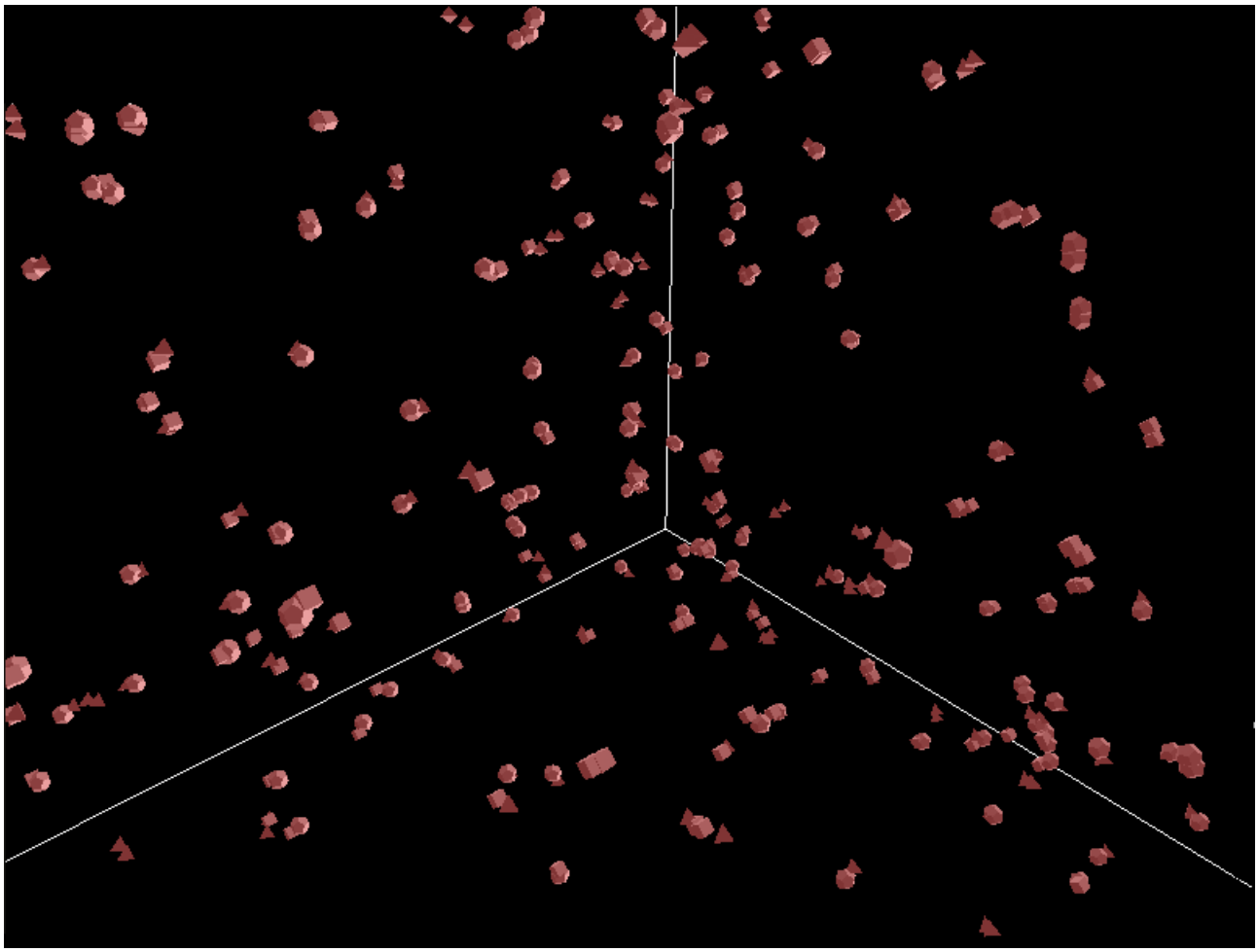
Collision Detection of Multiple Moving Objects on GPU

CS564 Computational Geometry - Progress Report

Isıl Doğa Yakut

May 17, 2010



Contents

1 Introduction

To create a physical-based simulation of objects in an environment, it is essential to be able to detect and analyze the collision between objects. Collision analysis is studied not only for physical-based simulation but also appear in robotics, animation, computer-aided design and computer graphics.

In my project I will create an physical-based simulation of an environment. The environment will consist of rigid convex polyhedron which can bump into each other and react accordingly. Specifically I will simulate the movement and interaction of multiple convex polyhedra in an virtual space according to their translational and rotational velocities. The simulation will run on a PC taking advantage of the GPU's parallel execution.

Throughout this report, I will give a survey on collision detection and response methods (section2), describe the domain of the GPU (section4).

2 Survey on Collision Detection and Response methods

2.1 Overview

Collision detection and response algorithms cover a wide-range of techniques. Due to the large number of algorithms available and aim, it is best to group them according to their goals and application

domains. Algorithms depend on object representation and deformation (section2.2.1), possible types of query (section2.2.2), simulation environment (section2.2.3).

Mainly, collision detection algorithms depend on the data-structure or the bounding volume they use. As for collision detection there is no specific approach that is used generally although a lot of work has been done.

2.2 Problem Domains

2.2.1 Object Representation and Deformation

The data structure selection determines the algorithm to be used most of the time. Data structure to be used for 3D object representation is no exception. There representation options which can be grouped as raw data, surfaces, solids and high level structures (Figure 1) The best data structure to

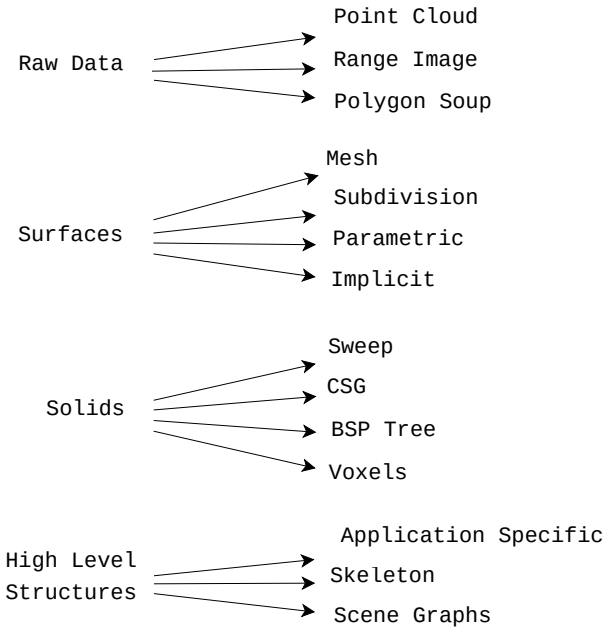


Figure 1: Data structures for 3D object representations

use varies depending on the aim of the application. I will try to briefly mention all of them with their use in an collision detection and response application.

Point Cloud Unstructured 3D point samples. This data structure is inconvenient One would have to compute all possible faces which is highly expensive. [2]

Range Image 3D point set mapped to depth images. This data structure is inconvenient. Even though the points are more ordered it is still a hard task, one would have to compute all possible faces which is highly expensive.

Polygon Soup Unstructured 3D set of polygons. Usually polygon soups refer to a set of polygons that may not be geometrically connected. If we restrict the set to form a closed manifold and be convex then this structure can be used for collision detection. [1]

Mesh Connected set of polygons usually triangles. Similar to polygon soup if we restrict the mesh to be convex then this structure can be used for collision detection.

Subdivision A Coarse mesh and a subdivision rule. This structure is usually used for refinement.

Parametric A set of spline patches used for continuity.

Implicit A set of points decided as on the surface, inner volume or outer volume using equation:

$$f(n) = \begin{cases} IN & \text{if } F(x,y,z) < 0 \\ ON & \text{if } F(x,y,z) > 0 \\ OUT & \text{if } F(x,y,z) = 0 \end{cases}$$

Sweep The volume defined by sweeping the moving object during unit time. If two swept volumes do not intersect then no collisions occurred. There are several methods using this data structure to compute collision

CSG The Constructive Solid Geometry(CSG) is used to create a complex surface using boolean operators to combine 3D objects. [6]

BSP Tree The space defined by recursively subdividing the initial space by hyperplanes. This data structure is mostly used for optimizing rendering. [4]

Voxels A set of voxels that defines the object. This can have an advantage for parallel execution.

Skeleton Are not applicable to object-object collision

Scene Graphs Are not applicable to object-object collision

Another important issue which adds to the complexity of the problem is the objects deformability. In simulations the objects in the environment can assumed to be non-deformable. Thus if the object is not deformable (very unlikely in real world) it is called a rigid body and if it can deform depending on the forces acting on it we call it deformable object. This selection can play an important role in the selection of algorithm and data structure to be used. I will not give information on deformable bodies since it is a very complex topic and my implementation will be on rigid bodies.

2.2.2 Type of Query on Collision Detection

The problem of collision detection can take a new form depending on query. The simplest query is whether two objects are intersecting or not. Further more if they are intersecting, we can question the contact points, the penetrating distance (the distance required to move them apart). Similarly if they are not intersecting, the query can be the points on the objects closest to the other one, the minimum distance between these points and if the objects have a motion information their estimated time of arrival (the time of the next collision).

2.2.3 Environment

Number of bodies

The environment can consist of only two object to apply collision detection. Or multiple bodies can exist in the environment. If only two objects are present then the computation in the environment is called "Pair processing", and if multiple objects exist it is called "N-body processing". [3]

Type of motion

The motion can be either static or dynamic.

If the motion is dynamic then usually the query in question is executed repeatedly on the same models over time as the objects move according to their translations and rotations.

Sometimes the motion is not dynamic. That is the objects do not move. In this case query might be the points/faces on the contact interface.[3]

3 Problem Domain, Adopted Approach and Methods

3.1 Algorithm Overview

This chapter gives the overall algorithm and the methods used for the parts of the algorithm. The flowchart of the algorithm is given in (Figure 2)

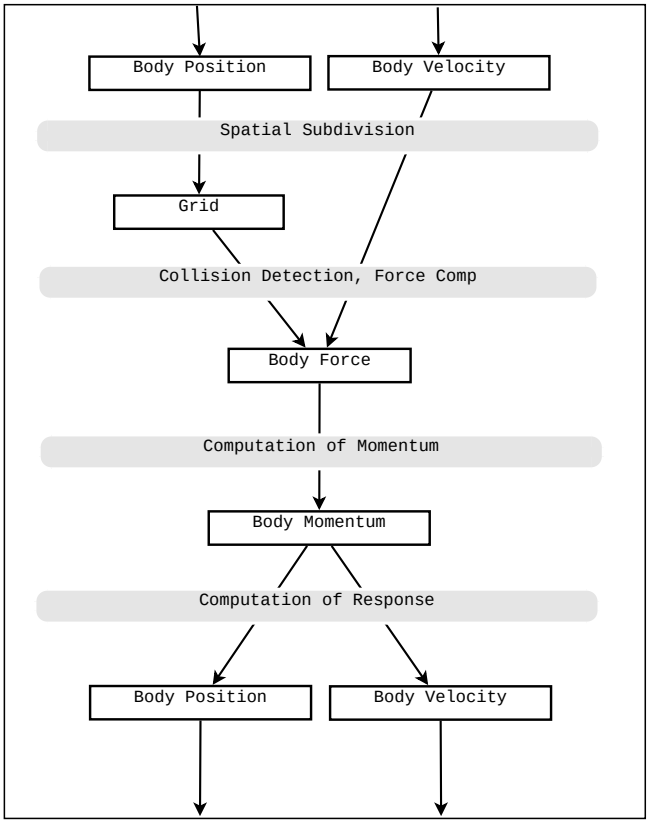


Figure 2: Algorithm overview

If we were to compute collision detection and the according response in a brute force manner then the required time would be $O(n^2)$. A good optimization would be to restrict the collision detection and response computations to object pairs whose distances to each other is small. This can be achieved by using a 3D grid which divides the 3D space into unit voxels. The grid can be generated using spatial subdivision (detailed in Section 3.2)

Next, for each body pair that are close enough to each other, collision detection should be applied. There are several ways such as using swept volumes(Section 2.2.1). I will use Eulerian distances which is a simpler approach. The bodies are represented with their spherical bounding volumes. This will not result with very accurate simulation. If time permits I will be implementing a more complex scheme for collision detection.

If a collision occurs between two objects then collision response should be computed accordingly. (detailed in Section 3.4)

3.2 Spatial Subdivision

Spatial subdivision divides the space into a uniform grid. Each voxel can consist of a number of bodies and each body is assigned to a cell according to its center. The body with its center at $\mathbf{o}(o_x, o_y, o_z)$ belongs to voxel $\mathbf{g}(g_x, g_y, g_z)$ with the equation:

$$g = \frac{(o - s)}{d}$$

where $\mathbf{s}(s_x, s_y, s_z)$ is the smallest grid corner and d is unit length of a voxel. So the collision can only occur between two bodies if they are in the same or adjacent voxels. (Figure 3)

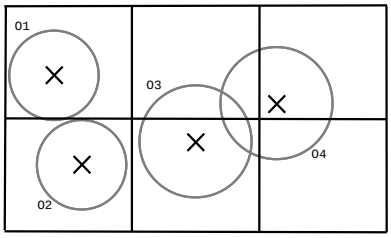


Figure 3: A 2D example of a grid and intersecting objects within that grid. Objects are illustrated by their bounding boxes as spheres

When in 3D, the required number of adjacent voxels equal to 26. Also, the most ideal unit length of a voxel should be the radius of the one with largest raduis value.

Notice that the each collision detection will be tested two times for both objects. We can further improve that by using radix sort on the cells but I omit that part for the time being.

3.3 Collision Detection and Force Computation

After the neighboring voxels are found, a total of 3x3=27 voxels are used for collision detection. For body i its voxel index is found. Then particles indices are looked up from 27 voxels to be testes with body i (Figure 4) .

I will detect the collision with a simple Eulerian distance test between the spherical bounding volumes. This approach may not be the most accurate approach but because of the performance issues I plan to keep this phase simple.

The collision response will use Distinct Element Method for the objects/bodies are treated as particles [5]. The motion of body i is given by :

$$\frac{d\mathbf{v}_i}{dt} = \frac{1}{m} \sum_{j \in contact} f_{ij}^c + \mathbf{g}$$

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i$$

where $\mathbf{v}_i$, $\mathbf{x}_i$, m, $\mathbf{g}$ and f_{ij}^c are the position vector of the body center, velocity of the body, mass, gravity and force of collision response respectively.

The paper uses the method proposed in [ref] for computing the force of collision response which uses spring and dashpot forces:

$$f_{ij}^{spring} = -k_s(d - |\mathbf{r}_{ij}|)\frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|}$$

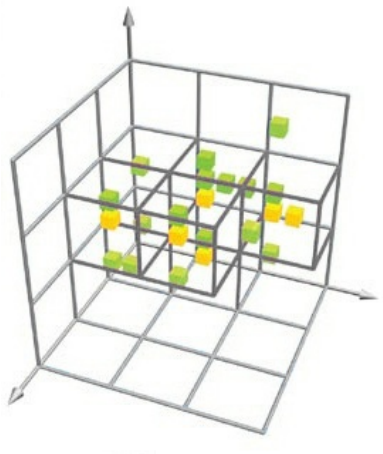


Figure 4: An example grid and objects with AABB's. My implementation will use spheres as bounding boxes.

$$f_{ij}^{damp} = \eta(\mathbf{v}_j - \mathbf{v}_i)$$

where $\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i$, $\mathbf{v}_i$, $\mathbf{v}_j$, d, k_s and η are the distance between particles, velocities of the particles, the sum of body radii, the spring coefficient and the damping coefficient respectively.

The sum of f_{ij}^{spring} and f_{ij}^{damp} gives the force from body j to body i.

3.4 Collision Response

3.4.1 Momentum

After we derive the forces acting upon body i, the momentum is computed using:

$$\frac{d\mathbf{P}}{dt} = \mathbf{F}$$

For each body that acts on the the body i the forces are summed up to compute the above equation.

3.4.2 Position and Velocity Updates

At this point we will have a single momentum value for each body. The velocity $\mathbf{v}$ of body i can be computed using the mass of the body as:

$$\mathbf{v} = \frac{\mathbf{P}}{\mathbf{M}}$$

Also the position can be computed using the familiar equation :

$$\frac{d\mathbf{x}}{dt} = \mathbf{v}$$

4 Implementation Details

4.1 GPU Programming and CUDA

CUDA is a parallel computing architecture developed by NVIDIA. There are several advantages of using CUDA when writing highly data-parallel programs since CUDA enable access to a fast shared

memory region which enables fast texture lookup which is usually the case when running a data parallel program.

The development environment will be PC with Windows 7 64-bit OS and NVIDIA GeForce 9600GT GPU.

4.2 Implementation specifications for GPU

Parallel execution of a program on GPU effects the data-structures to be used as well as the algorithm. The algorithm consist of four stages (Figure 2)

1. Spatial Subdivision
2. Collision Detection & Force Computation
3. Collision Response: Momentum Computation
4. Collision Response: Position and Velocity Update

4.2.1 Data Structures on GPU

For simulation of physical-based movement of convex objects, attributes of the objects must be stored. These attributes are typically stored in an 1D array when doing computation in CPU. But for computation on GPU it is better to choose a 2D array for storing the values because of the device restrictions and optimization reasons.

Each object has a index which is converted to 2D index for storing in a texture. Also two textures for each position, velocity and momentum attributes should exist since a read operation will execute on one of them and write operation on the other one. All textures compose a larger 2D texture which is called *flat 3D texture.*(Figure 5)

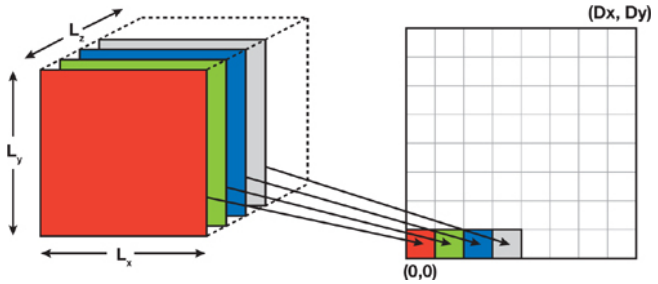


Figure 5: The mapping between several 2D textures and a flat 3D texture

4.2.2 Spatial Subdivision

Using a vertex shader, the voxels of the 3D grid are filled with object indices. Each voxel will have four values (RGBA) and object indices can be stored in these four places. But if we specify only one value to write to (for example R) then only one of the four possible objects will be stored in the voxel information. To avoid this side effect, I will use a method introduced by [ref]. Which handles the problem using depth test, stencil buffer and color-masks using a four pass algorithm. The pseudo code is as follows:

```
//=== 1 PASS
colorMask(GBA);
```

```
depthTest(less);
renderVertices();
//===2 PASS
colorMask(RBA);
depthTest(greater);
stencilTest(greater, 1);
stencilFunction(increment);
clear(stencilBuffer);
renderVertices();
//=== 3 PASS
colorMask(RGA);
clear(stencilBuffer);
renderVertices();
//=== 4 PASS
colorMask(RGB);
clear(stencilBuffer);
renderVertices();
```

4.2.3 Collision Detection and Force Computation

This stage of the algorithm is also executed in parallel so it is sufficient describe the computation done on only one object.

At this stage all candidate object indices are read from adjacent voxels and all colliding objects act on the force exerted on body i.

4.2.4 Collision Response

The momentum, position and velocity updates can be done in an simple fragment shader which will act on a texture pixel per pixel.

5 Discussion

I was working on a NVIDIA GeForce 9600GT device. To see the running time of the application it was necessary to implement a FPS counter. Number of bodies and the rendering effect the frame rate also. The table given below give the relation of these with the frame rate.

#bodies	fps w/ rendering	fps w/o rendering
1024	167	210
4096	84	190
16384	30	110
65536	10	60

As it can be seen from the table the rendering takes much of the time. I have improved my code using opengl "draw arrays". A further improvement would be to use vertex buffers. Currently evert vertex is rendered three times where it could be rendered once using vertex buffers.

Also the code once the collision detection is done using the grid, a second collision detection technique can be implemented for further accuracy.

References

[1] S. Gottschalk, M. C. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection, 1996.

[2] Jan Klein and Gabriel Zachmann. Point cloud collision detection, 2004.

[3] Ming C. Lin and Stefan Gottschalk. Collision detection between geometric models: A survey. In *In Proc. of IMA Conference on Mathematics of Surfaces*, pages 37–56, 1998.

[4] Rodrigo G. Luque, ao L. D. Comba, Jo and Carla M. D. S. Freitas. Broad-phase collision detection using semi-adjusting bsp-trees. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 179–186, New York, NY, USA, 2005. ACM.

[5] B. K. Mishra. A review of computer simulation of tumbling mills by the discrete element method: Part i–contact mechanics. *International Journal of Mineral Processing*, 71(1-4):73 – 93, 2003.

[6] Kai Poutrain and Magali Contensin. Dual brep-csg collision detection for general polyhedra. In *PG '01: Proceedings of the 9th Pacific Conference on Computer Graphics and Applications*, page 124, Washington, DC, USA, 2001. IEEE Computer Society.

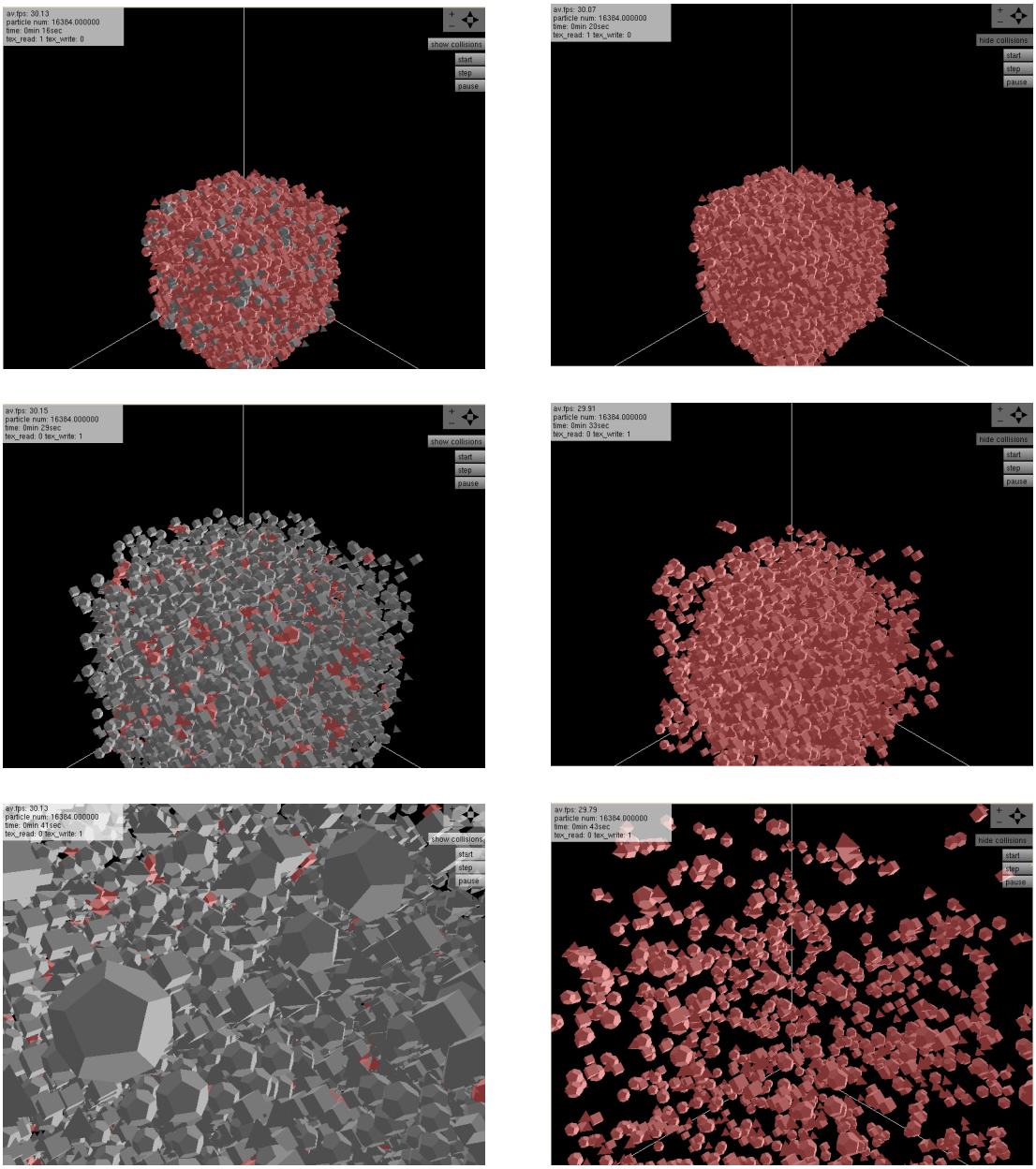


Figure 6: The images on the right show the environment with all the bodies rendered and moving. On the right, only the colliding objects are shown.