Physically Based Rendering读书笔记 part 2

计算机图形学

by guopei
原文地址:http://www.pbr-book.org/3ed-2018/contents.html

---

3 Shapes

3.1 Basic Shape Interface

```
<<Shape Declarations>>=
class Shape {
public:
    <<Shape Interface>>
       Shape(const Transform *ObjectToWorld, const Transfo
rm *WorldToObject,
             bool reverseOrientation);
       virtual ~Shape();
       virtual Bounds3f ObjectBound() const = 0;
       virtual Bounds3f WorldBound() const;
       virtual bool Intersect(const Ray &ray, Float *tHit,
           SurfaceInteraction *isect, bool testAlphaTextur
e = true) const = 0;
       virtual bool IntersectP(const Ray &ray,
                bool testAlphaTexture = true) const {
           Float tHit = ray.tMax;
           SurfaceInteraction isect;
           return Intersect(ray, &tHit, &isect, testAlphaT
exture);
       }
       virtual Float Area() const = 0;
       virtual Interaction Sample(const Point2f &u) const
= 0;
       virtual Float Pdf(const Interaction &) const {
```

```
            return 1 / Area();
        }
        virtual Interaction Sample(const Interaction &ref,
                                   const Point2f &u) const
{
            return Sample(u);
        }
        virtual Float Pdf(const Interaction &ref, const Vec
tor3f &wi) const;

    <<Shape Public Data>>
        ///模型空间到世界空间的变换和逆变换矩阵的指针，因为有内存池
设计
        const Transform *ObjectToWorld, *WorldToObject;
        ///法线是否需要翻转
        const bool reverseOrientation;
        const bool transformSwapsHandedness;

};
```

所有的shape都是定义在模型空间的，因此需要有从模型空间到世界空间的变换和逆变换。
Shape构造函数可以接受一个bool型参数：reverseOrientation，标明表面法线是否需要翻转。

```
<<Shape Method Definitions>>=
Shape::Shape(const Transform *ObjectToWorld,
        const Transform *WorldToObject, bool reverseOrient
ation)
    : ObjectToWorld(ObjectToWorld), WorldToObject(WorldToO
bject),
      reverseOrientation(reverseOrientation),
      //这里需要记录ObjectToWorld这个transform是否带有翻转手性
的特性
      transformSwapsHandedness(ObjectToWorld->SwapsHandedn
ess()) {
}
```

3.1.1 Bounding

pbrt中的物体可能极其复杂，这个时候就需要个包围盒，射线检测如果根本没有击中包围盒，就不用再做进一步测试了。这里用Bounds3f来表示AABB。

```
<<Shape Interface>>=
virtual Bounds3f ObjectBound() const = 0;
<<Shape Method Definitions>>+=
Bounds3f Shape::WorldBound() const {
    return (*ObjectToWorld)(ObjectBound());
}
```

WorldBound()函数，返回的是世界空间的包围盒。

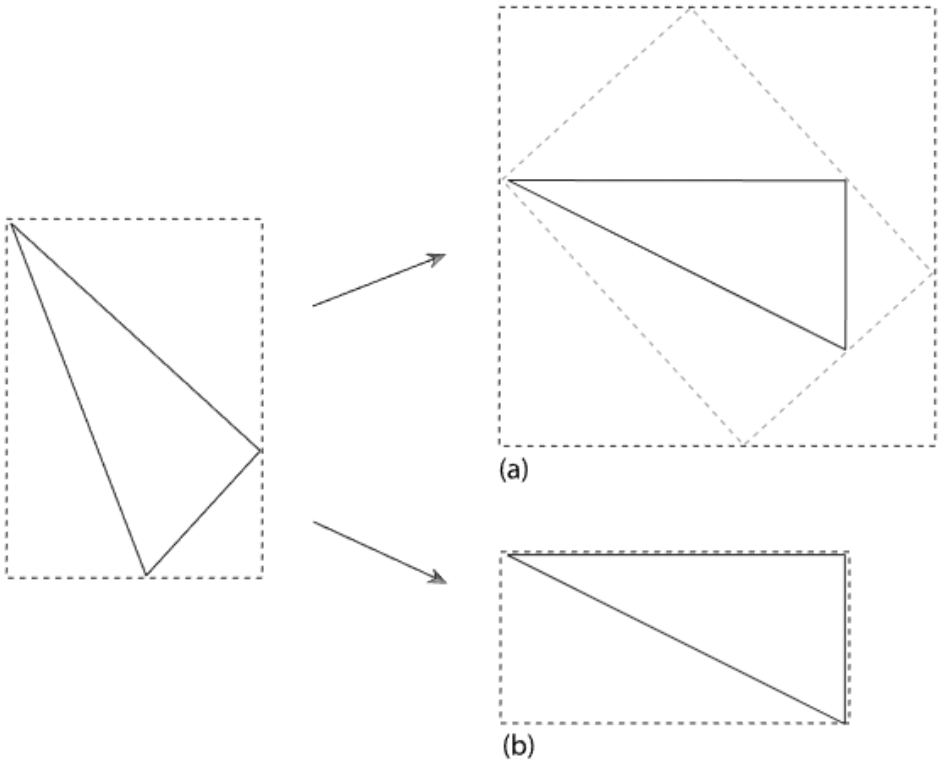


Figure 3.1: (a) A world space bounding box of a triangle is computed by transforming its object space bounding box to world space and then finding the bounding box that encloses the resulting bounding box; a sloppy bound may result. (b) However, if the triangle's vertices are first transformed from object space to world space and then bounded, the fit of the bounding box can be much better.

Shape类提供了一个最基本的包围盒变换(a)，它的派生类可以重载这个函数。

3.1.2 Ray-Bounds Intersections

射线和包围盒相交的函数。

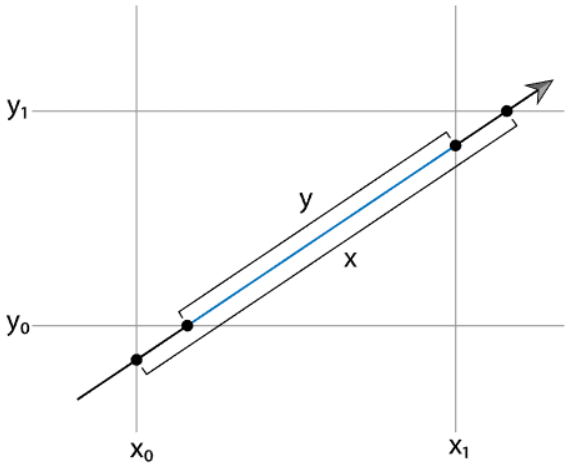


Figure 3.2: Intersecting a Ray with an Axis-Aligned Bounding Box. We compute intersection points with each slab in turn, progressively narrowing the parametric interval. Here, in 2D, the intersection of the x and y extents along the ray (blue segment) gives the extent where the ray is inside the box.

将包围盒的面分为三组，每组是两个互相平行的面(slab)

```
<<Geometry Inline Functions>>+=
template <typename T>
inline bool Bounds3<T>::IntersectP(const Ray &ray, Float *
hitt0,
        Float *hitt1) const {
    Float t0 = 0, t1 = ray.tMax;
    for (int i = 0; i < 3; ++i) {
        <<Update interval for ith bounding box slab>>
        //根据公式计算近端和远端，这里即使invRayDir为无穷大，下面
的代码依然能够正确执行
            Float invRayDir = 1 / ray.d[i];
            Float tNear = (pMin[i] - ray.o[i]) * invRayDir;
            Float tFar  = (pMax[i] - ray.o[i]) * invRayDir;
            <<Update parametric interval from slab intersec
tion  values>>
                if (tNear > tFar) std::swap(tNear, tFar);
                <<Update tFar to ensure robust ray-bounds in
tersection>>
                    tFar *= 1 + 2 * gamma(3);
                  //
                t0 = tNear > t0 ? tNear : t0;
                t1 = tFar  < t1 ? tFar  : t1;
                //如果没有找到正确的区间，直接退出
                if (t0 > t1) return false;

    }
```

```
    if (hitt0) *hitt0 = t0;
    if (hitt1) *hitt1 = t1;
    return true;
}
```

对于每组平面，需要计算两个射线与平面的交点，假设需要计算与x轴垂直的平面，这两个平面分别为$(x_1, 0, 0)$和$(x_2, 0, 0)$，法线为$(1, 0, 0)$，计算平面$ax + by + cz + d = 0$，则需要代入

$$\begin{aligned} 0 &= a(o_x + td_x) + b(o_y + td_y) + c(o_z + td_z) + d \\ &= (a, b, c) \cdot o + t(a, b, c) \cdot d + d \end{aligned}$$

解得

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

因为法线的y和z分量都是0，所以b和c也是0，a为1，系数d为$-x_1$，因此式子简化为

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

Bound3这个类拥有一个特化的IntersectP()函数，拥有一条额外的射线方向作为额外参数。

```
<<Geometry Inline Functions>>+=
template <typename T>
inline bool Bounds3<T>::IntersectP(const Ray &ray, const V
ector3f &invDir,
        const int dirIsNeg[3]) const {
    const Bounds3f &bounds = *this;
    <<Check for ray intersection against  and  slabs>>
       Float tMin =  (bounds[  dirIsNeg[0]].x - ray.o.x) *
 invDir.x;
       Float tMax =  (bounds[1-dirIsNeg[0]].x - ray.o.x) *
 invDir.x;
       Float tyMin = (bounds[  dirIsNeg[1]].y - ray.o.y) *
 invDir.y;
       Float tyMax = (bounds[1-dirIsNeg[1]].y - ray.o.y) *
 invDir.y;
       <<Update tMax and tyMax to ensure robust bounds int
ersection>>
       if (tMin > tyMax || tyMin > tMax)
```

```
            return false;
        if (tyMin > tMin) tMin = tyMin;
        if (tyMax < tMax) tMax = tyMax;

    <<Check for ray intersection against  slab>>
        Float tzMin = (bounds[  dirIsNeg[2]].z - ray.o.z) *
 invDir.z;
        Float tzMax = (bounds[1-dirIsNeg[2]].z - ray.o.z) *
 invDir.z;
        <<Update tzMax to ensure robust bounds intersection
>>
           tzMax *= 1 + 2 * gamma(3);

        if (tMin > tzMax || tzMin > tMax)
            return false;
        if (tzMin > tMin)
            tMin = tzMin;
        if (tzMax < tMax)
            tMax = tzMax;

    return (tMin < ray.tMax) && (tMax > 0);
}
```

3.1.3 Intersection Tests

Shaper::Intersect()函数是各个几何体用来判断射线相交的函数。

```
<<Shape Interface>>+=
virtual bool Intersect(const Ray &ray, Float *tHit,
    SurfaceInteraction *isect, bool testAlphaTexture = tru
e) const = 0;

<<Shape Interface>>+=
virtual bool IntersectP(const Ray &ray,
         bool testAlphaTexture = true) const {
    Float tHit = ray.tMax;
    SurfaceInteraction isect;
    return Intersect(ray, &tHit, &isect, testAlphaTextur
e);
}
```

其中testAlphaTexture是用于让texture的alpha通道来参与碰撞运算，透明的部分穿过去。

3.1.4 Surface Area

```
///计算面积
<<Shape Interface>>+=
virtual Float Area() const = 0;
```

3.1.5 Sideness

pbrt系统不支持这个特性，因为它是基于光线追踪的，正面能射到背面不能射到，在计算shadow ray的时候就会出现问题。

3.2 Spheres

球面是一种二次曲面，用x,y,z的二次多项式来表示。
许多表面可以用一种或两种形式表示，隐式或者参数化形式。
隐式3d表面可以表示为：

$$f(x,y,z)=0$$

所有满足条件的(x,y,z)都在表面上。
对于一个单位半径，中心在原点的球体来说，它符合等式$x^2+y^2+z^2-1=0$
许多表面也可以有参数化公式，将2D的点映射到3D
例如作为一个半径为r的球体，它的2D坐标为(θ,ϕ)，其中θ的区间为$[0,\pi]$，而ϕ的区间为$[0,2\pi]$。

$$\begin{aligned}x&=r\sin\theta\cos\phi\\y&=r\sin\theta\sin\phi\\z&=r\cos\theta\end{aligned}$$

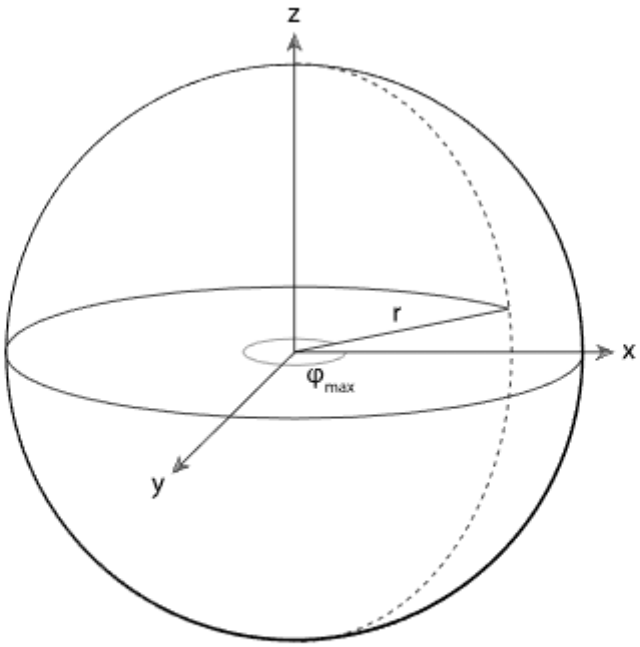


Figure 3.4: Basic Setting for the Sphere Shape. It has a radius of r and is centered at the object space origin. A partial sphere may be described by specifying a maximum ϕ value.

我能将函数$f(\theta, \phi)$转化为函数$f(u, v)$，在$[0, 1]^2$空间内。也能取部分球体，通过限制$\theta \in [\theta_{min}, \theta_{max}]$，以及$\phi \in [0, \phi_{max}]$，通过如下替换：

$$\phi = u\phi_{max}$$
$$\theta = \theta_{min} + v(\theta_{max} - \theta_{min})$$

这种形式可以很简单地进行纹理映射。

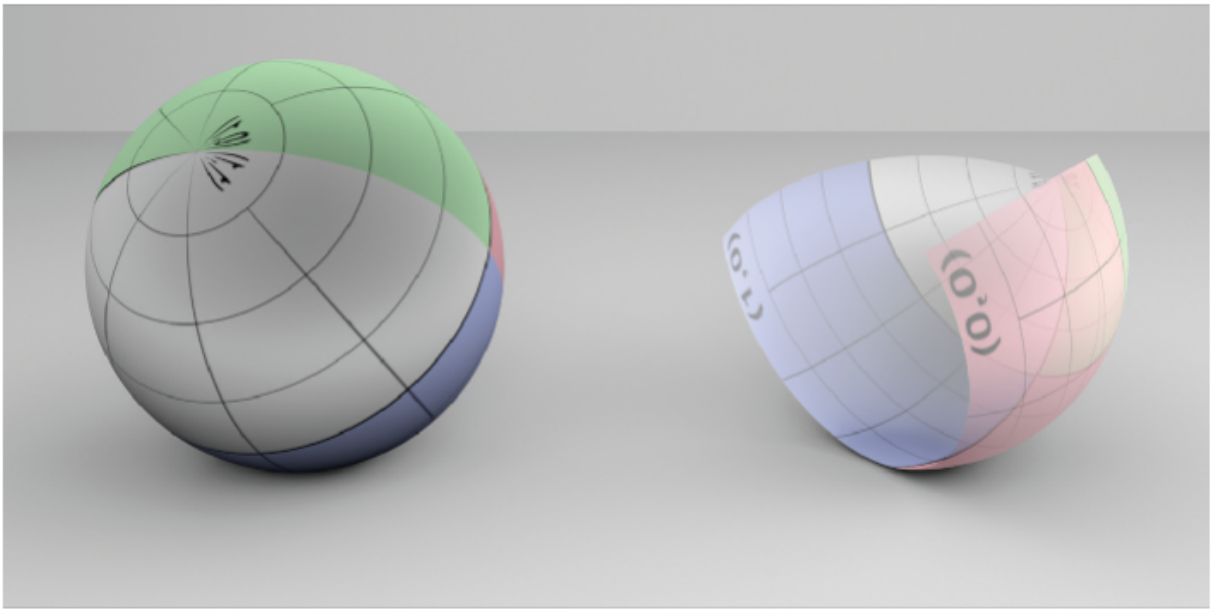


Figure 3.5: Two Spheres. On the left is a complete sphere, and on the right is a partial sphere (with $z_{\max} < r$ and $\phi_{\max} < 2\pi$). Note that the texture map used shows the (u, v) parameterization of the shape; the singularity at one of the poles is visible in the complete sphere.

```
<<Sphere Declarations>>=
class Sphere : public Shape {
public:
    <<Sphere Public Methods>>
    //允许在z方向上限制最大值和最小值
       Sphere(const Transform *ObjectToWorld, const Transf
orm *WorldToObject,
              bool reverseOrientation, Float radius, Float
 zMin, Float zMax,
              Float phiMax)
          : Shape(ObjectToWorld, WorldToObject, reverseOr
ientation),
            radius(radius), zMin(Clamp(std::min(zMin, zMa
x), -radius, radius)),
            zMax(Clamp(std::max(zMin, zMax), -radius, rad
ius)),
            thetaMin(std::acos(Clamp(zMin / radius, -1,
1))),
            thetaMax(std::acos(Clamp(zMax / radius, -1,
1))),
            phiMax(Radians(Clamp(phiMax, 0, 360))) { }
       Bounds3f ObjectBound() const;
       bool Intersect(const Ray &ray, Float *tHit, Surface
Interaction *isect,
           bool testAlphaTexture) const;
       bool IntersectP(const Ray &ray, bool testAlphaTextu
re) const;
       Float Area() const;
       Interaction Sample(const Point2f &u) const;
       Interaction Sample(const Interaction &ref, const Po
int2f &u) const;
       Float Pdf(const Interaction &ref, const Vector3f &w
i) const;

private:
    <<Sphere Private Data>>
       const Float radius;
       const Float zMin, zMax;
       const Float thetaMin, thetaMax, phiMax;

};
```

3.2.1 Bounding

```
<<Sphere Method Definitions>>=
Bounds3f Sphere::ObjectBound() const {
    return Bounds3f(Point3f(-radius, -radius, zMin),
                    Point3f( radius,  radius, zMax));
}
```

很容易理解

3.2.2 Intersection Test

完整代码如下：

```
<<Sphere Method Definitions>>+=
bool Sphere::Intersect(const Ray &r, Float *tHit,
        SurfaceInteraction *isect, bool testAlphaTexture)
const {
    Float phi;
    Point3f pHit;
    <<Transform Ray to object space>>
       Vector3f oErr, dErr;
       Ray ray = (*WorldToObject)(r, &oErr, &dErr);

    <<Compute quadratic sphere coefficients>>
       <<Initialize EFloat ray coordinate values>>
       //EFloat类型保存了额外的误差参数
          EFloat ox(ray.o.x, oErr.x), oy(ray.o.y, oErr.y),
 oz(ray.o.z, oErr.z);
          EFloat dx(ray.d.x, dErr.x), dy(ray.d.y, dErr.y),
 dz(ray.d.z, dErr.z);

       EFloat a = dx * dx + dy * dy + dz * dz;
       EFloat b = 2 * (dx * ox + dy * oy + dz * oz);
       EFloat c = ox * ox + oy * oy + oz * oz - EFloat(rad
ius) * EFloat(radius);

    <<Solve quadratic equation for t values>>
       EFloat t0, t1;
```

```
        //计算射线与球的交点，可能有0个，1个，2个交点
        if (!Quadratic(a, b, c, &t0, &t1))
            return false;
            //如果第一个交点比射线最大距离还大或者第二个交点小于0，
则直接退出
        <<Check quadric shape t0 and t1 for nearest interse
ction>>
           if (t0.UpperBound() > ray.tMax || t1.LowerBound
() <= 0)
               return false;
           //交点选比较近的那个
           EFloat tShapeHit = t0;
           if (tShapeHit.LowerBound() <= 0) {
               tShapeHit = t1;
               if (tShapeHit.UpperBound() > ray.tMax)
                   return false;
           }

    <<Compute sphere hit position and >>
        pHit = ray((Float)tShapeHit);
        //可能因为浮点误差的原因，击中的点落到了球体的另一面，这里来
处理浮点误差，算法后面会提到
        <<Refine sphere intersection point>>
           pHit *= radius / Distance(pHit, Point3f(0, 0,
0));

        if (pHit.x == 0 && pHit.y == 0) pHit.x = 1e-5f * ra
dius;
        //计算出对应的phi值
        phi = std::atan2(pHit.y, pHit.x);
        if (phi < 0) phi += 2 * Pi;

    //检测clip参数
    <<Test sphere intersection against clipping parameters
>>=
    if ((zMin > -radius && pHit.z < zMin) ||
         (zMax <  radius && pHit.z > zMax) || phi > phiMax)
 {
         //如果已经选了t1了，没有备选了，直接返回
         if (tShapeHit == t1) return false;
         //如果t1不合适，也得直接返回
```

```
        if (t1.UpperBound() > ray.tMax) return false;
        //选t1
        tShapeHit = t1;
        //重新计算pHit
        <<Compute sphere hit position and >>
           pHit = ray((Float)tShapeHit);
           <<Refine sphere intersection point>>
              pHit *= radius / Distance(pHit, Point3f(0,
0, 0));

           if (pHit.x == 0 && pHit.y == 0) pHit.x = 1e-5f
 * radius;
           phi = std::atan2(pHit.y, pHit.x);
           if (phi < 0) phi += 2 * Pi;
        //如果还不合适，则直接返回
        if ((zMin > -radius && pHit.z < zMin) ||
            (zMax <  radius && pHit.z > zMax) || phi > phi
Max)
            return false;
    }
    //接下来就要计算相关的偏导数
    <<Find parametric representation of sphere hit>>=
    //首先计算出在phi，z和theta范围内的uv
    Float u = phi / phiMax;
    Float theta = std::acos(Clamp(pHit.z / radius, -1,
1));
    Float v = (theta - thetaMin) / (thetaMax - thetaMin);
    //计算偏导dp/du,dp/dv
    <<Compute sphere  and >>
       Float zRadius = std::sqrt(pHit.x * pHit.x + pHit.y
 * pHit.y);
       Float invZRadius = 1 / zRadius;
       Float cosPhi = pHit.x * invZRadius;
       Float sinPhi = pHit.y * invZRadius;
       Vector3f dpdu(-phiMax * pHit.y, phiMax * pHit.x,
0);
       Vector3f dpdv = (thetaMax - thetaMin) *
           Vector3f(pHit.z * cosPhi, pHit.z * sinPhi,
                    -radius * std::sin(theta));
    //计算dn/du以及dn/dv
    <<Compute sphere  and >>
```

```
        Vector3f d2Pduu = -phiMax * phiMax * Vector3f(pHit.
x, pHit.y, 0);
        Vector3f d2Pduv = (thetaMax - thetaMin) * pHit.z *
 phiMax *
                          Vector3f(-sinPhi, cosPhi, 0.);
        Vector3f d2Pdvv = -(thetaMax - thetaMin) * (thetaMa
x - thetaMin) *
                          Vector3f(pHit.x, pHit.y, pHit.z);
        <<Compute coefficients for fundamental forms>>
           Float E = Dot(dpdu, dpdu);
           Float F = Dot(dpdu, dpdv);
           Float G = Dot(dpdv, dpdv);
           Vector3f N = Normalize(Cross(dpdu, dpdv));
           Float e = Dot(N, d2Pduu);
           Float f = Dot(N, d2Pduv);
           Float g = Dot(N, d2Pdvv);

        <<Compute  and  from fundamental form coefficients>
>
           Float invEGF2 = 1 / (E * G - F * F);
           Normal3f dndu = Normal3f((f * F - e * G) * invEG
F2 * dpdu +
                                    (e * F - f * E) * invEG
F2 * dpdv);
           Normal3f dndv = Normal3f((g * F - f * G) * invEG
F2 * dpdu +
                                    (f * F - g * E) * invEG
F2 * dpdv);

    <<Compute error bounds for sphere intersection>>
        Vector3f pError = gamma(5) * Abs((Vector3f)pHit);

    <<Initialize SurfaceInteraction from parametric inform
ation>>
        *isect = (*ObjectToWorld)(
            SurfaceInteraction(pHit, pError, Point2f(u, v),
 -ray.d, dpdu, dpdv,
                               dndu, dndv, ray.time,
this));

    <<Update tHit for quadric intersection>>
```

```
        *tHit = (Float)tShapeHit;

    return true;
}
```

如果球体圆心在原点，并且半径为r，那么隐式方程为

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

射线方程代入

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

除了t都是已知的，因此这个式子是t的二次函数，可以变为如下形式：

$$at^2 + bt + c = 0$$

其中

$$\begin{aligned} a &= d_x^2 + d_y^2 + d_z^2 \\ b &= 2(d_x o_x + d_y o_y + d_z o_z) \\ c &= o_x^2 + o_y^2 + o_z^2 - r^2 \end{aligned}$$

处理圆球部分片段也是必要的工作，在类中定义了z的范围和ϕ的范围，如果射线击中了clip掉的区域，则不能算击中。使用球体的参数化表示

$$\frac{y}{x} = \frac{r \sin\theta \sin\phi}{r \sin\theta \cos\phi} = \tan\phi$$

所以$\phi = \arctan\frac{y}{x}$。同时也有必要映射标准库中的std::atan()函数到$[0, 2\pi]$区间，以满足球体的定义。
对于计算偏导，这里以$\frac{\partial p}{\partial u}$的x分量，$\frac{\partial p_x}{\partial u}$为例。

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ \frac{\partial p_x}{\partial u} &= \frac{\partial}{\partial u}(r \sin\theta \cos\phi) \\ &= r \sin\theta \frac{\partial}{\partial u}(\cos\phi) \\ &= r \sin\theta(-\phi_{max} \sin\phi) \end{aligned}$$

由于

$$y = r \sin\theta \sin\phi$$

因此

$$\frac{\partial p_x}{\partial u} = -\phi_{max} y$$

类似的

$$\frac{\partial p_y}{\partial u} = \phi_{max} x$$

以及

$$\frac{\partial p_z}{\partial u} = 0$$

我们也可以用相似的办法计算出$\partial p/\partial v$，因此最终结果为

$$\frac{\partial p}{\partial u} = (-\phi_{max} y, \phi_{max} x, 0)$$

$$\frac{\partial p}{\partial v} = (\theta_{max} - \theta_{min})(z\cos\phi, z\sin\phi, -r\sin\theta)$$

3.2.3 Partial Derivatives of Normal Vectors *

暂略

3.2.4 SurfaceInteraction Initialization

有了表面参数以及相关的偏导数，SurfaceInteraction结构体就可以被几何体信息初始化了。

```
<<Initialize SurfaceInteraction from parametric informatio
n>>=
*isect = (*ObjectToWorld)(
    SurfaceInteraction(pHit, pError, Point2f(u, v), -ray.
d, dpdu, dpdv,
                       dndu, dndv, ray.time, this));
//射线检测的最后，要给tHit赋值
<<Update tHit for quadric intersection>>=
*tHit = (Float)tShapeHit;
```

tHit的含义，是在射线方向上离射线起点的距离，这里直接用tShapeHit来赋值。
这里值得思考的问题是，ObjectToWorld函数对于这个距离，会造成什么样的影响。很明显的，对于一个参数化距离而言，它可以被平移、缩放、旋转，然而很明显的，无论在object space还是world space，一个射线到一个几何体的距离都应该是相同的，因此这

里直接赋值。但是必须注意的是，一旦一条射线转换到了object space，就不能再将其归一化，归一化必然就错了。

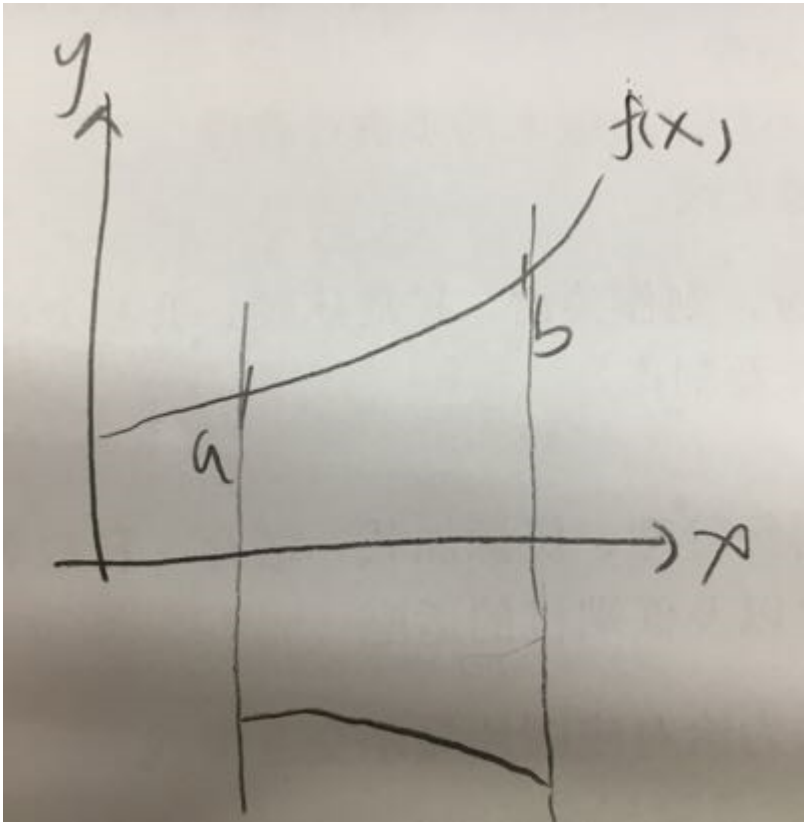
```
<<Sphere Method Definitions>>+=
bool Sphere::IntersectP(const Ray &r, bool testAlphaTextur
e) const {
    Float phi;
    Point3f pHit;
    <<Transform Ray to object space>>
       Vector3f oErr, dErr;
       Ray ray = (*WorldToObject)(r, &oErr, &dErr);

    <<Compute quadratic sphere coefficients>>
       <<Initialize EFloat ray coordinate values>>
       EFloat a = dx * dx + dy * dy + dz * dz;
       EFloat b = 2 * (dx * ox + dy * oy + dz * oz);
       EFloat c = ox * ox + oy * oy + oz * oz - EFloat(rad
ius) * EFloat(radius);

    <<Solve quadratic equation for t values>>
       EFloat t0, t1;
       if (!Quadratic(a, b, c, &t0, &t1))
           return false;
       <<Check quadric shape t0 and t1 for nearest interse
ction>>
          if (t0.UpperBound() > ray.tMax || t1.LowerBound
() <= 0)
              return false;
          EFloat tShapeHit = t0;
          if (tShapeHit.LowerBound() <= 0) {
              tShapeHit = t1;
              if (tShapeHit.UpperBound() > ray.tMax)
                  return false;
          }

    <<Compute sphere hit position and >>
       pHit = ray((Float)tShapeHit);
       <<Refine sphere intersection point>>
       if (pHit.x == 0 && pHit.y == 0) pHit.x = 1e-5f * ra
dius;
       phi = std::atan2(pHit.y, pHit.x);
```

```
        if (phi < 0) phi += 2 * Pi;

    <<Test sphere intersection against clipping parameters
>>
        if ((zMin > -radius && pHit.z < zMin) ||
            (zMax <  radius && pHit.z > zMax) || phi > phiM
ax) {
            if (tShapeHit == t1) return false;
            if (t1.UpperBound() > ray.tMax) return false;
            tShapeHit = t1;
            <<Compute sphere hit position and >>
            if ((zMin > -radius && pHit.z < zMin) ||
                (zMax <  radius && pHit.z > zMax) || phi >
 phiMax)
                return false;
        }

    return true;
}
```

Sphere::IntersectP()函数与Sphere::Intersect()基本上一模一样。但这个函数不需要返回交点，所以也就不用初始化SurfaceIntersection结构体了。

3.2.5 Surface Area

计算一个二次曲面的面积，通过积分来进行运算。如果一条曲线的函数为$y = f(x)$，其中x的区间为[a,b]，把这条曲线绕着x轴旋转一周形成了一个二次曲面，



它的面积公式为：

$$2\pi \int_a^b f(x)\sqrt{1+(f^{'}(x))^2}dx$$

其中，$f^{'}(x)=df/dx$。球体根据ϕ只取一部分的话，公式就变为

$$\phi_{max} \int_a^b f(x)\sqrt{1+(f^{'}(x))^2}dx$$

球体是一个圆环环绕而成，因此在沿着z轴的环曲线表达式为

$$f(z)=\sqrt{r^2-z^2}$$

其导数为

$$f^{'}(z)=-\frac{z}{\sqrt{r^2-z^2}}$$

特别的，z是有范围的，区间为$[z_{min},z_{max}]$，因此表面积的表达式为

$$
\begin{aligned}
A &= \phi_{max} \int_{z_{min}}^{z_{max}} \sqrt{r^2 - z^2} \sqrt{1 + \frac{z^2}{r^2 - z^2}} dz \\
&= \phi_{max} \int_{z_{min}}^{z_{max}} \sqrt{r^2 - z^2 + z^2} dz \\
&= \phi_{max} \int_{z_{min}}^{z_{max}} r dz \\
&= \phi_{max} r (z_{max} - z_{min})
\end{aligned}
$$

对于整个圆球而言，$\phi_{max} = 2\pi, z_{min} = -r, z_{max} = r$，此时$A = 4\pi r^2$。

```
<<Sphere Method Definitions>>+=
Float Sphere::Area() const {
    return phiMax * radius * (zMax - zMin);
}
```

3.3 Cylinders

对于圆柱体而言，z轴有个范围，ϕ有一个范围。

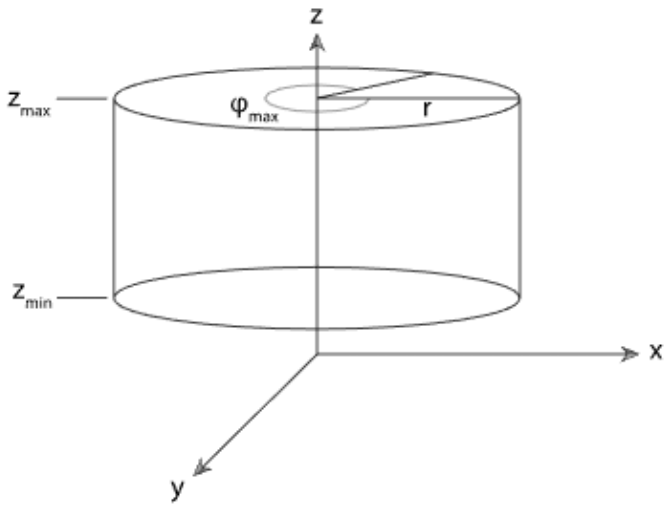


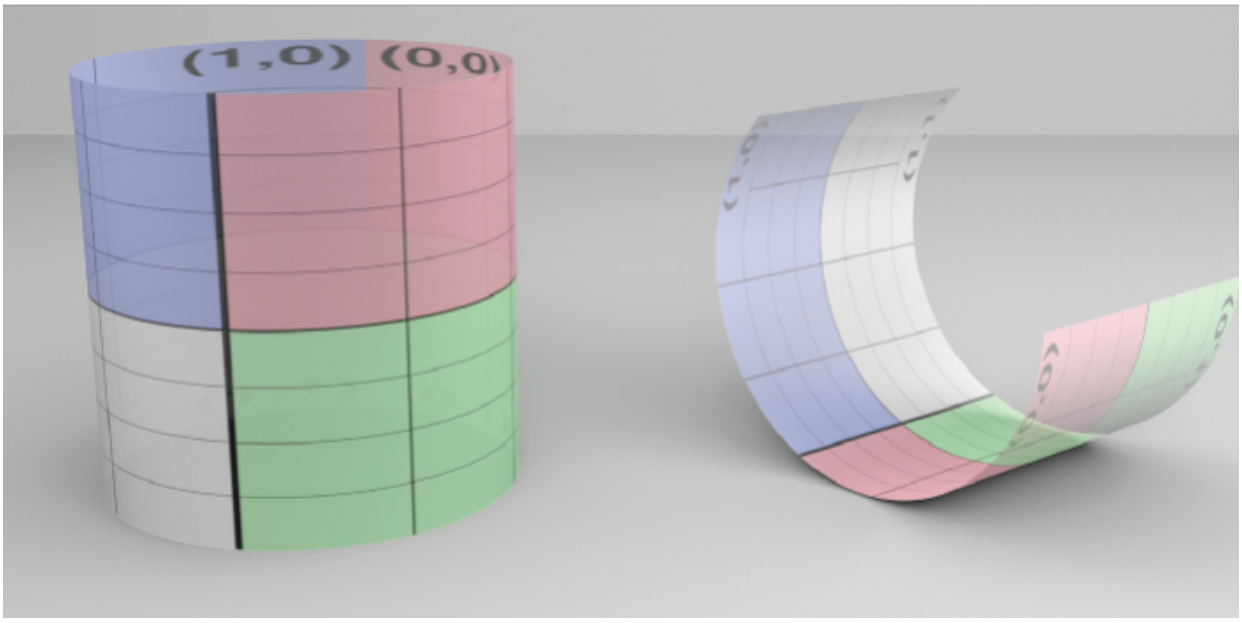
Figure 3.6: Basic Setting for the Cylinder Shape. It has a radius of r and covers a range along the z axis. A partial cylinder may be swept by specifying a maximum ϕ value.

```
<<Cylinder Declarations>>=
class Cylinder : public Shape {
public:
    <<Cylinder Public Methods>>
```

```
        Cylinder(const Transform *ObjectToWorld, const Tran
sform *WorldToObject,
                 bool reverseOrientation, Float radius, Flo
at zMin, Float zMax,
                 Float phiMax)
            : Shape(ObjectToWorld, WorldToObject, reverseOr
ientation),
              radius(radius), zMin(std::min(zMin, zMax)),
              zMax(std::max(zMin, zMax)),
              phiMax(Radians(Clamp(phiMax, 0, 360))) { }
        Bounds3f ObjectBound() const;
        bool Intersect(const Ray &ray, Float *tHit, Surface
Interaction *isect,
            bool testAlphaTexture) const;
        bool IntersectP(const Ray &ray, bool testAlphaTextu
re) const;
        Float Area() const;
        Interaction Sample(const Point2f &u) const;

protected:
    <<Cylinder Private Data>>
        const Float radius, zMin, zMax, phiMax;

};
```

参数化形式为：

$$
\begin{aligned}
\phi &= u\phi_{max} \\
x &= r\cos\theta \\
y &= r\sin\theta \\
z &= z_{min} + v(z_{max} - z_{min})
\end{aligned}
$$



3.3.1 Bounding

```
<<Cylinder Method Definitions>>=
Bounds3f Cylinder::ObjectBound() const {
    return Bounds3f(Point3f(-radius, -radius, zMin),
                    Point3f( radius,  radius, zMax));
}
```

非常简单

3.3.2 Intersection Tests

射线与圆柱体的碰撞检测公式，可以将射线公式直接代入圆柱体隐表达式。

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

将射线公式代入

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

整理为$at^2 + bt + c$的形式，得到

$$\begin{aligned} a &= d_x^2 + d_y^2 \\ b &= 2(d_x o_x + d_x o_x) \\ c &= o_x^2 + o_y^2 - r^2 \end{aligned}$$

对应代码

```
<<Compute quadratic cylinder coefficients>>=
```

```
<<Initialize EFloat ray coordinate values>>
   EFloat ox(ray.o.x, oErr.x), oy(ray.o.y, oErr.y), oz(ra
y.o.z, oErr.z);
   EFloat dx(ray.d.x, dErr.x), dy(ray.d.y, dErr.y), dz(ra
y.d.z, dErr.z);

EFloat a = dx * dx + dy * dy;
EFloat b = 2 * (dx * ox + dy * oy);
EFloat c = ox * ox + oy * oy - EFloat(radius) * EFloat(rad
ius);
```

下面是射线和圆柱体相交的代码片段

```
<<Cylinder Method Definitions>>+=
bool Cylinder::Intersect(const Ray &r, Float *tHit,
        SurfaceInteraction *isect, bool testAlphaTexture)
const {
    Float phi;
    Point3f pHit;
    <<Transform Ray to object space>>
       Vector3f oErr, dErr;
       Ray ray = (*WorldToObject)(r, &oErr, &dErr);

    <<Compute quadratic cylinder coefficients>>
       <<Initialize EFloat ray coordinate values>>
          EFloat ox(ray.o.x, oErr.x), oy(ray.o.y, oErr.y),
 oz(ray.o.z, oErr.z);
          EFloat dx(ray.d.x, dErr.x), dy(ray.d.y, dErr.y),
 dz(ray.d.z, dErr.z);

       EFloat a = dx * dx + dy * dy;
       EFloat b = 2 * (dx * ox + dy * oy);
       EFloat c = ox * ox + oy * oy - EFloat(radius) * EFl
oat(radius);

    <<Solve quadratic equation for t values>>
       EFloat t0, t1;
       //解二次方程
       if (!Quadratic(a, b, c, &t0, &t1))
           return false;
```

```
        <<Check quadric shape t0 and t1 for nearest interse
ction>>
           if (t0.UpperBound() > ray.tMax || t1.LowerBound
() <= 0)
               return false;
           EFloat tShapeHit = t0;
           if (tShapeHit.LowerBound() <= 0) {
               tShapeHit = t1;
               if (tShapeHit.UpperBound() > ray.tMax)
                   return false;
           }

    //这里是为了处理浮点误差，具体原理后面会讲
    <<Compute cylinder hit point and >>
       pHit = ray((Float)tShapeHit);
       <<Refine cylinder intersection point>>
          Float hitRad = std::sqrt(pHit.x * pHit.x + pHit.
y * pHit.y);
          pHit.x *= radius / hitRad;
          pHit.y *= radius / hitRad;

       phi = std::atan2(pHit.y, pHit.x);
       if (phi < 0) phi += 2 * Pi;
    //与圆球一样，测试交点
    <<Test cylinder intersection against clipping paramete
rs>>
       if (pHit.z < zMin || pHit.z > zMax || phi > phiMax)
 {
           if (tShapeHit == t1) return false;
           tShapeHit = t1;
           if (t1.UpperBound() > ray.tMax) return false;
           <<Compute cylinder hit point and >>
           if (pHit.z < zMin || pHit.z > zMax || phi > phi
Max)
               return false;
       }
        //计算uv
        <<Find parametric representation of cylinder hit>>

       Float u = phi / phiMax;
       Float v = (pHit.z - zMin) / (zMax - zMin);
```

```
        //求dp/du与dp/dv
        <<Compute cylinder  and >>
           Vector3f dpdu(-phiMax * pHit.y, phiMax * pHit.x,
 0);
           Vector3f dpdv(0, 0, zMax - zMin);

        <<Compute cylinder  and >>
           Vector3f d2Pduu = -phiMax * phiMax * Vector3f(pH
it.x, pHit.y, 0);
           Vector3f d2Pduv(0, 0, 0), d2Pdvv(0, 0, 0);
           <<Compute coefficients for fundamental forms>>
              Float E = Dot(dpdu, dpdu);
              Float F = Dot(dpdu, dpdv);
              Float G = Dot(dpdv, dpdv);
              Vector3f N = Normalize(Cross(dpdu, dpdv));
              Float e = Dot(N, d2Pduu);
              Float f = Dot(N, d2Pduv);
              Float g = Dot(N, d2Pdvv);

           <<Compute  and  from fundamental form coefficien
ts>>
              Float invEGF2 = 1 / (E * G - F * F);
              Normal3f dndu = Normal3f((f * F - e * G) * in
vEGF2 * dpdu +
                                       (e * F - f * E) * in
vEGF2 * dpdv);
              Normal3f dndv = Normal3f((g * F - f * G) * in
vEGF2 * dpdu +
                                       (f * F - g * E) * in
vEGF2 * dpdv);

    <<Compute error bounds for cylinder intersection>>
        Vector3f pError = gamma(3) * Abs(Vector3f(pHit.x, p
Hit.y, 0));

    <<Initialize SurfaceInteraction from parametric inform
ation>>
        *isect = (*ObjectToWorld)(
            SurfaceInteraction(pHit, pError, Point2f(u, v),
 -ray.d, dpdu, dpdv,
```

```
                                    dndu, dndv, ray.time,
this));

    <<Update tHit for quadric intersection>>
       *tHit = (Float)tShapeHit;

    return true;
}
```

求偏导数，直接给出结论

$$\frac{\partial p}{\partial u} = (-\phi_{max} y, \phi_{max} x, 0)$$

$$\frac{\partial p}{\partial v} = (0, 0, z_{max} - z_{min})$$

应用Weingarten equation(魏因加藤方程)，计算对于法线的uv方向的偏导，首先得到

$$\frac{\partial^2 p}{\partial u^2} = -\phi_{max}^2 (x, y, 0)$$

$$\frac{\partial^2 p}{\partial u \partial v} = (0, 0, 0)$$

$$\frac{\partial^2 p}{\partial v^2} = (0, 0, 0)$$

3.3.3 Surface Area

圆柱体就是一个卷起来的矩形，它的高为$z_{max} - z_{min}$，宽为$r\phi_{max}$

```
<<Cylinder Method Definitions>>+=
Float Cylinder::Area() const {
    return (zMax - zMin) * radius * phiMax;
}
```

3.4 Disks

圆盘是一个没有厚度的圆，并不涉及二次方程求解

```
<<Disk Declarations>>=
class Disk : public Shape {
public:
    <<Disk Public Methods>>
       Disk(const Transform *ObjectToWorld, const Transfor
m *WorldToObject,
            bool reverseOrientation, Float height, Float r
adius,
            Float innerRadius, Float phiMax)
           : Shape(ObjectToWorld, WorldToObject, reverseOr
ientation),
             height(height), radius(radius), innerRadius(i
nnerRadius),
             phiMax(Radians(Clamp(phiMax, 0, 360))) { }
       Bounds3f ObjectBound() const;
       bool Intersect(const Ray &ray, Float *tHit, Surface
Interaction *isect,
           bool testAlphaTexture) const;
       bool IntersectP(const Ray &ray, bool testAlphaTextu
re) const;
       Float Area() const;
       Interaction Sample(const Point2f &u) const;

private:
    <<Disk Private Data>>
       const Float height, radius, innerRadius, phiMax;

};
```

参数化方程为

$$\phi = u\phi_{max}$$

$$x = ((1-v)r + vr_i)\cos\phi$$

$$y = ((1-v)r + vr_i)\sin\phi$$

$$z = h$$

其中，r_i为内半径，这个时候这个disk中间是空心的。

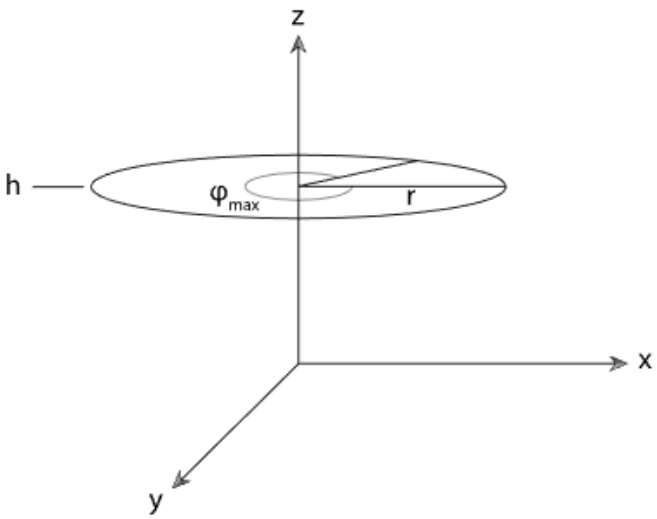


Figure 3.8: Basic Setting for the Disk Shape. The disk has radius r and is located at height h along the z axis. A partial disk may be swept by specifying a maximum ϕ value and an inner radius r_i.

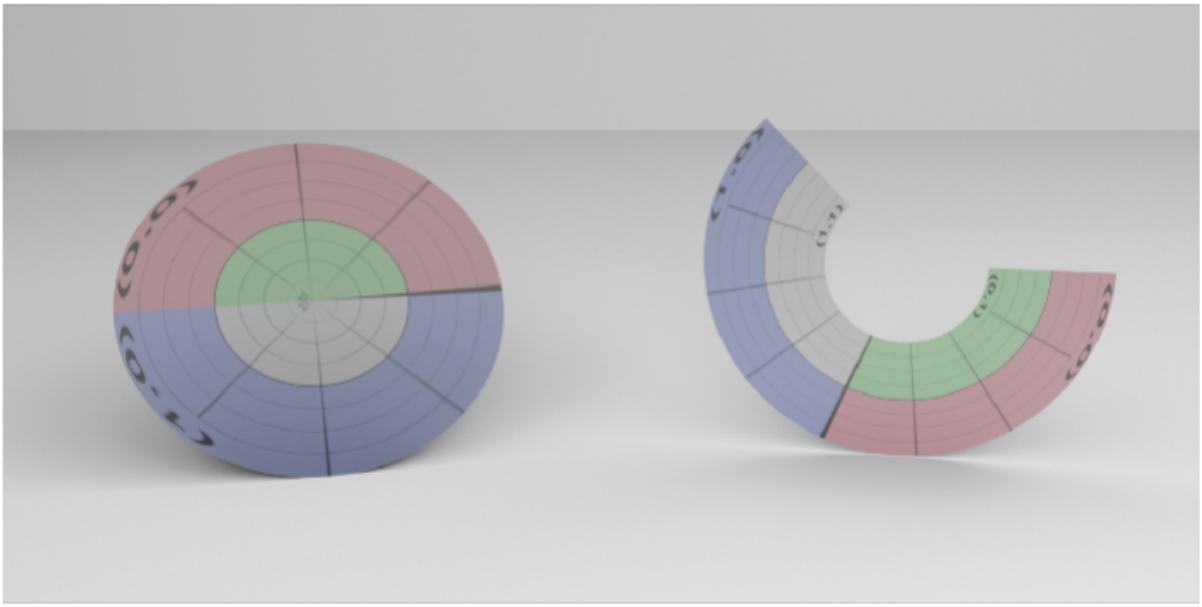


Figure 3.9: Two Disks. A complete disk is on the left, and a partial disk is on the right.

3.4.1 Bounding

跟圆柱体一模一样

```
<<Disk Method Definitions>>=
Bounds3f Disk::ObjectBound() const {
    return Bounds3f(Point3f(-radius, -radius, height),
                    Point3f( radius,  radius, height));
}
```

3.4.2 Intersection Tests

代码如下

```
<<Disk Method Definitions>>+=
bool Disk::Intersect(const Ray &r, Float *tHit,
        SurfaceInteraction *isect, bool testAlphaTexture)
const {
    <<Transform Ray to object space>>
       Vector3f oErr, dErr;
       Ray ray = (*WorldToObject)(r, &oErr, &dErr);

    <<Compute plane intersection for disk>>
        //检测射线是否与圆盘平行
       <<Reject disk intersections for rays parallel to th
e disk's plane>>
          if (ray.d.z == 0)
              return false;
         //首先检测能不能射到圆盘所在平面
       Float tShapeHit = (height - ray.o.z) / ray.d.z;
       if (tShapeHit <= 0 || tShapeHit >= ray.tMax)
           return false;
    //检测与平面交点，是否在两个半径之间的圆环内
    <<See if hit point is inside disk radii and >>
       Point3f pHit = ray(tShapeHit);
       Float dist2 = pHit.x * pHit.x + pHit.y * pHit.y;
       if (dist2 > radius * radius || dist2 < innerRadius
 * innerRadius)
           return false;
       //phi取值范围测试
       <<Test disk  value against >>
          Float phi = std::atan2(pHit.y, pHit.x);
          if (phi < 0) phi += 2 * Pi;
          if (phi > phiMax)
              return false;
    //计算dp/du和dp/dv
    <<Find parametric representation of disk hit>>
       Float u = phi / phiMax;
       Float rHit = std::sqrt(dist2);
       Float oneMinusV = ((rHit - innerRadius) /
                          (radius - innerRadius));
```

```
        Float v = 1 - oneMinusV;
        Vector3f dpdu(-phiMax * pHit.y, phiMax * pHit.x,
0);
        Vector3f dpdv = Vector3f(pHit.x, pHit.y, 0.) * (inn
erRadius - radius) /
                            rHit;
        Normal3f dndu(0, 0, 0), dndv(0, 0, 0);

    <<Refine disk intersection point>>
        pHit.z = height;

    <<Compute error bounds for disk intersection>>
        Vector3f pError(0, 0, 0);

    <<Initialize SurfaceInteraction from parametric inform
ation>>
        *isect = (*ObjectToWorld)(
            SurfaceInteraction(pHit, pError, Point2f(u, v),
 -ray.d, dpdu, dpdv,
                               dndu, dndv, ray.time,
this));

    <<Update tHit for quadric intersection>>
        *tHit = (Float)tShapeHit;

    return true;
}
```

首先要计算的就是射线在圆盘高度的那个位置，这步是为了检查t是否在$[0, t_{max}]$范围内。

$$h = o_z + td_z$$

得

$$t = \frac{h - o_z}{d_z}$$

3.4.3 Surface Area

就是圆环面积

$$A = \frac{\phi_{max}}{2}(r^2 - r_i^2)$$

3.5 Other Quadrics

pbrt支持3种其他的二次曲面：cones, paraboloids, hyperboloids

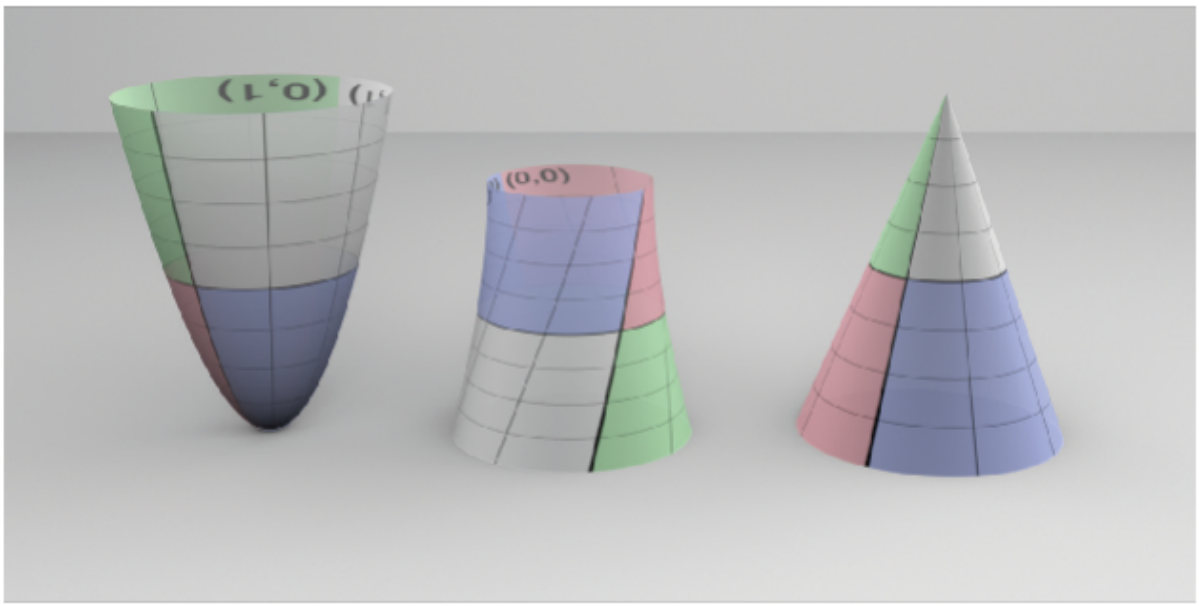


Figure 3.10: The Remaining Quadric Shapes. From left to right: the paraboloid, the hyperboloid, and the cone.

3.5.1 Cones

隐式方程围绕z轴，以及有半径r和高度h

$$(\frac{hx}{r})^2 + (\frac{hy}{r})^2 - (z-h)^2 = 0$$

参数化方程为

$$\phi = u\phi_{max}$$

$$x = r(1-v)\cos\phi$$

$$y = r(1-v)\sin\phi$$

$$z = vh$$

偏导数为

$$\frac{\partial p}{\partial u} = (-\phi_{max} y, \phi_{max} x, 0)$$

$$\frac{\partial p}{\partial v} = (-\frac{x}{1-v}, -\frac{y}{1-v}, h)$$

二阶导为

$$\frac{\partial^2 p}{\partial u^2} = -\phi_{max}^2 (x, y, 0)$$

$$\frac{\partial^2 p}{\partial u \partial v} = \frac{\phi_{max}}{1-v} (y, -x, 0)$$

$$\frac{\partial^2 p}{\partial v^2} = (0, 0, 0)$$

3.5.2 Paraboloids

隐式方程围绕z轴，半径为r，高度为h

$$\frac{hx^2}{r^2} + \frac{hy^2}{r^2} - z = 0$$

参数化形式为

$$\phi = u\phi_{max}$$

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

$$r = r_{max}\sqrt{\frac{z}{z_{max}}}$$

$$x = r\cos\phi$$

$$y = r\sin\phi$$

偏导数为

$$\frac{\partial p}{\partial u} = (-\phi_{max} y, \phi_{max} x, 0)$$

$$\frac{\partial p}{\partial v} = (z_{max} - z_{min})(\frac{x}{2z}, \frac{y}{2z}, 1)$$

$$\frac{\partial^2 p}{\partial u^2} = -\phi_{max}^2 (x, y, 0)$$

$$\frac{\partial^2 p}{\partial u \partial v} = \phi_{max}(z_{max} - z_{min})(-\frac{y}{2z}, \frac{x}{2z}, 0)$$

$$\frac{\partial^2 p}{\partial v^2} = -(z_{max} - z_{min})^2(\frac{x}{4z^2}, \frac{y}{4z^2}, 0)$$

3.5.3 Hyperboloids

隐式方程表达式

$$x^2 + y^2 - z^2 = -1$$

参数化形式

$$\phi = u\phi_{max}$$

$$x_r = (1 - v)x_1 + vx_2$$

$$y_r = (1 - v)y_1 + vy_2$$

$$x = x_r \cos\phi - y_r \sin\phi$$

$$y = x_r \sin\phi + y_r \cos\phi$$

$$z = (1 - v)z_1 + vz_2$$

偏导数

$$\frac{\partial p}{\partial u} = -\phi_{max}^2(x, y, 0)$$

$$\frac{\partial p}{\partial v} = ((x_2 - x_1)\cos\phi - (y_2 - y_1)\sin\phi, (x_2 - x_1)\sin\phi + (y_2 - y_1)\cos\phi, z_2 - z_1)$$

$$\frac{\partial^2 p}{\partial u^2} = -\phi_{max}^2(x, y, 0)$$

$$\frac{\partial^2 p}{\partial u \partial v} = \phi_{max}(-\frac{\partial p_y}{\partial v}, \frac{\partial p_x}{\partial v}, 0)$$

$$\frac{\partial^2 p}{\partial v^2} = (0, 0, 0)$$

3.6 Triangle Meshes

三角形面片是最适合表现复杂物体的数据形式，一个复杂场景通常有数百万个三角形。一个三角形由3个顶点组成，更有效的方式是使用顶点列表和索引列表。

考虑 Euler-Poincaré formula。

$$V - E + F = 2(1 - g)$$

其中，V为顶点数，E为边数，F为三角形数量，$g \in N$是mesh的genus。genus是一个比较小的数字，可以被解释为mesh的handles，类比于茶杯的手柄。对于一个三角形mesh，顶点和面有如下关系

$$E = \frac{3}{2} F$$

两个邻接三角形有一条共享边，每个三角形有3条边，因此有上面的关系。
把这两个公式联立，在$g = 0$的条件下，得到

$$F \approx 2V$$

大体来说，三角形的数量大概是顶点数量的2倍。因为每个面都会引用3个顶点，每个顶点被平均引用6次，因此当采用共享顶点列表的方式时，总统缩减后的存储空间大概是12 bytes的偏移量，再加上存储一个顶点的一半的存储空间6bytes，在这里假设顶点使用4bytes的float型变量，因此一共是18bytes，而不是储存全部顶点数据的36bytes。

```
<<Triangle Declarations>>=
struct TriangleMesh {
    <<TriangleMesh Public Methods>>
       TriangleMesh(const Transform &ObjectToWorld, int nT
riangles, const int *vertexIndices,
           int nVertices, const Point3f *P, const Vector3f
 *S, const Normal3f *N,
           const Point2f *uv, const std::shared_ptr<Textur
e<Float>> &alphaMask);

    <<TriangleMesh Data>>
       const int nTriangles, nVertices;
       std::vector<int> vertexIndices;
       ///顶点位置
       std::unique_ptr<Point3f[]> p;
       ///法线
       std::unique_ptr<Normal3f[]> n;
       ///切线
       std::unique_ptr<Vector3f[]> s;
       //纹理坐标
       std::unique_ptr<Point2f[]> uv;
       std::shared_ptr<Texture<Float>> alphaMask;
```

```
};
```

三角形数据在pbrt中有双重身份，一方面用来描述场景，另一方面，其他形状经常会几何体镶嵌，成为三角形mesh。
基于第二个身份，我们需要将三角形参数化。如果一个三角形可以通过解析一个参数化二次曲面的3个特定坐标(u,v)，获得其位置，这三组uv坐标可以通过插值的方式，获得射线击中位置的uv坐标。显式指定uv坐标也可以用与纹理映射，当一个外部程序创建了一个三角形mesh，希望设置uv坐标将那个mesh赋予纹理。

```
<<Triangle Method Definitions>>=
TriangleMesh::TriangleMesh(const Transform &ObjectToWorld,
        int nTriangles, const int *vertexIndices, int nVer
tices,
        const Point3f *P, const Vector3f *S, const Normal3
f *N,
        const Point2f *UV,
        const std::shared_ptr<Texture<Float>> &alphaMask)
    : nTriangles(nTriangles), nVertices(nVertices),
      vertexIndices(vertexIndices, vertexIndices + 3 * nTr
iangles),
      alphaMask(alphaMask) {
    <<Transform mesh vertices to world space>>
       p.reset(new Point3f[nVertices]);
       //这里把每个顶点转到了世界空间，这样射线检测的时候能省掉一步
转换。
       for (int i = 0; i < nVertices; ++i)
           p[i] = ObjectToWorld(P[i]);

    <<Copy UV, N, and S vertex data, if present>>
       if (UV) {
           uv.reset(new Point2f[nVertices]);
           memcpy(uv.get(), UV, nVertices * sizeof(Point2
f));
       }
       if (N) {
           n.reset(new Normal3f[nVertices]);
           for (int i = 0; i < nVertices; ++i)
               n[i] = ObjectToWorld(N[i]);
       }
       if (S) {
           s.reset(new Vector3f[nVertices]);
```

```
            for (int i = 0; i < nVertices; ++i)
                s[i] = ObjectToWorld(S[i]);
        }

}
```

3.6.1 Triangle

```
<<Triangle Declarations>>+=
class Triangle : public Shape {
public:
    <<Triangle Public Methods>>
       Triangle(const Transform *ObjectToWorld, const Tran
sform *WorldToObject,
                bool reverseOrientation,
                const std::shared_ptr<TriangleMesh> &mesh,
 int triNumber)
           : Shape(ObjectToWorld, WorldToObject, reverseOr
ientation),
             mesh(mesh) {
           v = &mesh->vertexIndices[3 * triNumber];
       }
       Bounds3f ObjectBound() const;
       Bounds3f WorldBound() const;
       bool Intersect(const Ray &ray, Float *tHit, Surface
Interaction *isect,
           bool testAlphaTexture) const;
       bool IntersectP(const Ray &ray, bool testAlphaTextu
re) const;
       Float Area() const;
       Interaction Sample(const Point2f &u) const;

private:
    <<Triangle Private Methods>>
       void GetUVs(Point2f uv[3]) const {
           if (mesh->uv) {
               uv[0] = mesh->uv[v[0]];
               uv[1] = mesh->uv[v[1]];
               uv[2] = mesh->uv[v[2]];
           } else {
```

```
                uv[0] = Point2f(0, 0);
                uv[1] = Point2f(1, 0);
                uv[2] = Point2f(1, 1);
            }
        }

    <<Triangle Private Data>>
        ///parent
        std::shared_ptr<TriangleMesh> mesh;
        ///索引缓冲区的第一个索引的指针
        const int *v;

};
```

另外还有创建三角形mesh的函数，用来将其他形状转化为三角形mesh。

```
<<Triangle Method Definitions>>+=
std::vector<std::shared_ptr<Shape>> CreateTriangleMesh(
        const Transform *ObjectToWorld, const Transform *W
orldToObject,
        bool reverseOrientation, int nTriangles,
        const int *vertexIndices, int nVertices, const Poi
nt3f *p,
        const Vector3f *s, const Normal3f *n, const Point2
f *uv,
        const std::shared_ptr<Texture<Float>> &alphaMask)
 {
    std::shared_ptr<TriangleMesh> mesh = std::make_shared<
TriangleMesh>(
        *ObjectToWorld, nTriangles, vertexIndices, nVertic
es, p, s, n, uv,
        alphaMask);
    std::vector<std::shared_ptr<Shape>> tris;
    for (int i = 0; i < nTriangles; ++i)
        tris.push_back(std::make_shared<Triangle>(ObjectTo
World,
            WorldToObject, reverseOrientation, mesh, i));
    return tris;
}
///boudning box
<<Triangle Method Definitions>>+=
```

```
Bounds3f Triangle::ObjectBound() const {
    <<Get triangle vertices in p0, p1, and p2>>
    return Union(Bounds3f((*WorldToObject)(p0), (*WorldToO
bject)(p1)),
                 (*WorldToObject)(p2));
}
<<Triangle Method Definitions>>+=
Bounds3f Triangle::WorldBound() const {
    <<Get triangle vertices in p0, p1, and p2>>
    return Union(Bounds3f(p0, p1), p2);
}
```

3.6.2 Triangle Intersection

射线与三角形相交函数

```
<<Triangle Method Definitions>>+=
bool Triangle::Intersect(const Ray &ray, Float *tHit,
        SurfaceInteraction *isect, bool testAlphaTexture)
const {
    <<Get triangle vertices in p0, p1, and p2>>
        ///得到三个顶点位置
       const Point3f &p0 = mesh->p[v[0]];
       const Point3f &p1 = mesh->p[v[1]];
       const Point3f &p2 = mesh->p[v[2]];

    <<Perform ray-triangle intersection test>>
       <<Transform triangle vertices to ray coordinate spa
ce>>
          <<Translate vertices based on ray origin>>
              ///转换到射线原点坐标系下
             Point3f p0t = p0 - Vector3f(ray.o);
             Point3f p1t = p1 - Vector3f(ray.o);
             Point3f p2t = p2 - Vector3f(ray.o);

          <<Permute components of triangle vertices and ra
y direction>>
             int kz = MaxDimension(Abs(ray.d));
             int kx = kz + 1; if (kx == 3) kx = 0;
             int ky = kx + 1; if (ky == 3) ky = 0;
```

```
            Vector3f d = Permute(ray.d, kx, ky, kz);
            p0t = Permute(p0t, kx, ky, kz);
            p1t = Permute(p1t, kx, ky, kz);
            p2t = Permute(p2t, kx, ky, kz);

         <<Apply shear transformation to translated verte
x positions>>
            Float Sx = -d.x / d.z;
            Float Sy = -d.y / d.z;
            Float Sz = 1.f / d.z;
            p0t.x += Sx * p0t.z;
            p0t.y += Sy * p0t.z;
            p1t.x += Sx * p1t.z;
            p1t.y += Sy * p1t.z;
            p2t.x += Sx * p2t.z;
            p2t.y += Sy * p2t.z;

       <<Compute edge function coefficients e0, e1, and e2
>>
          Float e0 = p1t.x * p2t.y - p1t.y * p2t.x;
          Float e1 = p2t.x * p0t.y - p2t.y * p0t.x;
          Float e2 = p0t.x * p1t.y - p0t.y * p1t.x;

       <<Fall back to double-precision test at triangle ed
ges>>
          if (sizeof(Float) == sizeof(float) &&
              (e0 == 0.0f || e1 == 0.0f || e2 == 0.0f)) {
              double p2txp1ty = (double)p2t.x * (double)p1
t.y;
              double p2typ1tx = (double)p2t.y * (double)p1
t.x;
              e0 = (float)(p2typ1tx - p2txp1ty);
              double p0txp2ty = (double)p0t.x * (double)p2
t.y;
              double p0typ2tx = (double)p0t.y * (double)p2
t.x;
              e1 = (float)(p0typ2tx - p0txp2ty);
              double p1txp0ty = (double)p1t.x * (double)p0
t.y;
              double p1typ0tx = (double)p1t.y * (double)p0
t.x;
```

```
            e2 = (float)(p1typ0tx - p1txp0ty);
        }

        <<Perform triangle edge and determinant tests>>
        if ((e0 < 0 || e1 < 0 || e2 < 0) && (e0 > 0 || e
1 > 0 || e2 > 0))
            return false;
        Float det = e0 + e1 + e2;
        if (det == 0)
            return false;

        <<Compute scaled hit distance to triangle and test
 against ray  range>>
        p0t.z *= Sz;
        p1t.z *= Sz;
        p2t.z *= Sz;
        Float tScaled = e0 * p0t.z + e1 * p1t.z + e2 * p
2t.z;
        if (det < 0 && (tScaled >= 0 || tScaled < ray.tM
ax * det))
            return false;
        else if (det > 0 && (tScaled <= 0 || tScaled > r
ay.tMax * det))
            return false;

        <<Compute barycentric coordinates and  value for tr
iangle intersection>>
        Float invDet = 1 / det;
        Float b0 = e0 * invDet;
        Float b1 = e1 * invDet;
        Float b2 = e2 * invDet;
        Float t = tScaled * invDet;

        <<Ensure that computed triangle  is conservatively
 greater than zero>>
        <<Compute  term for triangle  error bounds>>
            Float maxZt = MaxComponent(Abs(Vector3f(p0t.
z, p1t.z, p2t.z)));
            Float deltaZ = gamma(3) * maxZt;
```

```
            <<Compute  and  terms for triangle  error bounds
>>
               Float maxXt = MaxComponent(Abs(Vector3f(p0t.
x, p1t.x, p2t.x)));
               Float maxYt = MaxComponent(Abs(Vector3f(p0t.
y, p1t.y, p2t.y)));
               Float deltaX = gamma(5) * (maxXt + maxZt);
               Float deltaY = gamma(5) * (maxYt + maxZt);

            <<Compute  term for triangle  error bounds>>
               Float deltaE = 2 * (gamma(2) * maxXt * maxYt
 + deltaY * maxXt +
                                   deltaX * maxYt);

            <<Compute  term for triangle  error bounds and c
heck t>>
               Float maxE = MaxComponent(Abs(Vector3f(e0, e
1, e2)));
               Float deltaT = 3 * (gamma(3) * maxE * maxZt +
 deltaE * maxZt +
                                   deltaZ * maxE) *
std::abs(invDet);
               if (t <= deltaT)
                   return false;


    <<Compute triangle partial derivatives>>
       Vector3f dpdu, dpdv;
       Point2f uv[3];
       GetUVs(uv);
       <<Compute deltas for triangle partial derivatives>>

          Vector2f duv02 = uv[0] - uv[2], duv12 = uv[1] -
 uv[2];
          Vector3f dp02 = p0 - p2, dp12 = p1 - p2;

       Float determinant = duv02[0] * duv12[1] - duv02[1]
 * duv12[0];
       if (determinant == 0) {
           <<Handle zero determinant for triangle partial
 derivative matrix>>
```

```
                CoordinateSystem(Normalize(Cross(p2 - p0, p1
 - p0)), &dpdu, &dpdv);

        } else {
            Float invdet = 1 / determinant;
            dpdu = ( duv12[1] * dp02 - duv02[1] * dp12) * i
nvdet;
            dpdv = (-duv12[0] * dp02 + duv02[0] * dp12) * i
nvdet;
        }

    <<Compute error bounds for triangle intersection>>
        Float xAbsSum = (std::abs(b0 * p0.x) + std::abs(b1
 * p1.x) +
                         std::abs(b2 * p2.x));
        Float yAbsSum = (std::abs(b0 * p0.y) + std::abs(b1
 * p1.y) +
                         std::abs(b2 * p2.y));
        Float zAbsSum = (std::abs(b0 * p0.z) + std::abs(b1
 * p1.z) +
                         std::abs(b2 * p2.z));
        Vector3f pError = gamma(7) * Vector3f(xAbsSum, yAbs
Sum, zAbsSum);

    <<Interpolate  parametric coordinates and hit point>>
        Point3f pHit = b0 * p0 + b1 * p1 + b2 * p2;
        Point2f uvHit = b0 * uv[0] + b1 * uv[1] + b2 *
uv[2];

    <<Test intersection against alpha texture, if present>
>
        if (testAlphaTexture && mesh->alphaMask) {
            SurfaceInteraction isectLocal(pHit,
Vector3f(0,0,0), uvHit,
                Vector3f(0,0,0), dpdu, dpdv,
Normal3f(0,0,0), Normal3f(0,0,0),
                ray.time, this);
            if (mesh->alphaMask->Evaluate(isectLocal) == 0)
                return false;
        }
```

```
    <<Fill in SurfaceInteraction from triangle hit>>
       *isect = SurfaceInteraction(pHit, pError, uvHit, -r
ay.d, dpdu, dpdv,
           Normal3f(0, 0, 0), Normal3f(0, 0, 0), ray.time,
 this);
       <<Override surface normal in isect for triangle>>
          isect->n = isect->shading.n = Normal3f(Normalize
(Cross(dp02, dp12)));

       if (mesh->n || mesh->s) {
           <<Initialize Triangle shading geometry>>
              <<Compute shading normal ns for triangle>>
                 Normal3f ns;
                 if (mesh->n) ns = Normalize(b0 * mesh->n
[v[0]] +
                                             b1 * mesh->n
[v[1]] +
                                             b2 * mesh->n
[v[2]]);
                 else
                     ns = isect->n;

              <<Compute shading tangent ss for triangle>>
                 Vector3f ss;
                 if (mesh->s) ss = Normalize(b0 * mesh->s
[v[0]] +
                                             b1 * mesh->s
[v[1]] +
                                             b2 * mesh->s
[v[2]]);
                 else
                     ss = Normalize(isect->dpdu);

              <<Compute shading bitangent ts for triangle
 and adjust ss>>
                 Vector3f ts = Cross(ns, ss);
                 if (ts.LengthSquared() > 0.f) {
                     ts = Normalize(ts);
                     ss = Cross(ts, ns);
                 }
                 else
```

```
                    CoordinateSystem((Vector3f)ns, &ss, &
ts);

                <<Compute  and  for triangle shading geometr
y>>
                   Normal3f dndu, dndv;
                   if (mesh->n) {
                       <<Compute deltas for triangle partial
 derivatives of normal>>
                          Vector2f duv02 = uv[0] - uv[2];
                          Vector2f duv12 = uv[1] - uv[2];
                          Normal3f dn1 = mesh->n[v[0]] - mes
h->n[v[2]];
                          Normal3f dn2 = mesh->n[v[1]] - mes
h->n[v[2]];

                       Float determinant = duv02[0] *
duv12[1] - duv02[1] * duv12[0];
                       if (determinant == 0)
                           dndu = dndv = Normal3f(0, 0, 0);
                       else {
                           Float invDet = 1 / determinant;
                           dndu = ( duv12[1] * dn1 -
duv02[1] * dn2) * invDet;
                           dndv = (-duv12[0] * dn1 +
duv02[0] * dn2) * invDet;
                       }
                   }
                   else
                       dndu = dndv = Normal3f(0,0,0);

                isect->SetShadingGeometry(ss, ts, dndu, dnd
v, true);

            }
            <<Ensure correct orientation of the geometric norma
l>>
              if (mesh->n)
                  isect->n = Faceforward(isect->n, isect->shad
ing.n);
```

```
            else if (reverseOrientation ^ transformSwapsHand
edness)
                isect->n = isect->shading.n = -isect->n;

    *tHit = t;
    return true;
}
```

pbrt的射线三角形测试，是基于把三角形变换到射线坐标系，即射线是从(0,0,0)点沿着+z方向。
将顶点从世界空间，转换为ray-triangle相交空间需要3个步骤，一个平移矩阵T，一个置换矩阵P，一个切变矩阵S。
首先平移矩阵是要把顶点以射线起始点作为原点的变换

$$T = \begin{pmatrix} 1 & 0 & 0 & -o_x \\ 0 & 1 & 0 & -o_y \\ 0 & 0 & 1 & -o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

这个矩阵，在代码里就是直接应用到了顶点了

```
<<Translate vertices based on ray origin>>=
Point3f p0t = p0 - Vector3f(ray.o);
Point3f p1t = p1 - Vector3f(ray.o);
Point3f p2t = p2 - Vector3f(ray.o);
```

接下来调换每个顶点和射线的个分量的位置，让ray.d最大值的那个分量作为主分量，放到z的位置上

```
<<Permute components of triangle vertices and ray directio
n>>=
int kz = MaxDimension(Abs(ray.d));
int kx = kz + 1; if (kx == 3) kx = 0;
int ky = kx + 1; if (ky == 3) ky = 0;
Vector3f d = Permute(ray.d, kx, ky, kz);
p0t = Permute(p0t, kx, ky, kz);
p1t = Permute(p1t, kx, ky, kz);
p2t = Permute(p2t, kx, ky, kz);
```

最终，切边矩阵将射线方向对其到+z方向上

$$S = \begin{pmatrix} 1 & 0 & -d_x/d_z & 0 \\ 0 & 1 & -d_y/d_z & 0 \\ 0 & 0 & 1/d_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

对应c++代码

```
<<Apply shear transformation to translated vertex position
s>>=
Float Sx = -d.x / d.z;
Float Sy = -d.y / d.z;
Float Sz = 1.f / d.z;
p0t.x += Sx * p0t.z;
p0t.y += Sy * p0t.z;
p1t.x += Sx * p1t.z;
p1t.y += Sy * p1t.z;
p2t.x += Sx * p2t.z;
p2t.y += Sy * p2t.z;
```

这种坐标变换取决于射线本身，而不是三角形，因此在一个高性能的光线追踪器中，可以把这些信息存储在ray class中，而不是在处理每个三角形时重新计算一遍。经过这一系列的计算，实际上干的是就是让射线从原点沿着+z方向射出，看和三角形相交于哪点，这实际上就是在x,y为(0,0)这条线是否能投影到三角形上。

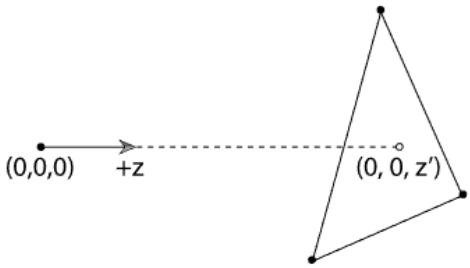


Figure 3.12: In the ray–triangle intersection coordinate system, the ray starts at the origin and goes along the $+z$ axis. The intersection test can be performed by considering only the xy projection of the ray and the triangle vertices, which in turn reduces to determining if the 2D point $(0, 0)$ is within the triangle.

为了搞明白这个算法是如何工作的，首先回顾两个向量的叉乘，它计算的是两个向量组成的平行四边形面积，对于2D向量a和b，面积为

$$a_x b_y - b_x a_y$$

这个面积的一半，就是这两个向量定义的三角形的面积，对于2D三角形，其中p_0，p_1和p_2是这三角形的3个点，其面积公式就变形为：

$$\frac{1}{2}\left((p_{1x}-p_{0x})(p_{2y}-p_{0y})-(p_{2x}-p_{0x})(p_{1y}-p_{0y})\right)$$

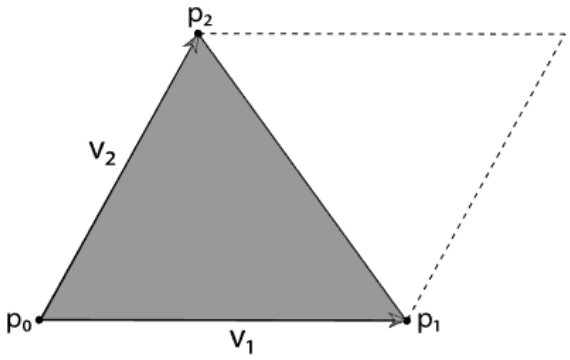


Figure 3.13: The area of a triangle with two edges given by vectors v_1 and v_2 is one-half of the area of the parallelogram shown here. The parallelogram area is given by the length of the cross product of $\mathbf{v}_1$ and $\mathbf{v}_2$.

我们可以使用这个表达式来定义边，已知p_0，p_1和第三个点p，根据叉乘公式，我们可以给出如下表达式：

$$e(p)=(p_{1x}-p_{0x})(p_y-p_{0y})-(p_x-p_{0x})(p_{1y}-p_{0y})$$

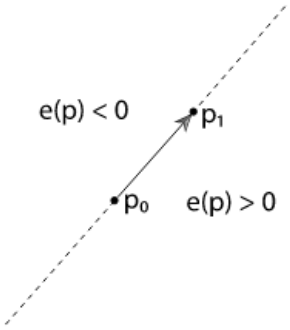


Figure 3.14: The edge function $e(\mathrm{p})$ characterizes points with respect to an oriented line between two points p_0 and p_1. The value of the edge function is positive for points p to the right of the line, zero for points on the line, and negative for points to the left of the line. The ray–triangle intersection algorithm uses an edge function that is twice the signed area of the triangle formed by the three points.

这么做的目的，就是如果点p在p_0和p_1的右侧，那么$e(p)>0$，因此很显然的，如果学这个点的三条边函数是同号的，那么，这个点一定在三角形内。
因为经过了坐标变换，点p的x和y都是0，因此有

$$\begin{aligned}e_0(p)&=(p_{2x}-p_{1x})(p_y-p_{1y})-(p_x-p_{1x})(p_{2y}-p_{1y})\\&=(p_{2x}-p_{1x}))(-p_{1y})-(p_{1x})(p_{2y}-p_{1y})\\&=p_{1x}p_{2y}-p_{2x}p_{1y}\end{aligned}$$

这里就很巧妙地避开了计算p点z坐标的问题，判断交点在不在三角形内，看来并不需要真正计算出来z坐标在哪。记下来，就是计算3条边的edge函数，这里用e_i来表示顶点$p_{(i+1)mod3}$和$p_{(i+2) mod 3}的edge函数。这个边函数本质上就是执行了叉乘。

```
<<Compute edge function coefficients e0, e1, and e2>>=
Float e0 = p1t.x * p2t.y - p1t.y * p2t.x;
Float e1 = p2t.x * p0t.y - p2t.y * p0t.x;
Float e2 = p0t.x * p1t.y - p0t.y * p1t.x;
```

这里有个很特殊的情况，就是e为0，因此有一个对应的双精度浮点数的函数版本来应对这种情况。
有两个条件来判断是否在三角形内，一个是三条边函数是否同号，另一个是三条边函数值相加是否为0，如果为0，说明这个点在边上，那么我们就认为与三角形不相交。

```
<Perform triangle edge and determinant tests>>=
if ((e0 < 0 || e1 < 0 || e2 < 0) && (e0 > 0 || e1 > 0 || e
2 > 0))
    return false;
Float det = e0 + e1 + e2;
if (det == 0)
    return false;
```

因为射线从原点出发，沿着+z轴前进，因此交点的z坐标就是射线计算求交的t值。计算z值，首先应该把三角形顶点的z坐标进行切变变换，之后交点的质心坐标可以通过插值得到

$$b_i = \frac{e_i}{e_0 + e_1 + e_2}$$

从本质上来说，e_i就是点p相对于对边的有向面积，因此上面的公式就是重心公式。
显然b_i求和等于1
插值z值的公式为

$$z = b_0 z_0 + b_1 z_1 + b_2 z_2$$

其中z_i是三个顶点在射线坐标系下的坐标。
如果t超出了有效范围，那么就不必花费多余的cpu周期来计算浮点数除法，于是就先用e_i对z_i进行插值操作，而不是b_i，如果结果上，与$d = e_0 + e_1 + e_2$符号相反，则很明显最终的t值是个负数，是个无效的交点。
相似的，还得确定$t < t_{max}$，这里依然要避免做除法，因此有了如下关系式：
在$e_0 + e_1 + e_2 > 0$的条件下

$$\sum_i e_i z_i < t_{max}(e_0 + e_1 + e_2)$$

否则

$$\sum_i e_i z_i > t_{max}(e_0 + e_1 + e_2)$$

代码：

```
<<Compute scaled hit distance to triangle and test against
 ray  range>>=
p0t.z *= Sz;
p1t.z *= Sz;
p2t.z *= Sz;
Float tScaled = e0 * p0t.z + e1 * p1t.z + e2 * p2t.z;
if (det < 0 && (tScaled >= 0 || tScaled < ray.tMax * det))
    return false;
else if (det > 0 && (tScaled <= 0 || tScaled > ray.tMax *
 det))
    return false;
```

最终计算出t值

```
<<Compute barycentric coordinates and  value for triangle
 intersection>>=
Float invDet = 1 / det;
Float b0 = e0 * invDet;
Float b1 = e1 * invDet;
Float b2 = e2 * invDet;
Float t = tScaled * invDet;
```

接下来为了算切线向量，需要得到$\partial p/\partial u$和$\partial p/\partial v$，但实际上对于一个三角形而言，其内部的所有的点的偏导都是一样的。
一个三角形可由一系列点所表示：

$$p_o + u\frac{\partial p}{\partial u} + v\frac{\partial p}{\partial v}$$

而一个三角形的三个顶点p_i，其中$i = 0, 1, 2$，以及对应的纹理坐标(u_i, v_i)。把它用上面的表达式写出来就应当形如：

$$p_I = p_0 + u_i\frac{\partial p}{\partial u} + v_i\frac{\partial p}{\partial v}$$

这也意味着从(u, v)坐标到三角形顶点有确定的仿射映射关系。为了计算$\partial p/\partial u$与$\partial p/\partial v$，我们从计算$p_0 - p_2$与$p_1 - p_2$开始。已知矩阵等式

$$\begin{pmatrix} u_0 - u_2 & v_0 - v_2 \\ u_1 - u_2 & v_1 - v_2 \end{pmatrix} \begin{pmatrix} \partial p/\partial u \\ \partial p/\partial v \end{pmatrix} = \begin{pmatrix} p_0 - p_2 \\ p_1 - p_2 \end{pmatrix}$$

因此

$$\begin{pmatrix} \partial p/\partial u \\ \partial p/\partial v \end{pmatrix} = \begin{pmatrix} u_0 - u_2 & v_0 - v_2 \\ u_1 - u_2 & v_1 - v_2 \end{pmatrix}^{-1} \begin{pmatrix} p_0 - p_2 \\ p_1 - p_2 \end{pmatrix}$$

最后，(u, v)变化量的逆矩阵的表达式为

$$\frac{1}{(u_0 - u_2)(v_1 - v_2) - (v_0 - v_2)(u_1 - u_2)} \begin{pmatrix} v_1 - v_2 & -(v_0 - v_2) \\ -(u_1 - u_2) & u_0 - u_2 \end{pmatrix}$$

代码

```
<<Compute triangle partial derivatives>>=
Vector3f dpdu, dpdv;
Point2f uv[3];
GetUVs(uv);
<<Compute deltas for triangle partial derivatives>>
   Vector2f duv02 = uv[0] - uv[2], duv12 = uv[1] - uv[2];
   Vector3f dp02 = p0 - p2, dp12 = p1 - p2;

Float determinant = duv02[0] * duv12[1] - duv02[1] * duv12
[0];
if (determinant == 0) {
    <<Handle zero determinant for triangle partial derivat
ive matrix>>
       CoordinateSystem(Normalize(Cross(p2 - p0, p1 - p
0)), &dpdu, &dpdv);

} else {
    Float invdet = 1 / determinant;
    dpdu = ( duv12[1] * dp02 - duv02[1] * dp12) * invdet;
    dpdv = (-duv12[0] * dp02 + duv02[0] * dp12) * invdet;
}
```

另外，还需要处理这个矩阵是奇异矩阵，无法获得逆矩阵的情况。这个只出现于这个三角形的三个参数化顶点是退化的，这时Triangle类只要选择任意坐标系来表示这个法线，dp/du和dp/dv方向什么的不重要了，只要它们是正交的就可以了。

```
<<Handle zero determinant for triangle partial derivative
 matrix>>=
CoordinateSystem(Normalize(Cross(p2 - p0, p1 - p0)), &dpd
u, &dpdv);
```

接下来就是用重心坐标系的参数对uv进行插值来获得相交点的uv坐标

```
<<Interpolate  parametric coordinates and hit point>>=
Point3f pHit = b0 * p0 + b1 * p1 + b2 * p2;
Point2f uvHit = b0 * uv[0] + b1 * uv[1] + b2 * uv[2];
```

接下来的函数GetUVs()获得三个顶点的uv坐标，直接返回mesh的uv坐标就可以了

```
<<Triangle Private Methods>>=
void GetUVs(Point2f uv[3]) const {
    if (mesh->uv) {
        uv[0] = mesh->uv[v[0]];
        uv[1] = mesh->uv[v[1]];
        uv[2] = mesh->uv[v[2]];
    } else {
        uv[0] = Point2f(0, 0);
        uv[1] = Point2f(1, 0);
        uv[2] = Point2f(1, 1);
    }
}
```

在返回交点信息前，还有一个事情要处理，就是alpha掩码纹理

```
<<Test intersection against alpha texture, if present>>=
if (testAlphaTexture && mesh->alphaMask) {
    SurfaceInteraction isectLocal(pHit, Vector3f(0,0,0), u
vHit,
        Vector3f(0,0,0), dpdu, dpdv, Normal3f(0,0,0), Norm
al3f(0,0,0),
        ray.time, this);
    if (mesh->alphaMask->Evaluate(isectLocal) == 0)
        return false;
}
```

掩码纹理认为当前点是镂空的，一样不能算相交。
接下来是计算法线的偏导数，首先一件事就是，因为三角形就是世界空间的，因此不用再进行空间变换了，另外，因为三角形就是个平面，因此法线的偏导实际上就是$(0,0,0)$

```
<<Fill in SurfaceInteraction from triangle hit>>=
*isect = SurfaceInteraction(pHit, pError, uvHit, -ray.d, d
pdu, dpdv,
    Normal3f(0, 0, 0), Normal3f(0, 0, 0), ray.time, this);
<<Override surface normal in isect for triangle>>
   isect->n = isect->shading.n = Normal3f(Normalize(Cross
(dp02, dp12)));

if (mesh->n || mesh->s) {
    <<Initialize Triangle shading geometry>>
        //填充shading geometry，shading geometry的目的就是为
了让几何体显得更光滑，因此这里全部都是插值操作
       //法线插值
       <<Compute shading normal ns for triangle>>
          Normal3f ns;
          if (mesh->n) ns = Normalize(b0 * mesh->n[v[0]] +
                                      b1 * mesh->n[v[1]] +

                                      b2 * mesh->n[v[2]]);
          else
              ns = isect->n;
        //副法线插值
       <<Compute shading tangent ss for triangle>>
          Vector3f ss;
          if (mesh->s) ss = Normalize(b0 * mesh->s[v[0]] +
                                      b1 * mesh->s[v[1]] +

                                      b2 * mesh->s[v[2]]);
          else
              ss = Normalize(isect->dpdu);
        //算出切线
       <<Compute shading bitangent ts for triangle and adj
ust ss>>
          Vector3f ts = Cross(ns, ss);
          if (ts.LengthSquared() > 0.f) {
```

```
                ts = Normalize(ts);
                ss = Cross(ts, ns);
            }
            else
                CoordinateSystem((Vector3f)ns, &ss, &ts);

        <<Compute  and  for triangle shading geometry>>
             //计算法线对于uv的偏导
            Normal3f dndu, dndv;
            if (mesh->n) {
                //如果mesh有法线，则计算
                <<Compute deltas for triangle partial deriva
tives of normal>>
                    Vector2f duv02 = uv[0] - uv[2];
                    Vector2f duv12 = uv[1] - uv[2];
                    Normal3f dn1 = mesh->n[v[0]] - mesh-
>n[v[2]];
                    Normal3f dn2 = mesh->n[v[1]] - mesh-
>n[v[2]];

                Float determinant = duv02[0] * duv12[1] - du
v02[1] * duv12[0];
                if (determinant == 0)
                    dndu = dndv = Normal3f(0, 0, 0);
                else {
                    Float invDet = 1 / determinant;
                    dndu = ( duv12[1] * dn1 - duv02[1] * dn
2) * invDet;
                    dndv = (-duv12[0] * dn1 + duv02[0] * dn
2) * invDet;
                }
            }
            else
                //没有的haunted，就是(0,0,0)
                dndu = dndv = Normal3f(0,0,0);

        isect->SetShadingGeometry(ss, ts, dndu, dndv,
true);

}
//最后是校正交点法线，以及处理反向
```

```
<<Ensure correct orientation of the geometric normal>>
    if (mesh->n)
        isect->n = Faceforward(isect->n, isect->shading.n);
    else if (reverseOrientation ^ transformSwapsHandedness)
        isect->n = isect->shading.n = -isect->n;
```

3.6.3 Shading Geometry

上面的代码已经表明了ShadingGeometry的处理流程，目标是让几何体显得更光滑，因此填充ShadingGeometry是一个插值的过程。

3.6.4 Surface Area

三角形面积，不用说了，大家都知道怎么算，底乘高除2

```
<<Triangle Method Definitions>>+=
Float Triangle::Area() const {
    <<Get triangle vertices in p0, p1, and p2>>
        const Point3f &p0 = mesh->p[v[0]];
        const Point3f &p1 = mesh->p[v[1]];
        const Point3f &p2 = mesh->p[v[2]];

    return 0.5 * Cross(p1 - p0, p2 - p0).Length();
}
```

3.7 Curves*

暂略

3.8 Subdivision Surfaces*

暂略

3.9 Managing Rounding Error*

暂略，太好了

Further Reading

略