PBRT: Geometry and Transformations

Saturday, October 17, 2020 8:44 PM

2 Geometry and transformations

Points, vectors, and rays are ***mathematical constructs*.**

Good abstractions and efficient implementations are critical because they are ubiquitous.

PBRT's geometric foundation.

2.1 Coordinate systems

The origin of the space and 3 ***linearly independent*** vectors that define the x, y, and z axes of the space are called the ***frame*** that defines the coordinate system.

A point exists in a given absolute position in space. Its coordinates depend on the frame being used. (Here we are decoupling the concepts of point and of coordinates. A given point will have different coordinates in each frame of reference.)

In computer graphics, rotation, scaling, translation, and shearing are all linear transformations. But we also use ***affine transformations*** there. Affine transformations are applied to affine spaces and subspaces. An ***affine space*** is a set of points with an underlying vector space. In affine spaces, there's no notion of origin, angle or distance. Origin, angle, and distance are only defined in Euclidean spaces, which are themselves affine spaces. Affine combinations are analogous to linear combinations, but they are more restricted in their definition: the sum of the scalar coefficients must be 1. This guy gives a concise introduction to affine spaces.

A frame's origin p_0 and its *n* basis vectors define an *n*-dimensional *affine space.*

There is a unique set of scalar coefficients s_i that expresses ***v*** as a linear combination of the basis vectors. (The matrix of a set of basis vectors is necessarily square and invertible. It follows by the Invertible Matrix Theorem that ***Ax=v*** has a unique solution ***x*,** i.e. there's a unique vector of coefficients *x,* for each *b.*)

$$v = s_1 v_1 + \cdots + s_n v_n$$

New: the scalars s_i that give values to the coefficients of a linear combination ***v*** of basis vectors are called the ***representation of v with respect to the basis***. They are the ***coordinates*** of the vector.

Vectors and points are not freely interchangeable: ***points*** are expressed in terms of the origin p_0 and the basis vectors:

$$p = p_0 + s_1 v_1 + \cdots + s_n v_n$$

Canonical, ***standard frame*** with origin (0, 0, 0) and basis vectors (1, 0, 0), (0, 1, 0), (0, 0, 1): ***world space*.**

2.1.1 Coordinate system handedness

There are only 2 different ways that the 3 coordinate axes can be arranged: with the z axis pointing towards you, or ***right-handed***, or away from you, ***left-handed*** (x points right, y points up). PBRT is left-handed.

(Use the term ***tuple*** instead of triplet.)

2.2.1 Dot product and cross product

Dot product has a relationship with the angle between the vectors:

$$(v \cdot w) = \|v\|\|w\| cos\theta$$

It follows that the dot product of 2 vectors is 0 when they are ***perpendicular*** to each other, i.e. θ is $\pi/2$, or one or both vectors are ***degenerate*** (equal to 0).

A set of mutually perpendicular or ***orthogonal unit vectors*** is called ***orthonormal***.

Some properties:

$$(v \cdot w) = (w \cdot v)$$

$$(sv \cdot w) = s(v \cdot w)$$

$$(u \cdot (v + w)) = (u \cdot v) + (u \cdot w)$$

Cross product is a vector that is perpendicular to the 2. It is computed the same way you would compute the following ***determinant*** using a cofactor expansion across the first row (note that the first row are vectors and the rest are scalars; this is just a mnemonic):

$$(v \times w) = \begin{bmatrix} \hat{\imath} & \hat{\jmath} & \hat{k} \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix}$$

$$(v \times w)_x = v_y w_z - v_z w_y$$

$$(v \times w)_y = \mathrm{v}_z w_x - v_x w_z$$

$$(v \times w)_z = v_x w_y - v_y w_x$$

where $\hat{\imath}$, $\hat{\jmath}$, and $\hat{k}$ are the basis vectors of 3-space.

The determinant here is a mnemonic, but also a little bit more than that: the norm of the cross product vector is the area of the parallelogram or the volume of the parallelepiped formed by the 2 vectors .

Now, there are 2 possible cross product vectors. The one you get depends on 2 things: the order of the operands $(v \times w)$ or $(w \times v)$ and the relative orientation of the 2: is *v* to the left or to the right of *w?* To understand what left and right means, consider this: the order of the basis vectors is what defines the orientation of the space; one is the orientation of the left hand (z points away from you) and the other the orientation of the right hand (z points toward you). Left handedness places $\hat{\jmath}$ to the left of $\hat{\imath}$*;* right handedness places *j to the right of* $\hat{\imath}$. Now, imagine that you are facing the plane described *by v and w:* if you have $(v \times w)$ and *w* is to the left of *v* (just like *j* is to the left of *i* in left handedness)*,* then the cross product will point away from you; if you flip the operands and have $(w \times v)$, you'll have flipped the direction of the cross product vector; now, if *w* is to the righ of *v*, then the cross product will point towards you; reversing the order of the operands will invert the direction of the cross product.

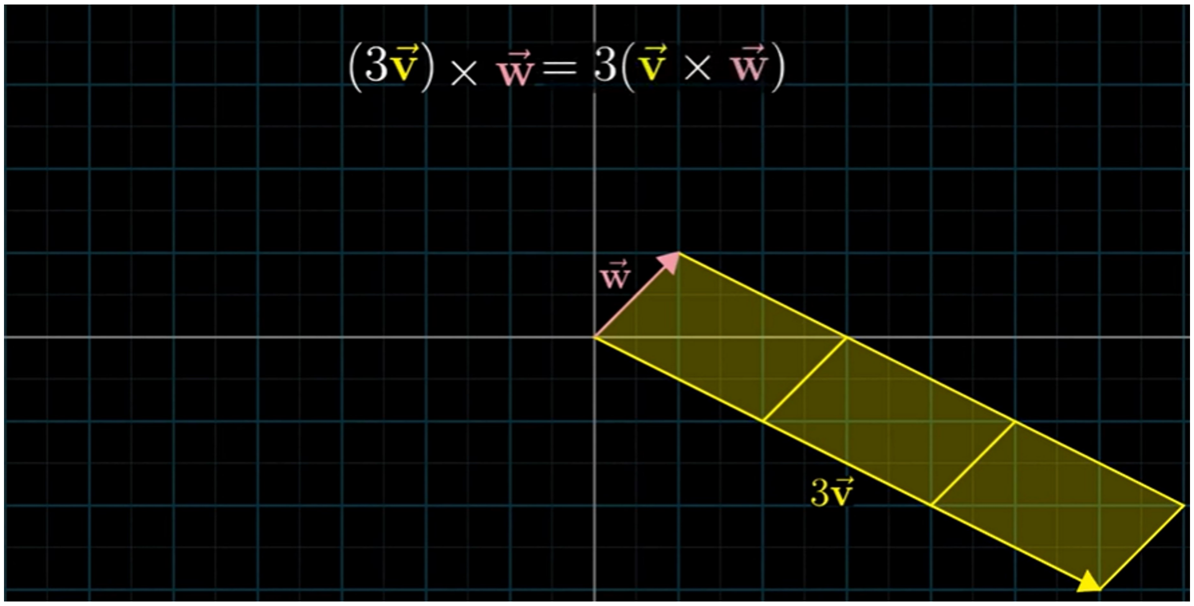
Cross product has a relationship with the angle between the vectors:

$$\|(v \times w)\| = \|v\|\|w\| sin\theta$$

It follows that the cross product of 2 parallel vectors is a degenerate vector (one that is equal to 0).

Some properties:

$(sv \times w) = s(v \times w)$



$T(v \times w) = T(v) \times T(w)$, where T is a linear transformation. ***Is it really linear? An example in page 118 of a scaling of a cross product seems to show otherwise.***

The ***catastrophic cancellation floating-point error*** is mentioned here and it can occur when you subtract 2 single-precision values that are very close together.

2.2.2 Normalization

$\hat{v}$ is the normalized version of v.

2.2.4 Coordinate system from a vector

You can construct the frame of a coordinate system out of just 1 vector: you need 2 vectors that are perpendicular to it.

The first one: between x and y, choose the component with largest absolute value; zero out the smaller of the 2, negate z, and swap the larger of the 2 with (negated) z.

Does this ***method*** have a name?

The 2nd one: compute the cross product of the 2.

2.3 Points

PBRT defines points as zero-dimensional locations in 2D or 3D space.

Subtracting a point from another results in the vector between them.
Subtracting a vector from a point results in the displaced point.

Scalar multiplication and addition of points only makes sense when computing the weighted sum of a set of points, and only when the weights' sum is 0.

Lines that overlap are called ***concurrent***.

The linear interpolation formula *(1-t)*p1 + tp2* also ***extrapolates***: when t < 0 or t > 1, the resulting point lies on the line that contains p1 and p2, but outside the segment that joins them.

2.4 Normals

PBRT distinguishes between normals and vectors. What sets normals apart is that they are always ***defined with respect to a surface*** and that they ***behave differently*** than vectors in some situations, ***particularly when applying linear transformations***. *(That's why you pass a different matrix to the vertex shader for the normal.)*

Normals only make sense in 3D.

A normal can't be added to a point and the cross product of 2 normals can't be computed.

*Normals are **not necessarily** normalized.*

Conversion between normals and vectors must be made with an explicit cast.

2.5 Rays

Rays are ***semi-infinite lines***.

Rays have ***origins***, not "starting points".

A ray's ***parametric form*** gives the set of points that it passes through:

$$r(t) = o + t\boldsymbol{d}, \qquad \boldsymbol{0 < t < \infty}$$

A ray may have a maximum extent, i.e. instead of being semi-infinite, $o + t_{max}\boldsymbol{d}$ would be its maximum extent. Ray-object intersections are recorded in t_{max}.

A ray knows what ***medium*** it departed from; different media have different properties and the properties of the ray origin's medium have an effect on the surface/medium it intersects.

C++ allows you to override the ***() operator***, called "function call operator".

2.5.1 Ray differentials

1st paragraph gives a glimpse of what ray differential are used for: texture antialiasing.

A ray differential is a ***pair of auxiliary rays*** that may accompany a given ray. The origin of these auxiliary rays is always the camera and they differ from the ray they accompany in that they are ***one sample away*** from it: one is offset in the *x* direction only and the other is offset in the *y* direction only.

The offsets of the ray differential are defined by that who generates them. For example, one Camera implementation generates a differential with an offset of 1 pixel, whereas the SamplerIntegrator uses a subpixel offset.

RayDifferential is a subclass of Ray.

2.6 Bounding boxes

At least 2 use cases:

- Multi-threading works by subdividing the image into rectangular tiles (and the scene into the corresponding axis-aligned regions of space) that can be processed independently.

- Bounding volume hierarchies.

PBRT implements ***axis-aligned bounding boxes (AABBs)***. An alternative was ***oriented bounding boxes (OBBs)*** that aren't necessarily axis-aligned.

The box is represented with 2 opposite corner points only.

2.7 Transformations

$p' = \boldsymbol{T}(\boldsymbol{p})$

$v' = \boldsymbol{T}(\boldsymbol{v})$

Linear, continuous, one-to-one and invertible.

Change of basis, change of frame, with a 4x4 matrix.

Multiplication by a 4x4 matrix can be used for 2 things:

- Transforming a point or vector within the same frame.
- Transforming a point or vector from one frame to another. (This is the most frequent use in PBRT, presumably because transforming within the same frame is more for real-time applications, whereas transforming to another frame is essential for offline rendering, i.e. going from world space to camera space.)

The Transform class uses a lot of memory for its matrix, relatively. Shape classes uses pointers to shared Transform instances, and Scenes use a TransformCache: a scene may have millions of objects; if each one had its own Transform, PBRT would use up too much memory.

Transforms are typically created during scene parsing and aren't modified after creation (there's typically no need).

2.7.1 Homogeneous coordinates

PBRT distinguishes between a point and vector (other than by their class) with the ***weight*** of their ***homogeneous representation***:

- Homogeneous representation of a ***point***: *(x, y, z, w)*, where *w ≠ 0* is the weight. A point thus has infinitely many homogeneous representations, one per value of *w.* The unique non-homogeneous representation is obtained like so:

  $(\frac{\boldsymbol{x}}{\boldsymbol{w}}, \frac{\boldsymbol{y}}{\boldsymbol{w}}, \frac{\boldsymbol{z}}{\boldsymbol{w}})$

- Homogeneous representation of a ***vector***: *(x, y, z, 0).*

A ***change of frame*** of a point or vector is achieved by left-multiplying it by a matrix that characterizes the transformation of the current frame's basis into the new frame's basis.

The Transform class converts the vectors and points that it receives to their homogeneous representation and gives them back in non-homogeneous form. Nowhere else in PBRT are homogeneous coordinates used.

2.7.2 Basic operations

PBRT implements the inverse of a matrix using row reduction, a.k.a. Gauss-Jordan elimination. Remember that row reduction on an augmented matrix *[A I]* results in *[I A^{-1}]* if A is invertible.

2.7.3 Translations

$T(\Delta x, \Delta y, \Delta z)$

The ***translation matrix***:

Translating a point (weight of homogeneous representation is 1 for points):

$$T(\Delta x, \Delta y, \Delta z)\ (x, y, z, 1) = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \mathbf{\Delta y} \\ 0 & 0 & 0 & \mathbf{1} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1x + 0y + 0z + \Delta x \\ 0x + 1\mathrm{y} + 0\mathrm{z} + \Delta y \\ 0x + 0y + 1z + \Delta z \\ 0x + 0y + 0z + 1 \end{bmatrix} = \begin{bmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ 1 \end{bmatrix}$$

Translating a vector (weight of homogeneous representation is 0 for vectors)

$$T(\Delta x, \Delta y, \Delta z)\ (x, y, z, 0) = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \mathbf{\Delta y} \\ 0 & 0 & 0 & \mathbf{1} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} 1x + 0y + 0z + 0\Delta x \\ 0x + 1\mathrm{y} + 0\mathrm{z} + 0\Delta y \\ 0x + 0y + 1z + 0\Delta z \\ 0x + 0y + 0z + 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

Note that translation leaves vectors intact: they don't get translated.

Properties:

- *Acts like identity when deltas are 0 and doesn't translate the point:*

  $T(0, 0, 0) = I$

  $$T(0, 0, 0)\ (x, y, z, 1) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \mathbf{0} \\ 0 & 0 & 0 & \mathbf{1} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \boldsymbol{I} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1x + 0y + 0z + 0 \\ 0x + 1\mathrm{y} + 0\mathrm{z} + 0 \\ 0x + 0y + 1z + 0 \\ 0x + 0y + 0z + 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- *Product of translations translates point by the sum of the deltas:*

  $T(\Delta x_1, \Delta y_1, \Delta z_1)\ T(\Delta x_2, \Delta y_2, \Delta z_2) = T(\Delta x_1 + \Delta x_2, \Delta y_1 + \Delta y_2, \Delta z_1 + \Delta z_2)$

  $$T(\Delta x_1, \Delta y_1, \Delta z_1)\ T(\Delta x_2, \Delta y_2, \Delta z_2)$$
  $$= \begin{bmatrix} 1 & 0 & 0 & \Delta x_1 \\ 0 & 1 & 0 & \Delta y_1 \\ 0 & 0 & 1 & \Delta z_1 \\ 0 & 0 & 0 & \mathbf{1} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & \Delta x_2 \\ 0 & 1 & 0 & \Delta y_2 \\ 0 & 0 & 1 & \Delta z_2 \\ 0 & 0 & 0 & \mathbf{1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \Delta x_2 + \Delta x_1 \\ 0 & 1 & 0 & \Delta y_2 + \Delta y_1 \\ 0 & 0 & 1 & \Delta z_2 + \Delta z_1 \\ 0 & 0 & 0 & \mathbf{1} \end{bmatrix}$$
  $$= T(\Delta x_1 + \Delta x_2, \Delta y_1 + \Delta y_2, \Delta z_1 + \Delta z_2)$$

- *Product of translations is commutative:*

  $T(\Delta x_1, \Delta y_1, \Delta z_1)\ T(\Delta x_2, \Delta y_2, \Delta z_2) = T(\Delta x_2, \Delta y_2, \Delta z_2) T(\Delta x_1, \Delta y_1, \Delta z_1)$

$T(\Delta x_1, \Delta y_1, \Delta z_1)\; T(\Delta x_2, \Delta y_2, \Delta z_2)$

$$= \begin{bmatrix} 1 & 0 & 0 & \Delta x_1 \\ 0 & 1 & 0 & \Delta y_1 \\ 0 & 0 & 1 & \Delta z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & \Delta x_2 \\ 0 & 1 & 0 & \Delta y_2 \\ 0 & 0 & 1 & \Delta z_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \Delta x_2 + \Delta x_1 \\ 0 & 1 & 0 & \Delta y_2 + \Delta y_1 \\ 0 & 0 & 1 & \Delta z_2 + \Delta z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & \Delta x_1 + \Delta x_2 \\ 0 & 1 & 0 & \Delta y_1 + \Delta y_2 \\ 0 & 0 & 1 & \Delta z_1 + \Delta z_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \Delta x_2 \\ 0 & 1 & 0 & \Delta y_2 \\ 0 & 0 & 1 & \Delta z_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & \Delta x_1 \\ 0 & 1 & 0 & \Delta y_1 \\ 0 & 0 & 1 & \Delta z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$= T(\Delta x_2, \Delta y_2, \Delta z_2) T(\Delta x_1, \Delta y_1, \Delta z_1)$

Observe that this means that you may apply a set of translations on a point in any order and the result will be the same.

- *Inverse of translation translates point in the opposite direction of the original translation:*

$T(\Delta x, \Delta y, \Delta z) = T(-\Delta x, -\Delta y, -\Delta z)$

$$T^{-1}(\Delta x, \Delta y, \Delta z)\;(x, y, z, 1) = \begin{bmatrix} 1 & 0 & 0 & -\Delta x_1 \\ 0 & 1 & 0 & -\Delta y_1 \\ 0 & 0 & 1 & -\Delta z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1x + 0y + 0z - \Delta x \\ 0x + 1y + 0z - \Delta y \\ 0x + 0y + 1z - \Delta z \\ 0x + 0y + 0z + 1 \end{bmatrix} = \begin{bmatrix} x - \Delta x \\ y - \Delta y \\ z - \Delta z \\ 1 \end{bmatrix}$$

Row reducing augmented matrix $[\; T(\Delta x_1, \Delta y_1, \Delta z_1) \quad I \;]$ *results in* $[\; I \quad T^{-1}(\Delta x, \Delta y, \Delta z) \;]$:

$$\begin{bmatrix} 1 & 0 & 0 & \Delta x & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \Delta y & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta z & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \sim \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & -\Delta x \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & -\Delta y \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & -\Delta z \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} = [\; I \quad T^{-1}(\Delta x, \Delta y, \Delta z) \;]$$

2.7.4 Scaling

$S(sx, sy, sz)$

The ***scaling matrix***:

Scaling a point (weight of homogeneous representation is 1 for points):

$$S(s_x, s_y, s_z)(x, y, z, 1) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x x + 0y + 0z + 0 \\ 0x + s_y y + 0z + 0 \\ 0x + 0y + s_z z + 0 \\ 0x + 0y + 0z + 1 \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \\ 1 \end{bmatrix}$$

Scaling a vector (weight of homogeneous representation is 0 for vectors)

$$S(s_x, s_y, s_z)(x, y, z, 0) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} s_x x + 0y + 0z + 0 \\ 0x + s_y y + 0z + 0 \\ 0x + 0y + s_z z + 0 \\ 0x + 0y + 0z + 0 \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \\ 0 \end{bmatrix}$$

Unlike translation, scaling does have an effect on vectors, not only on points.

Properties:

- *Acts like identity when scalars are 1 and doesn't scale the point or vector:*

  $S(1,1,1) = I$

  $$S(1,1,1)\ (x,y,z,1) = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & \mathbf{0}\\ 0 & 0 & 0 & \mathbf{1}\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix} = \boldsymbol{I}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix} = \begin{bmatrix}1x+0y+0z+0\\ 0x+1y+0z+0\\ 0x+0y+1z+0\\ 0x+0y+0z+1\end{bmatrix} = \begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix}$$

- *Product of scalings scales point or vector by the product of the scalars:*

  $$S(s1_x, s1_y, s1_z)S(s2_x, s2_y, s2_z) = S(s1_x \cdot s2_x, s1_y \cdot s2_y, s1_z \cdot s2_z)$$

  $$S(s1_x, s1_y, s1_z)S(s2_x, s2_y, s2_z)$$

  $$= \begin{bmatrix}s1_x & 0 & 0 & 0\\ 0 & s1_y & 0 & 0\\ 0 & 0 & s1_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}s2_x & 0 & 0 & 0\\ 0 & s2_y & 0 & 0\\ 0 & 0 & s2_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix} = \begin{bmatrix}s1_x \cdot s2_x & 0 & 0 & 0\\ 0 & s1_y \cdot s2_y & 0 & 0\\ 0 & 0 & s1_z \cdot s2_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

  $$= S(s1_x \cdot s2_x, s1_y \cdot s2_y, s1_z \cdot s2_z)$$

- *Product of scalings is commutative:*

  $$S(s1_x, s1_y, s1_z)S(s2_x, s2_y, s2_z) = S(s1_x, s1_y, s1_z)S(s2_x, s2_y, s2_z)$$

  Observe that this means that you may apply a set of scalings on a point or vector in any order and the result will be the same.

- *Inverse of scaling scales point or vector by the reciprocal of the scalar:*

  $$S^{-1}(s_x, s_y, s_z) = S\left(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z}\right)$$

When all scalars are equal, the scaling is ***uniform***; when they are different, it is ***nonuniform***.

2.7.5 x, y, and z axis rotations

$R_x(\theta)$, rotation around the *x* axis.

Rotating a point around the *x* axis (weight of homogeneous representation is 1 for points):

$$R_x(\theta)\ (x,y,z,1) = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & -\sin\theta & 0\\ 0 & \sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix} = \begin{bmatrix}x+0y+0z+0\\ 0x+y\cos\theta - z\sin\theta+0\\ 0x+y\sin\theta+z\cos\theta+0\\ 0x+0y+0z+1\end{bmatrix} = \begin{bmatrix}x\\ y\cos\theta - z\sin\theta\\ y\sin\theta + z\cos\theta\\ 1\end{bmatrix}$$

Vectors (weight of homogeneous representation is 0) are rotated the same way.

$R_y(\theta)$, rotation around the *y* axis.

$$R_y(\theta)\,(x,y,z,1) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos\theta + 0y + z\sin\theta + 0 \\ 0x + 1y + 0z + 0 \\ -x\sin\theta + 0y + z\cos\theta + 0 \\ 0x + 0y + 0z + 1 \end{bmatrix} = \begin{bmatrix} x\cos\theta + z\sin\theta \\ y \\ -x\sin\theta + z\cos\theta \\ 1 \end{bmatrix}$$

$R_z(\theta)$, rotation around the *z* axis.

$$R_z(\theta)\,(x,y,z,1) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta + 0z + 0 \\ x\sin\theta + y\cos\theta + 0z + 0 \\ 0x + 0y + 1z + 0 \\ 0x + 0y + 0z + 1 \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ z \\ 1 \end{bmatrix}$$

Properties:

- *Acts like identity when angle θ is 0 and doesn't rotate the point or vector:*

  $R_a(\theta) = \mathrm{I}$

- *Product of rotations rotates point or vector by the sum of the angles:*

  $R_a(\theta_1)\,R_a(\theta_2) = R_a(\theta_1 + \theta_2)$

- *Product of rotations is commutative:*

  $R_a(\theta_1)\,R_a(\theta_2) = R_a(\theta_2)\,R_a(\theta_1)$

- *Inverse of rotation rotates point or vector in the opposite angular direction:*
- ***Inverse** rotation matrix is **equal** to the **transpose** of the rotation matrix:*

  $R_a{}^{-1}(\theta) = R_a(-\theta) = R_a{}^{T}(\theta)$

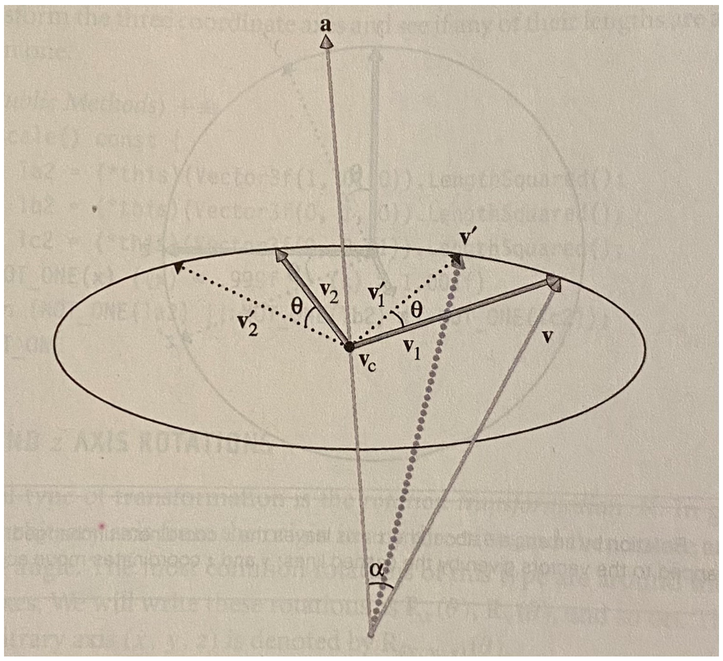
  R_a is an ***orthogonal matrix***: $A^{-1} = A^T$

  This property is great because computing an inverse (via row reduction, Gauss-Jordan elimination) is much more expensive than the transpose.

2.7.6 Rotation around an arbitrary axis

$R_{(x,y,z)}(\theta)$

The arbitrary axis to rotate around is given by a normalized vector ***a***. The vector to rotate around ***a*** is ***v.***



We try to describe a ***plane of rotation*** that is perpendicular to ***a*** and that rests at the tip of vector ***v***.

First, we obtain the ***origin*** of the ***plane of rotation***:

v_c is the ***scalar projection*** of ***v*** onto ***a:***

$$v_c = (v \cdot a)\ a$$

- If ***v*** is normalized (we know that ***a*** is), their dot product is the cosine of the angle between them.
- Why is ***a*** the vector that gets scaled to obtain v_c? Because the resulting vector needs to go in the direction of ***a.***
- Why scale by the cosine of the angle? Because v_c is the adjacent leg of the right triangle ΔOvv_c. Normalized vector ***a*** scaled by the cosine gives causes the tip of the resulting vector to be where the adjacent leg of the triangle ends.

v_c is the origin of the ***plane of rotation*** that is perpendicular to ***a*** and that rests at the tip of vector ***v*** (or at the tip of v_c itself).

Now we obtain the ***2 basis vectors*** of the ***plane of rotation***:

v_1 is just $v_1 - v_c$.

v_2 is perpendicular to both $v_1$and ***a:*** $v_2 = (v_2 \times a)$.

(As always with the cross product, handedness and the order of the operands determines the direction of the resulting vector. In this case, PBRT is left-handed. In left-handed orientation, if you face the plane of the operand vectors, if the second operand is to the left of the first one [the same way $\hat{\jmath}$ is to the left of $\hat{\imath}$], the resulting vector will point away from you.)

Now we obtain the ***rotated vector*** v':

On the ***plane of rotation***, a simple right triangle inscribed in the circle of rotation gives you an expression for v':

$v' = v_c + v_1 \cos\theta + v_2 \sin\theta$

But what we ultimately want is a single rotation matrix. This matrix encodes the rotations of the 3 standard basis vectors.

2.7.7 The look-at transformation

The look-at transformation transforms a point from camera space to world space (yes, from camera space to world space).

Given the camera's position, a look-at point, and an *up* vector, a frame for the camera space can be constructed:

$(position, left, up, forward)$

with non-position vectors normalized as needed, where $left = (up \ \times lookat - position)$ and $forward = lookat - position$.

$$\begin{bmatrix} L_x & U_x & F_x & P_x \\ L_y & U_y & F_y & P_y \\ L_z & U_z & F_z & P_z \\ 0 & 0 & 0 & \mathbf{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{c_x} \\ \boldsymbol{c_y} \\ \boldsymbol{c_z} \\ 1 \end{bmatrix}$$

where *L* is *left, U* is *up, F* is *forward,* and *P* is *position*. This matrix effects a ***change of basis***: the columns are the world space coordinates of the basis vectors of camera space, that is, the basis of camera space is expressed in relation to the basis of world space; left-multiplying a camera-space point by it will change the coordinates of this point to world space.

2.8.3 Applying transformations to normals

When a point on a surface is transformed, its normal needs to be transformed accordingly. The transformation *T* that transformed the point, however, doesn't transform the normal correctly.

To derive the transformation *S* for the normal, we note that the ***tangent*** to the surface at the point does get transformed correctly by *T.*

The orthogonality between the normal ***n*** and the tangent ***t*** leads us to the following relation:

$(n \cdot t) = n^T t = 0$

Note that $n^T t$ is 1x1, because, by convention, a vector ***v*** is a column vector. Note also that there's a $\pi/2$ angle between orthogonal vectors, the dot product of 2 normalized vectors is the cosine of their angle, and the cosine of this angle here is 0.

When properly transformed, the transformed normal and tangent maintain the same relation:

$0 = n'^T t'$
$0 = (Sn)^T Tt$
$0 = n^T S^T Tt$

The key observation is: the relation of orthogonality is maintained only when $S^T T = I$:

$0 = n^T S^T Tt$
$0 = n^T It$

$0 = n^T t$

which is the original relation of orthogonality between the vectors before the transformation.

So we want a matrix *S* such that $S^T T = I$. This equation is only satisfied by $S^T = T^{-1}$.

So the matrix *S* that ***transforms normals*** is $S = (T^{-1})^T$, the ***inverse transpose of T***.

2.8.5 Applying transformations to axis-aligned bounding boxes

An ***axis-aligned*** bounding box is represented by 2 of its diagonal corner points.

You can't transform an axis-aligned bounding box by just transforming the 2 diagonal corner points of its representation: take a rotation by $\pi/2$ around the *y* axis: *pMin* will now be to the right of *pMax*; or take a rotation by less than of $\pi/2$ around the *y* axis: the resulting axis-aligned bounding box will be a shrunken version of the original one because *pMin* and *pMax* will now be closer to each other than before.

The correct way to transform and axis-aligned bounding box is to transform each of the corner points and then compute the axis-aligned bounding box that contains them.

2.8.7 Transformations that change coordinate system handedness

There are transformations that change the handedness of the coordinate system. *(When? When transforming a point or vector? What transformations do that? An example is given in page 118, with scaling S(1, 1, -1).)*

A transformation with a 4x4 matrix whose ***upper-left 3x3 submatrix*** has a ***negative determinant*** changes the handedness of the coordinate system. *(I haven't been able to find further information.)*

2.10 Interactions

Surface Interactions represent information about the local differential geometry and material properties at a point on a **2D** surface. (Similar to *hit record* in Ray Tracing in One Weekend.)

Medium Interactions represent points where light scatters in participating media (smoke, clouds).

Surfaces and participating media are treated differently.

`Interaction` is base class of *`SurfaceInteraction`* and *`MediumInteraction`*.

Floating-point error handling: discussed in detail later, but the gist is that a ray-surface intersection point will be off by some amount; each kind of *Shape* has tolerance for a different amount of error; *participating media* have 0 tolerance. *An `Interaction`* carries the associated surface or medium's error tolerance with it. (The book says its actually a *bound,* not a *tolerance.* What's a *bound*?)

What's the difference between *participating media* and *scattering media?*

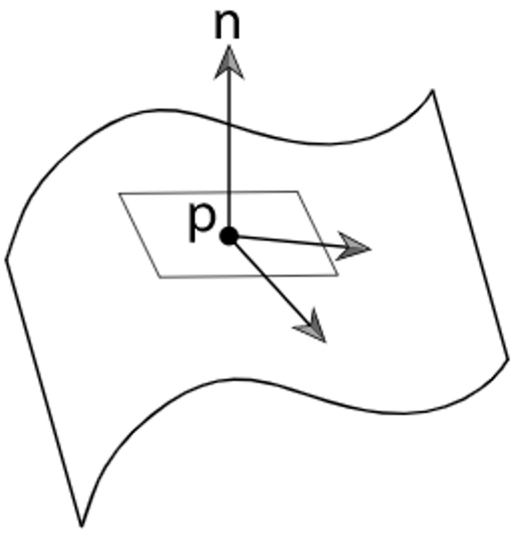
2.10.1 Surface interaction

A *surface interaction* decouples shading and geometric operations from the particular *Shape* of the intersected surface. All these routines need to know is contained in the *surface interaction*.

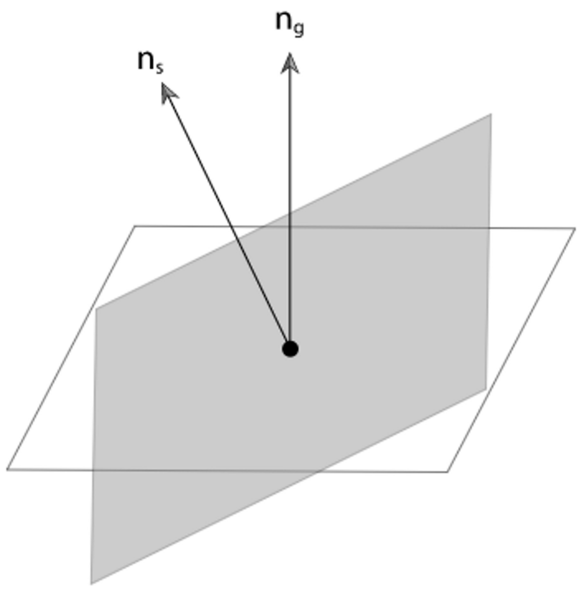
The surfaces of all *Shapes* in PBRT are described parametrically by a *(u, v)* coordinate at each point. Given a point's *(u, v)* coordinate, there's an isomorphism from a subspace of $\mathbb{R}^2$ onto a subspace of $\mathbb{R}^3$ that gives the point's 3D coordinates *(x, y z)*.

A surface interaction doesn't just store the point p and normal n of the intersection, but the corresponding (u, v) coordinate and the partial derivative of p with respect to u $\frac{\partial p}{\partial u}$, the partial derivative of p with respect to v $\frac{\partial p}{\partial v}$, the partial derivative of n with respect to u $\frac{\partial n}{\partial u}$, and the partial derivative of n with respect to v $\frac{\partial n}{\partial v}$. The partial derivatives measure how much the point and the normal change along the u and the v directions, where vectors *u* and *v* lie on the plane tangent to the point (and perpendicular to the normal).

p, n, (u, v), and the partial derivatives describe the ***local differential geometry*** around the point *p* of a surface.



The SurfaceInteraction holds a ***shading (differential) geometry*** comprised of a ***shading normal*** n_s and the partial derivatives of p and n_s with respect to u and v. Section 9.1 BSDFs explains that there's the ***geometric normal*** n_g and the ***shading normal*** n_s. The shading normal is created by bump mapping and, if the Shape is a triangle mesh, the shading normal is the per-vertex normal (or the interpolation?).



There are 2 reasons why the normal of a *surface interaction* may need to have its orientation flipped: 1) by default, the ***normals*** of Shapes ***point outward***, but the Shape may be configured to have its normal point inward instead; and 2) the Transform of the Shape may change the handedness of the Shape's local coordinate system (the determinant of the upper-left 3x3 submatrix is negative), affecting the normal; *surface interactions* are probably expected to use the default left-handed coordinate system, so coordinates of normals stored by them should be with respect to a left-handed coordinate system.

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 ***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 $\boldsymbol{r}(\boldsymbol{t}) = \boldsymbol{o} + \boldsymbol{td}$**.** We want to know if there is a $\boldsymbol{t}$ where $\boldsymbol{r}(\boldsymbol{t})$ 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 $\boldsymbol{t}$*:*

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

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: $(x_1, 0, 0)$
  Rightmost: $(x_2, 0, 0)$, where $x_1 < x_2$
  Normal: $(1, 0, 0)$

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

  Upper: $(0, y_1, 0)$

Lower: $(0, y_2, 0)$, where $y_1 < y_2$
Normal: $(0, 1, 0)$

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

  Front: $(0, 0, z_1)$
  Back: $(0, 0, z_2)$, where $z_1 < z_2$
  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

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

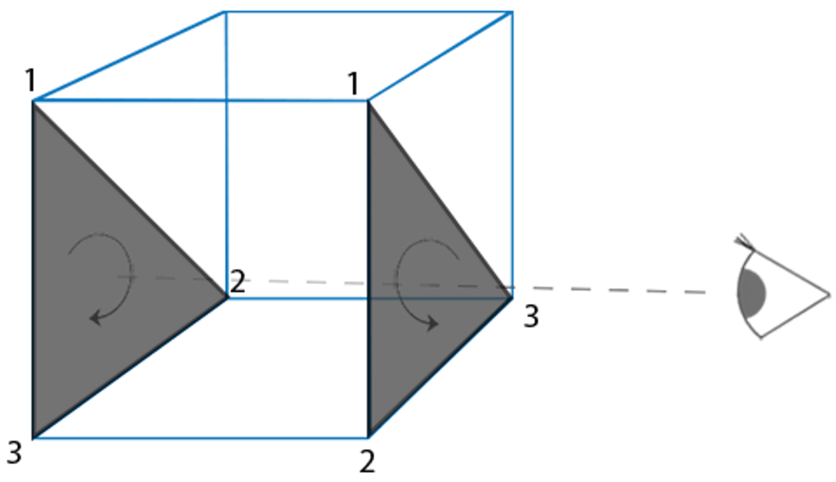
for up and down faces.

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.