空间跳跃加速的 GPU 光线投射算法

梁承志 高新波 邹 华 王向华
(西安电子科技大学电子工程学院,西安 710071)

摘 要 光线投射算法是一种应用广泛的体绘制基本算法,能产生高质量的图像,但是时间复杂度较高。实现了一种基于图形处理器的单步光线投射算法,并在此基础上提出了一种基于空间跳跃技术的光线投射算法,以实现加速。采用八叉树组织体数据,利用空间跳跃有效地剔除体数据中对重建图像无贡献的部分,降低了硬件的负载。一个片段程序即可完成光线方向的生成、光线投射、空体素跳跃和光线终止等。实验结果表明,该算法对于内部包含大量空体素的体数据重建能起到明显的加速作用。

关键词 图形处理器 空间跳跃 加速 光线投射
中图法分类号: TP391.41 **文献标识码**: A **文章编号**: 1006-8961(2009)08-1684-05

Accelerated GPU Ray-casting Algorithm Based on Space Leaping

LIANG Cheng-zhi, GAO Xin-bo, ZOU Hua, WANG Xiang-hua
(*School of Electronic Engineering, Xidian University, Xi'an* 710071)

Abstract Ray-casting is a widely used basic volume rendering algorithm. It can get high quality image but suffers from high computational complexity. A single-pass ray-casting algorithm is developed based on GPU(graphics processing unit), and on the basis of this algorithm an accelerated algorithm is proposed based on space leaping technique. Noncontributing region in the volume data coded with octree data structure can be eliminated by space leaping. Then the burden of GPU may be reduced effectively. Only one fragment program is needed to generate ray direction, cast ray, skip empty voxels and terminate ray, *et al.* The experimental results illustrate the algorithm can significantly accelerate the reconstruction of the volume data containing a lot of empty voxels.

Keywords GPU, space leaping, acceleration, ray-casting

1 引 言

近年来,随着图形处理器(GPU)由固定管线向可编程管线的发展,以及受 GPU 强大的浮点运算能力的驱动,许多基于 CPU 实现的光线投射体绘制算法被加以改进,以便能够在 GPU 上运行。同时,人们也提出了一些新的基于 GPU 的光线投射体绘制算法,其中 Kruger 提出了一种多步(Multi-pass)算法[1],该算法通过多次绘制包围盒的前后面(前后面均由顶点指定的顺序决定)得到光线方程后进行光线投射。Roettger 提出的算法则通过相机位置和绘制包围盒前面确定光线方程参数[2]。

现代显卡的架构非常适合光线投射这种以图像为序的体绘制,然而对于 512^3 规模的体数据在普通显卡上用该算法重建仍然难以达到实时,因此必须寻求更快的算法。

在基于 CPU 的光线投射体绘制中,为了提高绘制速度,人们提出了很多加速算法,如基于空间剖分技术的算法(BSP 算法、Kd-Tree 算法、Octree 算法等);基于像空间的自适应重采样算法;基于物空间的自适应调整采样频率的方法;利用光线间相关性,

基金项目: 国家自然科学基金项目(60771068);国家重点基础研究发展计划(973)项目(2006CB705700)
收稿日期:2009-03-05;**改回日期**:2009-05-11
第一作者简介:梁承志(1983~),男。西安电子科技大学信号与信息处理专业硕士研究生。主要从事科学计算可视化方面的研究。
E-mail:czliang2008@gmail.com

Yagel 等人提出的基于模板的光线投射算法[3];基于序列间相关性,Yagel 等人提出的 Coordinate-buffer(简称 C-buffer)方法[4]等。

基于统一渲染架构级别的显卡以及 Shader Model 4.0 API,首先对 Kruger 提出的多步实现的算法[1]进行了改进,实现了一种基于 GPU 的单步光线投射体绘制算法,降低了算法的复杂度。在此基础上综合基于 CPU 的光线投射体绘制加速算法,提出了一种基于 GPU 的空间跳跃(space leaping)加速算法。该算法利用八叉树组织体数据,得到体数据的统计信息,重建时依据当前转换函数剔除对重建图像无贡献的体数据块,一个片段程序即可实现光线方向的生成、光线投射、空体素的跳跃、光线终止(以及提前终止)的判断、颜色和阻光度的积累以及混合等操作,从而加速光线的前进,提高重建速度,而且不会影响重建图像的质量(空体素对重建图像无贡献)。实验结果表明,该算法能显著提高内部包含大量空体素的体数据的重建速度,对于 512^3 规模的体数据重建也能达到实时交互。

2　单步光线投射算法

光线投射算法的基本思想是从视平面(视口)每个像素发出一条光线,穿过体数据,基于最基本的光线吸收和发射模型,沿着光线方向对颜色和阻光度进行积累。

受 GPU 性能的限制,先前的基于 GPU 的光线投射算法只能多步实现。Kruger 实现的多步光线投射算法[1]包括:生成光线入射点;生成光线方向;光线投射;光线终止的判断。以上每步都要绘制包围盒一次,而且需要多个 2 维纹理保存中间变量,因此操作复杂,不便于实际应用。

基于现代 GPU 的强大平行处理能力,对算法进行了改进,改进后的算法主要由 3 部分组成:

(1) 光线终点的生成　利用 FBO(frame buffer object)渲染包围盒的后面(每个多边形都有前后两个面,定义顶点以逆时针方向指定的面为前面,另一个面为后面,3 维物体的前后面由组成该 3 维物体的多边形的面决定),得到光线终点坐标,结果如图 1(a)所示。包围盒的 3 维尺寸由 f_i 决定,$f_i = size_i \times space_i$,其中 $i \in \{x,y,z\}$,$size_i$ 为体数据沿 i 方向的尺寸,$space_i$ 为该方向体素间距,再将 f_i 归一化到 0-1;

(2) 光线方向的生成　渲染包围盒前面,得到光线入射点坐标。片段对应的光线终点坐标与入射点坐标相减即为视口上每一个像素的光线方向矢量。图 1(b)为包围盒的前面,没有渲染到纹理,而直接用于启动片段程序,生成每一个像素的光线方向,如图 1(c)所示。

(3) 光线投射　从光线起点开始沿光线方向按设定的步长采样,依据当前转换函数映射成颜色值和阻光度后并进行积累。

其中第 2 和第 3 部分在同一个片段程序中实现。这种生成光线方向的方法与投影方式无关,无论是透视投影还是平行投影,均可以通过渲染包围盒前后面得到视口每一个像素的光线方向以及光线起点坐标。

(a) 包围盒后面　　(b) 包围盒前面　　(c) 光线方向

图 1　包围盒渲染示例

Fig. 1　Rendering of bounding box

3 光线投射加速算法

当光线在体数据中穿行时有两种状态,一种是在空体素(透明体素)中穿行以搜索非空体素(不透明体素);另一种是在非空体素中进行颜色和阻光度的累积。由于在第 1 种状态下空体素对于重建图像没有任何贡献,因此尽可能快地跳过空体素将显著地加快算法的速度,同时对图像的质量不会带来任何影响,这是空间跳跃加速算法的基本思想。基于此提出的光线投射加速算法可分为 3 个阶段:(1)体数据子分;(2)纹理生成;(3)基于片段程序的空间跳跃。在算法中,使用八叉树将体数据均匀子分成若干子数据块,然后统计每块的最大最小数据值,再基于当前的转换函数生成纹理,之后就可以使用单步光线投射算法完成重建。

3.1 体数据子分

用八叉树组织 3 维体数据,逻辑结构如下:

假设要表示的形体 V 可以放在一个充分大的正方体 C 内,它的八叉树可以用以下的递归方法来定义:八叉树的每个节点与 C 的一个子立方体对应,树根与 C 本身相对应,如果 $V=C$,那么 V 的八叉树仅有树根;如果 $V\neq C$,则将 C 等分为 8 个子立方体,每个子立方体与树根的一个子节点相对应。只要某个子立方体不是完全空白或完全为 V 所占据,就要被 8 等分,如图 2(a)所示,从而对应的节点也就有了 8 个子节点。这样地递归判断、分割一直要进行到节点所对应的立方体或是完全空白,或是完全为 V 占据,或是其大小已是预先定义的体素大小,并且对它与 V 之交做一定的“舍入”,使体素或认为是空白的,或认为是 V 占据的。

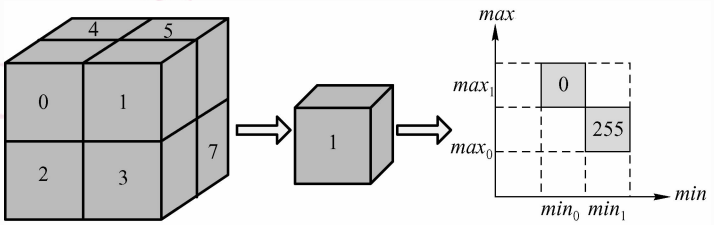
如此所生成的八叉树上的节点可分为 3 类:灰节点,它对应的立方体部分地为 V 所占据;白节点,它所对应的立方体中无 V 的内容;黑节点,它所对应的立方体全为 V 所占据。

后两类又称为叶子节点。形体 V 关于 C 的八叉树的逻辑结构是这样的:它是一棵树,其上的节点要么是叶节点,要么就是有 8 个子节点的灰节点。根节点与 C 相对应,其他节点与 C 的某个子立方体相对应。

实际的 3 维体数据不一定是规则的立方体,3 维尺寸可能是任意的,为此需要对八叉树进行改进以适应多种可能的情况,改进的八叉树结构如下:

根据体数据 3 维尺寸得到八叉树的 3 个方向的最大深度分别为 L_x, L_y, L_z, 其中 $2^{L_i} > size_i$ 且 $|2^{L_i} - size_i|$ 最小。则八叉树深度的取值区间为 $[0, L_i]$,其中 $i \in \{x,y,z\}$。对于八叉树某一级 l_i 节点包含的体素数量为 $d_i = \lceil size_i / 2^{l_i} \rceil$,其中 $\lceil \cdot \rceil$ 是向上取整,$size_i$ 为体数据沿方向的体素数量,d_i 用于确定该级节点在整个体数据中的位置。

用改进的八叉树结构对体数据进行子分,每个叶子节点对应于一个子数据块。在子分过程中每一子数据块都需要向外扩充一个体素,以避免在边界插值的时候出错(插值得到的值可能大于原来子数据块的最大值或小于原来子数据块的最小值),然后统计每一子数据块的最小值和最大值,并按原八叉树的结构存储,以用于生成 3 维纹理。



(a) 灰节点被8等分 (b) 叶子节点 (c) 2维纹理的生成

图 2 体数据子分及纹理生成示例

Fig. 2 Volume data subdivision and texture generation

3.2 纹理生成

依据体数据块的最大最小值生成 3 维纹理(tex_m),其中最大值(max)和最小值(min)分别存于纹理的 r 和 g 通道。所有叶子节点的最大值和最小值(均不重复)分别作为 2 维纹理(tex_empty)的两个坐标轴,按照当前转换函数,如果某一个纹素的

坐标(max,min)所在范围内的阻光度为 0(透明度为 1),则该纹素的值为 0,否则为 255,并存于 2 维纹理的 a 通道。当转换函数改变时,则重新生成并加载 2 维纹理。过程如图 2(c)所示。

3.3　基于片段程序的空间跳跃

生成纹理后就可以启动片段程序以完成重建。片段程序基于单步光线投射算法。在光线投射过程中,得到当前采样点的坐标后先采样 3 维纹理,即可得到当前采样点所属体数据块的最大值和最小值,用此最大最小值(max,min)作为 2 维纹理坐标,采样 2 维纹理得到标志位($flag$),如果为 0,则该节点所在块为空体素块,以大步长跳过。否则按小步长继续采样,并根据不同应用进行相应的处理。

整个光线投射以及空体素跳跃过程在同一个片段程序中实现,片段程序的伪代码如下所示:

```
//投射光线起始点与方向的生成
for 每个 fragment(i, j) do
  end[i, j]: = tex2D(texc, back);
  dir[i, j]: = end[i, j]-start[i, j];
  //光线投射
  if 当前采样点在体数据中 then
    min_ max: = tex3D(current_pos, tex_m);
    flag: = tex2D((min_ max), tex_empty);
    if(flag = 0) then
      跳过较大的步长;
    else
      颜色值与阻光度值的积累;
      按正常采样步长前进;
    end;
  end;
end;
```

$tex2D$ 和 $tex3D$ 分别是 2 维和 3 维纹理采样,$current\_pos$ 和 min_max 分别是 3 维和 2 维采样坐标。$texc$ 为当前 $fragment$ 对应的 2 维纹理坐标,$back$ 为保存光线终点的 2 维纹理,$end[i, j]$ 和 $start[i, j]$ 分别是光线终点和入射点,$dir[i, j]$ 为片段的光线方向。颜色与阻光度的积累在不同的应用有不同的实现。

4　实验与结果分析

为测试算法的加速效果选取了 6 组实验数据。测试平台为 Windows,内存为 2 G,显卡为 NVIDIA GeForce 8500GT,重建图像的分辨率(与视口分辨率相同)为 512 × 512,同一数据不同重建算法的转换函数相同。表 1 为几种重建算法的平均帧速率(绘制若干帧的平均速率,单位:帧/s)比较。其中,没有任何加速的光线投射算法为 RC,光线提前终止的光线投射算法为 RC-t,光线提前终止加上空间跳跃的光线投射算法为 RC-s,加速比 = (RC-s 平均帧速率 - RC-t 平均帧速率)/RC-t 平均帧速率。

表 1　3 种重建算法平均帧速率比较

Tab. 1　Average frame rate comparison of three reconstruction algorithm

单位:帧/s

数据	分辨率	RC	RC-t	RC-s	加速比(%)
头部	256 × 256 × 225	20.5	20.7	22.3	7.7
动脉瘤	512 × 512 × 512	18.1	18.1	30.6	69.1
引擎	256 × 256 × 256	20.0	20.6	23.5	14.1
脚部	256 × 256 × 256	20.1	20.6	23.6	14.6
血管造影 1	416 × 512 × 112	19.6	19.8	31.6	59.6
血管造影 2	512 × 512 × 79	21.1	21.8	27.3	25.2

从表 1 可以看出,加速算法对于内部包含大量空体素的体数据(如动脉瘤和血管造影)的重建能起到显著的加速效果。对于同一体数据,子分的数据块大小不同,其重建速度也有显著差别。子数据块大小对重建速度的影响如表 2 所示。

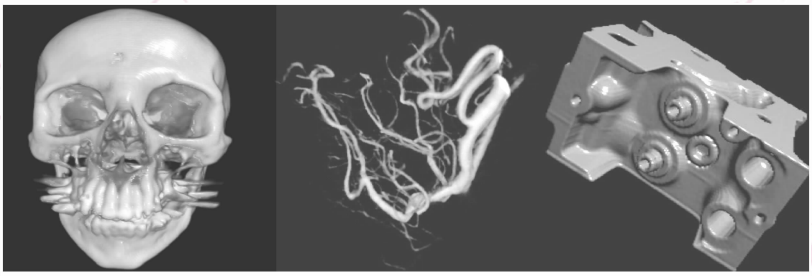
表 2　子数据块大小对平均帧速率的影响

Tab. 2　The impact on the average frame rate of volume data subdivision

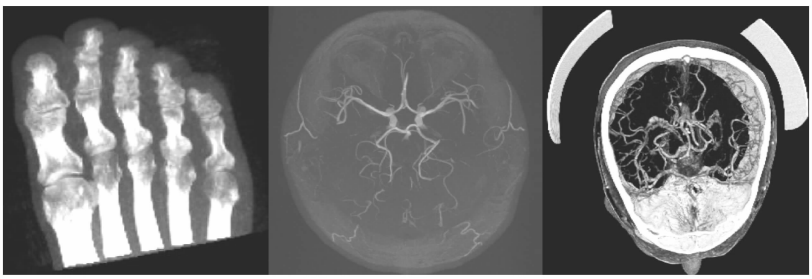
单位:帧/s

子数据块大小	动脉瘤	血管造影 1
128 × 128 × 128	12.2	12.5
64 × 64 × 64	20.7	15.3
32 × 32 × 32	30.6	20.4
16 × 16 × 16	30.6	31.6
8 × 8 × 8	30.6	20.4
4 × 4 × 4	23.5	18.7

从表 2 可以看出,子数据块大小选为 16 × 16 × 16 时,两种数据的平均重建速度达到最大。因此,在体数据子分时选择适当大小的子数据块就能实现明显的加速。而子数据块过大或过小时,重建性能将下降。子数据块大小的选择与 GPU 的物理参数有关,特别是流处理器的数量。6 组实验数据的重建图像如图 3 所示。



(a) 头部 (b) 动脉瘤 (c) 引擎



(d) 脚部 (e) 血管造影1 (f) 血管造影2

图 3 6 组数据的重建结果

Fig. 3 Reconstruction images of six volume data sets

5 结 论

通过分析先前的体绘制加速算法,结合单步光线投射算法,提出了一种基于片段程序的空间跳跃快速光线投射算法,通过剔除对重建图像无贡献的数据块实现加速。剔除的标准是当前设置的转换函数。实验结果表明,加速算法能显著加快内部包含大量空体素的体数据的重建,对于内部包含少量空体素的体数据重建也能起到一定的加速作用。

参考文献(References)

1 Kruger J, Westermann R. Acceleration techniques for GPU-based volume rendering [A]. In: Proceedings of IEEE Conference on Visualization [C], Seattle, Washington, USA, 2003: 287-292.

2 Roettger S, Guthe S, Weiskopf D, *et al*. Smart hardware-accelerated volume rendering [A]. In: Proceedings of Joint Eurographics/IEEE TCVG Visualization Symposium [C], Grenoble, France, 2003: 231-238.

3 Yagel R, Kaufman A. Template-based volume viewing [J]. Computer Graphics Forum, 1992, **11**(3): 153-157.

4 Yagel R, Shi Z. Accelerating volume animation by space-leaping [A]. In: Proceedings of IEEE Visualization'93 [C], Los Alamitos, CA, USA: IEEE Computer Society Press, 1993: 62-69.

5 Engel K, Hadwiger M, Kniss J M, *et al*. Real-time Volume Graphics [M]. Wellesley, Massachusetts, USA: Peter A K, Ltd, 2006, 45-65.

6 Stegmaier S, Strengert M, Klein T, *et al*. A simple and flexible volume rendering framework for graphics-hardware-based raycasting [A]. In: Proceedings of Volume Graphics'05 [C], Los Alamitos, CA, USA: IEEE Computer Society Press, 2005: 187-195.