# (Task 2) Intersecting Primitives

Nick Sharp edited this page 20 days ago · 4 revisions

Now that your ray tracer generates camera rays, we need to be able to answer the core query in ray tracing: "does this ray interesect this object". Here, you will start by implementing ray-primitive intersection routines against the two primitives in the starter code: triangles and spheres. Later, we will use a BVH to accelerate these queries, but for now we consider an intersection test against a single primitive.

The `Primitive` interface, which implemented by both the `triangle` and `sphere` class, contains two types of intersection routines:

- `bool Triangle::intersect(const Ray& r)` returns true/false depending on whether ray `r` hits the triangle.
- `bool Triangle::intersect(const Ray& r, Intersection *isect)` returns true/false depending on whether ray `r` hits the triangle, but also populates an `Intersection` structure with information describing the surface at the point of the hit.

You will need to implement both of these routines. Correctly doing so requires you to understand the fields in the `Ray` structure defined in `ray.h`.

- `Ray.o` represents the 3D point of origin of the ray
- `Ray.d` represents the 3D direction of the ray (this direction will be normalized)
- `Ray.min_t` and `Ray.max_t` correspond to the minimum and maximum points on the ray. That is, intersections that lie outside the `Ray.min_t` and `Ray.max_t` range **should not** be considered valid intersections with the primitive.

There are also two additional fields in the `Ray` structure that can be helpful in accelerating your intersection computations with bounding boxes (see the `BBox` class in `bbox.h`). You may or may not find these precomputed values helpful in your computations.

- `Ray.inv_d` is a vector holding (1/x, 1/d.y, 1/d.z)
- `Ray.sign[3]` hold indicators of the sign of each component of the ray's direction.

One important detail of the `Ray` structure is that `min_t` and `max_t` are `mutable` fields of the `Ray`. This means that these fields can be modified by constant member functions such as `Triangle::Intersect()`. When finding the first intersection of a ray and the scene, you almost certainly want to update the ray's `max_t` value after finding hits with scene geometry. By bounding the ray as tightly as possible, your ray tracer will be able to avoid unnecessary tests with scene geometry that is known to not be able to result in a closest hit, resulting in higher performance.

### Step 1: Intersecting Triangles

While faster implementations are possible, we recommend you implement ray-triangle intersection using the method described in the lecture slides. Further details of implementing this method efficiently are given in these notes.

There are two important details you should be aware of about intersection:

- When finding the first-hit intersection with a triangle, you need to fill in the `Intersection` structure with details of the hit. The structure should be initialized with:

    - `t` : the ray's t-value of the hit point
    - `n` : the normal of the surface at the hit point. This normal should be the interpolated normal (obtained via interpolation of the per-vertex normals according to the barycentric coordinates of the hit point)
    - `primitive` : a pointer to the primitive that was hit
    - `bsdf` : a pointer to the surface brdf at the hit point (obtained via `mesh->get_bsdf()` )
- When intersection occurs with the back-face of a triangle (the side of the triangle opposite the direction of the normal) you should **return the normal of triangle pointing away from the side of the triangle that was hit.**

Once you've successfully implemented triangle intersection, you will be able to render many of the scenes in the media directory. However, your ray tracer will be **very slow!**

#### Step 2: Intersecting Spheres

Please also implement the intersection routines for the `Sphere` class in `sphere.cpp` . Remember that your intersection tests should respect the ray's `min_t` and `max_t` values.

**Clone this wiki locally**

```
https://github.com/cmu462
```