Simulation in Computer Graphics

Shuhei Watanabe

March 31, 2021

1 Patricle motion

In the simulation of particle motion, there are three main tasks:

1. **Object Subdivision**: Divide objects into small components [1]
2. **Force Modeling**[2]: Introduce the model that dominates the motion of each component (e.g. Newton's 2nd law)
3. **Particle Motion Reproduction**: Solve the ordinary differential equation (ODE) to compute the position and the velocity at each time step

1.1 Object Subdivision

To compute the position and the velocity of object accurately, we divide the object of interest into small particles and compute the forces or the accelerations of each particle. Each particle has **mass** $m \in \mathbb{R}$, **volume** $V \in \mathbb{R}$, **density** $\rho \in \mathbb{R}$, **position** $\boldsymbol{x} \in \mathbb{R}^3$, **velocity** $\boldsymbol{v} \in \mathbb{R}^3$, and **force** $\boldsymbol{F} \in \mathbb{R}^3$. To compute the positions and the velocities at each iteration, we have to store these variables.

1.2 Particle Motion Reproduction

As mentioned in the previous section, each particle is governed by given force models and positions and velocities of each particle are computed based on given initial values and given force models. Since the positions and the velocities change over time and we have to consider **the existence of other particles**, the computation must be implemented using time discretization [3]. For example, the two objects tied by spring has to consider the interaction. In other words, the acceleration of a particle will be computed using not only its position and velocity, but also others' positions and velocities i.e. $\boldsymbol{a} = \boldsymbol{a}(\boldsymbol{x}_{\mathrm{self}}, \boldsymbol{v}_{\mathrm{self}}, \boldsymbol{x}_{\mathrm{others}}, \boldsymbol{v}_{\mathrm{others}})$.

1.3 Finite Differences

Since each particle motion is governed by ODE, we have to consider the interaction with other particles as well. The major solution is **Finite Difference Equation (FDE)**. In FDE, we consider the following assumption:

Assumption 1
Suppose ϵ is a real number and sufficiently small, then the function $f(x+\epsilon)$ can be approximated by the first order of Taylor series approximation, i.e.

$$f(x+\epsilon) \simeq f(x) + \frac{df(x)}{dx}\epsilon$$

where Taylor series approximation is $f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x)\epsilon^n}{n!}$

[1]The shape of components are arbitrary.
[2]A variety of models can be considered such as gravity, friction, compression.
[3]Even two-body problems are considered hard to solve analytically.

Using this assumption, derivatives of a given function can be trivially introduced as follows:

$$f(t+\Delta t) \simeq f(t) + \frac{df(t)}{dt}\Delta t \Rightarrow \frac{df(t)}{dt} \simeq \frac{f(x+\Delta t) - f(x)}{\Delta t} \tag{1}$$

The first equation of Eq. (1) is called **Explicit Euler** (e.g. $\boldsymbol{x}(t+\Delta t) \simeq \boldsymbol{x}(t) + \boldsymbol{v}\Delta t$) [4]. This equation includes the $O(\Delta t^2)$ error term.

There are many variants of FDE methods and the common goal is to eliminate all the terms whose dimension is less than k. Depending on this k, the accuracy of the method varies. Note that the computational complexity at each iteration is mostly dominated by **the computation of accelerations**. For this reason, it is important to consider the trade-off between the number of computation of accelerations [5] and the accuracy. Hereinafter, we denote $\boldsymbol{x}_i = \boldsymbol{x}(i\Delta t)$. In most cases, we will use the following series:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \frac{d\boldsymbol{x}_i}{dt}\Delta t + \frac{d^2\boldsymbol{x}_i}{dt^2}\frac{\Delta t}{2} + \frac{d^3\boldsymbol{x}_i}{dt^3}\frac{\Delta t^3}{6} + \frac{d^4\boldsymbol{x}_i}{dt^4}\frac{\Delta t^4}{24} + O(\Delta t^5) \tag{2}$$

1.3.1 Midpoint Method

$$\frac{d\boldsymbol{x}_{i+\frac{1}{2}}}{dt} = \frac{d\boldsymbol{x}_i}{dt} + \frac{d^2\boldsymbol{x}_i}{dt^2}\frac{\Delta t}{2} + O(\Delta t^2) \tag{3}$$

From (2) − (3) × Δt,

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \boldsymbol{v}_{i+\frac{1}{2}}\Delta t + O(\Delta t^3)$$

Note that $\boldsymbol{v}_i = \frac{d\boldsymbol{x}_i}{dt}$ holds in general.

1.3.2 Heun Method

$$\frac{d\boldsymbol{x}_{i+1}}{dt} = \frac{d\boldsymbol{x}_i}{dt} + \frac{d^2\boldsymbol{x}_i}{dt^2}\Delta t + O(\Delta t^2) \tag{4}$$

From (2) − (4) × $\Delta t/2$,

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + (\boldsymbol{v}_{i+1} + \boldsymbol{v}_i)\frac{\Delta t}{2} + O(\Delta t^3)$$

1.3.3 Ralston Method

$$\frac{d\boldsymbol{x}_{i+\frac{2}{3}}}{dt} = \frac{d\boldsymbol{x}_i}{dt} + \frac{d^2\boldsymbol{x}_i}{dt^2}\frac{2\Delta t}{3} + O(\Delta t^2) \tag{5}$$

From (2) − (5) × $3\Delta t/4$,

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \boldsymbol{v}_i\frac{\Delta t}{4} + \boldsymbol{v}_{i+\frac{2}{3}}\frac{3\Delta t}{4} + O(\Delta t^3)$$

1.3.4 3 / 8 Rule

$$\frac{d\boldsymbol{x}_{i+\frac{1}{3}}}{dt} = \frac{d\boldsymbol{x}_i}{dt} + \frac{d^2\boldsymbol{x}_i}{dt^2}\frac{\Delta t}{3} + \frac{d^3\boldsymbol{x}_i}{dt^3}\frac{\Delta t^2}{18} + \frac{d^4\boldsymbol{x}_i}{dt^4}\frac{\Delta t^3}{162} + O(\Delta t^4) \tag{6}$$

From (2) − $dx_i/dx \times 1/8$ − (6) × $3\Delta t/8$ − (5) × $3\Delta t/8$ − (4) × $1\Delta t/8$[6],

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \frac{1}{8}(\boldsymbol{v}_i + 3\boldsymbol{v}_{i+\frac{1}{3}} + 3\boldsymbol{v}_{i+\frac{2}{3}} + \boldsymbol{v}_{i+1})\Delta t + O(\Delta t^5)$$

[4]In generall, explicit method is the method computing the future value only with previously computed values.
[5]In the situation where we have to detect the collisions, the computational complexity will be much more expensive.
[6]I used the series up to the term of Δt^3 for each.

1.3.5 Classic Runge Kutta Method

$$\begin{aligned} k_1 &= v_i(x_i) \\ k_2 &= v_{i+\frac{1}{2}}(x_i + k_1\Delta t/2) \\ k_3 &= v_{i+\frac{1}{2}}(x_i + k_2\Delta t/2) \\ k_4 &= v_{i+1}(x_i + k_3\Delta t) \\ x_{i+1} &= x_i + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(\Delta t^5) \end{aligned}$$

1.4 Implementation and Performance

Since each particle interacts with each other and we have to consider the collisions, the positions and the velocities of each particle have to be computed at the same time. The flow of the computation in the case of Midpoint method is as follows:

1. Compute all the velocities and positions at time $t + \Delta t/2$
2. Detect the collisions and compute the forces based on obtained velocities and positions
3. Update the positions and velocities using the values at time t and $t + \Delta t/2$

The performance of simulation depends on the following three properties:

Definition 1

1. **consistent**: *If the discretization error vanishes as the time step Δt goes to zero*
2. **stable**: *If the error is not amplified within simulation steps*
3. **convergent**: *If the solution of FDE approaches that of ODE in the end*

Inconsistency is basically caused by the representational limit of numbers (e.g. 64 bit) in programming languages. For example, if the derivative is super small or large, float cannot deal with the digit and it leads to overflow or zero value. If it is not stable, velocities typically divergent. Additionally, there are two other terms for stability.

1. **Conditionally Stable**: The simulation can be stable under some conditions
2. **Unconditionally Stable**: The simulation is stable under all circumstances

The simulation is convergent **when the simulation is stable and consistent**.

It is important to consider the trade-off between a time step size and the accuracy as well. Since the update procedure is computationally expensive, we would like to take as large a time step size as possible. As mentioned above, the order of the error term in the Runge Kutta method is $O(\Delta t^5)$. Therefore, the method can take a larger time step size compared to explicit Euler. For example, while explicit Euler achieves the error order of $1/400$ by taking $\Delta t = 1/20$, the Runge Kutta method achieves the same order by $\Delta t = 3/10$. It means the Runge Kutta method can finish the simulation with six times less iterations [7].

[7] This discussion makes sense **if and only if the simulation is stable**.

1.5 Predictor-corrector schemes

This method literally performs the prediction of accelerations and the correction of the accelerations. Therefore, it **approximates accelerations twice** for each iteration. One benefit of this method is that it can achieves arbitrary accuracy using data from previous time steps. In fact, as seen in the explicit schemes, we can eliminate up to $O(\Delta t^k)$ when we use data from $k-1$ different time steps as follows:

$$\begin{aligned}
\boldsymbol{v}_{i+1}^{\text{predict}} &= \boldsymbol{v}_i + \frac{\Delta t}{2}(3\boldsymbol{a}_i - \boldsymbol{a}_{i-1}) + O(\Delta t^3)\\
\boldsymbol{v}_{i+1}^{\text{predict}} &= \boldsymbol{v}_i + \frac{\Delta t}{12}(23\boldsymbol{a}_i - 16\boldsymbol{a}_{i-1} + 5\boldsymbol{a}_{i-2}) + O(\Delta t^3)\\
&\dots\\
\boldsymbol{x}_{i+1}^{\text{predict}} &= \boldsymbol{x}_i + \frac{\Delta t}{2}(3\boldsymbol{v}_i - \boldsymbol{v}_{i-1}) + O(\Delta t^3)\\
\boldsymbol{x}_{i+1}^{\text{predict}} &= \boldsymbol{x}_i + \frac{\Delta t}{12}(23\boldsymbol{v}_i - 16\boldsymbol{v}_{i-1} + 5\boldsymbol{v}_{i-2}) + O(\Delta t^3)\\
&\dots
\end{aligned}$$

These coefficients can be easily derived in the same vein for explicit schemes. This method modifies $\boldsymbol{v}_{i+1}^{\text{predict}}$ in so-called the correction step.

$$\begin{aligned}
\boldsymbol{v}_{i+1} &= \boldsymbol{v}_i + \frac{\Delta t}{2}(\boldsymbol{a}(\boldsymbol{x}_{i+1}^{\text{predict}}, \boldsymbol{v}_{i+1}^{\text{predict}}) + \boldsymbol{a}_i) + O(\Delta t^3)\\
\boldsymbol{x}_{i+1} &= \boldsymbol{x}_i + \frac{\Delta t}{2}(\boldsymbol{v}_{i+1}^{\text{predict}} + \boldsymbol{v}_i) + O(\Delta t^3)
\end{aligned}$$

This step can also achieve better accuracy if you stored much more previous time steps. This order of the correction scheme is same as the Heun method.

In summary, this method requires twice approximation so-called prediction and correction steps, so the time complexity is twice expensive. However, **it can achieve arbitrary accuracy** if we store as much previous data as possible. Note that this accuracy is an indicator, and it totally depends on the stability and discontinuity of simulations. Plus, since this method requires the data from previous time steps, we have to **approximate the data before the initial state**, which can be obtained by the backward-Euler scheme. Lastly, the correction step can be performed several times to improve the accuracy.

1.6 Implicit scheme and linearization

Implicit schemes for velocities use the future accelerations.

1. **Implicit schemes**: More stable, but it cannot be processed directly because of the non-linearity of accelerations
2. **Linearization**: The solution for the non-linearity of the implicit scheme
3. **Linear system solver**: The solution for linear system with sparse and huge (, that is why practically not invertible) matrix

1.6.1 Implicit schemes

The explicit scheme has the following form:

$$\begin{aligned}
\boldsymbol{x}_{i+1} &= \boldsymbol{x}_i + \boldsymbol{v}_i \Delta t\\
\boldsymbol{v}_{i+1} &= \boldsymbol{v}_i + \boldsymbol{a}_i \Delta t
\end{aligned}$$

where $\boldsymbol{x}_{i+1} = \boldsymbol{x}(i\Delta t)$. On the other hand, the implicit scheme is the following form:

$$\begin{aligned}\boldsymbol{x}_{i+1} &= \boldsymbol{x}_i + \boldsymbol{v}_{i+1}\Delta t\\ \boldsymbol{v}_{i+1} &= \boldsymbol{v}_i + \boldsymbol{a}_{i+1}\Delta t\end{aligned}$$

Basically, the implicit scheme uses future values to approximate future values and does not use the current velocities as in the Heun method. Therefore, the implicit scheme can typically avoid the influences from collisions that are supposed to happen between the current and the successive time steps. The greatest benefit of the implicit scheme is the stability. However, since we use future accelerations and thus have to solve a huge linear system in the implicit scheme, this scheme requires the linearization and the solver.

1.6.2 Linearization

To linearlize future accelerations, we assume that **accelerations only depends on positions, but not on velocities**. By introducing such an assumption, the linear system with respect to future velocities can be obtained as follows:

$$\begin{aligned}\boldsymbol{v}_{i+1} &= \boldsymbol{v}_i + \boldsymbol{a}_{i+1}(\boldsymbol{x}_{i+1})\Delta t\\ \boldsymbol{v}_{i+1} &= \boldsymbol{v}_i + \boldsymbol{a}_{i+1}(\boldsymbol{x}_i + \boldsymbol{v}_{i+1}\Delta t)\Delta t \;(\because \text{Implicit Euler})\\ \boldsymbol{v}_{i+1} &= \boldsymbol{v}_i + (\boldsymbol{a}_i + \boldsymbol{J}\boldsymbol{v}_{i+1}\Delta t)\Delta t\end{aligned}$$

For the last transformation, we use the Taylor series approximation $\boldsymbol{a}_{i+1}(\boldsymbol{x}_i + \boldsymbol{v}_{i+1}\Delta t) = \boldsymbol{a}_i + (\nabla \boldsymbol{a}_i)\boldsymbol{v}_{i+1}\Delta t + O(v_{i+1}^2\Delta t^2)$ and $\boldsymbol{a}_{i+1}(\boldsymbol{x}_i) = \boldsymbol{a}_i$. Additionally, $\boldsymbol{J}$ is a jacobian matrix defined as follows:

$$\boldsymbol{J} = \left[\frac{\partial \boldsymbol{a}}{\partial x}, \frac{\partial \boldsymbol{a}}{\partial y}, \frac{\partial \boldsymbol{a}}{\partial z}\right]$$

Therefore, the following linear system can be obtained:

$$(I_3 - \Delta t^2 \boldsymbol{J})\boldsymbol{v}_{i+1} = \boldsymbol{v}_i + \boldsymbol{a}_i\Delta t$$

where $I_K \in \mathbb{R}^{K\times K}$ is an identity matrix. Since we consider the interaction of each particle and each position affects each acceleration, practically we also have to consider the jacobian with respect to other particles. Therefore, this notation can be expanded to the N particles and we obtain the following linear system:

$$(I_{3N} - \Delta t^2 \boldsymbol{J})\boldsymbol{V}_{i+1} = \boldsymbol{V}_i + \boldsymbol{A}_i\Delta t$$

where $\boldsymbol{V}_i \in \mathbb{R}^{3N}$ is the vector that stacks velocities of each particle and $\boldsymbol{A}_i \in \mathbb{R}^{3N}$ is the vector that stacks accelerations of each particle. Furthermore, the jacobian matrix is the following:

$$\begin{bmatrix}\boldsymbol{J}_1 & & & \\ & \boldsymbol{J}_2 & & \\ & & \ddots & \\ & & & \boldsymbol{J}_N\end{bmatrix}$$

where $\boldsymbol{J}_i$ $(1 \leq k \leq N)$ is the jacobian matrix for the k-th particle. The jacobian matrix for each particle is placed on the diagonal elements of $\boldsymbol{J}$ and other elements are the interacting terms, which are the influence from adjacent particles. The computation of the inverse matrix costs $O(N^3)$ and it is not feasible for large scale particle simulations. However, the combination of adjacent particles [8] is assumed to be small enough; therefore, this matrix is **sparse** and it can be solved using iterative methods. After solving the linear system, we obtain the position $\boldsymbol{x}_{i+1}$ using the solution $\boldsymbol{v}_{i+1}$. Note that each data $(\boldsymbol{J}_i, \boldsymbol{v}_i, \boldsymbol{a}_i)$ is held by each particle and interactive terms $\boldsymbol{J}_{i,j}$ are held by adjacent connections.

[8]If the i-th and the j-th particles are not adjacent each other, the submatrix $\boldsymbol{J}_{i,j}$ is zero matrix.
[9]If $A \in \mathbb{R}^n$ is positive definite and symmetric, it will converge in n step.

Table 1: Pros and cons of Conjugate gradient and Jacobi method

-	Conjugate gradient	Jacobi method
Pros	Fast and guarantee of convergence[9]	larger variety of applicability
	Often used for deformable objects	parallelizable
Cons	Conditional guarantee of convergence	Needs to tune parameter ω
	Not widely used compared to Jacobi method	Slower and longer iterative steps to converge

1.6.3 Solver for large and sparse linear system

In this section, we discuss how to solve $A\boldsymbol{x} = \boldsymbol{b}$ and we use this notation $A\boldsymbol{x} = \boldsymbol{b}$. When the size of $\boldsymbol{x}$ is large, it is not feasible to invert the matrix A ($\because$ the complexity is $O(n^3)$ when $A \in \mathbb{R}^{n\times n}$ [10]). For this reason, we approximate the solution $\boldsymbol{x}$ using the following iterative methods:

1. **Conjugate gradient**: Minimizes $\|A\boldsymbol{x} - \boldsymbol{b}\|^2$ using gradient descent while being the conjugate
2. **Jacobi method**: Decomposes A into the diagonal elements D, upper triangle elements U, lower triangle elements L (, where $A = D + L + U$) and iteratively optimize $\boldsymbol{x}$

The advantages and the disadvantages of each method are listed in Table 1. Note that **both methods are matrix-free method**, so we can apply this method to sparse matrices.

1.7 Semi-implicit Scheme

This method is a combination of implicit and explicit schemes. Velocities are computed by an explicit scheme and positions are updated by an implicit scheme. Therefore, this method does not require linear system solvers.

1.8 Second-order ODE

Second-order ODEs do not necessarily require velocities for update. The underlying concept for the second-order schemes are the equation between positions and accelerations as follows:

$$\boldsymbol{a}_i = \frac{\partial^2 \boldsymbol{x}_i}{\partial t^2}$$

The Taylor series approximation of $\boldsymbol{x}_{i+1}, \boldsymbol{x}_{i-1}$ are the followings:

$$\begin{aligned}
\boldsymbol{x}_{i+1} &= \boldsymbol{x}_i + \boldsymbol{v}_i \Delta t + \boldsymbol{a}_i \frac{\Delta t^2}{2} + \frac{d\boldsymbol{a}_i}{dt}\frac{\Delta t^3}{6} + O(\Delta t^4) \\
\boldsymbol{x}_{i-1} &= \boldsymbol{x}_i + \boldsymbol{v}_i(-\Delta t) + \boldsymbol{a}_i \frac{(-\Delta t)^2}{2} + \frac{d\boldsymbol{a}_i}{dt}\frac{(-\Delta t)^3}{6} + O(\Delta t^4) \\
&= \boldsymbol{x}_i - \boldsymbol{v}_i \Delta t + \boldsymbol{a}_i \frac{\Delta t^2}{2} - \frac{d\boldsymbol{a}_i}{dt}\frac{\Delta t^3}{6} + O(\Delta t^4)
\end{aligned} \tag{7}$$

Therefore, by summing up $\boldsymbol{x}_{i+1}$ and $\boldsymbol{x}_{i-1}$, we obtain the following equation:

$$\begin{aligned}
\boldsymbol{x}_{i+1} + \boldsymbol{x}_{i-1} &= 2\boldsymbol{x}_i + \boldsymbol{a}_i \Delta t^2 + O(\Delta t^4) \\
\boldsymbol{x}_{i+1} &= 2\boldsymbol{x}_i - \boldsymbol{x}_{i-1} + \boldsymbol{a}_i \Delta t^2 + O(\Delta t^4) \\
\boldsymbol{x}_{i+1} &= \boldsymbol{x}_i + \underbrace{\frac{\boldsymbol{x}_i - \boldsymbol{x}_{i-1}}{\Delta t}}_{\text{Velocity term}} \Delta t + \boldsymbol{a}_i \Delta t^2 + O(\Delta t^4)
\end{aligned} \tag{8}$$

[10]Each particle has 30-40 adjacents in fluid simulation, so the complexity of conjugate gradient is $O(30n^2) \simeq O(n^2)$

This equation is the basic update formula in second-order ODE schemes. In simulations, we assume that the initial positions $\boldsymbol{x}_0$ and the initial velocities $\boldsymbol{v}_0$ are given. The advantage of this method is even though we **do not necessarily need velocities** and we **only need to compute accelerations once**, we still obtain **the better accuracy** $O(\Delta t^4)$ than the explicit euler [11].

1.8.1 Verlet

Verlet uses the update shown in Eq. (8). The advantages of this method are the followings:

1. one acceleration computation per step
2. do not require velocities computation
3. the error order $O(\Delta t^4)$

On the other hand, when we need velocities in simulations, we need to compute $\boldsymbol{v}$ using finite difference equations (FDE). In this case, the error order of $\boldsymbol{v}$ is $O(\Delta t)$ and it is not accurate. We need velocities, for example, when we would like to introduce contact handling. Another issue of this method is the overhead caused by storing the previous positions.

1.8.2 Leap-frog

The update formula of this method are the followings:

$$\begin{aligned}\boldsymbol{x}_{i+1} &= \boldsymbol{x}_i + \boldsymbol{v}_{i+\frac{1}{2}}\Delta t \\ \boldsymbol{v}_{i+\frac{3}{2}} &= \boldsymbol{v}_{i+\frac{1}{2}} + \boldsymbol{a}_{i+1}\Delta t\end{aligned} \tag{9}$$

This method is the explicit scheme for both $\boldsymbol{x}$ and $\boldsymbol{v}$. However, this method uses different time points and this is motivated in terms of the stability. Furthermore, this method uses the acceleration computed using the velocities above. The major issue of this method is the computation of accelerations. As seen in Eq. (9), we do not compute $\boldsymbol{v}_i$ explicitly; therefore, if the computation of $\boldsymbol{a}$ relies on $\boldsymbol{v}$, we have to interpolate $\boldsymbol{v}_{i+1}$. If we choose the central differentiation $\boldsymbol{a}_i = \frac{\boldsymbol{v}_{i+1/2} - \boldsymbol{v}_{i-1/2}}{\Delta t}$, this method is equivalent to the Verlet method.

1.8.3 Velocity Verlet

In the Verlet method, the accuracy of velocities are remarkably lower than that of positions. To address this issue, the velocity Verlet are proposed. The update of both $\boldsymbol{x}$ and $\boldsymbol{v}$ are performed using explicit schemes as follows:

$$\begin{aligned}\boldsymbol{x}_{i+1} &= \boldsymbol{x}_i + \boldsymbol{v}_i\Delta t + \boldsymbol{a}_i\frac{\Delta t^2}{2} + O(\Delta t^3) \\ \boldsymbol{v}_{i+1} &= \boldsymbol{v}_i + (\boldsymbol{a}_{i+1} + \boldsymbol{a}_i)\frac{\Delta t}{2} + O(\Delta t^3) \\ (\because \text{RHS} &= \boldsymbol{v}_i + (\boldsymbol{a}_i + \frac{d\boldsymbol{a}_i}{dt}\Delta t + O(\Delta t^2) + \boldsymbol{a}_i)\frac{\Delta t}{2} + O(\Delta t^3) \\ &= \boldsymbol{v}_i + \boldsymbol{a}_i\Delta t + \frac{d\boldsymbol{a}_i}{dt}\frac{\Delta t^2}{2} + O(\Delta t^3))\end{aligned}$$

Note that this method assumes that the accelerations do not depend on the velocities. Although this method also requires only one acceleration computation per step. However, it achieves better accuracy than the explicit euler.

[11] Since explicit euler requires acceleration computation once, we compare to this method. The error order is $O(\Delta t)$.

1.8.4 Beeman

The Beeman method further improves the accuracy of the velocity verlet. The updates are performed using the followings:

$$\begin{aligned}\boldsymbol{x}_{i+1} &= \boldsymbol{x}_i + \boldsymbol{v}_i\Delta t + \left(\frac{2}{3}\boldsymbol{a}_i - \frac{1}{6}\boldsymbol{a}_{i-1}\right) + O(\Delta t^4)\\ \boldsymbol{v}_{i+1} &= \boldsymbol{v}_i + \left(\frac{5}{12}\boldsymbol{a}_{i+1} + \frac{2}{3}\boldsymbol{a}_i - \frac{1}{12}\boldsymbol{a}_{i-1}\right)\Delta t + O(\Delta t^4)\end{aligned}$$

The coefficients of the velocity update can be derived using the Taylor series approximation as well. This method also requires only one acceleration computation per step. However, extra storage for the previous accelerations can be overhead of the method.

1.8.5 Gear

The gear method is one of the predictor-corrector schemes. To introduce this method, we first define the notation used in the method.

Definition 2
$\forall i, k \in \mathbb{N}$, we define

$$\boldsymbol{r}_i^k = \frac{d\boldsymbol{x}_i^k}{dt}\frac{\Delta t^k}{k!}$$

Using this notation, each $\boldsymbol{x}_{i+1}, \boldsymbol{v}_{i+1}, \boldsymbol{a}_{i+1}$ can be represented as follows:

$$\begin{aligned}\boldsymbol{x}_{i+1} = \boldsymbol{r}_0^{i+1} &= \sum_{k=0}^{5} {}_k\mathrm{C}_0 \boldsymbol{r}_k^i + O(\Delta t^6)\\ \boldsymbol{v}_{i+1}\Delta t = \boldsymbol{r}_1^{i+1} &= \sum_{k=1}^{5} {}_k\mathrm{C}_1 \boldsymbol{r}_k^i + O(\Delta t^6)\\ \boldsymbol{a}_{i+1}\frac{\Delta t^2}{2} = \boldsymbol{r}_2^{i+1} &= \sum_{k=2}^{5} {}_k\mathrm{C}_2 \boldsymbol{r}_k^i + O(\Delta t^6)\end{aligned} \tag{10}$$

Note that for all $k = 0, \cdots, 5$, the following holds:

$$\boldsymbol{r}_j^{i+1} = \sum_{k=j}^{5} {}_k\mathrm{C}_j \boldsymbol{r}_k^i + O(\Delta t^6)$$

First, we predict $\boldsymbol{x}_{i+1}, \boldsymbol{v}_{i+1}, \boldsymbol{a}_{i+1}$ using Eq. (10). Additionally, we compute the acceleration $\boldsymbol{a}(\boldsymbol{r}_0^{i+1}, \frac{1}{\Delta t}\boldsymbol{r}_1^{i+1})$ using the yielded values. Then we define the error ϵ_{i+1} as follows:

$$\epsilon_{i+1} = \boldsymbol{r}_2^{i+1} - \boldsymbol{a}\big(\boldsymbol{r}_0^{i+1}, \frac{1}{\Delta t}\boldsymbol{r}_1^{i+1}\big) \tag{11}$$

In the correction scheme, we calibrate each predicted value as follows:

$$\boldsymbol{r}_k^{i+1} = \boldsymbol{r}_k^{i+1} - c_k\epsilon_{i+1} \tag{12}$$

where $(c_0, \cdots, c_5) = (\frac{3}{20}, \frac{251}{360}, 1, \frac{11}{18}, \frac{1}{6}, \frac{1}{60})$. The problem of this method is inconsistency. The inconsistency means $\boldsymbol{r}_2^{i+1} \neq \boldsymbol{a}(\boldsymbol{r}_0^{i+1}, \frac{1}{\Delta t}\boldsymbol{r}_1^{i+1})$ after the correction step. However, this method is efficient and still a popular method among the field. In summary, this method works as follows:

Table 2: Pros and cons of each scheme

Scheme	Pros	Cons
Explicit	Accurate	Unstable
	Can update with non-linear accelerations	Requires more accleration computations
	Simple implementation	for better accuracy
Implicit	Stable[12]	Less accurate
		Requires linearization of accelerations[13]
Second-order	Positions are accurate	Velocities are not accurate
	only one acceleration computation	
	not necessarily requires velocities	

1. **Initialization**: initializes $\boldsymbol{r}_0^0, \boldsymbol{r}_1^0, \boldsymbol{r}_2^0$ using given $\boldsymbol{x}_0, \boldsymbol{v}_0, \boldsymbol{a}_0$ and others as zero
2. **Prediction**: performs according to Eq. (10)
3. **Error computation**: computes using Eq. (11)
4. **Correction**: calibrates using Eq. (12)

1.9 Pros and cons of explicit, implicit, second-order ODE schemes

The advantages and the disadvantages of each scheme is summarized in Table 2 First, we have to check **the ratio of time step vs computation time**. Basically, implicit schemes are attractive in terms of the stability and thus it can take larger time steps. However, at least in particle simulations, we have to pay attention to so-called Courant number defined as follows:

$$C = \frac{\|\boldsymbol{v}\|\Delta t}{L}$$

where L is the diameter of particles. If C is larger than 1, the particle potentially bumps into each other or walls during $i\Delta t \sim (i+1)\Delta t$. Then the simulation cannot process the contact handling and particles go through each other or they just get out from the simulator.

2 Elastic solids

In the previous sections, we discuss how to update velocities and positions. From this section, we will see force models, i.e. acceleration computations. In this section, we handle the force model of elastic solids and the computation is performed by minimizing the deformation of an object. Note that the courant number $C = \frac{\|\boldsymbol{v}\|\Delta t}{\Delta x}$ is important for the elastic solids as well and it is modeled to be closed to $C \approx 1$.

2.1 Deformation and stress

Elastic solids are modeled using a set of particles and forces at each particle accounts for the resitance to deformation. More formally, the elastic solids are divided into finite elements and the definition of an element is as follows:

[13]Larger time steps do not negatively affect the performance of implicit schemes
[13]We can set arbitrary iteration for linear system solver, so the computation time can be dynamic.

Definition 3
Given a set of particles $X_i = \{\boldsymbol{x}_{i,1}, \cdots, \boldsymbol{x}_{i,n}\}$, the i-th element is the object which has X_i as the vertices

For example, if $n = 2$, the element is a line. If $n = 3$, it is a triangle. If $n = 4$, it is a tetrahedron.
The deformation of the i-th element $C_i(\boldsymbol{x}_{i,1}, \cdots, \boldsymbol{x}_{i,n}) \in \mathbb{R}$ is measured using the deviation from the initial element size V_i. For example, the following equations are the deformation [14] for $n = 2, 3$:

$$C_i(\boldsymbol{x}_1, \boldsymbol{x}_2) = \frac{1}{V_i}(|\boldsymbol{x}_1 - \boldsymbol{x}_2| - V_i)$$
$$C_i(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3) = \frac{1}{V_i}\left(\frac{1}{2}|(\boldsymbol{x}_2 - \boldsymbol{x}_1) \times (\boldsymbol{x}_3 - \boldsymbol{x}_1)| - V_i\right)$$

where V_i for the first equation is the initial distance between $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ and V_i for the second equation is the initial space of the triangle composed of $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3$. Note that V_i is a constant number and C is conventionally divided by V_i to be a non-dimensional number. Plus, since each element basically tries to stay as close to the initial state as possible, the following statement holds:

1. If $C_i = 0$, the object is not deformed, so no force happens
2. If $C_i > 0$, the object is larger than the initial state, so each particle tries to shrink inside
3. If $C_i < 0$, the object is smaller than the initial state, so each particle tries to expand

Using the deformation C_i, the stress (or internal pressure) of the i-th element $S_i(\boldsymbol{x}_{i,1}, \cdots, \boldsymbol{x}_{i,n}) \in \mathbb{R}$ is formulated as follows:

$$S_i(\boldsymbol{x}_{i,1}, \cdots, \boldsymbol{x}_{i,n}) = k_i C_i(\boldsymbol{x}_{i,1}, \cdots, \boldsymbol{x}_{i,n})$$

where k_i is the constant number representing the material stiffness. A larger k_i means more stiff and resists the deformation more. In contrast, a smaller k_i means softer and do not resists the deformation so much.

2.2 Elastic energy

Each element always resists to the deformation and this deformation can be quantified using the elastic energy defined as follows:

$$\begin{aligned} E_i(\boldsymbol{x}_{i,1}, \cdots, \boldsymbol{x}_{i,n}) &= \frac{1}{2}C_i S_i V_i \\ &= \frac{1}{2}k_i C_i^2 V_i \end{aligned}$$

Additionally, since this energy is always non-negative and the minimization of the energy leads to the stability of each element [15], this optimization can be performed using gradient descent. Therefore, the force model of elastic solids is the following:

$$\boldsymbol{F}_{i,j} = -\frac{\partial E}{\partial \boldsymbol{x}_{i,j}} = -k_i V_i C_i \frac{\partial C_i}{\partial \boldsymbol{x}_{i,j}}$$

This force $\boldsymbol{F}_{i,j}$ for the j-th particle of the i-th element is called **elastic forces**. This force holds the following properties:

1. The preservation of linear momentum: $\sum_{j=1}^{n} \boldsymbol{F}_{i,j} = \boldsymbol{0}$
2. The preservation of angular momentum: $\sum_{j=1}^{n} \boldsymbol{x}_j \times \frac{d\boldsymbol{F}_{i,j}}{dt} = \boldsymbol{0}$

In other words, elastic forces influence only the deformation, but not rotation or acceleration of the element or the entire object. That is why the elastic force is called **internal forces** [16].

[14] Strictly speaking, this formulation is not correct. However, for the simplicity, this formulation is used in practice.
[15] The optimal state for each element is to be non-deformed $E_i = 0$.
[16] One example of external forces is the gravitational force.

2.3 Damping force

Damping force is caused by frictions or viscosity. Since this force slows down the velocities or reduces the noise, it improves the stability of simulations. There are two ways to represent this force:

1. **Particle forces** $\boldsymbol{F}_i^{\text{damp}} = -\gamma \boldsymbol{v}_i$: This is an external force, so it can slow down the whole movement and stabilize the simulation, but leads to less accuracy
2. **Relative damping** $\boldsymbol{F}_{i,j}^{\text{damp}} = -\gamma\big((\boldsymbol{v}_i - \boldsymbol{v}_j)\cdot \boldsymbol{r}_{i,j}\big)\boldsymbol{r}_{i,j}$: This is an internal force, so it preserves linear and angular momentum and also reduces oscillations and noise

where $\boldsymbol{r}_{i,j} = \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|}$

2.4 Particle masses

Each element has several particles and each particle should have some mass. Additionally, it is important to reduce the influence on the result from the discretization. For this reason, we let each particle have some masses. In the case where all the elements are composed of three particles, since each element has three particles, each particle in an element has the $\frac{1}{3}$ of mass contribution from each element. Therefore, the mass for the i-th particle is defined as follows:

$$m_i = \sum_j \frac{\rho_j S_j}{3}$$

where ρ_j, S_j is the density and the space of the j-th element (or object) respectively. For the simplicity, we often use the following equation as well:

$$m_i = \rho_i \sum_j \frac{S_j}{3} \tag{13}$$

where ρ_i is an initial-user-defined parameter [17]. Using this mass m_i, the elastic acceleration for each particle can be computed as $\boldsymbol{a}_i = \boldsymbol{F}_{i,j}/m_i$. Note that due to the simple form, Eq. (13) is used more often.

2.5 Collision handling: particle vs plane

Let $\boldsymbol{x}_{\text{ref}}^1, \boldsymbol{x}_{\text{ref}}^2, \boldsymbol{x}_i \in \mathbb{R}^3$ be reference positions on the plane and the position for a particle, $\boldsymbol{n}$ be the unit normal vector with respect to $\boldsymbol{x}_{\text{ref}}^2 - \boldsymbol{x}_{\text{ref}}^1$, i.e. $\boldsymbol{n}\cdot(\boldsymbol{x}_{\text{ref}}^2 - \boldsymbol{x}_{\text{ref}}^1) = 0$, $\boldsymbol{v}_i$ be the velocity of the i-th particle, $\boldsymbol{V}_i$ be the velocity of the i-th particle after the collision, x_1, x_2, x_3 be the axes of the global coordinate system, x'_1, x'_2, x'_3 be the axes of the local coordinate system, $\boldsymbol{P}_i$ be the momentum of the i-th particle. Note that the axis x'_3 is horizontal to $\boldsymbol{n}$ and all the axes are vertical to each other.

2.5.1 The update of velocities

Since both $\boldsymbol{x}_{\text{ref}}^1, \boldsymbol{x}_{\text{ref}}^2$ are on the plane, if the angle between $\boldsymbol{x}_i - \boldsymbol{x}_{\text{ref}}^1$ and $\boldsymbol{n}$ is larger than or equal to 90 degree, it means the i-th paticle collides with the plane as shown in Figure 1. Therefore, it can be trivially detected by the sign of the inner product $\text{sign}(\boldsymbol{n}\cdot(\boldsymbol{x}_i - \boldsymbol{x}_{\text{ref}}^1))$ [18]. Additionally, when we consider the plane as a particle $\boldsymbol{x}_j$, the governing equation are the followings:

$$\begin{aligned}
\textbf{Momentum Preservation} &: m_i V_{i,x'_k} - m_i v_{i,x'_k} = P_{x'_k} = -m_j V_{j,x'_k} + m_j v_{j,x'_k} \text{ (for all } k = 1,2,3) \\
\textbf{Coefficient of Restitution} &: V_{i,x'_1} - V_{j,x'_1} = -e(v_{i,x'_1} - v_{j,x'_1}) \\
\textbf{Friction} &: P_{x'_k} = \mu P_{x'_3} \text{ (for } k = 1,2)
\end{aligned}$$

[17] It sounds strange to let each particle have the density, but conventionally we set the density for each particle, but not element.

[18] If it is positive, no collision, else collision.

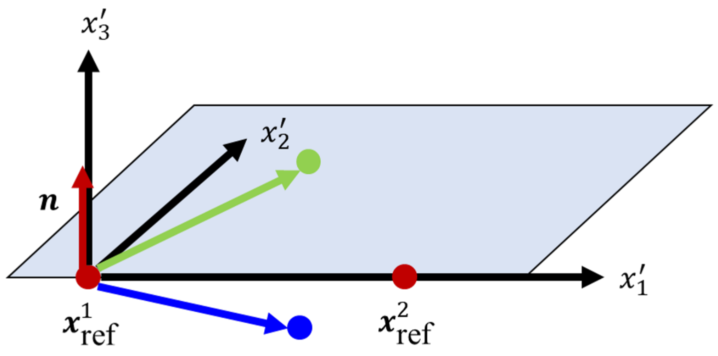
x'_3
x'_2
$\boldsymbol{n}$
x'_1
$\boldsymbol{x}^1_{\text{ref}}$
$\boldsymbol{x}^2_{\text{ref}}$

Figure 1: The visualization of the collision detection with a plane. Since the inner product between the blue line and red line (i.e. the normal) is negative, we can know the blue point and the plane collide.

where $e(0 < e \leq 1)$ is an elastic coefficient and $\mu(0 \leq \mu \leq 1)$ is a friction coefficient. This is 9×9 linear system to obtain three momentums and six velocities. Note that when we consider the collision to fixed planes, we can replace velocities with zero and mass with infinity. Such equations can be solved explicitly and we use the closed form to obtain the new velocities. However, since the obtained velocities do not guarantee the decay over time, this simple heuristic $V_{\cdot,x'_k} = \mu v_{\cdot,x'_k}$ is used for the update of $V_{\cdot,x'_k}$ for $k = 1, 2$ [19]. This update guarantees the decay $\|V_{\cdot,x'_k}\| \leq \|v_{\cdot,x'_k}\|$ [20].

2.5.2 The update of positions

When elements collide with planes, the positions are typically outside of the simulator. Therefore, we need to consider this phenomenon when updating the position. There are basically two solutions for this:

1. Do nothing
2. Project particles onto the plane by $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i^{\text{pred}} - (\boldsymbol{n} \cdot (\boldsymbol{x}_i - \boldsymbol{x}_{\text{ref}}))\boldsymbol{n}$

Note that it happens when $\boldsymbol{n} \cdot (\boldsymbol{x}_i - \boldsymbol{x}_{\text{ref}}) \leq 0$. This update works well for inelastic cases, but it does not for elastic cases.

2.6 Visualization

When we visualize the simulated result, we take finer grids for the visualization and rough grids for the simulation. The visualization scheme is as follows:

1. **preprocessing**: determine the closest element for each surface point and compute Barycentric coordinates with respect to the corresponding element
2. **simulation step**: compute surface-point positions using the pre-computed coefficients of Barycentric coordinates

The Barycentric coordinate for a tetrahedron is defined as follows:

$$\boldsymbol{x}_s = \sum_{k=1}^{4} \alpha_k \boldsymbol{x}_k$$

[19]In other words, the vertical direction to $\boldsymbol{r}_{i,j}$
[20]It is more important to satisfy this constraints.

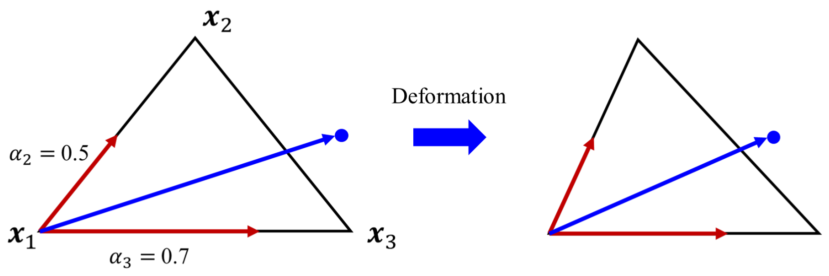

Figure 2: The weights for each direction are preserved and the positions of new elements can be easily estimated using the weighted sum, i.e. barycentric coordinate.

where $\sum_{k=1}^{4} \alpha_k = 1$ and α_k (for all $k = 1, 2, 3, 4$) is called Barycentric coordinates of $\boldsymbol{x}_s$ with respect to $\boldsymbol{x}_k$ (for all $k = 1, 2, 3, 4$). This linear system can be solved unless the closest element is degenerated [21] and once they are computed, they are just fixed throughout the visualization process as shown in Figure 2.

3 Particle fluids

3.1 Smoothed Particle Hydrodynamics (SPH)

SPH is an interpolation or discretization scheme that can be used for the computation of fluid solvers and this method itself is not a fluid simulation method. This method allows to interpolate quantities at arbitrary positions and approximate the spatial derivatives using a finite number of samples, i.e. adjacent particles. In the fluid simulation, we use SPH to approximate density, pressure, viscosity and external forces. The approximation is performed using the following equation:

$$\begin{aligned} Q_i &= \sum_j V_j Q_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sum_j \frac{m_j}{\rho_j} Q_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) \\ \nabla Q_i &= \sum_j \frac{m_j}{\rho_j} Q_j \nabla k(\boldsymbol{x}_i, \boldsymbol{x}_j) \\ \nabla^2 Q_i &= \sum_j \frac{m_j}{\rho_j} Q_j \nabla^2 k(\boldsymbol{x}_i, \boldsymbol{x}_j) \end{aligned} \tag{14}$$

where Q is the quantity, which we would like to approximate and V_j, ρ_j, m_j are volume, density and mass respectively and $k(\cdot, \cdot)$ is the kernel function. The typical kernel function is cubic spline defined as follows:

$$K_{i,j} = K\left(\frac{\|\boldsymbol{x}_i - \boldsymbol{x}_j\|}{R}\right) = K(r) = \alpha \begin{cases} (2-r)^3 - 4(1-r)^3 & (0 \leq r < 1) \\ (2-r)^3 & (1 \leq r < 2) \\ 0 & (2 \leq r) \end{cases}$$

where α is a normalization constant. Note that $K_{i,j} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) \in \mathbb{R}, \nabla K_{i,j} \in \mathbb{R}^3$. In this case, since the upper bound of r is 2, $2R$ is called **support** and when the support is larger, it leads to the larger number of neighbors and it affects the performance significantly. The properties of the kernel function are the followings:

1. **Normalized**: $\int_{\boldsymbol{x}' \in \mathcal{X}} k(\boldsymbol{x}, \boldsymbol{x}') d\boldsymbol{x}' = 1$

[21] If it is degenerated, we just take the second closest element.

2. **Compact support**: $\forall \boldsymbol{x}', k(\boldsymbol{x}, \boldsymbol{x}') = 0$, s.t. $\|\boldsymbol{x}' - \boldsymbol{x}\| > 2R$
3. **Positive semi-definite**: $\forall n \in \mathbb{N}_{\geq 1}, \forall \boldsymbol{a} \in \mathbb{R}^n, \sum_{i=1}^n \sum_{j=1}^n, a_i a_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq 0$
4. **Limit is dirac delta**: $\lim_{R \to +0} k(\boldsymbol{x}, \boldsymbol{x}') = \delta(\boldsymbol{x}, \boldsymbol{x}')$
5. **Twice differentiable**: $k(\boldsymbol{x}, \boldsymbol{x}')$ belongs to $\mathbb{C}^2$ class

Using this kernel function, the approximation is computed as follows:

$$
\begin{aligned}
Q(\boldsymbol{x}) &= (Q * k)(\boldsymbol{x}) = \int_{\boldsymbol{x}' \in \mathcal{X}} Q(\boldsymbol{x}') k(\boldsymbol{x}, \boldsymbol{x}') d\boldsymbol{x}' \\
\nabla Q(\boldsymbol{x}) &= \nabla (Q * k)(\boldsymbol{x}) = (\nabla Q * k)(\boldsymbol{x}) = (Q * \nabla k)(\boldsymbol{x}) \\
\mathit{\Delta} Q(\boldsymbol{x}) &= \mathit{\Delta}(Q * k)(\boldsymbol{x})(\mathit{\Delta} Q * k)(\boldsymbol{x}) = (Q * \mathit{\Delta} k)(\boldsymbol{x}) = (\nabla Q * \nabla k)(\boldsymbol{x})
\end{aligned}
$$

Note that the formulations can be proved using Convolution theorem $\mathcal{F}(f * g) = \mathcal{F}(f)\mathcal{F}(g)$ where $\mathcal{F}$ is the fourier transform and we use cross-correlation in the application as in convolutional neural networks.

3.2 Governing equations in particle approaches

There are mainly two types of approaches in fluid simulations. One is particle approaches or Lagrangian fluid simulation. The other is grid approaches or Eulerian fluid simulation. The Eulerian approaches first have the static cells over the simulating space and each cell does not move, but just store the quantities. On the other hand, the Lagrangian approaches let particles move around the space and each particle has its own quantities. This paper focuses on the Lagrangian approaches. The governing equations for the Lagrangian approaches are the followings:

$$
\frac{d\boldsymbol{v}}{dt} = \boldsymbol{g} + \nu \nabla^2 \boldsymbol{v} - \nabla \frac{p}{\rho}, \ \frac{d\boldsymbol{x}}{dt} = \boldsymbol{v}
$$

where the first equation is called the Navier-Stokes equation and ν is a viscosity coefficient and $\boldsymbol{g}$ is a gravitational acceleration unit. Note that $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$ is gradient operation and $\nabla \cdot \nabla = \mathit{\Delta} = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$ is Laplacian. The Navier-Stokes equation has the following three terms:

1. **Pressure acceleration**: The resistance to the changes of density and volume.
2. **Viscosity**: The resistance to deformation and the force toward the average velocity of the neighbors.
3. **External forces**: In simple cases, it is just a gravity.

We will discuss each term one by one.

3.2.1 Density

The explicit form for density is the following:

$$
\rho_i = \sum_j \frac{m_j}{\rho_j} \rho_j K_{i,j} = \sum_j m_j K_{i,j}
$$

The approximation of the quality is sufferred when there are only few samples around the position of interest. Since the following differential update gives zero when the densities around the particle is constant, the following equation is often used:

$$
\frac{d\rho_i}{dt} = -\rho_i \nabla \cdot \boldsymbol{v}_i = -\sum_j m_j (\boldsymbol{v}_i - \boldsymbol{v}_j) \cdot \nabla K_{i,j}
$$

Note that since this formulation is the divergence of the velocities, the formulation is different from the summation over the gradient and the left hand side is the derivative with respect to time, not positions.

3.2.2 Pressure acceleration

For the computation of pressure acceleration, the following equation is often used since it preserves the linear and angular momentum:

$$\begin{aligned}\boldsymbol{a}_i^{\text{pressure}} &\simeq -\nabla\frac{p_i}{\rho_i} = -\frac{1}{\rho_i}\nabla p_i - p_i\nabla\frac{1}{\rho_i} \quad \left(p_i = \max\left(k\left(\frac{\rho_i}{\rho_0} - 1\right), 0\right)\right) \\ &= -\sum_j m_j\left(\frac{p_i + p_j}{\rho_i\rho_j}\right)\nabla K_{i,j} \\ &\simeq -\sum_j m_j\left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right)\nabla K_{i,j}\end{aligned}$$

where ρ_0 is the initial global density and k is a user-defined stiffness. Intuitively, this term accelerates particles from high to low pressure to minimize density deviation $\frac{\rho_i}{\rho_0} - 1$. Note that the pressure has to be non-negative and we apply clipping to the pressure state equation. This clipping does not affect significantly, because when particles are further away, the pressure acceleration is not preferred.

3.2.3 Viscosity

Viscosity is represented by the divergence of the stress and the stress is proportional to velocities for fluids. Therefore, each particle will have the acceleration towards the average velocity of neighbors. For the viscosity computation, the following equation is often used since it can avoid the second derivative and thus it is more robust:

$$\nu\nabla^2\boldsymbol{v}_i \simeq 2\nu\sum_j\frac{m_j}{\rho_j}\frac{(\boldsymbol{v}_i - \boldsymbol{v}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)}{\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2 + 0.01R^2}\nabla K_{i,j} \tag{15}$$

This formulation is approximated by the Taylor series approximation as follows:

$$\begin{aligned}v(x') &\simeq v(x) + \frac{dv}{dx}(x' - x) + \frac{d^2v}{dx^2}\frac{(x' - x)^2}{2} \\ \frac{v(x') - v(x)}{x' - x} &\simeq \frac{dv}{dx} + \frac{d^2v}{dx^2}\frac{x' - x}{2}\end{aligned}$$

For the simplicity, we will see the derivation of the 1D version. By using the equation and the first order derivative in Eq. (14), we obtain the following:

$$\begin{aligned}\int\frac{(v(x') - v(x))}{x' - x}\frac{\partial k(x, x')}{\partial x}dx' &\simeq \int_{x-2R}^{x+2R}\left(\frac{dv}{dx} + \frac{d^2v}{dx^2}\frac{x' - x}{2}\right)\frac{\partial k(x, x')}{\partial x}d\boldsymbol{x}' \\ &= \underbrace{\left[-\left(\frac{dv}{dx} + \frac{d^2v}{dx^2}\frac{x' - x}{2}\right)k(x, x')\right]}_{=0\ (\because k(x, x+2R) = 0)} \\ &= \frac{1}{2}\frac{d^2v}{dx^2}\int_{x-2R}^{x+2R}k(x, x')dx' = \frac{1}{2}\frac{d^2v}{dx^2}\end{aligned}$$

where $\frac{\partial k(x,x')}{\partial x}dx' = -k(x, x')$. Using this result, we define the coefficient as 2 in Eq. (15) [22].

[22] In the multidimensional cases, this coefficient seems to be not correct.

3.3 Boundary handling

By introducing the static fluid samples, we can represent the boundary as follows:

$$Q_i = \sum_j \underbrace{\frac{m_j}{\rho_j} Q_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)}_{\text{Fluid samples}} + \sum_j \underbrace{\frac{m_j^b}{\rho_j^b} Q_j^b k(\boldsymbol{x}_i, \boldsymbol{x}_j^b)}_{\text{Boundary samples}}$$

Basically, fluid samples at boundary has high density and high pressure, so the pressure solves the contact without explicit handling as in rigid objects.

3.4 Implementation

1. **pre-setting**: first defines kernel, particle mass or spacing ($m_i = \rho_0 R^3$) [23], and integration scheme.
2. **Find neighbors of all particles**: The neighbors are defined based on the kernel.
3. **Compute density, pressure and non-pressure accelerations**: Compute by the equations discussed above.
4. **Update velocities and positions**: This computation relies on the integration scheme.

3.5 Sampling of particles

In the case of ideally sampled particles, the followings hold:

$$\begin{aligned}
\sum_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) &= \frac{\rho_i}{m_i} \\
\nabla k(\boldsymbol{x}_i, \boldsymbol{x}_j) &= -\nabla k(\boldsymbol{x}_j, \boldsymbol{x}_i) \\
\sum_j \nabla k(\boldsymbol{x}_i, \boldsymbol{x}_j) &= 0 \\
\sum_j (\boldsymbol{x}_i - \boldsymbol{x}_j) \times \nabla k(\boldsymbol{x}_i, \boldsymbol{x}_j) &= -\frac{1}{V_i} I
\end{aligned}$$

Note that ideally sampled means each particle is uniformly sampled across the simulated space, i.e. $V_i = V_j, \rho_i = \rho_j$ for arbitrary pairs (i, j). Since the properties above hold in the case of ideally sampled particles, we often use it to validate newly created simulations. On the other hand, in the case of erroneous sampling, i.e. particles are sampled densely or sparsely, SPH often contradicts with real-world phenomenona.

4 Neighbor search

Since SPH requires the summation of neighbors and neighbors dynamically change, it is important to be able to compute each neighbor efficiently. This is realized by space subdivision and it is required to reduce the memory consumption and the computational complexity.

4.1 Space subdivision

Spatial data structures are mainly required **fast query**, **fast construction** at each simulation step and **sparsity** over the simulation space. Since the computational complexity of the construction and the access are $O(C)$ and $O(1)$ where C is the number of grids, the space is usually subdivided into uniform grids.

[23] The ratio of Δx and R governs the number of neighbors.

Algorithm 1 Index sort

```
function INDEX SORT
    Initialize counter, location
    for $i = 0, \dots, n-1$ do                                  ▷ Iteration over each particle
        $j$ := the index of cell that the $i$-th particle belongs to
        counter[$j$]++
    for $i = 0, \dots, C-2$ do                                  ▷ Iteration over each cell
        counter[$i+1$] = counter[$i$] + counter[$i+1$]
    for $i = 0, 1, \dots, n-1$ do
        $j$ := the index of cell that the $i$-th particle belongs to
        counter[$j$]−−, location[counter[$j$]] = $i$
    Insert 0 in the head of counter
    return location, counter
```

4.1.1 neighbor search using uniform grids

In the case of D-dimensional space, each cell refers to 3^D cells around each cell. Therefore, smaller cell size can reduce the computation of each cell, but it leads to the larger number of cells to store. On the other hand, the larger cell size can reduce the memory size, but it leads to more computation for each cell. Note that the cell size is typically equal to kernel support. One variant of neighbor search is **verlet lists** and this method **updates the neighbor candidates once every certain step** based on the assumption that each particle does not move further than its size in one step. The neighbors are chosen from the candidate pool at each step. The major issue of this method is the memory consumption due to the storage of the candidate pool.

4.2 Sorting algorithm

In the space subdivision, we would like to have the data structure to know which cell each particle belongs to and which particles each cell has.

4.2.1 Index sort

The most naïve algorithm is index sorting. The algorithm is given in Algorithm 1 and location[counter[i]] $\sim$ location[counter[$i+1$]-1] are the indices of particles in the i-th cell. The iteration over each particle can be computed in parallel and since we do not have to allocate memory explicitly as in hashing, we can reduce memory allocation. The major problem of this method is lower cache-hit ratio [24]. This is because while each particle in the same cell has closer memory address to each other, but not necessarily to all the neighbors. In order to increase the cache-hit ratio, it is important to have spatial locality, which is the state where most neighbors locate in close memory.

4.2.2 Z-curve index

Z-curve index improves the cache-hit ratio by sorting cell indeices in a way that each neighbor cell locates closer. These indices can also be computed efficiently. Additionally, since each neighbor cell is closer in terms of memory location and the temporal coherence holds, only small portion of particles has to be sorted in each time step.

[24] During computation, computer stores elements recently used in cache and cache can be reused quickly. In this sense, the more we use cached data, the more we can reduce computational time.

4.3 Hashing

4.3.1 Spatial hashing

Index sort requires the memory allocation for each grid cell. However, if the simulation space is huge, this is not infeasible. Therefore, we use hashing and represent simulation space sparsely. Hash function h maps the cell index c to a number and we store hash for all the cells including at least one particle in a list. Hashing allows us to deal with infinite domains. Major problems of hashing are **hash collisions** and **reduced cache-hit ratio**. Although hash collision does not have a bad influence on the result, hash collision slows down the searching procedure. Additionally, the cache-hit ratio decreases, because the hash table is sparsely filled.

4.3.2 Compact hashing

This method uses **larger hash table** compared to spatial hashing to reduce hash collisions and particles are sorted with respect to z-curve every certain step. Therefore, when traversing the list, the cache-hit ratio improves. Additionally, if there is no hash collision in a cell, we need to look at only the adjacent cells for the particles in such a cell. For this reason, **hash-collision flag** speeds up the computation. Other than that, since it **updates**, but not rebuilds, **list of used cells** [25], it can employ temporal coherence. Note that if particles change their cells, the cell indices will be estimated.

On the other hand, since each neighbor in the list of used cells are not necessarily neighbors for each cell, spatial locality is not improved.

5 Rigid bodies

5.1 Rigid bodies vs elastic solid

In the previous section, we handle the elastic solids, which is deformable. In this section, we handle the rigid bodies, which is undeformable. While both elastic solid and rigid bodies are composed of multiple particles, each particle in rigid bodies is tied by non-stretchable strings and that in elastic solids is tied by springs. Therefore, the state variables of rigid bodies are position, velocity, orientation, angular momentum and mass distribution of the object. Note that elastic solids do not have the notation of orientation that is the rotation matrix mapping from the local coordinate to the global coordinate.

5.2 Particle representation

Each rigid body is represented by the following variables:

1. **Reference position** $\bar{\boldsymbol{x}}_g = [0, 0, 0]^\top$: Center of mass of the body
2. **Relative position of each particle** $\bar{\boldsymbol{x}}_i$: Position of the i-th particle in the body
3. **Absolute position of each particle** $\boldsymbol{x}_i = \boldsymbol{x}_g + R\bar{\boldsymbol{x}}_i$: Global position of the i-th particle and it is represented by the translation and the rotation of the relative vector

where $\boldsymbol{x}_g = \frac{\sum_i m_i \boldsymbol{x}_i}{M}$ is the gravitational point, M is the total mass of the body and R is the rotation matrix or orientation of the object. Note that $\bar{\boldsymbol{x}}$ refers to the relative vector from the center of mass throughout this section as shown in Figure 3.

[25] Spatial hashing does not use a list of used cells, but just uses mapping.

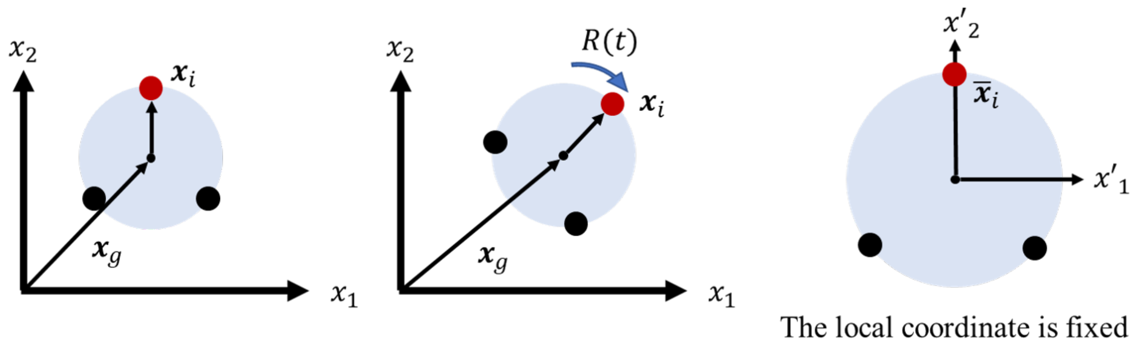

Figure 3: The visualization of the rigid body. Each position of the particles are represented using the position of the gravitational point $\boldsymbol{x}_g$ and the fixed relative position $\bar{\boldsymbol{x}}_i$ and the orientation R.

5.3 The quantities of rigid bodies

5.3.1 The linear acceleration

The linear acceleration of the center of mass can be computed from the mass of the rigid body and from the sum of all forces acting at arbitrary rigid body positions as follows:

$$\boldsymbol{F} = \sum_i m_i \frac{d^2\boldsymbol{x}_i}{dt^2} = \frac{d^2 \sum_i m_i \boldsymbol{x}_i}{dt^2} = M\frac{d^2\boldsymbol{x}_g}{dt^2}, \ \boldsymbol{a} = \frac{\boldsymbol{F}}{M} \tag{16}$$

From the equation, the force of the object is deduced to the force at the centeral point.

5.3.2 The orientation and the angular velocity

The orientation of a body can be computed using the rotation matrix R satisfying $R^\top = R^{-1}, \det R = 1$ as follows:

$$\begin{aligned} \boldsymbol{x}_i(t) &= \boldsymbol{x}_g(t) + R(t)\bar{\boldsymbol{x}}_i \\ \frac{d\boldsymbol{x}_i(t)}{dt} &= \frac{d\boldsymbol{x}_g(t)}{dt} + \frac{dR(t)}{dt}\bar{\boldsymbol{x}}_i + R(t)\frac{d\bar{\boldsymbol{x}}_i}{dt} \\ &= \underbrace{\frac{d\boldsymbol{x}_g(t)}{dt}}_{\text{velocity}} + \underbrace{\frac{dR(t)}{dt}\bar{\boldsymbol{x}}_i}_{\text{angular velocity}} \end{aligned} \tag{17}$$

The angular velocity of a body shows the direction of the axis of rotation and often refers to $\boldsymbol{\omega}$. Therefore, the displacement of a given position $\boldsymbol{x}$ is the following:

$$\begin{aligned} \frac{d\boldsymbol{x}}{dt} &= [\omega_2 x_3 - \omega_3 x_2 \ \ \omega_3 x_1 - \omega_1 x_3 \ \ \omega_1 x_2 - \omega_2 x_1]^\top \text{ (cross product)} \\ &= \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \boldsymbol{x} \\ &= \tilde{\boldsymbol{\omega}}\boldsymbol{x} \end{aligned}$$

where $\tilde{\cdot}$ is the cross product. Between the angular velocity and rotation, the following holds:

$$\frac{dR(t)}{dt}R(t)^\top = \tilde{\boldsymbol{\omega}} \implies \frac{dR(t)}{dt} = \tilde{\boldsymbol{\omega}}R(t)$$
$$\left(\because \frac{dRR^\top}{dt} = \boldsymbol{0} = \frac{dR}{dt}R^\top + R\left(\frac{dR}{dt}\right)^\top = \frac{dR}{dt}R^\top + \left(\frac{dR}{dt}R^\top\right)^\top, \frac{dR}{dt}R^\top = \begin{bmatrix} 0 & a & b \\ -a & 0 & c \\ -b & -c & 0 \end{bmatrix} = \tilde{\boldsymbol{\omega}}\right) \tag{18}$$

Therefore, the velocity of each particle can be reformulated as follows:

$$\begin{aligned}\frac{d\boldsymbol{x}_i(t)}{dt} &= \frac{d\boldsymbol{x}_g(t)}{dt} + \frac{dR(t)}{dt}\bar{\boldsymbol{x}}_i \\ &= \frac{d\boldsymbol{x}_g(t)}{dt} + \tilde{\boldsymbol{\omega}}(t)R(t)\bar{\boldsymbol{x}}_i \\ &= \frac{d\boldsymbol{x}_g(t)}{dt} + \tilde{\boldsymbol{\omega}}(t)(\boldsymbol{x}_i(t) - \boldsymbol{x}_g(t))\end{aligned}$$

5.3.3 The angular momentum

The angular momentum is defined as the product of mass distribution and the angular velocity as follows:

$$\begin{aligned}\boldsymbol{L}_i &= \boldsymbol{r}_i \times m_i\boldsymbol{v}_i = \boldsymbol{r}_i \times m_i(\boldsymbol{\omega} \times \boldsymbol{r}_i) \\ \boldsymbol{L} &= \sum_i \boldsymbol{L}_i = \sum_i \boldsymbol{r}_i \times m_i(\boldsymbol{\omega} \times \boldsymbol{r}_i) \\ &= \underbrace{-\sum_i m_i\tilde{\boldsymbol{r}}_i\tilde{\boldsymbol{r}}_i}_{\text{Inertia Tensor: } \boldsymbol{I}}\ \boldsymbol{\omega} \\ &= \boldsymbol{I}\boldsymbol{\omega}\end{aligned} \tag{19}$$

where $\boldsymbol{r}_i = \boldsymbol{x}_i - \boldsymbol{x}_g$. Inertia tensor represents how hard to rotate the object. Since $\boldsymbol{r}_i(t) = R(t)\bar{\boldsymbol{x}}_i$, when we define $\tilde{\boldsymbol{I}}$ as the cross product tensor of $\bar{\boldsymbol{x}}_i$, $\boldsymbol{I} = R(t)\tilde{\boldsymbol{I}}R(t)^\top$ and it can be precomputed.

5.3.4 The torque

The torque of the body is defined as the time derivative of the angular momentum and the product of the force and the length from the central point:

$$\boldsymbol{\tau} = \frac{d\boldsymbol{L}}{dt} = \sum_i \boldsymbol{r}_i \times \boldsymbol{F}_i \tag{20}$$

5.4 Simulation step

First, we compute fixed quantities and then we update the followings at each update:

1. **Force, Torque (per particle)**: Use Eqs. (16), (20)
   $\boldsymbol{\tau} = \sum_i \boldsymbol{r}_i \times \boldsymbol{F}_i, \boldsymbol{F} = \sum_i \boldsymbol{F}_i$

2. **Position, Linear velocity Rotation matrix, Angular momentum, Inertia tensor, Angular velocity (per body)**: Use Eqs. (18), (19), (20)
   $\boldsymbol{x}_g = \boldsymbol{x}_g + \boldsymbol{v}_g\Delta t, \boldsymbol{v}_g = \boldsymbol{v}_g + \frac{\boldsymbol{F}}{M}, R = R + \tilde{\boldsymbol{\omega}}R\Delta t, \boldsymbol{L} = \boldsymbol{L} + \boldsymbol{\tau}\Delta t, \boldsymbol{I} = R(t)\tilde{\boldsymbol{I}}R(t)^\top, \boldsymbol{\omega} = \boldsymbol{I}^{-1}\boldsymbol{L}$

3. **Position, Velocity (per particle)**: Use Eq. (17)
   $\boldsymbol{r}_i = \boldsymbol{x}_i - \boldsymbol{x}_g = R\bar{\boldsymbol{x}}_i, \boldsymbol{v}_i = (\boldsymbol{v}_g + \frac{\boldsymbol{F}}{M}\Delta t) + \tilde{\boldsymbol{\omega}}\boldsymbol{r}_i$

where all the updates use explicit Euler in this case, but we can use another FDE method. Note that since the error accumulates during simulations and $R(t)$ is likely to **not maintain the orthonormality**, orientation $R(t)$ has to be reorthonormalized and it is performed by **Gram-Schmidt orthonormalization**.

Layer 1	Object (Element 1 ~ 8)							
Layer 2	Element 1 ~ 4				Element 5 ~ 8			
Layer 3	Element 1, 2		Element 3, 4		Element 5, 6		Element 7, 8	
Layer 4	Element 1	Element 2	Element 3	Element 4	Element 5	Element 6	Element 7	Element 8

Figure 4: The conceptual visualization of the bounding volume hierarchies. The collision detection is performed from the first layer and if the shallower layer detects collisions, detections will be performed on the next layer. When we detect collisions in any leaf, it means the object intersects with another object. By starting from the rough detection, we can compute efficiently.

6 Bounding volume hierarchies

In simulations, collision detections are one of the most important topics and the bounding volume hierarchies are often used. Since collision detections can be an expensive process, **efficient intersection test and memory efficiency** are required. In addition, quicker generation update and the tight-fitting of the bounding box are also essential.

6.1 Basic concept

The bounding box encapsulates each element or object. Since the bounding box is the outer shell of elements or objects, we do not have to apply contact handling unless the bounding boxes overlap. Such overlaps can be roughly estimated using rough bounding boxes, so we first use large bounding boxes. If we detect intersects between objects, we make the bounding boxes smaller and compute collision detections further. Such procedure can realize **the efficient overlap test and the tight approximation** of the object. As shown in Figure 4, since the first layer can detect only rough collisions, we propagate through layers and if we detect collision in the final layer, i.e. the minimum possible elements or primitives, we process the contact handling. To efficiently perform the collision detection, the design of bounding volume hierarchies requires **balanced tree**, **minimal redundancy** (to have as small the number of layers that have primitives as possible) and **tight-fitting bounding volumes**.

6.2 Bounding volumes

The basic bounding volumes are the followings:

1. **sphere**: Each element has the center position and radius. The construction can be performed from AABB and the diagonal is the diameter. We can detect collision by the distance between two centers.

2. **axis-aligned bounding box (AABB)**: Each element has the center position and the distances from each surface (3 for 3D). We can detect collision by the distance between two centers along

Table 3: The properties of each bounding volume. D is the dimension of the simulation space and T is translation and R is rotation.

	Sphere	AABB	k-DOP	OBB
Variables	2	$D+1$	k	$k+1$
Invariance	T & R	T	T	T & R
Collision detection	1	D	k	Many[26]
Construction	-	-	A fixed set of normals	Max eigenvalue's direction

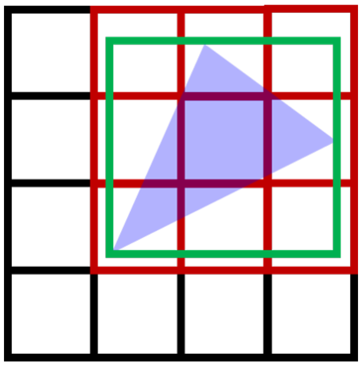

Figure 5: The visualization of the space subdivision by uniform grids. Each object belongs to the cells, which the bounding box touches.

each axis.

3. ***k*-th discrete-orientation polytope (*k*-DOP)**: Each element has the $k/2$ min-max intervals and we first fix the $k/2$-th normal directions. The collision detection is performed by checking the overlap of min-max for all the directions ($k/2$ directions). All the axes are fixed across all the objects.

4. **oriented bounding box (OBB)**: More free version of AABB. It checks along each face normal of two objects and all cross products of edges of those two. Since it checks along more directions, it requires more computation, but it can achieve rotation invariance. Each direction is selected for each object individually.

k-DOP can improve the approximation quality by increasing k, but it leads to expensive computation.

7 Space subdivision

Bounding volume hierarchy divides objects into smaller components and each component has its own bounding box. On the other hand, space subdivision divides space into cells and allocates each element to cells. Then, collision detections are computed by checking **the elements in the same cell**. Space subdivision is **efficient for the larger number of objects**. The implementation is the following:

1. Hash all vertices according to their cell

2. Hash all tetrahedrons (i.e. their vertices) according to the cells touched by their bounding box

[26]The face normals of two bounding boxes and the cross product of two edges.

3. Perform collision detection between vertices and tetrahedrons. If vertices are in the tetrahedrons, it is a collision or a self-collision in the case of their own vertices. This test can be easily performed using Barycentric coordinates.

Note that larger cell size leads to more primitives in a cell and each object belongs to more cells when the cell size is smaller. In the previous research, it has been reported that **the optimal cell size** should be **equal to the bounding box size**. The hyperparameters of this method are cell size, hash table size and a hash function. Each of them significantly affects the performance. For example, a larger hash table is important to reduce the hash collisions, but the positive effect diminishes for too large hash tables. To subdivide the space, the following methods are used:

1. **Uniform grid**: simple and efficient for the larger number of objects. Performance depends on the number of primitives, but not on the number of objects.

2. **K-d tree**: Divide space in the same way that decision tree does and each leaf includes the similar number of vertices. Collision query performs all the elements belonging to the partitions which overlap with the object of interest.

3. **BSP tree**: Generalized K-d tree and divides by linear lines to keep the number of nodes and the depth as small as possible.

Note that since the trees can distribute the number of primitives in each cell, there are only few cells with smaller number of particles and that is why they are more efficient than the uniform grid.