UNIVERSITÄT DES SAARLANDES
Prof. Dr.-Ing. Philipp Slusallek
Computer Graphics Group
Hugo Devillers (devillers@cg.uni-saarland.de)
Ömercan Yazici (yazici@cg.uni-saarland.de)

17. December 2021

Introduction to Computer Graphics
Assignment 6

Submission deadline for the exercises: 14. January 2022

The theoretical parts should be uploaded as a single PDF (scan, LaTeX, ...) in the assignment in Microsoft Teams. Please write your name, the name of your partner and the group name (e.g. group42) on top of the sheet. Both of the team members have to upload and submit the same PDF!
For guest students or those who have trouble with Teams, we can exceptionally accept submissions by email, in which caseplease send them by email to Ömercan.
The programming parts must 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.
To pass the course you need for every assignment at least 50% of the points.

6.1 Interpolation (5 + 5 Points)

Interpolation is a method of computing new data points from a given discrete set of data points. Linear interpolation interpolates the missing values of a function with one variable $f : \mathbb{R} \to \mathbb{R}$ linearly for the domain interval $[x_0, x_1]$. For example: given two data points $f(2) = 5$ and $f(4) = 8$ (interval is [2, 4]) one computes the approximate value of $f(x)$ for $x = 2.5$ as $f(x) \approx 5.75$.

Bilinear interpolation is an extension to linear interpolation for interpolating functions of two variables $f : \mathbb{R}^2 \to \mathbb{R}^2$. The key idea of bilinear interpolation is first to perform linear interpolation in one direction (i.e. x-direction) and afterwards in another direction (i.e. y-direction). In our case we will use bilinear interpolation later to interpolate texture color values.

- Show that the linear interpolated value of a function can be computed as following:

$$f(x) \approx f(x_0) + (x - x_0)\frac{f(x_1) - f(x_0)}{x_1 - x_0}, \text{ where } x \in [x_0, x_1]$$

- Derive an equation for computing bilinear interpolated value of $f(x, y)$. You can assume that the values of f at the four points (x_0, y_0), (x_0, y_1), (x_1, y_0) and (x_1, y_1) are given.

  Note: $x_0 \leq x \leq x_1$ and $y_0 \leq y \leq y_1$.

6.2 Fourier Transformation (10 Points)

The transformation of a signal f to Fourier space is given by:

$$\mathcal{F}(f) = F : F(k) = \int_{-\infty}^{+\infty} f(x) \cdot e^{-2\pi i k x} dx$$

Show that the Fourier transformation of the box function $B(x)$ is a *sinc* type function. The sinc function is defined as $sinc(x) = \frac{sin(\pi x)}{\pi x}$.

$$B(x) = \begin{cases} 0 & \texttt{for } x \leq -1 \\ 1 & \texttt{for } -1 < x < 1 \\ 0 & \texttt{for } 1 \leq x \end{cases}$$

6.3 Sampling Theory (3 + 3 + 3 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?

6.4 Basic textures (4 Points)

A texture is a function that takes an input coordinate (usually a 2D or 3D point) and returns a color. We introduce a new abstract class `Texture` to represent this concept.
In addition to the basic look up functionality, a derivative in X and Y direction of the texture can be computed. The derivative should be computed for each color channel separately.
Implement a `ConstantTexture` which should represent a constant-color function.

6.5 Materials (8 + 8 Points)

Implement the following materials:

- `LambertianMaterial` which should represent a diffuse material, with self-emission. The material is modulated not by color values, but by textures. Currently the texture is just a constant color, but in the future this will allow us to use more fancy texturing without reimplementing the materials.

  The values returned by the texture are expected to be in range $[0, 1]$, where 0 means complete absorption and 1 complete reflection.

- `PhongMaterial` which represents a purely specular material, defined by the color weight (as texture), and a reflection exponent.

6.6 Rendering mirrors (8 + 10 Points)

Implement a `MirrorMaterial`. The mirror is parametrized by `eta` (index of refraction) and `kappa` (absorption index) as discussed in the lecture.
A mirror is a substantially different material than those implemented so far: when computing the amount of light reflected towards the given direction `outDir`, you do not need to iterate over the light sources as the reflectance for all `inDir` (except one) is going to be 0. The material itself, rather than the integrator, knows which `inDir` vectors are worth considering.
In other words, the incoming radiance function needs to be *sampled* and this sampling process is controlled by the material. This sampling process is controlled by the following two member functions (previously left unimplemented):

- `useSampling()` merely indicates the rendering strategy that should be used for the material:
  - `SAMPLING_NOT_NEEDED` — the material can be rendered by iterating over the light sources.
  - `SAMPLING_ALL` — indicates, that iterating over the light sources is unproductive and you should sample the material directly.
  - `SAMPLING_SECONDARY` — a combination of both: you should iterate over the light sources, but also do sampling.

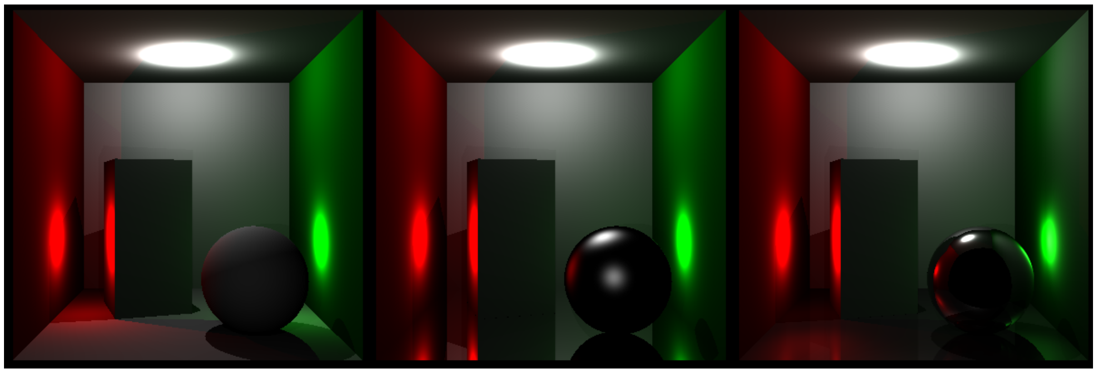

Figure 1: Final images produced in the Assignment 6, using Lambertian materials (a), Phong and mirror (b), and combining materials (c).

- `getSampleReflectance` — returns a pair: The `direction` vector for which irradiance is worth computing, and the `reflectance` of the material for the pair (`direction`,`outDir`).

In addition to the `MirrorMaterial`, update all other materials to include proper implementation of the sampling functions. If `useSampling` returns `SAMPLING_NOT_NEEDED`, the `getSampleReflectance` can return an arbitrary direction vector, and reflectance can be set to 0.
Given the new, extended functionality of the materials, we now need to update the integrator. We are introducing a new, `RecursiveRayTracingIntegrator` which should take the `Material::useSampling()` into account, and for materials requiring sampling it should cast secondary rays in order to compute the needed irradiance. The integrator name comes from the fact that you recursively invoke `getRadiance` for the secondary rays. The integrator should prevent infinite loops (e.g. when you have two parallel mirrors), by setting some maximum recursion depth. The recursion depth of 6 should be sufficient for the given test scenes, but you may want to permit a higher value for your own compositions.

6.7 Combining materials (10 Points)*

Implement a `CombineMaterial` which linearly combines an arbitrary set of other materials. The `CombineMaterial::add` allows you to add a new material, with a given weight into the combination. When lighting is computed, all the component materials should be considered and the result should be weighted accordingly.
Assume that at most one of the combined materials requires sampling.

6.8 Cook-Torrance Material (10 Points)*

Implement `CookTorranceMaterial` with Cook-Torrance BRDF model using the anisotropic extension of Blinn's micro-facet distribution.

6.9 Interpolation (3 + 3 + 4 + 4 Points)

It is common to represent textures as a collection of samples (an array of pixels, for instance). The reconstruction of the original signal can be done simply by interpolating neighbouring values. Your task is to implement the interpolation functions:

- `lerp` should interpolate between two values `px0` and `px1`. It is assumed that the values are provided for 1D points (0) and (1) and the input parameter `xPoint` is a 1D point in between.

- `lerpbar` should interpolate between three values `a`, `b`, `c` specified at the corners of an imaginary triangle. The interpolation point is specified by its barycentric coordinates (`aWeight`, `bWeight`, 1 - `aWeight` - `bWeight`).

- `lerp2d` is a bilinear interpolation.

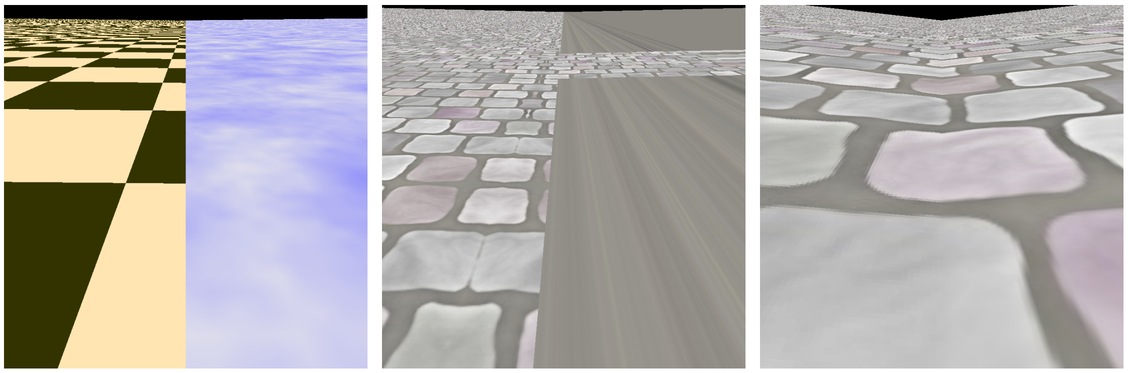

Figure 2: Textures. (a) checkerboard and perlin noise. (b) mirror and clamp border handling. (c) nearest-neighbour and bilinear interpolation.

- `lerp3d` is a trilinear interpolation.

The template parameter `T` of these functions is assumed to support:

- Multiplication by a scalar (`T*float`)
- Binary addition (`T+T`)

The `Point` class does not support addition and for that reason, when interpolating, it has to be cast to `Float4`. For convenience, we are overloading all these functions for `Point` explicitly and perform the necessary casting in `interpolate.cpp`.
Note that the template function implementation cannot be put in a `.cpp` file. Please provide the implementation in `interpolate-impl.h` which is going to be included every time you use `interpolate.h`.

6.10 Textures (5 + 10 + 30 Points)

Implement `FlatMaterial` which will be used for testing the textures. The material should not reflect any light, but should emit the color corresponding to the input texture.
Implement the following textures:

- `CheckerboardTexture`. The checkerboard is a 3D texture, taking all three coordinates into account. It should form a 3D version of a checker board, with each cube (`white` or `black`) having edge length 0.5.

- `PerlinTexture` using the concept of 3D value-based Perlin noise. The Perlin noise is configurable through the `addOctave`. The `frequency` 1 indicates that noise function is taken for the integer coordinates and then linearly interpolated in between. Higher frequency should map to a more dense noise.

  Once the noise value is computed, it should be used to interpolate between `white` (for value 1) and `black` (for value 0) colors.

  The integer noise function is provided for you as a convenience in `perlin.cpp`. For any integer input it produces a persistent random number in range $(-1, 1)$.

- `ImageTexture` which uses an image as a texture. The input coordinates are normalized. The image texture is additionally parametrized by how image borders are handled (`CLAMP`, `MIRROR`, `REPEAT`) and how values should be interpolated (`NEAREST`, `BILINEAR`) assuming node-centered sample values.

At the moment derivatives (`getColorDX` and `getColorDY`) do not have to be computed.

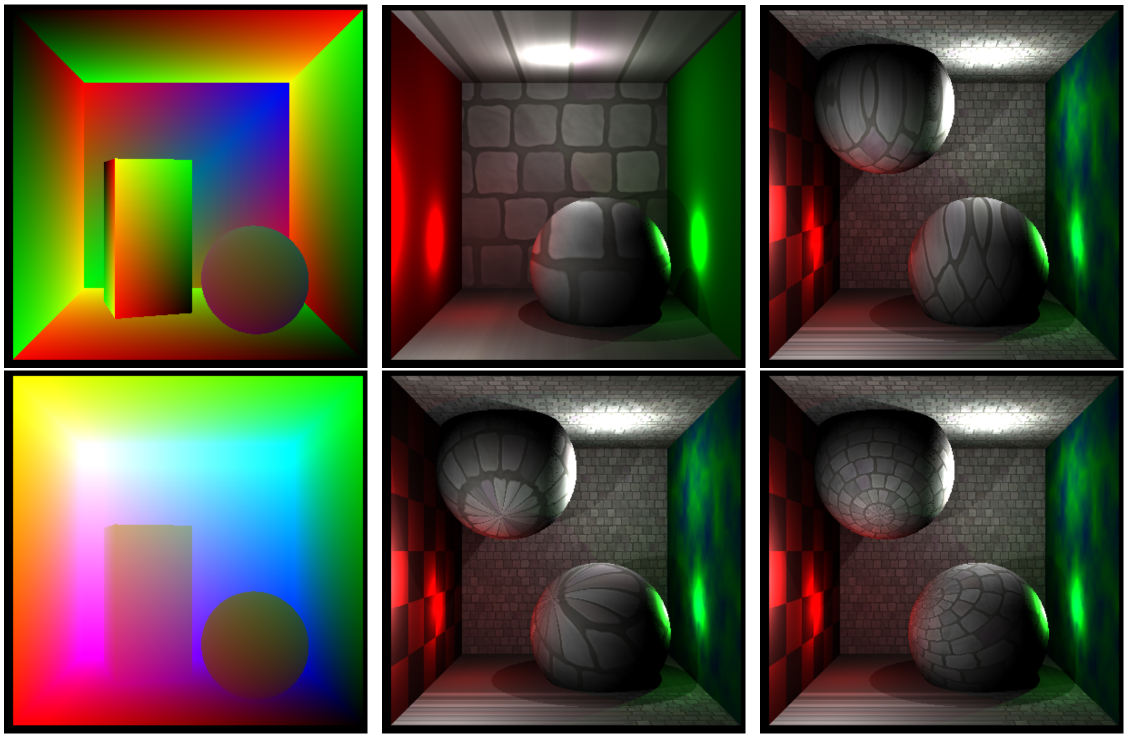

Figure 3: False colored images showing computed local coordinate (top left) as well as the hitpoint (bottom left). Images testing different coordinate mappers. No mapping (top center), triangle mapping on the walls and planar mapping for the spheres (top right), cylindrical mapping (bottom center), spherical mapping (bottom right).

6.11 Texture Coordinate Mappers (5 + 10 + 20 + 20 + 10 Points)

So far you have passed the local hit coordinate into the material (`texPoint`). More often than not however, you want to be able to transform the texture used by the material, without modifying the geometry. To accomplish that, we are using the texture coordinate mappers (`CoordMapper`).
The `CoordMapper::getCoords` gets a complete hit information (`Intersection`) and uses it to compute a new point in the texture space. In most cases the coordinate mapper will use `Intersection::local()`, but some more fancy mappers may require additional information.
Your task is to implement the following coordinate mappers:

- `WorldMapper` taking the world hit coordinate, scaling it and returning the result.
- `PlaneCoordMapper` defined by two vectors defining a texture plane in 3D space. The local hit coordinates should be projected onto that plane, yielding the texture coordinates.
- `CylindricalCoordMapper` projects local hit coordinates onto a cylinder. The cylinder is defined by:
  - `origin` — a point on the cylinder axis.
  - `longitudinalAxis` — the cylinder axis. The direction along which the y coordinate of the texture increases. The magnitude of the vector defines the scaling along the y direction.
  - `polarAxis` — a direction intersecting a cylinder at some point. This direction defines the $x = 0$ coordinate. The magnitude of the vector defines the scaling along the x coordinate. A vector length 1 means that a full angle maps to $x = 1$.

  The input `polarAxis` does not have to be perpendicular to `longitudinalAxis`. If that happens, you may need to compute the perpendicular vector first.

- `SphericalCoordMapper` projects local hit coordinates onto a sphere. The spherical coordinate system is defined by:
  - `origin` — the center of the sphere
  - `zenith` — the direction towards the north pole of the sphere. The magnitude defines the scaling along the y texture direction. The vector length 1 means that the north pole maps to $y = 1$ and south pole to $y = 0$.
  - `azimuthRef` — a direction defining the prime meridian (may not be perpendicular to *zenith*). The magnitude defines the scaling along the x texture direction. A vector length 1 means that a full angle maps to $x = 1$.
- Barycentric triangle mapper `TriangleMapper` — a coordinate mapper intended to be used for triangles. The mapper is defined by texture coordinates in the 3 vertices of the triangle. Given the hit point, it should use the local (barycentric) coordinates to compute the texture coordinates.

For the coordinate mappers to work correctly with the rest of the framework you must ensure that:

- `Intersection::local()` returns *Cartesian* local coordinates of the hit point. The exception is a triangle (and quad), which should produce barycentric coordinates, correctly normalized.
- `RayTracingIntegrator` and `RecursiveRayTracingIntegrator` should invoke the coordinate mapper to obtain the texture coordinates before querying the material.
- `Solid`s should still accept the `nullptr` as a coordinate mapper. If that happens, it should default to `WorldMapper` with uniform scaling factor 1.

To ease your task, we added two debug views that visualize the computed local coordinate as well as the hitpoint (x in red, y in green, z in blue).

6.12 Environment Map (10 Points)*

Implement the environment map — applying color to rays that do not intersect any normal geometry of your scene. You do not want to change the renderer nor the integrator to accomplish that. You can implement it the following way:

- Introduce a new special `Solid` that always intersects with any ray at infinity.
- Create a new coordinate mapper `EnvironmentMapper` that will ignore the local/global hit coordinates and use the hit ray direction instead.
- Use an emissive material (e.g. the `FlatMaterial`).

Some problems you may encounter:

- If lack of intersection, e.g. in `Intersection::operator bool()`, is detected by checking `Intersection::distance == FLT_MAX` your special solid intersection may be regarded as no-intersection. You may want to check if `Intersection::solid` is set to `nullptr` instead (and ensure that `Intersection::failure()` sets it as such.
- Recall that `DirectionalLight` is using `FLT_MAX` as a distance to the source. The special solid may cast shadow which you probably do not want. This may be easily solved by using $<$, instead of $\leq$, when comparing the ray distance instersection and distance to the light source.

6.13 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.

6.14 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.

6.15 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.

6.16 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.

[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.

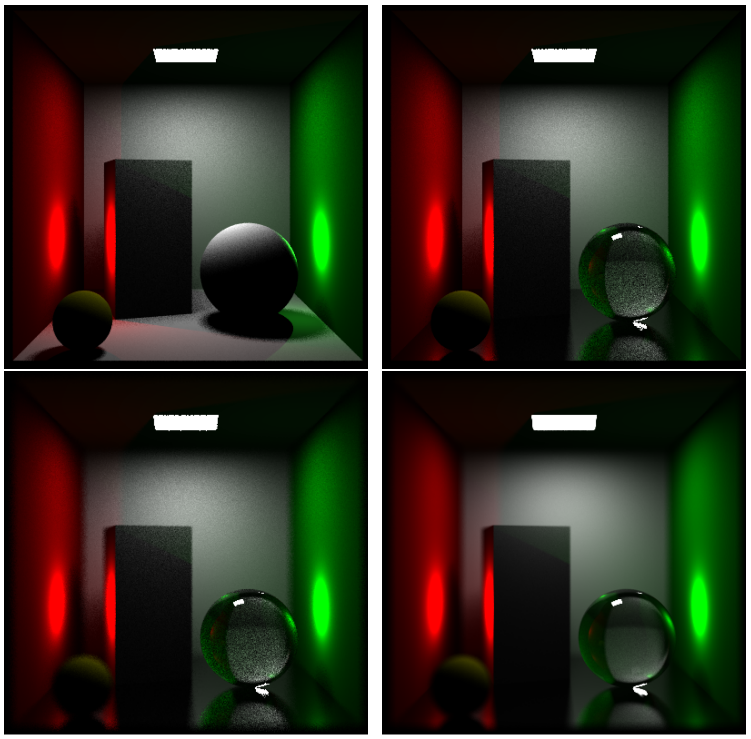

Figure 4: Distributed raytracing. (a) Super sampling and area light source (b) Glass and fuzzy mirror (c) depth of field (d) Higher-quality rendering with 1000 rays shot per pixel

6.17 Motion Blur (10 Points)*

Implement the Motion Blur effect.

6.18 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.

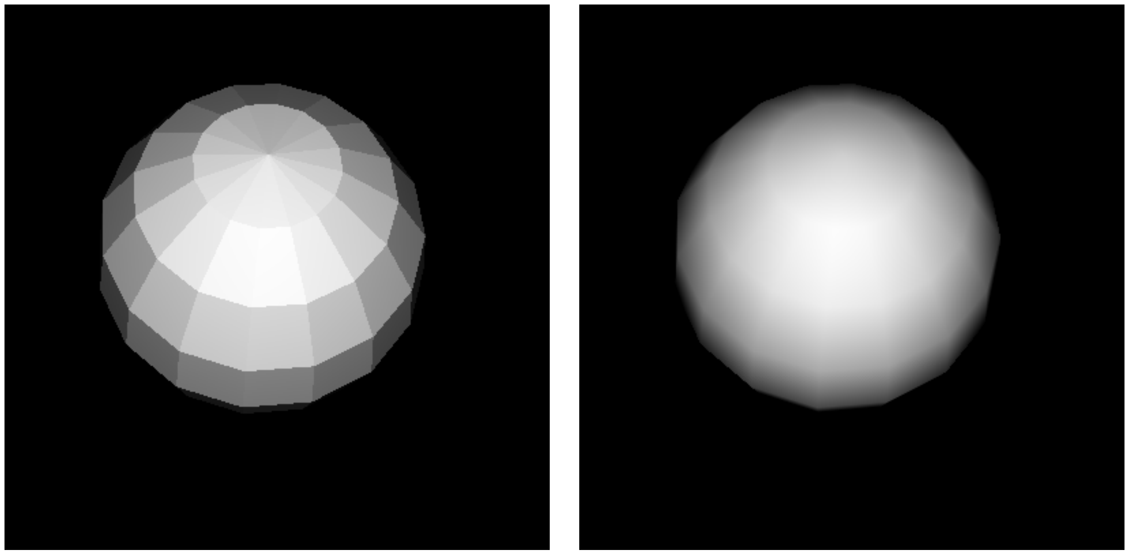

Figure 5: A tesselated sphere, without and with interpolated normals.