Physically Based Rendering读书笔记 part 1

计算机图形学

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

---

1. Introduction

1.1 Literate Programming

介绍了文学编程

1.2 Photorealistic Rendering and the Ray-Tracing Algorithm

几乎所有的照片级渲染都是基于光线追踪算法，光线追踪算法非常简单直接，它基于跟踪穿过场景的光线的光路，与场景中的物体交互、反弹。光线追踪器包含以下组件：

- Cameras 摄像机

- Ray-Object Instersection 射线与物体的交互：不但包含了射线集中了物体的那个位置，也包含了这个位置的信息，包括法线、材质。大多数光线追踪器可以测试射线上的多个物体，返回最近的击中点
- Light Sources 光源： 光线追踪器不仅仅给光源的位置建模，也要给能量分布的方式进行建模。
- Visibility 可见性：如果想要知道光源是否能传输能量到表面上的某一点，就必须知道这个点到光源之间是否有不会被遮挡的路径。解决这个问题并不复杂， 从表面上的点到光源创建一个反向的射线，寻找最近的击中点，并与光源到表面的距离做比较。
- Surface Scattering 表面散射：每个场景中的物体都要提供其表面的信息，包括光线应该如何与表面交互，以及散射光线的本质属性，
- Indirect Light Transport间接光传输：光线可以经过多次反射，以及穿透某些表面到达某个点，因此间接光计算也非常重要。
- Ray Propagation 光线传播：我们需要知道光线在穿越空间时发生了什么，例如雾、烟、大气层等。

1.2.1 Cameras

最简单的摄像机是pin-hole camera针孔摄像机

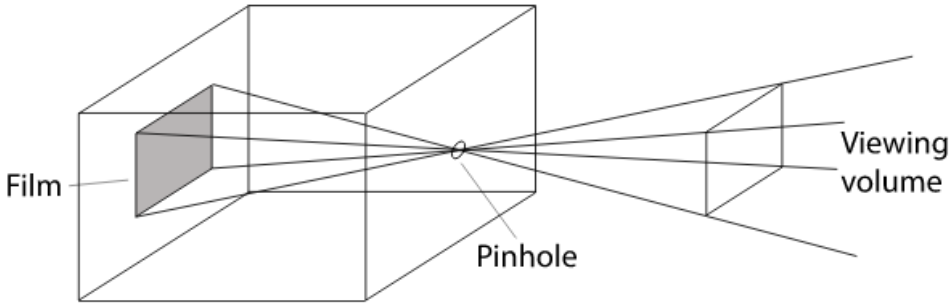


Figure 1.1: A Pinhole Camera.

模拟摄像机最重要的功能，是定义场景的哪些部分能被摄像机看到。图中的右侧三角形范围，就叫做Viewing Volume可视范围。

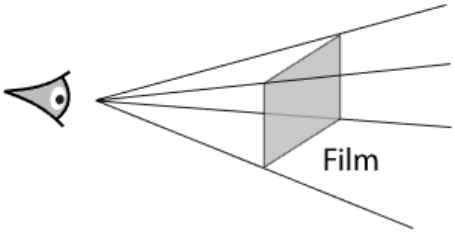


Figure 1.2: When we simulate a pinhole camera, we place the film in front of the hole at the near plane, and the hole is renamed the *eye*.

另一种表示方式就是在近裁减面放一个image，这也定义了Viewing Volume。这里，pinhole其实就是眼睛。

下面的问题就是image的每个像素点应该是什么颜色。作为一个针孔camera，只有从眼睛到那个像素形成的向量方向的光线才对最终图片上的像素有贡献，因此我们关心的是从图片上的一点到眼睛的这个向量发出的光线。
因此，camera simulator最重要的工作，就是找到图片上一点，然后生成一条射线。

1.2.2 Ray-Object Intersections

camera发出一条射线，首先要决定的就是射线击中了谁，在哪个位置击中的。击中点就是可见点。首先我们要参数化射线：

$$r(t) = o + td$$

o是射线起点，d是射线方向，t是参数。
拿到这个射线r之后，我们就可以把它与隐表面函数$F(x, y, z) = 0$求交点。
例如一个球体：

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

代入射线公式：

$$(o_x + td_x)^2 + (o_y + td_y)^2 + (o_z + td_z)^2 - r^2 = 0$$

只有t是未知数。方程没有解则代表不相交，有解的话，就取最小正解作为交点。
只知道个位置并不够，还得知道击中的表面的属性。首先击中点的材质表现必须确定，传给接下来的光线追踪算法的步骤中。其次击中点额外的结合信息也是必要的。例如表面的法线n一直都是需要的。
暴力遍历所有对象进行射线检测能达到目的，但是更加有效的是通过一种加速结构，让算法达到$O(IlogN)$的复杂度，其中I是像素数量。

1.2.3 Light Distribution

我们的终极目标是得到离开表面某点射向camera的光的量，因此我们首先得知道射入那个点的光量。这涉及到光在场景中的geometric and radiometric distribution，几何和辐射度分布。几何分布指的是光的位置，形状信息，即光是从具有一定形状的光源表面发射出去的。
我们研究击中点周围的微表面接收到的光。我们假设点光源的输出功率为Φ，并且往各个方向平等的辐射能量。这意味着一个单位球体每单位面积的输出能量就是$\frac{\Phi}{r\pi}$。
假设有两个圆球，很明显，半径更大的那个每单位面积接收到的能量比半径小的那个要低，因为同等的能量散布给了更大的面积。并且有能量正比于$\frac{1}{r^2}$，其中r为半径。

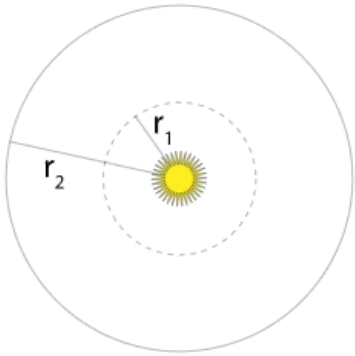


Figure 1.4: Since the point light radiates light equally in all directions, the same total power is deposited on all spheres centered at the light.

如果一个微平面dA相比于光源到当前点这个方向有个角度θ，则dA正比于$\cos\theta$。

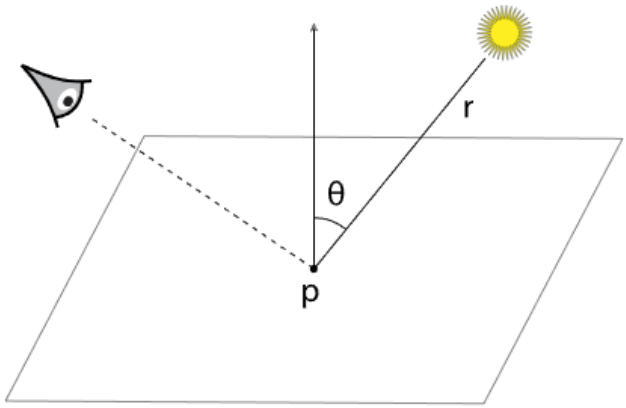


Figure 1.3: Geometric construction for determining the power per area arriving at a point due to a point light source. The distance from the point to the light source is denoted by r.

把这两件事放一起，就得到了dE(differential irradiance)：

$$dE = \frac{\Phi \cos\theta}{4\pi r^2}$$

这个两个基本理论：

- cos 衰减
- 距离平方衰减

是光照公式中的重要组成部分。
场景中的多光源很容易计算，只要遍历每个光源单独处理再求和就可以了。

1.2.4 Visibility

上面所说的光照分布忽略了重要的组成部分：阴影。光线可以照亮表面上的一个点有一个前提条件，就是它没有被遮挡。在光线追踪器当中，做了一个反向操作：从表面某点

向光源发射一条射线，称为shadow ray，这条射线被半路遮挡了，说明光源无法到达这里。

1.2.5 Surface Scattering

现在我们有了给一个点着色的信息：位置和入射光，接下来需要决定的是光线在表面是入射散射的。我们关心的是，光线反射到camera的那部分能量。

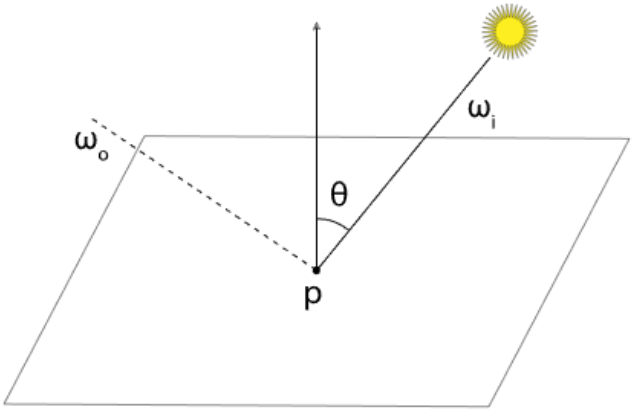


Figure 1.6: The Geometry of Surface Scattering. Incident light arriving along direction ω_i interacts with the surface at point p and is scattered back toward the camera along direction ω_o. The amount of light scattered toward the camera is given by the product of the incident light energy and the BRDF.

场景中的每个物件都有材质，用BRDF来描述，BRDF函数来决定对于来自入射方向$\boldsymbol{\omega_i}$和出射方向$\boldsymbol{\omega_o}$有多少能量会反射出去。对于点p，有BRDF函数$\boldsymbol{f_r(p, \omega_o, \omega_i)}$。

```
for each light:
    if light is not blocked:
        incident_light = light.L(point)
        amount_reflected =
            surface.BRDF(hit_point, camera_vector, light_v
ector)
        L += amount_reflected * incident_light
```

代码中的L代表radiance。
将BRDF推而广之，有BTDF, BSDF(bidirectional scattering distribution function), BSSRDF(bidirectional subsurface scattering reflectance distribution function)。

1.2.6 Indirect Light Transport

Turner Whitted最早的光线跟踪强调递归特性，这是一个关键特性，使得间接高光反射和传输成为可能。光线从camera出发，到达一面镜子，在这里递归地重新开始光线跟踪流程。

通俗地讲，从物体到到摄像机的光线分为两个部分，一个是这个物体的自发光，另一个就是反射过来的光线。对应的公式就是render equation渲染方程：

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_{S^2} f(p,\omega_o,\omega_i) L_i(p,\omega_i) \left|\cos\theta_I\right| d\omega_i$$

直接解这个方程除非是一些非常简单的场景，否则是不可能的。因此我们必须作一些简单的假设，或者使用数值积分。
Whitted的算法简化了这个积分，只考虑来自于光源的方向，以及完美反射和折射的方向，简而言之就是将积分简化为了累加计算。
但是这个算法经过扩展，可以不仅仅适用于完美镜面反射。例如跟踪很多接近镜面反射的方向，然后平均他们的贡献，就得到了一个高光反射的估计。实际上，我们永远可以递归地追踪一个光路，只要它击中了一个物体。例如我们可以随机选择一个入射光线ω_i，然后通过BRDF函数得到它对于最终光线的贡献权重。这是一个简单但有效的方法，可以生成真实感非常高的图片，因为它能捕获所有物体间的内部反射。但随机发射光线带来的问题，是使得结果难以收敛。
这里我们引入光线树的概念，从摄像机开始的光线作为根，每一个它衍生出来的光线，都有一个权重。

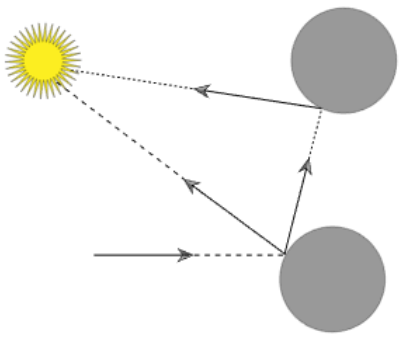


Figure 1.8: Recursive ray tracing associates an entire tree of rays with each image location.

1.2.7 Ray Propagation

目前为止的讨论都是基于光线在真空中传播，比如点光源平均地向四周辐射能量。但一旦遇到烟雾、雾气等参与介质(participating medium)，这个假设就不再对了。这些效果在模拟时非常重要，户外场景会大量收到参与介质的影响。
参与介质通过两个途径影响光线传播：

- 参与介质会使得光线衰减，通过吸收(absorbing)和向其他方向散射(scattering)两种方式，我们通过计算Transmittance T传输率来计算这种效果。传输率告诉我们从击中点到起始点有多少光被散射出去。
- 参与介质也可以添加到光线当中。例如这个介质本身是发光的(火焰)，或者介质向其他方向散射回到摄像机的射线。我们通过计算volume light transport equation(体

积光传输方程)来解决这个问题。

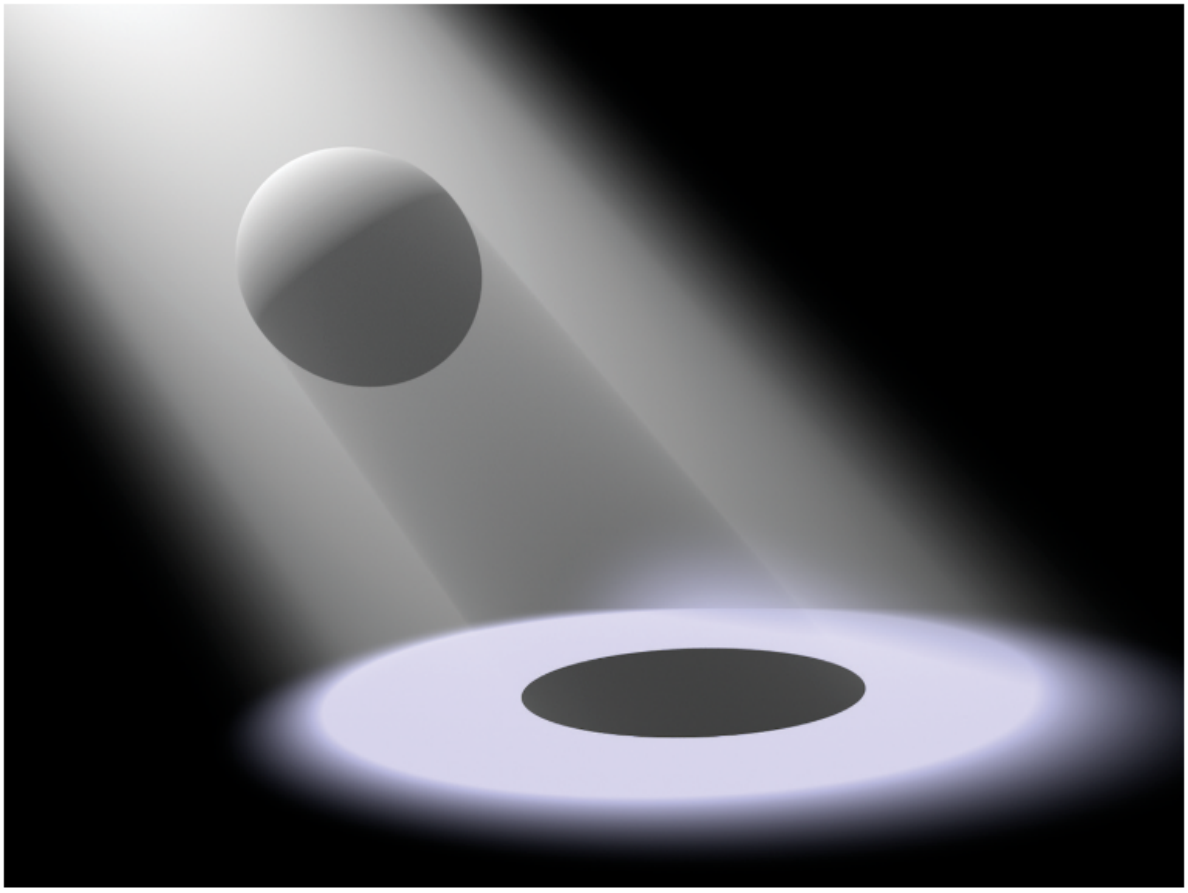


Figure 1.10: A Spotlight Shining on a Sphere through Fog. Notice that the shape of the spotlight's lighting distribution and the sphere's shadow are clearly visible due to the additional scattering in the participating medium.

1.3 pbrt: System Overview

1.3.1 Phases of Execution

一共两个阶段：

- Parsing Phase：读取场景文件，读取数据，最终得到的是Scene class和Integrator class的对象，Scene中的是场景内几何体数据，Integrator则是渲染算法，实现光照方程。
- Main Render Loop：执行入口是Integrator::Render()，渲染逻辑都在这里执行，SampleIntegrator在执行渲染之后将文件存盘。

1.3.2 Scene Representation

初始化函数pbrtInit()
销毁清理函数pbrtCleanup()

```
<<Main program>>=
int main(int argc, char *argv[]) {
    Options options;
    std::vector<std::string> filenames;
    <<Process command-line arguments>>
    pbrtInit(options);
    <<Process scene description>>
    pbrtCleanup();
    return 0;
}

<<Scene Declarations>>=
class Scene {
public:
    <<Scene Public Methods>>
       Scene(std::shared_ptr<Primitive> aggregate,
             const std::vector<std::shared_ptr<Light>> &li
ghts)
           : lights(lights), aggregate(aggregate) {
           <<Scene Constructor Implementation>>
       }
       const Bounds3f &WorldBound() const { return worldBo
und; }
       bool Intersect(const Ray &ray, SurfaceInteraction *
isect) const;
       bool IntersectP(const Ray &ray) const;
       bool IntersectTr(Ray ray, Sampler &sampler, Surface
Interaction *isect,
           Spectrum *transmittance) const;

    <<Scene Public Data>>
       std::vector<std::shared_ptr<Light>> lights;

private:
    <<Scene Private Data>>
       std::shared_ptr<Primitive> aggregate;
       Bounds3f worldBound;

};
<<Scene Public Methods>>=
Scene(std::shared_ptr<Primitive> aggregate,
```

```
        const std::vector<std::shared_ptr<Light>> &lights)
    : lights(lights), aggregate(aggregate) {
    <<Scene Constructor Implementation>>
        worldBound = aggregate->WorldBound();
        for (const auto &light : lights)
            light->Preprocess(*this);

}
```

场景文件中包含Lights列表，Primitive aggregate
Light包含形状和能量分布数据。
Prmitive包含形状和材质数据，scene中包含的是primitive aggregate。aggregate提供了一个加速查询的结构。
Intersect函数会返回相交测试结果以及surface数据，IntersectP则不会返回surface数据，通常用于计算shadow ray。

1.3.3 Integrator Interface And SamplerIntegrator

渲染动作是通过Integrator接口来实现的。SamplerIntegrator是Integrator其中一个实现。

```
<<Integrator Declarations>>=
class Integrator {
public:
    <<Integrator Interface>>
        virtual ~Integrator();
        virtual void Render(const Scene &scene) = 0;

};

<<SamplerIntegrator Declarations>>=
class SamplerIntegrator : public Integrator {
public:
    <<SamplerIntegrator Public Methods>>
        SamplerIntegrator(std::shared_ptr<const Camera> cam
era,
                std::shared_ptr<Sampler> sampler)
            : camera(camera), sampler(sampler) { }
        virtual void Preprocess(const Scene &scene, Sampler
 &sampler) { }
        void Render(const Scene &scene);
```

```
        virtual Spectrum Li(const RayDifferential &ray, con
st Scene &scene,
            Sampler &sampler, MemoryArena &arena, int depth
 = 0) const = 0;
        Spectrum SpecularReflect(const RayDifferential &ra
y,
            const SurfaceInteraction &isect, const Scene &s
cene, Sampler &sampler,
            MemoryArena &arena, int depth) const;
        Spectrum SpecularTransmit(const RayDifferential &ra
y,
            const SurfaceInteraction &isect, const Scene &s
cene, Sampler &sampler,
            MemoryArena &arena, int depth) const;

protected:
    <<SamplerIntegrator Protected Data>>
        std::shared_ptr<const Camera> camera;

private:
    <<SamplerIntegrator Private Data>>
        std::shared_ptr<Sampler> sampler;

};
```

其中sampler是采样器，例如在渲染方程中，用于生成随机采样方向。
Camera是摄像机类，持有Film对象，Film是最终的渲染结果。
Preprocess是执行前处理函数。

1.3.4 The Main Rendering Loop

Scene和Integrator构建完毕后，开始执行渲染，调用Integrator::Render()。

```
<<SamplerIntegrator Method Definitions>>=
void SamplerIntegrator::Render(const Scene &scene) {
    //预处理
    Preprocess(scene, *sampler);
    <<Render image tiles in parallel>>
        <<Compute number of tiles, nTiles, to use for paral
lel rendering>>
            //获得整个输出图片需要处理的部分
```

```
            Bounds2i sampleBounds = camera->film->GetSampleB
ounds();
            Vector2i sampleExtent = sampleBounds.Diagonal();
            const int tileSize = 16;
            //向上取一个tile数量
            Point2i nTiles((sampleExtent.x + tileSize - 1) /
 tileSize,
                           (sampleExtent.y + tileSize - 1) /
 tileSize);
          //多核并行处理
         ParallelFor2D(
             [&](Point2i tile) {
                 <<Render section of image corresponding to
 tile>>
                    <<Allocate MemoryArena for tile>>
                    //申请内存空间
                       MemoryArena arena;

                    <<Get sampler instance for tile>>
                    //拿到tile的序号
                       int seed = tile.y * nTiles.x + tile.
x;
                       std::unique_ptr<Sampler> tileSampler
 = sampler->Clone(seed);
                      //计算当前tile的像素范围
                    <<Compute sample bounds for tile>>
                       int x0 = sampleBounds.pMin.x + tile.x
 * tileSize;
                       int x1 = std::min(x0 + tileSize, samp
leBounds.pMax.x);
                       int y0 = sampleBounds.pMin.y + tile.y
 * tileSize;
                       int y1 = std::min(y0 + tileSize, samp
leBounds.pMax.y);
                       Bounds2i tileBounds(Point2i(x0, y0),
 Point2i(x1, y1));

                    <<Get FilmTile for tile>>
                       std::unique_ptr<FilmTile> filmTile =
                           camera->film->GetFilmTile(tileBou
nds);
```

```
                    <<Loop over pixels in tile to render the
m>>
                        for (Point2i pixel : tileBounds) {
                            //遍历每一个pixel
                            tileSampler->StartPixel(pixel);
                            do {
                                //对于当前pixel，sampler负责采
样，并且不止采样一次
                                <<Initialize CameraSample for
 current sample>>
                                    CameraSample cameraSample
 = tileSampler->GetCameraSample(pixel);
                                //生成射线
                                <<Generate camera ray for cur
rent sample>>
                                    RayDifferential ray;
                                    Float rayWeight = camera->
GenerateRayDifferential(cameraSample, &ray);
                                    ray.ScaleDifferentials(1 /
 std::sqrt(tileSampler->samplesPerPixel));


                                <<Evaluate radiance along cam
era ray>>
                                    Spectrum L(0.f);
                                    //获取这次计算得到的亮度信息
                                    if (rayWeight > 0)
                                        L = Li(ray, scene, *ti
leSampler, arena);
                                    <<Issue warning if unexpec
ted radiance value is returned>>
                                        if (L.HasNaNs()) {
                                            Error("Not-a-number
 radiance value returned "
                                                  "for image sa
mple.  Setting to black.");
                                            L = Spectrum(0.f);
                                        }
                                        else if (L.y() < -1e-5)
 {
```

```
                                        Error("Negative lum
inance value, %f, returned "
                                              "for image sa
mple.  Setting to black.", L.y());
                                        L = Spectrum(0.f);
                                    }
                                    else if (std::isinf(L.y
())) {
                                        Error("Infinite lum
inance value returned "
                                              "for image sa
mple.  Setting to black.");
                                        L = Spectrum(0.f);
                                    }

                                <<Add camera ray's contributi
on to image>>
                                    filmTile->AddSample(camera
Sample.pFilm, L, rayWeight);

                                <<Free MemoryArena memory fro
m computing image sample value>>
                                    arena.Reset();

                            } while (tileSampler->StartNextSa
mple());
                        }

                    <<Merge image tile into Film>>
                        camera->film->MergeFilmTile(std::move
(filmTile));


            }, nTiles);


    <<Save final image after rendering>>
        camera->film->WriteImage();

}
```

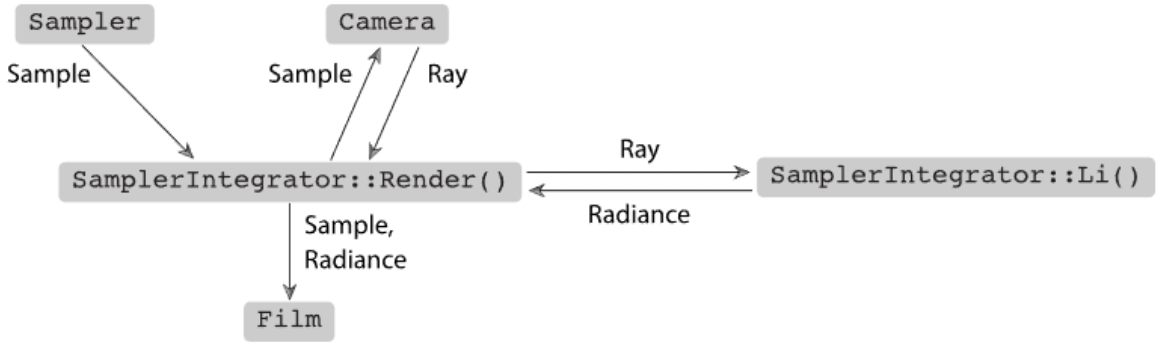
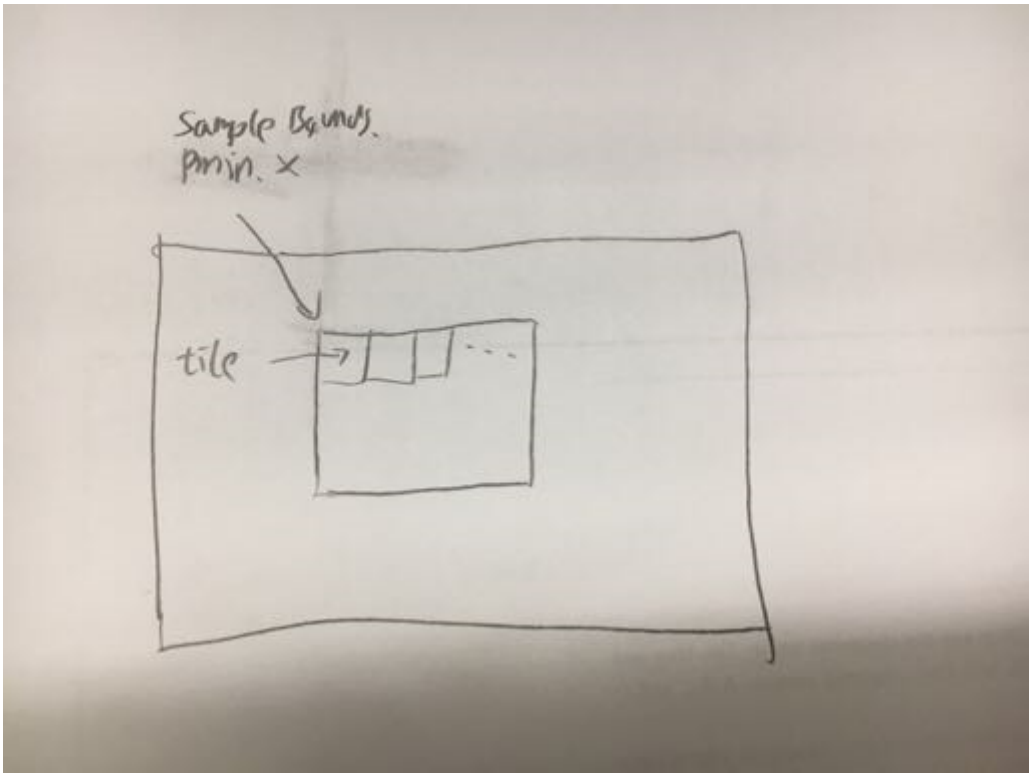


Figure 1.17: Class Relationships for the Main Rendering Loop in the `SamplerIntegrator::Render()` Method in `core/integrator.cpp`. The `Sampler` provides a sequence of sample values, one for each image sample to be taken. The `Camera` turns a sample into a corresponding ray from the film plane, and the `Li()` method implementation computes the radiance along that ray arriving at the film. The sample and its radiance are given to the `Film`, which stores their contribution in an image. This process repeats until the `Sampler` has provided as many samples as are necessary to generate the final image.

充分利用多核处理器的特性，将屏幕进行分块，每个块用一个线程处理。
在pbrt中，对于屏幕，统一使用16x16个块。
计算当前tile块的pixel bounds



1.3.5 An Integrator for Whitted Ray Tracing

下面这个Integrator是基于Whitted Ray Tracing算法。这个积分器准确计算了高光材质的光的反射、传输信息，但是它不管其他的间接光。

```
<<WhittedIntegrator Declarations>>=
class WhittedIntegrator : public SamplerIntegrator {
```

```
public:
    <<WhittedIntegrator Public Methods>>
       WhittedIntegrator(int maxDepth, std::shared_ptr<con
st Camera> camera,
                  std::shared_ptr<Sampler> sampler)
           : SamplerIntegrator(camera, sampler), maxDepth
(maxDepth) { }
       Spectrum Li(const RayDifferential &ray, const Scene
 &scene,
           Sampler &sampler, MemoryArena &arena, int dept
h) const;

private:
    <<WhittedIntegrator Private Data>>
       const int maxDepth;

};
<<WhittedIntegrator Public Methods>>=
WhittedIntegrator(int maxDepth, std::shared_ptr<const Came
ra> camera,
        std::shared_ptr<Sampler> sampler)
    : SamplerIntegrator(camera, sampler), maxDepth(maxDept
h) { }
```

Whitted Integrator的Li函数会递归地沿着反射和折射光线计算亮度。在计算深度到达maxDepth时停止计算。

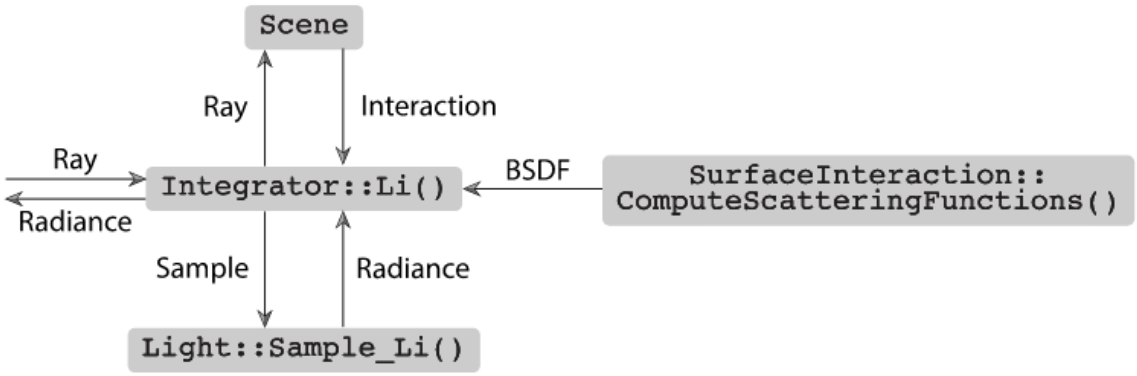


Figure 1.19: Class Relationships for Surface Integration. The main rendering loop in the `SamplerIntegrator` computes a camera ray and passes it to the `Li()` method, which returns the radiance along that ray arriving at the ray's origin. After finding the closest intersection, it computes the material properties at the intersection point, representing them in the form of a BSDF. It then uses the Lights in the Scene to determine the illumination there. Together, these give the information needed to compute the radiance reflected back along the ray at the intersection point.

```
///用一条射线到场景中采样，返回一个亮度
<<WhittedIntegrator Method Definitions>>=
Spectrum WhittedIntegrator::Li(const RayDifferential &ray,
        const Scene &scene, Sampler &sampler, MemoryArena
 &arena,
        int depth) const {
    Spectrum L(0.);
    //首先寻找射线与场景是否相交
    <<Find closest ray intersection or return background r
adiance>>
       SurfaceInteraction isect;
       if (!scene.Intersect(ray, &isect)) {
           //不相交，则把直射的环境光返回
           for (const auto &light : scene.lights)
               L += light->Le(ray);
           return L;
       }
    //与场景物体相交
    <<Compute emitted and reflected light at ray intersect
ion point>>
       <<Initialize common variables for Whitted integrato
r>>
          //在相交数据中返回了法线和出射方向wo
          Normal3f n = isect.shading.n;
          Vector3f wo = isect.wo;
        //决定表面参数，以及使用哪个bsdf函数
       <<Compute scattering functions for surface interact
ion>>
          isect.ComputeScatteringFunctions(ray, arena);

        //如果相交点表面有自发光，这里返回自发光亮度
       <<Compute emitted light if ray hit an area light so
urce>>
          L += isect.Le(wo);
        //遍历所有灯光
       <<Add contribution of each light source>>
          for (const auto &light : scene.lights) {
              Vector3f wi;
              Float pdf;
              VisibilityTester visibility;
```

```
                //灯光采样，当前相交点的入射光线强度，同时返回wi，当然还有pdf，用来做monte carlo积分估计，以及可见性。但是它并没有管这个光线是否会被遮挡，所以还返回了个Visibility Tester
                Spectrum Li = light->Sample_Li(isect, sampler.Get2D(), &wi,
                                                &pdf, &visibility);
                if (Li.IsBlack() || pdf == 0) continue;
                //bsdf决定入射光有百分之多少会反射给出射光
                Spectrum f = isect.bsdf->f(wo, wi);
                if (!f.IsBlack() && visibility.Unoccluded(scene))
                    L += f * Li * AbsDot(wi, n) / pdf;
            }
        //到此为止，自发光、灯光的贡献计算完毕，下面开始计算间接光照
        if (depth + 1 < maxDepth) {
            <<Trace rays for specular reflection and refraction>>
             //计算高光反射
            L += SpecularReflect(ray, isect, scene, sampler, arena, depth);
            //计算高光传输
            L += SpecularTransmit(ray, isect, scene, sampler, arena, depth);

        }

    return L;
}
///计算高光反射
<<SamplerIntegrator Method Definitions>>+=
Spectrum SamplerIntegrator::SpecularReflect(const RayDifferential &ray,
        const SurfaceInteraction &isect, const Scene &scene,
        Sampler &sampler, MemoryArena &arena, int depth) const {
    <<Compute specular reflection direction wi and BSDF value>>
        Vector3f wo = isect.wo, wi;
```

```
        Float pdf;
        BxDFType type = BxDFType(BSDF_REFLECTION | BSDF_SPE
CULAR);
        //计算高光信息，输出强度以及入射方向，意思是，如果有高光，那
么必须从这个入射方向过来，并且告诉你反射率是多少
        Spectrum f = isect.bsdf->Sample_f(wo, &wi, sampler.
Get2D(), &pdf, type);
         //接下来要计算的，是真正能提供高光的光线强度到底是多少，因
此这里需要发射一个射线来获取
    <<Return contribution of specular reflection>>
        const Normal3f &ns = isect.shading.n;
        if (pdf > 0 && !f.IsBlack() && AbsDot(wi, ns) != 0)
 {
            <<Compute ray differential rd for specular refl
ection>>
               RayDifferential rd = isect.SpawnRay(wi);
               if (ray.hasDifferentials) {
                   rd.hasDifferentials = true;
                   rd.rxOrigin = isect.p + isect.dpdx;
                   rd.ryOrigin = isect.p + isect.dpdy;
                   <<Compute differential reflected directi
ons>>
                      Normal3f dndx = isect.shading.dndu *
 isect.dudx +
                                              isect.shading.dndv *
 isect.dvdx;
                      Normal3f dndy = isect.shading.dndu *
 isect.dudy +
                                              isect.shading.dndv *
 isect.dvdy;
                      Vector3f dwodx = -ray.rxDirection - w
o, dwody = -ray.ryDirection - wo;
                      Float dDNdx = Dot(dwodx, ns) + Dot(w
o, dndx);
                      Float dDNdy = Dot(dwody, ns) + Dot(w
o, dndy);
                      rd.rxDirection = wi - dwodx +
                          2.f * Vector3f(Dot(wo, ns) * dndx
 + dDNdx * ns);
                      rd.ryDirection = wi - dwody +
```

```
                            2.f * Vector3f(Dot(wo, ns) * dndy
 + dDNdy * ns);

                }
             //有了Li入射光的强度，通过render equation返回间接光
的specular部分
            return f * Li(rd, scene, sampler, arena, depth
 + 1) * AbsDot(wi, ns) /
                    pdf;
        }
        else
            return Spectrum(0.f);

}
```

如果光线没有击中场景物体，则返回击中的光源亮度。如果击中了场景物体，则要按照render equation去计算亮度了。
Visibility Tester是通过 shadow ray的方式来确定是否发生了遮挡。Sample_Li函数如此设计可以避免不必要的shadow ray测试，例如如果表面不能透光(transmittance)，那么打在表面背面的光便不会对最终结果有贡献。
之后就会处理SpecularReflect和Specular Transmit
bsdf->Sample_f函数在给定反射类型以及出射方向的条件下，返回入射光方向，以及光照强度。我们用这个函数，来寻找与完美镜面反射和折射相关的入射光方向，然后我用这个方向来发射射线，求得这个方向上的亮度贡献，有了这个亮度贡献，再求出镜面反射折射的亮度。
Specular Transmit与Specular Reflect处理方式相同。

1.4 Parallelization of pbrt

在做光线追踪的时候，可以用并行的方式去做。
Amdahl's law，有一个n核的cpu，有百分比为s的串行任务，则最大加速可能性为：

$$\frac{1}{s+\frac{1}{n}(1-s)}$$

反正就是这么个统计公式，其实并不重要，看看就行了。

1.4.1 Data Races and Coordination

线程操作就会有竞争，有两种方法解决：互斥锁和原子操作。
另外还有一个叫做transactional memory的新型机制，部分cpu已经开始支持这个操作，

对于内存的写入操作打包成一个事务。

1.4.2 Conventions in pbrt

在pbrt中，射线跟踪都是对于内存进行读取操作，没有写入，解析场景文件是在一个线程里完成的，因此也没有多线程互操作的问题。但如果要修改场景内的资源，则要小心。

1.4.3 Thread Safety Expectations in pbrt

pbrt中大多数类都要求线程安全，要么是本身就是可重入的，要么就是要加线程锁。底层类不必考虑线程安全问题，因为这会带来额外的开销。

1.5 How to Proceed through this Book

略过

1.6 Using and Understanding the Code

略过

1.7 A Brief History of Physically Based Rendering

略过

1.8 Further Reading

继续略过

2 Geometry and Transformations

2.1 Coordinate Systems

pbrt用左手坐标系

2.2 Vectors

略

2.3 Points

略

2.4 Normals

略

2.5 Ray

2.5.1 Ray Differentials

为了让输出的纹理能有更好的抗锯齿效果，因此需要在ray上增加额外信息。它用于Texture类的计算。
它从Ray类派生，并且多了两根辅助射线，这两根辅助射线是相对于当前射线，在投影面上偏移了1个单位的方向。有了这个信息，Texture类就可以用于抗锯齿时估计平均值。

```
class RayDifferential : public Ray {
public:
    <<RayDifferential Public Methods>>
       RayDifferential() { hasDifferentials = false; }
       RayDifferential(const Point3f &o, const Vector3f &
d,
                  Float tMax = Infinity, Float time = 0.f,
                  const Medium *medium = nullptr)
           : Ray(o, d, tMax, time, medium) {
           hasDifferentials = false;
       }
       RayDifferential(const Ray &ray) : Ray(ray) {
           hasDifferentials = false;
       }
       bool HasNaNs() const {
           return Ray::HasNaNs() ||
```

```
                (hasDifferentials && (rxOrigin.HasNaNs() ||
    ryOrigin.HasNaNs() ||
                                        rxDirection.HasNaNs()
    || ryDirection.HasNaNs()));
        }
        void ScaleDifferentials(Float s) {
            //让参考射线偏移一定范围
            rxOrigin = o + (rxOrigin - o) * s;
            ryOrigin = o + (ryOrigin - o) * s;
            rxDirection = d + (rxDirection - d) * s;
            ryDirection = d + (ryDirection - d) * s;
        }

      <<RayDifferential Public Data>>
        bool hasDifferentials;
        Point3f rxOrigin, ryOrigin;
        Vector3f rxDirection, ryDirection;

    };
```

2.6 Bounding Boxes

略，没啥好讲的

2.7 Transformations

略，就是基本的矩阵变换

2.8 Applying Transformations

2.8.1 Points

2.8.2 Vectors

2.8.3 Normals

法线变换的方式与点不太一样。

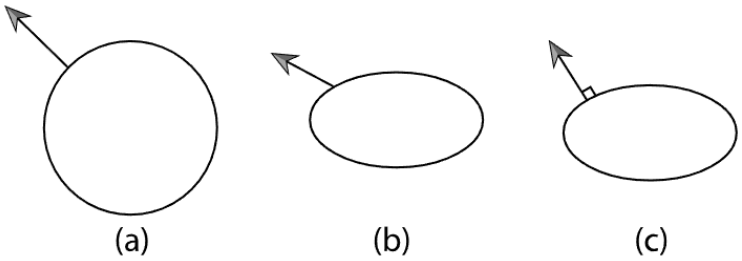


Figure 2.14: Transforming Surface Normals. (a) Original circle, with the normal at a point indicated by an arrow. (b) When scaling the circle to be half as tall in the y direction, simply treating the normal as a direction and scaling it in the same manner gives a normal that is no longer perpendicular to the surface. (c) A properly transformed normal.

首先，法线n与切线t是正交的，因此：

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

当用矩阵M转换表面上的一个点时，新的切线向量t'为Mt，那么转换后的法线n'应该等于Sn，其中S是4x4的矩阵。但为了让切线和法线保持正交，就需要维持下面这个等式：

$$\begin{aligned} 0 &= (n')^T t' \\ 0 &= (Sn)^T Mt \\ 0 &= (n)^T S^T Mt \end{aligned}$$

当$S^T M = I$的时候，等式成立，即$S^T = M^{-1}$，因此$S = (M^{-1})^T$。
于是我们得到结论：转换法线的矩阵，首先要将转换矩阵取逆矩阵，然后再转置。

2.8.4 Rays

2.8.5 Bounding Boxes

2.8.6 Composition of Transformations

2.8.7 Transformations and Coordinate System Handedness

transform本身可能带有转换左右手坐标系的功能

```
<<Transform Method Definitions>>+=
bool Transform::SwapsHandedness() const {
    Float det =
        m.m[0][0] * (m.m[1][1] * m.m[2][2] - m.m[1][2] *
 m.m[2][1]) -
        m.m[0][1] * (m.m[1][0] * m.m[2][2] - m.m[1][2] *
 m.m[2][0]) +
```

```
            m.m[0][2] * (m.m[1][0] * m.m[2][1] - m.m[1][1] *
    m.m[2][0]);
        return det < 0;
    }
```

2.9 Animating Transformations

引入transform动画，需要处理的就是关键帧插值问题。
在这里把变换矩阵M分拆为SRT矩阵

$$M = SRT$$

分别进行处理

2.9.1 Quaternions

四元数实现，略

2.9.2 Quaternion Interpolation

四元数插值，略

2.9.3 Animated Transform Implementation

transform插值实现，略

2.9.4 Bounding Moving Bounding Boxes

AnimatedTransform::MotionBounds()函数，传入一个bounding box，返回经过一段时间段内的bounding box（不仅仅是结尾时间点的bounding box，而且包括了变换过程）。
这里有两种简单情形，

- 如果前后两帧变换相同，则只要用开始的那个变换计算就行了。
- 如果变换只包括缩放和平移，则最终的bounding box包含了所有开始时间和结束时间这两个时间点，经过变换的顶点位置，即两个包围盒合并。
  证明：
  如果变换中的旋转为单位旋转，则

$$a(M_0, M_1, p, t) = T(t)S(t)p$$

  对于p的x分量：

$$a(M_0, M_1, p, t)_x = [(1-t)s_{0,0} + ts'_{0,0}]p_x + (1-t)d_{0,3} + td'_{0,3}$$
$$= [s_{0,0}p_x + d_{0,3}] + [-s_{0,0}p_x + s'_{0,0}p_x - d_{0,3} + d'_{0,3}]t$$

其中$s_{0,0}$为M_0矩阵的对应元素，$s'_{0,0}$为M_1矩阵的对应元素，平移变换元素用字母d表示。这是个线性变换，因此极值就在开始和结束点。

```
<<AnimatedTransform Method Definitions>>+=
Bounds3f AnimatedTransform::MotionBounds(const Bounds3f &
b) const {
    if (!actuallyAnimated)
        return (*startTransform)(b);
    if (hasRotation == false)
        return Union((*startTransform)(b), (*endTransform)
(b));
    <<Return motion bounds accounting for animated rotatio
n>>
       Bounds3f bounds;
       for (int corner = 0; corner < 8; ++corner)
           bounds = Union(bounds, BoundPointMotion(b.Corne
r(corner)));
       return bounds;

}
```

一旦带上旋转，事情就复杂了。极值可能出现在两个时间点中间，并没有一个简单的办法去解决这个问题。这里会给出一个常用方法。
我们用一个稍微简单保守的的包围盒来包含bounding box 8个顶点。我们要在动画事件段内，寻找bounding box的任意一个顶点p的极值a。回顾微积分中在一个定义域内寻找函数极值的办法，要么在两个端点，要么在中间导数为0的地方。因此这个最终的bounding box应该包含开始和结束的bounding box的顶点，以及过程中所有达到极值的顶点。

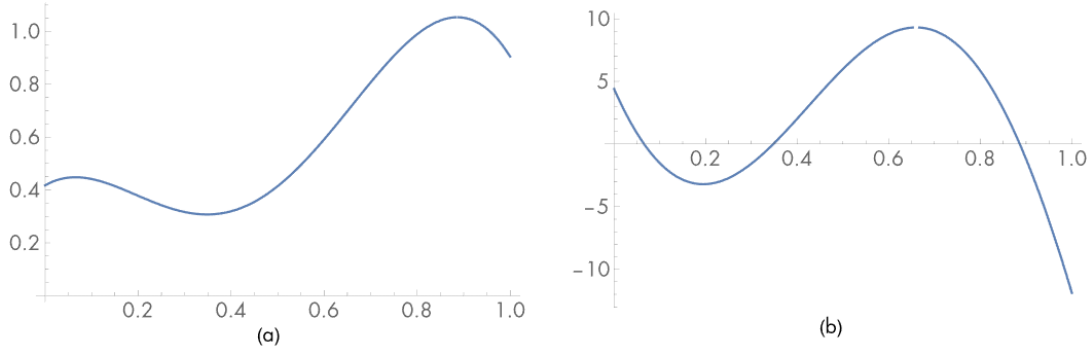


Figure 2.18: (a) Motion of the x coordinate of a point p as a function of time, as determined by two keyframe matrices. (b) The derivative of the motion function, Equation (2.12). Note that extrema of the motion function in the given time range correspond to zeros of the derivative.

图2.18展示的是一个坐标轴在动画过程中的值，明显的，极值出现在导数为0的地方。为了获得这个点的bound，我们开始推导，首先顶点变换函数a扩展为带旋转的版本：

$$a(M_0, M_1, p, t) = T(t)R(t)S(t)p$$

对于时间t的导数表达式$\frac{\partial a(M_0,M_1,p,t)}{\partial t}$非常复杂，但如果给定了变换矩阵以及点p的话，表达式能简单的多。这里将此函数记为$a_{M,p}(t)$，这里直接给出表达式：

$$\frac{da_{M,p}(t)}{dt} = c_1 + (c_2 + c_3 t)\cos(2\theta t) + (c_4 + c_t t)\sin(2\theta t)$$

θ对应的$\cos\theta$是两个四元数的点乘，5个c参数是vector类型的，基于前后两个矩阵以及点p的值。
现在，我们可以确定流程：

1. 拿到关键帧的变换矩阵和点p，计算出所有c系列参数。
2. 代入上面那个式子，计算这个式子等于0时的t为多少，可能有多个t值。

对于第1步，实际上c_i向量就是点p的xyz分量的线性组合。

$$c_i(p) = k_{i,c} + k_{i,x}p_x + k_{i,y}p_y + k_{i,z}p_z$$

只要有k_I系列参数以及点p，就能求出来了。

```
<<AnimatedTransform Private Data>>+=
struct DerivativeTerm {
    DerivativeTerm(Float c, Float x, Float y, Float z)
        : kc(c), kx(x), ky(y), kz(z) { }
    Float kc, kx, ky, kz;
    Float Eval(const Point3f &p) const {
        return kc + kx * p.x + ky * p.y + kz * p.z;
    }
```

```
  };
<<AnimatedTransform Private Data>>+=
///每个维度都有一个c1参数
DerivativeTerm c1[3], c2[3], c3[3], c4[3], c5[3];


<<AnimatedTransform Method Definitions>>+=
Bounds3f AnimatedTransform::BoundPointMotion(const Point3f
 &p) const {
    Bounds3f bounds((*startTransform)(p), (*endTransform)
(p));
    Float cosTheta = Dot(R[0], R[1]);
    Float theta = std::acos(Clamp(cosTheta, -1, 1));
    for (int c = 0; c < 3; ++c) {
        <<Find any motion derivative zeros for the compone
nt c>>
           Float zeros[4];
           int nZeros = 0;
           //计算那个方程
           IntervalFindZeros(c1[c].Eval(p), c2[c].Eval(p),
 c3[c].Eval(p),
                             c4[c].Eval(p), c5[c].Eval(p),
 theta,
                             Interval(0., 1.), zeros, &nZe
ros);

        //得到时间t之后，就可以代入方程，求得极值
        <<Expand bounding box for any motion derivative ze
ros found>>
           for (int i = 0; i < nZeros; ++i) {
               Point3f pz = (*this)(Lerp(zeros[i], startTi
me, endTime), p);
               bounds = Union(bounds, pz);
           }

    }
    return bounds;
}
```

在 AnimatedTransform的构造函数中，会填充c1到c5的所有必要参数。
IntervalFindZeros函数是用来计算那个求导后的方程等于0的时候的解，但并没有一个解

析解，需要进行数值计算求解，可以用bisection-based搜索，或者牛顿迭代法。但这个方程本身性质比较好，是光滑的，并且有有限个0值。
但用数值计算也有问题，如果曲线仅仅是短暂地穿过横轴的话，可能会忽略掉这个解。因此我们使用interval arithemetic区间算术。
假设有个函数$f(x) = 2x$。如果我们有一个区间$[a, b] \subset R$，观察这个区间，f的值域是$[2a, 2b]$，换种写法就是$f([a, b]) \subset [2a, 2b]$。
更宽泛地讲，所有对于数字的操作，它的区间扩展版本描述了它如何作用于一个区间上。例如我们有两个区间$[a, b]$和$[c, d]$，我们有如下关系：

$$[a, b] + [c, d] \subset [a + c, b + d]$$

这个式子的意思是，如果我们将$[a, b]$和$[c, d]$区间的两个值相加，那这个结果一定在区间$[a + c, b + d]$中。
区间算术的一个重要的性质就是它返回的区间是保守的。如果$f([a, b]) \subset [c, d]$，并且如果$c > 0$，我们可以确定$[a, b]$中的任何一个值都无法使得函数f的值为负数。
下面展示如何在区间上计算方程

$$\frac{da_{M,p}(t)}{dt} = c_1 + (c_2 + c_3 t)\cos(2\theta t) + (c_4 + c_t t)\sin(2\theta t)$$

以及运用计算出来的保守范围的优势，去寻找函数跨过横轴的小区间。
首先定义Interval类

```
<<Interval Definitions>>=
class Interval {
public:
  <<Interval Public Methods>>
      Interval(Float v) : low(v), high(v) { }
      Interval(Float v0, Float v1)
          : low(std::min(v0, v1)), high(std::max(v0, v1)) {
 }
      Interval operator+(const Interval &i) const {
          return Interval(low + i.low, high + i.high);
      }
      Interval operator-(const Interval &i) const {
          return Interval(low - i.high, high - i.low);
      }
      //对于乘法而言，涉及到了正负号，所以都得乘一遍以找到最大最小值
      Interval operator*(const Interval &i) const {
          return Interval(std::min(std::min(low * i.low,  h
igh * i.low),
```

```
                                        std::min(low * i.high, h
igh * i.high)),
                            std::max(std::max(low * i.low,  h
igh * i.low),
                                        std::max(low * i.high, h
igh * i.high)));
     }


  Float low, high;
};
```

sin和cos函数也有区间版本。这里假设定义域是$[0, 2\pi]$。

```
inline Interval Sin(const Interval &i) {
            Float sinLow = std::sin(i.low), sinHigh =
std::sin(i.high);
            if (sinLow > sinHigh)
                std::swap(sinLow, sinHigh);
            //越过了最大值，取最大值
            if (i.low < Pi / 2 && i.high > Pi / 2)
                sinHigh = 1.;
            //越过了最小值，取最小值
            if (i.low < (3.f / 2.f) * Pi && i.high > (3.f
/ 2.f) * Pi)
                sinLow = -1.;
            return Interval(sinLow, sinHigh);
    }
```

有了区间运算机制，终于可以去实现IntervalFindZeros函数了，这个函数的作用就是去寻找那个导数函数等于0的时候t的值。

```
<<Interval Definitions>>+=
void IntervalFindZeros(Float c1, Float c2, Float c3, Float
 c4,
        Float c5, Float theta, Interval tInterval, Float *
zeros,
        int *zeroCount, int depth = 8) {
    //计算这个方程在区间内的值域
```

```
    <<Evaluate motion derivative in interval form, return
if no zeros>>
       Interval range = Interval(c1) +
           (Interval(c2) + Interval(c3) * tInterval) *
               Cos(Interval(2 * theta) * tInterval) +
           (Interval(c4) + Interval(c5) * tInterval) *
               Sin(Interval(2 * theta) * tInterval);
       //如果值域不跨过0，则没有解
       if (range.low > 0. || range.high < 0. || range.low
 == range.high)
           return;
    //因为是保守区间，所以范围要更宽泛一些，这里可以递归地将区间拆
分，左边一半右边一半。传入的depth来决定可以拆多细
    if (depth > 0) {
        <<Split tInterval and check both resulting interva
ls>>
           Float mid = (tInterval.low + tInterval.high) *
0.5f;
           IntervalFindZeros(c1, c2, c3, c4, c5, theta,
               Interval(tInterval.low, mid), zeros, zeroCo
unt, depth - 1);
           IntervalFindZeros(c1, c2, c3, c4, c5, theta,
               Interval(mid, tInterval.high), zeros, zeroC
ount, depth - 1);

    } else {
        //牛顿迭代法求解
        <<Use Newton's method to refine zero>>
           Float tNewton = (tInterval.low + tInterval.hig
h) * 0.5f;
           for (int i = 0; i < 4; ++i) {
               Float fNewton = c1 +
                   (c2 + c3 * tNewton) * std::cos(2.f * th
eta * tNewton) +
                   (c4 + c5 * tNewton) * std::sin(2.f * th
eta * tNewton);
               Float fPrimeNewton =
                   (c3 + 2 * (c4 + c5 * tNewton) * theta)
 *
                       std::cos(2.f * tNewton * theta) +
```

```
                    (c5 - 2 * (c2 + c3 * tNewton) * theta)
 *
                        std::sin(2.f * tNewton * theta);
                if (fNewton == 0 || fPrimeNewton == 0)
                    break;
                tNewton = tNewton - fNewton / fPrimeNewton;
            }
            zeros[*zeroCount] = tNewton;
            (*zeroCount)++;

    }
}
```

首先列出需要求解的方程

$$\frac{da_{M,p}(t)}{dt} = c_1 + (c_2 + c_3 t)\cos(2\theta t) + (c_4 + c_t t)\sin(2\theta t)$$

通过区间计算，算出个保守区间，保守区间肯定是比较宽的，因此可以通过递归细分来分成更细的空间，最终每个子空间通过牛顿迭代法来寻找解。牛顿迭代法公式：

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

牛顿迭代法需要对$a_{M,p}(t)$求二阶导。

$$\frac{d^2 a_{M,p}(t)_x}{dt^2} = (c_{3,x} + 2\theta(c_{4,x} + c_{5,x}t))\cos(2\theta t) + (c_{5,x} - 2\theta(c_{2,x} + c_{3,x}t))\sin(2\theta t)$$

如果一个区间内有多个零点，则只取一个，因为区间被划分的比较小，所以问题不大。

2.10 Interactions

本章最后要介绍的数据结构是SurfaceInteraction，它表示了在一个2D表面上的一个点的信息。相关的类是MediumInteraction，是用来表示光线辐射到参与媒介participating media上的点。
这两个类都从Interaction类派生

```
<<Interaction Declarations>>=
struct Interaction {
    <<Interaction Public Methods>>
       Interaction() : time(0) { }
```

```
        Interaction(const Point3f &p, const Normal3f &n, co
nst Vector3f &pError,
                    const Vector3f &wo, Float time,
                    const MediumInterface &mediumInterface)
            : p(p), time(time), pError(pError), wo(wo), n
(n),
              mediumInterface(mediumInterface) { }
        bool IsSurfaceInteraction() const {
            return n != Normal3f();
        }
        Ray SpawnRay(const Vector3f &d) const {
            Point3f o = OffsetRayOrigin(p, pError, n, d);
            return Ray(o, d, Infinity, time, GetMedium(d));
        }
        Ray SpawnRayTo(const Point3f &p2) const {
            Point3f origin = OffsetRayOrigin(p, pError, n,
 p2 - p);
            Vector3f d = p2 - origin;
            return Ray(origin, d, 1 - ShadowEpsilon, time,
 GetMedium(d));
        }
        Ray SpawnRayTo(const Interaction &it) const {
            Point3f origin = OffsetRayOrigin(p, pError, n,
 it.p - p);
            Point3f target = OffsetRayOrigin(it.p, it.pErro
r, it.n, origin - it.p);
            Vector3f d = target - origin;
            return Ray(origin, d, 1-ShadowEpsilon, time, Ge
tMedium(d));
        }
        Interaction(const Point3f &p, const Vector3f &wo, F
loat time,
                    const MediumInterface &mediumInterface)
            : p(p), time(time), wo(wo), mediumInterface(med
iumInterface) { }
        Interaction(const Point3f &p, Float time,
                    const MediumInterface &mediumInterface)
            : p(p), time(time), mediumInterface(mediumInter
face) { }
        bool IsMediumInteraction() const { return !IsSurfac
eInteraction(); }
```

```
        const Medium *GetMedium(const Vector3f &w) const {
            return Dot(w, n) > 0 ? mediumInterface.outside
 :
                                   mediumInterface.inside;
        }
        const Medium *GetMedium() const {
            Assert(mediumInterface.inside == mediumInterfac
e.outside);
            return mediumInterface.inside;
        }

    <<Interaction Public Data>>
        ///击中点
        Point3f p;
        Float time;
        ///浮点误差
        Vector3f pError;
        ///射线的反方向
        Vector3f wo;
        ///击中点的法线
        Normal3f n;
        ///介质信息
        MediumInterface mediumInterface;

};
```

2.10.1 Surface Interaction

几何体上的某一点（通常是射线击中的那点）的信息，通过SurfaceInteraction数据结构来存储。

```
<<SurfaceInteraction Declarations>>=
class SurfaceInteraction : public Interaction {
public:
    <<SurfaceInteraction Public Methods>>
        SurfaceInteraction() { }
        SurfaceInteraction(const Point3f &p, const Vector3f
 &pError, const Point2f &uv,
            const Vector3f &wo, const Vector3f &dpdu, const
 Vector3f &dpdv,
```

```
            const Normal3f &dndu, const Normal3f &dndv,
            Float time, const Shape *sh);
        void SetShadingGeometry(const Vector3f &dpdu, const
 Vector3f &dpdv,
             const Normal3f &dndu, const Normal3f &dndv, bo
ol orientationIsAuthoritative);
        void ComputeScatteringFunctions(const RayDifferenti
al &ray,
            MemoryArena &arena, bool allowMultipleLobes = f
alse,
            TransportMode mode = TransportMode::Radiance);
        void ComputeDifferentials(const RayDifferential &r)
 const;
        Spectrum Le(const Vector3f &w) const;

    <<SurfaceInteraction Public Data>>
        ///击中点的uv坐标
        Point2f uv;
        ///击中点坐标对于uv的偏导
        Vector3f dpdu, dpdv;
        ///法线方向对于uv的偏导å
        Normal3f dndu, dndv;
        const Shape *shape = nullptr;
        //shading结构体中，又存了一份上面的数据
        struct {
            Normal3f n;
            Vector3f dpdu, dpdv;
            Normal3f dndu, dndv;
        } shading;
        const Primitive *primitive = nullptr;
        BSDF *bsdf = nullptr;
        BSSRDF *bssrdf = nullptr;
        mutable Vector3f dpdx, dpdy;
        mutable Float dudx = 0, dvdx = 0, dudy = 0, dvdy =
0;

};
```

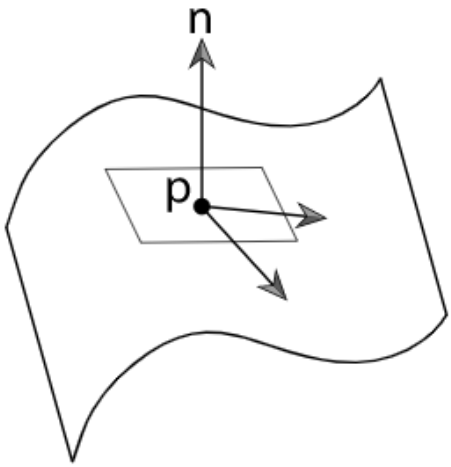


Figure 2.19: The Local Differential Geometry around a Point p. The parametric partial derivatives of the surface, $\partial p/\partial u$ and $\partial p/\partial v$, lie in the tangent plane but are not necessarily orthogonal. The surface normal n is given by the cross product of $\partial p/\partial u$ and $\partial p/\partial v$. The vectors $\partial n/\partial u$ and $\partial n/\partial v$ (not shown here) record the differential change in surface normal as we move u and v along the surface.

这种表示方式隐含着位置p是uv的函数，即$p = f(u, v)$，尽管对于有些形状来说不是这么回事，但在pbrt中所有的形状都有一个参数化的描述， 因此这么写有它方便的地方。在构造函数中，会填充所有数据。
这里的shading geometry是用来存储轻微扰动信息的结构体，例如被bump mapping和三角形的顶点法线插值。

```
<<SurfaceInteraction Method Definitions>>=
SurfaceInteraction::SurfaceInteraction(const Point3f &p,
        const Vector3f &pError, const Point2f &uv, const V
ector3f &wo,
        const Vector3f &dpdu, const Vector3f &dpdv,
        const Normal3f &dndu, const Normal3f &dndv,
        Float time, const Shape *shape)
    : Interaction(p, Normal3f(Normalize(Cross(dpdu, dpd
v))), pError, wo,
                  time, nullptr),
      uv(uv), dpdu(dpdu), dpdv(dpdv), dndu(dndu), dndv(dnd
v),
      shape(shape) {
      //这里额外了填充了一个叫做shading geometry的结构体，一开
始，它和surface geometry是一样的
    <<Initialize shading geometry from true geometry>>
       shading.n = n;
       shading.dpdu = dpdu;
       shading.dpdv = dpdv;
```

```
        shading.dndu = dndu;
        shading.dndv = dndv;
    //文件中可能反转了法线或者手性发生变化，这里用来处理这个情况
    <<Adjust normal based on orientation and handedness>>
        if (shape && (shape->reverseOrientation ^
                      shape->transformSwapsHandedness)) {
            n *= -1;
            shading.n *= -1;
        }

}
```

表面的法线在pbrt有特殊含义，对于一个用于区域光的几何体，光发射的方向是其法线方向，背向法线的方向是没有光线发射的。pbrt中文件描述有个ReverseOrientation字段，来标记是否反转了法线。
pbrt中形状的坐标系是左手坐标系，如果要转换到右手坐标系，则法线也要进行处理。考虑缩放矩阵$S(1,1,-1)$，我们本可以认为用这个矩阵来切换法线的手性，但我们用来计算法线的方法为$n = \frac{\partial p}{\partial u} \times \frac{\partial p}{\partial v}$，

$$\begin{aligned} S(1,1,-1)\frac{\partial p}{\partial u} \times S(1,1,-1)\frac{\partial p}{\partial v} &= S(-1,-1,1)\frac{\partial p}{\partial u} \times \frac{\partial p}{\partial v} \\ &= S(-1,-1,1)n \\ &\neq S(1,1,-1)n \end{aligned}$$

很显然，转换手性之后应该再做一次反转操作才能得到正确的法线。

```
<<Adjust normal based on orientation and handedness>>=
if (shape && (shape->reverseOrientation ^
              shape->transformSwapsHandedness)) {
    n *= -1;
    shading.n *= -1;
}
```

SetShadingGeometry可以用来更新数据

```
<<SurfaceInteraction Method Definitions>>+=
void SurfaceInteraction::SetShadingGeometry(const Vector3f
 &dpdus,
        const Vector3f &dpdvs, const Normal3f &dndus,
```

```
        const Normal3f &dndvs, bool orientationIsAuthorita
tive) {
        //这里shading geometry就跟surface geometry不一样了
    <<Compute shading.n for SurfaceInteraction>>
       shading.n = Normalize((Normal3f)Cross(dpdus, dpdv
s));
       if (shape && (shape->reverseOrientation ^
                     shape->transformSwapsHandedness))
           shading.n = -shading.n;
       if (orientationIsAuthoritative)
           n = Faceforward(n, shading.n);
       else
           shading.n = Faceforward(shading.n, n);

    <<Initialize shading partial derivative values>>
       shading.dpdu = dpdus;
       shading.dpdv = dpdvs;
       shading.dndu = dndus;
       shading.dndv = dndvs;

}
```

因为都是执行同样的叉乘，shading geometry代表的是一个轻微的扰动，所以二者当中，它们的法线至少应该是在同一个半球上的，基于不同的上下文，可以设定它们其中一个是权威数据，指向了正确的表面外部，所以需要传入orientationIsAuthoritative变量来指明这一点。
同样，transform结构体也可以变换SurfaceInteraction类型的数据。

```
<<Transform Method Definitions>>+=
SurfaceInteraction
Transform::operator()(const SurfaceInteraction &si) const
{
    SurfaceInteraction ret;
    <<Transform p and pError in SurfaceInteraction>>
       ret.p = (*this)(si.p, si.pError, &ret.pError);

    <<Transform remaining members of SurfaceInteraction>>
       const Transform &t = *this;
       ret.n = Normalize(t(si.n));
       ret.wo = t(si.wo);
       ret.time = si.time;
```

```
        ret.mediumInterface = si.mediumInterface;
        ret.uv = si.uv;
        ret.shape = si.shape;
        ret.dpdu = t(si.dpdu);
        ret.dpdv = t(si.dpdv);
        ret.dndu = t(si.dndu);
        ret.dndv = t(si.dndv);
        ret.shading.n = Normalize(t(si.shading.n));
        ret.shading.dpdu = t(si.shading.dpdu);
        ret.shading.dpdv = t(si.shading.dpdv);
        ret.shading.dndu = t(si.shading.dndu);
        ret.shading.dndv = t(si.shading.dndv);
        ret.dudx = si.dudx;
        ret.dvdx = si.dvdx;
        ret.dudy = si.dudy;
        ret.dvdy = si.dvdy;
        ret.dpdx = t(si.dpdx);
        ret.dpdy = t(si.dpdy);
        ret.bsdf = si.bsdf;
        ret.bssrdf = si.bssrdf;
        ret.primitive = si.primitive;
        //    ret.n = Faceforward(ret.n, ret.shading.n);
        ret.shading.n = Faceforward(ret.shading.n, ret.n);

    return ret;
}
```