UNIVERSITÄT DES SAARLANDES
Prof. Dr.-Ing. Philipp Slusallek
Computer Graphics Group
Arsène Pérard-Gayot (perard@cg.uni-saarland.de)

16. December 2019

# Introduction to Computer Graphics
## Assignment 7

**Submission deadline for the exercises**: 06. January 2020

The paper copies for the theoretical parts of the assignments will be collected at the beginning of the lecture on the due date. The programming parts must instead be marked as release before the beginning of the lecture on the due date.
The code submitted for the programming part of the assignments is required to reproduce the provided reference images. The submission ought to tag the respective commit in your group's git repository as well as attach the mandatory generated images to the release. The submission should also contain a creative image show-casing all extra-credit features that have been implemented.
The projects are expected to compile and work out of the box. A successful build by Drone CI is a good indicator. If it fails to do so on our Windows or Linux computers, we will try to do so on the CIP-pool students' lab as a "fallback".
**To pass the course you need for every assignment at least 50% of the points.**

## 7.1 Sampling Theory (10 Points)

Let $f(x)$ be an infinite signal that fulfills the Nyquist property, thus the highest frequency of the signal is smaller than $\frac{1}{2T}$ if $T$ is the sampling distance. Consider a regular sampling $f_T(x)$ of $f(x)$ with sample distance $T$.

**a)** Is an exact signal reconstruction of $f(x)$ possible? If so, why?

**b)** What mathematical operations in Fourier space need to be performed to reconstruct the image?

**c)** What mathematical operations in image space need to be performed to reconstruct the image?

## 7.2 Color Space (10 Points)

Compute the position of the sRGB color (1,0,1) in the CIE-XYZ and CIE-xy color space. Argue why gamma correction in sRGB is irrelevant for the transformation of this particular color.

## 7.3 Super Sampling (10 Points)

In order to reduce the aliasing effects one can shoot multiple, slightly perturbed rays per pixel and average the results out. You will need to:

- Implement `Renderer::setSamples` which should specify how many rays should be shot for each pixel.
- The `Renderer::render` should take that information into account. If more than one ray per pixel is being shot, it should no longer go through the center of the pixel, but anywhere within the pixel area.

We are providing a `random()` function[1] in `random.h`, which returns a floating point number in the range $[0, 1)$. The function is thread-safe.

[1] We are using the Mersenne Twister algorithm. The state of the machine is kept in a thread-local variable, so calls to the function in one thread have no impact on the results in another.

## 7.4 Area Lights (20 Points)

Materials can emit light on their own, but so far that kind of light remained ignored during lighting: Shadow rays have been shot only to "virtual" lights (`PointLight`, `SpotLight` and `DirectionalLight`). Your task is to bring those two together. Implement an `AreaLight` that accepts any `Solid` as an argument and uses its material to compute the light intensity. When computing the lighting you should:

- Use `Solid::sample()` to sample a random point on the surface of the solid. Implement the said method for `Triangle` and `Quad`. Please note, that we changed the signature in order to return a struct that contains also the normal of the sampled surface point. Please adapt the interface of all derived solids accordingly.
- Use `Material::getEmission()` to get the light intensity. Currently, the arguments to `getEmission()` can be set to dummy values. We will be using only `LambertianMaterial` with constant textures.

Area light sources generate soft shadows when the source is only partially covered by an obstacle. The multisampling technique from the previous exercise can greatly help reducing the noise.

## 7.5 Distributed Materials (20 + 20 Points)

So far, all materials described their interaction with light in a deterministic way. This, combined with the fact that a material could specify only a single direction of interest through `Material::getSampleReflectance()` is very limiting. In this exercise we allow materials to have multiple directions that are worth sampling and expect it to pick one at random.
Implement:

- `GlassMaterial`, representing a material that reflects and refracts incoming light. A single index of refraction `eta` defines the new medium. If the incoming ray is coming from "below" the solid (the angle between the ray direction and normal is greater than 90°), assume that the ray is leaving the medium and use the inverse of `eta`.
- `FuzzyMirrorMaterial`, which behaves similarly to mirror, but the reflected ray is randomly perturbed. The perturbation is defined by a maximum angle that the ray may deviate from the perfect reflection. You can approximate this deviation by sampling a random point $p$ on an imaginary disc perpendicular to the perfect reflection ray, at a unit distance from the hit point. Then, set the ray direction such that it goes through $p$.

You may also want to update the `CombineMaterial` such that it supports multiple sampling sub-materials by picking one at random, but we are not going to test it.

## 7.6 Depth of Field (10 Points)

Implement the depth of field effect, as an extension to the perspective camera (`DOFPerspectiveCamera`). Two new parameters appear:

- `focalDistance` defines the distance from the center of the camera to an imaginary plane at which all the objects should appear sharp.
- `apertureRadius` defines the maximum ray origin perturbation to control the strength of DOF.

## 7.7 Motion Blur (10 Points)*

Implement the Motion Blur effect.

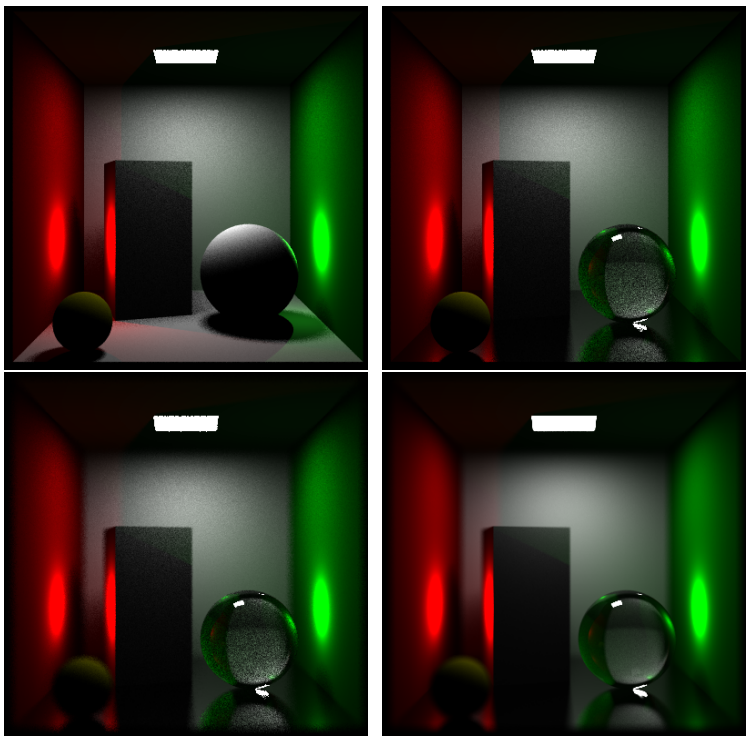

Figure 1: Distributed raytracing. (a) Super sampling and area light source (b) Glass and fuzzy mirror (c) depth of field (d) Higher-quality rendering with 1000 rays shot per pixel

## 7.8 Smooth Edges (5 Points)*

Most models are described by a collection of flat surfaces. This becomes very apparent when doing lighting as each surface has a single, constant normal.
A common technique for keeping the polygon count low but making it less apparent, is to perturbe the normal vector depending on the location where the ray hit the flat surface. For example, a triangle can be given three fake normals on its vertices; when a ray hits the primitive, the normals are interpolated for the given hit point.
Your task is to implement a `SmoothTriangle` solid, which should perform this normal interpolation. The triangle intersection routine should remain unchanged with the exception for the returned normal within the `Intersection` structure.

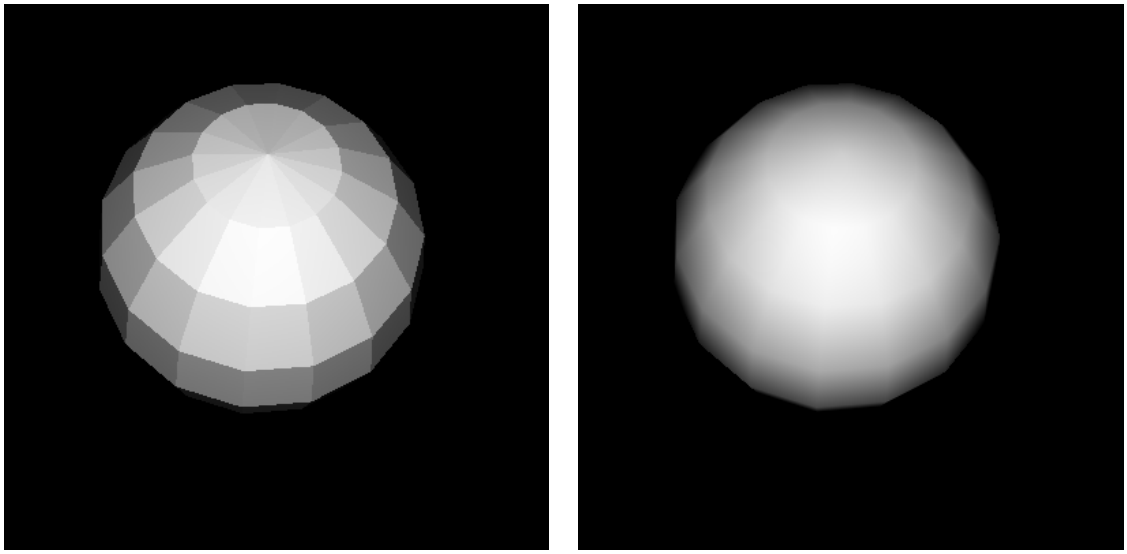

Figure 2: A tesselated sphere, without and with interpolated normals.