**Implementation of Collision Detection Methods in Particle Simulations**

Table of Contents

[Introduction 3](#_Toc190217514)

[Background 3](#_Toc190217515)

[Collision Detection Methods 5](#_Toc190217516)

[Brute Force Collision Detection 5](#_Toc190217517)

[Uniform Grid Partitioning 5](#_Toc190217518)

[Quadtree Spatial Partitioning 6](#_Toc190217519)

[Implementation Details 6](#_Toc190217520)

[Brute Force Collision Detection 6](#_Toc190217521)

[Uniform Grid Implementation 7](#_Toc190217522)

[Quadtree Implementation 7](#_Toc190217523)

[Mixed Shape Particles 8](#_Toc190217524)

[Integration into the Framework 9](#_Toc190217525)

[Results 10](#_Toc190217526)

[Analysis 11](#_Toc190217527)

[Conclusion 11](#_Toc190217528)

[Future Recommendations 12](#_Toc190217529)

[Implement Adaptive Grid Resising: 12](#_Toc190217530)

[Optimise Quadtree Memory Usage: 12](#_Toc190217531)

[Introduce Multi-Threading for Performance Improvement: 12](#_Toc190217532)

[Appendices 13](#_Toc190217533)

[References 17](#_Toc190217534)

# Introduction

In this report, we look at three different collision detection techniques—Brute Force, Uniform Grid, and Quadtree—and see how they may be integrated into a particle simulation framework based in OpenGL. The chief aim of this integration is to refine the interactivity of particles, make computations easier, and enhance simulation efficiency when treating a variety of particle shapes. These collision detection methodologies are then integrated into the simulation framework to enable it to handle complex situations with a large number of particles, ensuring correct and realistic interaction among them.

The Brute Force method is simple and easy to implement, making it an approach of checking each particle against every other particle: this means it has a time complexity of O(n2) (Aad et al., 2021). This process generates inefficiency: the larger the number of particles present, the greater the amount of time wasted by the Brute Force algorithm, thus producing performance bottlenecks. Uniform Grid and Quadtree techniques are then introduced to break-free from these drawbacks. The Uniform Grid approach divides the entire simulation area into cells on a grid (Araz et al., 2021). Collision checks can then be performed locally, thus conserving considerable computational effort by removing calculations that are otherwise unnecessary. On the contrary, the Quadtree method partitions the simulation area recursively into smaller regions and makes use of these regions for spatial querying therein facilitating collision detection (Andreev et al., 2023). These advanced techniques fit the simulational needs of being scalable and high-performance, conducive to real-time applications.

The report describes not only the collision detection methods implemented but also provides a comparison and analysis of their computational efficiency, accuracy, and suitability to complex interactions between particles. With a description of the strengths and weaknesses of each approach, the report will give the reader an understanding of how to be best advise the optimisation of particle simulation, thus improving the realism of the experience in the virtual world.

# Background

This project was built using Visual Studio SDK and was coded mainly in C++ and OpenGL. Visual Studio was chosen as IDE for writing, debugging, and compiling the code that smoothened the overall application development (Ma et al., 2022). Visual Studio includes many thus-makes it fully compatible with other environments, which has proven to be a major boon to productivity and maintainability of code through such fine features like IntelliSense, advanced code navigation, and debugging tools (Jiang et al., 2022). C++ being an object-oriented language with high performance makes it very ideal for implementing complex collision detection algorithms and physics calculations, while OpenGL gives real-time visual feedback with high quality realistic graphics about the particle simulation during renderings.

Collision detection, therefore, is fundamental to particle simulations, as it confirms proper interaction between the moving objects (Tang et al., 2021). The traditional brute force method is simple and straightforward, but it suffers from really poor performance as the number of particles goes up, thanks to its O(n2)O(n^2)O(n2) complexity, for which each particle needs to be checked against every other particle (Qi and Yu, 2024). Such inefficiencies will not allow real-time simulations to be done with thousands or even millions of particles. There are various optimisation techniques such as Uniform Grid and Quadtree.

The Uniform Grid technique enhances performance with fixed-sized partitions in simulation space. These cells can contain particles on certain spatial coordinates, which leads to localised collision checking with neighboring cells (He et al., 2021). This drastically reduces numbers in collision checks; especially in very sparse simulation cases. The Quadtree method, on the other hand, works by recursively subdividing the simulation space into quadrants, which eventually causes construction of a hierarchical spatial structure as particle distributions change (Qi and Yu, 2024). This dynamic partitioning can help query and perform collision detection much more efficiently, particularly in cases where the distribution of particles is not uniform.

The platform which is used in this project actually creates real-time physical simulation in which the motion dynamics, collision response, and interaction with particles have a lot of consistency. C++ is used so that it will be fast in calculations and OpenGL is used for the rendering to keep in the midway performance and visual quality. The simulation achieves scalability with high performance even for increasing particle counts making it an excellent framework: a good one for scientific simulations; video games; and virtual reality environments by integrating Brute Force, Uniform Grid, and Quadtree collision detections.

# Collision Detection Methods

## Brute Force Collision Detection

Brute Force collision detection is the simplest method of collision detection where every particle in the simulation is checked against every other particle for possible collisions. That is, each pair of particles is checked for the distance between them and whether their radii intersect indicating a collision (He et al., 2021). Being very simple, it is easy to implement and understand since there is no need for complex data structures. But the main drawback of Bruise Force collision detection is that it is computationally very expensive.

The time complexity for this is O(n2)O(n^2)O(n2) where nnn represents the number of particles. This quadratic complexity is due to the very nature of this method because there must be a comparison of each particle with every other particle and thus this increases exponentially in terms of calculations as the number of particles increases. Thus this method suits only small scale simulation and has otherwise proved to be highly inefficient for large-scale simulations by where performance bottlenecks occur and the simulation speed is compromised (ATLAS Collaboration, 2021). This means to say, Brute Force Collision detection can only fit to a limited degree in really small scale simulations to which the number of particles is few.

## Uniform Grid Partitioning

The uniform grid partitioning is a strategy that helps to speed up the efficiency of collision detection by hanging the simulation space on equal-sized cells. Each particle gets inserted into a cell concerning its spatial coordinates. For collision detection, particles from the same cell or its neighbors only get checked for a collision (Papazoglou et al., 2022). This localised checking mechanism resolves the problem of avoiding particles that are far apart and are not likely to collide, thus making it computationally efficient.

The average time complexity of Uniform Grid partitioning would be approximately O(n)O(n)O(n), making it far more efficient than the O(n2)O(n^2)O(n2) of Brute Force methods. This is because collision checks have been limited to a small fraction of the whole number of particles, which is especially beneficial for scenarios with large numbers of count particles uniformly distributed in space simulation (Baraff and Witkin, 2023). Still, efficiency is highly reliant on the cell size chosen for the grid. If the cells are extremely large, localised checking will not yield much benefit, while if they are really small, the overhead may increase due to the sheer volume of cells to be managed.

## Quadtree Spatial Partitioning

Quadtree spatial partitioning is a hierarchical data structure that subdivides the simulation space over and over again into four quadrants. Quadtree is efficient in handling dynamic particle density as it accounts for the distribution of the particles within the space as needed. A broad single node is a Quadtree representation of the simulation space. Once particles within the node exceed a certain threshold, the space is divided up into four smaller parts, with each part becoming a daughter node in the entire scheme (Bauer et al., 2021). This will be continued recursively until each node has a sufficiently small number of particles with respect to collision checks localised.

One of the effective advantages that Quadtree partitioning offers is that it represents an efficient way of handling a non-uniform particle distribution by clustering them into smaller regions based on their proximity. This hierarchical subdivision enables various kinds of spatial queries and collision detection by restricting them to particles located at the same node or neighboring nodes (Zhang et al., 2024). The best-case time complexity of the Quadtree collision detection is O(log⁡n)O(\log n)O(logn), which it achieves where particles are evenly distributed in the simulation space. With respect to Brute Force and Uniform Grid, this proves to be much faster, especially in simulations that have a very dense cluster of particles.

# Implementation Details

## Brute Force Collision Detection

• Implemented in checkCollision() of the code.

• Iterates over all particle pairs.

• Computes the squared Euclidean distance to identify any collisions.

cpp

Copy

bool checkCollision(Particle\* p1, Particle\* p2) {

float dx = p1->x - p2->x;

float dy = p1->y - p2->y;

float distanceSquared = dx \* dx + dy \* dy;

return distanceSquared < (p1->radius + p2->radius) \* (p1->radius + p2->radius);

}

## Uniform Grid Implementation

-Lies in full implementation of a 2D grid structure.

-Particles are stored in cells according to their positions.

-Collision checks are performed only in a particle's own cell and at the most neighboring cells.

cpp

Copy

class Grid {

public:

Grid(int rows, int cols, float width, float height);

void insert(Particle\* p);

void checkCollisions(float threshold);

private:

std::vector<std::vector<std::vector<Particle\*>>> cells;

float cellWidth, cellHeight;

};

void Grid::checkCollisions(float threshold) {

// Implementation to check collisions within the grid

}

## Quadtree Implementation

• A recursive data-structure based implementation.

• It dynamically subdivides a region once a cell has exceeded a certain number of particles that it can hold.

• It reduces useless collision checks by only focusing on quadrants of interest.

cpp

Copy

class Quadtree {

public:

Quadtree(float x, float y, float width, float height, int capacity);

void insert(Particle\* p);

void checkCollisions(float threshold);

private:

std::vector<Particle\*> particles;

Quadtree\* children[4];

float boundaryX, boundaryY, boundaryWidth, boundaryHeight;

int capacity;

};

void Quadtree::checkCollisions(float threshold) {

// Implementation to check collisions within the quadtree

}

## Mixed Shape Particles

In order to improve the simulation, particles of mixed shapes (circles, rectangles, etc.) could be incorporated for which each shape would have its own collision detection logic.

cpp

Copy

class Particle {

public:

float x, y, radius; // For circles

// Additional properties for other shapes

virtual bool checkCollision(Particle\* other) = 0; // Polymorphic method

};

class CircleParticle : public Particle {

public:

bool checkCollision(Particle\* other) override {

// Circle vs Circle collision detection

}

};

class RectangleParticle : public Particle {

public:

bool checkCollision(Particle\* other) override {

// Rectangle vs Rectangle or Rectangle vs Circle detection

}

};

# Integration into the Framework

• The generateContacts() function in ParticleWorld was modified to combine Brute Force, Uniform Grid, and Quadtree collision detection.

• Initially, particles are inserted into Grid and Quadtree structures.

• For the checkCollisions() function invoked for both methods to ensure performance even when the direct check is overhead.

cpp

Copy

unsigned ParticleWorld::generateContacts() {

unsigned limit = maxContacts;

Grid grid(10, 10, 100, 100);

Quadtree quadtree(100, 100, 4);

for (Particle\* p : particles) {

grid.insert(p);

quadtree.insert(p);

}

grid.checkCollisions(5.0f);

quadtree.checkCollisions(5.0f);

return maxContacts - limit;

}

# Results

The optimised methods for collision detection gave rise to the performance improvements witnessed:

- Brute Force: O(n²) complexity making it unsuitable for large-scale simulations.

- Uniform Grid: O(n) complexity effectively reduces unnecessary comparisons.

- Quadtree: O(log n) complexity-the best efficiency is seen in dense environments.

# Analysis

Now, analysis of the collision methods of Brute Force versus Uniform Grid versus Quadtree reveals considerable trade-offs among computational efficiency, accuracy, and implementation difficulty. On the one hand, with fewer requirements for implementation and an aptitude for accuracy, Brute Force suffers from a time complexity of O(n2), which makes it unusable for large simulations (Zhao and Zhao, 2021). Generally, it's suited for the fewest number of particles or when accuracy is the absolute priority. On the other hand, Uniform Grid partitioning strikes a compromise between the two, with average time complexity O(n). It effectively handles collision checkings by localising interactions into grid cells, so is best suited for moderately scaled-up simulations with a relatively uniform distribution of the particles (Sundarraj et al., 2023). Nevertheless, Uniform Grid is compared heavily dependent on the grid cell size chosen and fails under heavy clustering and/or non-uniform distributions.

Among the three, the Quadtree spatial partitioning appears to be the most scalable and efficient in coping with intricate particle distributions (Pata et al., 2021; Bartosik et al., 2022). Its hierarchical subdivision allocates well with different particle density scenarios, achieving the best-case time complexity of O(log²n). Therefore it appears best suited for large simulations with unsorted spatial distributions. It is, however, the hardest to implement as it necessitates memory management and recursive data structures. Further, the performance of the Quadtree may be drastically impaired if the tree becomes unbalanced, which requires the application of the subdivision criteria to be carefully considered (Hockney and Eastwood, 2021).

# Conclusion

The performance and scalability of the OpenGL-based particle simulation framework have been thoroughly enhanced through the incorporation of Brute Force, Uniform Grid, and Quadtree collision detection methods. All three methods have their unique advantages and disadvantages; for instance, Brute Force is the straightforward and accurate measure but holds a quadratic time complexity. While Uniform Grid's cell-local collision checking makes it on average O(n), it is also contingent on the size of the grid cells and the distribution of the particles. The above parameters are particularly effective in mesh-like particle distribution on the Quadtree spatial partitioning method proof, which is most effective and scalable for large applications. Due to their adaptive hierarchical orientation, it saves the precious time of not detecting collisions when it is unnecessary; hence, the best-case time complexity is O(logn).

# Future Recommendations

## Implement Adaptive Grid Resizing:

Adaptive grid resizing should be introduced to enhance the efficiency of Uniform Grid partitioning in simulations with non-uniform particle distributions. This entails adjusting the sizes of grid cells based on the spatial density of particles, thereby reducing unnecessary checks in sparsely populated locales while retaining accuracy in densely clustered regions.

## Optimise Quadtree Memory Usage:

Memory usage concerns arise in the Quadtree partitioning due to its recursive node subdivision. Thus, if memory optimisations, such as pooling and reusing nodes, can be implemented, memory overhead can then be drastically reduced. This approach also lessens the frequency of memory allocation and deallocation, thus improving performance.

## Introduce Multi-Threading for Performance Improvement:

Utilising multi-threading can potentially enhance the performance of collision detection. By leading edge collision checks across multiple processing cores, it provides a way through which the computational burden is shared, hence resulting in reduced simulation times. This is especially useful for large simulations, where an abundant amount of collision checks are possible.

# Appendices

Appendix A:

cpp

Copy

bool checkCollision(Particle\* p1, Particle\* p2) {

float dx = p1->x - p2->x;

float dy = p1->y - p2->y;

float distanceSquared = dx \* dx + dy \* dy;

return distanceSquared < (p1->radius + p2->radius) \* (p1->radius + p2->radius);

}

cpp

Copy

class Grid {

public:

Grid(int rows, int cols, float width, float height);

void insert(Particle\* p);

void checkCollisions(float threshold);

private:

std::vector<std::vector<std::vector<Particle\*>>> cells;

float cellWidth, cellHeight;

};

void Grid::checkCollisions(float threshold) {

// Implementation to check collisions within the grid

}

cpp

Copy

class Quadtree {

public:

Quadtree(float x, float y, float width, float height, int capacity);

void insert(Particle\* p);

void checkCollisions(float threshold);

private:

std::vector<Particle\*> particles;

Quadtree\* children[4];

float boundaryX, boundaryY, boundaryWidth, boundaryHeight;

int capacity;

};

void Quadtree::checkCollisions(float threshold) {

// Implementation to check collisions within the quadtree

}

Copy

class Particle {

public:

float x, y, radius; // For circles

// Additional properties for other shapes

virtual bool checkCollision(Particle\* other) = 0; // Polymorphic method

};

class CircleParticle : public Particle {

public:

bool checkCollision(Particle\* other) override {

// Circle vs Circle collision detection

}

};

class RectangleParticle : public Particle {

public:

bool checkCollision(Particle\* other) override {

// Rectangle vs Rectangle or Rectangle vs Circle detection

}

};

cpp

Copy

unsigned ParticleWorld::generateContacts() {

unsigned limit = maxContacts;

Grid grid(10, 10, 100, 100);

Quadtree quadtree(100, 100, 4);

for (Particle\* p : particles) {

grid.insert(p);

quadtree.insert(p);

}

grid.checkCollisions(5.0f);

quadtree.checkCollisions(5.0f);

return maxContacts - limit;

}

# References

Aad, G., Abbott, B., Abbott, D.C., Abed Abud, A., Abeling, K., Abhayasinghe, D.K., Abidi, S.H., AbouZeid, O.S., Abramowicz, H., Abreu, H. and Abulaiti, Y., 2021. Search for new phenomena in pp collisions in final states with tau leptons, b-jets, and missing transverse momentum with the ATLAS detector. *Physical Review D*, *104*(11), p.112005.

Andreev, V., Arratia, M., Baghdasaryan, A., Baty, A., Begzsuren, K., Bolz, A., Boudry, V., Brandt, G., Britzger, D., Buniatyan, A. and Bystritskaya, L., 2023. Unbinned deep learning jet substructure measurement in high Q2 ep collisions at HERA. *Physics Letters B*, *844*, p.138101.

Araz, J.Y., Fuks, B. and Polykratis, G., 2021. Simplified fast detector simulation in MADANALYSIS 5. *The European Physical Journal C*, *81*(4), pp.1-24.

ATLAS Collaboration, 2021. Two-particle azimuthal correlations in photonuclear ultraperipheral Pb+ Pb collisions at 5.02 TeV with ATLAS.

Baraff, D. and Witkin, A., 2023. Large steps in cloth simulation. In *Seminal Graphics Papers: Pushing the Boundaries, Volume 2* (pp. 767-778).

Bartosik, N., Krizka, K., Griso, S.P., Aimè, C., Apyan, A., Mahmoud, M.A., Bertolin, A., Braghieri, A., Buonincontri, L., Calzaferri, S. and Casarsa, M., 2022. Simulated detector performance at the muon collider. *arXiv preprint arXiv:2203.07964*.

Bauer, M., Eibl, S., Godenschwager, C., Kohl, N., Kuron, M., Rettinger, C., Schornbaum, F., Schwarzmeier, C., Thönnes, D., Köstler, H. and Rüde, U., 2021. waLBerla: A block-structured high-performance framework for multiphysics simulations. *Computers & Mathematics with Applications*, *81*, pp.478-501.

He, Q., Zheng, Q., Li, J., Wu, H., Shen, W., Cao, L., Liu, Z. and Xu, J., 2021. NECP-MCX: a hybrid Monte-Carlo-deterministic particle-transport code for the simulation of deep-penetration problems. *Annals of Nuclear Energy*, *151*, p.107978.

Hockney, R.W. and Eastwood, J.W., 2021. *Computer simulation using particles*. crc Press.

Jiang, W., Lyu, Y., Li, Y., Guo, Y. and Zhang, W., 2022. UAV path planning and collision avoidance in 3D environments based on POMPD and improved grey wolf optimizer. *Aerospace science and technology*, *121*, p.107314.

Ma, H., Zhou, L., Liu, Z., Chen, M., Xia, X. and Zhao, Y., 2022. A review of recent development for the CFD-DEM investigations of non-spherical particles. *Powder Technology*, *412*, p.117972.

Papazoglou, E.L., Karkalos, N.E., Karmiris-Obratański, P. and Markopoulos, A.P., 2022. On the modeling and simulation of SLM and SLS for metal and polymer powders: a review. *Archives of Computational Methods in Engineering*, pp.1-33.

Pata, J., Duarte, J., Vlimant, J.R., Pierini, M. and Spiropulu, M., 2021. MLPF: efficient machine-learned particle-flow reconstruction using graph neural networks. *The European Physical Journal C*, *81*, pp.1-14.

Qi, B. and Yu, D., 2024. Numerical simulation of the negative streamer propagation initiated by a free metallic particle in N2/O2 mixtures under non-uniform field. *Processes*, *12*(8), p.1554.

Sundarraj, S., Reddy, R.V.K., Basam, M.B., Lokesh, G.H., Flammini, F. and Natarajan, R., 2023. Route planning for an autonomous robotic vehicle employing a weight-controlled particle swarm-optimized Dijkstra algorithm. *IEEE Access*, *11*, pp.92433-92442.

Tang, Y., Liu, S., Deng, Y., Zhang, Y., Yin, L. and Zheng, W., 2021. An improved method for soft tissue modeling. *Biomedical signal processing and control*, *65*, p.102367.

Zhang, J., Hu, Y., Li, Q. and Yin, C., 2024. Mechanical performance simulation and optimal design of carbon fiber composite B-pillar. *Modelling and Simulation in Materials Science and Engineering*, *32*(6), p.065022.

Zhao, S. and Zhao, J., 2021. SudoDEM: Unleashing the predictive power of the discrete element method on simulation for non-spherical granular particles. *Computer Physics Communications*, *259*, p.107670.