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

December 9, 2022

Computer Graphics WS 22/23
Assignment 6

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.

6.1 Bilinear Interpolation (20 Points)

You may have noticed that textures can become pixelated, especially near the camera, where one texel covers multiple pixels on the screen. To alleviate this, we will implement bilinear filtering. This allows us to linearly interpolate between texels and thus get a smoother result.
Take a look at `Texture::color(float u, float v)` in `common/texture.cpp` and implement bilinear filtering. Note that just using `CImg`s `linearAt` methods is not a valid solution for this example; you have to implement the filtering yourself. If everything is implemented correctly, you should get smooth textures.

6.2 Material Shader (5 + 5 + 15 + 10 + 15 Points)

You have already implemented various individual shaders. We now want to create a new shader that combines various techniques you have learned bevore and uses image maps for additional artistic freedom.

Diffuse Map Implement diffuse mapping in the `MaterialShader`. The idea is basically the same as in lambertian shading, but instead of having one albedo value over the whole surface one samples the diffuse map to obtain the surface color. To get the texture coordinates of the intersection point use `Ray::surface`.

Specular Map A specular map is a grayscale image that represents the specularity at each point on the object. White means glossy and black means diffuse. Implement the specularity term you already know from the phong shader and use the specular map to scale the result.

Alpha Map An alpha map describes the local opacity of the object. White means opaque and black means transparent. Implement alpha mapping in the `MaterialShader`. Remember that you only have to propagate those rays, where the object is not fully opaque. Also keep in mind that the reflection map does also propagate rays, so you cannot just change the ray passed to the function, but have to create a copy of it. The color modulation should be a linear interpolation of the current color and the propagated ray based on the local opacity value and the global scaling.

Normal Map In order to add additional detail to otherwise boring and flat objects we want to add normal maps. In a normal map each pixel color describes the normal at the corresponding position.

Note that you do not have negative colors, instead normal values are mapped to color channels in such a way that each component $[0, 1]$ in color space corresponds to $[-1, 1]$ in normal space. Additionally you will have to transform your normal into tangent space. Don't worry, most of that is already done for you. The result should finally be linearly interpolated between the original surface normal and the normal from the normal map based on the `normalCoefficient`.

Reflection Map A reflection map is a grayscale image that represents the reflective property of the surface. White means a perfect mirror, whereas black means no reflection. You already know how to implement a mirror shader, and the value in the reflection map is simply a linear interpolation factor between the color of the mirrored ray and the color of the surface. Remember that you have to use the normal map when calculating the reflected ray.

6.3 Supersampling (20 + 10 Points)

a) Until now, we have used exactly one ray per pixel, with the pixel being a rectangular area $p = a \times a$ and the ray going through center. However, as you may have already noticed this leads to aliasing artifacts (stair stepping at edges, etc.) for large ray distances. To counteract this effect, we use multiple rays per pixel and calculate the average. Subdivide the pixel into $n = s \times s$ equally sized regions, all of which are sampled once. The final pixel color is the average over all samples. Take a look at the `SimpleRenderer` and create your own `SuperRenderer` class in `renderer/superrenderer.cpp`, which allows setting s and which renders a super sampled image.

b) Create a scene in which the differences between normal sampling and super sampling becomes very obvious (aliasing artifacts). Render both images for comparison.

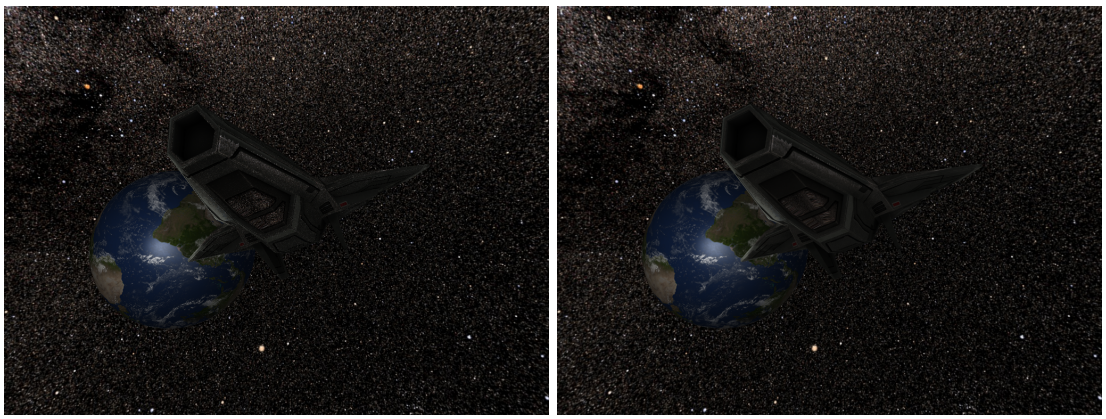
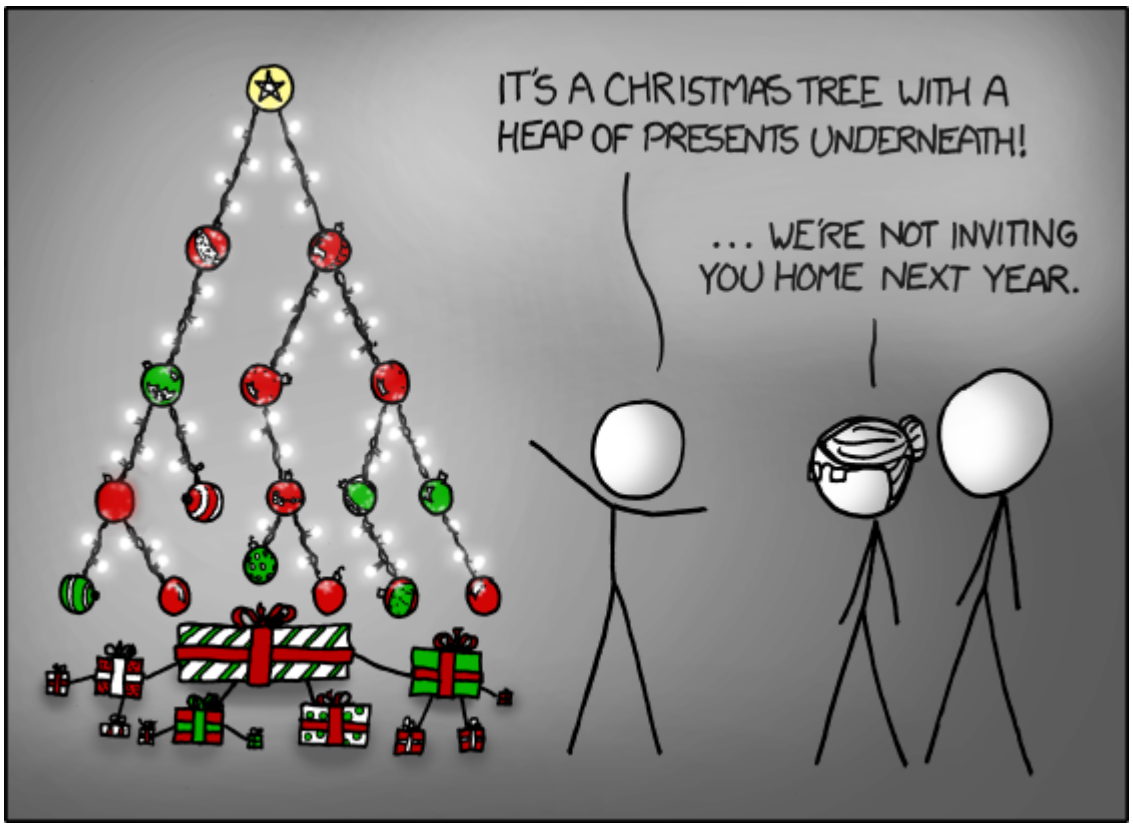


Figure 1: Result without and with 4×4 uniform super-sampling; for this scene there is very little impact on image quality.



Check the internet for free models that you can use. For example:
`http://www.turbosquid.com`
`http://www.blendswap.com`
`https://3dwarehouse.sketchup.com`