PBRT: Shapes

Tuesday, November 10, 2020 11:37 PM

3 Shapes

"Careful abstraction of geometric shapes in a ray tracer is a key component of a clean system design, and shapes are the ideal candidate for an object-oriented approach."

The renderer doesn't know any details of the underlying shape, it deals only with the abstract *Primitive* interface. The *Primitive* interface hides the *Shape* interface behind it, which in turn hides the details of the underlying geometry.

A Shape stores an ***isomorphism between object space and world space***. The *ObjectToWorld* Transform maps the coordinate vector of a point of the Shape onto a world-space coordinate. Its inverse, the *WorldToObject* Transform does the opposite. These Transforms are pointers really; multiple Shapes may share the same Transform to reduce memory usage.

3.1.1 Bounding

An axis-aligned bounding box is an effective and inexpensive implementation of a Shape's ***bounding volume***, because they have a compact representation and ray-aabb intersections are cheap to compute.

An AABB is not super tight, so Shape implementations may decide to override the method that computes the bounding volume.

The transformation of a Bounds3f involves computing a new AABB that contains the original AABB's corner points. What the world-space AABB of a Shape actually bounds is then the object-space AABB; as a result, the Shape may not be tightly bound by the world-space AABB. A better method is to transform the coordinates of the Shape to world space first and then bound it.

3.1.2 Ray-bounds intersections

A ray intersects a bounding box at 2 points, that is, at 2 values of the ray's parameter $\boldsymbol{t}$*:* t_{near} and t_{far}.

A ray-box intersection test actually performs 3 tests: one on each pair of parallel faces. The volume contained between 2 parallel faces is called a ***slab***. So technically, these are called ***ray-slab*** intersection tests.

Recall that the ray's *tMax* restricts the reach of the ray to *[0, r(tMax)].* Slab-ray Intersections outside this interval don't count. And if the ray's origin is contained inside the box, $t_{near} = 0$.

A ***degenerate interval*** is one of the form *[a, a].* If a ray doesn't intersect a given box, the parametric interval returned will be degenerate.

Recall that a ***ray*** is a vector function $\boldsymbol{r(t) = o + td}$**.** We want to know if there is a $\boldsymbol{t}$ where $\boldsymbol{r(t)}$ satisfies the ***implicit equation of the plane***:

$$ax + by + cy + d = 0$$

$$a(o_x + td_x) + b(o_y + td_y) + c(o_z + td_z) + d = 0$$

$$ao_x + atd_x + bo_y + btd_y + co_z + ctd_z + d = 0$$

$$ao_x + bo_y + co_z + tad_x + tbd_y + tcd_z + d = 0$$

$$ao_x + bo_y + co_z + t(ad_x + bd_y + cd_z) + d = 0$$

This is a *scalar equation*, not a *vector equation*. The sum can't mix vector terms and scalar terms. The *scalar product* comes in handy here for expressing the sum of component-wise products: $ao_x + bo_y + co_z = (a, b, c) \cdot \boldsymbol{o}$.

$$(a, b, c) \cdot \boldsymbol{o} + t(a, b, c) \cdot \boldsymbol{d} + d = 0$$

Now solve for ***t****:*

$$t = \frac{-((a, b, c) \cdot \boldsymbol{o}) - d}{(a, b, c) \cdot \boldsymbol{d}}$$

From Planes, we know that the equation of the plane comes from the following orthogonality relation between the normal n and the vector PP_0 between a pair of points that lie in it is:

$$n \cdot PP_0 = 0$$

Expanding the dot product, we see that the coefficients *a, b, c* of the equation are the components of the normal n:

$$\begin{aligned} \mathbf{n} \cdot \overrightarrow{P_0 P} &= 0 && \text{Dot product of orthogonal vectors} \\ \langle a,\, b,\, c \rangle \cdot \langle x - x_0,\, y - y_0,\, z - z_0 \rangle &= 0 && \text{Substitute vector components.} \\ a\,(x - x_0) + b\,(y - y_0) + c\,(z - z_0) &= 0 && \text{Expand the dot product.} \\ a\,x + b\,y + c\,z &= d. && d = a\,x_0 + b\,y_0 + c\,z_0 \end{aligned}$$

Since the box is axis-aligned, the 6 planes of its faces are described with the following normals and points:

- The pair of faces that are perpendicular to the *X* axis:

  Leftmost: $(\mathrm{x}_1, 0, 0)$
  Rightmost: $(\mathrm{x}_2, 0, 0)$, where $\mathrm{x}_1 < x_2$
  Normal: $(1, 0, 0)$

- The pair of faces that are perpendicular to the *Y* axis:

  Upper: $(0,\ \mathrm{y}_1, 0)$
  Lower: $(0, \mathrm{y}_2, 0)$, where $\mathrm{y}_1 < y_2$
  Normal: $(0, 1, 0)$

- The pair of faces that are perpendicular to the *Z* axis:

  Front: $(0, 0, \mathrm{z}_1)$
  Back: $(0, 0, \mathrm{z}_2)$, where $\mathrm{z}_1 < z_2$
  Normal: $(0, 0, 1)$

With this information, we can simplify the ***t*** equation of each face.

$$t_1 = \frac{x_1 - o_x}{d_x}$$

and

$$t_2 = \frac{x_2 - o_x}{d_x}$$

for left and right faces.

$$t_1 = \frac{y_1 - o_y}{d_y}$$

and

$$t_2 = \frac{y_2 - o_y}{d_y}$$

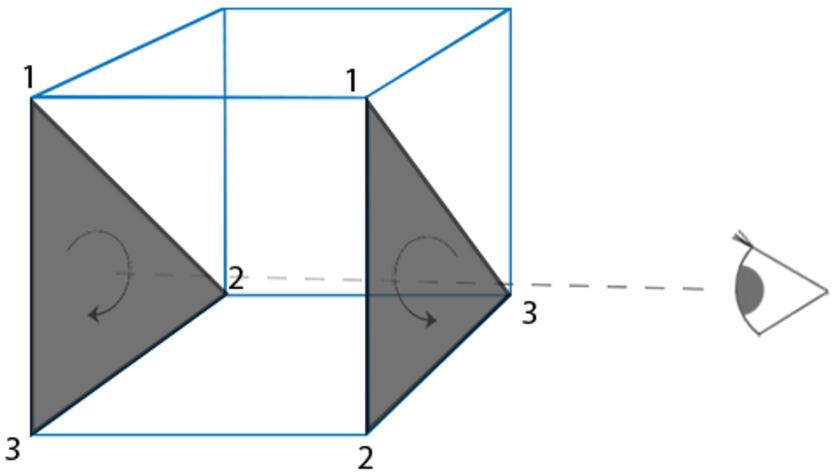
for up and down faces.

And similarly for front and back faces.

3.1.5 Sidedness

Sidedness refers to the ability of some rendering pipelines of doing back-face culling. OpenGL supports it, for instance. The winding order of the vertices of a triangle is enough to tell which face a triangle is showing; when the triangle is showing the back-face and it is part of a closed object, that triangle can be culled so that it doesn't participate in the Z-buffer algorithm.

The triangle on the left is showing its back face and it's part of a closed object: triangles that show their front face occlude the triangle in question



PBRT doesn't do back-face culling for reasons noted in the book.

3.9 Managing rounding error

Error may be introduced every time a floating-point operation is performed, because the intended resulting ***real number*** may not be representable with a ***floating-point number***.

Consequences for ray intersections:

- Valid intersections may be missed.

- Computed ray-shape intersection points may be above or below the surface of the shape (above or below the actual incidence point). These points are the wrong origins of reflected and shadow rays.

  An intersection point below the surface causes re-intersection with the surface. A shadow ray would incorrectly determine that the surface of interest occludes itself from the light source. In the case of outgoing reflected rays, Ray Tracing in One Weekend says that this is the source of ***shadow acne*** and solves it by ignoring intersections of reflected rays that occur at $t < 0.001$ (so that they may escape the surface). PBRT says that this is a ***bad way*** of dealing with the problem.

  An intersection point above the surface causes an incoming ray to miss a valid intersection, and outgoing reflected rays and shadow rays to cause detached reflections and shadows.

3.9.1 Floating-point arithmetic

Fixed-point numbers are equally spaced real number. For example, a spacing of $\frac{1}{256} = 0.00390625$ between represented numbers.

Floating-point numbers devote a fixed number of bits to the sign, to the significand, and to the exponent.

The belief that floating-point arithmetic is unpredictable is wrong. The IEEE 754 standard specifies floating-point representation and arithmetic.