

Ray Casting Algorithm Using CUDA

LI Ruo-yu¹, ZHANG Wen²

1. Institute of Command and Technology of Equipment, Beijing, China

2. Institute of Command and Technology of Equipment, Beijing, China

1.975172720@qq.com, 2. zhw139@139.com

Abstract: Volume rendering can visualize the inside information of data, but it's very time consuming. Using the programmable graphics processing units (GPU) as parallel stream processor can improve the performance, but this process was awkward because it used graphics metaphors and APIs to perform non-graphics computation. The newest programming environment/APIs-NVIDIA's CUDA allow developers to access the programmable units directly without the need to go through graphics APIs or special-purpose graphics hardware. We implement Ray Casting algorithm on GPU with CUDA and different rendering effects are achieved in real-time according to various blending functions of the sampling points.

Keywords: Ray Casting; GPU; CUDA; volume rendering

基于 CUDA 的 Ray Casting 算法研究

李若愚¹, 张 文²

1. 装备指挥技术学院, 北京, 中国, 101416

2. 装备指挥技术学院, 北京, 中国, 101416

1. 975172720@qq.com, 2. zhw139@139.com

【摘要】 体绘制算法高保真地显示三维数据, 借助视觉思维能力, 洞察其内部规律。采用传统 GPU 的流加速处理可提高体绘制速度, 但需要将计算部分映射成图形 API, 实现复杂。针对这一问题, 本文采用统一渲染架构 GPU 通过其通用计算 API (CUDA) 直接对 GPU 内部的流处理器编程, 改进了传统的体绘制算法。本文面向当代 GPU 体系结构, 研究多 GPU 流处理器上的并行光线投射算法, 实现各种绘制效果, 测试数据能够达到实时绘制速度。

【关键词】 光线投射算法; 图形处理器; 计算统一设备架构; 体绘制

1 引言

为了能够继续满足摩尔定律的计算加速需求, 硬件界引入了多核技术, 多核CPU适合开发任务并行性。与此同时, 通过游戏和图形市场需求的不断创新, GPU 已经超越了它们传统的图形处理功能, 逐渐演变成了流处理加速器, 具有足够强大的编程能力, 适用于数据并行性的通用计算结构。GPU一直采用多核(甚至超多核)设计。从提供计算速度的能力来看, GPU已远远超越了CPU, 而且GPU拥有更大的内存带宽。使用GPU来充当流处理加速器, 在高性能计算业界引发了相当的关注。目前, 截止到2008年底, NVIDIA公司已经推出了GT200系列GPU, 其旗舰产品GeForce280拥有240个流处理器, 计算峰值能够达到千亿次, 带宽已达成100GB/s。

体绘制是可视化复杂 3D 数据的技术, 近十几年来体绘制技术在算法研究方面取得了巨大进展, 先后提出了光线投射(Ray Casting)、足迹表法(Splatting)、体元投射法、错切一变形(Shear-Warp)、频域体绘制、以及基于纹理映射的体绘制等众多算法。然而体绘制算法的计算开销普遍较高, 很难在普通微机上实现实时地交互绘制, 对于海量数据的可视化处理通常要借助于并行技术。可视化计算是一种数据密集型计算、本身存在固有的并行性, 非常适合用 GPU 做并行流加速处理。因此, 采用 GPU 做流加速处理以提高计算速度, 已经成为当今的热点研究问题。

2 研究近况

J.Kruger 等人于 2003 年将传统的多遍纹理合成技术与可编程 GPU 绘制相结合, 将体数据以三维纹理的方式载入到 GPU 纹理内存中, 并集成空域跳过

(Empty Space Skipping)与早期光线终止(Early Ray Termination)等加速绘制算法,提出了一个实时而高质量的 GPU 体绘制算法^[1],从此拉开了 GPU 实现体绘制算法的序幕。随着 GPU 硬件性能的不不断提升,其后有多篇文献对其进行改进,实现了更为灵活的基于 GPU 的体绘制算法。2005 年,Stegmaier 利用 GeForce6 之后的新显卡提供的动态分支循环能力,将多遍绘制算法改进成单遍绘制即可得出最终的结果图像^[2],同时提升了算法的灵活性,可以在 GPU Ray Casting 中运用更为复杂的光学模型,该算法采用汇编语言实现。同年,Strengert 等人在单遍绘制算法的基础上,提出一种混合图像/物体空间的空域跳过加速算法,极大的提升了绘制速度^[3]。在 2006 年,作者 Ruijters 详细分析了 GPU 体绘制中各种执行瓶颈的原因所在,为 GPU 体绘制算法的优化给出了重要参考,同时也提出了有价值的优化算法^[4]。

用 GPU 加速体绘制算法虽然取得了很多成果,但是大部分都是针对传统架构的 GPU 进行的,设计算法时必须将各个执行步骤映射到图形渲染管道中,而且缺少面向 GPU 的软件开发平台的支持。2006 年底,随着面向 DirectX10 的统一渲染架构的 GPU 的发布,GPU 的可编程性能有了大幅度的提升。2007 年, NVIDIA 发布了面向统一渲染架构 GPU 的通用计算软件平台 CUDA (Compute Unified Device Architecture, 计算统一设备架构)。这使得通用计算不再像过去所谓的 GPGPU 架构那样必须将计算映射到图形 API 中, CUDA 提供通用的 API, 直接访问 GPU 内部的众多流处理器,使得程序的设计更加灵活,充分发挥 GPU 的并行处理能力,能够执行更加复杂的操作。所以,面向当代 GPU 的体绘制算法及其优化方法有待于进一步研究。本文通过 CUDA 重新设计并实现了 Ray Casting 体绘制算法,并在此基础上又实现了多种绘制效果,能够在原有的基础上进一步提高绘制速度。

3 基于 GPU 的 Ray Casting 算法

Ray Casting 算法最先由 Levoy M. 于 1988 年提出^[5],是像序体绘制的经典算法。如图 1 所示,它是从投影平面的每一个像素点发射出一条光线,穿过三位体数据场,并按照从前向后或从后向前的顺序进行采样点光属性的混合,最终得到二维投影图像。对于三位体数据场中的每个采样点,可以采用最近邻或三线性的方法插值出其光属性。

基于 GPU 的 Ray Casting 算法的主要思想是把体数据读入到计算机内存之后,当成 3D 纹理绑定到 GPU 纹理内存中,然后再通过核(kernel)函数来访问纹

理内存,进行 Ray Casting 算法的计算。在 Ray Casting 算法中最重要和最大的工作量就是重采样和插值计算,这对于 CPU 是很耗时的。当把体数据存入 GPU 纹理内存之后,可以很方便地由硬件完成双、三次线性插值,快速得到重采样点的数值。此外,由 GPU 的众多流处理器同时执行核函数,并行地完成对多条光线的遍历,从而达到加速处理的目的。

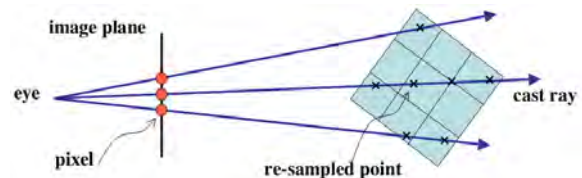


图 1. Ray Casting 算法

3.1 传统架构 GPU 的 Ray Casting 算法

尽管采用 GPU 进行通用计算取得了很好的效果,但传统图形应用接口(Cg)仍然将 GPU 抽象成一个包括纹理、三角形和像素在内的图像绘制器,寻找一种能够使用这些基本元素的映射算法通常比较困难。

传统架构的 GPU 将绘制过程固化为一种流水线的模式,称为图形渲染管道(Graphics Pipeline)。在 GPU 渲染管道中,具有可编程性的有顶点处理器和片段处理器。顶点着色器操作在空间的几何点上,适用于几何操作类的应用;而像素着色器由于是操作在像素一级的单元上,具有较大容量的纹理空间,可以模拟纹理数据及其操作。传统架构 GPU 的体绘制算法必须要将算法的各个实现过程映射到图形渲染管道中,为了充分发挥 GPU 的计算效能,需要将计算任务合理地分配到顶点处理器和片段处理器中,以最大限度地挖掘算法的并行性^[6]。算法实现步骤如下:

- 1) 初始化 OpenGL, 设置各种执行参数。将体数据映射成可以被 GPU 读入的三维纹理,将传输函数映射成一维或二维纹理。
- 2) 编写顶点程序,计算投影坐标和光线方向。
- 3) 编写片段程序,计算光线与体数据的交点,实现对体数据的遍历,采样和计算。

3.2 统一渲染架构 GPU 的 Ray Casting 算法

统一渲染架构的 GPU 整合了顶点处理器、像素处理器以及新加入的几何处理器的所有工作。因此,软件开发者无需再区分指令的属性,统一送到联合管线的 API 下。这种设计使得设计 GPU 算法时不必局限于图形流水线的固定模式,提高了程序的设计更加灵活

性,充分发挥了 GPU 的并行处理能力,更好地使 GPU 负载均衡,提高 GPU 利用率。

统一渲染架构的 GPU 提供通用计算 API(CUDA)。CUDA 能够统一 CPU 串行编程和 GPU 并行计算,它通过一个 NVIDIA C 编译器(NVCC),对代码进行预编译,将 GPU 代码和 CPU 代码分离,GPU 代码被编译成 GPU 执行的机器码,也就是生成调用 CUDA Driver 的代码。而 CPU 的 C 代码则由标准的 C 编译器进行编译。然后通过 Linker 把编译好的模块组合在一起。

基于 CUDA 的 Ray Casting 算法分两步实现:主机(host)执行部分、设备(device)执行部分。host 部分完成如下工作:(1)初始化 OpenGL,设置各项执行参数;(2)将体数据映射成可以被 GPU 读入的三维纹理;(3)将传输函数映射成一维或二维纹理。device 部分就是由 GPU 众多流处理器并行执行的程序,通常称为核函数,主要完成以下工作:(1)计算光线方向和投影坐标;(2)计算光线与体数据的交点;(3)按步长遍历体数据,进行光属性混合。

4 Ray Casting 算法的 CUDA 实现

4.1 主机执行部分

要使用 GPU 对数据进行并行处理,首先需要通过 PCI-E 总线将体数据从 CPU 载入到 GPU 当中,此时体数据以 3D 数组的形式存储。CUDA 实现了内存/显存的调度算法,且对于调用者来说是完全透明的。为了在 GPU 中采用高速的硬件插值来计算重采样点的值,还需要将 3D 数组绑定到 3D 纹理之中。

```
cudaChannelFormatDesc channelDesc =
cudaCreateChannelDesc<uchar>();
CUDA_SAFE_CALL( cudaMalloc3DArray(&d_volumeArray,
&channelDesc,
volumeSize) );
cudaMemcpy3DParms copyParams = {0};
copyParams.srcPtr =
make_cudaPitchedPtr((void*)
h_volume,
volumeSize.width*sizeof(uc
har), volumeSize.width,
volumeSize.height);
copyParams.dstArray = d_volumeArray;
copyParams.extent = volumeSize;
copyParams.kind = cudaMemcpyHostToDevice;
CUDA_SAFE_CALL( cudaMemcpy3D(&copyParams) );
texture<uchar, 3, cudaReadModeNormalizedFloat>
dataTex;
dataTex.normalized = true;
dataTex.filterMode = cudaFilterModeLinear;
```

```
CUDA_SAFE_CALL(cudaBindTextureToArray(dataTex,
d_volumeArray, channelDesc));
```

在提取纹理重采样点之后再还需要对其作用传输函数,得到与之对应的颜色和不透明度值,同样也将传输函数以 1D 纹理的形式绑定到 GPU 当中。通过以上预处理之后,需要 3 维重建的体数据以及传输函数就已经以纹理的形式存储在 GPU 中,可通过 GPU 的众多流处理器直接访问纹理体数据,完成对体数据的遍历、采样及合成。

4.2 设备执行部分

在进行光线投射算法之前,需要在空间坐标系中建立体数据的空间模型。把体数据包围盒定义为 $\text{boxmin}=(-1.0,-1.0,-1.0)$, $\text{boxmax}=(1.0,1.0,1.0)$ 的立方体,重心位于坐标原点。将像平面大小设为 2×2 ,中心位于 $(0,0,2)$ 处,视点坐标位于 $(0,0,4)$,光线从视点发出。如图所示:

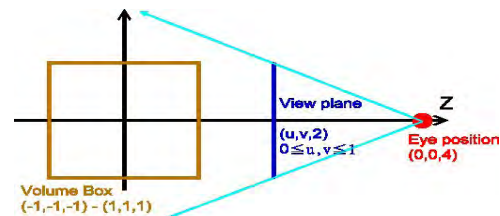


图 2. 空间模型示意图

由 GPU 中的众多流处理器并行地进行多条光线的遍历,光线是从屏幕的每个像素点发出,需要根据光线的方向判断出是否与体数据包围盒相交。对于每个流处理器,可根据其执行线程 ID 确定该像素点在屏幕中的位置,然后再根据光线起始点位置来计算出光线的方向。代码实现如下:

```
uint x = __umul24(blockIdx.x, blockDim.x) +
threadIdx.x;
uint y = __umul24(blockIdx.y, blockDim.y) +
threadIdx.y;
float u = (x / (float) imageWidth)*2.0f-1.0f;
float v = (y / (float) imageHeight)*2.0f-1.0f;
eyeRay.origin = make_float3(0.0f, 0.0f, 4.0f);
eyeRay.dir = normalize(make_float3(u, v, -
2.0f));
```

设光线的参数方程为: $\text{rayPos} = \text{rayOri} + t * \text{rayDir}$ 。需要根据参数 t 来确定采样点的位置, t 的取值范围是 $[\text{tnear}, \text{tfar}]$, 其中 tnear 是光线进入点的 t 值, tfar 是光线离开点的 t 值。

若光线与体数据相交,则按步长遍历体数据。可通过从前向后或从后向前的顺序进行颜色值和不透明度的积累。当采用从前向后遍历体数据时,可采用光线早期终止优化方法提高算法的执行效率。代码如下:

```
float tnear, tfar;
bool hit = intersectBox(eyeRay, boxmin, boxmax,
&tnear, &tfar); //判断光线是否与体数据包围盒相交
if (!hit) return;
```

```

if (tnear < 0.0f) tnear = 0.0f;
float4 sum = make_float4(0.0f);
float t = tnear;
//按步长从前向后遍历体数据
while(t < tfar){
    float3 pos = eyeRay.o + eyeRay.d*t;
    pos = pos*0.5f + 0.5f;
    float sample = tex3D(tex, pos.x,
    pos.y, pos.z);
    float4 col = tex1D(transferTex, sample);
    sum += col.w * (col - sum);
    t += step;
    if(sum.w > 0.98) break; //光线的早期终止
}

```

在空间坐标系中体数据包围盒坐标定义为 $\text{boxmin}=(-1.0,-1.0,-1.0)$, $\text{boxmax}=(1.0,1.0,1.0)$ 。而 3D 纹理坐标已被单位化, 介于 $[0,1]^3$ 之间, 所以当取得采样点的空间坐标时, 需要将其映射成 3D 纹理坐标, 以便得到体数据的采样值。

5 各种绘制效果的实现方法

按照上述方法, 对体数据重采样后再作用传输函数, 按照从前向后的顺序进行颜色和不透明度的积累, 生成最终图像。该绘制方法能够高保真地显示出体数据的内部信息, 是最传统且使用最广泛的绘制方法。但是该方法需要对所有采样点进行颜色值和不透明度的积累, 计算开销较大; 而且绘制出的图像包含了体数据的所有信息, 信息的取舍依赖于传输函数。为此, 针对不同需要应采用不同的绘制方法显示出各种的效果, 在兼顾图像质量的同时也最大限度地提高绘制速度, 使算法的实用性达到最佳。

灰度效果: 灰度效果是一种最原始的绘制方法, 直接把体数据值作为灰度颜色和不透明度进行颜色的积累, 不需要作用传输函数。绘制出的图像较暗, 且比较模糊。由于原始的固定图形渲染管道无法在纹理采样后插入一个步骤再送入帧缓存参与融合, 所以早期采用固定管线的 GPU 通常采用灰度效果绘制。

最大密度投射效果: 对体数据进行遍历时, 将每条光线上所有采样点中的最大值作为该像素的最终颜色值, 不需要作用传输函数。

法向量显示效果: 首先需要设定一个等值面的阈值, 按照从前向后的顺序遍历体数据, 当光线上第一次遇到大于该阈值的采样点时, 计算该点的法向量, 并把计算出的法向量作为发出该光线的像素的最终 RGB 值。

在体绘制中, 由于没有面的信息, 所以无法直接计算出法向量。通常用各数据点的梯度来代替法向量,

为了保证图像质量的同时兼顾运算的复杂性, 通常采用计算简单的基于中心差分的梯度计算方法。

光照效果: 要实现光照效果需要在核函数中根据 Phong 光照模型: $I = K_a + K_d N \cdot L + K_s (N \cdot H)^n$ 计算出光照后的颜色值。其中, L 为入射光; N 为数据点的法向量, 即梯度; H 为入射光与视法向的半角向量; K_a, K_d 和 K_s 分别为环境光系数、漫反射光系数和镜面反射光系数; n 为高光指数。

首先计算出梯度信息作为体数据的法向量, 然后再根据光照方向矢量 L 和观察者视线 V 计算半角矢量

$$H = \frac{L + V}{|L + V|}$$

半透明效果: 根据需要设定两个等值面, 将外部等值面上的数据赋予特别的颜色, 内部等值面则进行光照处理, 再将两个等值面混合, 得到最后的绘制效果。

6 绘制效果及绘制速度比较

本文所用的硬件配置如下: CPU 为 Core2 E8400 3.0GHz, 内存为 2G 的 DDR3 型, 显卡为 NVIDIA Geforce 9600GT, 显存为 512MB。编程环境为 Window XP 下的 Visual Studio 2005 结合 CUDA 进行开发。对人体头部数据($256 \times 256 \times 225$)、心脏数据($256 \times 256 \times 256$)进行绘制, 数据大概 15MB 左右, 显示窗口大小为 512×512 像素。

各种绘制效果如图 3 所示。最大密度投射绘制出的图像比较清晰, 具有很好的对比度, 但是交互性较差, 用户无法根据实际需要有选择性的绘制体数据的内部信息。法向效果虽然没有作用传输函数, 但是将体数据的法向量最为颜色值进行绘制, 图像较清晰, 光线柔和, 但是所绘制的是表面信息, 而且无法调整最终图像的颜色值。光照效果绘制出的图像真实感较强, 但是只能绘制出表面信息, 无法显示出体数据的内部信息。采用半透明绘制时, 可根据实际需要适当调整等值面的阈值大小, 高保真地显示出用户感兴趣的体数据信息。

表 1. 各种绘制效果的绘制速度比较

(单位: fps)

	传统 绘制	灰度	最大密 度	法向表 示	光照	半透 明
头部	37.34	56.54	62.82	64.66	57.64	52.92
心脏	38.79	55.87	63.91	62.24	53.89	51.76

对于这种规模的数据很难以 CPU 实时地交互绘制。采用 GPU 并行加速处理之后绘制速度有了很大幅度的提升, 能够对一定规模的体数据进行实时地三维重建。传统的光线投射绘制方法需要将体数据的全部

信息可视化, 绘制速度相对较慢; 其它的各种绘制效果与传统的绘制效果相比在绘制速度方面有了一定的提高。

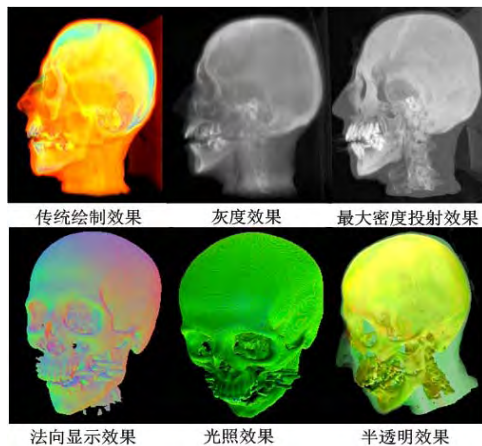


图 3. 各种绘制效果展示

7 结论

当代 GPU 强大的可编程性能能够使计算开销较大的体绘制算法实时交互地显示, 提高了算法的执行速度。面向统一渲染架构 GPU 的体绘制算法充分提高了代码的设计灵活性, 绘制效果更好, 效率更高。

然而采用 GPU 作为并行计算设备同样存在着一些

问题。首先, GPU 有限的内存容量限制了所处理数据的规模。此外 GPU 与系统主存之间只能通过 PCI-E 总线传输数据(8GB/S), 无法实现高速传输。

一般医学体数据中“透明”体素占到 70% 以上。若直接对体数据进行重采样, 得到的大部分采样点对图像的最终合成是无贡献的。将这些“透明”体素区域剔除能有效的提高算法的执行效率。剔除这些透明数据需要采用特殊的数据结构(如八叉树)。面向统一渲染架构 GPU 的体绘制算法的优化方法有待于进一步研究。

References (参考文献)

- [1] Krueger J, Westermann R. Acceleration Techniques for GPU-based Volume Rendering [C]. In Proceedings of IEEE Conference on Visualization 2003, Seattle, 2003, 38-45
- [2] S. Stegmaier, et al. A simple and flexible volume rendering framework for graphics- hardware-based Raycasting [C]. Proceedings of Volume Graphics, New York, 2005, 187-195
- [3] Thomas Klein, Magnus Strengert, et al. Exploiting Frame-to-Frame Coherence for Accelerating High-Quality Volume Raycasting on Graphics Hardware [C]. Proceedings of the IEEE Visualization 2005, 223-230
- [4] Daniel Ruijters, Anna Vilanova. Optimizing GPU Volume Rendering [J]. Winter School of Computer Graphics, Pilzen, 2006, 14,
- [5] M. Levoy. Display of surfaces from volume data [J]. IEEE transaction on computer graphics and applications, 1988, 8(3): 29-37
- [6] Xu jingjun, Yang xin, Gao yan. Ray Casting-Based volume Rendering Algorithm Using GPU Programming. [J]. Journal of Computer-Aided Design & Computer Graphics. 2007, 19(2), P257-262
储璟骏 杨新 高艳. 使用 GPU 编程的光线投射体绘制算法 [J]. 计算机辅助设计与图形学学报 2007, 19(2), P257-262.