Advanced Topics in Computer Graphics I

Universität Bonn
Institut für Informatik II
April 15, 2025

Summer term 2025
Prof. Dr. Reinhard Klein
Domenic Zingsheim

Sheet R02 - OptiX-Raycasting

Hand in your solutions via eCampus by Tue, 29.04.2025, **12:00 p.m.**. Compile your solution to the theoretical part into a single printable PDF file. For the practical part, hand in a single ZIP file containing only the exercise* folder within the src/ directory. Please refrain from sending the entire framework.

Assignment 1) Raycasting *(6Pts)*

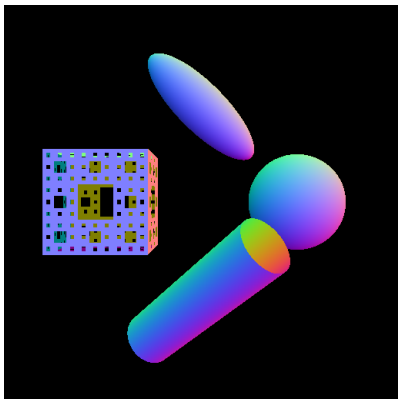

Figure 1: Expected output after implementing all steps.

In this exercise you will make yourselves familiar with the NVIDIA OptiX API for real-time raytracing, which will be the basis for future exercises. We found the following non-exhaustive list of literature on the web useful for learning about OptiX, which you might consider reading before working on this exercise:

- "How to Get Started with OptiX 7" [2] provides a nice introduction to the architecture of real-time raytracing APIs general and OptiX in particular.
- "RTX accelerated ray tracing with OptiX" [1] [3].

You might also want to have a look at the samples that ship with the OptiX SDK. Don't be afraid to look at the header files which define the classes, structs and functions of the OptiX API, they are usually accompanied by useful comments.

If you need instructions for the installation process, please refer to the 0th exercise sheet. If CMake is unable to find the OptiX SDK, you can copy the `include/` directory of the SDK and place it into the directory `external/OptiX` inside the exercise framework.

This exercise introduces a scene representation with a bunch of related classes:

- `opg/raytracingpipeline.{h,cpp}`: The `RayTracingPipeline` represents the executable shader code that is executed on the GPU. `SceneComponents` in the `Scene` add modules (compiled .cu files) and define entry points in those modules for the various shader stages.
- `opg/shaderbindingtable.{h,cpp}`: The `ShaderBindingTable` defines which hitgroups should be invoked for intersections with each `ShapeInstance` in the scene. A hitgroup contains up to three different shaders for intersection test, anyhit test and when an intersection with a shape is the closest hit on a ray. But the shader binding table also defines which shader programs to invoke if no shape in the scene was intersected by a ray (miss program) and the ray generation program (i.e. the main function of the pipeline). In addition to the shader programs directly related to tracing rays, the shader binding table can define a list of "callable" shaders which can be used to get functionality equivalent to virtual functions in C++ host code. Callable shaders can be invoked from other shader stages by specifying an index into the shader binding table.

[1] `https://drive.google.com/drive/folders/1_IYHwAZ4EoMcDmS0TknP-TZd7Cwmab_I`

- `opg/scene/scene.{h,cpp}`: The `Scene` class is responsible for managing the `RayTracingPipeline`, `ShaderBindingTable` and `SceneComponents`. It also builds and manages the bounding volume hierarchy of the whole scene, by creating an instance acceleration structure (IAS), containing transformed geometry acceleration structures (GAS) of the shapes in the scene. It's `traceRays` method is used to create a rendering of the scene contents.
- `opg/scene/sceneloader.{h,cpp}`: The `SceneLoader` is used in later exercises to load the contents of a scene from an xml file. This week we still create the scene components manually in the C++ code.
- `opg/scene/properties.{h,cpp}`: The `Properties` class is used in the creation of `SceneComponents`. Specifying a long list of parameters when calling a constructor is quite cumbersome and error-prone in C++. The `Properties` class maps strings to typed values. The arguments to the constructors of scene components are wrapped in an instance of the `Properties` class. If a property is not specified, a default value can be used. This class is also helpful for instantiating `SceneComponents` from xml files.
- Interface for scene components are specified in the `opg/scene/interface/` directory. There is usually a regular `.h` file defining the `SceneComponent`, and a `.cuh` file defining the relevant structs and methods for the device code.
  - `SceneComponent`: The base class for all scene components.
  - `BSDF` The base class for all bi-directional scattering distribution functions. BSDFs will be used in the next exercise.
  - `Emitter` The base class for all components that are responsible for emitting lights. Emitters will be used in the next exercise.
  - `RayGenerator` The ray generator is the "main" component that is responsible for actually launching the ray-tracing operations and contains the main logic.
  - `Shape` When tracing rays we are not restricted to representing geometry as triangle meshes. All geometry components derive from the `Shape` class. When tracing rays, a bounding volume hierarchy of the scene is traversed. The Shape components are responsible for providing an acceleration structure (AST) handle of the contained geometry for this purpose.
- Common scene components are defined in the `opg/scene/components/` directory. There is usually a regular `.h` and `.cpp` file defining the `SceneComponent`, and a `.cuh` file defining the relevant structs and methods for the device code, as well as a `.cu` file containing the device code with entry points for the different functionality of the component.
  - `BufferComponent`: This is a basic helper-component that simply owns a `DeviceBuffer`.
  - `BufferViewComponent`: This is a basic helper-component that simply wraps a `BufferView` into a `BufferComponent`.
  - `ShapeInstance`: The shape instance is a central component type, since it is responsible for instantiating `Shapes`, placing them in the scene, and assigning a `BSDF` and `Emitter`. We won't use `BSDFs` and `Emitters` just yet.
  - `Camera`: This component specifies a camera view into the scene, that is used by the `RayGenerator` to initialize rays for the output image.

This exercise consists of three `Shape` scene components (`Mesh`, `Sphere` and `Cylinder`), as well as the `RayCastingRayGenerator`, a ray generator that simply casts rays and outputs the surface normal to the output image. For each component the respective `.h`, `.cpp`, `.cuh` and `.cu` files can be found in `src/exercise02_Raycasting`.

a) *Generate Camera Rays.* The `__raygen__main` shader program in `raycastingraygenerator.cu` is responsible for generating rays for each pixel of the output image. Use the information provided in the `params.camera` property to cast rays through a perspective camera in the *ray*

generation program. If done successfully, you will be able to move the camera around using the left and right mouse buttons in the GUI.

2 Pts

b) *Add Scene Object.* Create and instantiate a new `Sphere` shape in the `createSceneGeometry()` function in the `main.cpp` file. Set the parameters such that the object represents an ellipsoid with radii $(0.25, 0.75, 0.25)$ along its x-, y- and z-axis, respectively. Rotate the ellipsoid by $\frac{\pi}{4}$ around the z-axis and translate it to the position $(0, 1, 0)$.

1 Pts

c) *Cylinder Intersection.* The OptiX API does not only support standard triangle meshes, but also custom primitives. For custom primitives the ray-primitive intersection has to be implemented in an *intersection* program. In the bounding volume hierarchy of the scene, the custom primitive is represented as a generic AABB. Analogous to the `Sphere` shape that is already implemented, add an implementation for a cylinder primitive by completing the following steps:

(i) Implement the `__intersection__cylinder` method for the intersection test in the `cylinder.cu` file.

(ii) Implement the `__closesthit__cylinder` method which sets the surface normal, tangent and uv coordinate of a hit in the `cylinder.cu` file.

(iii) Instantiate a new cylinder scene object in the `createSceneGeometry()` function in the `main.cpp` file.

2 Pts

d) *Discard Intersections using Anyhit Program.* Finally, we want to cutout some parts of the cube's surface in the `__anyhit__mesh` shader program defined in `mesh.cu`. The anyhit program in general is used to discard some ray-primitive intersections, for example due to transparency. Cutout the holes of a level 3 Menger sponge given the UV coordinates in $[0, 1]^2$ on the face of a cube. Note that this does not yield a true level 3 Menger sponge in the rendering, but only the outermost surface elements.

1 Pts

Theoretical Assignment

Assignment 2) Ray-Sphere intersecion *(3Pts)*

Derive the formula for ray-sphere intersection formally. The ray-sphere intersection is used in the practical part of this exercise.

Assignment 3) Ray-Triangle intersection *(2Pts)*

In the lecture, an efficient algorithm for ray triangle intersection is presented. There are, however, more methods to compute ray-triangle intersections. One is based on barycentric coordinates [4]. The other method [1] transforms the ray from the global coordinate system into a second system where the triangle is placed in the XY plane at (0,0) (1,0) (0,1) and the Z axis is parallel to the ray direction. Write down the algorithms formally and compare them with regard to numerical stability and speed.

References

[1] Tomas Möller and Ben Trumbore. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, page 7. ACM, 2005. URL: `https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf`.

[2] Keith Morley. How to get started with optix 7. https://devblogs.nvidia.com/how-to-get-started-with-optix-7/, 2019. Accessed: 2020-05-04.

[3] Ingo Wald and Steven G Parker. Rtx accelerated ray tracing with optix. In *ACM SIGGRAPH 2019 Courses*, pages 1–1. 2019.

[4] WIKIPEDIA. Barycentric coordinates on triangles. http://en.wikipedia.org/wiki/Barycentric_coordinate_system#Barycentric_coordinates_on_triangles, 2016. Accessed: 2016-05-05.

Good luck!