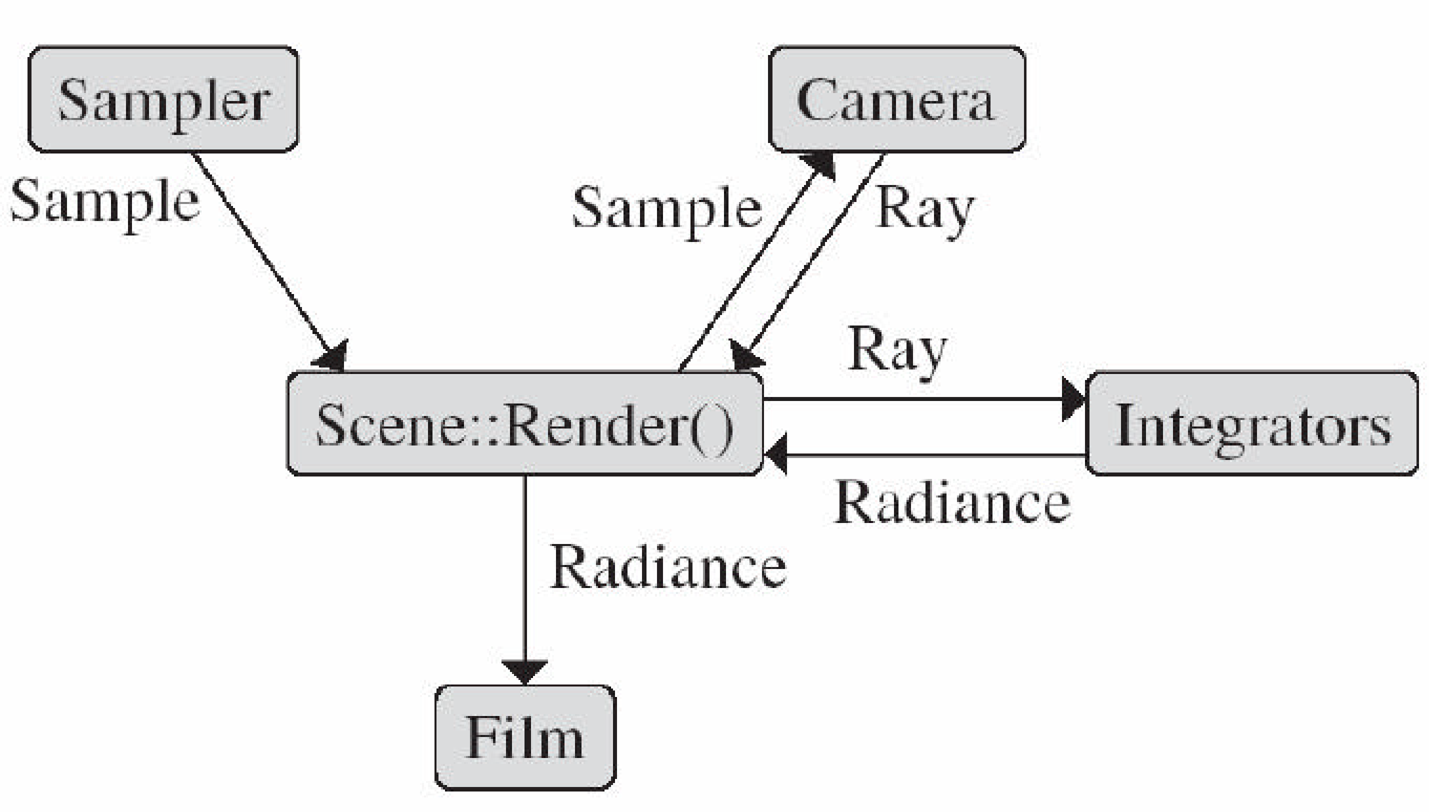
CS 563 Advanced Topics in Computer Graphics

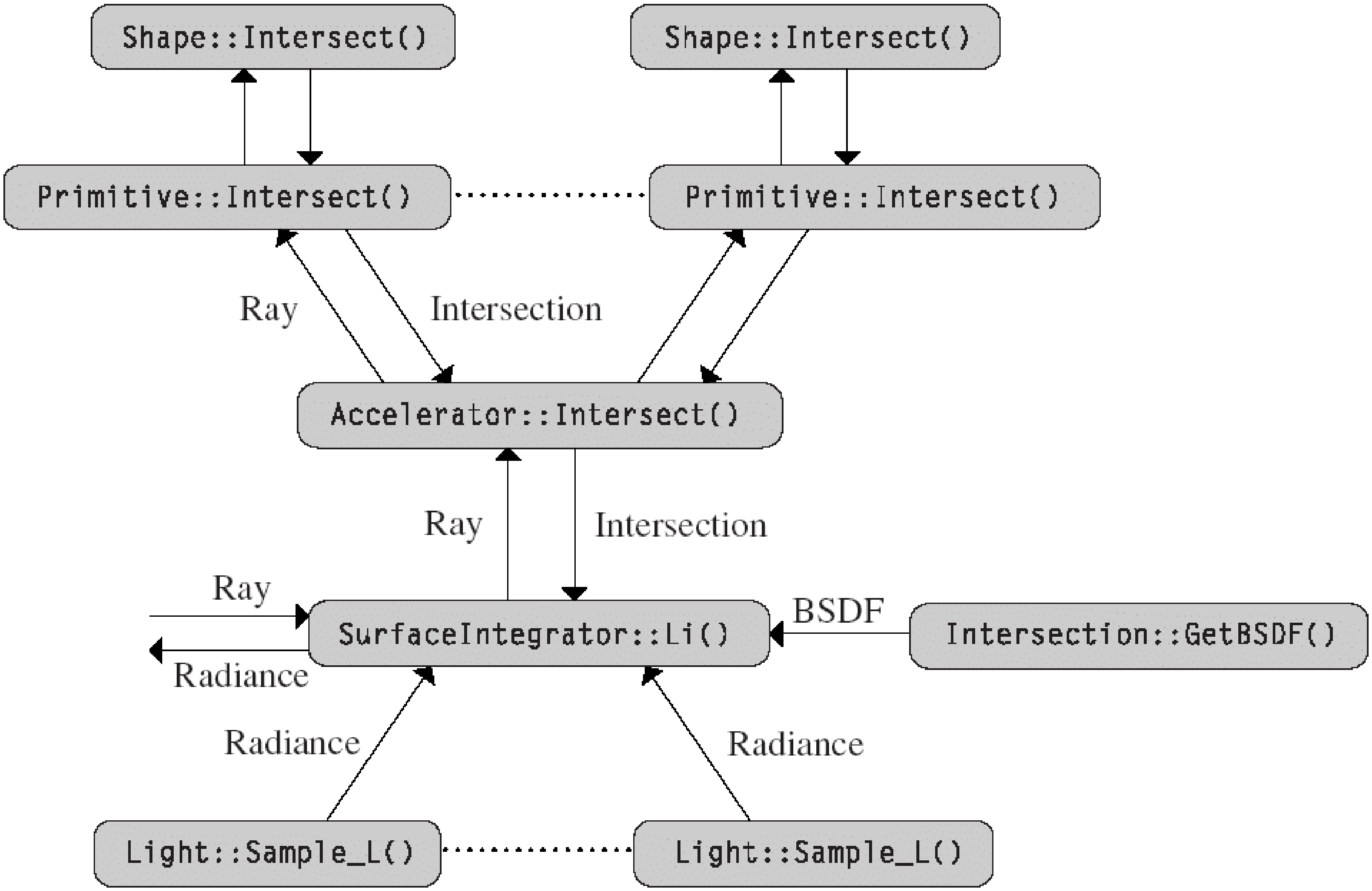
by Emmanuel Agu

PBRT Flow

- Parsing: uses lex and yacc: core/pbrtlex.l and core/pbrtparse.y
- After parsing, a **scene** object is created (core/scene.*)
- Rendering: **`Scene::Render()`** is invoked.



PBRT Architecture

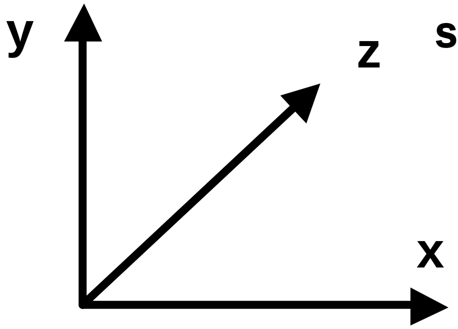


Geometric classes

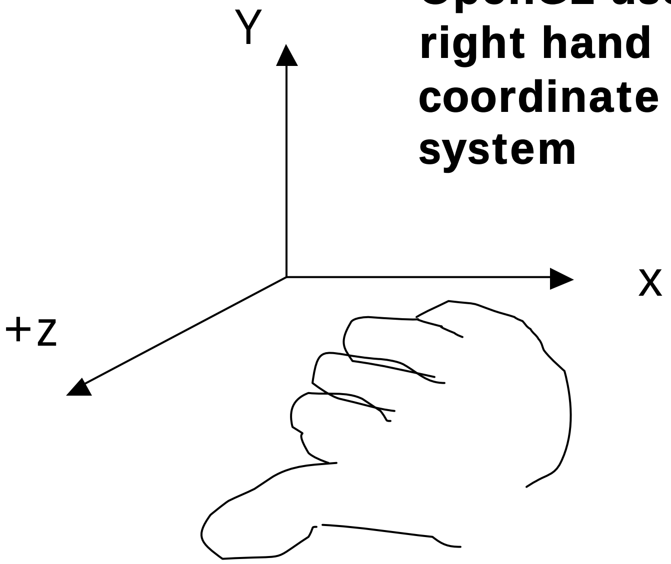
- Chapter 2: Representation and operations for the basic math:
  - points, vectors and rays.
  - core/geometry.* and core/transform.*
- Chapter 3 (Shapes): Actual scene geometry such as triangles and spheres.
- Chapter 4: Acceleration structures (uniform grid, kd-tree, BVH, etc)

Coordinate system

- Points, vectors and normals:
  - 3 floating-point coordinate values: *x, y, z* defined under a coordinate system.
- A coordinate system defined by:
  - Origin + frame
- Handedness?



PBRT uses right hand coordinate system



OpenGL uses right hand coordinate system

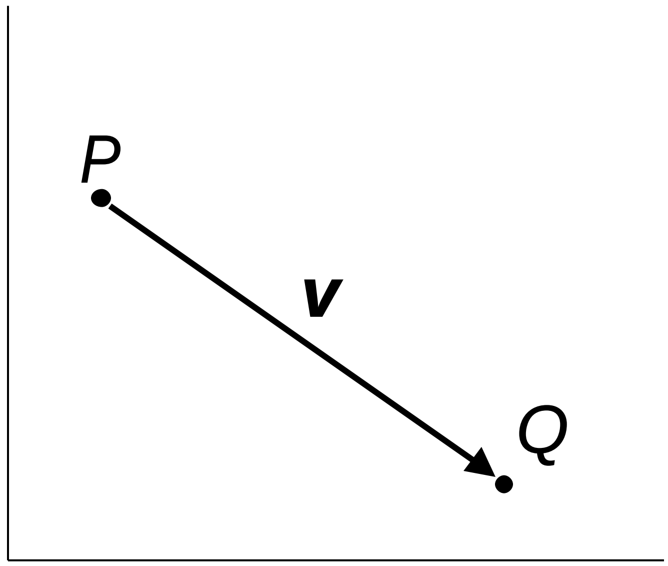
Vector-Point Relationship

- Diff. b/w 2 points = vector

$$\mathbf{v} = Q - P$$

- Sum of point and vector = point

$$\mathbf{v} + P = Q$$



Vector Operations

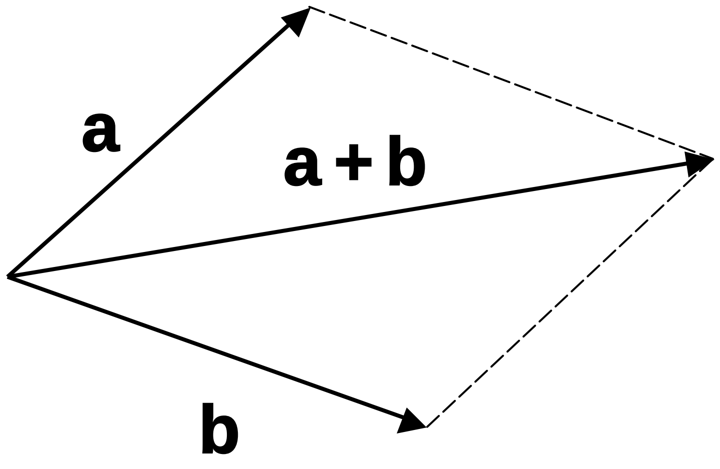
- Define vectors

$$\mathbf{a} = (a_{1,} a_2, a_3)$$
$$\mathbf{b} = (b_{1,} b_2, b_3)$$

- and scalar, *s*

Then vector addition:

$$\mathbf{a} + \mathbf{b} = (a_1 + b_{1,} a_2 + b_2, a_3 + b_3)$$



Vector Operations

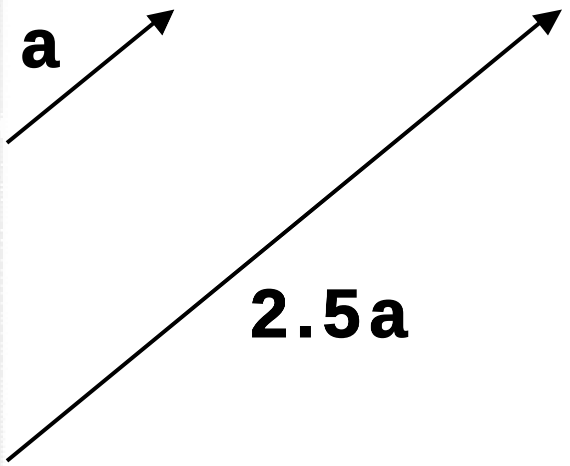
- Scaling vector by a scalar

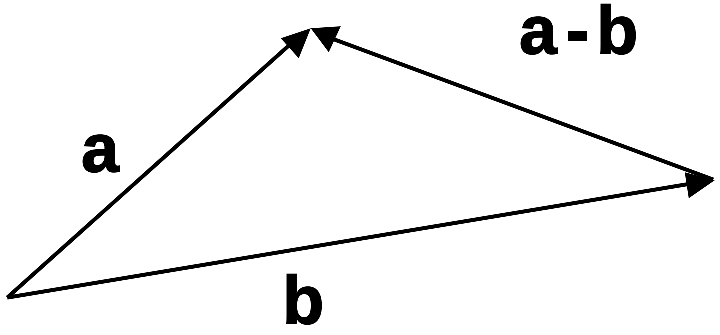
$$\mathbf{a}s = (a_1 s, a_2 s, a_3 s)$$

Note vector subtraction:

$$\mathbf{a} - \mathbf{b}$$

$$= (a_1 + (-b_1), a_2 + (-b_2), a_3 + (-b_3))$$





Magnitude of a Vector

- Magnitude of **a**

$$|\mathbf{a}| = \sqrt{a_1{}^2 + a_2{}^2 .... + a_n{}^2}$$

- Normalizing a vector (unit vector)

$$\hat{\mathbf{a}} = \frac{\mathbf{a}}{|\mathbf{a}|} = \frac{vector}{magnitude}$$

- Note magnitude of normalized vector = 1. i.e

$$\sqrt{a_1{}^2 + a_2{}^2 .... + a_n{}^2} = 1$$

Vectors

```
class Vector {
  public:
     <Vector Public Methods>
     float x, y, z;
}
```

(no need to use selector and mutator)

Dot and cross product

Dot(v, u)

AbsDot(v, u)

$$v \cdot u = \|v\|\|u\| \cos q$$

Cross(v, u)

$$\|v \times u\| = \|v\|\|u\| \sin q$$

(v, u, v×u) form a coordinate system

$$(v \times u)_x = v_y u_z - v_z u_y$$
$$(v \times u)_y = v_z u_x - v_x u_z$$
$$(v \times u)_z = v_x u_y - v_y u_z$$

Normalization

- **PBRT vector methods**
  - **`Length(v)-`** returns length of vector, v
  - **`LengthSquared(v) – (`**returns length of v)^2
  - **`Normalize(v)`** returns a vector, does not normalize in place

Coordinate system from a vector

Construct a local coordinate system from a vector.

```
inline void CoordinateSystem(const Vector &v1,
                             Vector *v2, Vector *v3)
```

- V1 normalized already.
- Construct v2: perpendicular vector of v1 by
  - Zero out 1 component of v1
  - Swap other 2 components
- V1 x v2 = v3: 3rd vector

Points

Points are different from vectors

```
explicit Vector(const Point &p);
```

You have to convert a point to a vector explicitly (no accidents, know what you are doing).

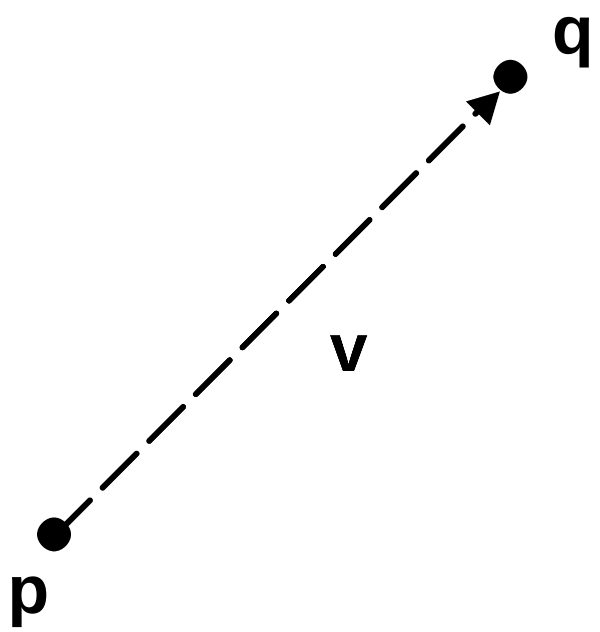
☒ `Vector v=p;`

☑ `Vector v=Vector(p);`

Operations for points

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

q=p+v;
q=p-v;
v=q-p;
```



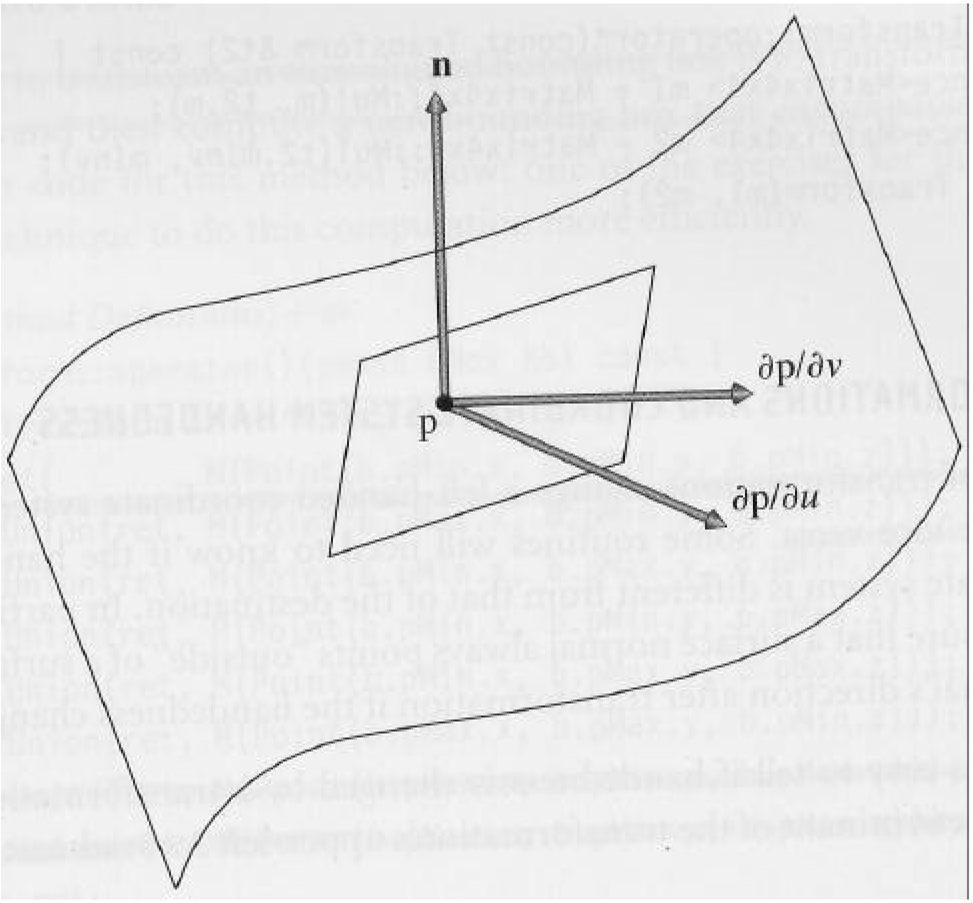
```
r=p+q;
a*p; p/a;
```

(This is only for the operation α p+β q.)

```
PBRT supports:
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 sometimes
- Particularly when applying transformations.
- Implementation similar to **`Vector`**, except
  - Normal cannot be added to a point
  - Cannot take the cross product of two normals.
- **`Normal`** is not necessarily normalized.
- Conversion between **`Vector`** and **`Normal`** must be explicit

Rays

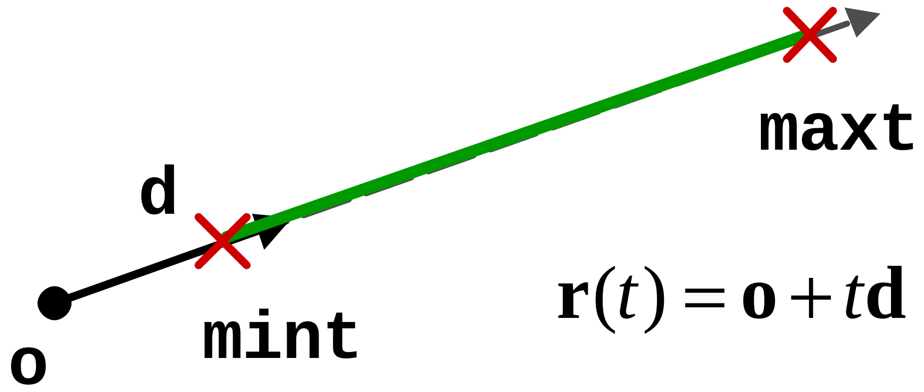
```
class Ray {
public:
   <Ray Public Methods>
   Point o;
   Vector d;
   mutable float mint, maxt;
   float time;
};
```

(They may be changed even if **`Ray`** is **`const`**.)

Initialized as **`RAY_EPSILON`** to avoid self intersection.

(for motion blur)

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

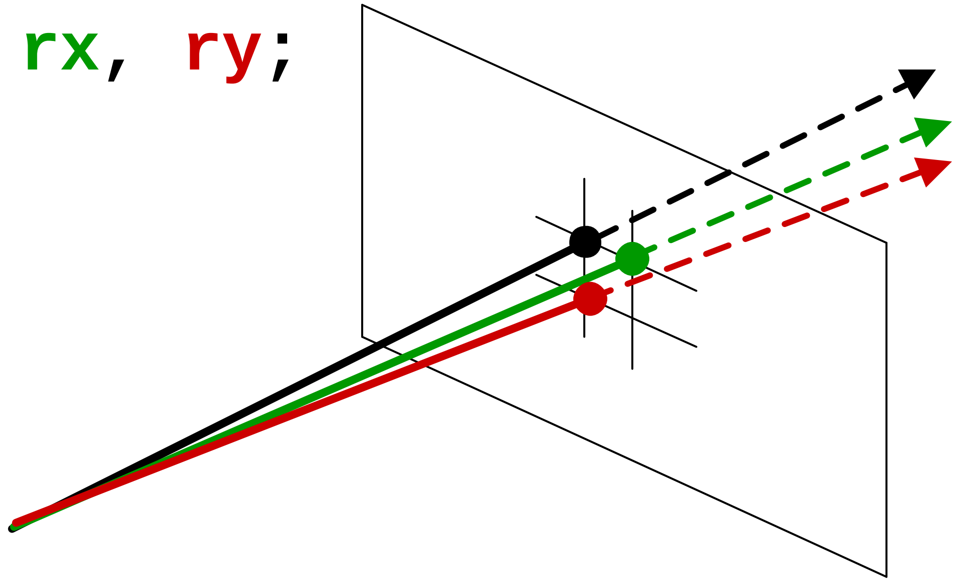


$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \qquad 0 \le t \le \infty$$

Ray differentials

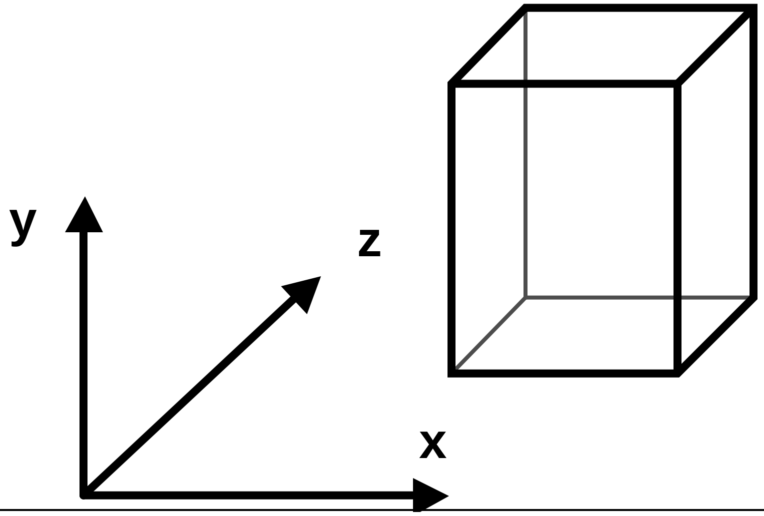
- Used to estimate projected area for a small part of a scene
- Used for **texture** antialiasing.

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



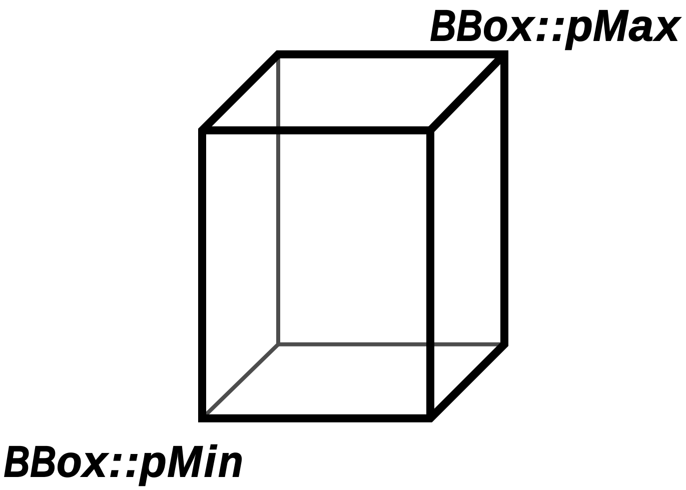
Bounding boxes

- Avoid intersection tests inside a volume if ray doesn’t hit *bounding volume*.
- Benefits depends on:
  - Expense of testing volume *vs* objects inside
  - Tightness of the bounding volume.
- Popular bounding volumes: sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB).



Bounding boxes

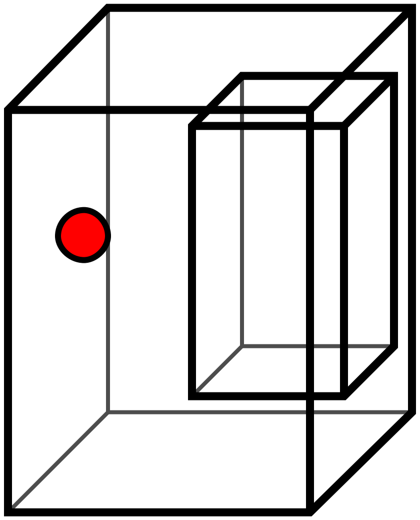
```
class BBox {
public:
   <BBox Public Methods>
   Point pMin, pMax;
}
```

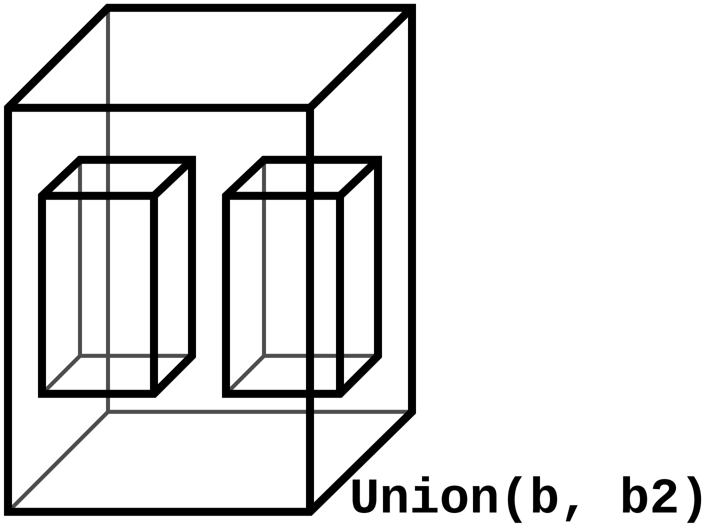


`Point p,q; BBox b; float delta;`

`BBox(p,q)   // no order for p, q`

`Union(b,p)` – Given point & Bbox, return new larger bounding box containing point (bbox) and Bbox.

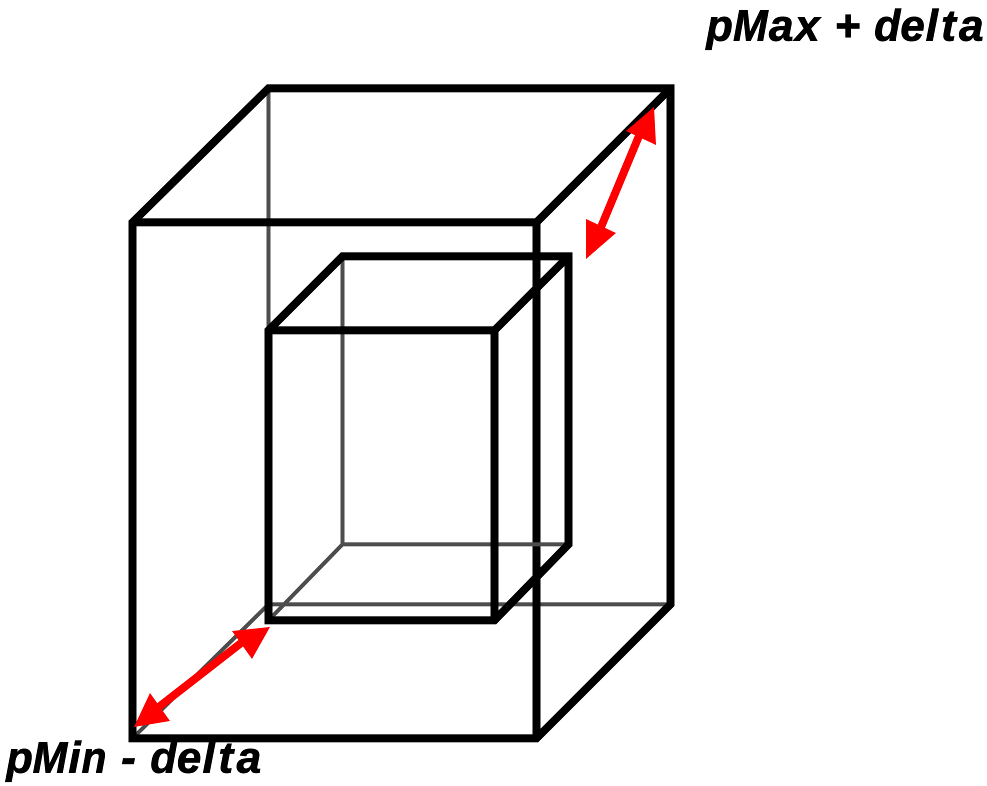




Bounding boxes

`Point p,q; BBox b;`

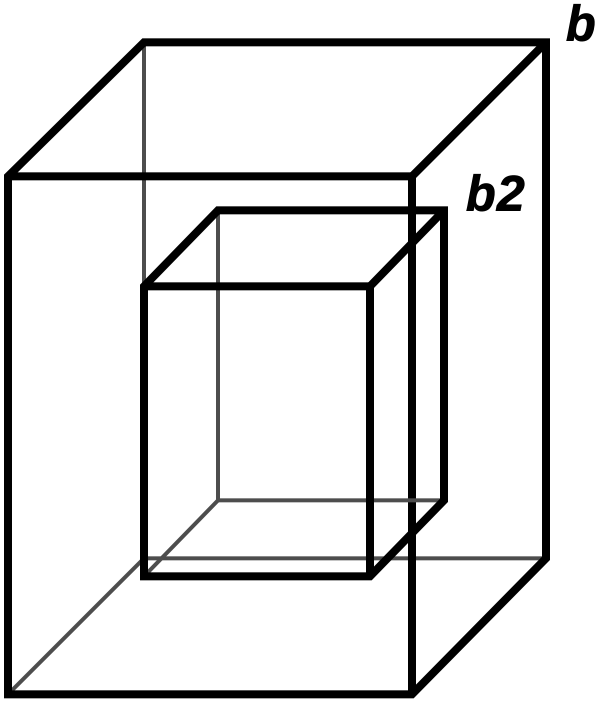
`b.Expand(delta):` Expand old bounding box by factor delta



Bounding boxes

`Point p,q; BBox b;`

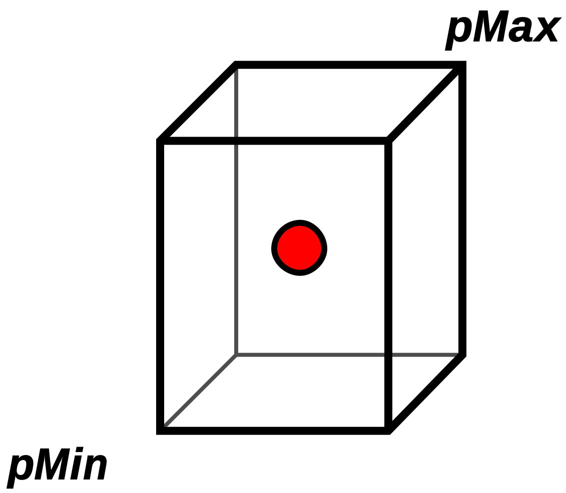
- **`b.Overlaps(b2):`** do two bounding boxes overlap each other in x,y,z
- Returns boolean. True (overlaps) or false (does not overlap)



Bounding boxes

`Point p,q; BBox b;`

- **`b.Inside(p):`** Is point p inside bounding box? Returns boolean (true or false)
- **`Volume(b):`** Returns volume of bounding volume (x * y * z)



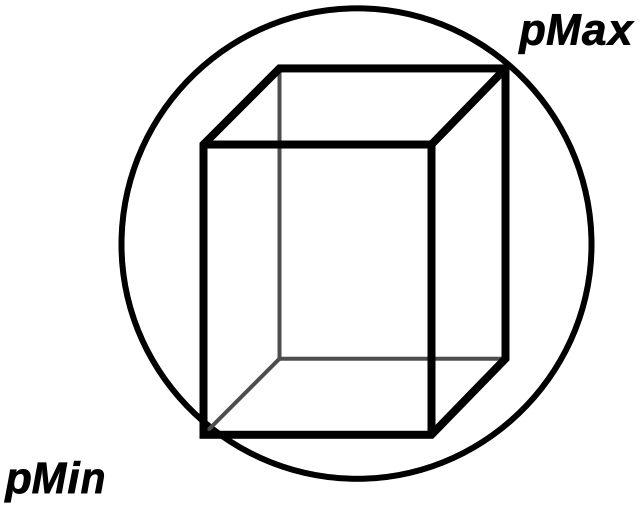
Bounding boxes

`Point p,q; BBox b;`

`b.MaximumExtent()`(*which bounding box axis is the longest; useful for building kd-tree)*

`b.BoundingSphere(c, r)` (*returns center and radius of bounding sphere)*

- Example: generate random ray which intersects scene geometry



Transformations

```
class Transform {
...
private:
   Reference<Matrix4x4> m, mInv;
}
```

save space, but can’t be modified after construction

- Transform stores element of 4x4 matrix
- Also computes and stores matrix inverse, mInv (avoid repeatedly computing inverse)

Transformations

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

$$T(dx,dy,dz)=\begin{pmatrix}1 & 0 & 0 & dx\\ 0 & 1 & 0 & dy\\ 0 & 0 & 1 & dz\\ 0 & 0 & 0 & 1\end{pmatrix}$$

$$R_x(q)=\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & \cos q & -\sin q & 0\\ 0 & \sin q & \cos q & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

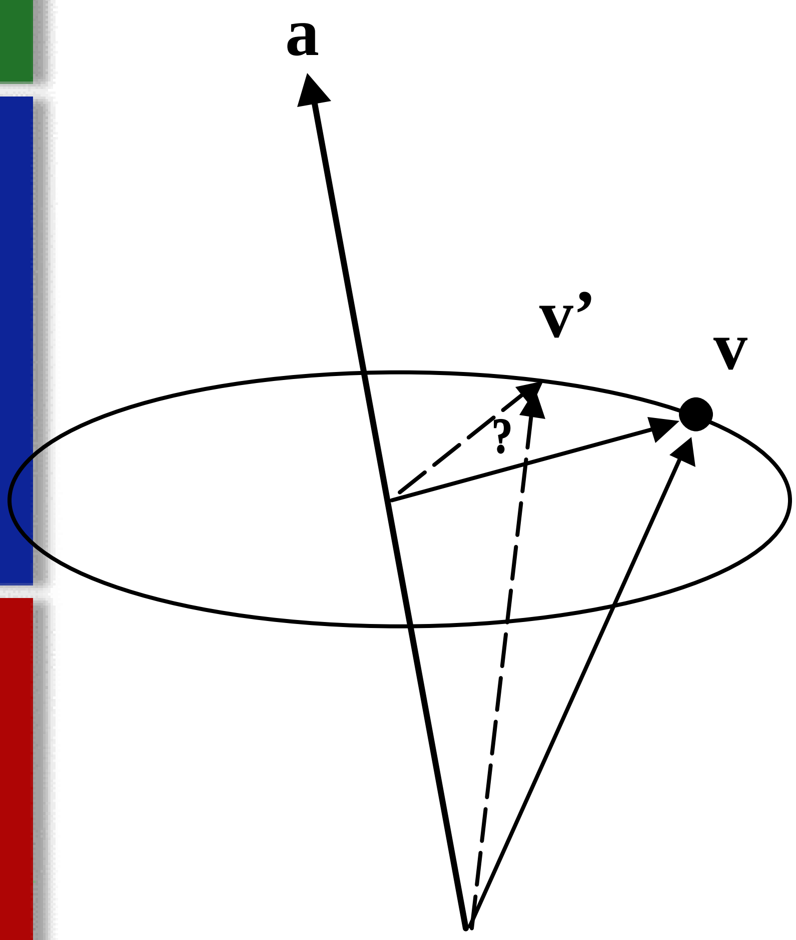
$$S(sx,sy,sz)=\begin{pmatrix}sx & 0 & 0 & 0\\ 0 & sy & 0 & 0\\ 0 & 0 & sy & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

$$R_x(q)^{-1}=R_x(q)^{\mathrm{T}}$$

Question: How does x-roll matrix above differ based on axes handedness?

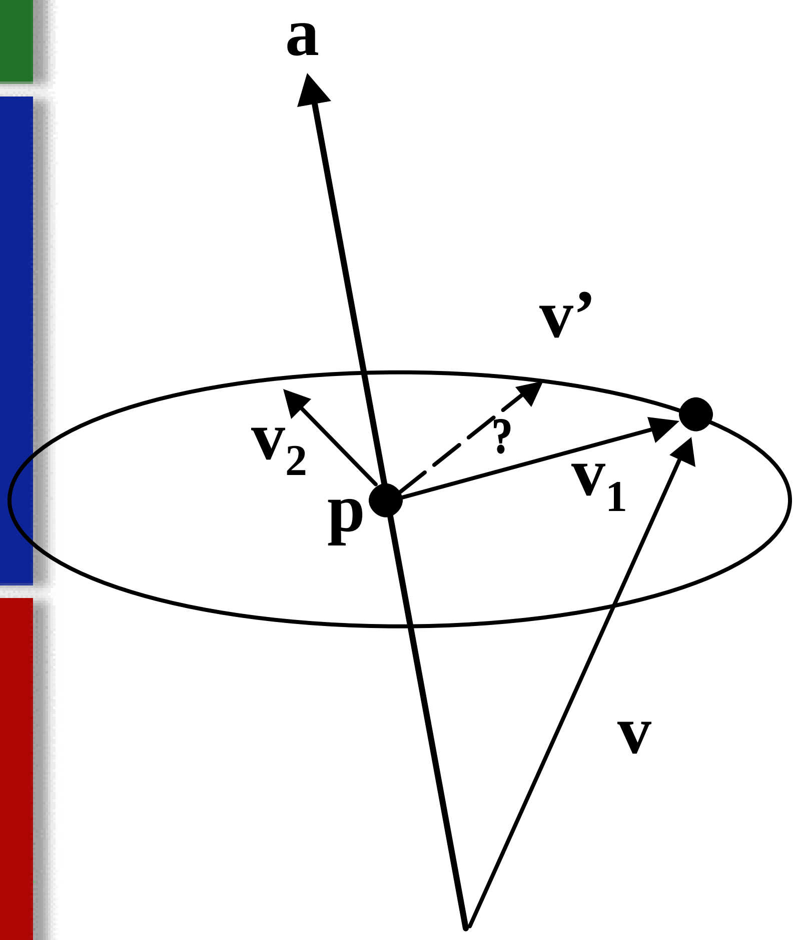
Rotation around an arbitrary axis

- Rotate(a, Vector(1,1,1))



Rotation around an arbitrary axis

- Rotate(a, Vector(1,1,1))



$$\mathbf{p} = \mathbf{a}(\mathbf{v} \cdot \mathbf{a})$$

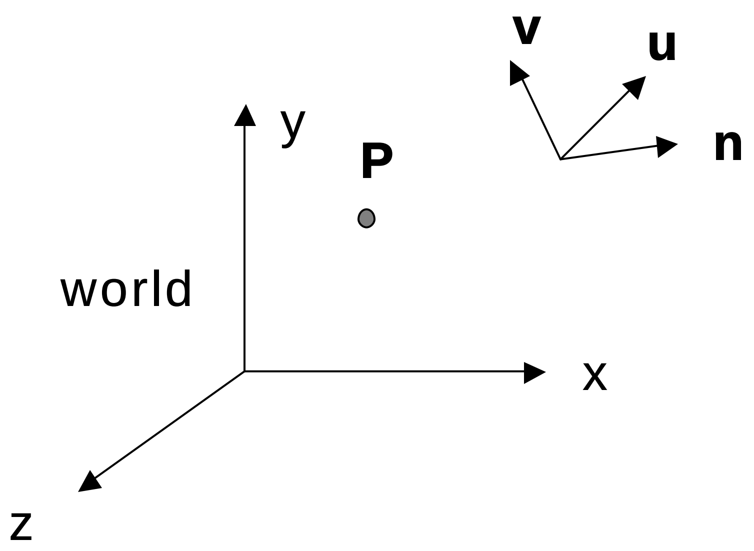
$$\mathbf{v}_1 = \mathbf{v} - \mathbf{p}$$

$$\mathbf{v}_2 = \mathbf{v}_1 \times \mathbf{a} \quad |\mathbf{v}_2| = |\mathbf{v}_1|$$

$$\mathbf{v}' = p + \mathbf{v}_1 \cos q + \mathbf{v}_2 \sin q$$

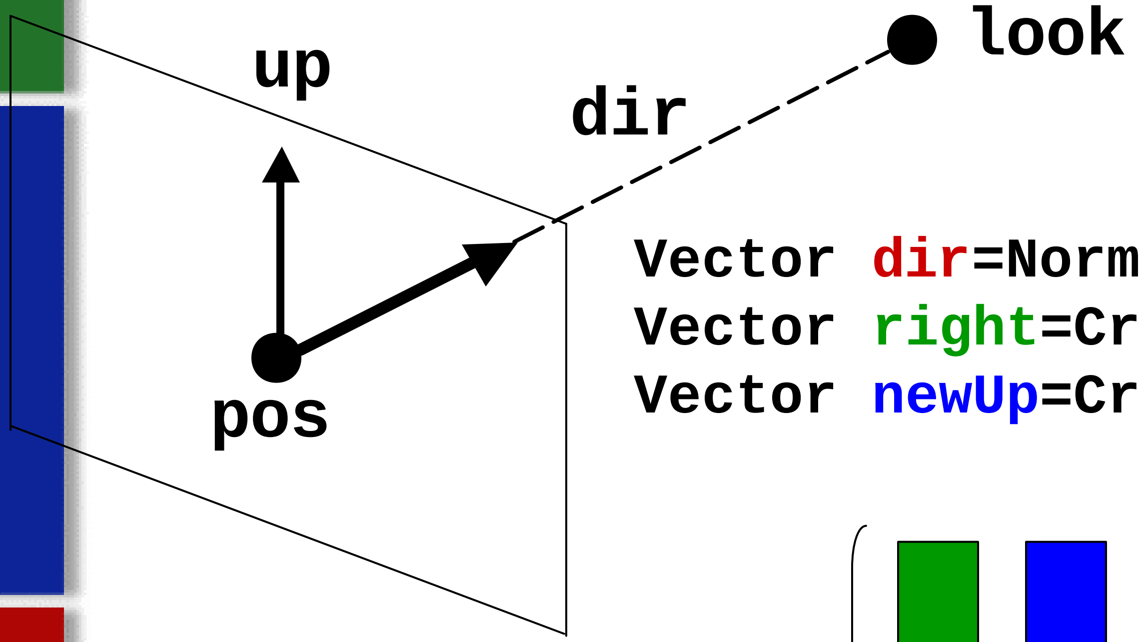
LookAt Transformation

- Caller specifies:
  - camera (eye position),
  - Look at point
  - Up vector
- Want to compute 4x4 transform matrix that converts from world space to eye space

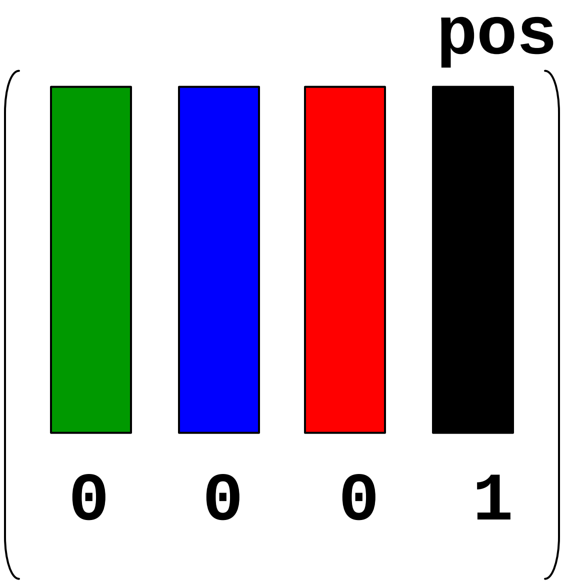


Look-at

- `LookAt(Point &pos, Point look, Vector &up)`



```
Vector dir=Normalize(look-pos);
Vector right=Cross(dir, Normalize(up));
Vector newUp=Cross(right,dir);
```

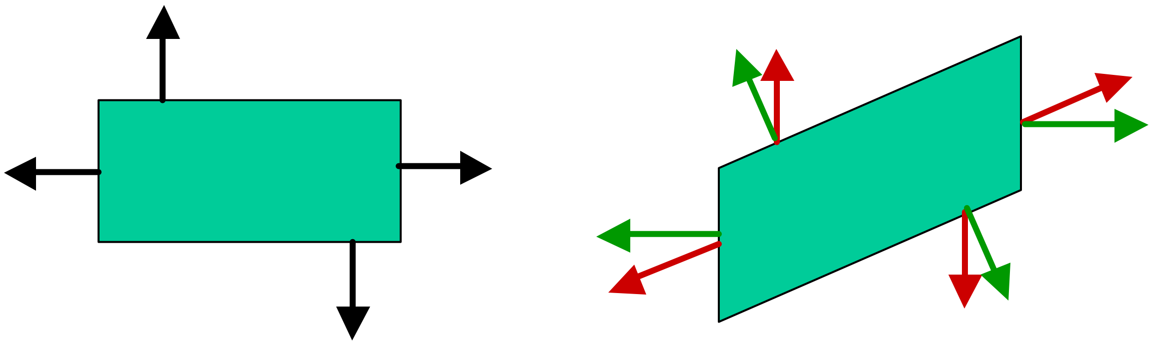


Applying transformations

- **Point**: `q=T(p), T(p,&q)`
  use homogeneous coordinates implicitly

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

- **Vector**: `u=T(v), T(u, &v)`
- **Normal**: treated differently than vectors
  because of anisotropic transformations



$$\mathbf{n}\cdot\mathbf{t}=\mathbf{n}^{\mathrm{T}}\mathbf{t}=0$$

$$(\mathbf{n}')^{\mathrm{T}}\mathbf{t}'=0$$

$$(\mathbf{S}\mathbf{n})^{\mathrm{T}}\mathbf{M}\mathbf{t}=0$$

$$\mathbf{n}^{\mathrm{T}}\mathbf{S}^{\mathrm{T}}\mathbf{M}\mathbf{t}=0$$

$$\mathbf{S}^{\mathrm{T}}\mathbf{M}=\mathbf{I}$$

$$\boxed{\mathbf{S}=\mathbf{M}^{-\mathrm{T}}}$$

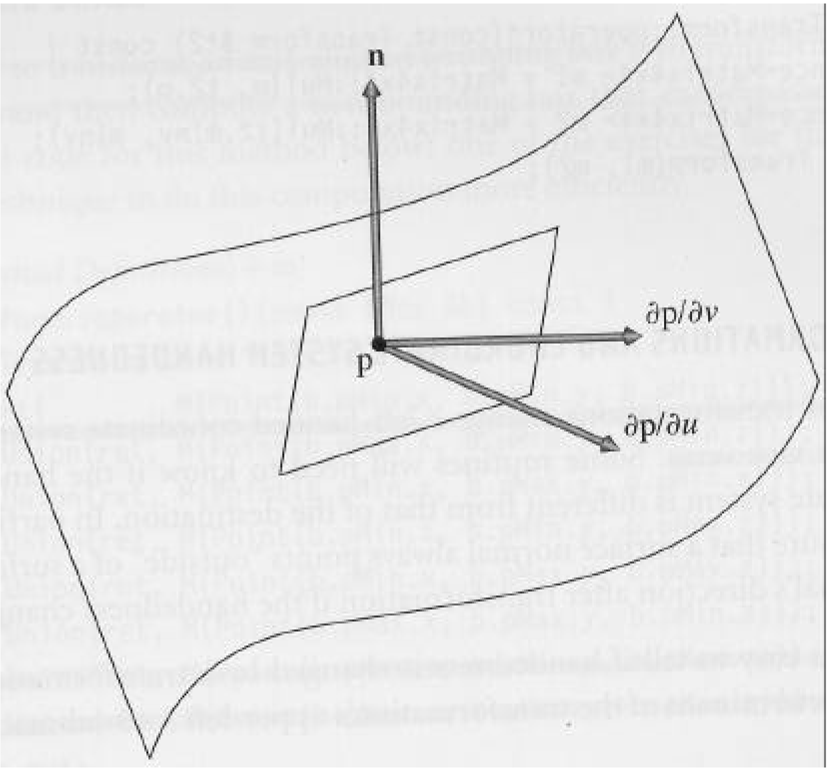
- **Transform** should keep its inverse
- For orthonormal matrix, $\mathbf{S}=\mathbf{M}$

Applying transformations

- **`Transform Bbox?`**
  - transform its 8 corners and expand to include all 8 points.

Differential geometry

- **`DifferentialGeometry`**: a self-contained representation for a particular point on a surface so that all the other operations in pbrt can be executed without referring to the original shape. Contains
- Position
- Surface normal
- Parameterization
- Parametric derivatives
- Derivatives of normals
- Pointer to shape



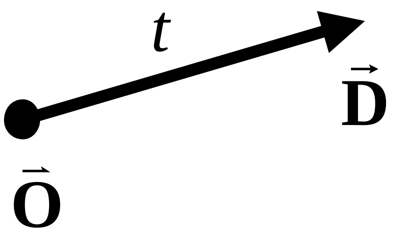
Ray-Surface Intersection

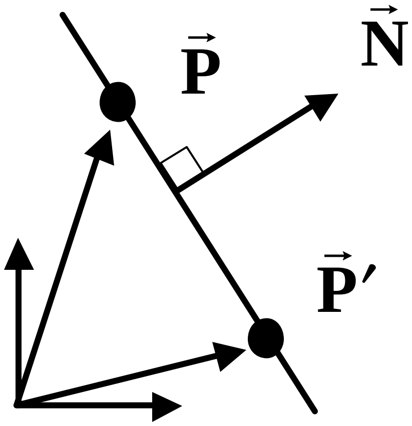
Ray-Plane Intersection

- Ray:

$$\vec{\mathbf{P}} = \vec{\mathbf{O}} + t\,\vec{\mathbf{D}}$$

$$0 \le t < \infty$$





- Plane:
-

$$(\vec{\mathbf{P}} - \vec{\mathbf{P}}') \bullet \vec{\mathbf{N}} = 0$$

$$ax + by + cz + d = 0$$

- Solve for intersection
- Substitute ray equation into plane equation

$$(\vec{\mathbf{P}} - \vec{\mathbf{P}}') \bullet \vec{\mathbf{N}} = (\vec{\mathbf{O}} + t\vec{\mathbf{D}} - \vec{\mathbf{P}}') \bullet \vec{\mathbf{N}} = 0$$

$$t = -\frac{(\vec{\mathbf{O}} - \vec{\mathbf{P}}') \bullet \vec{\mathbf{N}}}{\vec{\mathbf{D}} \bullet \vec{\mathbf{N}}}$$

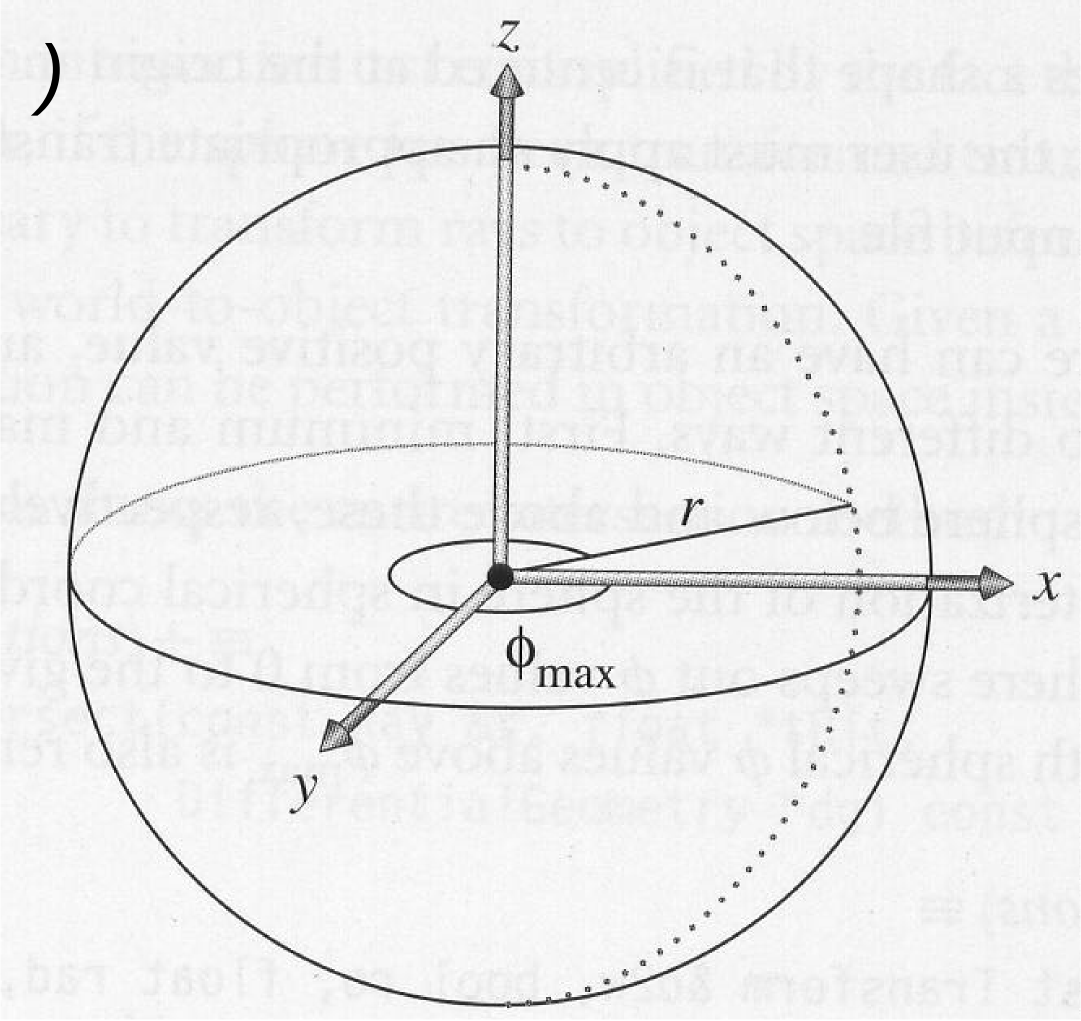
Sphere

- A sphere of radius *r* at the origin
- Implicit: $x^2+y^2+z^2-r^2=0$
- Parametric: *f(*? ,? *)*

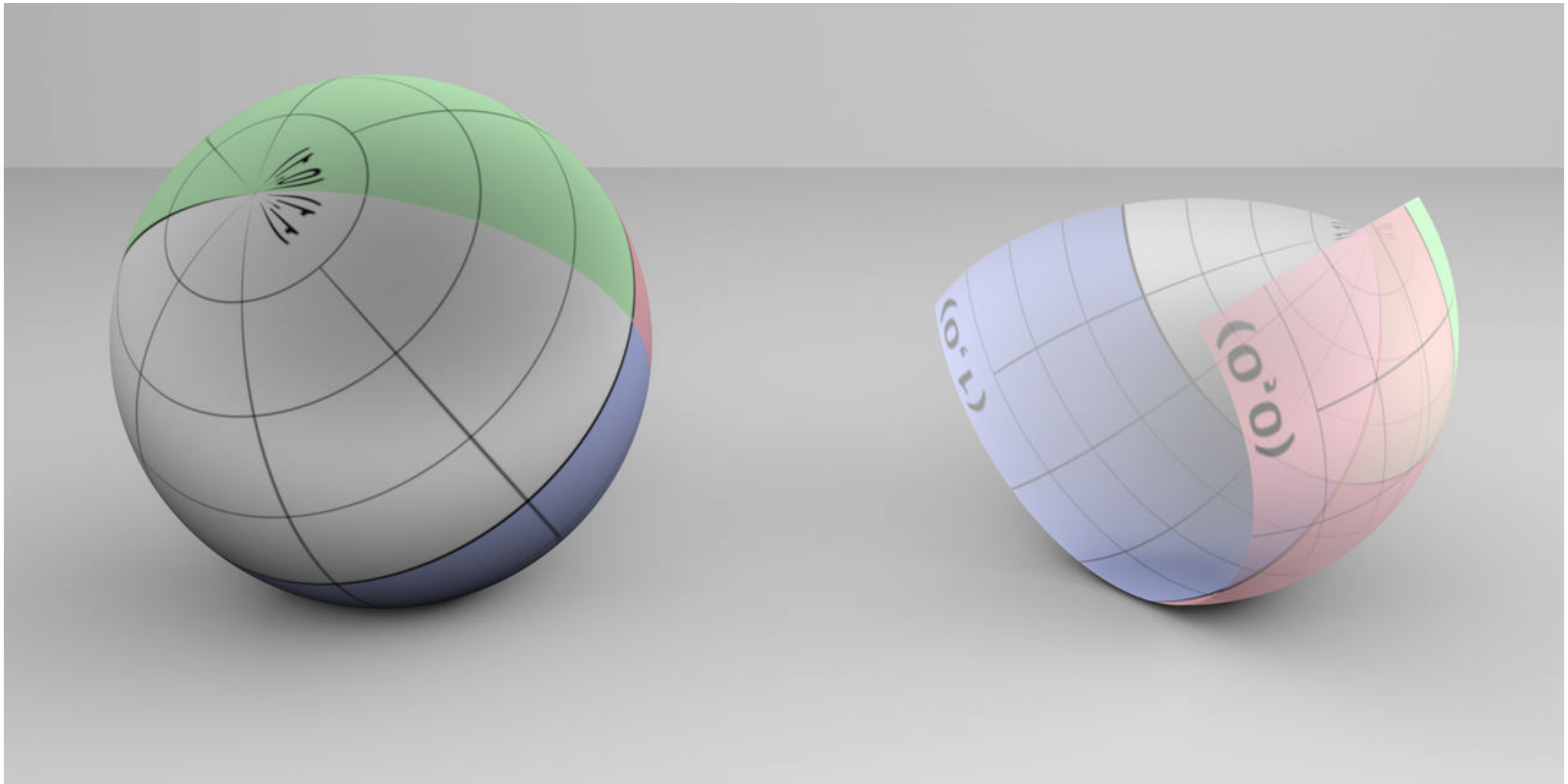
$x=r\sin ?\ \cos ?$

$y=r\sin ?\ \sin ?$

$z=r\cos ?$



Sphere



Algebraic solution

- Perform in object space, **`WorldToObject(r, &ray)`**
- Assume that ray is normalized for a while

$$x^2 + y^2 + z^2 = r^2$$

$$(o_x + td_x)^2 + (o_y + td_y)^2 + (o_z + td_z)^2 = r^2$$

$$At^2 + Bt + C = 0$$

Step 1

$$A = d_x^2 + d_y^2 + d_z^2$$

$$B = 2(d_x o_x + d_y o_y + d_z o_z)$$

$$C = o_x^2 + o_y^2 + o_z^2 - r^2$$

Algebraic solution

$$t_0 = \frac{-B - \sqrt{B^2 - 4AC}}{2A} \qquad t_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

Step 2

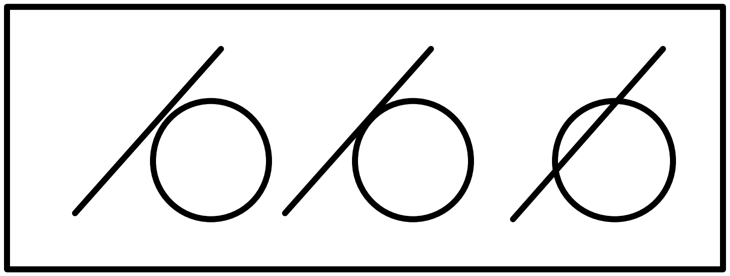
If (B^2-4AC<0) then the ray misses the sphere. B^2-4AC=0?

Step 3

Calculate t_0 and test if $t_0<0$

Step 4

Calculate t_1 and test if $t_1<0$



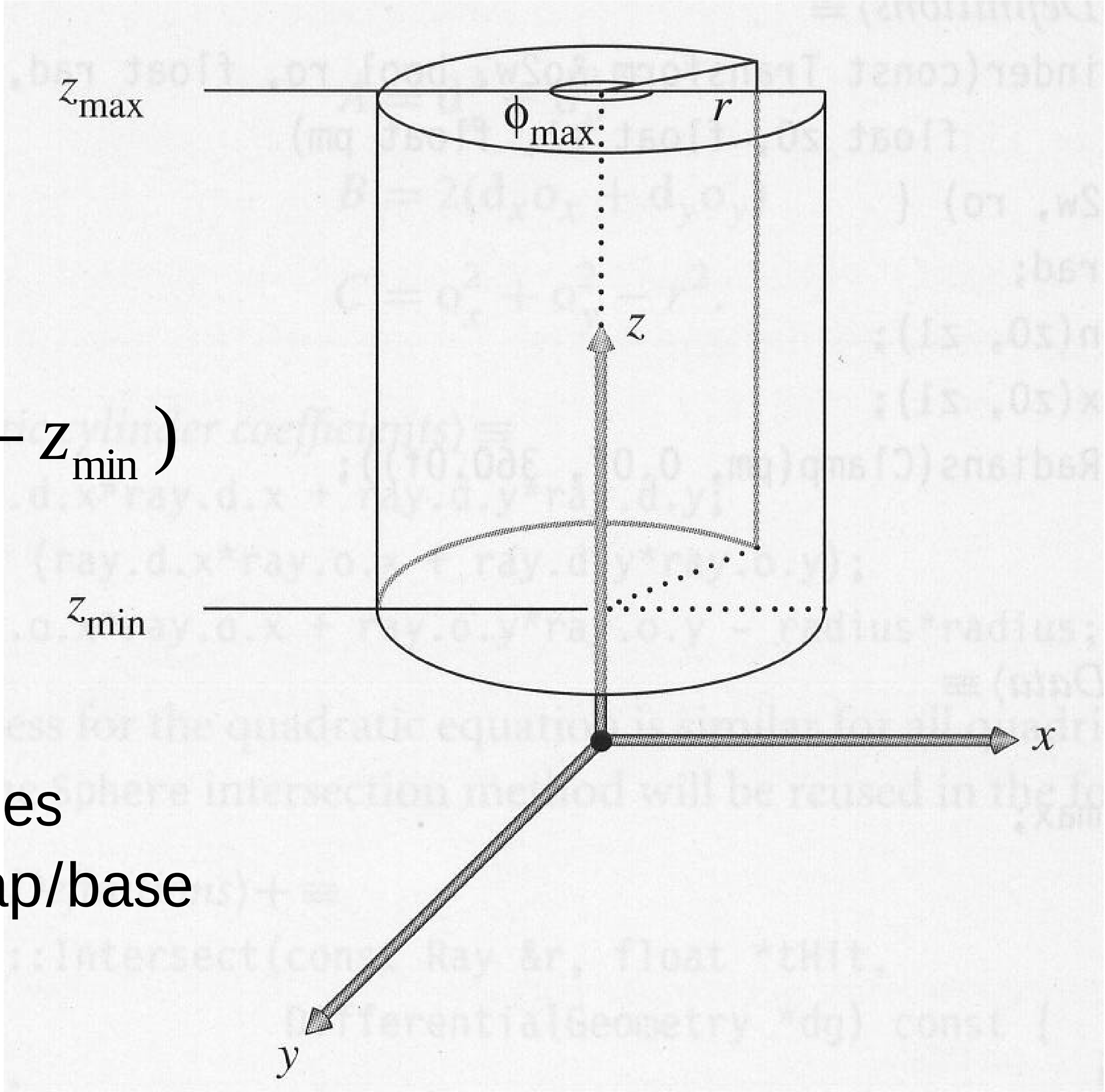
Cylinder

$$f = u f_{\max}$$

$$x = r \cos f$$

$$y = r \sin f$$

$$z = z_{\min} + v(z_{\max} - z_{\min})$$



- First consider sides
- Later consider cap/base

Cylinder

- Implicit equation for cylinder

$$x^2 + y^2 - r^2 = 0$$

- Substituting in ray equation

$$(o_x + td_x)^2 + (o_y + td_y)^2 = r^2$$

- Giving

$$At^2 + Bt + C = 0$$

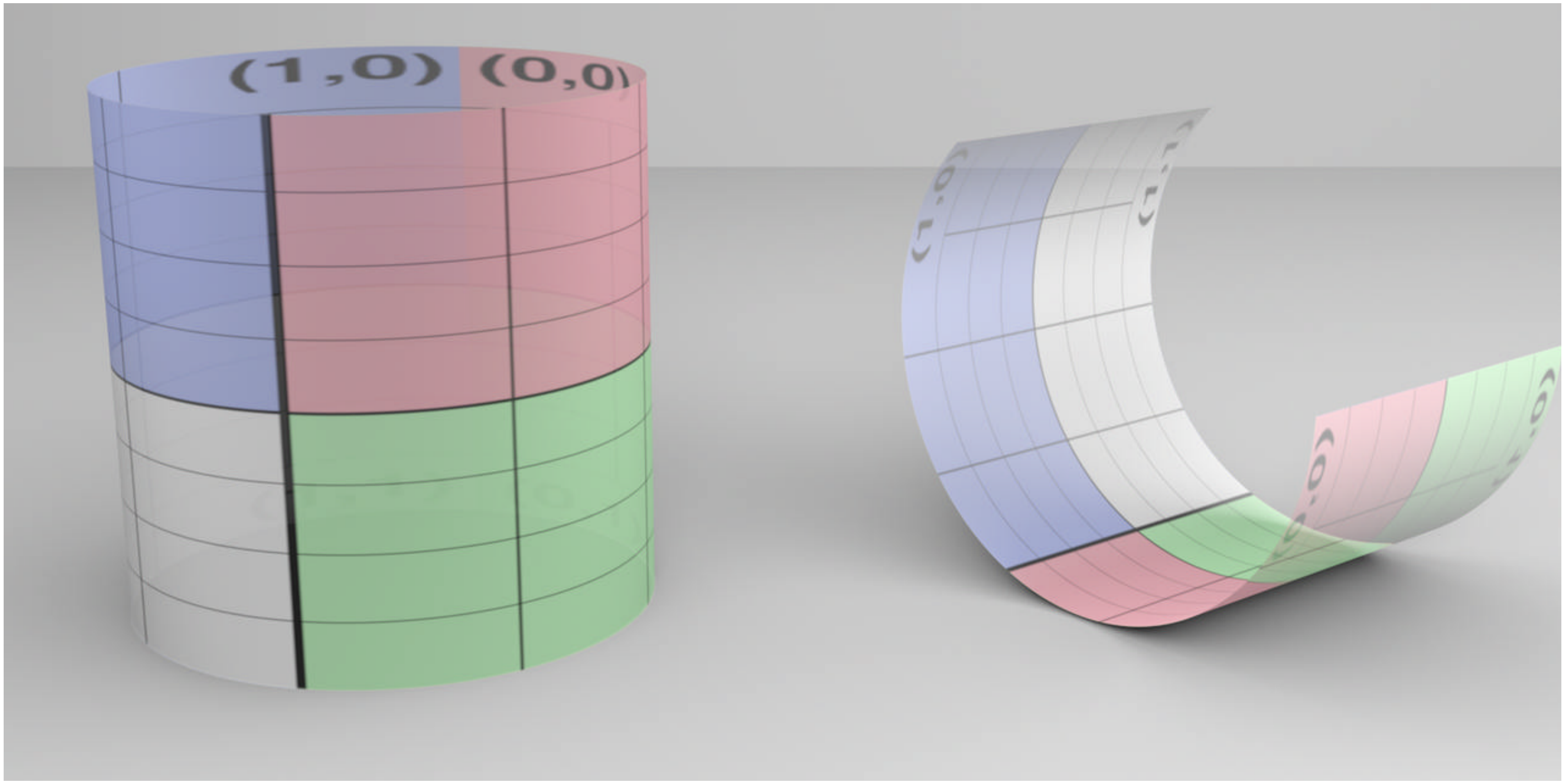
$$A = d_x^2 + d_y^2$$

$$B = 2(d_x o_x + d_y o_y)$$

$$C = o_x^2 + o_y^2 - r^2$$

Solve for t

Cylinder



References/Shamelessly stolen

- Pat Hanrahan, CS 348B, Spring 2005 class slides
- Yung-Yu Chuang, Image Synthesis, class slides, National Taiwan University, Fall 2005
- Kutulakos K, CSC 2530H: Visual Modeling, course slides
- UIUC CS 319, Advanced Computer Graphics Course slides