

Geometry and Transformations

Digital Image Synthesis

Yung-Yu Chuang

9/24/2008

with slides by Pat Hanrahan

Geometric classes

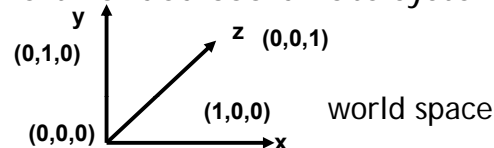


- Representation and operations for the basic mathematical constructs like points, vectors and rays.
- Actual scene geometry such as triangles and spheres are defined in the "Shapes" chapter.
- core/geometry.* and core/transform.*
- Purposes of learning this chapter
 - Get used to the style of learning by tracing source code
 - Get familiar to the basic geometry utilities because you will use them intensively later on

Coordinate system



- Points, vectors and normals are represented with three floating-point coordinate values: x , y , z defined under a coordinate system.
- A coordinate system is defined by an origin p_o and a frame (linearly independent vectors v_i).
- A vector $v = s_1v_1 + \dots + s_nv_n$ represents a direction, while a point $p = p_o + s_1v_1 + \dots + s_nv_n$ represents a position. They are not freely interchangeable.
- pbrt uses left-handed coordinate system.



Vectors



```
class Vector {
    public:
        <Vector Public Methods>
        float x, y, z;
}
no need to use selector (getX) and mutator (setX)
because the design gains nothing and adds bulk to its usage
```

Provided operations: **Vector u, v; float a;**

v+u, v-u, v+=u, v-=u

-v

(v==u)

a*v, v*=a, v/a, v/=a

a=v[i], v[i]=a

Dot and cross product



$$\text{Dot}(\mathbf{v}, \mathbf{u}) \quad \mathbf{v} \cdot \mathbf{u} = \|\mathbf{v}\| \|\mathbf{u}\| \cos \theta$$

`AbsDot(v, u)`

`Cross(v, u)`

$$\|\mathbf{v} \times \mathbf{u}\| = \|\mathbf{v}\| \|\mathbf{u}\| \sin \theta$$

Vectors \mathbf{v} , \mathbf{u} , $\mathbf{v} \times \mathbf{u}$
form a frame

$$(\mathbf{v} \times \mathbf{u})_x = v_y u_z - v_z u_y$$

$$(\mathbf{v} \times \mathbf{u})_y = v_z u_x - v_x u_z$$

$$(\mathbf{v} \times \mathbf{u})_z = v_x u_y - v_y u_x$$



Coordinate system from a vector



Construct a local coordinate system from a vector.

```
inline void CoordinateSystem(const Vector &v1,
                             Vector *v2, Vector *v3)
{
    if (fabsf(v1.x) > fabsf(v1.y)) {
        float invLen = 1.f/sqrtf(v1.x*v1.x + v1.z*v1.z);
        *v2 = Vector(-v1.z * invLen, 0.f, v1.x * invLen);
    }
    else {
        float invLen = 1.f/sqrtf(v1.y*v1.y + v1.z*v1.z);
        *v2 = Vector(0.f, v1.z * invLen, -v1.y * invLen);
    }
    *v3 = Cross(v1, *v2);
}
```

Normalization



`a=LengthSquared(v)`

`a=Length(v)`

`u=Normalize(v)` *return a vector, does not normalize in place*

Points



Points are different from vectors; given a coordinate system $(\mathbf{p}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$, a point \mathbf{p} and a vector \mathbf{v} with the same (x, y, z) essentially means

$$\mathbf{p} = (x, y, z, 1) [\mathbf{v}_1 \ \mathbf{v}_2 \ \mathbf{v}_3 \ \mathbf{p}_0]^T$$

$$\mathbf{v} = (x, y, z, 0) [\mathbf{v}_1 \ \mathbf{v}_2 \ \mathbf{v}_3 \ \mathbf{p}_0]^T$$

`explicit Vector(const Point &p);`

You have to convert a point to a vector explicitly (i.e. you know what you are doing).

`Vector v=p;`

`Vector v=Vector(p);`

Operations for points



Vector v ; Point p, q, r ; float a ;

$q = p + v$;

$q = p - v$;

$v = q - p$;

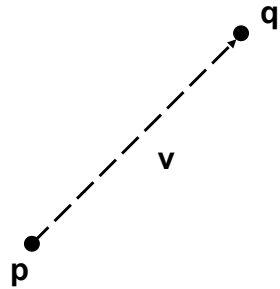
$r = p + q$;

$a * p$; p / a ;

(This is only for the operation $\alpha p + \beta q$.)

Distance(p, q);

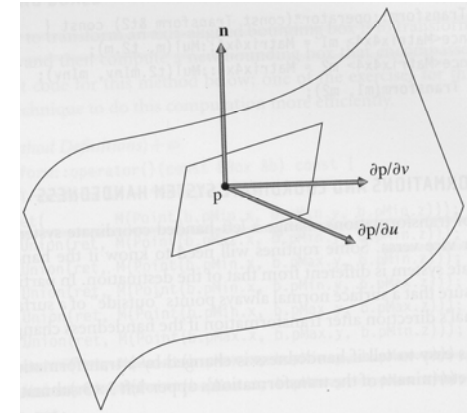
DistanceSquared(p, q);



Normals



- A *surface normal* (or just *normal*) is a vector that is perpendicular to a surface at a particular position.



Normals



- Different than vectors in some situations, particularly when applying transformations.
- Implementation similar to **Vector**, but a normal cannot be added to a point and one cannot take the cross product of two normals.
- Normal** is not necessarily normalized.
- Only explicit conversion between **Vector** and **Normal**.

Rays



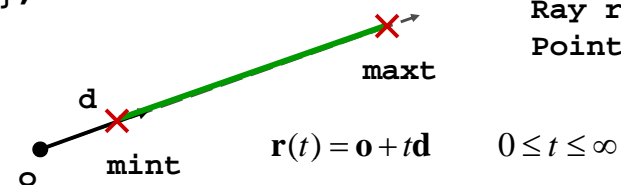
```
class Ray {
public:
    <Ray Public Methods>
    Point o;
    Vector d;
    mutable float mint, maxt;
    float time;
};
```

(They may be changed even if Ray is const. This ensures that o and d are not modified, but mint and maxt can be.)

Initialized as RAY_EPSILON to avoid self intersection.

(for motion blur)

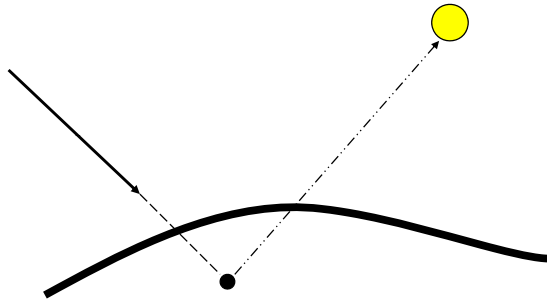
Ray r(o, d);
Point p=r(t);



Rays



```
Ray(): mint(RAY_EPSILON), maxt(INFINITY),  
time(0.f) {}
```



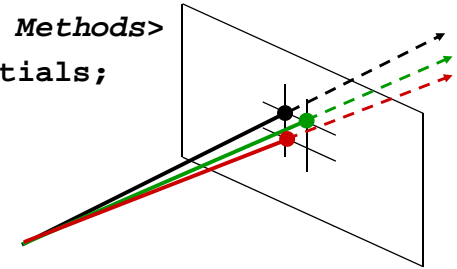
The reason why we need epsilon. Unfortunately, there is not a universal epsilon that works for all scenes.

Ray differentials



- Subclass of `Ray` with two auxiliary rays. Used to estimate the projected area for a small part of a scene and for antialiasing in Texture.

```
class RayDifferential : public Ray {  
public:  
    <RayDifferential Methods>  
    bool hasDifferentials;  
    Ray rx, ry;  
};
```

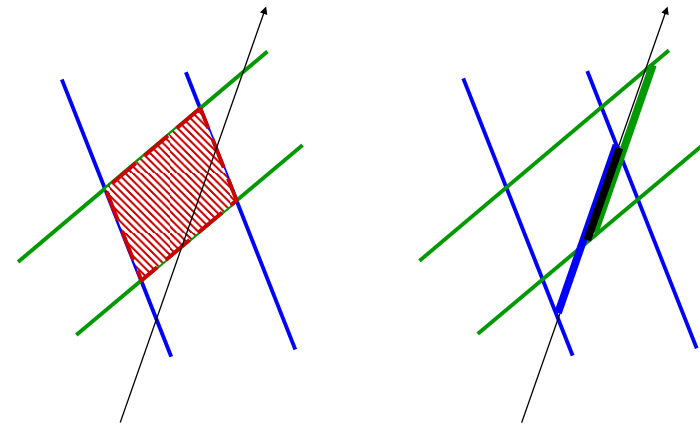


Bounding boxes



- To avoid intersection test inside a volume if the ray doesn't hit the *bounding volume*.
- Benefits depends on the expense of testing volume *v.s.* objects inside and the tightness of the bounding volume.
- Popular bounding volume, sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB) and slab.

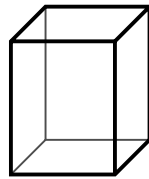
Bounding volume (slab)



Bounding boxes



```
class BBox {
public:
    <BBox Public Methods>
    Point pMin, pMax;
}
```



```
Point p,q; BBox b; float delta; bool s; two options
b = BBox(p,q) // no order for p, q of storing
b = Union(b,p)
b = Union(b,b2)
b = b.Expand(delta)
s = b.Overlaps(b2)
s = b.Inside(p)
Volume(b)
b.MaximumExtent() which axis is the longest; for building kd-tree
b.BoundingSphere(c, r) for generating samples
```

Transformations

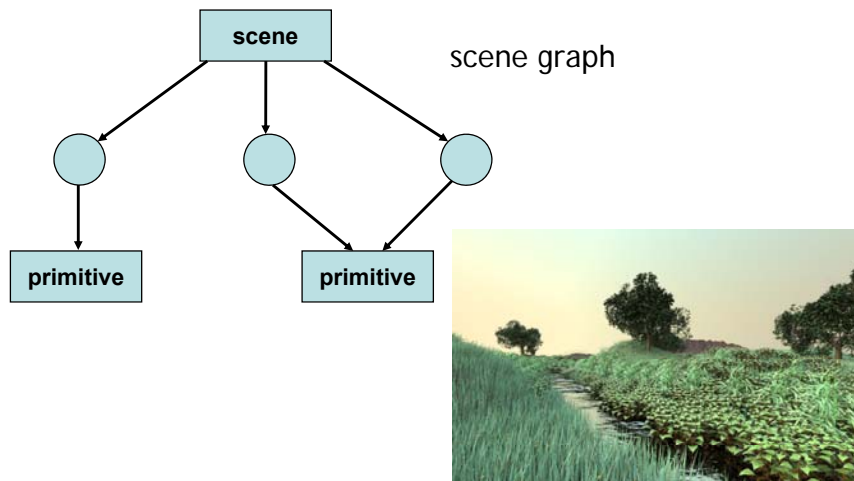


- $p'=T(p); v'=T(v)$
- Only supports transforms with the following properties:
 - Linear: $T(av+bu)=aT(v)+bT(u)$
 - Continuous: T maps the neighbors of p to ones of p'
 - Ont-to-one and invertible: T maps p to single p' and T^{-1} exists
- Represented with a 4x4 matrix; homogeneous coordinates are used implicitly
- Can be applied to points, vectors and normals
- Simplify implementations (e.g. cameras and shapes)

Transformations



- More convenient, instancing



Transformations



```
class Transform {
...
private:
    Reference<Matrix4x4> m, mInv;
} save space, but can't be modified after construction
Usually not a problem because transforms are pre-specified
in the scene file and won't be changed during rendering.

Transform() {m = mInv = new Matrix4x4; }
Transform(float mat[4][4]);
Transform(const Reference<Matrix4x4> &mat);
Transform(const Reference<Matrix4x4> &mat,
A better way const Reference<Matrix4x4> &minv);
to initialize
```

Transformations



- Translate(Vector(dx,dy,dz))
- Scale(sx, sy, sz)
- RotateX(a)

$$T(dx, dy, dz) = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$S(sx, sy, sz) = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R_x(\theta)^{-1} = R_x(\theta)^T$$

because R is orthogonal

Example for creating common transforms

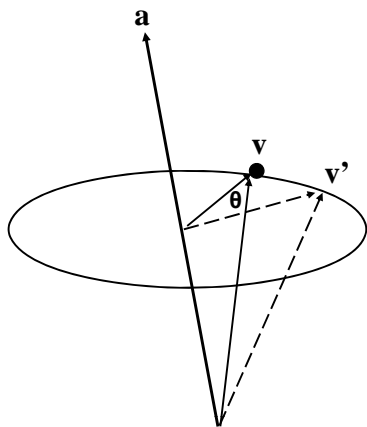


```
Transform Translate(const Vector &delta) {
    Matrix4x4 *m, *minv;
    m = new Matrix4x4(1, 0, 0, delta.x,
                    0, 1, 0, delta.y,
                    0, 0, 1, delta.z,
                    0, 0, 0, 1);
    minv = new Matrix4x4(1, 0, 0, -delta.x,
                       0, 1, 0, -delta.y,
                       0, 0, 1, -delta.z,
                       0, 0, 0, 1);
    return Transform(m, minv);
}
```

Rotation around an arbitrary axis



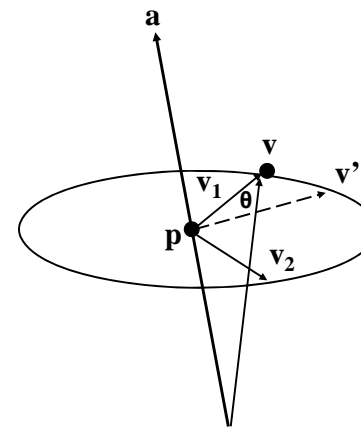
- Rotate(theta, axis) *axis is normalized*



Rotation around an arbitrary axis



- Rotate(theta, axis) *axis is normalized*



$$\begin{aligned} \mathbf{p} &= \mathbf{a}(\mathbf{v} \cdot \mathbf{a}) \\ \mathbf{v}_1 &= \mathbf{v} - \mathbf{p} \\ \mathbf{v}_2 &= \mathbf{a} \times \mathbf{v}_1 \quad |\mathbf{v}_2| = |\mathbf{v}_1| \\ \mathbf{v}' &= \mathbf{p} + \mathbf{v}_1 \cos \theta + \mathbf{v}_2 \sin \theta \end{aligned}$$

Rotation around an arbitrary axis



- Rotate(theta, axis) *axis is normalized*

$$\begin{pmatrix} & \mathbf{M} & & \\ & & & \mathbf{v} \\ & & & \\ & & & \end{pmatrix} = \mathbf{v}' = p + \mathbf{v}_1 \cos \theta + \mathbf{v}_2 \sin \theta$$

$$\begin{aligned}
 \mathbf{p} &= \mathbf{a}(\mathbf{v} \cdot \mathbf{a}) \\
 \mathbf{v}_1 &= \mathbf{v} - \mathbf{p} \\
 \mathbf{v}_2 &= \mathbf{a} \times \mathbf{v}_1 \quad |\mathbf{v}_2| = |\mathbf{v}_1|
 \end{aligned}$$

Rotation around an arbitrary axis



- `m[0][0]=a.x*a.x + (1.f-a.x*a.x)*c;`
- `m[1][0]=a.x*a.y*(1.f-c) + a.z*s;`
- `m[2][0]=a.x*a.z*(1.f-c) - a.y*s;`

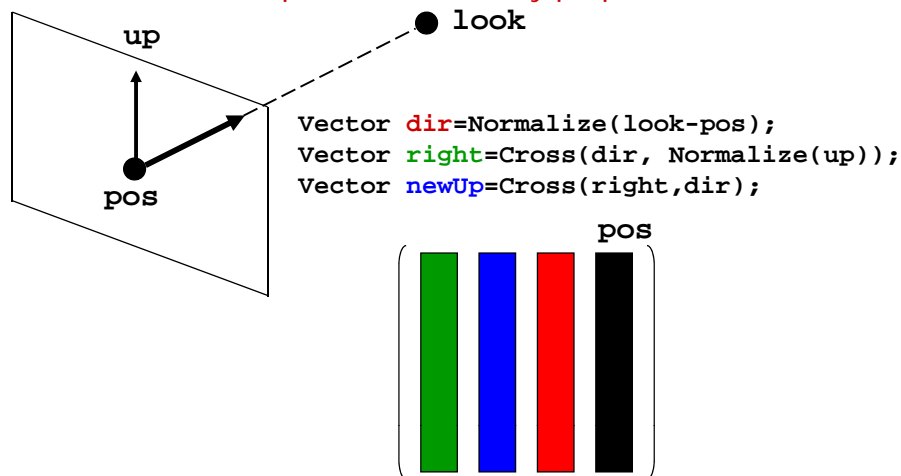
$$\begin{pmatrix} & & & & \\ & \mathbf{M} & & & \\ & & & & \mathbf{v} \\ & & & & \\ & & & & \end{pmatrix} = \mathbf{v}' = p + \mathbf{v}_1 \cos \theta + \mathbf{v}_2 \sin \theta$$

$$\begin{aligned}
 \mathbf{p} &= \mathbf{a}(\mathbf{v} \cdot \mathbf{a}) \\
 \mathbf{v}_1 &= \mathbf{v} - \mathbf{p} \\
 \mathbf{v}_2 &= \mathbf{a} \times \mathbf{v}_1 \quad |\mathbf{v}_2| = |\mathbf{v}_1|
 \end{aligned}$$

Look-at



- LookAt(Point &pos, Point look, Vector &up)
up is not necessarily perpendicular to dir

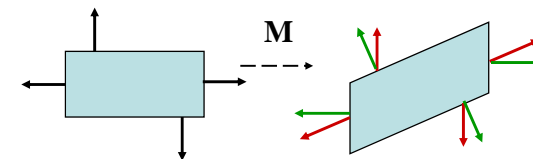


Applying transformations



- **Point:** $q=T(p)$, $T(p, \&q)$
use homogeneous coordinates **implicitly**
- **Vector:** $u=T(v)$, $T(u, \&v)$
- **Normal:** treated differently than vectors
because of anisotropic transformations

Point: (p, 1)
Vector: (v, 0)



$$\begin{aligned}
 \mathbf{n} \cdot \mathbf{t} &= \mathbf{n}^T \mathbf{t} = 0 \\
 (\mathbf{n}')^T \mathbf{t}' &= 0 \\
 (\mathbf{S}\mathbf{n})^T \mathbf{M}\mathbf{t} &= 0 \\
 \mathbf{n}^T \mathbf{S}^T \mathbf{M}\mathbf{t} &= 0
 \end{aligned}$$

- **Transform** should keep its inverse
- For orthonormal matrix, $\mathbf{S}=\mathbf{M}$

$$\begin{aligned}
 \mathbf{S}^T \mathbf{M} &= \mathbf{I} \\
 \mathbf{S} &= \mathbf{M}^{-T}
 \end{aligned}$$

Applying transformations



- **BBox**: transforms its eight corners and expand to include all eight points.

```
BBox Transform::operator()(const BBox &b) const {
    const Transform &M = *this;
    BBox ret(      M(Point(b.pMin.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret,M(Point(b.pMax.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret,M(Point(b.pMin.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret,M(Point(b.pMin.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret,M(Point(b.pMin.x, b.pMax.y, b.pMax.z)));
    ret = Union(ret,M(Point(b.pMax.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret,M(Point(b.pMax.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret,M(Point(b.pMax.x, b.pMax.y, b.pMax.z)));
    return ret;
}
```

Differential geometry



- **DifferentialGeometry**: a self-contained representation for a particular point on a surface so that all the other operations in pbprt can be executed without referring to the original shape. It contains
 - Position
 - Parameterization (u,v)
 - Parametric derivatives (dp/du , dp/dv)
 - Surface normal (derived from $(dp/du) \times (dp/dv)$)
 - Derivatives of normals
 - Pointer to shape

