Universität des Saarlandes
Prof. Dr.-Ing. Philipp Slusallek
Computer Graphics Group
Hugo Devillers (devillers@cg.uni-saarland.de)
Ömercan Yazici (yazici@cg.uni-saarland.de)

19. November 2020

Introduction to Computer Graphics
Assignment 2

Submission deadline for the exercises: 26. November 2020

MacOS build issues Users of modern versions of MacOS have reported trouble with building the framework, epecially getting the `invalid operands to binary expression` error. We have established the issue comes from Apple Clang defining the `assert` macro. We'd like you to add this snipet in the `core/asser.h` file at line 40:

```
#ifdef __APPLE__
#undef assert
#endif
```

This should fix both issues for students, but also tutors who are using Macs for correcting.

Clarifications The question of handwritten solutions has come up, our policy is to allow it, but we recommend students to type in their solution (for example in LaTeX) as it removes the handwriting as a variable. If you can, we ask you to do so, otherwise be careful about your handwriting legibility. In particular, avoid cursive.
Additionally, we'd like you to provide some additional information for the practical part in a README file, containing information such as your development platform (OS, compiler, versions, ...), and what features you implemented/skipped.

The role of tests Finally, we'd like to clarify the role of tests. Passing all tests is neither a necesary nor a sufficient indicator of getting the best grade. Tests are by nature not able to capture all possible variants of implementation, and some of them can lead to false positives or negatives. Additionally not every point is covered by a test. So all in all, and much like in any software project, please view tests for what they are: a way for you and us to quickly evaluate *roughly* how correct something is, and to check for some specific things. You should generally not favour test compliance over matching the sample images, or having code that is expresses things reasonnably clearly. Generally messy code is the result of a messy mental model.

The theoretical parts should be uploaded as a single PDF (scan, LaTeX, ...) in the assignment in Microsoft Teams. Please write your name, the name of your partner and the group name (e.g. group42) on top of the sheet. Both of the team members have to upload and submit the same PDF!
For guest students or those who have trouble with Teams, we can exceptionally accept submissions by email, in which case please send them by email to Hugo.
The programming parts must be marked as release 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 tag the respective commit in your group's git repository as well as attach the mandatory generated images to the release. The submission should also contain a creative image show-casing all extra-credit features that have been implemented, if applicable for the assignment.
The projects are expected to compile and work out of the box. A successful build by Drone CI is a good indicator.
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, which is the center of the inscribed circle. We define the inscribed circle as the circle whose center is equidistant to each edge of T.

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:

- The set of all points $p = (x, y, z) \in \mathrm{R}^3$ satisfying equation $Ax+By+Cz+D = 0$ with $A, B, C, D \in \mathrm{R}$ The constructed normal n is unique except for its sign.
- 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.
  In contary to the first subtask, there exists only one n this time.

Note that the same plane can be defined by several positions of a.

2.3 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.

2.4 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. For triangles/quads, the local coordinates are the barycentric coordinates. For all other solids considered in the course, they are equal to the cartesian coordinates of the hit point.

  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.

2.5 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.

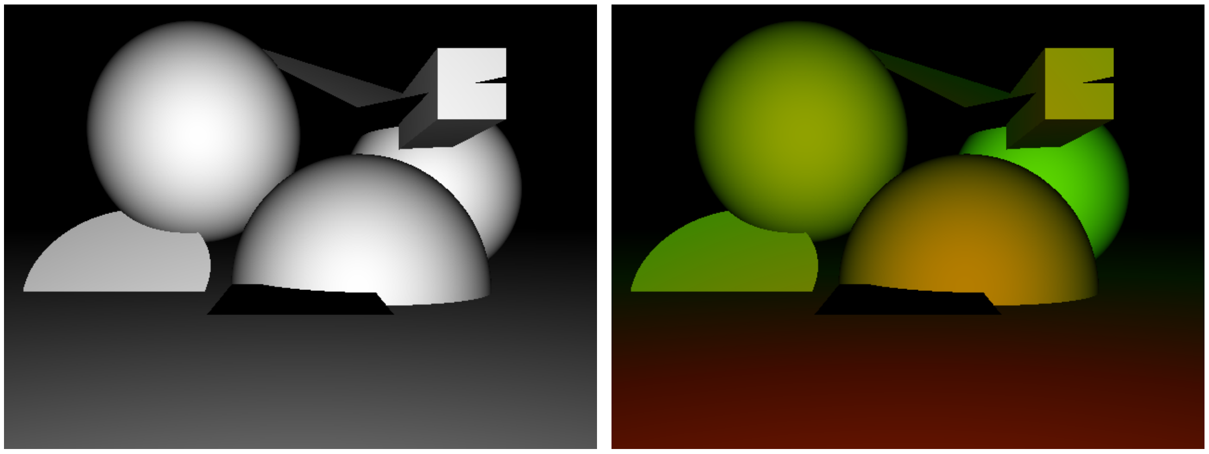

Figure 1: Final images produced in the Assignment 2 with all the solids. Left: result using RayCastIntegrator. Right: result using RayCastingDistIntegrator.

2.6 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.