CUDA raytracing algorithm for visualizing discrete element model output

Anders Damsgaard Christensen, *20062213*

***Abstract*—A raytracing algorithm is constructed using the CUDA API for visualizing output from a CUDA discrete element model, which outputs spatial information in dynamic particle systems. The raytracing algorithm is optimized with constant memory and compilation flags, and performance is measured as a function of the number of particles and the number of pixels. The execution time is compared to equivalent CPU code, and the speedup under a variety of conditions is found to have a mean value of 55.6 times.**

***Index Terms*—CUDA, discrete element method, raytracing**

I. INTRODUCTION

VISUALIZING systems containing many spheres using traditional object-order graphics rendering can often result in very high computational requirements, as the usual automated approach is to construct a meshed surface with a specified resolution for each sphere. The memory requirements are thus quite high, as each surface will consist of many vertices. Raytracing [1] is a viable alternative, where spheric entities are saved as data structures with a centre coordinate and a radius. The rendering is performed on the base of these values, which results in a perfectly smooth surfaced sphere. To accelerate the rendering, the algorithm is constructed utilizing the CUDA API [2], where the problem is divided into $n \times m$ threads, corresponding to the desired output image resolution. Each thread iterates through all particles and applies a simple shading model to determine the final RGB values of the pixel.

Previous studies of GPU or CUDA implementations of ray tracing algorithms reported major speedups, compared to corresponding CPU applications (e.g. [3], [4], [5], [6]). None of the software was however found to be open-source and GPL licensed, so a simple raytracer was constructed, customized to render particles, where the data was stored in a specific data format.

A. Discrete Element Method

The input particle data to the raytracer is the output of a custom CUDA-based Discrete Element Method (DEM) application currently in development. The DEM model is used to numerically simulate the response of a drained, soft, granular sediment bed upon normal stresses and shearing velocities similar to subglacial environments under ice streams [7]. In contrast to laboratory experiments on granular material, the discrete element method [8] approach allows close monitoring of the progressive deformation, where all involved physical parameters of the particles and spatial boundaries are readily available for continuous inspection.

The discrete element method (DEM) is a subtype of molecular dynamics (MD), and discretizes time into sufficiently small timesteps, and treats the granular material as discrete grains, interacting through contact forces. Between time steps, the particles are allowed to overlap slightly, and the magnitude of the overlap and the kinematic states of the particles is used to compute normal- and shear components of the contact force. The particles are treated as spherical entities, which simplifies the contact search. The spatial simulation domain is divided using a homogeneous, uniform, cubic grid, which greatly reduces the amount of possible contacts that are checked during each timestep. The grid-particle list is sorted using Thrust[1], and updated each timestep. The new particle positions and kinematic values are updated by inserting the resulting force and torque into Newton's second law, and using a Taylor-based second order integration scheme to calculate new linear and rotational accelerations, velocities and positions.

B. Application usage

The CUDA DEM application is a command line executable, and writes updated particle information to custom binary files with a specific interval. This raytracing algorithm is constructed to also run from the command line, be non-interactive, and write output images in the PPM image format. This format is chosen to allow rendering to take place on cluster nodes with CUDA compatible devices.

Both the CUDA DEM and raytracing applications are open-source[2], although still under heavy development.

This document consists of a short introduction to the basic mathematics behind the ray tracing algorithm, an explaination of the implementation using the CUDA API [2] and a presentation of the results. The CUDA device source code and C++ host source code for the ray tracing algorithm can be found in the appendix, along with instructions for compilation and execution of the application.

II. RAY TRACING ALGORITHM

The goal of the ray tracing algorithm is to compute the shading of each pixel in the image [9]. This is performed by creating a viewing ray from the eye into the scene, finding the closest intersection with a scene object, and computing the resulting color. The general structure of the program is demonstrated in the following pseudo-code:

Contact: anders.damsgaard@geo.au.dk
Webpage: http://users-cs.au.dk/adc
Manuscript, last revision: October 20, 2011.

[1]http://code.google.com/p/thrust/
[2]http://users-cs.au.dk/adc/files/sphere.tar.gz

```
for each pixel do
  compute viewing ray origin and direction
  iterate through objects and find the closest hit
  set pixel color to value computed from hit ↩
      point, light, n
```

The implemented code does not utilize recursive rays, since the modeled material grains are matte in appearance.

A. Ray generation

The rays are in vector form defined as:

$$\mathbf{p}(t) = \mathbf{e} + t(\mathbf{s} - \mathbf{e}) \tag{1}$$

The perspective can be either *orthograpic*, where all viewing rays have the same direction, but different starting points, or use *perspective projection*, where the starting point is the same, but the direction is slightly different [9]. For the purposes of this application, a perspective projection was chosen, as it results in the most natural looking image. The ray data structures were held flexible enough to allow an easy implementation of orthographic perspective, if this is desired at a later point.

The ray origin $\mathbf{e}$ is the position of the eye, and is constant. The direction is unique for each ray, and is computed using:

$$\mathbf{s} - \mathbf{e} = -d\mathbf{w} + u\mathbf{u} + v\mathbf{v} \tag{2}$$

where $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$ are the orthonormal bases of the camera coordinate system, and d is the focal length [9]. The camera coordinates of pixel (i, j) in the image plane, u and v, are calculated by:

$$u = l + (r - l)(i + 0.5)/n$$

$$v = b + (t - b)(j + 0.5)/m$$

where l, r, t and b are the positions of the image borders (left, right, top and bottom) in camera space. The values n and m are the number of pixels in each dimension.

B. Ray-sphere intersection

Given a sphere with a center $\mathbf{c}$, and radius R, a equation can be constrained, where $\mathbf{p}$ are all points placed on the sphere surface:

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0 \tag{3}$$

By substituting the points $\mathbf{p}$ with ray equation 1, and rearranging the terms, a quadratic equation emerges:

$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2\mathbf{d} \cdot (\mathbf{e} - \mathbf{c})t + (\mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - R^2 = 0 \tag{4}$$

The number of ray steps t is the only unknown, so the number of intersections is found by calculating the determinant:

$$\Delta = (2(\mathbf{d} \cdot (\mathbf{e} - \mathbf{c})))^2 - 4(\mathbf{d} \cdot \mathbf{d})((\mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - R^2 \tag{5}$$

A negative value denotes no intersection between the sphere and the ray, a value of zero means that the ray touches the sphere at a single point (ignored in this implementation), and a positive value denotes that there are two intersections, one when the ray enters the sphere, and one when it exits. In the code, a conditional branch checks wether the determinant is positive. If this is the case, the distance to the intersection in ray "steps" is calculated using:

$$t = \frac{-\mathbf{d} \cdot (\mathbf{e} - \mathbf{c}) \pm \sqrt{\eta}}{(\mathbf{d} \cdot \mathbf{d})} \tag{6}$$

where

$$\eta = (\mathbf{d} \cdot (\mathbf{e} - \mathbf{c}))^2 - (\mathbf{d} \cdot \mathbf{d})((\mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - R^2)$$

Only the smallest intersection (t_{minus}) is calculated, since this marks the point where the sphere enters the particle. If this value is smaller than previous intersection distances, the intersection point $\mathbf{p}$ and surface normal $\mathbf{n}$ at the intersection point is calculated:

$$\mathbf{p} = \mathbf{e} + t_{\text{minus}}\mathbf{d} \tag{7}$$

$$\mathbf{n} = 2(\mathbf{p} - \mathbf{c}) \tag{8}$$

The intersection distance in vector steps (t_{minus}) is saved in order to allow comparison of the distance with later intersections.

C. Pixel shading

The pixel is shaded using *Lambertian* shading [9], where the pixel color is proportional to the angle between the light vector ($\mathbf{l}$) and the surface normal. An ambient shading component is added to simulate global illumination, and prevent that the spheres are completely black:

$$L = k_a I_a + k_d I_d \max(0, (\mathbf{n} \cdot \mathbf{l})) \tag{9}$$

where the a and d subscripts denote the ambient and diffusive (Lambertian) components of the ambient/diffusive coefficients (k) and light intensities (I). The pixel color L is calculated once per color channel.

D. Computational implementation

The above routines were first implemented in CUDA for device execution, and afterwards ported to a CPU C++ equivalent, used for comparing performance. The CPU raytracing algorithm was optimized to shared-memory parallelism using OpenMP [10]. The execution method can be chosen when launching the raytracer from the command line, see the appendix for details. In the CPU implementation, all data was stored in linear arrays of the right size, ensuring 100% memory efficiency.

III. CUDA IMPLEMENTATION

When constructing the algorithm for execution on the GPGPU device, the data-parallel nature of the problem (SIMD: single instruction, multiple data) is used to deconstruct the rendering task into a single thread per pixel. Each thread iterates through all particles, and ends up writing the resulting color to the image memory.

The application starts by reading the discrete element method data from a custom binary file. The particle data, consisting of position vectors in three-dimensional Euclidean space ($\mathbf{R}^3$) and particle radii, is stored together in a `float4` array, with the particle radius in the `w` position. This has

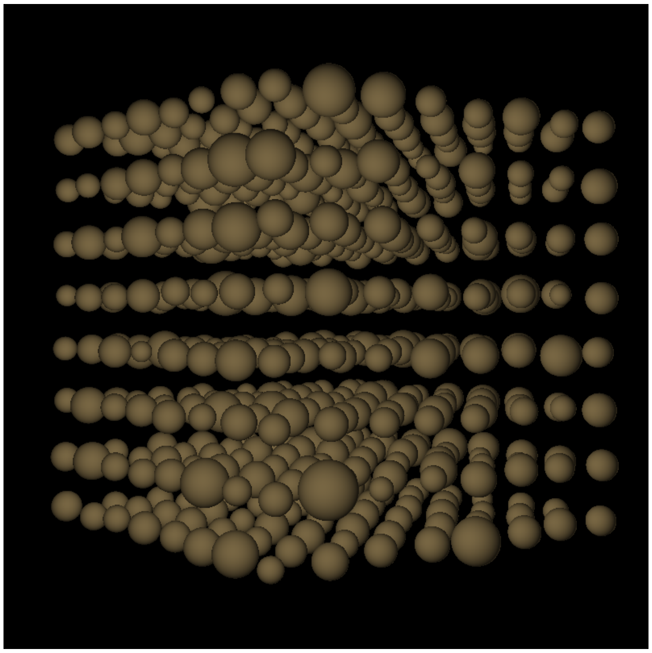

Fig. 1. Sample output of GPU raytracer rendering of 512 particles.

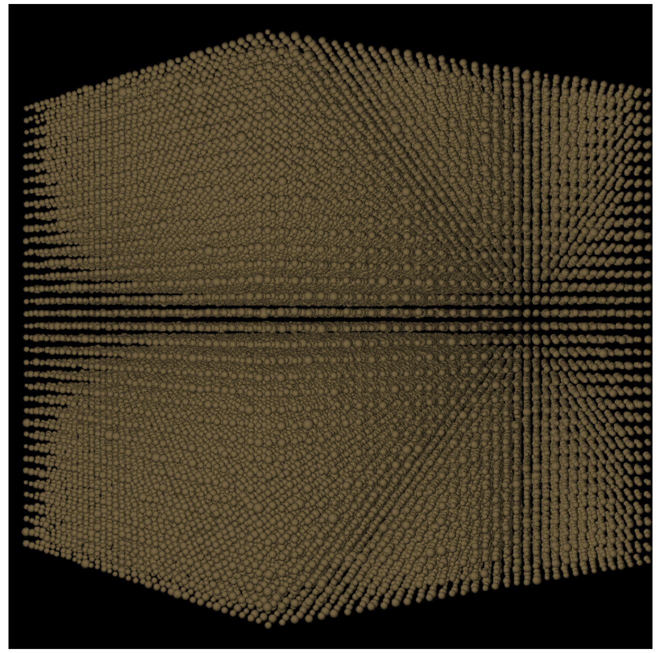

Fig. 2. Sample output of GPU raytracer rendering of 50 653 particles.

large advantages to storing the data in separate `float3` and `float` arrays; Using `float4` (instead of `float3`) data allows coalesced memory access [11] to the arrays of data in device memory, resulting in efficient memory requests and transfers [12], and the data access pattern is coherent and convenient. Other three-component vectors were also stored as `float4` for the same reasons, even though this sometimes caused a slight memory redundancy. The image data is saved in a three-channel linear `unsigned char` array. Global memory access are coalesced whenever possible. Divergent branches in the kernel code were avoided as much as possible [11].

The algorithm starts by allocating memory on the device for the particle data, the ray parameters, and the image RGB values. Afterwards, all particle data is transfered from the host- to the device memory.

All pixel values are initialized to $[R, G, B] = [0, 0, 0]$, which serves as the image background color. Afterwards, a kernel is executed with a thread for each pixel, testing for intersections between the pixel's viewing ray and all particles, and returning the closest particle. This information is used when computing the shading of the pixel.

After all pixel values have been computed, the image data is transfered back to the host memory, and written to the disk. The application ends by liberating dynamically allocated memory on both the device and the host.

A. Thread and block layout

The thread/block layout passed during kernel launches is arranged in the following manner:

```
dim3 threads(16, 16);
dim3 blocks((width+15)/16, (height+15)/16);
```

The image pixel position of the thread can be determined from the thread- and block index and dimensions. The layout corresponds to a thread tile size of 256, and a dynamic number of blocks, ensured to fit the image dimensions with only small eventual redundancy [13]. Since this method will initialize extra threads in most situations, all kernels (with return type `void`) start by checking wether the thread-/block index actually falls inside of the image dimensions:

```
int i = threadIdx.x + blockIdx.x * ↩
    blockDim.x;
int j = threadIdx.y + blockIdx.y * ↩
    blockDim.y;
unsigned int mempos = x + y * blockDim.x ↩
    * gridDim.x;
if (mempos > pixels)
  return;
```

The linear memory position (`mempos`) is used as the index when reading or writing to the linear arrays residing in global device memory.

B. Image output

After completing all pixel shading computations on the device, the image data is transfered back to the host memory, and together with a header written to a PPM[3] image file. This file is converted to the PNG format using ImageMagick.

C. Performance

Since this simple raytracing algorithm generates a single non-recursive ray for each pixel, which in turn checks all spheres for intersection, the application is expected to scale in the form of $O(n \times m \times N)$, where n and m are the output image dimensions in pixels, and N is the number of particles.

[3] http://paulbourke.net/dataformats/ppm/

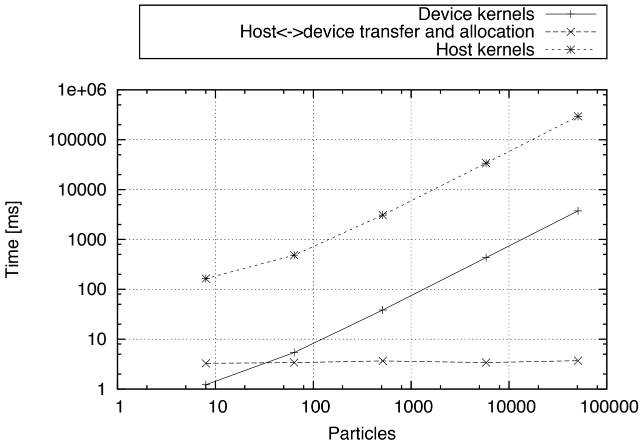

Fig. 3. Performance scaling with varying particle numbers at image dimensions 800 by 800 pixels.

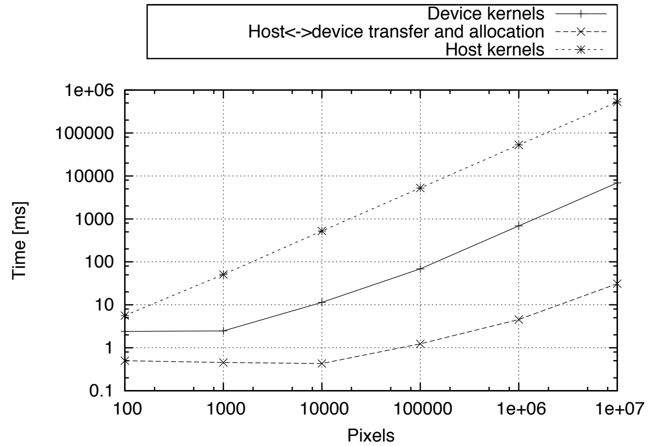

Fig. 4. Performance scaling with varying image dimensions ($n \times m$) with 5832 particles.

The data transfer between the host and device is kept at a bare minimum, as the intercommunication is considered a bottleneck in relation to the potential device performance[11]. Thread synchronization points are only inserted were necessary, and the code is optimized by the compilers to the target architecture (see appendix).

The host execution time was profiled using a `clock()` based CPU timer from `time.h`, which was normalized using the constant `CLOCKS_PER_SEC`.

The device execution time was profiled using two `cudaEvent_t` timers, one measuring the time spent in the entire device code section, including device memory allocation, data transfer to- and from the host, execution of the kernels, and memory deallocation. The other timer only measured time spent in the kernels. The threads were synchronized before stopping the timers. A simple CPU timer using `clock()` will *not* work, since control is returned to the host thread before the device code has completed all tasks.

Figures 3 and 4 show the profiling results, where the number of particles and the image dimensions were varied. With exception of executions with small image dimensions, the kernel execution time results agree with the $O(n \times m \times N)$ scaling predicion.

The device memory allocation and data transfer was also profiled, and turns out to be only weakly dependant on the particle numbers (fig. 3), but more strongly correlated to image dimensions (fig. 4). As with kernel execution times, the execution time converges against an overhead value at small image dimensions.

The CPU time spent in the host kernels proves to be linear with the particle numbers, and linear with the image dimensions. This is due to the non-existant overhead caused by initialization of the device code, and reduced memory transfer.

The ratio between CPU computational times and the sum of the device kernel execution time and the host—device memory transfer and additional memory allocation was calculated, and had a mean value of 55.6 and a variance of 739 out of the 11 comparative measurements presented in the figures. It should be noted, that the smallest speedups were recorded when using very small image dimensions, probably unrealistic in real use.

As the number of particles are not known by compilation, it is not possible to store particle positions and -radii in constant memory. Shared memory was also on purpose avoided, since the memory per thread block (64 kb) would not be sufficient in rendering of simulations containing containing more than $16\,000$ particles ($16\,000$ `float4` values). The constant memory was however utilized for storing the camera related parameters; the orthonormal base vectors, the observer position, the image dimensions, the focal length, and the light vector.

Previous GPU implementations often rely on k-D trees, constructed as an sorting method for static scene objects[3], [5]. A k-D tree implementation would drastically reduce the global memory access induced by each thread, so it is therefore the next logical step with regards to optimizing the ray tracing algorithm presented here.

IV. Conclusion

This document presented the implementation of a basic ray tracing algorithm, utilizing the highly data-parallel nature of the problem when porting the work load to CUDA. Performance tests showed the expected, linear correlation between image dimensions, particle numbers and execution time. Comparisons with an equivalent CPU algorithm showed large speedups, typically up to two orders of magnitude. This speedup did not come at a cost of less correct results.

The final product will come into good use during further development and completion of the CUDA DEM particle model, and is ideal since it can be used for offline rendering on dedicated, heterogeneous GPU-CPU computing nodes. The included device code will be the prefered method of execution, whenever the host system allows it.

References

[1] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[2] NVIDIA, *CUDA C Programming Guide*, 3rd ed., NVIDIA Corporation: Santa Clara, CA, USA, Oct 2010.

[3] D. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-D tree GPU raytracing," *Association for Computing Machinery, Inc.*, pp. 167–174 pp., 2007.

[4] M. Shih, Y. Chiu, Y. Chen, and C. Chang, "Real-time ray tracing with cuda," *Algorithms and Architectures for Parallel Processing*, pp. 327–337, 2009.

[5] S. Popov, J. Günther, H. Seidel, and P. Slusallek, "Stackless kd-tree traversal for high performance gpu ray tracing," in *Computer Graphics Forum*, vol. 26, no. 3. Wiley Online Library, 2007, pp. 415–424.

[6] D. Luebke and S. Parker, "Interactive ray tracing with cuda," *NVIDIA Technical Presentation, SIGGRAPH*, 2008.

[7] D. Evans, E. Phillips, J. Hiemstra, and C. Auton, "Subglacial till: formation, sedimentary characteristics and classification," *Earth-Science Reviews*, vol. 78, no. 1-2, pp. 115–176, 2006.

[8] P. Cundall and O. Strack, "A discrete numerical model for granular assemblies," *Géotechnique*, vol. 29, pp. 47–65, 1979.

[9] P. Shirley, M. Ashikhmin, M. Gleicher, S. Marschner, E. Reinhard, K. Sung, W. Thompson, and P. Willemsen, *Fundamentals of computer graphics*, 3rd ed. AK Peters, 2009.

[10] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. The MIT Press, 2007, vol. 10.

[11] NVIDIA, *CUDA C Best Practices Guide Version*, 3rd ed., NVIDIA Corporation: Santa Clara, CA, USA, Aug 2010, cA Patent 95,050.

[12] L. Nyland, M. Harris, and J. Prins, "Fast n-body simulation with cuda," *GPU gems*, vol. 3, pp. 677–695, 2007.

[13] J. Sanders and E. Kandrot, *CUDA by example*. Addison-Wesley, 2010.

APPENDIX A
TEST ENVIRONMENT

The raytracing algorithm was developed, tested and profiled on a mid 2010 Mac Pro with a 2.8 Ghz Quad-Core Intel Xeon CPU and a NVIDIA Quadro 4000 for Mac, dedicated to CUDA applications. The CUDA driver was version 4.0.50, the CUDA compilation tools release 4.0, V0.2.1221. The GCC tools were version 4.2.1. Each CPU core is multithreaded by two threads for a total of 8 threads.

The CUDA source code was compiled with `nvcc`, and linked to `g++` compiled C++ source code with `g++`. For all benchmark tests, the code was compiled with the following commands:

```
g++ -c -Wall -O3 -arch x86_64 -fopenmp ...
nvcc -c -use_fast_math -gencode ↩
    arch=compute_20,code=sm_20 -m64 ...
g++ -arch x86_64 -lcuda -lcudart -fopenmp ↩
    *.o -o rt
```

When profiling device code performance, the application was executed two times, and the time of the second run was noted. This was performed to avoid latency caused by device driver initialization.

The host system was measured to have a memory bandwidth of 4642.1 MB/s when transfering data from the host to the device, and 3805.6 MB/s when transfering data from the device to the host.

APPENDIX B
SOURCE CODE

The entire source code, as well as input data files, can be found in the following archive http://users-cs.au.dk/adc/files/sphere-rt.tar.gz The source code is built and run with the commands:

```
make
make run
```

With the `make run` command, the Makefile uses ImageMagick to convert the PPM file to PNG format, and the OS X command `open` to display the image. Other input data files are included with other particle number magnitudes. The syntax for the raytracer application is the following:

```
./rt <CPU | GPU> <sphere-binary.bin> ↩
    <width> <height> <output-image.ppm>
```

This appendix contains the following source code files:

LISTINGS

1 rt_kernel.h 5
2 rt_kernel.cu 5
3 rt_kernel_cpu.h 7
4 rt_kernel_cpu.cpp 7

A. *CUDA raytracing source code*

Listing 1. rt_kernel.h

```
#ifndef RT_KERNEL_H_
#define RT_KERNEL_H_

#include <vector_functions.h>

// Constants
__constant__ float4 const_u;
__constant__ float4 const_v;
__constant__ float4 const_w;
__constant__ float4 const_eye;
__constant__ float4 const_imgplane;
__constant__ float  const_d;
__constant__ float4 const_light;


// Host prototype functions

extern "C"
void cameraInit(float4 eye, float4 lookat, float ↩
    imgw, float hw_ratio);

extern "C"
void checkForCudaErrors(const char* ↩
    checkpoint_description);

extern "C"
int rt(float4* p, const unsigned int np,
       rgb* img, const unsigned int width, const ↩
           unsigned int height,
       f3 origo, f3 L, f3 eye, f3 lookat, float imgw);

#endif
```

Listing 2. rt_kernel.cu

```
#include <iostream>
#include <cutil_math.h>
#include "header.h"
#include "rt_kernel.h"

__inline__ __host__ __device__ float3 f4_to_f3(float4 ↩
    in)
{
  return make_float3(in.x, in.y, in.z);
}

__inline__ __host__ __device__ float4 f3_to_f4(float3 ↩
    in)
{
  return make_float4(in.x, in.y, in.z, 0.0f);
}

// Kernel for initializing image data
__global__ void imageInit(unsigned char* _img, ↩
    unsigned int pixels)
{
  // Compute pixel position from threadIdx/blockIdx
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  unsigned int mempos = x + y * blockDim.x * gridDim.x;
  if (mempos > pixels)
    return;
```

```

    _img[mempos*4]     = 0;   // Red channel
    _img[mempos*4 + 1] = 0;   // Green channel
    _img[mempos*4 + 2] = 0;   // Blue channel
}

// Calculate ray origins and directions
__global__ void rayInitPerspective(float4* _ray_origo,
                                   float4* _ray_direction,
                                   float4 eye,
                                   unsigned int width,
                                   unsigned int height)
{
  // Compute pixel position from threadIdx/blockIdx
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = threadIdx.y + blockIdx.y * blockDim.y;
  unsigned int mempos = i + j * blockDim.x * gridDim.x;
  if (mempos > width*height)
    return;

  // Calculate pixel coordinates in image plane
  float p_u = const_imgplane.x + (const_imgplane.y - const_imgplane.x)
              * (i + 0.5f) / width;
  float p_v = const_imgplane.z + (const_imgplane.w - const_imgplane.z)
              * (j + 0.5f) / height;

  // Write ray origo and direction to global memory
  _ray_origo[mempos]     = const_eye;
  _ray_direction[mempos] = -const_d*const_w + p_u*const_u + p_v*const_v;
}

// Check wether the pixel's viewing ray intersects with the spheres,
// and shade the pixel correspondingly
__global__ void rayIntersectSpheres(float4* _ray_origo,
                                    float4* _ray_direction,
                                    float4* _p,
                                    unsigned char* _img,
                                    unsigned int pixels,
                                    unsigned int np)
{
  // Compute pixel position from threadIdx/blockIdx
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  unsigned int mempos = x + y * blockDim.x * gridDim.x;
  if (mempos > pixels)
    return;

  // Read ray data from global memory
  float3 e = f4_to_f3(_ray_origo[mempos]);
  float3 d = f4_to_f3(_ray_direction[mempos]);
  //float  step = length(d);

  // Distance, in ray steps, between object and eye initialized with a large value
  float tdist = 1e10f;

  // Surface normal at closest sphere intersection
  float3 n;

  // Intersection point coordinates
  float3 p;

  // Iterate through all particles
  for (unsigned int i=0; i<np; i++) {

    // Read sphere coordinate and radius
    float3 c = f4_to_f3(_p[i]);
    float  R = _p[i].w;

    // Calculate the discriminant: d = B^2 - 4AC
    float Delta = (2.0f*dot(d,(e-c)))*(2.0f*dot(d,(e-c)))  // B^2
                  - 4.0f*dot(d,d)   // -4*A
                  * (dot((e-c),(e-c)) - R*R);  // C

    // If the determinant is positive, there are two solutions
    // One where the line enters the sphere, and one where it exits
    if (Delta > 0.0f) {

      // Calculate roots, Shirley 2009 p. 77
      float t_minus = ((dot(-d,(e-c)) - sqrt( dot(d,(e-c))*dot(d,(e-c)) - dot(d,d)
                      * (dot((e-c),(e-c)) - R*R) ) ) / dot(d,d));

      // Check wether intersection is closer than previous values
      if (fabs(t_minus) < tdist) {
    p = e + t_minus*d;
    tdist = fabs(t_minus);
    n = normalize(2.0f * (p - c));   // Surface normal
      }

    } // End of solution branch

  } // End of particle loop

  // Write pixel color
  if (tdist < 1e10) {

    // Lambertian shading parameters
    float dotprod = fabs(dot(n, f4_to_f3(const_light)));
    float I_d = 40.0f;  // Light intensity
    float k_d = 5.0f;  // Diffuse coefficient

    // Ambient shading
    float k_a = 10.0f;
    float I_a = 5.0f;

    // Write shading model values to pixel color channels
    _img[mempos*4]     = (unsigned char) ((k_d * I_d * dotprod
                         + k_a * I_a)*0.48f);
    _img[mempos*4 + 1] = (unsigned char) ((k_d * I_d * dotprod
                    + k_a * I_a)*0.41f);
    _img[mempos*4 + 2] = (unsigned char) ((k_d * I_d * dotprod
                    + k_a * I_a)*0.27f);
  }
}


extern "C"
__host__ void cameraInit(float4 eye, float4 lookat, float imgw, float hw_ratio)
{
  // Image dimensions in world space (l, r, b, t)
  float4 imgplane = make_float4(-0.5f*imgw, 0.5f*imgw, -0.5f*imgw*hw_ratio, 0.5f*imgw*hw_ratio);

  // The view vector
  float4 view = eye - lookat;

  // Construct the camera view orthonormal base
  float4 v = make_float4(0.0f, 1.0f, 0.0f, 0.0f);  // v: Pointing upward
  float4 w = -view/length(view);           // w: Pointing backwards
  float4 u = make_float4(cross(make_float3(v.x, v.y, v.z),
                         make_float3(w.x, w.y, w.z)),
         0.0f); // u: Pointing right

  // Focal length 20% of eye vector length
  float d = length(view)*0.8f;

  // Light direction (points towards light source)
  float4 light = normalize(-1.0f*eye*make_float4(1.0f, 0.2f, 0.6f, 0.0f));

  std::cout << "  Transfering camera values to constant memory\n";

  cudaMemcpyToSymbol("const_u", &u, sizeof(u));
  cudaMemcpyToSymbol("const_v", &v, sizeof(v));
  cudaMemcpyToSymbol("const_w", &w, sizeof(w));
  cudaMemcpyToSymbol("const_eye", &eye, sizeof(eye));
  cudaMemcpyToSymbol("const_imgplane", &imgplane, sizeof(imgplane));
  cudaMemcpyToSymbol("const_d", &d, sizeof(d));
```

```
    cudaMemcpyToSymbol("const_light", &light, sizeof(light));
}

// Check for CUDA errors
extern "C"
__host__ void checkForCudaErrors(const char* checkpoint_description)
{
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        std::cout << "\nCuda error detected, checkpoint: " << checkpoint_description
                  << "\nError string: " << cudaGetErrorString(err) << "\n";
        exit(EXIT_FAILURE);
    }
}


// Wrapper for the rt kernel
extern "C"
__host__ int rt(float4* p, unsigned int np,
                rgb* img, unsigned int width, unsigned int height,
        f3 origo, f3 L, f3 eye, f3 lookat, float imgw) {

    using std::cout;

    cout << "Initializing CUDA:\n";

    // Initialize GPU timestamp recorders
    float t1, t2;
    cudaEvent_t t1_go, t2_go, t1_stop, t2_stop;
    cudaEventCreate(&t1_go);
    cudaEventCreate(&t2_go);
    cudaEventCreate(&t2_stop);
    cudaEventCreate(&t1_stop);

    // Start timer 1
    cudaEventRecord(t1_go, 0);

    // Allocate memory
    cout << "  Allocating device memory\n";
    static float4 *_p;              // Particle positions (x,y,z) and radius (w)
    static unsigned char *_img;     // RGBw values in image
    static float4 *_ray_origo;      // Ray origo (x,y,z)
    static float4 *_ray_direction;  // Ray direction (x,y,z)
    cudaMalloc((void**)&_p, np*sizeof(float4));
    cudaMalloc((void**)&_img, width*height*4*sizeof(unsigned char));
    cudaMalloc((void**)&_ray_origo, width*height*sizeof(float4));
    cudaMalloc((void**)&_ray_direction, width*height*sizeof(float4));

    // Transfer particle data
    cout << "  Transfering particle data: host -> device\n";
    cudaMemcpy(_p, p, np*sizeof(float4), cudaMemcpyHostToDevice);

    // Check for errors after memory allocation
    checkForCudaErrors("CUDA error after memory allocation");

    // Arrange thread/block structure
    unsigned int pixels = width*height;
    float hw_ratio = (float)height/(float)width;
    dim3 threads(16,16);
    dim3 blocks((width+15)/16, (height+15)/16);

    // Start timer 2
    cudaEventRecord(t2_go, 0);

    // Initialize image to background color
    imageInit<<< blocks, threads >>>(_img, pixels);

    // Initialize camera
    cameraInit(make_float4(eye.x, eye.y, eye.z, 0.0f),
               make_float4(lookat.x, lookat.y, lookat.z, 0.0f),
               imgw, hw_ratio);
    checkForCudaErrors("CUDA error after cameraInit");

    // Construct rays for perspective projection
    rayInitPerspective<<< blocks, threads >>>(
        _ray_origo, _ray_direction,
        make_float4(eye.x, eye.y, eye.z, 0.0f),
        width, height);

    // Find closest intersection between rays and spheres
    rayIntersectSpheres<<< blocks, threads >>>(
        _ray_origo, _ray_direction,
        _p, _img, pixels, np);


    // Make sure all threads are done before continuing CPU control sequence
    cudaThreadSynchronize();

    // Check for errors
    checkForCudaErrors("CUDA error after kernel execution");

    // Stop timer 2
    cudaEventRecord(t2_stop, 0);
    cudaEventSynchronize(t2_stop);

    // Transfer image data from device to host
    cout << "  Transfering image data: device -> host\n";
    cudaMemcpy(img, _img, width*height*4*sizeof(unsigned char), cudaMemcpyDeviceToHost);

    // Free dynamically allocated device memory
    cudaFree(_p);
    cudaFree(_img);
    cudaFree(_ray_origo);
    cudaFree(_ray_direction);

    // Stop timer 1
    cudaEventRecord(t1_stop, 0);
    cudaEventSynchronize(t1_stop);

    // Calculate time spent in t1 and t2
    cudaEventElapsedTime(&t1, t1_go, t1_stop);
    cudaEventElapsedTime(&t2, t2_go, t2_stop);

    // Report time spent
    cout << "  Time spent on entire GPU routine: "
         << t1 << " ms\n";
    cout << "  - Kernels: " << t2 << " ms\n"
         << "  - Memory alloc. and transfer: " << t1-t2 << " ms\n";

    // Return successfully
    return 0;
}
```

B. CPU raytracing source code

Listing 3. rt_kernel_cpu.h

```
#ifndef RT_KERNEL_CPU_H_
#define RT_KERNEL_CPU_H_

#include <vector_functions.h>

// Host prototype functions

void cameraInit(float3 eye, float3 lookat, float imgw, float hw_ratio);

int rt_cpu(float4* p, const unsigned int np,
       rgb* img, const unsigned int width, const unsigned int height,
       f3 origo, f3 L, f3 eye, f3 lookat, float imgw);

#endif
```

Listing 4. rt_kernel_cpu.cpp

```
#include <iostream>
#include <cstdio>
#include <cmath>
#include <time.h>
#include <cuda.h>
```

```
#include <cutil_math.h>
#include <string.h>
#include "header.h"
#include "rt_kernel_cpu.h"

// Constants
float3 constc_u;
float3 constc_v;
float3 constc_w;
float3 constc_eye;
float4 constc_imgplane;
float  constc_d;
float3 constc_light;

__inline__ float3 f4_to_f3(float4 in)
{
  return make_float3(in.x, in.y, in.z);
}

__inline__ float4 f3_to_f4(float3 in)
{
  return make_float4(in.x, in.y, in.z, 0.0f);
}

__inline__ float lengthf3(float3 in)
{
  return sqrt(in.x*in.x + in.y*in.y + in.z*in.z);
}

// Kernel for initializing image data
void imageInit_cpu(unsigned char* _img, unsigned int pixels)
{
  for (unsigned int mempos=0; mempos<pixels; mempos++) {
    _img[mempos*4]     = 0; // Red channel
    _img[mempos*4 + 1] = 0; // Green channel
    _img[mempos*4 + 2] = 0; // Blue channel
  }
}

// Calculate ray origins and directions
void rayInitPerspective_cpu(float3* _ray_origo,
                float3* _ray_direction,
            float3 eye,
                         unsigned int width,
            unsigned int height)
{
  int i;
  #pragma omp parallel for
  for (i=0; i<width; i++) {
    for (unsigned int j=0; j<height; j++) {

      unsigned int mempos = i + j*height;

      // Calculate pixel coordinates in image plane
      float p_u = constc_imgplane.x + (constc_imgplane.y - constc_imgplane.x)
    * (i + 0.5f) / width;
      float p_v = constc_imgplane.z + (constc_imgplane.w - constc_imgplane.z)
    * (j + 0.5f) / height;

      // Write ray origo and direction to global memory
      _ray_origo[mempos]     = constc_eye;
      _ray_direction[mempos] = -constc_d*constc_w + p_u*constc_u + p_v*constc_v;
    }
  }
}

// Check wether the pixel's viewing ray intersects with the spheres,
// and shade the pixel correspondingly
void rayIntersectSpheres_cpu(float3* _ray_origo,
                             float3* _ray_direction,
                             float4* _p,
            unsigned char* _img,
            unsigned int pixels,
            unsigned int np)
{
  int mempos;
  #pragma omp parallel for
  for (mempos=0; mempos<pixels; mempos++) {

    // Read ray data from global memory
    float3 e = _ray_origo[mempos];
    float3 d = _ray_direction[mempos];
    //float  step = lengthf3(d);

    // Distance, in ray steps, between object and eye initialized with a large value
    float tdist = 1e10f;

    // Surface normal at closest sphere intersection
    float3 n;

    // Intersection point coordinates
    float3 p;

    // Iterate through all particles
    for (unsigned int i=0; i<np; i++) {

      // Read sphere coordinate and radius
      float3 c = f4_to_f3(_p[i]);
      float  R = _p[i].w;

      // Calculate the discriminant: d = B^2 - 4AC
      float Delta = (2.0f*dot(d,(e-c)))*(2.0f*dot(d,(e-c)))  // B^2
    - 4.0f*dot(d,d)  // -4*A
    * (dot((e-c),(e-c)) - R*R);  // C

      // If the determinant is positive, there are two solutions
      // One where the line enters the sphere, and one where it exits
      if (Delta > 0.0f) {

    // Calculate roots, Shirley 2009 p. 77
    float t_minus = ((dot(-d,(e-c)) - sqrt( dot(d,(e-c))*dot(d,(e-c)) - dot(d,d)
      * (dot((e-c),(e-c)) - R*R) ) ) / dot(d,d));

    // Check wether intersection is closer than previous values
    if (fabs(t_minus) < tdist) {
      p = e + t_minus*d;
      tdist = fabs(t_minus);
      n = normalize(2.0f * (p - c));   // Surface normal
    }

      } // End of solution branch

    } // End of particle loop

    // Write pixel color
    if (tdist < 1e10) {

      // Lambertian shading parameters
      float dotprod = fabs(dot(n, constc_light));
      float I_d = 40.0f;  // Light intensity
      float k_d = 5.0f;  // Diffuse coefficient

      // Ambient shading
      float k_a = 10.0f;
      float I_a = 5.0f;

      // Write shading model values to pixel color channels
      _img[mempos*4]     = (unsigned char) ((k_d * I_d * dotprod
    + k_a * I_a)*0.48f);
      _img[mempos*4 + 1] = (unsigned char) ((k_d * I_d * dotprod
    + k_a * I_a)*0.41f);
      _img[mempos*4 + 2] = (unsigned char) ((k_d * I_d * dotprod
    + k_a * I_a)*0.27f);
    }
  }
}


void cameraInit_cpu(float3 eye, float3 lookat, float imgw, float hw_ratio)
{
  // Image dimensions in world space (l, r, b, t)
  float4 imgplane = make_float4(-0.5f*imgw, 0.5f*imgw, -0.5f*imgw*hw_ratio, 0.5f*imgw*hw_ratio);
```

```

  // The view vector
  float3 view = eye - lookat;

  // Construct the camera view orthonormal base
  float3 v = make_float3(0.0f, 1.0f, 0.0f);  // v: Pointing upward
  float3 w = -view/lengthf3(view);           // w: Pointing backwards
  float3 u = cross(make_float3(v.x, v.y, v.z), make_float3(w.x, w.y, w.z)); // u: Pointing right

  // Focal length 20% of eye vector length
  float d = lengthf3(view)*0.8f;

  // Light direction (points towards light source)
  float3 light = normalize(-1.0f*eye*make_float3(1.0f, 0.2f, 0.6f));

  std::cout << "  Transfering camera values to constant memory\n";

  constc_u = u;
  constc_v = v;
  constc_w = w;
  constc_eye = eye;
  constc_imgplane = imgplane;
  constc_d = d;
  constc_light = light;
}


// Wrapper for the rt algorithm
int rt_cpu(float4* p, unsigned int np,
       rgb* img, unsigned int width, unsigned int height,
       f3 origo, f3 L, f3 eye, f3 lookat, float imgw) {

  using std::cout;

  cout << "Initializing CPU raytracer:\n";

  // Initialize GPU timestamp recorders
  float t1_go, t2_go, t1_stop, t2_stop;

  // Start timer 1
  t1_go = clock();

  // Allocate memory
  cout << "  Allocating device memory\n";
  static unsigned char *_img;        // RGBw values in image
  static float3* _ray_origo;         // Ray origo (x,y,z)
  static float3* _ray_direction;     // Ray direction (x,y,z)
  _img        = new unsigned char[width*height*4];
  _ray_origo     = new float3[width*height];
  _ray_direction = new float3[width*height];

  // Arrange thread/block structure
  unsigned int pixels = width*height;
  float hw_ratio = (float)height/(float)width;

  // Start timer 2
  t2_go = clock();

  // Initialize image to background color
  imageInit_cpu(_img, pixels);

  // Initialize camera
  cameraInit_cpu(make_float3(eye.x, eye.y, eye.z),
                 make_float3(lookat.x, lookat.y, lookat.z),
            imgw, hw_ratio);

  // Construct rays for perspective projection
  rayInitPerspective_cpu(
      _ray_origo, _ray_direction,
      make_float3(eye.x, eye.y, eye.z),
      width, height);

  // Find closest intersection between rays and spheres
  rayIntersectSpheres_cpu(
      _ray_origo, _ray_direction,
      p, _img, pixels, np);

  // Stop timer 2
  t2_stop = clock();

  memcpy(img, _img, sizeof(unsigned char)*pixels*4);

  // Free dynamically allocated device memory
  delete [] _img;
  delete [] _ray_origo;
  delete [] _ray_direction;

  // Stop timer 1
  t1_stop = clock();

  // Report time spent
  cout << "  Time spent on entire CPU raytracing routine: "
       << (t1_stop-t1_go)/CLOCKS_PER_SEC*1000.0 << " ms\n";
  cout << "  - Functions: " << (t2_stop-t2_go)/CLOCKS_PER_SEC*1000.0 << " ms\n";

  // Return successfully
  return 0;
}
```

APPENDIX C
BLOOPERS

This section contains pictures of the unfinished raytracer in a malfunctioning state, which can provide interesting, however unusable, results.

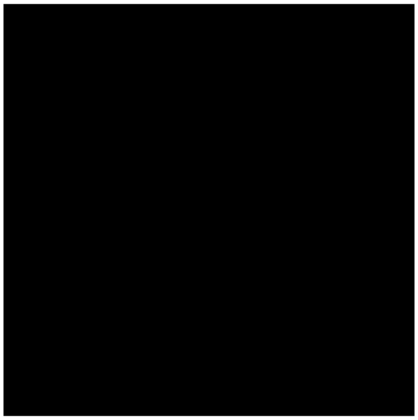

Fig. 5. An end result encountered way too many times, e.g. after inadvertently dividing the focal length with zero.

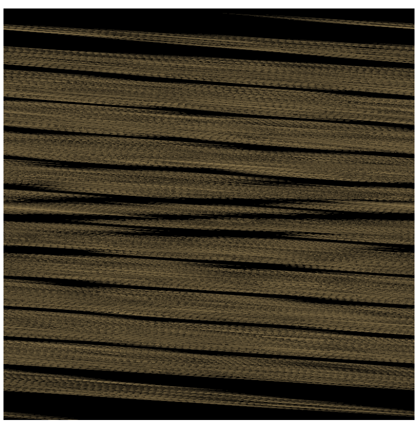

Fig. 6. Twisted appearance caused by wrong thread/block layout.

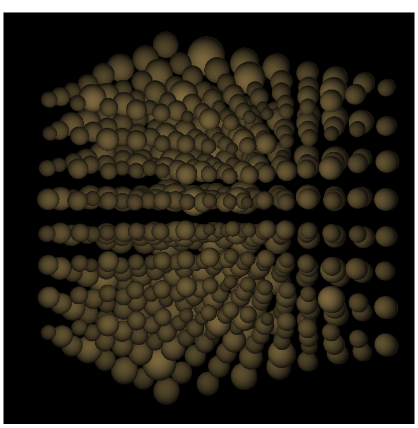

Fig. 7. A problem with the distance determination caused the particles to be displayed in a wrong order.

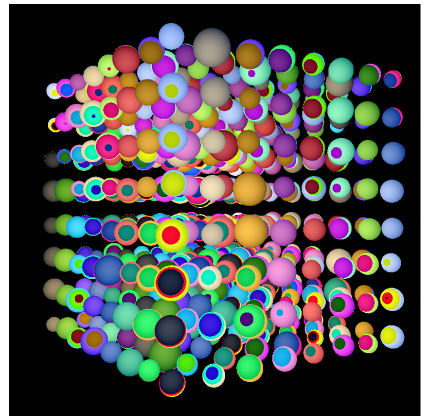

Fig. 8. Colorful result caused by forgetting to normalize the normal vector calculation.

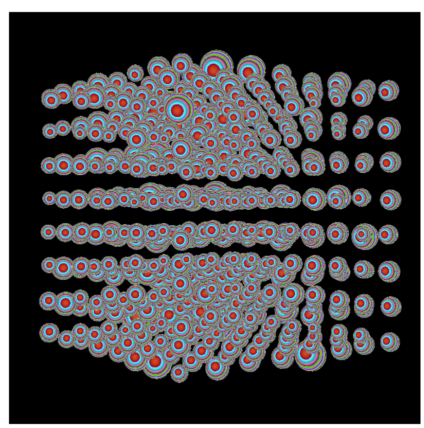

Fig. 9. Another colorful result with wrong order of particles, cause by using `abs()` (meant for integers) instead of `fabs()` (meant for floats).