Interactive Computer Graphics
Coursework – Task 7 (*assessed)

Bernhard Kainz, Antoine Toisoul, and Daniel Rueckert

March 5, 2017

Task 7: GPU ray tracing

Task 7a: Simple GPU ray tracing *

In this exercise you will implement a very simple ray tracer. Since the render to texture shader as used in Task 6 provides already a rather static camera setup for rendering a screen aligned textured quad, you can use this camera also to virtually shoot rays into a scene. However, since efficient ray tracing usually requires space partitioning for polyhedral geometry, we simplify the test scene in this exercise to objects that can be described mathematically: *planes* and *spheres*. Note that you will overwrite the per-polygon shaders in this exercise and that they will become useless in the pipeline. You can therefore deactivate them. For this task we provide an example scene and basic shader setup in raytracing.xml, which can be downloaded from CATE. In this example, the render-to-texture 2D (R2T) vertex shader already defines positions and for rays in camera direction for a 16:10 aspect ratio and the R2T Fragment Shader defines a simple scene. Without any implementation the render window will show a statically black render window.

Your task is to implement the ray tracing algorithms in the R2T Fragment Shader. Therefore, you will need to follow every ray until it reaches a maximum ray tracing depth. This value could be for example 42.

For simple ray tracing you may test every ray for an intersection with every object in the scene until the ray does not hit any of the objects or until it reaches the maximum ray-trace depth. Objects can be defined by using a mathematical definition of spheres and a plane as done in the example xml pipeline.

You should also compute shadows by using additional *shadow rays*. You can do this calculation in a separate `computeShadow(in Intersection intersect)` function. When computing shadow rays, you should slightly move the ray origin outwards of the object along the surface normal or alter the ray direction slightly using the pseudo random number generator provided in `rnd()` to avoid numerical problems.

The objects in the scene are defined by their intersect functions, which you can implement as for example

`shpere_intersect(Sphere sph, Ray ray, inout Intersection intersect)`

and

`plane_intersect(Plane pl, Ray ray, inout Intersection intersect)`.

The plane intersection should additionally vary the ray hit color, so that a

checkerboard pattern results.

When your ray tracer is ready you should be able to produce an image similar to Figure 1.

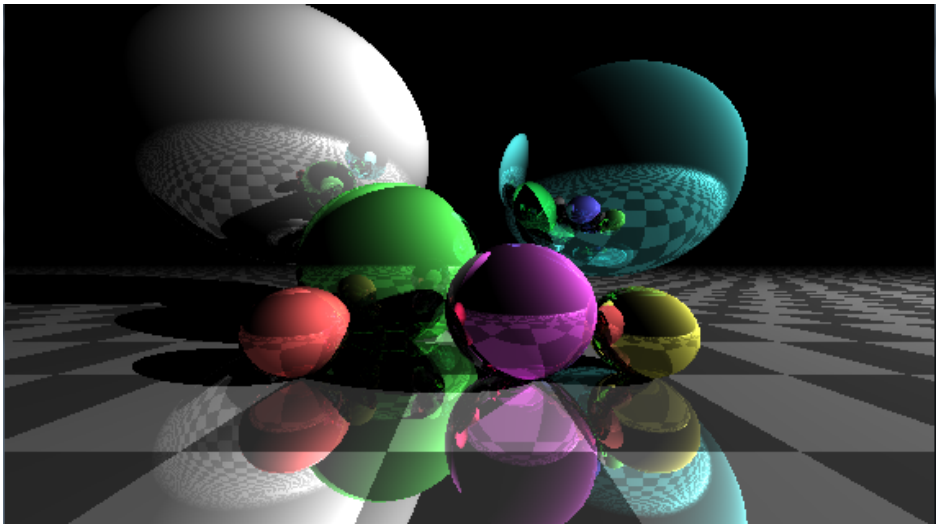


Figure 1: Example result from simple geometric ray tracing.

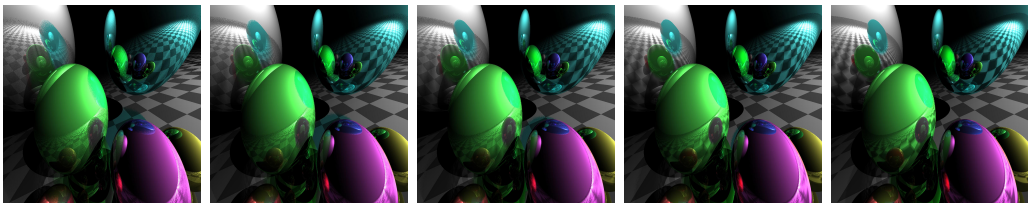
You should also implement mouse based scene interaction as you have been using it throughout the exercises. You can do this with the `uniform` matrices `pMatrix` and `mvMatrixScene`. These matrices will change when you use the mouse interaction scheme from as used in the framework. You may use them to *either* manipulate the initial ray direction *or* to alter the object's position. Think about the smarter option and make the right choice!

If you implemented the above described simple ray-tracer with shadow-rays and mouse based interaction successfully you will gain 75% of the maximum points for this task. The remaining 25% can be achieved by extending this exercise with your own ray-tracing extensions. For example you could implement transparent and refracting objects, light scattering effects, caustics, soft shadows, and many more. It is up to you which extension you choose!

Task 7b: Simple Monte-Carlo Path tracing

Monte-Carlo Path tracing is used to simulate global illumination. The algorithm aims to integrate over *all* the illuminance arriving to a single point on the surface of an object. A simple approximation of the algorithm can be achieved by sending several, slightly tilted rays instead of a single ray into the scene and to split them into more than one secondary ray following a random direction at each intersection point. This is an open ended task and you can implement as many path tracing features as you like. Be aware of the limitations of the used hardware. Processing many rays may quickly exhaust your GPU, which might lead to a crash of the rendering system.

Figure 2 shows an example for Monte-Carlo Path tracing with different numbers of secondary path rays.



(a) 5 path rays (b) 10 path rays (c) 15 path ray (d) 30 path rays (e) 50 path rays

Figure 2: Example result from Monte-Carlo ray tracing with a different number of Monte-Carlo rays.

HAVE A LOT OF FUN!!