


Abbildung 1: So soll das Ergebnis aussehen: Viele Hasen, viele Lichter.

In dieser Bonusaufgabe wollen wir eine Hasenparty für die Stanford-Hasen veranstalten, mit vielen Hasen und vielen bunten Lichtern. Dabei stoßen wir jedoch schnell auf ein Problem: Angenommen wir haben n Hasen und m Lichtquellen. Unser bisheriger Renderer würde jedes Objekt in einem einzelnen Draw-Call auf den Bildschirm zeichnen und für jedes Fragment jeweils über alle Lichtquellen iterieren und die notwendigen Beleuchtungsberechnungen machen, auch wenn Teile des Objekts gar nicht sichtbar sind, weil diese in einem folgenden Draw-Call von anderen Fragmenten überdeckt werden. Im schlimmsten Fall müssen wir so bis zu $m \times n$ Beleuchtungsberechnungen *pro Pixel* ausführen, was bei vielen Objekten und/oder Lichtern schnell teuer werden kann. Diesen Rendering-Ansatz nennt man *Forward-Rendering*.

In diesem Bonusblatt werden wir einen alternativen Ansatz kennenlernen, der in vielen Echtzeit-Graphikengines Anwendung findet: *Deferred-Rendering*. Beim Deferred-Rendering werden die Beleuchtungsberechnungen verzögert (daher “deferred”), indem in einem ersten Rasterisierungs-Schritt nur abstrakte Szenen-Informationen wie Farbwerte, Normalen, Tiefeninformationen und Materialeigenschaften gerendert werden:



In einem zweiten Schritt werden diese Szenen-Informationen dann benutzt um nun die Beleuchtungsberechnungen für jeden Pixel auszuführen. Der große Vorteil ist, dass wir nun für jeden Pixel nur noch m Beleuchtungsberechnungen anstellen müssen und uns unter Umständen viel Rechenarbeit sparen können. Schwierig wird es nur mit Transparenz, doch die betrachten wir hier noch nicht.

Schauen wir jetzt in den Code:

Implementation

Um unsere Beleuchtungsberechnungen aufschieben zu können, müssen wir die Szeneninformationen irgendwo zwischenspeichern. Das machen wir in einem *Framebuffer*, ein Framebuffer ist eine Sammlung von Texturen, in die wir direkt zeichnen können. Wir haben Framebuffer schon benutzt ohne es zu wissen, denn auch um auf den Bildschirm zu zeichnen brauchen wir einen Framebuffer, den unser Fenster automatisch erzeugt. Dieser Framebuffer ist der Default-Framebuffer mit dem Index 0. Wir brauchen jedoch einen zweiten Framebuffer, den `gBuffer` (Geometry-Buffer), den wir genauso anlegen wie alles andere in OpenGL und zwar mit den `glGenFramebuffers` und `glDeleteFramebuffers` Befehlen. Wie bei den anderen OpenGL-Objekten ist die Erzeugung und Zerstörung in einer Klasse `Framebuffer` gekapselt.

Außerdem brauchen wir drei Texturen, je eine für Farbe, Normale und Tiefeninformation. Eine Textur ist einfach ein Bildspeicher in OpenGL und wie für den Framebuffer gibt es eine Klasse `Texture`. Wir müssen für diese Texturen jedoch noch Speicher allokieren und ihnen eine Größe und ein Format zuordnen, das machen wir mit `glTexImage2D`:

```
1  void allocateGBufferTextures(const vec2& resolution, Texture&
       colorTexture, Texture& normalTexture, Texture& depthTexture) {
2      colorTexture.bind(Texture::Type::TEX2D);
3      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, resolution.x, resolution
           .y, 0, GL_RGBA, GL_FLOAT, NULL);
4      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
5      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
6
7      normalTexture.bind(Texture::Type::TEX2D);
```

```
8       glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16_SNORM, resolution.x,
            resolution.y, 0, GL_RGBA, GL_SHORT, NULL);
9       glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
10      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
11
12      depthTexture.bind(Texture::Type::TEX2D);
13      glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH32F_STENCIL8, resolution.x,
            resolution.y, 0, GL_DEPTH_STENCIL,
            GL_FLOAT_32_UNSIGNED_INT_24_8_REV, NULL);
14      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
15      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
16  }
```

Interessant sind hier vor allem die Formate: Die Farbe ist ein linearer 32-bit **`float`**-Vektor mit 4 Komponenten (`GL_RGBA32F`), die Normale ein ganzzahliger vorzeichenbehafteter 16-bit Vektor mit 4 Komponenten, der normalisiert ist, d.h. die höchste Zahl kodiert die Fließkomma 1 und die niedrigste Zahl die -1 (`GL_RGBA16_SNORM`), und neben der 32-bit float Tiefe speichern wir noch 8 Stencilbits, die man für verschiedene Zwecke benutzen kann (`GL_DEPTH32F_STENCIL8`). Euch wird auffallen, dass die 4. Komponente der Normale und der Farbe unbenutzt bleibt, Grafikkarten rechnen jedoch lieber mit 4 Komponenten und diese 4. Komponente lässt sich theoretisch für Zusatzinformationen nutzen.

```
1  allocateGBufferTextures(resolution, colorTexture, normalTexture,
       depthTexture);
2
3  gBuffer.attach(Framebuffer::Type::DRAW, Framebuffer::Attachment::COLOR0
       , colorTexture.handle);
4  gBuffer.attach(Framebuffer::Type::DRAW, Framebuffer::Attachment::COLOR1
       , normalTexture.handle);
5  gBuffer.attach(Framebuffer::Type::DRAW, Framebuffer::Attachment::
       DEPTH_STENCIL, depthTexture.handle);
6
7  gBuffer.bind();
8  std::array<GLenum, 2> drawBuffers = {GL_COLOR_ATTACHMENT0,
       GL_COLOR_ATTACHMENT1};
9  glDrawBuffers(drawBuffers.size(), drawBuffers.data());
```

Diese 3 Texturen müssen jetzt noch als Attachment an den Framebuffer gebunden werden und mit `glDrawBuffers` teilen wir OpenGL mit, dass wir diese auch mit unserem Fragment-Shader beschreiben wollen. Der Tiefenpuffer ist Teil der statischen Pipeline und wird von OpenGL beschrieben.

```
1  ////////////////// Geometry pass /////////////////
2  gBuffer.bind();
3  glEnable(GL_DEPTH_TEST);
4  glDepthMask(GL_TRUE);
5  glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
6
7  meshShader.set("uWorldToClip", worldToClip);
8  meshShader.set("uLocalToWorld", localToWorld);
```

```
9  meshShader.set("uGridSize", static_cast<int>(ceil(sqrt(bunnyCount))));
10 meshShader.bind();
11 mesh.draw(bunnyCount);
```

In unserer Rendermethode müssen wir den Framebuffer nun binden um ihn zu bemalen.

Es gibt aber noch eine weiter Neuerung: Statt nur einen Hasen zu malen, malen wir direkt alle Hasen auf einmal, indem wir der `draw` Methode eine Anzahl als Parameter übergeben. Das ist schneller da es die Kommunikation zwischen Grafikkarte und CPU reduziert, die sonst relativ teuer ist. Diese Technik heißt *Instancing*, das Problem ist jedoch, dass die Uniforms jetzt für jedes Objekt identisch sind, weshalb wir gerade den gleichen Hasen an der gleichen Stelle einfach mehrmals zeichnen. Um die einzelnen Instanzen zu unterscheiden stellt uns OpenGL im Vertex-Shader die Variable `int gl_InstanceID` zur Verfügung. Wir können `gl_InstanceID` dann benutzen um Objekteigenschaften prozedural zu generieren oder z.B. aus Uniform-Arrays auszulesen.

> ***Bonusaufgabe 1 (2 Punkte)*** Benutzt `gl_InstanceID`, um in `projection.vert` den einzelnen Instanzen verschiedene Positionen (`offset`) und Farben (`color`) zu geben. Ihr könnt die Hasen dafür in einem Gitter anordnen, `uGridSize` ist die gerundete Wurzel aus der Hasenanzahl und damit die Kantenlänge eines quadratischen Gitters, Ihr könnt die Hasen jedoch auch auf andere Arten anordnen.

Der Fragment-Shader (`geometry.frag`) bleibt beim Deferred-Rendering relativ minimal: Wir schreiben einfach die interpolierten Vertex-Attribute in die dafür vorgesehen Texturen unseres Framebuffers:

```
1  #version 330 core
2
3  in vec3 interpNormal;
4  in vec3 interpPosition;
5  in vec3 color;
6
7  layout (location = 0) out vec4 gColor;
8  layout (location = 1) out vec4 gNormal;
9
10 void main() {
11     gColor.xyz = (mod(floor(interpPosition.y * 15.0), 2.0) * 0.5 + 0.5)
           * color;
12     gNormal.xyz = normalize(interpNormal);
13 }
```

Hier generieren wir zusätzlich ein prozedurales Streifenmuster, die Farbe wird jedoch meistens mittels Texturkoordinaten aus einer objektspezifischen Farbtextur gelesen.

Nach diesem Rendering-Pass liegen in unseren 3 Texturen die Farb-, Normalen- und Tiefeninformationen unserer Szene. Im Composing-Schritt binden wir diese 3 Texturen zum Lesen an unseren Shader und zeichnen damit ein Dreieck, das den gesamten Bildschirm abdeckt, in unser Fenster:

```
////////////////// Lighting pass /////////////////
Framebuffer::bindDefault();
glDisable(GL_DEPTH_TEST);
glDepthMask(GL_FALSE);

colorTexture.bind(Texture::Type::TEX2D, 0);
normalTexture.bind(Texture::Type::TEX2D, 1);
depthTexture.bind(Texture::Type::TEX2D, 2);
deferredCompose.set("uClipToWorld", inverse(worldToClip));
deferredCompose.set("uResolution", App::resolution);
deferredCompose.set("uLightPositions", uLightPositions);
deferredCompose.set("uLightColors", uLightColors);
deferredCompose.bind();
fullscreenTriangle.draw();
```

Mit `Framebuffer::bindDefault()` binden wir zuerst unseren Window-Framebuffer (Index 0) als aktiven Framebuffer. Die 3 Framebuffer Texturen binden wir jetzt jeweils an die drei Textureinheiten 0, 1, und 2. Eine Textureinheit ist dabei eine Hardwarekomponente der CPU, die eine Textur filtert und ausliest.

Der Vertex-Shader im Composing-Schritt bringt unser Dreieck einfach auf den Bildschirm und berechnet uns eine Texturkoordinate $\in [0,1]^2$, mit der wir aus unseren Framebuffer-Texturen lesen können:

```
#version 330 core

layout (location = 0) in vec3 position;

out vec2 texCoord;

void main() {
    gl_Position = vec4(position, 1.0);
    texCoord = position.xy * 0.5 + 0.5;
}
```

Im Fragment-Shader des Composing-Schritts (`compose.frag`) findet jetzt die eigentliche Arbeit statt:

```
#version 330 core

in vec2 texCoord;

uniform sampler2D tColor;
uniform sampler2D tNormal;
uniform sampler2D tDepth;

uniform mat4 uClipToWorld = mat4(1.0);
uniform vec2 uResolution = vec2(800, 600);
uniform float uBrightness = 1.0;

```

```glsl
uniform int uLightCount;
uniform vec3 uLightPositions[256];
uniform vec3 uLightColors[256];

out vec3 fragColor;

void main() {
    vec3 color = texture(tColor, texCoord).rgb;
    vec3 normal = texture(tNormal, texCoord).xyz;
    float depth = texture(tDepth, texCoord).r;
    // TODO: Use texCoord or gl_FragCoord and depth to calculate the
       fragment position in clip space
    vec4 clipPos = vec4(0.0, 0.0, 0.0, 1.0);
    // TODO: Obtain the world space postion of the fragment
    vec4 worldPosition = vec4(0.0, 0.0, 0.0, 1.0);
    // Note: Divide worldPosition by w

    vec3 lighting = vec3(0.0);
    if (depth < 1.0) {
        for (int i = 0; i < uLightCount; i++) {
            vec3 lightPos = uLightPositions[i];
            vec3 lightColor = uLightColors[i];

            // Calculate lighting
            vec3 lightDir = lightPos - worldPosition.xyz;
            float lightDist = length(lightDir);
            lightDir /= lightDist; // Normalize light direction
            lightColor /= lightDist * lightDist; // Light falloff
            lighting += max(dot(normal, lightDir), 0.0) * lightColor;
        }
        lighting *= uBrightness;
        fragColor = lighting * color;
    } else {
        fragColor = vec3(0.0);
    }
}
```

Wir lesen `color`, `normal` und `depth` mit der `texture`-Funktion aus unseren drei Texturen aus. Dafür müssen wir OpenGL über Uniforms noch mitteilen in welchen Textureinheiten die einzelnen Texturen liegen:

```
deferredCompose.bindTextureUnit("tColor",  0);
deferredCompose.bindTextureUnit("tNormal", 1);
deferredCompose.bindTextureUnit("tDepth",  2);
```

`tColor`, `tNormal` und `tDepth` sind `sampler2D`, ein Sampler ist einfach ein Verweis auf eine Textureinheit mit einem Typ, der beschreibt was für eine Textur die Textureinheit hält.

Für Pixel, die im Vordergrund liegen (`depth < 1.0`) laufen wir jetzt über alle Lichtquellen und summieren die Beleuchtungsterme auf. Da wir Punktlichtquellen haben, brauchen wir die World-Space-

Koordinate des Fragments, um die Helligkeit zu berechnen, die invers quadratisch abhängig vom Abstand der Lichtquelle ist. Die World-Space-Koordinate lässt sich jedoch einfach aus der Fragment Tiefe und Bildschirmposition ableiten.

> ***Bonusaufgabe 2 (3 Punkte)*** Benutzt `depth` und `texCoord` oder `gl_FragCoord` und `uResolution` um die Fragment-Position in Clip-Space bzw. Normalized-Device-Coordinates zu ermitteln und transformiert diese dann in den World-Space um `worldPosition` zu berechnen.

Die Farben und Positionen der Punktlichtquellen werden auf der CPU generiert und als Uniform-Arrays an den Fragment-Shader übergeben:

```
vec3 xzCircle(float r, float t) { return r * vec3(cos(t), 0.0f, sin(t)); }

void genLightPositions(std::vector<vec3>& lightPositions, float radius, float t, size_t count) {
    lightPositions.reserve(count);
    srand(42);
    for (size_t i = 0; i < count; i++) {
        if (i >= lightPositions.size()) lightPositions.push_back(vec3());
        vec4 rand = linearRand(vec4(-1.0f), vec4(1.0f));
        lightPositions[i] = xzCircle(rand.x, t * rand.y) + xzCircle(-rand.z, t * rand.w);
        lightPositions[i] *= radius;
        lightPositions[i].y = linearRand(0.0f, 1.0f);
    }
}

void genLightColors(std::vector<vec3>& lightColors, size_t count) {
    lightColors.reserve(count);
    for (size_t i = lightColors.size(); i < count; i++) {
        lightColors.push_back(linearRand(vec3(0.0f), vec3(1.0f)));
    }
}
```

Die Farben ziehen wir zufällig mit `glm::linearRand`, für die animierten Positionen benutzen wir ein sogenanntes Double Pendulum, d.h. wir addieren zwei Kreisbahnen mit zufälligen Radii und Winkelgeschwindigkeiten. Dadurch entsteht eine chaotische aber glatte Bewegung.