

Geometric classes

Geometric classes represent objects in three-dimensional space. There are three board geometric classes: **Point**, **Vector**, and **Shape**. The first is pretty self-explanatory; it represents a point. **Vector** and **Shape** are more detailed classes and are explained below.

Vector

The **Vector** class represents any 2-D or 3-D vector quantity. It also supports vector arithmetics, including dot and cross products. The `dot()` function is also used to define the L2-norm. All **Vectors** are explicitly convertible to **bools** - if a **Vector** does not have any NaN value, then it can be converted to **true**. This can be used to check whether a **Vector** has any NaN value, similar to how a pointer can be checked if it's a **nullptr**. A derived class called **MutableVector** can be used to modify any of the coordinates, though it's not sure whether it will be useful. The more important derived class is the **Direction** class, which dictates that the L2-norm be 1. Any non-zero **Vector** can be converted into a non-NaN **Direction**.

Shape

The abstract **Shape** class provides a template for volumetric objects. All classes derived from **Shape** must implement the following functions in order to be instantiated:

1. `xMin()`, `xMax()`, and so on: locate the extremum points in each of the three coordinates.
2. `surfaceArea()`
3. `volume()`
4. `surfaceContains(const Point&)`: checks whether the **Point** is strictly on the surface.
5. `encloses(const Point&)`: checks whether the **Point** is strictly inside ***this** (i.e. not on the surface or outside).
6. `encloses(const Shape&)`: checks whether the other **Shape** is completely contained within ***this**. This function evaluates to **true** even if the two **Shapes** intersect on the surface of ***this**.
7. `overlaps(const Shape&)`: checks whether the two **Shapes** overlap (i.e. the region of intersection is non-zero in volume).
8. `contentsOverlap(const Shape&)`: checks whether the other **Shape** overlap with any **Shape** contained within ***this**. This function may only differ from `overlaps()` if ***this** is of type **BoundingBox** (see below).

9. `distanceToSurface(const Point&, const Direction&)`: calculates the smallest, positive distance that a point has to travel to reach the surface.
 - If the point is inside the shape, the distance equation will have positive and negative roots, and only the smallest positive root is returned.
 - If the point is on the surface, returns 0 if it leaves the surface, and the distance to the other end if it enters the surface.
 - If the point is outside the shape, returns NAN if it never enters the shape, else the smallest distance to the surface.
10. `normal(const Point&)`: checks if the point is on the surface, and returns the outward normal vector if it is.
11. `print(std::ostream& os)`: outputs the shape into `std::cout`.

Additionally, `Shape` also has two member variables that represent the index of refraction and the total macroscopic cross section, respectively. They are both `doubles` and therefore assumed to be constant. In the future, it might make sense to have them as a separate class (let's say, `Property`), in order to capture their respective dependencies on various parameters.

Right now, there are two implemented derived classes of `Shape`: `Sphere` and `Box`. `Sphere` requires a `Point` representing the origin and a radius. `Box` requires two `Point` representing the two opposing vertices. The `Sphere` class is our current representation of droplet particles, which has the advantage of easy implementations of the aforementioned functions. A more accurate geometric class may be implemented in the future to account for the effects of surface tension and gravity. The `Box` class probably will never be used to represent physical objects, but rather to construct a derived `BoundingBox` class. In this context, a `Box` is not a *terminal shape*, but a `Sphere` is.

Tree classes

The motive for using a Tree data structure is to quickly locate which —Shape— object (potentially out of thousands or millions) that a ray of photon will enter next, which is our version of the nearest neighbor search algorithm. A brute-force approach will have a time-complexity of order $\mathcal{O}(N)$, which in itself is not ideal. Furthermore, distance calculations are expensive because of the use of the square-root function (in case of the `Sphere` class) or some other functions that are potentially more expensive. A tree is a hierarchical data structure that groups objects based on certain common features. In this case, an Octree is used to group 3D shapes into each of the 8 octants of equal volume. Using a tree structure, the average time-complexity is down to $\mathcal{O}(\log N)$, which scales extremely well for large datasets. The majority of distance calculations involving those octants are to a surface of a box, and therefore are linear in nature, leaving a very few expensive calculations at the end.

BoundingBox

The **BoundingBox** class is derived from **Box** and represents an axis-aligned bounding box (AABB). It is the building block of an **Octree** and therefore is constrained to have at most 8 children. A **Node** is defined as a pointer to a **Shape** - more specifically, a `std::unique_ptr<Shape>` - that is contained within a **BoundingBox**. The pointed-to **Shape** itself can be either another **BoundingBox** (which means there are further subdivisions within that particular octant) or a terminal shape (which means they are leaf nodes).

Octree

Snell's Law and Orientation of the Transmitted Ray

Let $\hat{\mathbf{i}}$ be the incident vector, $\hat{\mathbf{n}}$ be the (outward) normal vector, and $\hat{\mathbf{t}}$ be the transmitted vector. Also the index of refraction of a medium is labeled n .

Our job is to find $\hat{\mathbf{t}}$, given $\hat{\mathbf{i}}$, $\hat{\mathbf{n}}$, n_1 and n_2 . The vectors are related through Snell's Law, and in vector notation it reads

$$n_1 (\hat{\mathbf{i}} \times \hat{\mathbf{n}}) = n_2 (\hat{\mathbf{t}} \times \hat{\mathbf{n}}), \quad (1)$$

or

$$\hat{\mathbf{t}} \times \hat{\mathbf{n}} = \mathbf{c}, \quad (2)$$

for the new vector \mathbf{c} defined as $\mathbf{c} = \frac{n_1}{n_2} (\hat{\mathbf{i}} \times \hat{\mathbf{n}})$. Note that \mathbf{c} and $\hat{\mathbf{n}}$ are orthogonal.

We start solving for $\hat{\mathbf{t}}$ by decomposing it into two components: one parallel and one orthogonal to $\hat{\mathbf{n}}$:

$$\hat{\mathbf{t}} = t_{\parallel} \hat{\mathbf{n}} + t_{\perp} \hat{\mathbf{p}} \quad (3)$$

Since the cross product of the parallel component and $\hat{\mathbf{n}}$ is the zero vector, we require the cross product of the orthogonal component and $\hat{\mathbf{n}}$ be equal to \mathbf{c} .

$$t_{\perp} \hat{\mathbf{p}} \times \hat{\mathbf{n}} = \mathbf{c} \quad (4)$$

Since \mathbf{c} and $\hat{\mathbf{n}}$ are already orthogonal, one of the possible values for the unit vector $\hat{\mathbf{p}}$ is $\hat{\mathbf{p}} = \hat{\mathbf{n}} \times \hat{\mathbf{c}}$ (the other possibility is its additive inverse). Note that the unit vector $\hat{\mathbf{c}}$ is \mathbf{c} normalized to unit length. Using the vector triple product identity $\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \cdot \mathbf{c})\mathbf{b} - (\mathbf{a} \cdot \mathbf{b})\mathbf{c}$, it can be showed that

$$\begin{aligned} \hat{\mathbf{p}} \times \hat{\mathbf{n}} &= (\hat{\mathbf{n}} \times \hat{\mathbf{c}}) \times \hat{\mathbf{n}} \\ &= \hat{\mathbf{n}} \times (\hat{\mathbf{c}} \times \hat{\mathbf{n}}) \\ &= (\hat{\mathbf{n}} \cdot \hat{\mathbf{n}})\hat{\mathbf{c}} - (\hat{\mathbf{n}} \cdot \hat{\mathbf{c}})\hat{\mathbf{n}} \\ &= 1\hat{\mathbf{c}} - 0\hat{\mathbf{n}} \\ \hat{\mathbf{p}} \times \hat{\mathbf{n}} &= \hat{\mathbf{c}} \end{aligned} \quad (5)$$

Substituting the above expression into Equation 4, we arrive at the expression for t_{\perp} :

$$\begin{aligned} t_{\perp} \hat{\mathbf{p}} \times \hat{\mathbf{n}} &= \mathbf{c} \\ t_{\perp} \hat{\mathbf{c}} &= \mathbf{c} \\ t_{\perp} &= \|\mathbf{c}\| \end{aligned} \tag{6}$$

The parallel component of $\hat{\mathbf{t}}$ does not contribute to the cross product, so theoretically t_{\parallel} can assume any value and Equation 2 will hold true. However, since $\hat{\mathbf{t}}$ is a unit vector, we require that

$$\begin{aligned} t_{\parallel}^2 + t_{\perp}^2 &= 1 \\ t_{\parallel} &= \pm \sqrt{1 - t_{\perp}^2} \end{aligned}$$

The choice of sign on t_{\parallel} now depends on the orientation of the incident vector $\hat{\mathbf{i}}$. When the incident ray **enters** a surface, the transmitted vector $\hat{\mathbf{t}}$ will point inwards, away from the normal vector. Conversely, when the incident ray **exits** the surface, $\hat{\mathbf{t}}$ will point outwards. Therefore,

$$t_{\parallel} = \begin{cases} -\sqrt{1 - t_{\perp}^2} & , \hat{\mathbf{i}} \cdot \hat{\mathbf{n}} < 0 \\ \sqrt{1 - t_{\perp}^2} & , \hat{\mathbf{i}} \cdot \hat{\mathbf{n}} > 0 \end{cases}$$

Therefore, the final form of the transmitted vector $\hat{\mathbf{t}}$ is

$$\begin{aligned} \hat{\mathbf{t}} &= \|\mathbf{c}\| (\hat{\mathbf{n}} \times \hat{\mathbf{c}}) + \text{sgn}(\hat{\mathbf{i}} \cdot \hat{\mathbf{n}}) \sqrt{1 - \|\mathbf{c}\|^2} \hat{\mathbf{n}} \\ \mathbf{c} &= \frac{n_1}{n_2} (\hat{\mathbf{i}} \times \hat{\mathbf{n}}) \end{aligned} \tag{7}$$

There are two special considerations. First, if $\|\mathbf{c}\| > 1$, the radical in Equation 7 is complex-valued, and no refraction occurs. This phenomenon is known as total internal reflection. Second, if the incident ray is tangent to the surface it intersects with ($\hat{\mathbf{i}} \cdot \hat{\mathbf{n}} = 0$), it neither enters or exits the surface. As a result, there is no refraction or reflection.