一种基于GPU的粒子系统火焰模拟

邱宇峰[1] 曾国荪[2]

(同济大学电子与信息工程学院计算机科学及技术系 上海 201804)[1]

(国家高性能计算机工程技术中心同济分中心 上海 201804)[2]

摘 要 针对传统火焰模拟耗时、模拟真实性不理想这一问题,提出了一种借助GPU(图形处理器)高通用计算能力进行并行模拟的粒子系统火焰模拟方法。该方法采用基于方位角和仰角的粒子散射器、基于层流火焰轮廓计算公式的火焰外形计算及通过拉格朗日插值方法平滑火焰骨架线等手段提高了紊流火焰的模拟真实度。在提高性能方面,该方法使用全局存储空间存储粒子信息,避免了因使用纹理存储而产生的反复绑定的开销,同时结合GPU强大的通用计算能力,通过采用CUDA(计算统一设备架构)编写的并行算法,实现了基于GPU的拉格朗日插值并行求解及并行计算、更新粒子属性,并就如何确定块内线程数量作了论述。实验结果表明,该方法满足了火焰模拟的真实性和实时性要求,且较传统方法有很大的提高。

关键词 粒子系统,图形处理器,通用计算,方位角,仰角,拉格朗日插值,计算统一设备架构

中图法分类号 TP338 **文献标识码** A

Fire Simulation with GPU-based Particle System

QIU Yu-feng[1] ZENG Guo-sun[2]

(Deptartment of Computer, College of Electronic and Information Engineering, Tongji University, Shanghai 201804, China)[1]

(Tongji Branch, National Engineering & Technology Center of High Performance Computer, Shanghai 201804, China)[2]

Abstract This approach aims at improving the performance and reality of fire simulation in virtue of powerful ability of GPUPU. An emitter composed of yaw and pitch was designed to control the particle stream precisely. The formula of laminar flame was modified to outline the shape of turbulent fire. Besides, Lagrange interpolation was used to smooth and get the accurate fire skeleton around which particles move even when it becomes twist. In order to improve the performance, global memory was adopted to store particles to prevent the spending of binding texture memory repeatedly; parallelization of Lagrange interpolation and properties of particles update were realized of CUDA. All these measures contribute to reach a desirable real-time simulation speed and acquire an improvement of performance.

Keywords Particle system, GPU, General-purpose computation, Yaw, Pitch, Lagrange interpolation, CUDA

如何更真实、更高效地模拟复杂多变的自然现象,一直是计算机图形学和并行计算很重要的研究课题。继W. T. Reeves提出了用粒子系统模拟不规则模糊物体的方法[1]后,基于粒子系统的研究成果层出不穷。已有的研究成果[2-5]中有些采用粒子系统结合纹理技术的方法,有些则是通过结合流体模型来获得到良好的模拟效果。尽管粒子系统是研究不规则物体在计算机上生成的优秀方法,但文献[6]也指出,当基于CPU的传统粒子系统的粒子数量超过10k时,很难做到实时模拟,其性能瓶颈主要在于CPU与存储的吞吐能力弱。GPU具有强大的图形处理能力和并行处理能力,因此文献[7-9]借此加速了模拟火焰,取得了性能上的提高。但是目前基于GPU计算加速的做法通常选取纹理空间存储粒子,而粒子的属性需要实时更新,因此带来一定的绑定开销,且其速度不比采用全局存储空间快。

针对上述问题,本文提出了一种新的基于GPU粒子系统的火焰模拟方法。在提高火焰模拟真实度方面,本文在文献[10]的"火焰骨架线"基础上,通过拉格朗日插值法对火焰骨架上n个节点的计算,得到平滑的骨架线。在模拟火焰受干扰力作用时,粒子运动轨迹根据火焰骨架线的扭曲调整自己的位置,从而得到不失真的紊流火焰。本文设计了一个基于方位角和仰角的粒子散射器,用于精确控制粒子流。此外,本文方法中的火焰轮廓也参照一定的物理模型,再结合纹理贴图法,得到了形态较真实的紊流火焰。性能方面,本文设计的GPU粒子系统的主要存储和运算由GPU完成,极大地避免了CPU与GPU之间的数据通讯瓶颈,且粒子存储于全局存储空间内,避免了使用纹理存储所附带产生的反复绑定的开销。本文还采用CUDA编写基于GPU硬件加速的并行算法来并行计算、更新粒子的属性。实现了基于GPU的拉格朗日插值并行算法,并对CUDA block大小的定义做了论述。本文方法达到了实时模拟所需的性能要求,并较传统方法有

到稿日期:2009-05-07 返修日期:2009-07-07 本文受863专项课题(2007AA01Z425),973计划课题(2007CB316502)资助。

邱宇峰(1985-),男,硕士研究生,研究领域为异构计算、高性能计算,E-mail:qsurefond@163.com;曾国荪(1964-),男,博士,教授,博士生导师,研究领域为异构计算、并发理论、可信计算、信息安全。

很大的提高。

1 基于GPU粒子系统的设计

1.1 数据存储

粒子系统分为粒子管理层和粒子两部分。粒子管理层负责管理粒子的生成、散射等，其属性包括生存周期、粒子数量、重力加速度、扰动等；粒子属性则包括位置、透明度、颜色、速度、生存时间、重力等。

粒子的位置、颜色、速度和生命值是最为关键的属性。本文设计的粒子都是状态保存粒子[6]，每画一帧都要对其属性值计算一次，且需要保存粒子的前一次状态。GPU的纹理存储空间可以被缓存，与全局存储空间、共享存储空间相比访问速度更快。但是，GPU为了维持数据一致性，将其设置为线程只读，因而纹理存储空间只适用于作为索引表供频繁访问。文献[8]采用创建纹理来存储粒子数据，每次计算后交换纹理的输入和输出角色。这样的方法需要反复绑定纹理存储，而全局存储可读可写。因此相比较而言，使用纹理存储并不比使用全局存储空间效率高，所以本文的设计是将粒子属性存储在全局存储空间中。

考虑到粒子管理系统的属性被频繁读取且很少改变，因此本文设计将粒子管理系统的属性存放于纹理存储空间中。

1.2 基于GPU全局存储的渲染流程

本文结合纹理技术渲染火焰，并且本文方法主要使用GPU全局存储，因此其渲染流程与采用纹理存储的方法不一样。本文方法的渲染流程如图1所示。

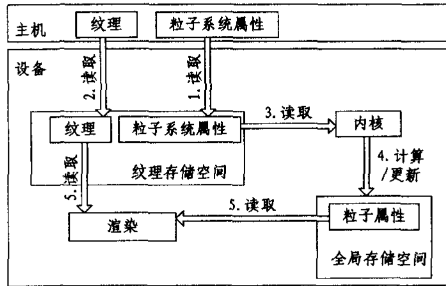


图1 基于全局存储的渲染流程

GPU粒子系统绘制火焰的处理流程如下。

步骤1 主机将粒子系统属性加载到纹理存储。

步骤2 每画一帧，主机将绘制火焰用到的纹理加载到纹理存储。

步骤3 内核从纹理存储空间（通常是从纹理缓存）读取粒子系统属性。

步骤4 在设备上执行多线程（内核是多线程操作的单位），从而更新位于全局存储中的粒子属性。

步骤5 从纹理存储和全区存储中取出纹理和粒子属性进行渲染。

步骤6 重复步骤2。

2 火焰设计

2.1 基于方位角与仰角的粒子散射

2.1.1 基于方位角和仰角的粒子散射器

粒子散射是生成粒子、确定粒子初始速度与方向的关键步骤，对保持火焰模拟的真实度至关重要。

为了使产生的粒子不在一条直线上，采用了方位角 *yaw* 和仰角 *pitch* 两个量，如图2所示。本文由于采用DirectX进行图像输出，故采用DirectX的"右手坐标系"，坐标示意图参看文献[11]。其中 *yaw* 表示 xoz 平面上粒子的散射角度，是与正 x 轴的夹角；*pitch* 表示粒子相对于3个坐标轴的角度，用来确定初始速率的大小，与3坐标轴正轴相夹的 *pitch* 均相同，*pitch* 保存在粒子属性中。这里的 *yaw* 与 *pitch* 均用分度表示：

$$yaw=\Delta\times 2\pi \tag{1}$$

$$pitch=\Delta\times\frac{2\pi\theta}{360^\circ} \tag{2}$$

其中，$0<\Delta\leqslant 1$，$-90^\circ<\theta<90^\circ$。

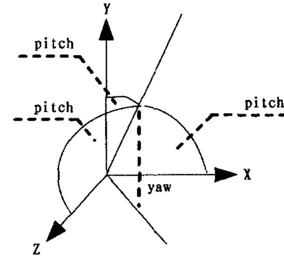


图2 方位角和仰角示意图

2.1.2 速度初始化

我们设计3坐标轴方向的初始速度的方向与分量在(0，1)区间内，下面是初始速度公式：

$$\begin{cases}v_x=\sin(pitch)\times\cos(yaw)\\ v_y=\cos(pitch)\times\cos(0)\\ v_z=\sin(pitch)\times\sin(yaw)\end{cases} \tag{3}$$

说明：*yaw* 是与正 x 轴相夹的角，与 z 轴相夹的角为 $90^\circ-yaw$。因此对 v_x 而言，采用 $\cos(yaw)$，对 v_z 而言采用 $\sin(yaw)$；同理，*pitch* 角对 v_y 而言采用 $\cos(pitch)$，而对 v_x 和 v_z 则采用 $\sin(pitch)$。

接下来对初始速度进行放大，使其符合现实世界中的速度矢量。分两步实施，操作如下：首先确定现实世界速度标量（对3坐标相同，因此采用标量），与设计粒子生命值的思路相同，我们随机化粒子速度。

$$v_0=v_{basic}+\Delta\times v_{var} \tag{4}$$

其中，$0<\Delta\leqslant 1$。

根据式(3)确定的初始速度方向实施速度放大：

$$\begin{cases}v_{init.x}=v_x\times v_0\\ v_{init.y}=v_y\times v_0\\ v_{init.z}=v_z\times v_0\end{cases} \tag{5}$$

至此完成了新生成粒子的初始速度 v_{init} 设置工作。

注：本文的速度更新是基于欧拉方法，即 $v(t+\Delta t)=v(t)+a\Delta t$，这里的 a 是粒子的重力和重力场的合力。

2.2 拉格朗日插值平滑火焰骨架

Philippe Beaudoin[10]等人提出了"火焰骨架"的概念，通过节点所处的扰力速度场求得火焰骨架的 n 个节点。在该方法中，火焰外形轮廓根据骨架相邻节点间的距离以及所处的柱坐标系分段求取。正是因为分段求取外轮廓，使得这种做法在多紊流条件下的模拟情况并不十分真实，所以本文根据该方法求得的 n 个骨架节点从另一角度进行处理。

首先，用拉格朗日插值法平滑火焰骨架线并获取骨架线上的各点坐标；然后，层流火焰外焰轮廓根据骨架线调整轮廓

以达到紊流状态下的火焰效果。此外，在粒子的运动过程中，粒子的运动轨迹也参照骨架线做相应的变化。

2.2.1 平滑火焰骨架

给定 n 个火焰骨架节点 $S_0, S_1, \cdots, S_n$，即给定一个骨架线上 P 点的 y 坐标分量，采用拉格朗日插值法就可以求得相应的骨架线上 P' 点的 x 和 z 的坐标分量：

$$P'_{x,z} = \text{lagrange}(S_0, S_1, \cdots, S_n, P_y) \tag{6}$$

通过式(6)求得 P' 点处的偏移量：

$$\Delta f = P'_{x,z} - S_{0.xz} \tag{7}$$

Δf 如图 3 所示。求得的火焰骨架线以及 Δf 均为火焰轮廓和粒子位置变迁的基础。

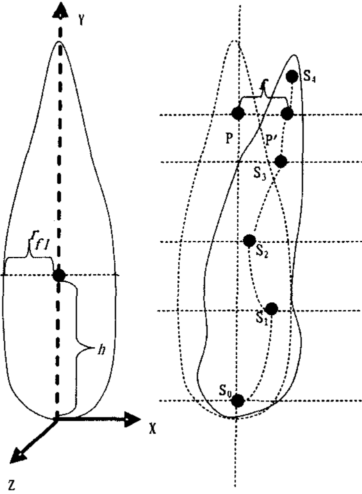


图 3 火焰轮廓与骨架线

2.2.2 外形轮廓

文献[12]给出了层流射流扩散火焰高度与火焰半径之间的关系。根据火焰模拟的特点简化和改变一系列物理变量，将其改造为如下公式：

$$\frac{r_{fl}}{h} = \left[\frac{64}{3Re^2}\left\{\left(\frac{3}{8}Re\frac{r_0}{h}\right)^{\frac{1}{2}} - 1\right\}\right]^{\frac{1}{2}} \tag{8}$$

其中，$Re = \frac{\overline{v_{\text{init}}} r_0}{v}$，$r_0$ 是粒子流的半径；$\overline{v_{\text{init}}}$ 是燃料流初始速率，在设计中，这个值相当于所有粒子的初始平均速率；v 是运动粘性，这里简化为 1。文献[12]原式中的 f_s 这里简化为 1，r_{fl} 和 h 如图 3 所示。式(8)能够确定层流状态下的火焰外轮廓。为了解决紊流条件下的火焰模拟，式(8)的基础上参照火焰骨架线进行调整。

考虑到受紊流影响的火焰高度比层流火焰的高度小，如图 3 所示，即在火焰骨架扭曲之后存在一定的高度差，故我们对式(7)中的 h 做一个调整。设 h 处于 $S_{i-1}.y$ 与 $S_i.y$ 之间，$1 \leqslant i \leqslant n$，式(8)变为：

$$\frac{r_{fl}'}{h\left(1 - \frac{s_i.y - s_{i-1}.y}{|s_{i-1}s_i|}\right)} = \left[\frac{64}{3Re^2}\left\{\left(\frac{3}{8}Re\frac{r_0}{h\left(1 - \frac{s_i.y - s_{i-1}.y}{|s_{i-1}s_i|}\right)}\right)^{\frac{1}{2}} - 1\right\}\right]^{\frac{1}{2}} \tag{9}$$

通过式(9)得到的火焰半径和式(6)，显然很容易就能通过 Δf 和骨架线 s_0 点的坐标求得火焰轮廓线上各点的坐标。

2.2.3 位置

每个粒子每运动一次，就先计算其最新位置的 y 坐标，根据这个 y 坐标获取火焰骨架上相同 y 坐标的点 S，再计算 S 相对于火焰骨架根节点 S_0 的 x 和 z 分量的偏量，然后做出相应的位置改变。

根据粒子属性中保存的 $pitch$，先计算 y 方向上的位置分量：

$$p_y(t+\Delta t) = (p_y(t) + v_y\Delta t)(1 - \cos(pitch)) \tag{10}$$

其中，v_y 为 t 时刻粒子在 y 方向的速率。根据 $p_y(t+\Delta t)$ 获取火焰骨架上的相同 y 坐标的点 S，再计算 S 相对于火焰骨架根节点 S_0 的 x 和 z 分量的偏量，记为 Δf_x 和 Δf_z，于是有了 $t+\Delta t$ 时刻粒子的位置：

$$\begin{cases} p_x(t+\Delta t) = (p_x(t) + v_x\Delta t) + \Delta f_x \\ p_y(t+\Delta t) = (p_y(t) + v_y\Delta t)(1 - \cos(pitch)) \\ p_z(t+\Delta t) = (p_z(t) + v_z\Delta t) + \Delta f_z \end{cases} \tag{11}$$

3 GPU 并行加速

3.1 拉格朗日 GPU 并行算法

P. K. Jana 在文献[13]中给出了一种适合多处理器结构的并行插值算法。由于 GPU 有多个执行核，适合多线程运算，因此本文在此基础上设计了一种基于 GPU 的用 CUDA 实现的多线程并行插值算法。这里给出此改进算法的 CUDA 伪代码，并给出改进的算法效率。

1)算法描述

假设已通过 Philippe Beaudoin 方法[10]获取了骨架上 n 个节点的坐标，分别记作 $s_i(0<i<n)$，这里的 s_i 是三维坐标。假设需要的线程数为 n^2 个，在理论上形成 $n \times n$ 网格的拓扑结构，线程索引为 (i,j)，代表第 i 行第 j 列的线程，$0 \leqslant i,j \leqslant n-1$。给 $(0,0)$ 线程分配 3 个存储单元，记为 $A(0,0)$，$B(0,0)$ 和 $D(0,0)$；给 (k,k) 线程 $(0<k \leqslant n-1)$ 分配存储单元 $A(k,k)$ 和 $B(k,k)$；给其余的线程 (i,j) 只分配一个存储单元 $A(i,j)$ $(0 \leqslant i,j \leqslant n-1$ 且 $i \neq j)$。

```
#Begin
〈CPU〉 step1:
  1.1: Allocate 4 float4 vector-arrays A[n], B[n], D[n], S[n] in Device
  1.2: Initialize A[n], B[n], D[n], S[n];
       Copy S[n] from host to device through cudaMemcpy( HostToDevice )
〈CPU〉 step2: While( S[0].y < y < S[n-1].y ) do
       1) Compute x-coordinate with Parallel_Lagrange ( A[n], B[n], C[n], S[n], y) in Device from step3 to step9
       2) Compute z-coordinate with Parallel_Lagrange ( A[n], B[n], C[n], S[n], y) in Device from step3 to step9
       done
〈GPU〉 step3: Do the step3.1 and step3.2 in parallel
  3.1: for all thread (i,j), (i=threadIdx.x, j= threadIdx.y), A(i,j) =S[j].y
  3.2: for all thread (i,j), if (i==j), B(i,i)=y
〈GPU〉 step4: Do the step4.1 to step4.4 in parallel
  4.1: for all thread (i,j), if(i==j), A(i,i)=B(i,i)-A(i,i)
  4.2: for all thread (i,j), if(j < i), A(i,j)=A(i,j)-A(j,i)
  4.3: for all thread (i,j), if(i < j), A(i,i)=A(i,i)-A(j,i)
  4.4: for all thread (i,j), if(i==j), B(i,i) = A(i,i)
〈GPU〉 step5: Do the step5.2 in parallel
  5.1: for thread (0,0), B(0,0)=∏_{k=0}^{n-1} B(k,k)
```

5.2: for all thread (i,j), if(i==j), $A(j,j)=\prod_{k=0}^{n-1}A(k,j)$

〈GPU〉 step6: Do the step6 in parallel
for all thread (i,j), B(i,j)=S(i).x
(if compute z-coordinate, B(i,j)=S(i).z)

〈GPU〉step7: Do the step7 in parallel
for all thread (i,j), if(i==j), A(i,i)=B(i,i)/A(i,i)

〈GPU〉 step8: Store results of Largange intrepolation to A(0,0)

8.1: $A(0,0)=\sum_{i=0}^{n-1}A(i,i)$

8.2: A(0,0)=A(0,0) * D(0,0)

〈CPU〉 step9: Copy A(0,0) from device to host through Memcpy(Device->Host)

〈CPU〉 step10: Compute z-coordinate from step2

#End

2)算法分析

经分析得知，并行算法的 setp3，setp4，setp6 和 setp7 需要 O(1)步，step5 和 step8 需要 $\log(n)$ 时间步，因而基于 GPU 的拉格朗日算法共需要 $2\log(n)+O(1)$ 时间步。

3.2 GPUPU 多线程更新粒子属性

我们采用分治方法多线程更新粒子，每个线程分别管理一部分粒子的更新操作。这里不考虑粒子之间的相互作用力，更新操作没有粒子间的相互数据依赖性。由于粒子众多，我们不能为每个粒子都设置一个线程来负责对该粒子的更新操作，因此只能让每个线程负责一片区域内粒子的操作。这里假设一片区域包含对 k 个粒子的操作。

以下我们将使用 CUDA 的 grid，block 和 thread，相关概念参看文献[14]。这里假设有 N 个粒子，$N=$ width * height，存放粒子属性的数组为 particles[width][height]。若设定 block 的维度为(m,m)，则相应的 grid 的维度为(width/m，height/m)。下面给出基于 CUDA 的算法伪代码。

#begin

〈CPU〉step1：每画一帧就调用运行在设备上的多线程方法 kernel_particleUpdate()

〈CPU〉step2：设线程索引为(i,j)，设线程 ID 为 tid，对于大小为(Dx，Dy)的二维 block[14]，索引为 (i,j)的线程的 tid 为(x+yDx)。设粒子的索引为(s,t)，ID 为 tid 的线程负责操作(s+t)/k 的粒子，依次对粒子的众多属性进行计算和更新操作

〈CPU〉step3：用_syncthreads() 对所有线程进行同步

#end

经分析得知，只需 1 步即可完成每一帧的更新操作。步骤 3 所需的同步操作也会带来一定的性能损耗，但相比 CPU 算法处理每一帧的更新操作，其速度加快了近 N 倍，也省略了 CPU 与 GPU 之间的通讯延迟。

3.3 Block 的 thread 数量确立

在 NVIDIA 的 GPU 里，基本的处理单元是 SP(Streaming Processor)。GPU 里面有多个 SP 可做并行计算，一定数量的 SP 附加其他的单元组成一个 SM(Streaming Multiprocessor)。GPU 在执行时会将 block 交给 SM 执行 thread，而 block 里的 thread 又被分成以 warp 为单位的组执行。在 CUDA 中，wrap 是 SM 执行的最小单位。所以不同于 CPU 算法，基于 GPU 的算法至关重要的是确定 block 内线程数量。

在 compute capability 1.0/1.1 的 GPU 中，每个 SM 可以同时管理最多 768 个线程，每个 wrap 的大小最好为 32[14]，每个 SM 可容纳的 block 最多为 8 个。这里可获得如下两种最好情况：

(1)wrap 数最大即 24 个，分配给每个 SM 的 block 数为 3，每个 block 的 thread 数为 256。

(2)每个 SM 的 block 数最大即 8 个，则 wrap 数为 3，每个 block 的 thread 为 96。

但是不可忽视的是，每个线程的执行还需要 shared memory，register 的配合。SM 里的多个 block 共享这些资源，如果资源不够分配，则 CUDA 会相应减少 block 的数量，继而影响分配给 SM 的线程数量。设 shared memory 为 S，register 数量为 R，每个 block 占用的 shared memory 为 b_s，每个 thread 占用的 register 数为 t_r，w 为 wrap 数量，每个 SM 所确定的 block 数量为 b，线程数量为 t，则 $b=S/b_s$，$t=R/t_r$，$w=t/32$。GPU 的 register 数量即使很多(如 8192 个)，但若每个 thread 占用 16 个，则仅能供 512 个 thread 同时使用，小于 SM 所能同时处理的最大线程数 768。另外，shared memory 也有限，限制了 block 的数量。在选择 block 的线程数量上，原则是若 t 接近 SM 能同时处理的最大线程数，则定 t，再计算需要多少个 block；若 b 比较接近 SM 能同时处理的 block 数，则定 b，再确定每个 block 的 thread 数量；若 w 较接近最大 wrap 数，则定 w，再确定 b 和块内 thread 数。

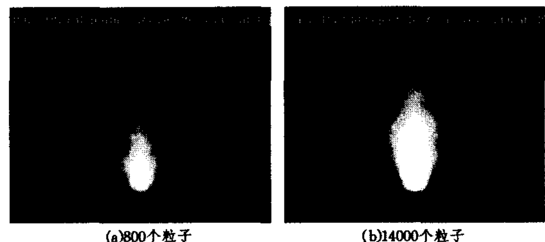
4 实验结果与分析

本文方法实验环境：NVIDIA GeForce 9800GT，Intel Core2 Duo 2.33GHz，2.00GB RAM，Windows XP，Visual Studio 2008，DirectX SDK(August 2008)，CUDA2.1 编译环境。

4.1 实验截图

图 4 展示了文献[10]中采用的骨架线所形成的火焰，图 5 为用拉格朗日插值法平滑火焰骨架线所形成的火焰。其中，图 4 和图 5(a)均用 800 个粒子模拟火焰；图 5(b)为 14000 个粒子的火焰形态，绘制速度为 70 帧/s 左右。

图 4　未使用骨架线平滑



(a)800个粒子　　(b)14000个粒子

图 5　拉格朗日法平滑火焰骨架线

4.2 绘制效率对比

表 1 为基于 GPU 算法的粒子系统与采用本文方法的粒子系统的绘制性能对比表，图 6 为表 1 的图表表示图。根据

表1,当粒子数为14000个左右时,传统CPU算法实现的粒子系统只有10帧/s左右的绘制速度,而本文方法实现的火焰模拟能达到70帧/s以上的绘制速度,完全能达到实时模拟的要求。

表1 本文方法与CPU算法粒子系统对比结果

粒子数(个)	传统方法(帧/s)	本文方法(帧/s)
750	620	790
1000	450	600
2000	350	390
6000	155	160
8000	80	121
10000	34	98
14000	12	70

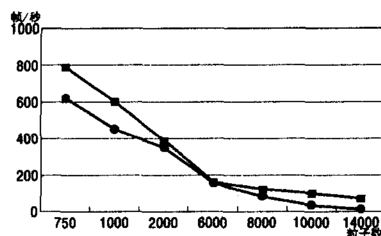


图6 传统算法与本文算法的粒子系统性能对比

4.3 实验结果分析

通过对比图4与图5,表明在受到外力干扰时,本文采用的拉格朗日插值平滑火焰骨架的方法所模拟的火苗摇摆幅度和扭曲状更符合自然界的形态。根据实验截图,能看清火焰的内、外焰,模拟结果较为真实,其得益于本文粒子散射器的独特设计和粒子颜色、大小的变化设计。

图6和表1表明本文方法相比CPU算法的粒子系统具有明显的性能优势。值得注意的是,在粒子数为6000个时,基于CPU方法和本文方法的绘制效率接近,经分析,系此时的粒子规模使得CUDA在同步grid时有性能上的损耗,所获得的处理速度与CPU算法的粒子系统计算速度接近所致。除此以外,本文方法相比CPU方法其性能大幅提高。

综上,本文方法完全能达到实时模拟的要求,模拟的火焰效果逼真。

结束语 本文主要介绍了一种基于GPU算法的粒子系统火焰模拟方法。方法的优点是采用了GPU的全局存储空间存储粒子的属性,避免了使用纹理存储空间而产生的绑定、释放操作带来的延迟;用GPUPU改造了拉格朗日并行算法,根据火焰骨架的节点,求取平滑火焰骨架上各点的坐标,粒子根据火焰骨架上对应点与骨架根基点 x 轴和 z 轴的位移,做位置的相对移动,从而保证火焰在扰动状态下不失真;利用GPUPU的并行方法多线程计算更新粒子的属性,并结合纹理贴图方法,用DirectX与GPU互操纵渲染火焰。实验结果表明,此方法在提高火焰实时性方面有很大的作用。

未来的研究方向主要有以下几点:

(1)N-body问题是粒子系统的一种,本文提到的方法并未考虑粒子间的相互作用,这方面工作应该多结合N-body问题的研究成果。

(2)LBM[15]方法、物理模型[16]方法都有其特点,可以结合两类方法的优点来提高真实性。

(3)结合CPU和GPU的优点,消除现有硬件瓶颈,提高运算速度。GPU虽然有强大的并行计算能力,但仍然无法完全替代通用处理器。目前Intel,AMD等CPU厂商正致力于将GPU与CPU整合到一起,那么如何对CPU与GPU进行合理的分配,如何挖掘和整合各自的优点成了一大难题。解决负载均衡成了解决这一问题的重点,这方面国内也有一定的优秀成果[17]。解决好CPU与GPU的整合问题,能将并行计算推上一个新台阶,势必将推动仿真领域的革新。

参考文献

[1] Reeves W T. Particle Systems-A Technique for Modeling a Class of Fuzzy Objects[J]. ACM Computer Graphics, 1983, 17(3): 359-376

[2] Matthias U, Andrzej T. Cloud Simulation in Virtual Environments[C]//IEEE Visualization Proceedings. IEEE, 1998: 98-104

[3] Chen J X, Wegman E J, Fu X, et al. Near Real-time Simulation of Particle Systems[C]//Proceedings of International Workshop on Distributed Interactive Simulation and Real-time Applications. 1999: 33-40

[4] Takeshita D, Shin O T A, et al. Particle-based Visual Simulation of Explosive Flames[C]//Proceedings of the 11th Pacific Conference on Computer Graphics and Applications. USA, New York: IEEE, 2003: 482-486

[5] 杨仕颖,彭真明,刘迎春.基于粒子系统的导弹尾焰和尾迹的实时模拟[J].系统仿真学报,2008,20(19):5181-5184

[6] Cai Xing-quan, Li Jin-hong, Yang Jian, et al. Advanced GPU-based State-Preserving Particle System[C]//Proceedings of the 7th World Congress on Intelligent Control and Automation. USA, New York: IEEE, 2008: 4230-4246

[7] 李建明,吴云龙,迟忠先,等.基于流体模型和GPU加速的火焰实时仿真[J].系统仿真学报,2007,19(19):4382-4385

[8] 张汉清,张科.基于GPU粒子系统的战场实时雨雪效果模拟[J].计算机仿真,2007,24(10):200-203

[9] Bhattacharjee S, Patidar S, Narayanan P J. Real-time Rendering and Manipulation of Large Terrains[C]//Sixth Indian Conference on Computer Vision. Indian: Graphics & Image Processing, 2008: 551-559

[10] Beaudoin P, Paquet S, Poulin P. Realistic and Controllable Fire Simulation[C]//Proceedings of Graphics Interface. Ottawa Ontario: Canadian Human-Computer Communications Society, 2001: 159-166

[11] DirectX. SDK Manual[M]. Microsoft, August 2008: 3-70

[12] 赵坚行.燃烧的数值模拟(第一版)[M].北京:科学出版社,2002:16-68

[13] Jana P K, Siniia B P. Fast Parallel Algorithm for Polynomial Interpolation[J]. Computers Math, 1995, 29(4): 85-92

[14] Cuda N. Programming Guide Version 2.0(Sencond Edition)[M]. NVIDIA, 2008: 1-75

[15] Wei Xiao-ming, Li Wei, Mueller K, et al. The Lattice-Boltzmann Method for Simulating Gaseous Phenomena[J]. IEEE Transactions on Visualization and Computer Graphics, 2004, 10(2): 164-176

[16] Nguyen D Q, Ronald F, Jensen H W. Physically Based Modeling and Animation of Fire[J]. ACM Transactions on Graphics, 2002, 21(3): 721-728

[17] 曾国荪,陆鑫达.异构计算中的负载共享[J].软件学报,2000,11(4):551-556