Exploiting ray tracing technology through OptiX to compute particle interactions with cutoff in a 3D environment on GPU

Bérenger Bramas

Exploiting ray tracing technology through OptiX to compute particle interactions with cutoff in a 3D environment on GPU

Bérenger Bramas[1]

[1]*Inria Nancy, ICube Laboratory, University of Strasbourg, France*
Berenger.Bramas@inria.fr

Abstract

Computing on graphics processing units (GPUs) has become standard in scientific computing, allowing for incredible performance gains over classical CPUs for many computational methods. As GPUs were originally designed for 3D rendering, they still have several features for that purpose that are not used in scientific computing. Among them, ray tracing is a powerful technology used to render 3D scenes. In this paper, we propose exploiting ray tracing technology to compute particle interactions with a cutoff distance in a 3D environment. We describe algorithmic tricks and geometric patterns to find the interaction lists for each particle. This approach allows us to compute interactions with quasi-linear complexity in the number of particles without building a grid of cells or an explicit kd-tree. We compare the performance of our approach with a classical approach based on a grid of cells and show that, currently, ours is slower in most cases but could pave the way for future methods.

Keywords: CUDA, GPU, HPC, OptiX, Particle interactions, Ray tracing, Scientific computing

1 Introduction

High-performance computing (HPC) is a key technology in scientific computing. Since the early 2000s, the use of graphics processing units (GPUs) has become standard in HPC, and they now equip many of the fastest supercomputers[1]. GPUs are massively parallel processors that allow computations to be performed on thousands of cores, making them perfectly suited for inherently parallel problems. The use of GPUs in scientific computing has led to incredible performance gains in many fields, such as molecular dynamics, fluid dynamics, astrophysics, and machine learning.

Most scientific computing applications that use GPUs are based on the CUDA programming model, and to a lesser extent, the OpenCL programming model. They do not exploit all the features of GPUs, particularly those dedicated to 3D rendering. Among these features, ray tracing is a powerful technology used to render 3D scenes. In this method, rays are cast from the camera to the scene, and the intersections of the rays with the objects in the scene are computed to generate the colors of the pixels in the image. For instance, on a GeForce RTX 4090, this unit can generate images at 87 frames per second with 4K resolution (3840 x 2160 pixels), managing millions of triangles per frame.

In this paper, we are interested in evaluating how ray tracing technology could be used to compute particle interactions in a 3D environment. Our motivation is twofold: we want to evaluate if it is possible to use ray tracing technology, and we want to create the algorithmic patterns needed for that purpose.

With these aims, we focus on the computation of particle interactions in a 3D environment, which is a common problem in scientific computing. When the potential of the interaction kernel decreases exponentially with distance, the interactions can be computed with a cutoff distance, i.e., the interactions are only computed between particles that are closer than a given distance, achieving less but still satisfactory accuracy. This allows the complexity of the interactions to drop from $O(N^2)$ to $O(N)$, where N is the number of particles if the cutoff distance is small enough. Classical methods to compute such interactions are based on the use of a grid of cells or an explicit kd-tree to quickly find the interaction lists for each particle. In this paper, we aim to avoid using such data structures and instead exploit ray tracing technology.

The contributions of this paper are as follows:

- We propose a method to compute particle interactions with a cutoff distance in a 3D environment based on ray tracing technology.
- We describe algorithmic techniques and geometric patterns to find the interaction lists for each particle.
- We compare the performance of our approach with a classical approach based on a grid of cells.

The rest of the paper is organized as follows. In Section 2, we present the prerequisites. In Section 3,

[1] https://top500.org/

we review the related work. In Section 4, we present our proposed approach. In Section 7, we present the performance study. Finally, we conclude in Section 8.

2 Prerequisites

2.1 Particle interactions

Computing the interactions between N particles in a 3D environment is a common problem in scientific computing. These interactions are usually modeled by a potential function V that depends on the distance between the particles. A straightforward way to compute the interactions is to evaluate the potential function for all pairs of particles. However, the potential function can be short-range or long-range. When the potential function is short-range, the interactions can be computed with a cutoff distance, i.e., the interactions are only computed between particles that are closer than a given distance. This reduces the complexity of the interactions from $O(N^2)$ to $O(N)$, where N is the number of particles, but this is only possible if we have an efficient way to find the particles' neighbors. Moreover, the positions of the particles are usually updated after each computation step. Consequently, the system used to find the interactions between the particles should be updated at each iteration of the simulation.

A possible solution to get the interaction list is to build a grid of cells mapped over the simulation space, where each cell contains the particles that are inside. The cells have a width equal to the cutoff distance C. Then, for each particle, the interactions are computed with the particles that are in the same cell and in the neighboring cells. Building the cells and finding the interaction lists for each particle can be done in $O(N)$: we start by computing the cell index for each particle, then we order the particles in a new array by assigning a unique index per particle using atomic operations, and finally, we move the particles to a new array. Each of these three operations can be implemented with a parallel loop over the N particles.

2.2 Graphics processing units

Graphics processing units (GPUs) are massively parallel processors that allow computations to be performed on thousands of cores. The hardware design of GPUs has been optimized for graphics rendering, particularly for the rendering of 3D scenes. To this end, GPUs have features dedicated to 3D rendering, such as texture mapping, rasterization, and ray tracing. Internally, GPUs are organized in a hierarchy of processing units, including streaming multiprocessors (SMs), warp schedulers, and execution units.

NVIDIA has proposed the CUDA programming model to develop parallel applications for GPUs. CUDA is designed to express algorithms in a way that can be mapped to the GPUs' hardware organization. For instance, thread blocks are distributed across SMs, and threads are executed in warps. There are also keywords and functions to use shared memory, constant memory, and texture memory. Thus, creating optimized applications for GPUs requires an understanding of the hardware architecture of GPUs and potentially adapting algorithms to their specificities.

2.3 Ray tracing

Among the various features of GPUs dedicated to 3D rendering, ray tracing is a powerful technology used to render 3D scenes and is widely used in video games. It is a hardware-accelerated technique that computes the interactions of rays with objects in the scene. Ray tracing generates pixel colors in the image by casting rays from the camera to the scene. For each intersection of a ray with an object, the pixel's color is computed by considering the object's material properties and lighting conditions. A ray can potentially be reflected or refracted by the object, or it may continue and intersect with several non-opaque objects, allowing for recursive computation of interactions.

Typically, ray tracing kernels are implemented in the shaders of the graphics pipeline. Shaders are small programs executed on the GPU for each pixel in the image and are usually written in specific languages, such as GLSL for OpenGL or HLSL for DirectX. They run in parallel on the GPU, and the interactions of rays with objects are computed concurrently. However, NVIDIA has proposed a way to use ray tracing technology in the CUDA programming model through the OptiX library [1]. With OptiX, we still face constraints in implementing our kernels, but we can write everything in CUDA.

3 Related work

The main work on particle interactions on GPU has been proposed by Nyland et al. [2], to compute the gravitational potential. They described an efficient implementation using shared memory that became a standard implementation on GPU. We also study different implementation in a previous study [3] with a focus on cases when thy are few particles per cell, and we have showed that using shared memory was usually not useful.

Using Optix and CUDA for implementing a scientific computing application has been done for simulation related to optics, where ray tracing can be used for its original purpose. For instance, in the development of Opticks an GPU Accelerated optical photon simulation [4, 5], or the modeling of optical 3D measuring devices [6].

There is existing work that explores the use of ray tracing to build the interaction lists between particles [7]. To our knowledge, this is the first attempt in this direction. In this approach, each particle is represented by a surrounding box, and a single ray is

emitted from each particle. These rays are of minimal length, based on the idea that detection is only necessary when a particle is close to the surrounding box. If an intersection with a box is detected, the IDs of the source and target particles are stored in a list.

However, this approach has several limitations. First, the description provided is very high-level, making it difficult to implement and reproduce the results. Second, the explanation lacks clarity regarding how the rays are cast, how intersections are detected, and whether the method is applicable in all scenarios. For example, there are no details on the direction of the rays, the size of the boxes, or how particles that are inside the boxes but do not intersect are handled. Lastly, the source code is not publicly available.

4 Approach overview

The core idea of our approach is to represent particles using geometric primitives and to use ray tracing to find the neighbors of each particle by detecting intersections with these primitives. We consider two different representations of the particles: spherical and square (composed of triangles). The spherical approach is simpler and more intuitive, but the square approach is expected to be more efficient as it is the representation used in 3D rendering.

In both cases, we use the following algorithmic pattern:

1. We build a geometric representation for each particle.
2. We cast rays from each particle in specific directions to find the neighbors, depending on the representation.
3. We filter the intersections to avoid computing the same interaction multiple times.
4. We compute the interactions between the particles that are closer than C.

To reduce overhead, we aim to use as few rays as possible and ensure they do not intersect with too many particles that are not within the distance C.

In terms of implementation, we use the OptiX library to develop the ray tracing kernels, which can be used in conjunction with the CUDA programming model. Specifically, in the OptiX API, we create a scene by providing a list of geometric primitives. We then provide a CUDA kernel that launches the rays, where each ray has an origin, a direction, a starting point, and an endpoint. Usually, one CUDA thread is used to launch one ray. Finally, a callback is invoked by OptiX when a ray intersects with a primitive or if no intersection is found between the starting and ending points (in 3D rendering, this usually means that the background color should be used).

The data accessible from the callback is limited. OptiX can provide information about the intersection, such as the intersection point, the normal on the surface, the distance from the ray's origin, and the index of the primitive that was hit. Additionally, the user can pass information from the CUDA kernel that launches the rays to the callback using a payload. A payload is a user-defined data structure that is passed along with a ray as it traverses the scene. It allows the ray to carry information that can be read or modified. The number of payload variables is limited (usually 16 32-bit integers in recent versions).

When the hit callback is invoked, and we want the ray to continue, there are two possibilities. The first is to launch a new ray from the intersection point in a new direction. This is done by storing the intersection distance in a payload variable, returning from the callback, and then launching a new ray from the intersection point using a loop in the CUDA kernel to reach the desired distance. The second possibility is to inform OptiX that we want to continue the traversal by calling a corresponding function in the intersection callback. It is clear that the second possibility is more efficient in most cases and should be used in practice.

Additionally, we cannot allocate memory in the callbacks, so we cannot build complex data structures, such as lists, to store intersection lists. Consequently, if we want to filter the intersections, we cannot fill an array with indices and check if an index exists in the array to ensure uniqueness; instead, we must use geometric properties to filter the intersections.

In the remainder of the section, we consider that the target particle is the particle for which we want to find the neighbors.

5 Spherical representation

In this section, we consider the case where the particles are represented by spheres. In OptiX, a sphere is a geometric primitive defined by its center and radius, and it is possible to instantiate several spheres in the scene, all with the same radius, which is what we use. The questions to be solved are the following:

1. What should be the radius of the spheres?
2. What should be the origins and directions of the rays?
3. Can rays intersect multiple times with the same sphere, and if yes, how should they be filtered?

In our model, we will use three rays for each particle, one in each direction of the coordinate system. Consider a sphere of radius C centered at the origin in a three-dimensional space. The points on this sphere that are at the farthest distance from the three coordinate axes are located in the corners of a cube inscribed within the sphere. For instance, one such point at distance C from the origin lies in the direction $(1,1,1)$.

These 8 points, corresponding to the vertices of the cube, are all at a distance C from the origin, and we want to know how far they are from the coordinate axes. This can be calculated as follows: since the points have coordinates where $|x| = |y| = |z|$, we use the equation of the sphere $x^2 + y^2 + z^2 = C^2$. If we take the point for which $x = y = z$, it gives $3x^2 = C^2$, resulting in $x = \frac{C}{\sqrt{3}}$. Therefore, each of these points is at a distance of $\frac{C \times \sqrt{2}}{\sqrt{3}}$ from any of the three coordinate axes.

We use this information to define the radius of the spheres and the length of our rays. The radius is set to $r = \frac{C \times \sqrt{2}}{\sqrt{3}}$. In this scenario, it is sufficient that the rays go up to $l = \frac{C}{\sqrt{3}}$ in each direction relatively to the particle's position (so a single ray goes from $-l$ to l). We provide a simplified 3D rendering of the spheres in Figure 1 that illustrate our model.

However, if $l < r$, there are positions where the sphere could simply englobe the rays, and we would miss some intersections (when the source and target are closer than $r - l$ in the three dimensions). Therefore, we set $l = r$ and add an ε to the radius of the sphere to ensure that the rays intersect with the sphere in all cases, obtaining $r = \frac{C \times \sqrt{2}}{\sqrt{3}} + \varepsilon$ and $l = \frac{C \times \sqrt{2}}{\sqrt{3}}$. The ε is a small value such that it must be impossible that two particles are closer than ε, or some intersections will be missed (see Appendix I for more details).

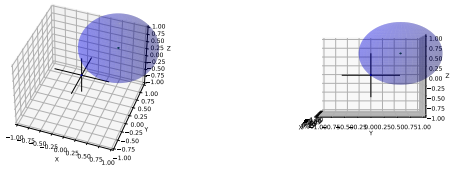


Figure 1: 3D Spherical representation of a source and target particles distance from $C = 1$. The source sphere has a radius of $\frac{\sqrt{2}}{\sqrt{3}}$ and the rays, represented by segments, are of length $\frac{1}{\sqrt{3}}$ in each direction. In this case, the sphere could englobe the rays.

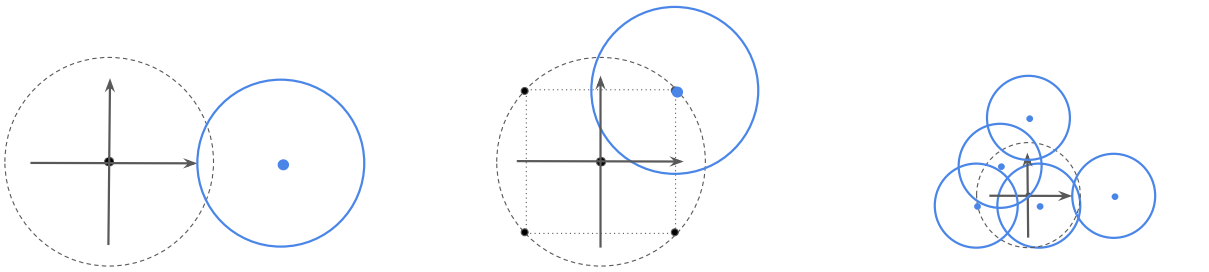
When the source and target particles are perfectly aligned on one axis, the ray will intersect for particles distant from the extremity by $r + \varepsilon$. Therefore, two particle distant from $l + r + \varepsilon$ can interact. This case and any intermediate situation where the source/target are actually too far can easily be filtered by checking the distance. In Figure 2, we provide a 2D representation of the particles using spheres on intricate cases.

However, each ray can potentially intersect with a sphere several times, and the same sphere can be intersected by several rays of the same target particle, so we need to filter them. It is impossible to maintain a global list that all the rays can access to check if an intersection has already been found, or even a single list per ray to ensure that it does not intersect with the same sphere. Therefore, we must do this based on geometric rules as described in Algorithm 1. When we detect an intersection, we get the position of the source and target particles and first check they are within the cutoff distance. Then, we check if the closest ray to the source particle is the current one, and if yes, we can proceed with the computation (see Appendix I for more details).

Algorithm 1: Sphere intersection callback

Data: Optix variables
Result: Callback when a ray intersect with a sphere

1 **Function** callback()
2 **begin**
  /* Get the center of the sphere (source position) */
3 $q \leftarrow$ `optixGetSphereData()`
  /* Current particle position (target position) */
4 $point \leftarrow$ `getPayloadPartPos()`
  /* Compute differences and distances */
5 $diff\_pos \leftarrow$ $\{\texttt{abs}(point.x - q.x), \texttt{abs}(point.y - q.y), \texttt{abs}(point.z - q.z)\}$
6 $diff\_pos\_squared \leftarrow$ $\{diff\_pos.x^2, diff\_pos.y^2, diff\_pos.z^2\}$
7 $dist\_squared \leftarrow diff\_pos\_squared.x + diff\_pos\_squared.y + diff\_pos\_squared.z$
  /* Get the cutoff distance from payload */
8 $c \leftarrow$ `getPayloadC()`
  /* Ensure it is in the cutoff distance */
9 **if** $dist\_squared < c^2$ **then**
10 $dist\_axis\_squared \leftarrow$ $\{diff\_pos\_squared.y + diff\_pos\_squared.z, diff\_pos\_squared.x + diff\_pos\_squared.z, diff\_pos\_squared.x + diff\_pos\_squared.y\}$
11 $ray\_dir \leftarrow$ `optixGetWorldRayDirection()`
12 $closest\_axis \leftarrow$ `getClosestAxis`$(dist\_axis\_squared)$
13 $closest\_axis\_is\_ray\_dir \leftarrow (ray\_dir == closest\_axis)$
  /* Ensure this ray and this intersection are the good one */
14 **if** $closest\_axis\_is\_ray\_dir$ **then**
    /* Call computation kernel */



2D representation when the source and target are aligned, the source will be detected at *r + l + epsilon*.

2D representation with the four positions the farest from the rays. The source sphere intersect with both rays.

2D representation with multiple sources that have their spheres intersected with the rays.

Figure 2: 2D Spherical representation of the particles in three different cases. In the first one (left), we what will be the positioned of the farthest source particle and how it will be detected by the ray. In the second one (center), we show the case where the source particle is the farthest from the rays (it also shows that in 2D the sphere radius could be smaller). In the last one (right), we show different source particles with their spheres and the rays that will intersect with them.

6 Double squares representation

Most 3D rendering applications use triangles to represent the objects in the scene, which motivated us to create a second model that relies on triangles instead of spheres. In our model, we use four triangles to draw two squares, which are positioned opposite each other. Each square has a width of $C+\varepsilon$ and is positioned at a distance of $C/2$ along the X-axis relative to the particle, one in each direction. We provide Code 1 in Appendix II, which shows how we generate the triangles from the particles' positions.

We then launch four rays, all with the same length and direction, but positioned at the corners of the squares. Each ray is positioned at $-C/2$ from the center of the square and has a length of $C+\varepsilon$. The ε is used to ensure that when the source and target particles have the same x coordinate and their squares overlap, the rays cross the triangles. We provide in Figure 3 the 2D representation of the particles using squares (which are lines in 2D).

From this description, one particle can be seen as a box of sides $(C+\varepsilon, C+\varepsilon, C)$. The rays can be seen as the four edges of the box in the x direction, and the triangles composed the front and back faces. If any two boxes have an intersection, we will detect it as shown in Figure 3.

Potentially, the ray will intersect with the squares of the target particle, but this can easily be filtered by checking either the coordinates or the index of the geometric elements. Additionally, if the target and source particles are aligned on the y or z axis, two rays will intersect with the source's squares. To filter these intersections, we proceed as shown in Algorithm 2. We compare the coordinates between the source and the target and proceed as follows: If y and z are different, we perform the computation (only the current ray will intersect). If y is equal, we use the ray of index 0 if z is smaller, and the ray of index 2 if z is greater. If z is equal, we use the ray of index 0 if y is smaller, and the ray of index 1 if y is greater. Otherwise, only the ray of index 0 will be used for computation (all four rays will intersect).

7 Performance study

7.1 Experimental setup.

Hardware We have used two NVidia GPUs:

- A100 with 40GB hBM2, 48KB of shared-memory, 108 multi-processors, zero RT Cores, 8192 CUDA Cores max single-precision performance 19.5TFLOPS, and max tensor performance 311.84TFLOPS.
- RTX8000 with 48GB GDDR6, 48KB of shared-memory, 72 multi-processors, 72 RT Cores, 4608 CUDA Cores, maximum Ray casting of 10Giga Rays/sec, max single-precision performance 16.3TFLOPS, and max tensor performance 130.5TFLOPS.

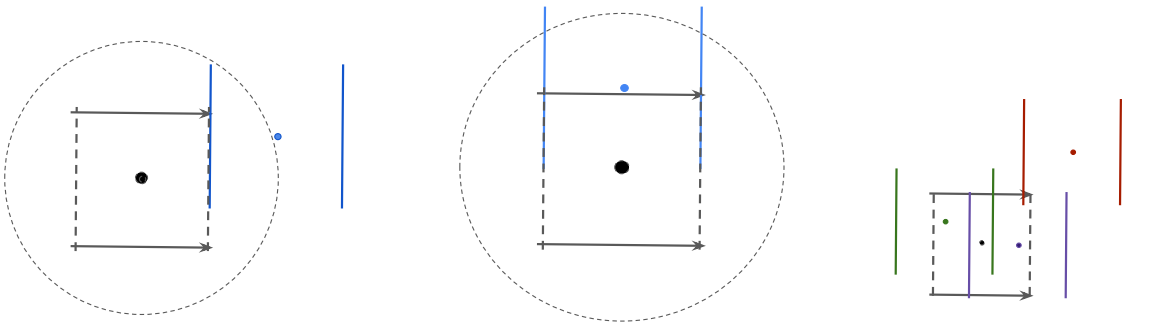
Despite the lack of RT cores, the A100 is capable of executing ray tracing kernels, the GPU then use its other units to behave similarly.

Software We have implemented the proposed approach in OptiX 8 [2]. We use the GNU compiler 11.2.0 and the NVidia CUDA compiler 12.3. The source code is available online [3].

The code was compiled with the following flags: `-arch=sm_75` for the RTX8000, `-arch=sm_80` for

[2] `https://developer.nvidia.com/rtx/ray-tracing/optix`

[3] `https://gitlab.inria.fr/bramas/particle-interaction-with-optix`



2D representation when the source and target are distant from C in X direction, their squares overlap.

2D representation when the source and target have the same X position, both of their squares overlap.

2D representation with multiple sources that have their squares intersected with the rays.

Figure 3: Double squares (line) representation of the particles. On the left, we show how the squares can overlap for particles that are too far, but which can be easily filtered with the distance. In the middle, we show how particles that have the same x coordinate can have their squares that overlap. On the right, we show different source particles with their squares and the rays that will intersect with them.

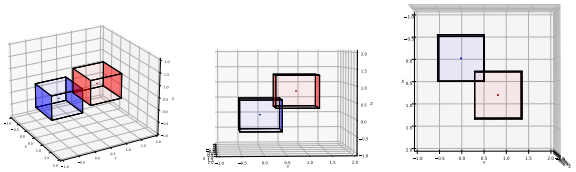
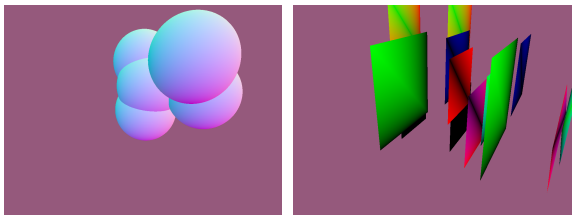


Figure 4: 3D Spherical representation of a source and target particles. Some rays (thick lines) intersect with the squares.

the A100 (and `-O3 -DNDEBUG` on both). We execute each kernel 5 times and take the average as reference. We use a cutoff of $1/d$, with d being 2, 4, 8, 16 or 32, and we put $p \times 3^d$ particles, with p being all power of two from 1 to 32. We perform the computation in single floating point precision.

We provide in Figure 5 the 3D rendering of the primitives using ray tracing: in Figure 5a for the spheres and in Figure 5b for the squares. These figures were drawn using the OptiX API and ray tracing from camera to the scene.



(a) 3D rendering of sphere primitives.

(b) 3D rendering of square primitives.

Figure 5: 3D rendering of the primitives using conventional ray tracing.

Results. We provide the results in Figure 6. For all configurations, we measured the initialization step (light color) and the computation step (dark color). For the OptiX-based implementation, the initialization step involves building the scene by calling the OptiX API to create the primitives. For the CUDA version, the initialization step involves building the grid of cells. Therefore, for the OptiX-based version, the computation step includes the time spent in the kernel that launches the rays, the time spent in the callback that is called upon intersection, and the time to perform the computation. For the CUDA version, the computation step is the time spent in the kernel that computes the interactions.

First, we examine the performance difference between the triangles (blue) and the spheres (red). On the A100, both models provide similar performance, but we can notice that the ration of initialization (light color) over computation (dark color) time is higher for the spheres, which means that the computation step is more efficient for the spheres than the triangles. This suggests that in a scenario where the initialization stage needs to be performed only once (i.e., the elements are not moving), using spheres could be a good choice. However, on the RTX8000, the triangles are faster than the spheres for all configurations. While the initialization step of the sphere is faster than the triangles for few particles, it is not the case in general.

Second, we assess the performance difference between the two GPUs. The A100 results are presented in Figures 6a, 6c, 6e, 6g, and 6i, and the RTX8000 results are presented in Figures 6b, 6d, 6f, 6h, and 6j. Both GPUs achieve comparable performance and a similar ratio of initialization to computation time. The raw performance of the A100 is expected to be better than the RTX8000, but the OptiX implementation is not optimized for the A100, and the RTX8000 has more RT cores that could accelerate the computation.

Algorithm 2: Triangle intersection callback

Data: Optix variables
Result: Callback when a ray intersect with a triangle

```
1  Function callback()
2  begin
       /* Get information on the
          intersected triangle                    */
3      vertices ←
        optixGetTriangleVertexData
        (gas, prim_idx, sbtGASIndex, 0.f, vertices)

       /* Retreive the source position from
          the triangle vertices                   */
4      Declare q as float3
5      q.y ←
        (max(vertices[0].y, vertices[1].y, vertices[2].y) +
        min(vertices[0].y, vertices[1].y, vertices[2].y))/2

6      q.z ←
        (max(vertices[0].z, vertices[1].z, vertices[2].z) +
        min(vertices[0].z, vertices[1].z, vertices[2].z))/2

7      c ← getPayloadC()
8      if (prim_idx mod 4) < 2 then
9          q.x ← vertices[0].x + c/2
10     else
11         q.x ← vertices[0].x − c/2
       /* Get target particle position            */
12     point ← getPayloadPartPos()
       /* Compute distance                        */
13     dist_p1_p2 ← distance(point, q)
       /* Ensure it is not a self
          intersection and it is in the
          cutoff distance                         */
14     if dist_p1_p2 < cAND dist_p1_p2 > ε then
15         ray_idx ← getPayloadRayidx()
16         is_ray_for_compute ← (point.y ≠ q.y
            AND point.z ≠ q.z)
17         OR ((point.z < q.z AND
            ray_idx == 0) OR (point.z > q.z
            AND ray_idx == 2))
18         OR ((point.y < q.y AND
            ray_idx == 0) OR (point.y > q.y
            AND ray_idx == 1))
19         OR ray_idx == 0
20         if is_ray_for_compute then
               /* Perform the computation         */
```

Moreover, we do not use the tensor cores of the A100, which is the main difference between the two GPUs, and these must be used to approach the theoretical performance.

Finally, we compare the performance between the OptiX triangles (blue), the OptiX spheres (red), and the CUDA implementation (green). The OptiX-based implementation is always slower than the CUDA implementation for d from 2 to 8. In these cases, the initialization step is negligible in the CUDA implementation, and the computation step is the main bottleneck. Consequently, avoiding the use of a grid of cells does not provide any benefit. We can also see that for $d = 8$ (see Figures 6e and 6f), the computation step is shorter for the CUDA version. This is because, in the OptiX implementations, the computation step includes not only the computation of interactions but also the computation of the intersection list using rays. Moreover, the memory access pattern is different in the OptiX-based implementations (closed particles in memory are not necessary close in space, and have different neighbors).

For $d = 16$, the OptiX-based implementations are faster for 4096 and 8192 particles (i.e., one or two particles per cell on average in the CUDA version). In this case, the computation cost is very low, and the CUDA version spends most of its time in the initialization step. However, this is less visible for $d = 32$, where only the OptiX triangles on the RTX8000 are faster for a few particles.

8 Conclusion

In this paper, we proposed exploiting ray tracing technology to compute particle interactions with a cutoff distance in a 3D environment. We described how we built the interaction list using ray intersections with spheres or triangles. Our results show that while our approach offers a small gain in the preprocessing stage by avoiding the construction of a grid of cells, it is slower than the classical approach during the computation step. However, we are convinced that such methods could deliver better performance in the future or for specific applications, and that our work will motivate the community to investigate further in this direction.

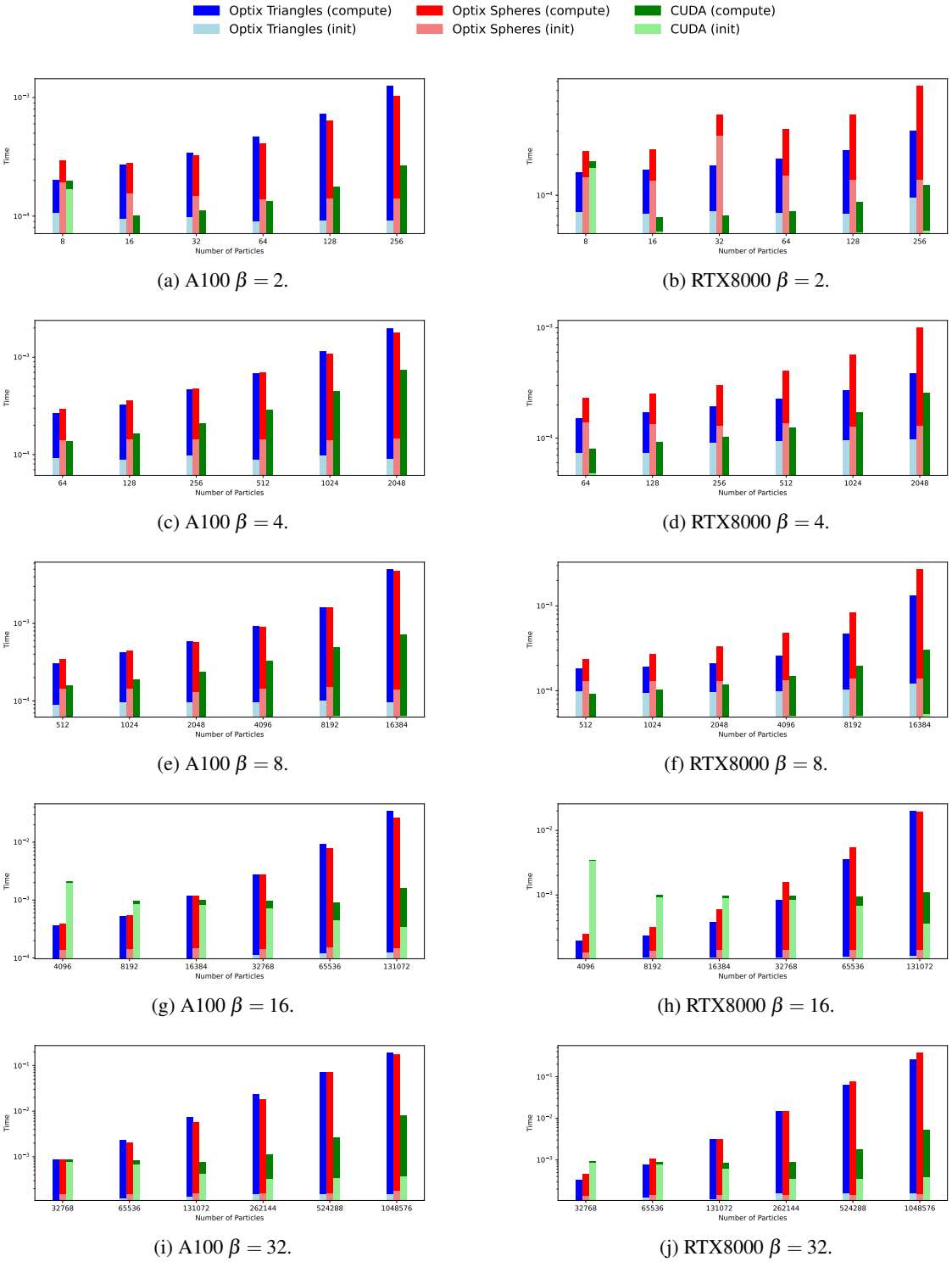
Aknoledgements

Competing interests

The author have declared that no competing interests exist.

Authors' contribution

BB conceived the idea and contributed to the formulation and evolution of the overarching research goals and aims. BB conducted the research and investigation process, including performing experiments and collecting data. Additionally, BB designed and implemented computer programs, including the development and testing of supporting algorithms. BB wrote the initial draft, revised the manuscript, and prepared the final presentation of the published work. BB also conducted the experiments, analyzed the results, and read and approved the final manuscript.



(a) A100 $\beta = 2$.

(b) RTX8000 $\beta = 2$.

(c) A100 $\beta = 4$.

(d) RTX8000 $\beta = 4$.

(e) A100 $\beta = 8$.

(f) RTX8000 $\beta = 8$.

(g) A100 $\beta = 16$.

(h) RTX8000 $\beta = 16$.

(i) A100 $\beta = 32$.

(j) RTX8000 $\beta = 32$.

Figure 6: Performance results for the two GPUs for our approaches (Optix spheres and Optix triangles) and the pure CUDA implementation that use a grid of cells (CUDA). β is the divisor coefficient of the simulation box. The cutoff distance is $1/\beta$ and there are β^3 cells in the grid in the Cuda version.

Acknowledgements

Experiments presented in this paper were carried out using the PlaFRIM experimental test-bed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr).

References

[1] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "Optix: a general purpose ray tracing engine," *ACM Trans. Graph.*, vol. 29, no. 4, jul 2010. [Online]. Available: https://doi.org/10.1145/1778765.1778803

[2] H. Nguyen, *Gpu gems 3*. Addison-Wesley Professional, 2007, chapter 31, https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda.

[3] D. Algis, B. Bramas, E. Darles, and L. Aveneau, "Efficient GPU Implementation of Particle Interactions with Cutoff Radius and Few Particles per Cell," in *International Symposium on Parallel Computing and Distributed Systems (PCDS2024)*. Singapore, Singapore: IEEE, Sep. 2024. [Online]. Available: https://inria.hal.science/hal-04621128

[4] S. Blyth, "Opticks: Gpu optical photon simulation for particle physics using nvidia® optixtm," in *EPJ Web of Conferences*, vol. 214. EDP Sciences, 2019, p. 02027.

[5] ——, "Integration of juno simulation framework with opticks: Gpu accelerated optical propagation via nvidia® optix™," in *EPJ Web of Conferences*, vol. 251. EDP Sciences, 2021, p. 03009.

[6] A. Keksel, S. Schmidt, D. Beck, and J. Seewig, "Scientific modeling of optical 3d measuring devices based on gpu-accelerated ray tracing using the nvidia optix engine," in *Modeling Aspects in Optical Metrology IX*, vol. 12619. SPIE, 2023, pp. 117–125.

[7] S. Zhao, Z. Lai, and J. Zhao, "Leveraging ray tracing cores for particle-based simulations on gpus," *International Journal for Numerical Methods in Engineering*, vol. 124, no. 3, pp. 696–713, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.7139

Appendix

I Discussion on the sphere model

In Figure 7, we show the different possibilities depending on the radius r and ray's length l. As it can be seen, even if the sources located at a distance of C from the target could have their spheres that intesecte with the rays when $r > l$, we must set $l = r$ to ensure that the rays will intersect with the sphere in all cases, especially when the source and target are close.

In our filtering algorithm, we consider that there is always at least one ray that is intersected once by the sphere and that it is the closest one to the center of the sphere. This can be demonstrated as follow.

2D case Let us consider a cercle of radius r centered at (x_c, y_c), and a cross centered at the origin with a length of r in all directions. We consider the cases where the cercle is located in the first quarter, i.e., $0 \leq x_s \leq r$ and $0 \leq y_s \leq r$, but the demonstration remains valid for other quarters.

The equation of a cercle is given by:

$$(x - x_c)^2 + (y - y_c)^2 = r^2. \tag{1}$$

The coordinates of the intersection points of the cercle with the axis are given by:

$$\begin{aligned} x_0 &= x_c - \sqrt{r^2 - y_c^2} \quad \text{and} \quad x_1 = x_c + \sqrt{r^2 - y_c^2} \\ y_0 &= y_c - \sqrt{r^2 - x_c^2}. \quad \text{and} \quad y_1 = y_c + \sqrt{r^2 - x_c^2} \end{aligned} \tag{2}$$

We provide the Figure 8 that shows where these points are located on the sphere and the cross.

x_0 and y_0 are the coordinates of the intersection points of the cercle that remain on the cross as the cercle get away from the origin. On the other hand, x_1 and y_1 are the coordinates of the intersection points of the cercle that are the farthest from the origin of the cross and that can potentially be too far to remain on the cross (they will be on the corresponding axis but behind l).

Lemma 1. *Given a circle C and let e_x and e_y be the two bars of the cross of length less than r, then the number of intersections of C with e_x or with e_y is strictly less than* 2*.*

Proof: As we consider that it is impossible that both x_1 and y_1 exist at the same time, we consider that these two equations cannot be true at the same time

$$x_c + \sqrt{r^2 - y_c^2} \leq r \quad \text{and} \quad y_c + \sqrt{r^2 - x_c^2} \leq r. \tag{3}$$

Which can be simplified as

$$\begin{aligned} x_c < r - \sqrt{r^2 - y_c^2} \quad \text{and} \quad & \sqrt{r^2 - x_c^2} \leq r - y_c \\ & r^2 - x_c^2 \leq r^2 - 2ry_c + y_c^2 \\ & x_c^2 > 2ry_c - y_c^2 \\ & x_c > \sqrt{2ry_c - y_c^2} \end{aligned} \tag{4}$$

Ending up with

$$\sqrt{2ry_c - y_c^2} < r - \sqrt{r^2 - y_c^2} \tag{5}$$

For our definition range of $y_c \in [0, r]$, this equation has no solution (see Figure 9), which confirms that C will always intersect with the cross at most once for e_x or e_y.

The second statement is to show that the ray that is intersected once is the closest to the center of the sphere.

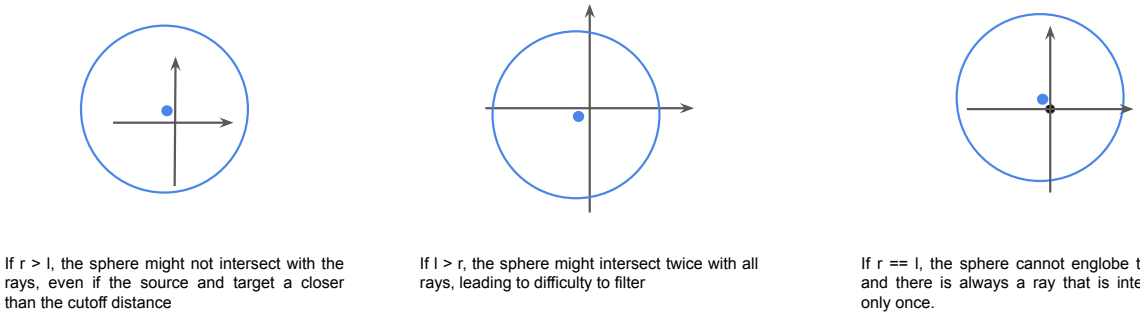


Figure 7: 2D Spherical representation of the particles in three different cases with $r > l$, $l > r$ and $r == l$.

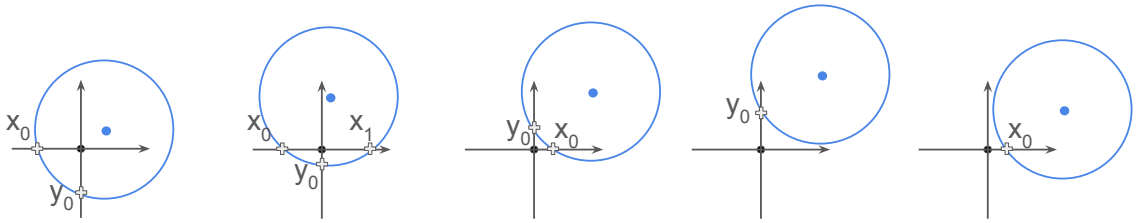


Figure 8: 2D Spherical representation of the crossing points between the sphere and the cross.

Lemma 2. *Given a circle C and e one of the two bars of the cross of length less than r, such as the number of intersection of C with e is equal to* 1*, then the distance of e with C is smaller than the distance of the other bar to C.*

Proof: Consider that C intersects with e_y twice and e_x once, it means that $y_1 < r$ and $x_1 > r$, i.e. $y_c + \sqrt{r^2 - x_c^2} < r$ and $x_c + \sqrt{r^2 - y_c^2} > r$. We aim to demonstrate that the inequality

$$y_c + \sqrt{r^2 - y_c^2} < x_c + \sqrt{r^2 - x_c^2} \tag{6}$$

holds if and only if $x_c > y_c$.

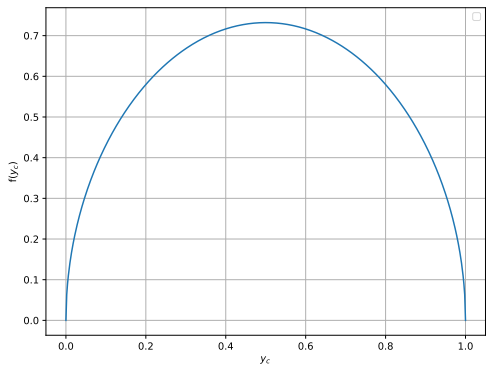


Figure 9: Plot of the equation $\sqrt{2ry_c - y_c^2} - r + \sqrt{r^2 - y_c^2}$, for r = 1.

We start with the inequality:

$$y_c + \sqrt{r^2 - y_c^2} < x_c + \sqrt{r^2 - x_c^2}. \tag{7}$$

Subtracting y_c from both sides, we obtain:

$$\sqrt{r^2 - y_c^2} < x_c - y_c + \sqrt{r^2 - x_c^2}. \tag{8}$$

We can further simplify this to:

$$\sqrt{r^2 - y_c^2} - \sqrt{r^2 - x_c^2} < x_c - y_c. \tag{9}$$

The inequality now compares two quantities: $\sqrt{r^2 - y_c^2} - \sqrt{r^2 - x_c^2}$ and $x_c - y_c$.

- The term $\sqrt{r^2 - y_c^2}$ represents the horizontal distance from the point (x_c, y_c) to the vertical axis.
- The term $\sqrt{r^2 - x_c^2}$ represents the vertical distance from the point (x_c, y_c) to the horizontal axis.

Let's consider the case where $x_c > y_c$:

- If $x_c > y_c$, then $x_c - y_c > 0$.
- Additionally, $\sqrt{r^2 - y_c^2} > \sqrt{r^2 - x_c^2}$ because $y_c < x_c$.

This implies that the term $\sqrt{r^2 - y_c^2} - \sqrt{r^2 - x_c^2}$ is positive, and it is less than $x_c - y_c$, proving that the inequality holds under this condition.

Thus, for the inequality $y_c + \sqrt{r^2 - y_c^2} < x_c + \sqrt{r^2 - x_c^2}$ to hold, it is necessary that $x_c > y_c$. So, the x axis is the closest axis to the center of the sphere and is intersected once.

3D case To convert this proof from 2D to 3D, we need to extend the concepts from the circle and cross to a sphere and a coordinate axis-aligned cross in three dimensions. Consider a sphere with radius r centered at (x_c, y_c, z_c) in 3D space, and a cross (or coordinate axes) centered at the origin with each axis extending from $-l$ to l. We are interested in analyzing the intersection of the sphere with the axes, focusing particularly on the first octant where $0 \leq x_c \leq l$, $0 \leq y_c \leq l$, and $0 \leq z_c \leq l$.

The equation of the sphere is given by:

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2. \tag{10}$$

The coordinates of the intersection points of the sphere with the axes are found by setting two of the coordinates to zero in the sphere's equation:

- Intersection with the x-axis (set $y = 0$ and $z = 0$):

$$x = x_c \pm \sqrt{r^2 - y_c^2 - z_c^2} \tag{11}$$

- Intersection with the y-axis (set $x = 0$ and $z = 0$):

$$y = y_c \pm \sqrt{r^2 - x_c^2 - z_c^2} \tag{12}$$

- Intersection with the z-axis (set $x = 0$ and $y = 0$):

$$z = z_c \pm \sqrt{r^2 - x_c^2 - y_c^2} \tag{13}$$

Let's denote the intersection points on the positive half of the axes as:

- $x_1 = x_c + \sqrt{r^2 - y_c^2 - z_c^2}$
- $y_1 = y_c + \sqrt{r^2 - x_c^2 - z_c^2}$
- $z_1 = z_c + \sqrt{r^2 - x_c^2 - y_c^2}$

We need to analyze whether these points lie within the bounds of the cross, i.e., whether $x_1 \leq l$, $y_1 \leq l$, and $z_1 \leq l$.

Assume that one of these coordinates, say x_1, exceeds l. This would mean that the intersection does not lie on the cross, i.e., $x_c + \sqrt{r^2 - y_c^2 - z_c^2} > l$. Similarly, for y_1 and z_1, we require that:

$$y_c + \sqrt{r^2 - x_c^2 - z_c^2} > l \quad \text{or} \quad z_c + \sqrt{r^2 - x_c^2 - y_c^2} > l. \tag{14}$$

These conditions cannot all be true simultaneously for x_c, y_c, and z_c within the defined range, similar to the 2D case. Thus, a sphere will intersect the cross at most once per axis.

Next, we determine the axis closest to the sphere's center. If $x_c > y_c > z_c$, we aim to prove that the intersection on the x-axis occurs first (i.e., is the smallest).

Starting with:

$$x_c + \sqrt{r^2 - y_c^2 - z_c^2} < y_c + \sqrt{r^2 - x_c^2 - z_c^2}$$
$$\text{and} \quad x_c + \sqrt{r^2 - y_c^2 - z_c^2} < z_c + \sqrt{r^2 - x_c^2 - y_c^2}. \tag{15}$$

These can be simplified, following similar steps as in the 2D case:

$$\sqrt{r^2 - y_c^2 - z_c^2} - \sqrt{r^2 - x_c^2 - z_c^2} < y_c - x_c$$
$$\text{and} \quad \sqrt{r^2 - y_c^2 - z_c^2} - \sqrt{r^2 - x_c^2 - y_c^2} < z_c - x_c. \tag{16}$$

The argument follows that since $x_c > y_c > z_c$, the inequalities hold true, confirming that the x-axis is the closest, and thus it is intersected first.

The 3D extension of the proof shows that a sphere intersects each axis of a coordinate cross at most once, and the axis closest to the sphere's center (in the order of $x_c > y_c > z_c$) will have the intersection point closest to the origin.

II Convertion from particles's positions to triangles

```
for(int i = 0; i < nbPoints; i++)
{
    const float3 point = points[i];
    std::array<float3, 8> corners;
    for(int idxCorner = 0 ; idxCorner < 8 ; ++idxCorner){
        corners[idxCorner].z = point.z + (idxCorner&1 ? (cutoffRadius/2)+epsilon : (-cutoffRadius/2)-epsilon );
        corners[idxCorner].y = point.y + (idxCorner&2 ? (cutoffRadius/2)+epsilon : (-cutoffRadius/2)-epsilon );
        corners[idxCorner].x = point.x + (idxCorner&4 ? cutoffRadius/2 : -cutoffRadius/2 );
    }
    vertices.push_back(corners[0]);
    vertices.push_back(corners[1]);
    vertices.push_back(corners[3]);

    vertices.push_back(corners[0]);
    vertices.push_back(corners[2]);
    vertices.push_back(corners[3]);

    vertices.push_back(corners[4]);
    vertices.push_back(corners[5]);
    vertices.push_back(corners[7]);

    vertices.push_back(corners[4]);
    vertices.push_back(corners[6]);
    vertices.push_back(corners[7]);
}
```

Code 1: Triangles generation from particles's positions.