UNIVERSITÄT DES SAARLANDES
Prof. Dr.-Ing. Philipp Slusallek
Computer Graphics Group
Stefan Lemme (lemme@cg.uni-saarland.de)

5. November 2018

Introduction to Computer Graphics
Assignment 2

Submission deadline for the exercises: 12. November 2018

The paper copies for the theoretical parts of the assignments will be collected at the beginning of the lecture on the due date. The programming parts must instead be sent by email to lemme@cg.uni-saarland.de before the beginning of the lecture on the due date.
The code submitted for the programming part of the assignments is required to reproduce the provided reference images. The submission ought to include a reference to the respective commit in your group's git repository as well as the mandatory generated images. The submission should also contain a creative image show-casing all extra-credit features that have been implemented.
The projects are expected to compile and work out of the box. A successful build by Drone CI is a good indicator. If it fails to do so on our Windows or Linux computers, we will try to do so on the CIP-pool students' lab as a "fallback".
To pass the course you need for every assignment at least 50% of the points.

2.1 Triangle properties (2 + 4 Points)

A triangle T is defined by its 3 vertices P_1, P_2, P_3.

- Compute the barycentric coordinates of the center of the mass of T.
- Compute the barycentric coordinates of the incenter of T (center of the inscribed circle)

2.2 Infinite plane representation (5 + 5 Points)

On the lecture we define an infinite plane by an arbitrary point a on the plane and a normal n. This is however not the only way to define a plane.
Derive the values a and n when the plane is defined as:

- A set of points (x, y, z) satisfying equation $Ax + By + Cz + D = 0$
- As a set of 3 points p_1, p_2, p_3 lying on the plane defined in counter-clockwise order with respect to intended suface normal direction.

Note that the same plane can be defined by several positions of a, but there exists only one n.

2.3 Ray-Surface Intersection (7 + 13 Points)

Given a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ with origin $\mathbf{o} = (o_x, o_y, o_z)$ and direction $\mathbf{d} = (d_x, d_y, d_z)$, derive the equations to compute the parameter t for the intersection point(s) of the ray and the following surfaces:

a) An infinite plane defined by three points p_1, p_2, p_3 lying on the plane

b) A sphere, specified by $(a_x - C_x)^2 + (a_y - C_y)^2 + (a_z - C_z)^2 = R^2$ (or $(\mathbf{a} - \mathbf{C})^2 = R^2$) with center $\mathbf{C} = (C_x, C_y, C_z)$ and radius $R \in \mathcal{R}$.

2.4 Ray Quadric Intersection (15 + 5 + 20 Points)*

Given a ray $R(t) = O + t \cdot D$ and quadric $ax^2 + by^2 + cz^2 + dxy + exz + fyz + gx + hy + iz + j = 0$.

a) Compute the values t for which the ray intersects the quadric.

b) Derive the ray-sphere intersection formula from it, as a special case.

c) Implement a quadric `Solid` in the framework.

2.5 The base of the ray tracer (30 Points)

So far we have been shooting rays into space, but we did not have any objects to intersect them with. In this assignment we introduce basic concepts and primitives to enable this. We ask you to implement the methods of following classes based on the provided header files: RayCastingIntegrator, SimpleGroup, Intersection and Solid. Additionaly, we ask you to implement the method Renderer::render which should call Integrator::getRadiance. This is the foundation for Exercise 2.5 and 2.6 in which we ask you to render images based on intersections of rays with solids.

- `Primitive` represents any geometrical body (may be arbitrary complex) that a ray may try to intersect. Each primitive provides:
  - `getBounds()` which will return a bounding box encapsulating all the geometry of the primitive. At this time however you don't need to implement it as returned class `BBox` is nowhere defined. For this not implemented functionality, you can use the macro `NOT_IMPLEMENTED` within the body of the function.
  - `intersect(ray,previousBestDistance)` finds a ray parameter t of the input ray representing the intersection point between the ray and the primitive. The value t must be between 0 and `previousBestDistance`. If intersection happens only beyond this range, it should not be reported.
  - `setMaterial , setCoordMapper` will be used in the future to set the material for the primitive; at the moment however we are not using materials so the functions can do nothing as they are not being called yet. (`NOT_IMPLEMENTED`)
- `Solid` is a special case of a `Primitive` representing a single geometric object. A surface holds a single material and coordinate-mapper (currently ignored). In addition to the functionality provided in `Primitive`, it also provides:
  - Information of the surface area of the solid (`getArea`)
  - Surface sampler (`sample`) which will be used in the future to sample a random point on the surface. At the moment however we are not discussing sampling and this function should do nothing (`NOT_IMPLEMENTED`).
- `Group` is a different `Primitive`, that may hold multiple (sub)primitives. A group may be a simple container, but may additionally do some indexing to speed up the process of ray tracing. The inherited `intersect` function is expected to find the nearest intersection point of all contained primitives. In addition, it provides:
  - `add` method, allowing a new primitive to be added to the scene. You can assume that the added primitive is not yet part of the group and no cycles are created.
  - `rebuildIndex` is called once after the primitives are added, permitting the indexing structure (if it exists) to rebuild itself.
- `Intersection` helper class holds the information about ray-primitive intersection. The class holds:
  - The `distance` parameter t at which the intersection occurred.
  - The `ray` which was used for the intersection.
  - The `solid` primitive object that was hit by the ray.

- The `normal` vector of the object at the hit position.
- The `local` , object-space coordinates of the hit point. The nature of the local coordinate system may depend on the type of the object that was hit. It most cases, you consider the cartesian coordinates with respect to some pivoting point of the object. For some solids however, you want a very specific coordinate system, e.g. a `Triangle` would yield barycentric coordinates.

In addition the `Intersection` provides:

- Named constructor (static method returning a new object) `failure` producing an `Intersection` object indicating that no intersection occurred.
- `operator bool` cast operator which returns `false` only when the object indicates no intersection.

- `Integrator` is a class that computes the amount of incoming light in a given direction. To achieve this goal, the integrator may shoot and intersect rays with the scene, or perform some other arbitrary computation. **Only in the integrator may the intersection process begin**.

  Upon construction, the `Integrator` takes the `World` object for which it computes the lighting.

- `World` is a class representing the whole scene with all its objects and light sources. In the current assignment though, lights are ignored.

Apart from the above constructs, we ask you to implement simple realization of these:

- `SimpleGroup` should be a simple container without any indexing structure. The intersection process shall iterate over all members.
- `RayCastingIntegrator`, as a simple example of an `Integrator` intersects an input ray with the scene, and if the hit is successful should return a gray color whose value is the dot product between the ray direction and the hit surface normal.

2.6 Basic Integrators (10 Points)

Implement `RayCastingDistIntegrator` which would be a new kind of integrator that would take the hit distance into an account. The color of the pixel is an interpolated value between `nearColor` and `farColor` (provided in the constructor) depending on the distance value in relation to `nearDist` and `farDist` parameters. The resulting color should be the product between this color and the cosine value of the ray direction and the hit surface normal.

2.7 Solids (2 + 3 + 7 + 5 + 9 + 9 Points)

We ask you to implement the intersection methods for the following classes of solids and run the provided function a_solids() to produce the images shown in Fig. 1:

- `InfinitePlane`
- `AABox` — An axis aligned box defined at construction time by two opposing corners.
- `Sphere`
- `Disc`
- `Triangle`
- `Quad` — A parallelogram defined by one vertex and two spanning vectors.

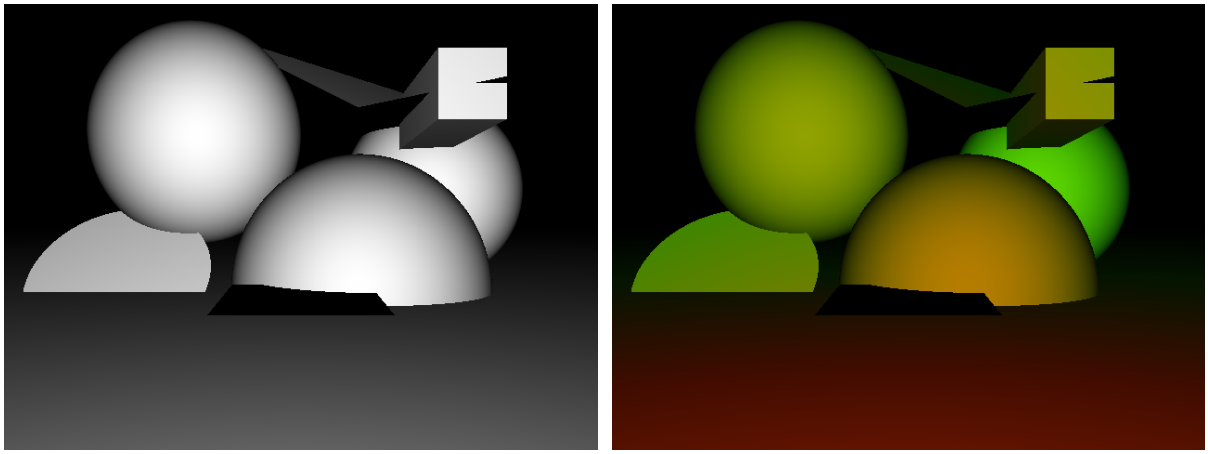

Figure 1: Final images produced in the Assignment 2 with all the solids. Left: result using RayCastIntegrator. Right: result using RayCastingDistIntegrator.