基于GPU加速的辐射度光照算法的研究及应用

于 平
（河北省气象技术装备中心 石家庄 050000）

摘 要：辐射度光照算法作为一种广泛应用的全局光照模型，直接影响了闭合场景的光照渲染效率和真实感程度。传统辐射度算法求解复杂、计算量大，改进的逐步求精辐射度算法可以即时地计算面片形状因子，但是对于面片数目较多的复杂场景，求解仍然非常耗时。为此提出了一种利用图形处理器（GPU）来实现逐步求精辐射度光照的算法，该算法充分利用GPU的并行计算能力和可编程性，将算法中面片可见性的判断和形状因子的计算完全在GPU中完成。采用GLSL着色器语言编制特定的着色器程序，片段着色器程序实现形状因子的计算和每个面片辐射度的求解。通过应用程序创建自己的帧缓冲区对象，利用离屏渲染、纹理缓存等技术来进行渲染和中间结果存储。改进后的算法对于简单场景的光照渲染可以达到交互的速度，对于复杂场景会大大提高其计算速度，减少光照渲染时间，改进其光照渲染的效果。

关键词：逐步求精辐射度；GPU；形状因子；离屏渲染
中图分类号：TP391.41 **文献标识码**：A **国家标准学科分类代码**：510.4050

Research and application on radiosity illumination based on GPU hardware acceleration

Yu Ping
（Hebei Province Meteorological Technical Equipment Center, Shijiazhuang 050000, China）

Abstract: Radiosity is a widely used technique for global illumination, which makes a direct impact on the illumination rendering efficiency and realism in the indoor scene. The traditional radiosity requires huge computation. Although the improved progressive refinement radiosity can instantly calculate the form factor, the whole process is still very time-consuming when the number of patches is large. For the above reasons, the progressive refinement radiosity based on GPU hardware acceleration is proposed, which takes advantage of the parallel computing capabilities of GPU. The judgment of patch visibility and the calculation of form factor are completely achieved in the GPU . The calculation of form factor and the solution of patch radiosity are accomplished in the fragment shader. The application create the frame buffer object, which can carry out the off screen rendering and render to texture. This improved method can reach the speed of interaction for a simple scene, and it will greatly enhance the calculation speed and improve the illumination rendering effect for a complex scene.

Keywords: progressive refinement radiosity; GPU; form factor; off screen rendering

1 引 言

现阶段生成真实感图像最重要的光照模型是由Whitted[1]在1980年提出的光线跟踪算法和美国Cornell大学与日本广岛大学的学者在1984年提出的辐射度算法[2]。

光线跟踪算法跟踪从视点发向场景的光线，求出光线与场景中可见物体的交点。尽管可以成功地模拟折射、反射、阴影等光照效果，但光线跟踪算法很难模拟景物的多重漫反射效果，并且其消耗的计算量非常大。

辐射度光照算法来源于热辐射工程，它的理论基础是能量守恒定律，算法通过求解方程来计算所有景物表面的辐射能量值。与光线跟踪算法不同的是，当给定光源等属性后，整个场景中所有景物表面的辐射度能量值的分布已经确定，并且不会随着视点的变化而变化，所以说辐射度是一个与视点无关的算法，这也是它被普遍使用在真实感渲染方法中的一个主要原因。

传统辐射度光照算法中形状因子和辐射度方程的求

收稿日期：2016-08

解需要大量运算，渲染一个复杂场景会花费很长时间，这也给算法带来了一定的应用限制。为了解决这一问题，Cohen 等人研究出了半立方体形状因子法[3] 和逐步求精[4]的辐射度算法。半立方体法通过直接利用图形硬件的 Z-Buffer 结合预计算的查找表来计算形状因子，很大程度上加速了形状因子的计算过程；逐步求精辐射度算法通过面片对场景的辐射度贡献，把面片排序，按顺序来处理以逐步求解线性方程组，加快求解的收敛速度。在此基础上，Hanrahan 等人采用层次结构技术成功的将辐射度方法的计算降低成线性复杂度[5]，这些研究为辐射段算法的广泛应用打下了坚实的基础。尽管不断有辐射度改进方法的提出，但辐射度的计算过程仍然非常耗时。

随着可编程图形硬件的发展，尤其在可编程硬件高级语言（Cg、GLSL 等）出现后，就可以充分利用可编程图形处理单元 GPU 的并行计算能力[6] 来加速辐射度的计算。本文着重研究了如何利用 GPU 来实现逐步求精辐射度光照算法。首先，讨论了关于 GPU 运算中的一些基本理论，介绍了 GPU 可编程着色器的创建流程以及如何替换 OpenGL 固定管线操作。然后，提出了利用 GPU 来求解形状因子和面片辐射度的改进算法，通过在应用程序中创建自己的帧缓冲区对象，并为其附加链接相应的纹理缓冲区对象和渲染缓冲区对象，通过两遍离屏渲染操作来实现辐射度的一次求解。

2 GPU 计算简介

为了利用 GPU 进行图形计算，必须对它的结构有一定的了解，把 GPU 应用于通用计算问题的最大困难是它们的设计非常专用化。结果导致 GPU 编程在计算机图形 API 和编程语言之间上建立了一道壕沟，在开始对 GPU 编程前，首先要想想 GPU 擅长做什么。很明显它们擅长于计算机图形计算，计算机图形计算的两个主要属性是数据并行性和独立性，不仅是把相同或相似的计算应用到很多顶点和片段组成的流，而且在每个元素上的计算只有一点点（或完全没有）依赖于其他元素。为了把通用的 CPU 上的计算映射到 GPU 的专用硬件上，必须要熟悉 GPU 的图形流水线、着色器的创建流程以及 GPU 的存储器类型。

2.1 GPU 的图形流水线

图形硬件流水线[7] 以流水的方式处理大量的顶点、几何图元和片段。当今图形硬件设计上最明显的的趋势是在图形处理器（GPU）内部提供更多的可编程性，如图 1 所示为展示了一个可编程图形处理器渲染管道[8-9] 的图形流水线，其中的阴影部分则为 GPU 的两种可编程处理器：顶点处理器和片段处理器[10]。二者可以运行相应的顶点着色器程序（vertex shader）和片段着色器程序（fragment shader）。

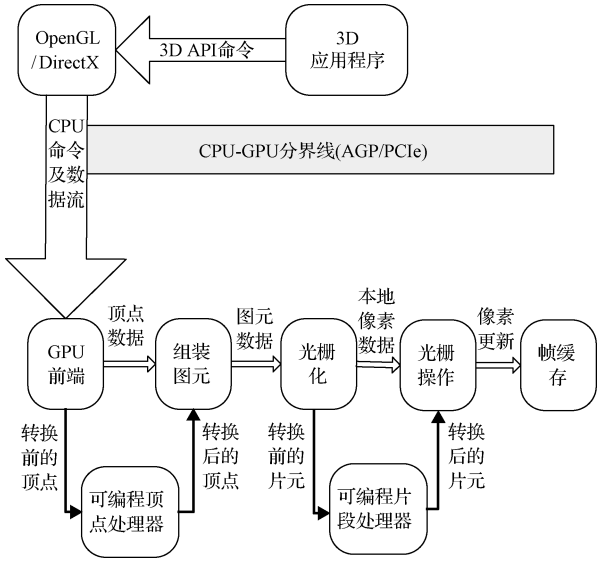

图 1 GPU 的图形流水线

顶点处理器处理顶点流，对传入的顶点值进行一系列基本变换。顶点处理器输出变换后的值经固定管线进行图元装配、裁剪、光栅化插值等操作得到一系列的片段，传递给片段处理器。片段处理器将一个片段程序应用到流中的每个片段上，片段处理器有能力获取来自纹理的数据，并且可以对其操作，片段处理器主要用来计算每个像素的最终颜色。

2.2 着色器的创建流程

本文采用 OpenGL 着色器语言 GLSL 来创建可编程的着色器，语言版本为 1.30 版对应于 OpenGL3.0 版本，并且支持 GL_ARB_compatibility 扩展。想要用可编程着色器来代替固定管线功能[11]，必须在应用程序中按照顺序执行以下步骤：

1）调用 glCreateShaderObjectARB 创建一个或多个空的重色器对象；

2）调用 glShaderSourceARB，为这些着色器提供源代码；

3）调用 glCompileShaderARB 编译各个着色器对象；

4）调用 glCreateProgramObjectARB 创建着色器程序对象；

5）调用 glAttachObjectARB 将所有着色器对象附加到程序对象中；

6）调用 glLinkProgramARB 链接程序对象；

7）调用 glUseProgramObjectARB，将可执行着色器程序作为 OpenGL 当前状态的一部分安装，替换固定管线；

8）如果着色器程序中有 uniform 变量（一致变量），那么应该在调用 glLinkProgramARB 之后调用 glGetUniformLocationARB 来查询 uniform 变量的位置，然后再通过调用 glUniformARB 来设置着色器中 uniform 变量的值；

9)如果顶点着色器会使用用户定义的属性变量,那么应用程序在链接之前通过调用 glBindAttribLocationARB 来赋值。

着色器程序创建的简单流程如图 2 所示。

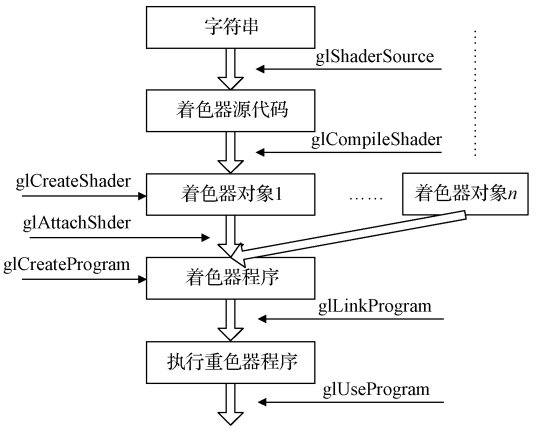

图 2 着色器的创建流程

2.3 GPU 存储器类型

相对于串行微处理器的主存储器、cache 和寄存器,图形处理器(GPU)有属于自己的存储器体系结构,但是这个存储器体系结构是针对加速图形操作设计的,主要用于流式编程模型,而不是通用、串行的计算。GPU 存储器只能通过抽象的图形编程接口来访问,每个这样的抽象可以想象成不同的流类型[12],GPU 程序员可以看到的 3 种流类型是顶点流、帧缓冲区流和纹理流。第 4 种流是片段流,在 GPU 中产生并完全消耗,程序员是看不到的。GPU 采用流式并行计算模式[13],可以对每个数据进行独立的并行计算,所谓"对数据进行并行独立计算"即流内的任意元素的计算不会依赖于其他同类型的数据。而所谓"并行计算"[14]是指多个数据可以同时被使用,多个数据并行运算的时间与一个数据单独执行的时间是相同的。如图 3 所示为一个现代 GPU 的流水线,3 个用户可以访问的流以及在流水线中它们可以被用到的位置。

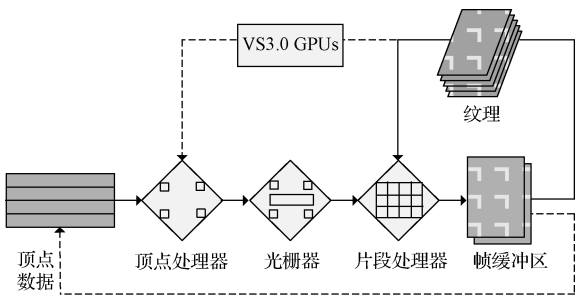

图 3 现代 GPU 中的流

1)顶点流

顶点流是通过图形 API 的顶点缓冲区指定,这些流容纳了顶点位置和多种顶点属性如:纹理坐标、颜色、法线等,它们可以用作顶点程序的任意输入流数据。直到最近,顶点流的更新还只能通过把数据从 CPU 传到 GPU 中来完成。GPU 不允许写入顶点流。然而最近的 API 增强已经使 GPU 可能写入顶点流,通过"复制到顶点缓冲区"或"渲染到顶点缓冲区"[15]来完成。

2)片段流

片段流是由光栅器产生的,并被片段处理器所消耗。它们是片段程序的输入流,但是由于它们完全是在图形处理器内部建立和消耗的,所以它们对程序员来说是不可直接访问的。片段流的值包括来自顶点处理器的所有插值输出:位置、颜色、纹理坐标等。片段程序不能随机访问片段流,允许对片段流随机访问会在片段流元素之间产生依赖,因此会破坏编程模型的数据并行保证。如果算法确实要求对片段流随机访问,那么必须将这个流先保存到存储器中,并转换为一个纹理流。

3)帧缓冲区流

帧缓冲区的流由片段着色器写入,它们传统上被用作容纳要显示到屏幕上的像素。然而流式 GPU 计算用帧缓冲区来容纳中间计算阶段的结果。除此之外,现代 GPU 可以同时写入多个帧缓冲区表面。片段和顶点程序都不能随机访问帧缓冲区的流,然而 CPU 通过图形 API 可以直接读取它们[16]。

4)纹理流

纹理是唯一一种可以被片段程序和 Vertex Shader 3.0 GPU 顶点程序随机访问的 GPU 存储器。如果程序员需要任意地索引入一个顶点、片段或帧缓冲区流,它们必须先转换成一个纹理。纹理可以被 CPU 或 GPU 读取和写入。GPU 通过直接渲染到纹理或者把数据从帧缓冲区复制到纹理寄存器来写入纹理[17]。

3 基于 GPU 的逐步求精辐射度算法

3.1 逐步求精辐射度算法简介

辐射度算法是全局光照问题的有限元方法,它把场景分割成许多小面片,计算有多少能量在这些面片之间进行传递。每对面片之间的能量传递用一个名为形状因子的方程描述,下面是一个形状因子方程的简化版:

$$F_{ij}=\frac{\cos\theta_i\cos\theta_j}{\pi r^2}V(i,j)$$

式中:θ_i,θ_j 为两个面片法线与二者连线的夹角,r 为二者之间的距离。方程由两部分组成:几何体项和可见性项。几何体项表明了面片 i 和 j 之间传递的能量是它们之间距离和相对方位的函数。如果面片之间有遮挡,可见性项 $V(i,j)$ 的值为 0;若相互完全可见,则值为 1。

传统辐射度算法构造和求解了大量含有成对形状因子的线性方程组。这些方程描述了面片的辐射度是其他面片传给该面片总能量的函数,权重是面片间形状因子和该面片的反射率。因此传统的线性方程组需要 $o(N^2)$ 的存储空间(N 为场景中面片数),而这是大场景所不允许的。

逐步求精算法即时的计算了形状因子，避免了大量的存储要求。在逐步求精算法中，场景中每个面片保持了两个能量值：累积能量和残余能量，累积能量用在辐射度计算过程中；残余能量又可以叫未发射能量，是每个面片当前具有的辐射度能量。选择场景中一个面片作为“发射者”，测试从这个发射者到其他所有面片的可见性。如果一个接收面片是可见，那么从发射者传递给接收面片能量。这个能量乘上接收面片的反射系数得到其接收到的实际能量，这个结果既加到接收面片的累积能量上，也加到其残余能量上。现在认为发射者已经过度发射了其能量，故将其残余能量值清零，接着选择下一个发射者。这个过程不断重复，直到选取的发射者的能量值降低到一个阈值，辐射度求解结束。

在GPU中实现逐步求精辐射度算法，首先要从发射者的视角来渲染场景，然后把能量散射到所有其他可见多边形的纹理中，这是最初始的方法，其性能很差，这个方法的问题是需要GPU写入多个任意纹理的任意位置，这种散布操作是当前的图形硬件很难做到的[18]。反之，可以对接收多边形进行迭代运算，并测试每个片段的可见性，这就利用了GPU数据并行的能力，对许多片段并行执行相同的一小段片段程序。该方法在快速收敛的情况下既保证了接收多边形辐射度的高质量重建，又保证了发射辐射度的低存储量要求[19]。

3.2 改进的逐步求精辐射度算法实现

基于GPU的逐步求精辐射度算法的伪代码如下所示：

初始化发射者的残余能量 E；

while（没有收敛时）

{

从发射面片的视角渲染场景，把每个多边形的ID作为颜色进行渲染，结果存储在创建的帧缓冲区的纹理中，同时把多边形的光照贴图中每个像素的坐标同样存储在纹理中；

再次进行渲染，读取第一次渲染得到的纹理中的ID值，判断可见性；

if（元素可见）

{

计算形状因子 FF；

计算接收到的能量值 $\Delta E=\rho \cdot FF \cdot E$；

然后把计算的能量值渲染到帧缓冲区的渲染缓冲区对象中存储；

}

在应用程序中根据第一次渲染得到的纹理中的每个像素的坐标值和第二次渲染得到的渲染缓冲区中的辐射度值进行结果读取，来更新每个多边形的光照贴图；

发射者的光照贴图能量值清零；

再次计算具有最大能量值的多边形即为下一个发射者；

}

这种辐射度解法的每次发射能量都需要两次渲染计算：可见遍和重建遍。

3.2.1 可见遍计算

可见遍是从发射者的视角来离线渲染场景，把场景中每个多边形的ID作为颜色来进行渲染，利用可编程顶点重色器来把顶点投影到一个半球体上，最终把结果渲染到纹理中存储；同时把每个多边形光照贴图中的像素坐标写入纹理中存储。这样该纹理中的每个单元存储的就是对当前发射者可见的多边形的ID和像素的坐标。具体步骤如下：

1）在应用程序中创建一个空的纹理texture，用来存储接下来执行离屏渲染的结果。

2）需要在应用程序中创建自己的帧缓冲区对象[20]（frame buffer objet）而不是用窗口系统所提供的默认帧缓冲区，默认帧缓冲区是图形服务器的显示系统可用的唯一帧缓冲区，它是自己可以在屏幕上看到的唯一帧缓冲区。而应用程序自己创建的帧缓冲区对象可以实现离屏渲染，即把当前一次的渲染结果存储到提前创建的纹理texture中作为临时变量存储，是不能在显示器上显示的。具体代码如下所示，该代码是应用程序中创建自己的帧缓冲区对象并为其附加生成纹理缓冲区的主要伪代码。

```
//创建空的纹理 texture
glGenTextures(1,&texture);
glBindTexture(GL_TEXTURE_2D,texture);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA8,TexWidth,TexHeight,0, GL_RGBA, GL_UNSIGNED_BYTE,NULL);
//创建自己的帧缓冲区对象 FrameBuffer
glGenFramebuffers(1,&framebuffer);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, framebuffer);
//为创建的帧缓冲区对象附加纹理缓冲区
glFramebufferTexture2D (GL_DRAW_FRAMEBUFFER,GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texture,0);
/* 接下来设置一下投影矩阵、模型视图矩阵，在顶点着色器中的一致变量会用到。再用着色器程序代替固定管线功能，进行渲染操作 */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(CRender::s_camangle,CRender::s_aspect,CRender::s_nearplane,CRender::s_farplane);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(-cam->pos.x,-cam->pos.y,-cam->pos.z);
//用多边形的ID作为颜色进行场景渲染
```

DrawScene ();

3)在顶点着色器中创建执行半球投影的着色器程序,并在应用程序中对着色器中用到的一致变量进行初始化赋值,用它代替固定管线上顶点处理操作。

具体代码如下所示,该代码为顶点着色器中半球投影的代码,用 GLSL 着色器语言编写。

```
uniform vec2 NearFar; //近和远平面
varying float4 ProjPos;//半球投影后的位置
void hemiproj( )
{
  //几何体变换到眼睛空间
vec4 mpos=gl_ModelViewMatrix * gl_Vertex;
//把点投影到单位半球体上的一个点
vec 3 hemi_pt=normalize(mpos. xyz);
//计算 F-N,但让硬件来除放在 w 分量中的 z
float f_minus_n=NearFar. y-NearFar. x;
ProjPos. xy=hemi_pt. xy * f_minus_n;
//独立计算投影深度,使用 OpenGL 的正交功能
ProjPos. z=(-2 * mpos. z- NearFar. y-NearFar. x);
ProjPos. w= f_minus_n;
}
```

3.2.2 重建遍计算

重建遍把每个潜在的接收多边形用正交投影绘制到一个和多边形纹理分辨率大小一样的帧缓存中。这样在辐射度纹理的单元和帧缓存的每个片段建立一一对应的关系。此处的帧缓存对象也是由应用程序自己创建的,并且为其附加渲染缓冲区对象,通过离屏渲染操作,用渲染缓冲区对象来存储辐射度计算过程中每个片段的辐射度光照值。再在应用程序中从创建的帧缓冲区中读取存储计算结果的图像,替换现有的纹理图像。具体步骤如下:

1)在应用程序中创建一个帧缓冲区对象(frame buffer objet),帧缓冲区对象本身是没有任何存储空间的,必须为其附加连接纹理对象或者渲染缓冲区对象,不同于第一遍渲染,此处是为其附加连接渲染缓冲区对象(render buffer object),渲染缓冲区对象是一个数据存储区,包含一副图像和内部渲染格式。此处存储的是片段着色器计算完每个片段得到的辐射度光照值。

2)在应用程序中为创建的渲染缓冲区对象分配存储空间并指定图像格式,其中存储空间的大小与第一遍渲染得到的纹理大小一样。使得光照贴图纹理中的像素与帧缓存的每个片段建立一一对应的关系,这样就可以把每个多边形计算得到的辐射度光照值从帧缓存中重新复制到多变形的纹理中,为下一次求解准备。一旦为渲染缓冲区创建了存储空间后,就可以把其附加到创建的帧缓冲区对象上,然后就可以向其进行渲染。

3)编写着色器程序替换 OpenGL 固定管线操作,对每个多边形再次进行渲染,在片段着色器中对光栅化后的每个片段进行可见性判断,通过在片段着色器中读取第一遍渲染得到的帧缓冲区的纹理对象,来判断当前片段是否在该纹理中,若在则对发射者可见,计算形状因子值,进而求出该片段接受到的光照值,并把结果存储到渲染缓冲区中。具体如代码 1 和代码 2 所示所示,代码 1 为片段着色器实现片段可见性判断的着色器代码。代码 2 为片段着色器实现形状因子计算与辐射度能量值计算的着色器代码。

代码 1:片段可见性测试代码

```
//元素的相机空间位置
varying float4 ProjPos;
//接收者的 ID,在应用程序为一致变量赋值
uniform vec3 RecvID ;
//第一遍渲染的结果存储到纹理对象中,在应用程序为该一致变量赋值
uniform sampler2D HemiBuffer;
bool Visible( )
{
//把元素投影到半球上
  vec3 proj=normalize(ProjPos);
  //向量从[-1,1]放缩到[0,1]进行纹理查找
  proj. xy= proj. xy * 0. 5+0. 5;
  //获取半球项缓冲区中投影后的值
  vec3 xtex=tex2D(HemiItemBuffer, proj. xy);
  //比较半球项缓冲区的值和片段的 ID 是否相等
  return all(xtex==RecvID);
}
```

代码 2:形状因子和辐射度计算的片段程序代码

```
//接收元素在世界空间的位置、法线由插值得到
//接收元素反射率设为固定值,应用程序赋值
varying vec3 RecvPos;
varying vec3 RecvNormal;
uniform vec3 RecvColor;
//发射元素在世界空间的位置、法线
uniform vec3 ShootPos;
uniform vec3 ShootNormal;
//发射元素残余能量
uniform vec3 ShootEnergy;
vec3 FormFactorEnergy( )
{
  //从发射者到接收者的向量
  vec3 r= ShootPos-RecvPos;
  //发射者到接收者的距离平方
  float distance=dot(r,r);
  r=normalize(r);
  //接收者、发射者法线与 r 夹角
  float cosi= dot(RecvNormal,r);
  float cosj= -dot(ShootNormal,r);
```

```
    //计算形状因子
    const float pi=3.141 592 6;
    float Fij=max(cosi·cosj,0)/(pi·distance);
    Fij·=Visible( );//返回可见性0或者1
    //计算接收者获得的能量
    vec3 delta= ShootEnergy· RecvColor· Fij;
    Return delta;
}
void main( )
{
    //计算得到的辐射度光照值写入片段的颜色,存储到创建的渲染缓冲区对象
    gl_FragColor= FormFactorEnergy( );
}
```

4)在应用程序中根据第一次渲染得到的纹理对象中存储的像素坐标数据和第二次得到的渲染缓冲区对象中的每个像素的辐射度值,从帧缓冲区中进行读取,并把结果更新到每个多边形自身的光照贴图中。然后对于发射者的光照贴图纹理值清零,这样就得到一次逐步求精辐射度算法的解。

5)在应用程序中计算每个多边形对应的纹理的总能量值,找出具有最大辐射度能量值的纹理对应的多边形ID,作为下一个发射者,重复上面可见遍和重建遍渲染过程,直到最大能量值低于某个阈值时,停止求解。把每个多边形对应的光照贴图进行纹理打包成大的光照贴图纹理[21],这样就可以把整个场景渲染到窗口系统提供的帧缓冲区中用于显示。具体代码如下所示,该代码为应用程序创建帧缓冲区对象、渲染缓冲区对象,并从帧缓冲区中读取渲染结果的主要伪代码。

```
//创建渲染缓冲区对象,并对其分配存储空间
glGenRenderbuffers(1,&color);
glBindRenderbuffer(GL_RENDERBUFFER,
color);
glRenderbufferStorage(GL_RENDERBUFFER,
GL_RGBA, TexWidth,TexHeight);
//创建帧缓冲区对象
glGenFramebuffers(1,&framebuffer);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER,
framebuffer);
//把渲染缓冲区对象附加到帧缓冲区对象中
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER,GL_COLOR_ATTACHMENT0,GL_RENDERBUFFER,color);
/*接下来用可编程着色器程序代替固定管线操作,对每个潜在多边形进行渲染,并把渲染以后的结果从创建的帧缓冲区中复制加到多边形的纹理中*/
//纹理环境函数设置为纹理值与被处理的片段的颜色值进行累加
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_ADD);
//渲染场景中的每个多边形
DrawSene();
//把每个多边形离屏渲染到帧缓冲区的值复制到原来的纹理texName中
glBindTexture(GL_TEXTURE_2D,texName);
glCopyTexSubImage2D(GL_TEXTURE_2D,0,0,0,x,y, texWidth,texHeight);
/*全部多边形渲染完后,再找出下一个具有最大辐射度能量值的多边形作为新的发射者*/
int Find_Max_Energy_lm(float energy)
{
  int i,j=-1;
  vector v;
  float f;
//遍历所有多边形的纹理贴图
  for( i=0;i<lm.num;i++ )
  {
    //每个多边形对应的当前总能量值
v=lm[i])->totalenergy;
f=Intensity(v);  //辐射度能量的强度
    if (f>energy)
    {
  energy=f;   //更新找到的最大能量值
  j=i;        //记录多边形的ID号
    }
  }
  return j;          //返回找到的多边形ID
}
```

4 实验结果及分析

通过在个人计算机上使用图形硬件接口OpenGL和高级着色语言GLSL,来实现基于GPU硬件加速的逐步求精辐射度光照算法,得到了一些验证算法有效性的实验结果。实验以虚拟室内环境为场景,计算机配置如下:

硬件环境:处理器:Intel(R) Core(TM) i3-41760CPU 3.60 GHz;硬盘:500 G;内存:4.00 G;显卡:NVIDIA GeForce GT705,显存1 G。

软件环境:系统:Windows 7 旗舰版 32位操作系统;开发工具:Visual Studio 2005。

场景一组成如下:1 602个面,6 431个顶点,2个光源,14 445个面片,是简单场景。实验结果如图4所示,采用基于GPU硬件加速的简单场景辐射度光照效果。

场景二组成如下:8 549个面,35 785个顶点,6个光源,86 077个小面片组成,是比较复杂的场景,如图5所示为采用基于GPU硬件加速的复杂场景辐射度光照效果。

由上面两幅图可以看出基于GPU实现的逐步求精辐

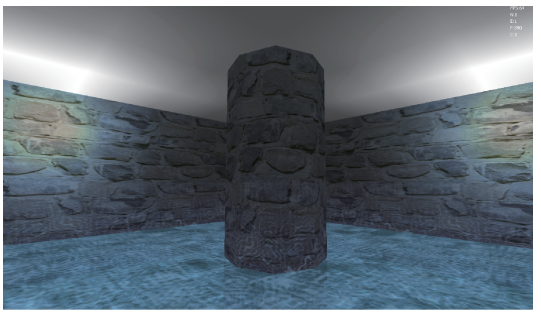

图4 GPU硬件加速的简单场景辐射度光照效果

图5 GPU硬件加速的复杂场景辐射度光照效果

射度光照算法能够很好地模拟真实的室内场景中的光照效果，如模糊阴影和间接光照等。

如表1所示为对于简单场景、复杂场景的逐步求精辐射度算法在CPU和GPU中的渲染时间比较结果。

表1 渲染时间比较表

场景	场景中面片数	场景中光源数	CPU中渲染时间/s	GPU中渲染时间/s
简单场景	14 445	2	4.0	2.0
复杂场景	86 077	6	139.0	79.0

由表1可知，通过在GPU中来实现逐步求精辐射度算法，利用GPU本身计算能力的优势，可以很大程度上加快辐射度方程的求解，使光照渲染的时间进一步减少，并且得到了较好的光照效果。

5 结 论

本文主要研究了虚拟室内场景中辐射度光照算法的求解技术，对比研究了光线跟踪算法和辐射度算法，从闭合的虚拟室内场景出发，着重研究了逐步求精辐射度光照算法。针对逐步求精辐射度光照算法的形状因子计算和辐射度求解的问题进行了分析与改进，采用可编程图形硬件GPU来进一步改进该算法，改进后的算法保留了逐步求精算法本身的收敛优势，又利用了GPU处理器强大的并行计算能力，渲染速度得到很大提升，光照效果也得到了很大改进，该算法在时间和效果上达到了一个很好的平衡。

本文讨论的辐射度算法是标准辐射度算法即只考虑了漫反射之间的光能传递，对镜面反射不能很好的解决，下一步的工作方向是在辐射度算法中加入光线跟踪原理以期获得一个更加完善的整体光照模型。

参 考 文 献

[1] WHITTED T. An improved illumination model for shaded display[C]. ACM SIGGRAPH 2005 Courses. ACM,2005: 4.

[2] 孙家广，胡事民. 计算机图形学基础教程[M]. 北京：清华大学出版社，2005:4-5.

[3] COHEN M F, GREENBERG D P. The hemi-cube: A radiosity solution for complex environments[J]. Computer Graphics,1985,19(3):31-40.

[4] COHEN M F, CHEN S E, WALLACE J R. A progressive refinement approach to fast radiosity image generation[J]. Computer Graphics, 1988, 22(4): 75-84.

[5] REN P, WANG J, GONG M, et al. Global illumination with radiance regression functions[J]. ACM Transactions on Graphics (TOG), 2013, 32(4): 130.

[6] 吴铮，张磊，李宁. 基于GPU的机载高分SAR运动补偿和自聚焦[J]. 国外电子测量技术，2015，34(8)：94-99.

[7] 姚旺，胡欣，刘飞，等. 基于GPU的高性能并行计算技术[J]. 计算机测量与控制，2014，22(12)：4160-4162.

[8] GRAY K. Microsoft DirectX 9 programmable graphics pipeline[M]. Microsoft Pr,2003.

[9] 陈娇，余晓锷. 基于GPU的体绘制框架[J]. 计算机应用与软件，2013，30(9)：283-286.

[10] 田泽，张淑，张骏，等. 图形处理器片段处理单元的设计与实现[J]. 计算机应用，2014(增刊2)：357-360.

[11] 陈继选，王毅刚. 基于OSG的GLSL着色器编辑环境的[J]. 计算机系统应用，2011，20(3)：153-156

[12] 王化雨，刘惠义，冯艳蓉，等. 基于GPU的复杂三角网格模型多分辨率绘制[J]. 电子测量技术，2016，39(1)：35-39.

[13] BUCK I, FOLEY T, HORN D, et al. Brook for GPUs: stream computing on graphics hardware[C]. ACM Transactions on Graphics (TOG), 2004, 23(3): 777-786.

[14] 鲁强，周新. 基于在线检测动态一维下料问题的GPU并行蚁群算法[J]. 仪器仪表学报，2015，36(8)：1774-1782.

（下转第57页）