zEngine-Stone

2006年3月6日，11:41:00 | fire

C++编程规范
Exceptional C++ Style
Effecive C++
More EffectiveC++
C++ Primer
Cg Tutorail
Dx9 Sdk
Dx10Sdk
Sort,Search,A*
GPU GEMS1
GPU GEMS2

WinMine3D ·

fire 2006-03-06 11:41 发表评论

注释(0)

pbr随笔

2006年1月19日，17:31:00 | fire

[光的基本传递模型]
1 在一个要渲染的场景中，我们认为光能由预先指定的光源发出，然后我们以光线来描述光能的传递过程，当整个场景中的光能信息被我们计算出来后，我们收集这些信息转化为顶点的亮度。
2 光线经过物体表面可以产生反射和漫反射，光线透过物体可以产生折射和散射。具体产生哪种出射效果，依据物体的表面属性而定。物体的表面一般不会是理想的某种单一属性的表面，表面可以同时存在反射，折射，漫反射等多种属性，各种属性按一定比例混合之后才是其表面反射模型。
3 一点的在某一个视线方向上的光亮度＝该点在该方向的自身发光亮度＋半球入射光能在该方向所产生的反射光亮度.
4 关于散射，高度真实的散射是一个很难模拟的物理过程，一般在渲染中都不会采用过于复杂的物理模型来表示散射，而是采用一些取巧的办法来计算散射。
5 在常见的渲染中，有两种效果很难模拟，但是它们会使人眼觉得场景更真实。
[1]color bleeding :入射光为漫反射，受光表面属性为漫反射，出射光是漫反射。比如把一本蓝色的纸制的书靠近白色的墙，墙上会有浅浅的蓝晕。
[2]caustics:入射光为镜面反射或折射，受光表面属性为漫反射，出射光是漫反射。比如把一个装了红色葡萄酒的酒杯放在木桌上面，会有光透过杯中的酒在桌上形成一块很亮的红色区域。

[传统的阴影算法]:
游戏中传统的光照算法，是利用公式法来计算特定类型光源的直接光照在物体表面所产生的反射和漫反

射颜色，然后再使用阴影算法做阴影补偿。标准的阴影算法不能计算面光源，改进以后的阴影算法通过对面光源采样，可以模拟出软阴影的效果。但是这些方法计算的光照都是来自直接光源的，忽略了光的传播过程，也就无法计算出由光的传播所产生的效果。通过特定的修正，我们也可以计算特定的反射折射或漫反射过程，但是无法给出一种通用并且物理正确的方法。目前游戏中大多是采用改进的阴影算法来进行渲染，它的优点是效率比较高，结合预计算的话，还是可以产生比较生动可信的效果。

[传统的逆向光线追踪]

正如前面描述的那样，要想计算光能在场景中产生的颜色，最自然的考虑就是，从光源出发，正向跟踪每一根光线在场景中的传递过程，然后收集信息。然而这个想法在被提出的来的那个时代的计算机硬件上是不可能实现的，当时人们认为，正向光线追踪计算了大量对当前屏幕颜色不产生贡献的信息，而且它把看不见的物体也计算在内，极大的浪费了效率。

于是人们想出的另一个方法是：只计算有用的，从人眼出发，逆向跟踪光线。

逆向光线追踪从视点出发，向投影屏幕发出光线，然后追踪这个光线的传递过程。如果这个光线经过若干次反射折射后打到了光源上，则认为该光线是有用的，递归的计算颜色，否则就抛弃它。很显然，这个过程是真实光线投射的逆过程，它同样会产生浪费（那些被抛弃的逆向光线），而且只适用于静态渲染。

逆向光线追踪算法中的顶点亮度主要包括三个方面：

1由光源直接照射而引起的光亮度

2来自环境中其它景物的反射折射光在表面产生的镜面反射光亮度

3来自环境中其它景物的反射折射光在表面产生的规则透射光亮度

4预设定的顶点漫反射颜色

显然，这一过程仅跟踪景物间的镜面反射光线和规则透射光线，忽略了至少经过一次漫反射之后光能传递，而且该算法中的物体表面属性只能是单一的，因而它仅模拟了理想表面的光能传递。

对于该算法的具体描述:

1从视点出发，经过投影屏幕上的每一个像素向场景发射一根虚拟的光线。

2求光线与场景最近的交点。

3递归跟踪：

(1)如果当前交点所在的景物表面为理想镜面，光线沿其镜面反射方向继续跟踪。

(2)如果当前交点所在的景物表面为规则投射表面，光线沿其规则投射方向继续跟踪。

4递归异常结束：

(1)光线与场景中的景物没有交点

(2)当前交点所在的景物表面为漫反射表面

(3)跟踪层次已经超过用户设定的最大跟踪层数

(4)所跟踪的光线对显示像素的光亮度的贡献小于一预先设定的阀值

5递归正常结束：

(1)光线于光源相交，取得光亮度值，按递归层次反馈。

传统的光线追踪技术可以较好的表现出反射折射效果，也可以生成真实度比较高的阴影。但是他的光照都比较硬，无法模拟出非常细腻的柔化效果。

光线追踪需要对大量的光线进行多次与场景中物体的求交计算。如何避免这些求交计算成为光线追踪追求效率的本质。早期的光线追踪算法都是通过各种空间划分技术来避免无谓的求交检测，这些方法对于之后的理论同样有效，常见的空间划分方法分为两类，一类是基于网格的平均空间划分，一类是基于轴平行的二分空间划分。

[蒙特卡罗光线追踪]

1对传统的逆向光线追踪的改进

传统的逆向光线追踪算法有两个突出的缺点，就是表面属性的单一，和不考虑漫反射。我们不难通过模型的修正来缓解这两个问题。我们首先认为一个表面的属性可以是混合的，比如它有20%的成分是反射，30%的成分是折射，50%的成分是漫反射。这里的百分比可以这样理解，当一根光线打在该表面后，它有20%的概率发生反射，30%的概率发生折射，50%的概率发生漫反射。然后我们通过多次计算

光线跟踪，每次按照概率决定光线的反射属性，这样在就把漫反射也考虑了进去。具体的算法如下：
(1)从视点出发，经过投影屏幕上的每一个像素向场景发射一根虚拟的光线。
(2)当光线与景物相交时按照俄罗斯轮盘赌规则决定他的反射属性。
(3)根据不同的反射属性继续跟踪计算，直到正常结束或者异常结束。如果反射的属性为漫反射，则随机选择一个反射方向进行跟踪。
(4)重复前面的过程，把每次渲染出来的贴图逐像素叠加混合，直到渲染出的结果达到满意程度。
该方法是一种比较简易的基于物理模型的渲染，其本质就是通过大量的随机采样来模拟半球积分。这种方法在光照细节上可以产生真实度很高的图像，但是图像质量有比较严重的走样，而且效率极其低下。

2蒙特卡罗光线追踪-采样
蒙特卡罗光线追踪的本质就是通过概率理论，把半球积分方程进行近似简化，使之可以通过少量相对重要的采样来模拟积分。蒙特卡罗光线追踪理论中的采样方案有很多，有时候还要混合使用这些采样方案。
蒙特卡罗光线追踪已经是一个比较完备的渲染方案，他极大的解决了光线追踪的模型缺陷和效率问题，使得在家用图形硬件上做基于物理的渲染成为一种可能。但是我们仍然无法实时的进行计算，而且如何解决图像走样的问题也是蒙特卡罗光线追踪的一大难点。
相对于普通光线追踪，蒙特卡罗光线追踪引入了更复杂的漫反射模型，从而增加了需要跟踪的光线数量。但是他又通过采样算法减少了需要跟踪光线，所以其核心效率取决于采样模型。
与普通光线追踪一样，为了减少不必要的求交检测，蒙特卡罗光线追踪也需要使用空间划分技术，最常用的是平衡kdtree。蒙特卡罗光线追踪虽然是一种逆向光线追踪算法，但是其采样的理论却与光线追踪的方向无关，可以用于任何一种渲染方案。
此外，使用蒙特卡罗光线追踪不容易计算caustics现象。也就是说它不容易计算由镜面反射或者规则透射引起的漫反射。（但是很容易计算由漫反射引起的镜面反射或者规则透射；）

蒙特卡罗光线追踪本身也是一种逆向光线跟踪。逆向光线追踪最初被设计出来是为了只计算那些会影响最终屏幕像素的光能传递过程，这一思想在早期硬件并不发达，对最终影响要求也不高的年代是非常实用的。但是我认为由于对屏幕上每个像素的跟踪都是无关的，即每两次跟踪之间都不会建立通信说哪些是计算过的，哪些是没计算过的，所以这里面必然会包含大量的重复计算的中间过程。当我们对图像所表现效果的真实度非常高的时候，必然会产生巨量的采样，然后重复计算的问题就会被放大，而由逆向追踪思想带来的那些优势也将荡然无存。而且，对场景中光能贡献越大的光源应该被越多的采样跟踪覆盖到，但是逆向光线跟踪只是对屏幕上每个像素反复遍历追踪，其结果应该趋向于采样平均覆盖各个光源，如果要想对高亮度光源采很多的样本，必然也会导致对其它光源也过多的采了样本，这会非常浪费效率。重新考虑正向光线追踪，光由光源发出，打在场景之中，每一次光能转化都被记录下来，最后只要收集这些信息就可以知道任意点上面的亮度，这个方法的描述非常的贴近真实的自然，关键在于如何保证速度。
另外，完全的逆向光线追踪根本就不应该作为实时渲染的算法，道理很简单，光能的传递过程不变，只要视点一变，就要重新计算。

[辐射度算法]
辐射度的算法分为三个步骤
1先把场景中的面划分为一个个小的patch，然后计算两个patch之间的形式因子。两个patch之间的形式因子表示了一个patch出射的光有多少比例会被另一个patch接收。对于任意一个有n个patch的场景来说，总有n*(n-1)个形式因子。
2通过迭代法来找到一个光能传递的平衡状态
3把第二步所产生的亮度值作为顶点色渲染
辐射度算法会非常的慢，而且如果不考虑额外的复杂度，辐射度算法很难计算镜面反射，改进的辐射度算法可以缓解这一问题。很多研究者都试图结合光线追踪和辐射度这两种方法，以期达到各自的优势。

[photonmaping+final Gathering]:
前面谈到，正向光线追踪才是最自然的光能传递的描述，由此，在1994年，有人提出了photonmaping算

法。photonmaping是一个两步的算法，第一步通过正向光线跟踪来构建光子图，第二步通过光子图中的信息来渲染整个场景。它的核心思想是从光源开始追踪光能的传递，把每一个传递中间过程都记录下来，最后按照投影或者逆向光线追踪来收集这些信息，以达到渲染的目的。由于中间每一个光线和场景的相交都被记录下来，所以他很自然的避免了逆向光线追踪中重复计算的问题。

具体的，这两步算法又可以分为下面四步。
1从光源发射出N根采样光线。光线的方向和光源的类型有关。采样光线的数目选择与光源自身的亮度有关，越亮的光源应该选择越多的采样。
2光子打到场景中，一步步传递，把光能传递的过程记录下来，结果放在kdtree中
3用逆向光线追踪或者反投影的方法找到可视点
4使用逆向光线追踪和半球积分（比如最终聚集）方法收集光子图中的信息，从而计算可视点的光亮度。

首先要选择一个光源，然后才能发射一个光子。对于场景中的多个光源，每次做发射一个光子采样的时候，不能完全的随机选择光源，一个光源被选中的概率要正相关于他的在该场景中的能量
典型的，光源一般被分为：
(1)点光源:
点光源的数据结构仅仅是三维空间中的一个坐标。对点光源所发出的光线进行采样时，可以在包围该点的单位球上任选一点，然后以球心到该点的射线作为采样光线。也有人建议用单位立方体包围盒采样来代替单位球。
(2)方形面光源:
方形面光源上的每一个点都可以看做一个只能从靠近面法向量一侧发射光线的点光源。
(3)其它光源:
任意空间形状和物理特性的光源，只能具体问题具体分析。

一旦选好了初始要追踪的光子向量的位置和方向，我们就可以开始一次正向追踪。
一般的，光子在与场景中景物的相交的情况可以分为三类
(1)如果光子打到了镜面反射表面或者规则折射表面，不用做任何记录，继续追踪。
(2)如果光子打到了漫反射表面，则把光子所携带的能量和入射方向记录下来。
如果入射光是折射或者反射光，则把光子记入caustics map,否则就记入全局map;
其实，对于每一次相交，我们即可以记录入射光子，也可以记录出射光子。但是我们选择了记录入射光子。
我们记录镜面反射与规则折射的光子信息是没有意义的，因为我们不可能把所有的镜面反射和规则折射过程都记录下来，所以这一类亮度还是要通过逆向光线追踪或其它方法来计算。但是我们记录漫反射过程中的采样信息是有用的。因为我们可以通过某一点的部分入射采样光子来近似的模拟该点的全部入射光子。然后我们可以计算该点任意方向上的出射光子。这也决定了我们只能记录入射光子信息而不是出射光子信息。记录入射光子还可以让我们通过选择不同的brdf甚至不同的简化模型来重构每一次反射过程，这样我们就可以随心所欲的计算。
那么我们后面如何通过一点的入射光子来计算该点的出射光子呢，我们选择一个包围该点的范围很小的球空间，把这个空间里所有的入射光子按照半球积分模型计算，就可以算出该点的出射光子。
(3)如果光子打到的表面既有一定的镜面反射属性，又有一定的漫反射属性，则依据两种属性各自所占的百分比，使用俄罗斯轮盘赌原则来决定该次的反射属性。
(4)光子再决定了反射属性之后，还要依据反射属性再随机一次，以判定其是被表面吸收还是发射出去。

构造好光子贴图之后，我们就可以在第二步收集这些信息来计算顶点亮度。
我们首先来看一个对光能半球积分简化过了的公式：
L*f = (L(l)+L(c)+L(d))*(f(s)+f(d))
这个公式中，L表示入射光的集合，f表示该点的表面反射属性集合。
L(l)表示直接光照，L(c)表示纯粹的反射折射光，L(d)表示至少经历了一次漫反射的入射光
f(s)表示镜面反射或者规则透射brdf，f(d)表示漫反射brdf。

L*f的结果就是出射光的亮度，我们要做的就是如何快速的计算L*f。
我们把上面的等式分化一下：
L*f = L(l)*(f(s)+f(d))
+ f(s)*L(c) + f(s)*L(d)
+ f(d)*L(c)
+ f(d)*L(d)
如果直接采用半球积分方程进行计算，需要大量的采样，我们这种分化把半球积分分化为四部分，对不同的部分采用不同的办法计算，这样每一种都不会产生大量的采样，合起来的计算复杂度远远低于原来不分开计算的。

(1)直接光源照射，反射属性为所有。
(2)入射光源为镜面反射或者规则透射或者漫反射，反射属性为镜面反射或者规则透射。
(3)入射光源为纯反射或透射，反射属性为漫反射。
(4)入射光源为至少经过一次漫反射的，反射属性为漫反射。

对于(1)，我们采用shadow ray的方法计算直接光照。
对于(2)，我们采用经典Monte Carlo光线追踪来计算。
对于(3)，我们收集来自caustics map中的光子信息。
对于(4)，我们收集来自全局map中的光子信息。

这样，一次典型的正向光线追踪的计算就完成了。即使是photon map算法，对于普通硬件，暂时也只能用于静态渲染。但是我们依然可以把它用在游戏中，比如在地图编辑器中对静态光源和大型静态场景进行预渲染，如果光源是变化的，那么对光源变化的过程采样，渲染后在通过插值计算来模拟光源变化。通过基于光线追踪计算出的图像，具有很高的光真实感，可以令用户产生赏心悦目的感受。

[photonmap实时渲染方案的想法]
1 区别于静态渲染，不是一次发射所有必须的光子，而是只产生少量的光子，把相关信息保存在光子图中，然后每桢逐步递加光子，过了一定时间以后，就抛弃旧的光子信息。
2 构造类似于windows脏矩形思想的脏光线算法。

.

fire 2006-01-19 17:31 发表评论

注释(0)

关于脏光线的一些想法

2006年1月19日，17:30:00 | fire

要想计算实时的光线追踪,除了改进模型以外,一个很重要并且通用的计算思想应该被重视:只计算变化过了的光能传递过程.
以photonmap为例,类似于windows的脏矩形算法,我们应该想办法标记出由于光源或者场景的改变而引起了哪些传递过程的改变.我们可以管它叫做脏光线算法(我随便起的名字,不知道是不是有人已经实现了类似的想法).
一个简单的想法是:给初始的每个光子标记它的光源来源.在一次相交中,入射光子叫做父光子,而新产生的光子被叫做子光子,子光子的光源属性等于它所有的父光子光源的和集.当某一个光源产生变化时,我们先从photonmap中除去所有标记有相关光源属性的光子,保留剩下的.然后每一桢我们再逐量追加采样,结合传统的每桢删除旧光子,增加新光子的方法,应该会有更好的效果.假想一下,当光源被分割到多个独立的空间的时候,而每个空间又有可能被玩家看到的话,这个想法应该会有效.

2005.12.13

fire 2006-01-19 17:30 发表评论

注释(0)