TU BRAUNSCHWEIG
Prof. Dr.-Ing. Marcus Magnor
Institut für Computergraphik
Contact: cgg@cg.cs.tu-bs.de

25.11.2022

Computer Graphics WS 22/23
Assignment 3

Throughout the course you will implement your own minimal raytracer. In each exercise you will extend your raytracer a little further. To make the task easier, you are provided with a basic raytracing framework so that you just have to **fill in** the missing core parts.

Please use different colors in your drawings and also make sure that formulas are recognizable in your source code. Be prepared to present the completed assignments on **Friday, 9:45** in your given time slot. To keep presentation time short, make sure that the last commit contains the original scene file which generates the results shown below.

Note that some of the pictures in this exercise do not seem physically plausible. That is intentional and will become better, when we implement some more sophisticated surface shaders.

3.1 Ambient Light (5 Points)

Instead of implementing an ambient term in each surface shader, we simulate ambient lighting by special light sources, which are always visible, regardless of the scene geometry. This is achieved by a simple trick: we set the illumination direction to the negative surface normal, thus ensuring that every surface point is treated equally by this light source. Implement a new `AmbientLight` class in `light/ambientlight.cpp` and `light/ambientlight.h` using what you already know from point lights. The files should be automatically included by cmake. If everything is implemented correctly, the image produced by cmake target `tracey_ex3_1` should look like this:

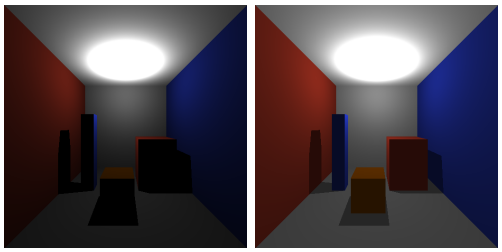

Figure 1: Left: Lighting situation 1 without ambient light. Right: With ambient light.

3.2 Spot Lights (30 Points)

Spot lights behave much like point light sources (originating from a single point, using intensity and attenuation) except that they point into a certain direction. If a point $\vec{x}$ to be illuminated is seen from the position $\vec{p}$ of the light source under an angle α with the spotlight's major direction $\vec{d}$, i.e $\alpha = acos\left(\frac{\vec{x}-\vec{p}}{\|\vec{x}-\vec{p}\|} \cdot \frac{\vec{d}}{\|\vec{d}\|}\right)$, this results in a cone of full illumination up to angle α_{min}, no illumination above angle α_{max}, and a falloff between those two angles. Here you can use a simple linear falloff. Implement a new `SpotLight` class in `light/spotlight.cpp` and `light/spotlight.h` and build the target `tracey_ex3_2`. If everything is implemented correctly, your image should look like this:

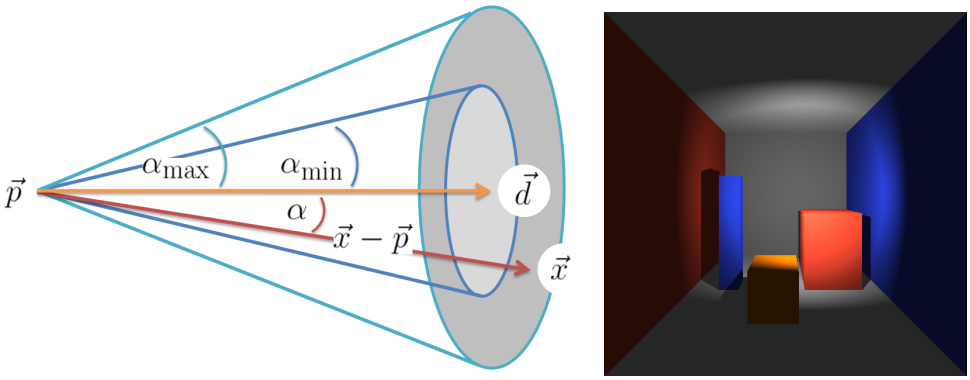

Figure 2: Left: Schematic spotlight. Right: Spotlight lighting.

3.3 Obj Model loader (45 Points)

Until now we have hard coded our scene descriptions in `ex*.cpp`. This is of course not practical. The OBJ file format is a simple ASCII file format that represents 3D geometry. It consists of the position of each vertex, the UV position of each texture coordinate, vertex normals and the faces that make up each polygon. To learn more about the format have a look at `http://paulbourke.net/dataformats/obj/`. Make yourself familiar with the file format and have a look at the file `teapot.obj`. Study how triangles are stored in the OBJ format Implement the method `Scene::loadObj` which should return a vector of triangles from a given obj file. It is not needed yet, but make sure to not only set the correct vertex position (`Triangle::setVertex`) but also the correct vertex normals (`Triangle::setNormal`) and texture coordinates (`Triangle::setSurface`). If your OBJ importer works as expected you should see teapots in your scene.

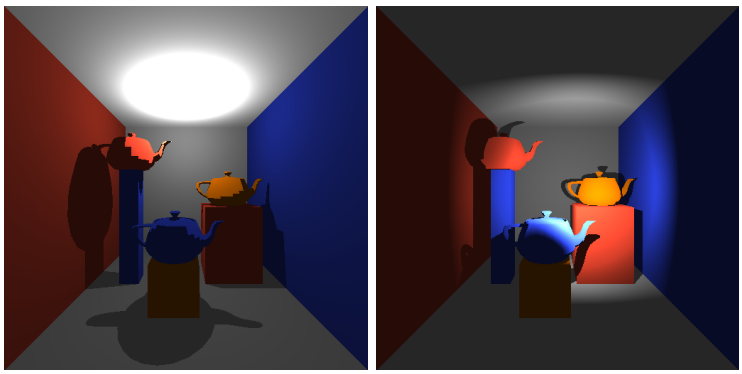

Figure 3: Scene with additional geometry in different lighting situations.

3.4 A simple speed up (20 Points)

You have noticed that your raytracer has become significantly slower due to the large number of primitives added by the mesh loader. In order to speed up computation a little, we want to introduce bounding boxes around our loaded objects. This way we can quickly check if a ray intersects the bounding box and only then test it against all the primitives in the object.

Write a new class `ObjModel` in `primitive/objmodel.h` and `primitive/objmodel.cpp`, which can load an `.obj`-File using `Scene::loadObj` and compute its axis-aligned bounding box using an instance of `primitive/box.cpp`. In `ObjModel::intersect` check for intersection with the bounding box and if a hit has been detected, check all primitives in the model for an intersection.
If you have done everything correctly, your scene should look the same, but render much faster.

3.5 *Useful Links*

- Description of the OBJ file format `http://paulbourke.net/dataformats/obj/`
- File IO `http://www.cplusplus.com/doc/tutorial/files/`