Particle Simulation with GPUs Shading Languages

Francisco A. Madera[1], Francisco Moo-Mena[2], Enrique Ayala[3], Luis F. Curi[4]

[1,2,3,4] Facultad de Matemáticas, Universidad Autónoma de Yucatán
Mérida, Yucatán 97110, México

Abstract

The usage of GPUs (Graphics Processing Units) in graphics is essential due to their parallel feature to perform operations. In this work we implement a particle simulation program using shading languages to improve the interaction among particles. We implement a parallel algorithm to detect collisions among animated particles. Particles are stored in a linear buffer with several threads using the Compute Shader (CS). To parallelize the process data are arranged in workgroups with specified size, which represent the threads. We compare the performance of the algorithm implemented in shading languages against the sequential version. We also demonstrate how programmable GPUs are a powerful tool to display large point datasets (particles) at interactive frame rate. We research the possibilities that GPUs and shading languages offer for rendering particles and the improvements in speed and quality using instance rendering.

Keywords: *Particle Simulation, Parallel Programming, GPU rendering.*

1. Introduction

Particles are utilized in computer graphics to simulate fluids, smoke, fire, etc. Scientists usually need huge datasets of point data to understand the behavior of phenomena. Nevertheless, the computational cost of visualizing such huge datasets is high and scientists have to work with samples or fragments of the datasets. There are hundreds of implementations of particle systems for interactive applications and kinematics visual effects systems, but most implementation details are unpublished.

Particle-based datasets are usually generated by numeric simulations, like molecular dynamics, and in everyday scenarios they often consist of up to several million particles and several thousand of time steps. Although such datasets do not necessarily exceed the resources of single PCs, many existing tools are hardly able to cope with data of such size interactively.

In a particle system each point is given as a set of properties, which influences the particles behavior in space and time. In point-based systems however the points are just a representation of characteristics of the data, without carrying any specific information. Particle systems iteratively execute two stages. First, it executes the simulation stage, updating the particles properties. The second stage is the rendering stage, which allows different types of visualizations.

Point data's main characteristic is usually its position in space. In addition to this, point data can have more additional attributes which can be shown in visual terms such as size, color, direction or shape. Therefore, point data needs to be represented visually by geometrical shapes. The animation process (first stage) in computer graphics involves motion, collision detection and collision response among the objects. In our simulation the objects are the particles and we implement the motion and collision detection algorithms in GPUs.

The parallelization of algorithms is a common technique to speed up procedures. There is a variety of techniques to parallelize a sequential process, using CPUs, GPUs, or any other architecture. In this work we implement a simulation of particles using GPUs, focusing on the arrangements of threads by varying the number of workgroups and the size of the workgroups in the GPU.

Our system includes a GPU-accelerated Eulerian solver that is suited for real-time use because it is unconditionally stable, takes constant calculation time per frame, and provides good visual fidelity. We develop an algorithmic framework that will allow adapting as the throughput of consumer parallel processors increases, in particular the threads allowed. Specifically, we investigate the performance of the program by using a linear arrangement to store the particles in the memory of the GPU.

A particle is represented by a sphere and contains a velocity and a position in the scene. By using the Euler numerical method we can move thousands of spheres in parallel. In the collision detection process, using a brute force method, a sphere requires to be tested against all the other particles, giving a square complexity time. By using the parallelism we reduce the time in depth. Our approach basically arranges the particles in different arrays of threads using the CS. A linear arrangement requires a double cycle for the overlapping test while a squared array

requires a single cycle. The final goal of this paper is to show how utilizing the GPU via shader programming can improve performance over sequential techniques.

Our paper is structured as follows: First, we review previous work that is related to ours. We then describe the process of the proposed previewing system (Section 3). Here we introduce the system's functionality of the parallel method and how this is achieved. In Section 4, the collision detection shader is described, there are two types: sphere-sphere and sphere-plane. The second shader is explained in Section 5, the instance rendering technique. Finally, we analyze the processing and rendering performance of our system in Section 6.

2. Previous Work

Particle-based techniques are used in many applications, from interactive simulation of fluids and smoke for games to astrophysics simulations and molecular dynamics. Recent research has also applied particle methods to soft body and cloth simulation, and there is some hope that one day these techniques will allow an efficient unification of rigid, soft body and fluid simulations where everything can interact with everything else seamlessly. There are two basic types of simulation – Eulerian (grid-based) methods, which calculate the properties of the simulation at a set of fixed points in space, and Lagrangian (particle) methods, which calculate the properties of a set of particles as they move through space.

There has been increasing work towards unified simulation models recently, and point-based methods are well suited to the problem. Müller et al. [1] use a point based representation to model elastic and plastic solids that can topologically deform. Solenthaler et al. [2] described a method for simulating fluids and solids based on smoothed particle hydrodynamics (SPH). Becker et al. [3] improve the rotational invariance of their method using a co-rotated deformation model based on SPH. Martin et al. [4] address issues with degenerate particle configurations using elastons which provide a more accurate measure of local deformation.

By moving all data-intensive computation onto the data-parallel processor, the GPU, we achieve results that will scale nearly linearly with the number of GPU processing cores. The system presented in [5] includes a GPU accelerated Eulerian fluid solver that is suited for real-time use because it is unconditionally stable, takes constant calculation time per frame, and provides good visual fidelity. The Navier-Stokes solver, uses numerical methods that have guaranteed stability and constant per-frame computation time, but are accurate enough to capture visually important flow features. Akinci et al. [6] propose a momentum-conserving two-way coupling method of SPH fluids and arbitrary rigid objects based on hydrodynamic forces.

Granular materials exhibit a large range of interesting macroscopic phenomena, including piling, flow or fracture. Effects such as piling may be well described using large amounts of small rigid bodies, with inter-body frictional contact governing the geometry of piles. Flow, on the other hand, may be well described by incompressible fluid models [7]. The simulation of granular materials has a large impact in the engineering field, for the analysis of terrains and avalanches, and also in the animation industry [8, 15].

Spatial hashing is utilized in [9]. For a used cell without hash collisions, all particles are in the same spatial cell and, hence, the potential neighbors are the same. Particle System Interface is a piece of software designed to perform common tasks related to particle systems for clients, while providing them with a set of parameters whose values can be adjusted to create different particle systems [10]. In [11] authors describe how to efficiently implement a particle system in CUDA, including interactions between particles using a uniform grid data structure.

Multigrid methods are inherently more parallel, Macklin [12] used a hash-grid for particle neighbor finding. A full discussion of grid-based methods is, however, beyond the scope of this paper. By moving all data-intensive computation onto the data-parallel processor, the GPU, we achieve results that will scale nearly linearly with the number of GPU processing cores.

3. Algorithm Outline

Particles are attractive for their simplicity and ease of implementation, while being flexible enough to represent the range of objects we wish to simulate. This section describes how to implement a simple particle system in OpenGL, including particle collisions using a linear data structure. There are three main steps to implement the first stage of a particle system:
1. Integration (motion)
2. Building the data structure
3. Processing collisions

We assume having n_p particles which are placed above the ground; then particles start falling down. The solver calculates the new position and updates the velocity according to equation (1). This new velocity causes the

particle to continue traveling downwards and collides with the ground. The collision response calculates the corrected position to make the particle travel upwards. The first collision occurs with the plane, so that a collision response is performed. Particles can be represented with points, but spheres are the most natural representation of atoms, particles and other data sample, which means a greater acceptance and understanding in scientific research. The fact that it is rotationally invariant is most useful.

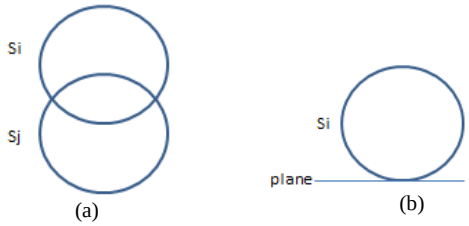


Figure 1. The two collision detection cases, (a) sphere-sphere and (b) sphere-plane.

3.1 Motion

Particle's motion can be simulated by using Euler numerical calculation. To animate the particles, we'll use the standard kinematics equation for objects under constant acceleration [13].

$$P(t) = P_0 + v_0 t + \frac{1}{2} a t^2 \quad (1)$$

The above equation describes the position of a particle at time *t*. P_0 is the initial position, v_0 is the initial velocity, and *a* is the acceleration. The integration step is the simplest step. It integrates the particle attributes (position and velocity) to move the particles through space. We use Euler integration for simplicity; the velocity is updated based on applied forces and gravity, and then the position is updated based on the velocity. We'll update the particle positions incrementally, solving the equations of motion based on the forces involved at the time each frame is rendered. A common technique is to make use of the Euler method, which approximates the position and velocity at time *t* based on the position, velocity, and acceleration at an earlier time.

The Euler method is actually numerically integrating the Newtonian equation of motion. This is one of the simplest techniques for doing so. However, it is a first-order technique, which means that it can introduce a significant amount of error. More accurate techniques include Verlet integration, and Runge Kutta integration. As our particle simulation is designed to look good and physical accuracy is not of high importance, the Euler method should suffice.

3.2 Collision

Collisions considered in the simulation are sphere-plane and sphere-sphere. A sphere *i* is represented as $S_i(C_i, r_i)$, where C_i is the center and r_i the radius. The ground of the scene is represented by a plane XZ with a normal vector <0,1,0>, having $y=h$ (value in the Y-axis). Thus, S_i collides with the plane, if $C_i.x = h$. A collision between spheres S_i and S_j occurs if equation (2) is true. This inequality verifies the squared distance between two spheres, using the 3-vector Euclidean norm, $\| \cdot \|_2$, and requires *11* basic operations.

$$(\|C_i - C_j\|_2)^2 < (r_i - r_j)^2 \quad (2)$$

This way, the sequential process is formed by the motion, the sphere-plane collision detection, and the sphere-sphere collision detection procedures. These procedures are called in every frame, so that particles change their position very often. We also include a collision response when a sphere-plane collision happens, where spheres change their velocity vector in a contrary direction to simulate the bouncing. Thus, a particle requires two vectors for position and velocity, 32-bit and 64-bit IEEE 754 binary floating-point format per float (32x6=192 bits minimum, and 64x6=384 maximum). Code 1 shows the routine to move particles in the scene with the sequential version.

```
0 for(unsigned short i=0; i<NUM_PARTICLES; i++)
  {
1   p=pointSec[i]; v=velSec[i];

2   pp=p; vv=v;

3   pp = p + v*DT + G*0.5*DT*DT;
4   vp = v + G*DT;

5   if(IsInPlane(pp))
    {
6     vp = BouncePlane( p, v, Vector3D( p.x,-5, p.z) );
7     pp = p + vp*DT + G*0.5*DT*DT;
    }

8   pointSec[i] = pp;
9   velSec[i] = vp;

  }
```

Code 1. The loop cycle for the motion and the sphere-plane collision detection particles simulation.

The loop cycle is performed for each sphere. In lines 3 and 4, the Euler motion integration is calculated according to equation (1). Then, the sphere plane collision detection is tested in line 5. If this condition is true, the collision response is called with the function *BouncePlane* (line 6), which change the velocity vector of the sphere to the opposite direction. Sphere plane collision detection (*IsInPlane*) and Sphere collision response (*BouncePlane*) takes one operation so that they can be considered in the loop cycle. Therefore, the loop cycle takes $O(n_p)$ time.

Collision between spheres requires a double loop cycle to test *Si* again *Sj*. The routine is called in line 2 of Code 2 by using equation (2). This process takes $O(n_p^2)$ time. Figure 2 illustrates the diagram of the particle simulation procedures.

```
0 for(unsigned short i=0; i<NUM_PARTICLES; i++)
1   for(unsigned short j=i+1; j<NUM_PARTICLES; j++)
    {
2     if(Colision_Esfera_Esfera(pointSec[i], pointSec[j], 0.5, 0.5))
3     { color_sphere[i]=1; color_sphere[j]=1;
4     NumCol++;
      }

    }
```

Code 2. The sphere-sphere collision detection.

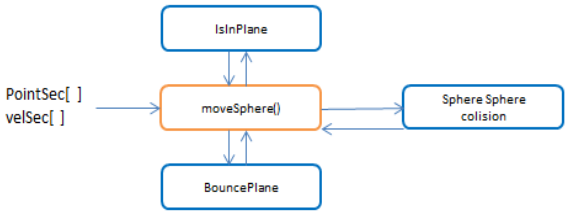


Figure 2. The simulation particle's schematic overview.

4. The First Shader: The Parallel Version

In this section, the implementation in GPU is described. By constructing all objects from particles, we significantly reduce the number of collision types we need to process, and avoid complex algorithms for generating contacts between mesh based representations.

The compute processor (CS) is a programmable unit that operates independently from the other shader processors. It is not part of the graphics pipeline and its visible side effects are through changes to images, storage buffers, and atomic counters. A compute shader operates on a group of work items called a work group. A work group is a collection of shader invocations that execute the same code, potentially in parallel. An invocation within a work group may share data with other members of the same work group through shared variables and issue memory and control barriers to synchronize with other members of the same work group.
Our particle system is optimized for GPU implementation so that all particles may be processed in parallel, with no global communication. Since particles are now independent, the particle simulation loop can be performed in parallel. The particle positions and velocities are both stored in a *float4* array data type as shown in Code 3. The positions are actually allocated in an OpenGL vertex buffer object (VBO) so that they can be rendered from GPU directly. The use of shared memory means that this method does not become bound by memory bandwidth. We employed three buffers, position, velocity, and colors. We use the CS to parallelize the algorithm due to its simplicity to work with graphics in OpenGL. To implemented Code 1 in parallel, we required to arrange the particles in an array, where a thread is responsible to compute the motion and collision of each particle. Different kinds of arrays are available in GLSL (1-array, 2-array, 3-array) to store the data in the GPU memory.

```
struct Particle
{
    float pos[3];
    float vel[3];
    float color[4];
}
```

Code 3. The particle data structure is formed by three attributes: position, velocity, color vector.

There exist some global variables to inform about the threads arrangement. Let be *i* the *i*th element of group *k*, we have the following variables:
gl_GlobalInvocationID.x is the global index of the thread *nk+i*
gl_LocalInvocationID.x is the local index of the thread *i*
gl_WorkGroupID.x is the local index of the thread *k*

A layout statement declares the work group size to be $n \times 1 \times 1$. The code defines gravity (G) and the time step (DT). G is a vec3 (3D vector) so that it can be used in a single line of code to produce the particle's next position. The code runs once for each particle. The variable *gid* is the global ID, that is, the particle's number in the entire list of particles. *gid* indexes into the array of structures. Our approach begins with an unordered set of particle positions. Additional attributes can be associated to each particle, such as a color index, a velocity index, a size indicator, or any other classifier resulting from the application that generates the particles. The threads arrangement is shown in Figure 3.

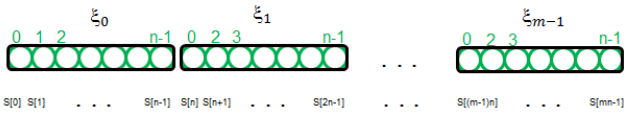


Figure 3. Spheres are arranged in *m* threads, and threads are joined in workgroups of size *n*.

We employed *m* groups of size *n* each, having *mn* particles distributed in a 1-array. A thread works on each particle,

so that operations are performed in parallel for each particle.

At the start, particles are placed on the top, above the ground so that they are falling down. The problem arises when particles collide with the ground, and a collision response must be activated. Since the ground is represented by a plane, it is outside of the threads arrangement. The code equals to the lines 5-7 of Code 1 of the sequential version. Interaction between particles is required to detect collisions. The position of sphere *i* is needed to compute the collision of a pair of spheres using equation (2). Memory shared (Code 4) is allowed for the elements of a group, so that collision among spheres can be performed in the particles of the same group. A group can be stored in an array in parallel, so that each group has their own array called sphereGroup as illustrated in Figure 4.

```
sphereGroup[lid] = Positions[gid].xyz
memorybarrierShared()
```

Code 4. A temporal array, sphereGroup[ ], is utilized in the shared memory to store the elements of the same workgroup.

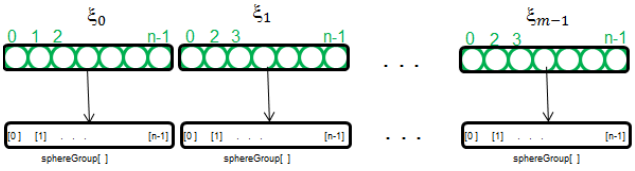


Figure 4. Elements of a workgroup are stored in a temporal array with shared memory.

Then, we can use a loop cycle to access each element of sphereGroup[ ] to interact with each particle. The problem with this case is that there is not interaction between particles of different groups.

To calculate the collision between spheres of different groups, a double loop is required (Code 5). The first loop specifies the group which will be copied to the shared memory. In the first iteration, particles of group *i=0* will be compared with the particles of all the groups. To make this comparison, a nested loop is required. When *j=0*, the first element of each group is compared with all the elements of group *i=0*, in parallel. In the next iteration, when *j=1*, the second element of each group is compared with all the elements of group i=0, in parallel. And this procedure continues until *j=n-1*, when the last element of each group is compared with all the elements of group *i=0*, in parallel. This process is illustrated in Figure 5.

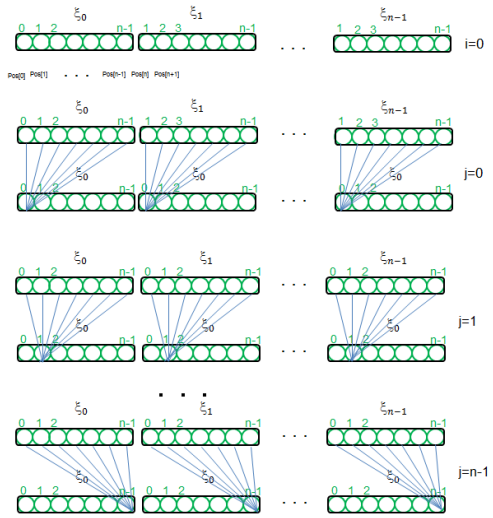


Figure 5. Interaction among all the particles with the particles of the first workgroup.

Thus, the same process is repeated for each group, that is, for *i=0* to *m-1*. The time complexity $O((m-1)n)$ is compared against the sequential version $O(n_p\ n_p)$, reducing dramatically due to *m* and *n* are much less than n_p. This is supported with several trials in the Experiments Section.

```
for (uint i = 0; i <  gl_NumWorkGroups.x; i++)
{
   them = Colours[i * gl_WorkGroupSize.x + lid].x;
   themVec = Positions[ i * gl_WorkGroupSize.x + lid ].xyz;

   ColorGroup[lid] = them;
   sphereGroup[lid] = themVec;

   memoryBarrierShared();
   barrier();
   for (uint j = 0; j < gl_WorkGroupSize.x; j++)
    {
      if(gid != i*gl_WorkGroupSize.x+j)
      if(IsInsideSphere( meVec, vec4( sphereGroup[j].x, sphereGroup[j].y, sphereGroup[j].z, 0.5) ))
          count=1;
    }
    barrier();
}
```

Code 5. A double loop cycle is required for the particles interaction.

As we observed, we only have to store in shared memory one group. The VBO colouring serves to specify the color of each sphere. As this is a vector of 4 floats, the first 3 values serve to specify the color of the sphere and the fourth value to indicate if a collision with another sphere takes place.

5. The Second Shader: Instance Rendering

The direct use of GPU shaders for certain rendering techniques allow programmers to maximize the size of the datasets that their graphic applications can handle. Rendering data at interactive speed requires a considerable

amount of computational power. Consumer graphics hardware nowadays delivers this power through modern graphics processing units. Data can be bound to OpenGL vertex buffer objects for rendering, which allows us to keep all particle data on the GPU without any transfers across the PCI-express bus. Having thousands of particles can reduce the frame rate due to the operations performed. There are different methods to render particles: single points, quadratic shaped spheres, and polygonal shaped spheres.

By using an uploaded buffer containing per-instance data, each instance can then be transformed to its correct location on the GPU. Typical applications that can benefit from hardware instancing include rendering of crowds and vegetation, which usually require a large number of instances at the same time as there exists much repetition.

We can display particles as spheres using the *glutSolidSphere* function from the glut library or displaying the sphere mesh (Code 6), using CPU.

```
for(int i=0; i<NUM_PARTICLES; i++)
{
   glPushMatrix();
   glTranslatef(ptr[i].x,ptr[i].y,ptr[i].z); glutSolidSphere(0.5, 10,10);
   glPopMatrix();
}
```

Code 6. Displaying spheres with *glutSolidSphere*.

Particle-based rendering is originally based on graphical point primitives (i. e. GL_POINT in OpenGL). This approach scales very well for very large numbers of particles, but is usually restricted to simple particle types like point sprites. For increased particle complexity or arrays of mesh instances there are several instancing techniques available [14]. They usually employ VBOs, hardware-supported instancing, or programmable geometry shaders.

For each particle type, the corresponding VBOs (vertices and normal vectors) are activated. To instantiate the particle the VBOs are drawn once (One call of *glDrawArrays* per particle). The transformation from particle-space into object space can be implemented either by using the built-in model view matrix, or by using a simple shading program, which results in better performance due to fewer state changes. Because of the high number of OpenGL function calls, this rendering approach has the highest CPU load. Particles can be represented with points as illustrated in Code 7.

The high number of function calls of the simple VBO-based rendering, which result in high CPU load, can be reduced using hardware-supported instancing. The hardware instancing rendering method, which follows an idea similar to the VBO-based rendering, trades the high CPU load for the additional overhead of requiring the upload of the particle data to the graphics memory as textures.

```
glPointSize(5);
glBindBuffer(GL_ARRAY_BUFFER, posSSBO);
glVertexPointer(4,GL_FLOAT, 0, (void *)0);
glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_POINTS, 0, NUM_PARTICLES);
glDisableClientState(GL_VERTEX_ARRAY);
glBindBuffer( GL_ARRAY_BUFFER, 0 );
```

Code 7. Displaying spheres with *glDrawArrays*, using points.

We can visualize polyhedral (or mesh) shaped particles rather than quadratic surface based sphere. Rendering arbitrary polyhedral shapes is traditionally achieved by means of a polygon mesh. On the one hand, a data set of 1 000 000 particles easily requires up to (NumPolygons x 1 000 000) triangles to be rendered. Code 8 demonstrates the instructions to display the polygons of the sphere mesh. This fact can be exploited to optimize the rendering performance, allowing for interactive visualization of data sets with up to several millions of particles.

```
glBegin(GL_TRIANGLES);
for(int i=0; i<NUM_WORK_GROUPS*1; i++)
for(int k=0; k<p.polygons_qty; k++)
{
      glNormal3f( p.normalv[ p.polygon[k].a ].x, p.normalv[ p.polygon[k].a ].y, p.normalv[ p.polygon[k].a ].z);
      glVertex3f( p.vertex[ p.polygon[k].a ].x+pointSec[i].x, p.vertex[ p.polygon[k].a ].y+pointSec[i].y, p.vertex[ p.polygon[k].a ].z+pointSec[i].z);

      glNormal3f( p.normalv[ p.polygon[k].b ].x, p.normalv[ p.polygon[k].b ].y, p.normalv[ p.polygon[k].b ].z);
      glVertex3f( p.vertex[ p.polygon[k].b ].x+pointSec[i].x, p.vertex[ p.polygon[k].b ].y+pointSec[i].y, p.vertex[ p.polygon[k].b ].z+pointSec[i].z);

      glNormal3f( p.normalv[ p.polygon[k].c ].x, p.normalv[ p.polygon[k].c ].y, p.normalv[ p.polygon[k].c ].z);
      glVertex3f( p.vertex[ p.polygon[k].c ].x+pointSec[i].x, p.vertex[ p.polygon[k].c ].y+pointSec[i].y, p.vertex[ p.polygon[k].c ].z+pointSec[i].z);
}
glEnd();
```

Code 8. Displaying spheres with *GL_TRIANGLES*.

To overcome this drawback, we proceed to construct a shader with instance rendering. This consists on declaring an object, and display in GPU several instances of the same object varying some of its properties: color, position, etc. So we have a scene where we are drawing a lot of models where most of these models contain the same set of vertex data, but with different world transformations.

When drawing many instances of a model like this we quickly reach a performance bottleneck because of the many drawing calls. Compared to rendering the actual vertices, telling the GPU to render the vertex data with functions like *glDrawArrays* or *glDrawElements* eats up quite some performance since OpenGL must make necessary preparations before it can draw your vertex data (like telling the GPU which buffer to read data from, where to find vertex attributes and all this over the relatively slow CPU to GPU bus). So even though rendering your vertices is fast, giving your GPU the commands to render them isn't. It would be much more convenient if we could send some

data over to the GPU once and then tell OpenGL to draw multiple objects with a single drawing call using this data.

Instancing is a technique where we draw many objects at once with a single render call, saving us all the CPU → GPU communications each time we need to render an object; this only has to be done once. To render using instancing all we need to do is change the render calls *glDrawArrays* and *glDrawElements* to *glDrawArraysInstanced* and *glDrawElementsInstanced* respectively.

These instanced versions of the classic rendering functions take an extra parameter called the instance count that sets the number of instances we want to render. We thus sent all the required data to the GPU only once, and then tell the GPU how it should draw all these instances with a single call. The GPU then renders all these instances without having to continually communicate with the CPU.

We use a built-in shader variable called *gl_InstanceID* which, not suprisingly, tells us the current instance index. We can use this index to locate instance specific data in uniform variable arrays. When object is loaded, vertex[ ] and polygons[ ] are stored. For instance rendering we declare three VBOs, the offset, the color, and the normal vectors for each instance. A shader is created, values for each instance is sent from CPU to GPU. Additionally, we utilize some matrices for the affine transformations: Model, View, projection matrices, as shown in Code 9.

```
1  #version 440
2  in vec3 position;
3  in vec3 offset;
4  in vec3 coloring;
5  in vec3 normalv;
6
7  out vec3 theColor;
8  out vec3 LightIntensity;
9
10 uniform mat4 MVP;
11 uniform mat3 NormalMatrix;
12 uniform mat4 ProjectionMatrix;
13 uniform mat4 ModelViewMatrix;
14
15 uniform vec4 LightPosition;
16 uniform vec3 Kd;
17 uniform vec3 Ld;
18
19 void main()
20 {
21       vec3 tnorm = normalize( NormalMatrix * normalv);
22       vec4 eyeCoords = ModelViewMatrix * vec4(position,1.0);
23       vec3 s = normalize(vec3(LightPosition - eyeCoords));
24       LightIntensity = Ld * coloring * max( dot( s, tnorm ), 0.0 );
25       gl_Position = MVP*vec4(position.x+offset.x, position.y+offset.y, position.z+offset.z, 1);
26       theColor = vec3(coloring.x, coloring.y, coloring.z);
27 }
28
29
```

Code 9. The shader program with *instance rendering*.

Having an object represented with a polygonal mesh, we can define its position as the center of all the polygons: γ. The instances to be rendered, $\gamma 0$, $\gamma 1$, … have several values: position, color, normal vector, etc. $\gamma i.pos$, $\gamma i.color$, $\gamma i.normal$. The position is calculated with a displacement of the γ's position, color and normal vectors are calculated for each object. The number of instances must be defined. In the case we employ another object different than a sphere, we should change the collision detection routine. Line 15 of Code 9 shows the instruction to particle position while line 26 calculates the color to be sent to the fragment shader. The other instructions are related to the lighting process.

6. Experiments

To evaluate our algorithm we implement it on an NVidia GeForce 590 GTX using GLSL and CUDA. We tested the code varying the number of particles (depending on the type of particle: point, quadratic surface, mesh object), using different work group sizes. The GeForce GTX 400/500 family of GPUs is based on NVIDIA's Fermi architecture. It has 1024 CUDA cores and 1.5 GB per GPU (it has two of them), with 327.7 GB/sec of memory bandwidth. We tested the time of the two shaders, the Compute Shader (CS), and the instance rendering (IRS). For the CS, we tested the number of colliding spheres, the time required to test both the sphere-sphere collision detection, and the sphere-plane collision detection. For the IRS, we test the time required and its comparison with the sequential version displaying routines.

We prepared two animations, according to their start position and velocity. The first animation considers a random start position and velocity, while the second animation aligns the particles by workgroups. In the first position we set the velocity to <0,random,0> for each particle, indicating that they are moving vertically only; this is to obtain more interaction among particles and therefore more collisions. The number of spheres utilized is increasing by 2^k, starting with $2^6 = 64$, until $2^{14} = 16{,}384$ spheres.

We begin with the first animation. Starting with the sphere-sphere collision detection, the time obtained in the sequential version is 0 ms for 64 to 256 spheres, obtaining a varied number of colliding spheres 0-4, 7-15, and 20-40 for 64, 129, and 256 spheres respectively. The time in the parallel version does not vary (16 ms), with 0-4, 6-20, and 15-48 colliding spheres. As we observed, the number of colliding spheres is similar in both versions, but the sequential version is faster due to the parallelism is not well exploited with a few number of spheres.

With 512 spheres, the time utilized for both versions is the same, 16 ms. The size of the interval of the number of

colliding spheres is similar for both versions: the parallel version [93 ms-139 ms] and the sequential version [100 ms-159 ms]. From 1,024 spheres, the sequential version takes much more time than the parallel version (16 ms). We require 23 ms, 87 ms, 340 ms, 1390 ms, and 5560 ms for 1020, 2048, 4096, 8192, and 16,384 spheres respectively. The parallelism starts making its job. The other type of collision, sphere-plane, takes 0 ms in the parallel version, while in the sequential version takes 0 ms with less than 512 spheres, 2 ms with 1024 spheres, 3 ms with 2048 spheres, 7 ms with 8,192 spheres and 13 ms with 16,384 ms.

In the rendering process, we measure the number of frames per second, and the type of object's display (mesh triangles, *glutSolidSphere* function). The fps in the parallel version does not vary, 60 fps, using the second shader, IRS. In the sequential version we have 60 fps for less than 512 spheres, 36 fps for 1024 spheres, 12 fps for 2018 spheres, and 4 fps for more than 4,046 spheres. Objects can be rendered in the CPU by using the mesh object or the *glutSolidSphere* function, while in the GPU we use the IRS with 0 ms time. The rendering pass in the CPU takes 1 ms for 128 spheres, 2 ms for 256 spheres, 3 ms for 512 spheres, 8 ms for 1024 spheres, 15 ms for 2048 spheres, 32 ms for 4096 spheres, 60 ms for 8192 spheres, and 120 ms for 16,384 spheres. The rendering step is faster using the object mesh than using the *glutSolidSphere*. It takes too much time to follow the animation of the sequential version, however we can continue the simulation of the parallel version up to 256x128=32,768 spheres: the time increases the double (47 ms) and the number of collisions is 32,000.

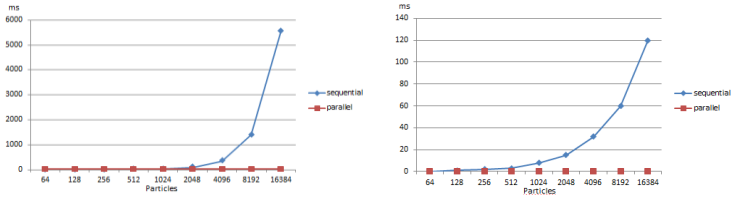
In the second animation, spheres are aligned by workgroup, the times are similar. We can have *m* workgroups of size *n*. With 128 spheres we can have two arrangements: 8x16, 16x8. The following arrangements can be used for the indicated number of spheres: 128 spheres (8x16, 16x8), 256 spheres (8x32, 16x16, 32x8), 512 spheres (8x64, 16x32, 64x 8). The number of collisions can vary in every arrangement, but the difference is minimum.

In Table 1, the times of the simulation are shown. The number of frames per second is described in the second column, while the rendering time is shown in the third and fourth columns. Column five shows the time taken of the collision detection between the sphere and the plane in the sequential version. The last two columns write the time of the collision detection between the pairs of spheres of the sequential and the parallel versions respectively.

Table 1. Times tested for the two versions of the sphere collision detection.

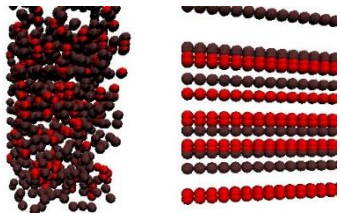
No. Particles	fps	sequential rendering	parallel rendering	sequential collision sphere-plane	sequential collision sphere-sphere	parallel collision sphere-sphere
64	60	0	0	0	0	16
128	60	1	0	0	0	16
256	61	2	0	0	0	16
512	62	3	0	0.5	5	16
1024	36	8	0	1	23	16
2048	12	15	0	2	87	16
4096	4	32	0	3	340	16
8192	4	60	0	7	1390	16
16384	4	120	0	13	5560	16

Figure 5 illustrates the time in milliseconds of the two operations: arithmetic and rendering.



(a) (b)

Figure 5. (a) Time in milliseconds of the sphere-sphere collision detection and (b) the rendering process, for both, the sequential and the parallel versions.



(a) (b)

Figure 6. The two particle simulations: (a) with a random starting position, and (b) spheres group aligned.

7. Conclusion

We have implemented two GLSL shaders on an NVIDIA GTX590 GPU. Arithmetic and rendering operations were analyzed to improve in depth the time efficiency of their correspondent sequential versions. In the arithmetic operation, collision detection, we discover that the time taken in the parallel implementation is the same for 16,000 particles or less, independently of the arrangement used. On the other hand, the rendering animation time was reduced by using the rendering instance, where we obtained less than 1 ms for 16,000 particles or less. The rendering process was performed for an object mesh (sphere) of 240 vertices and 80 polygons. Our method can be applied to other kind of mesh objects, by using the correspondent collision detection operation.

As a further work, grid methods can be compared with the

linear arrangement to pursuit an enhancement in the efficiency. Also, the use of spatial partitioning could be implemented in order to reduce the number of collision tests. The CUDA implementation would be useful to compare with our method.

Acknowledgments

We would like to thank to the Universidad Autónoma de Yucatán and the CONACYT México for their financial support.

References

[1] Müller, M., Keiser, R., Nealen, A., Pauly, M., Gross, M., and Alexa, M. 2004. Point based animation of elastic, plastic and melting objects. In Proceedings of the 2004 ACM. SIGGRAPH/Eurographics symposium on Computer animation, Eurographics Association, 141–151.

[2] Solenthaler, B., Schafli, J., and Pajarola, R. 2007. A unified particle model for fluid–solid interactions: Research articles. Comput. Animat. Virtual Worlds vol. 18, No. 1 (Feb.), 69–82.

[3] Becker, M., Ihmsen, M., and Teschner, M. 2009. Corotated sph for deformable solids. In Proceedings of the Fifth Eurographics conference on Natural Phenomena, Eurographics Association, 27–34.

[4] Martin, S., Kaufmann, P., Botsch, M., Grinspun, E., and Gross, M. 2010. Unified simulation of elastic rods, shells, and solids. In ACM SIGGRAPH 2010 Papers, ACM, New York, NY, USA, SIGGRAPH '10, 39:1–39:10.

[5] Jonathan Cohen, Sarah Tariq, Simon Green. Interactive Fluid-particle Simulation Using Translating Eulerian Grids, Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, 2010, pp. 15-22.

[6] Nadir Akinci, Markus Ihmsen, Gizem Akinci, Gizem Barbara Solenthaler, Matthias Teschner. Versatile Rigid-fluid Coupling for Incompressible SPH. ACM Trans. Graph., 2012, vol. 31, no. 4, pp 621-628.

[7] Iván Alduán, Miguel Otaduy. SPH Granular Flow with Friction and Cohesion. Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2011, pp. 25-32.

[8] Ammann C., Bloom D., Cohen J. M., Courte J., Flores L., Hasegawa S., Kalaitzidis N., Tornberg T., Treweek L., Winter B., Yang C.: The birth of sandman. In ACM SIGGRAPH 2007 sketches (2007).

[9] Markus Ihmsen, Nadir Akinci, Markus Becker, Matthias Teschner. A Parallel SPH Implementation on Multi-Core CPUs, Computer Graphics Forum, 2011.

[10] Daniel Schroeder and Howard J. Hamilton. Desirable Elements for a Particle System Interface. International Journal of Computer Games Technology, Volume 2014 (2014).

[11] Simon Green, Particle Simulation using CUDA. Nvidia, May 2010. http://docs.nvidia.com/cuda/samples/5_Simulations/particles/doc/particles.pdf

[12] Miles Macklin, Matthias Müller, Nuttapong Chentanez, Tae-Yong. Unified Particle Physics for Real-time Applications. ACM Trans. Graph, 2004, vol. 33, no. 4, pp. 1531-1542.

[13] Kenny Erleben, Jon Sporring, Knud Henriksen, Henrik Dohlmann. Physics-Based Animation, Charles River Media.

[14] Instance Rendering with GLSL
http://www.learnopengl.com/#!Advanced-OpenGL/Instancing

[15] Allen C., Bloom D., Cohen J. M., Treweek L.: Rendering tons of sand. In ACM SIGGRAPH 2007 sketches (2007).

Francisco A. Madera received his B. Sc. Degree from the Universidad Autónoma de Yucatán, México; his PhD from the University of East Anglia, UK. Dr. Madera teaches subjects related to computer graphics and videogames development; and his research is focused on computer graphics and GPU programming.

Francisco Moo-Mena is a Professor in Computer Sciences at Universidad Autónoma de Yucatán, in Mérida, Mexico. From the Institute National Polytéchnique de Toulouse, in France, he received a Master Degree in Computer Science and a PhD, in 2003 and 2007, respectively. He also received another Master Degree in Distributed Systems from the Instituto Tecnológico y de Estudios Superiores de Monterrey, Mexico, in 1997. He received a BS in Computer Systems Engineering from the Instituto Tecnológico de Mérida, Mexico, in 1995. His research interests include Parallel and Distributed Computing, CUDA, Self-healing systems, and Web services Architectures.

Enrique Ayala is a lecturer in Computer Sciences at Universidad Autónoma de Yucatán, in Mérida, México. He received a Master Degree in Distributed Systems and Networks from the Instituto Tecnológico y de Estudios Superiores de Monterrey, México, in 2002. He received a BS in Computer Systems Engineering from the Instituto Tecnológico de Morelia, México, in 1993. His research interests include Computer Networks, Parallel and Distributed Computing and GPU Programming.

Luis F. Curi is a lecturer in Computer Science in undergraduate and postgraduate programs at the Universidad Autónoma de Yucatán, México. He received a Ph.D. in Computer Science from the University of Reading, U.K. and a Master in Computer Science from ITESM-Monterrey, México. His research interest focuses on Parallel and Distributed Computing, Theory of Computing and Algorithms.