University of Tübingen
Prof. Dr.-Ing. Hendrik P.A. Lensch
Chair of Computer Graphics
Faezeh Zakeri (faezeh-sadat.zakeri@uni-tuebingen.de)
Lukas Ruppert (lukas.ruppert@uni-tuebingen.de)
Raphael Braun (raphael.braun@uni-tuebingen.de)
Andreas Engelhardt (andreas.engelhardt@uni-tuebingen.de)
Philipp Langsteiner (philipp.langsteiner@student.uni-tuebingen.de)

24. January 2023

Computer Graphics
Assignment 10

Submission deadline for the exercises: 31. January 2023, 0:00

Source Code Solutions

- Upload **only** the files that you changed.
  Do NOT include your build folder, results, meshes, ext, or exercise sheet in the zip file you upload.
- Zip them first and upload the **.zip** file on ILIAS.

Written Solutions

Written solutions have to be submitted digitally as one PDF file via ILIAS.

10.1 Light Field Rendering (20 Points)

In this exercise we will determine the pixel-value for a virtual camera using an image based rendering technique.
Let's say we are given a Light Field in st - uv parameterization, thus we know the geometry of all rays in the Light Field. For simplicity we only consider the s and u coordinates in this exercise.
The Light Field with the s and u coordinates and the corresponding rays is visualized in Figure 1. The pixel-value of all rays is given in Table 1.
Your task is to interpolate the value for the red ray in Figure 1 from its four closest rays.

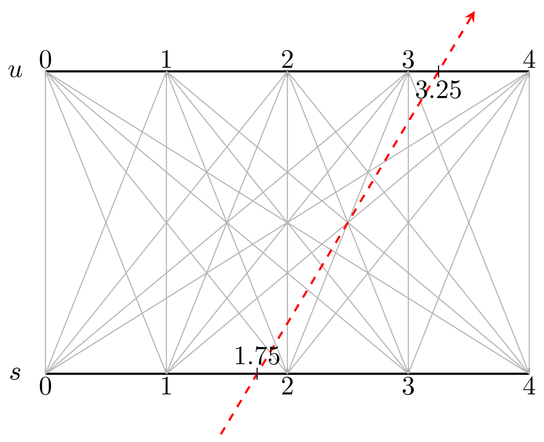


Figure 1: $s-u$ parameterization of the rays in a Light Field. Each ray is described by a s and u coordinate. All existing rays of the Light Field are hinted in light gray. You are supposed to interpolate the *value* of the red dashed ray.

s	u	value	s	u	value	s	u	value	s	u	value	s	u	value
0	0	10	1	0	100	2	0	130	3	0	160	4	0	180
0	1	30	1	1	220	2	1	140	3	1	110	4	1	100
0	2	130	1	2	150	2	2	200	3	2	220	4	2	230
0	3	12	1	3	90	2	3	110	3	3	150	4	3	180
0	4	200	1	4	200	2	4	180	3	4	160	4	4	140

Table 1: Pixel values of all rays in the Light Field.

10.2 Sphere Tracing: Pen and Paper (20 Points)

Given the scene and ray from Figure 2, execute sphere-tracing along the ray. We expect to see the sampling positions along the ray, as well as the circles, which touch the closest surface points around those sampling positions.

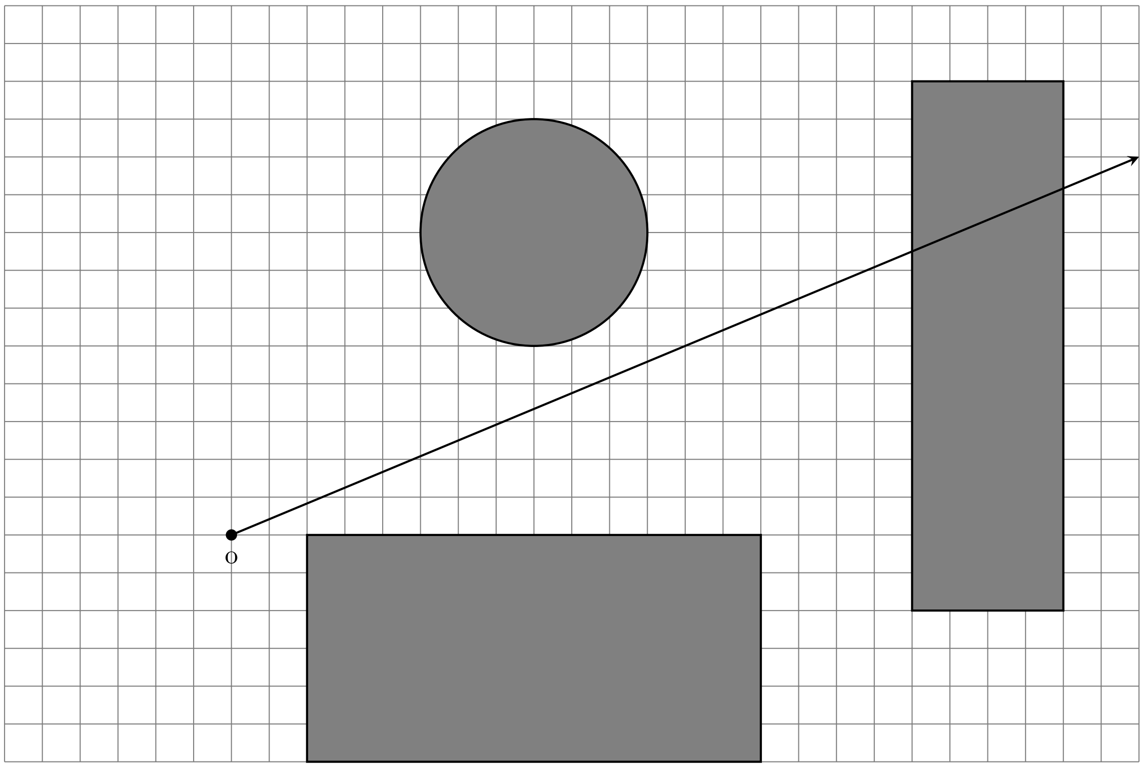


Figure 2: Scene for sphere tracing. The ray starts at o. Draw in the samples along the ray, that would be created during sphere-tracing. Also draw in the circles that intersect with the closest surfaces, which are needed to determine the sampling positions.

10.3 Sphere Tracing: 3D Rendering (15 + 10 = 25 Points)

Instead of tracing rays against the triangles of a mesh, we can also trace spheres against an implicit surface representation, e.g. a signed distance function (SDF). You can find the signed distance function of many shapes online: https://iquilezles.org/articles/distfunctions/. A small selection of them is already implemented. You don't have to implement any of them yourself, but feel free to add more shapes on your own time if you're interested.

Your tasks are the following:

a) Compute the intersection point using sphere tracing:

```
std::pair<Point3D, uint32_t> SignedDistanceFunction::sphereTrace(Ray ray) const
```

Use sphere tracing to advance the position of the `ray` into its `direction` until the distance to the surface becomes less than ε (`epsilon`). Or until you have travelled a distance greater than `ray.tMax` away from the `ray`'s `origin`.

Return your final position (hit or miss) and the number of steps you took. You can visualize the number of steps taken in the GUI, e.g. Figure 3, to see where sphere tracing is more efficient and where it becomes costly.

b) Compute the normal using central differences:

```
Normal3D SignedDistanceFunction::computeNormal(Point3D pos) const
```

A nice property of signed distance functions is that their derivative always points into the direction of the surface normal. While some SDFs would allow us to compute this derivative analytically, here, we go for a simpler and more general approach:

Given some SDF $f(\mathbf{p})$, we approximate its gradient as follows:

$$\nabla f(\mathbf{p}) \approx \frac{1}{2\varepsilon} \begin{pmatrix} f(\mathbf{p} + (\varepsilon, 0, 0)^T) - f(\mathbf{p} - (\varepsilon, 0, 0)^T) \\ f(\mathbf{p} + (0, \varepsilon, 0)^T) - f(\mathbf{p} - (0, \varepsilon, 0)^T) \\ f(\mathbf{p} + (0, 0, \varepsilon)^T) - f(\mathbf{p} - (0, 0, \varepsilon)^T) \end{pmatrix} \tag{1}$$

Rather than dividing by 2ε and trusting on the gradient to have a magnitude of 1 (which is the case for exact SDFs), simply `normalize` the resulting vector and return it as the surface normal for the given intersection point.

You can also find this method described here: https://iquilezles.org/articles/normalsSDF/.

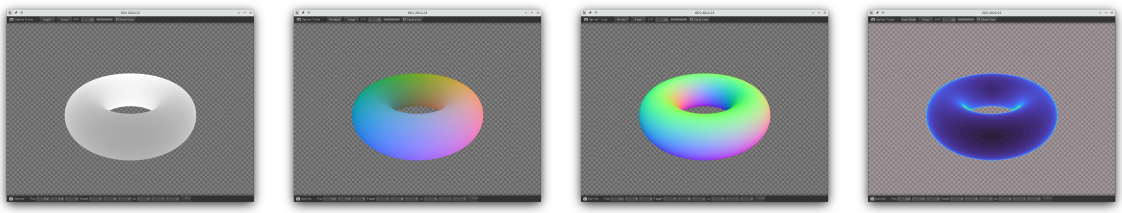


Figure 3: Sphere tracing can be used to render implicit surfaces defined by signed distance functions. Here, we restrict ourselves to simple debug outputs like depth, position, normal, and number of steps taken. However, we could also compute more advanced shading and even global illumination given just this information.

10.4 Volume Rendering: Homogeneous Volumes (5 + 20 + 10 = 35 Points)

In this task, we will do some simple volume rendering. For the practical part, we only consider homogeneous volumes with constant absorption μ_a and no scattering ($\mu_s = 0$). This allows us to render basic volumetric effects as shown in Figure 4.

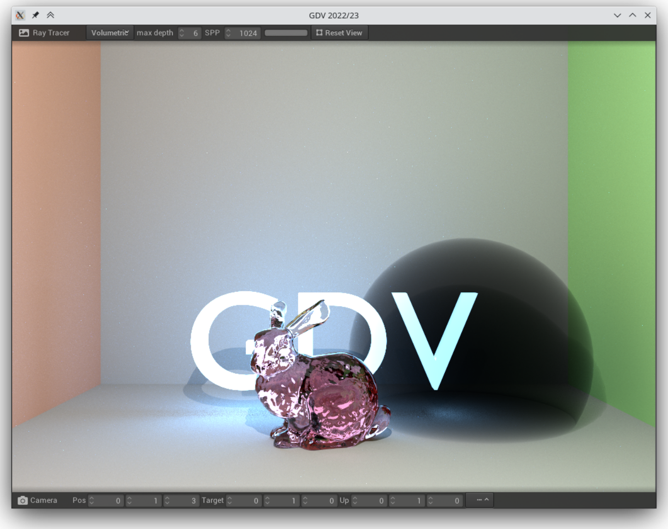
a) Implement the transmittance $T(z)$ in a homogeneous volume in

```
Color Material::Absorption::transmittance(float distance) const
```

it is defined as

$$T(z) = e^{-\tau(z)}, \quad \text{where} \quad \tau(z) = z\mu_a.$$

Here, z is the distance travelled in the medium.


GDV 2022/23

Ray Tracer · Volumetric · max depth 6 · SPP 1024 · Reset View

Camera · Pos 0 1 3 · Target 0 1 0 · Up 0 1 0

Figure 4: Transmittance allows us to render colored glass or simple smoke.

b) In order to render an image with transmittance effects, one needs a volumetric path tracer. You can find it in `volumepathtracer.cpp`. It is an extension of the path tracer introduced in exercise 5 (now in `raytracer.cpp`). For now, it only supports transmissive volumes, but no scattering.

Explain how it computes direct light sampled from the scene's light sources and point out the differences to the non-volumetric path tracer.

c) If we were to also implement scattering in our volume, we could define that with a scattering coefficient μ_s and a phase function. Then, whenever we are inside a volume, we would have to sample free paths along the ray which do not hit any of the volume's particles and then potentially scatter into different directions once we hit a scattering particle.

Write down the formula for sampling free-flight distances t in a homogeneous volume with attenuation coefficient $\mu_t = \mu_a + \mu_s$ (absorption + scattering).

How do you expect the rendering cost to change as the probability for scattering μ_s increases?