Worksheet 1

Ray tracing is the easiest and the most general technique for visualising 3D models. It is easy to create many sophisticated lighting effects using ray tracing, but ray tracing is still slower than rasterization. Hence, ray tracing is used primarily for rendering of photorealistic images in applications where image quality rather than a high frame rate is the primary concern. The purpose of the exercises for the first weeks is to help you learn how a ray tracer works. A ray tracing framework is available on DTU Learn which provides a coding infrastructure such that you only need to implement the essential parts.

Learning Objectives

- Implement ray casting (tracing rays from eye to first surface intersection).
- Use a pinhole camera model for generating rays in a digital scene.
- Compute intersection of rays with three-dimensional primitive objects (planes, triangles, spheres).
- Compute shading of diffuse surfaces using Kepler's inverse square law and Lambert's cosine law.

Getting Started

First of all, you need to get the framework up and running. And you need to familiarise yourself with the math library and material file format used in the framework. Go through the following steps to get started.

- Compile the framework and take a screenshot of the rasterized preview of the default scene. If you are using Visual Studio, simply open the solution file called `render.sln` and press 'F5'.[1]
- Familiarise yourself with the OptiX math library using the following OptiX Documentation links: vectors, math library functions, the Ray struct, the Onb struct, the Aabb class.
  The framework includes this math library, which follows the CUDA[2] and Cg[3] syntax, to enable a smooth transition to GPU ray tracing. You need neither CUDA nor a GPU to use the framework.
- Find out how the default scene is defined by investigating the file `RenderEngine.cpp`. Note that material properties are loaded from a Wavefront MTL file.[4] Familiarise yourself with this file format. In addition, note that many rendering parameters are set in the `RenderEngine` class constructor and that the comments in the function `keyboard` document the user interface of the program.

Ray Casting

In this first exercise, your job is to implement the very basics of ray tracing which are ray generation, ray-object intersection, and shading of diffuse surfaces. This simple, non-recursive, visible-surface ray tracing is typically referred to as ray casting.

- Implement the loop over all pixels. Do this by completing the function `render` in the file `RenderEngine.cpp`. The loop over all image rows is given, you need to insert a loop over all pixels in a row and store, in the flat `image` array, the result of calling the function `compute_pixel`. Find this function in `RayCaster.cpp` and (temporarily) change the result to red (1, 0, 0). Press 'r' on the keyboard to run the ray tracing code. If your loop is correctly implemented, the render result should be all red.
- Generate rays using a pinhole camera model.[5] Do this by completing the camera implementation in the file `Camera.cpp` of the `raytrace` project. You need to implement the following three func-

[1] We recommend that you compile in Release mode unless you really need the debugging functionality for finding a problem in your code. Go to Release mode by switching from Debug to Release in the Solution Configuration dropdown menu.

[2] https://developer.nvidia.com/cuda

[3] https://developer.nvidia.com/Cg and https://www.nvidia.com/object/cg_tutorial_home.html

[4] http://paulbourke.net/dataformats/mtl/

[5] See the short lecture note "Ray Generation Using a Pinhole Camera Model" for more details. It is available on File Sharing.

tions: `set`, `get_ray_dir`, and `get_ray`.[6] Return to the `compute_pixel` function and use the `get_ray` function to generate a ray for the given pixel index. Try to output the ray direction as the pixel color. Multiply by one half and add one half to ensure that you get positive color values. The result should be a dark blue image with red increasing from left to right and green increasing from bottom to top.

- Implement conditional acceptance of an intersection. Do this by implementing the function `closest-_hit` in the file `Accelerator.cpp` (use modification of the maximum trace distance $t_{\max}$ to implement the conditional acceptance). Return to the `compute_pixel` function and insert an if-statement which returns the result of calling a shader if there is a closest hit, or the background colour if there is not. The render result should be all blue (as the background colour is blue).
- Implement ray-plane intersection, ray-sphere intersection, and ray-triangle intersection. Do this by implementing the function `intersect` in the files `Plane.cpp`, `Sphere.cpp`, and `Triangle.cpp`. For every intersection test that you implement, check that the primitive appears in your render result. Once the render result looks like the preview, press 'b' on the keyboard to store it in `out.png`.
- Shade the diffuse surfaces according to the point light in the scene. Do this by implementing the function `sample` in the file `PointLight.cpp` and the function `shade` in the file `Lambertian.cpp`. Use Kepler's inverse square law of radiation and Lambert's cosine law. Press '1' on the keyboard before rendering to use these shaders. Only trace shadow rays if you have time. This will be part of the Week 3 exercises. Store the render result.

Worksheet 1 Deliverables

Renderings of the default scene (e.g. preview, flat reflectance shading, and shading of diffuse objects). Compare preview and ray casting using the default flat reflectance shading to ensure that ray generation and ray-object intersection work as expected. Include relevant code snippets. Worksheet deliverables, such as these, should be included in your lab journal at the final hand-in.

Reading Material

The curriculum for Worksheet 1 is (105 pages)

B Chapters 1–2. *Introduction* and *Miscellaneous Math*.

B Chapters 3–5. *Raster Images*, *Ray Tracing*, and *Surface Shading*.

Supplementary reading material:

- Frisvad, J. R. *Ray Generation Using a Pinhole Camera Model*. Lecture Note, Technical University of Denmark, August 2012.
- Frisvad, J. R. *Ray-Triangle Intersection*. Lecture Note, Technical University of Denmark, July 2011.

JERF 2022

[6]Computing the field of view angle in the `set` function enables zooming by pressing 'z' and 'Z' on the keyboard.