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

December 2, 2022

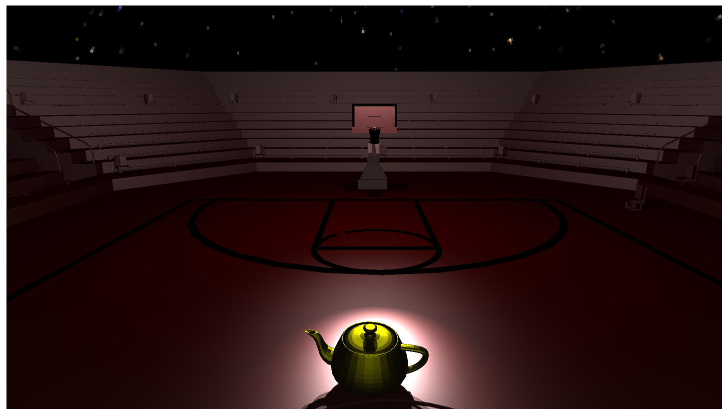
Computer Graphics WS 22/23
Assignment 5

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.

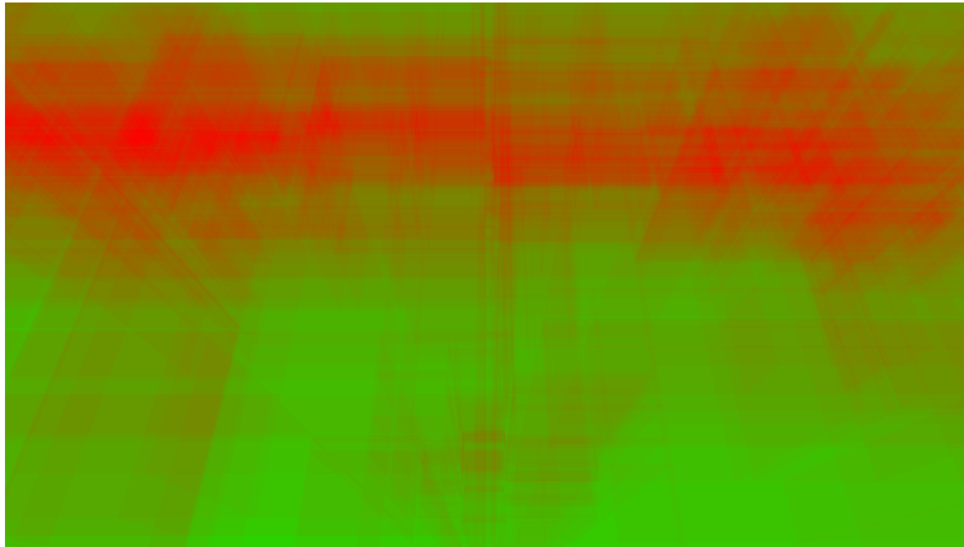
5.1 KD-Tree Acceleration (50 Points)

So far our ray tracer has only used bounding boxes for reducing the number of ray/primitive intersections. This works well for small scenes, but for larger ones we need a data structure to speed up the process of finding the first hit of a ray with the primitives. In the lecture you will learn about kd-trees (`https://en.wikipedia.org/wiki/K-d_tree`). A kd-tree is a binary search tree, which subdivides your scene space and contained primitives with axis-aligned planes (exactly one per node). This helps speed up the intersection routine by first testing the ray against this data structure and only testing against primitives in leaf nodes, which are known to be hit by a specific ray.
Have a look at `scene/fastscene.cpp` and implement the missing parts. When determining the split dimension use the dimension in which your local bounding area is the widest. When determining the split position, use the median of the minimum bounds of the primitives. If everything is implemented correctly, the result should still look like the image below but render a lot faster.



5.2 KD-Tree visualization (20 Points)

To check the correctness of your splitting, implement a new renderer in `renderer/kdtreerenderer.cpp`. This renderer should utilize the function `FastScene::countNodeIntersections` to create a map, which contains the number of splitting planes each viewing ray must pass to traverse the whole scene. Then colorize this map as follows: 0 intersections → green, maximum number of intersections → red and for in-between values interpolate between those extremes.
If you have done everything correctly your `result_kd.png` should look like this:



5.3 Parallelization (30 Points)

To further speed up your code, implement parallelization of the loop(s) in `SimpleRenderer::renderImage`.
Use the standard threading library (class `std::thread`). Think about which part of the code needs to be guarded from race conditions and solve problems by `std::mutex`es or `std::atomic`s.
Render your image again and look at the CPU utilization with `htop`. Try different access patterns for each thread (rows, batches of rows, pixel blocks, etc.) and compare execution times with the `std::chrono::steady_clock` class and Rays/s counts.

- `https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial/`
- `https://en.cppreference.com/w/cpp/header/thread`
- `https://en.cppreference.com/w/cpp/header/atomic`
- `https://en.cppreference.com/w/cpp/header/mutex`