Gestión Eficiente de Interacciones entre Partículas

Práctica 3 - Curso 2023-24

Miguel Cuevas Escrig Carlos Izquierdo Gil

24 de abril de 2024

Índice

1. Introducción

El propósito de esta práctica es implementar dos problemas. En ambos se trata de simular sistema de partículas en los que hay múltiples interacciones entre las mismas. En el primero no se realizará una gestión eficiente de las interacciones mientras que, en el segundo, sí.

2. Implementación

En este apartado se describen las implementaciones realizadas para poder resolver los problemas planteados. Se han implementado dos modelos de colisiones entre particulas , un modelo de colision de la partícula con un plano y dos estructuras de datos para la gestión eficiente de las interacciones entre partículas.

2.1. Partícula-Plano

Para detectar si una partícula colisiona con un plano, se debe de comprobar si la distancia entre la partícula y el plano es menor que el radio de la misma. Para ello se comprueba el resultado de proyectar el vector que va desde el extremo del plano al centro de la particula con la normal del plano, obteniendo un escalar.

Una vez se conoce la distancia, se restituye la particula en la direccion de la normal del plano y se procede a actualizar las velocidades de salida de la particula completando asi el proceso.

El código de la implementación se muestra a continuación:

```
 void planeCollision(ArrayList<PlaneSection> planes)
{
   for (int i = 0; i < planes.size(); i++)
   {
      float distancia = planes.get(i).getDistance(_s);

      if(distancia < _radius && planes.get(i).checkLimits(_s))
      {
         planeCollisionHandler(planes.get(i), distancia);
      }
   }
}

void planeCollisionHandler(PlaneSection plane, float distancia)
{
   float incS = _radius - distancia;
   PVector normal = plane.getNormal();
   _s.add(PVector.mult(normal, incS));

   float vn = _v.dot(normal);
   PVector velNormal = PVector.mult(normal, vn);
   PVector velTangential = PVector.sub(_v, velNormal);

   _v = PVector.sub(velTangential, PVector.mult(velNormal, Kr));
}
```

2.2. Partícula-Partícula

Para detectar si una partícula colisiona con otra partícula, se debe de comprobar si la distancia entre las partículas es menor que la suma de los radios de las mismas. Para ello se calcula el vector que va desde el centro de la partícula hasta el centro de la otra partícula y se comprueba si su magnitud. Una vez se conoce la distancia, se implementan diferentes modelos de colisión para calcular la nueva posición y velocidad de las partículas tras la colisión.

2.2.1. Modelo de Velocidades

El modelo basado en velocidades se encarga de restituir la partícula en la dirección de la normal de la colisión y de actualizar las velocidades de salida de la partícula. Para ello se descompone la velocidad de cada una en sus componentes tangencial y normal a la colisión y se calculan las nuevas velocidades de salida invirtiendo la componente normal y aplicando un factor de restitución para simular la pérdida de energía. Además se ajusta la velocidad de salida para que dependa de las masas de cada partícula siguiendo la fórmula de la colisión elastica.

$$v_1' = \frac{(m_1 - m_2)u_1 + 2m_2u_2}{m_1 + m_2}$$
$$v_2' = \frac{(m_2 - m_1)u_2 + 2m_1u_1}{m_1 + m_2}$$

Siguiendo lo descrito anteriormente, el código de la implementación se muestra a continuación:

```
PVector VUnit = d.copy();
VUnit.normalize();

PVector n1 = PVector.mult(VUnit, _v.dot(d)/distanciaMag);
PVector n2 = PVector.mult(VUnit, otherParticle._v.dot(d)/distanciaMag);
PVector t1 = PVector.sub(_v, n1);
PVector t2 = PVector.sub(otherParticle._v, n2);

float L = minDist - distanciaMag;
float vrel = PVector.sub(n1, n2).mag();
_s.add(PVector.mult(n1, -L/vrel));
otherParticle._s.add(PVector.mult(n2, -L/vrel));

float u1 = n1.dot(d)/distanciaMag;
float u2 = n2.dot(d)/distanciaMag;

float new_v1 = (( _m - otherParticle._m) * u1 + 2 * otherParticle._m * u2)
/ (_m + otherParticle._m);
new_v1 *= Kr;

float new_v2 = (( otherParticle._m - _m) * u2 + 2 * _m * u1)
/ (_m + otherParticle._m);
new_v2 *= Kr;

PVector n1_prime = PVector.mult(VUnit, new_v1);
_v = PVector.add(n1_prime.mult(Kr), t1);

PVector n2_prime = PVector.mult(VUnit, new_v2);
otherParticle._v = PVector.add(n2_prime.mult(Kr), t2);
```

2.2.2. Modelo de Muelles

Este modelo no restituye las partículas, sino que se añaden fuerzas en la dirección opuesta a la colisión que se calculan utilizando un muelle. Para ello se calcula la elongación del muelle, siendo la elongación de reposo la suma entre los dos radios.

$$F_{muelle} = k_e \cdot (distanciaParticulas - distanciaReposo)$$

El código de la implementación se muestra a continuación:

```
float dx = otherParticle._s.x - _s.x;
float dy = otherParticle._s.y - _s.y;
float angle = atan2(dy, dx);

float targetX = _s.x + cos(angle) * minDist;
float targetY = _s.y + sin(angle) * minDist;

float Fmuellex = (targetX - otherParticle._s.x) * Ke;
float Fmuelley = (targetY - otherParticle._s.y) * Ke;

_F.x -= Fmuellex;
_F.y -= Fmuelley;
```

```
    otherParticle._F.x += Fmuellex;
    otherParticle._F.y += Fmuelley;
```

2.3. Estructuras de Datos

Sin estructura de almacenamiento eficiente, el tiempo de cálculo de las interacciones entre partículas aumenta de forma exponencial con el número de partículas porque se deben comprobar cada una de las colisiones con el resto de particulas. Para evitar esto, se han implementado dos estructuras de datos que permiten reducir el tiempo de cálculo reduciendo las comprobaciones de colision a solo las particulas vecinas.

2.3.1. Grid

El grid divide el espacio en celdas de tamaño fijo y asigna cada partícula a una celda. Para comprobar las colisiones, se comprueban las partículas de la celda y las celdas vecinas. Para añadir una particula en la estructura se averiguan los indices de la celda a la que pertenece y se añade a la lista de particulas de esa celda. Para comprobar las colisiones se obtienen las particulas vecinas accediento a la celdas Norte, Sur, Este y Oeste de la celda a la que pertenece la particula. Esto se implementa de la siguiente forma:

```
    void addParticle(Particle p)
{
    int i = int(p.getPos().x/_cellSize);
    int j = int(p.getPos().y/_cellSize);

    if (i >= 0 && i < _nRows && j >= 0 && j < _nCols)
    {
        _cells[i][j]._vector.add(p);
    }
}

ArrayList<Particle> getNeighbors(PVector s)
{
    int i = int(s.x/_cellSize);
    int j = int(s.y/_cellSize);
    ArrayList<Particle> neighbors = new ArrayList<Particle>();
    if (i > 0 && i < _nRows-1 && j > 0 && j < _nCols-1)
    {
        neighbors.addAll(_cells[i][j]._vector);
        neighbors.addAll(_cells[i+1][j]._vector);
        neighbors.addAll(_cells[i-1][j]._vector);
        neighbors.addAll(_cells[i][j+1]._vector);
        neighbors.addAll(_cells[i][j-1]._vector);
    }

    return neighbors;

}
```

2.3.2. Hash

El modelo de hash parte del mismo punto que el grid, dividiendo el espacio en celdas. La diferencia es que no se almacena en una matriz, sino en una tabla hash de una unica dimension. De esta forma se consigue una gestion mas eficiente del espacio de almacenamiento.

Para añadir una particula en la estructura se averiguan los indices de la celda a la que pertenece utilizando la funcion de hash

$$h(i,j) = 73856093 * x + 19349663 * y + 83492791 * z \% M$$

Donde M es el tamaño de la tabla hash y x, y, z son las coordenadas de la particula en la pantalla.

Para poder obtener las particulas vecinas se utiliza, se crean particulas fantasma en las celdas vecinas y se obtienen los indices.

Este proceso se implementa de la siguiente forma:

```
int getIndex(Particle p)
{
   long xd = int(floor(p.getPos().x / _cellSize));
   long yd = int(floor(p.getPos().y / _cellSize));
   long zd = int(floor(p.getPos().z / _cellSize));

   long suma = 73856093*xd + 19349663*yd + 83492791*zd;
   int index = int(suma % _numCells);

   if(index < 0)
   {
      return 0;
   }
   else
   {
      return index;
   }
}

ArrayList<Particle> getNeighbors(PVector s)
{
   ArrayList<Particle> neighbors = new ArrayList<Particle>();

   GhostParticle N = new GhostParticle(new PVector(s.x,s.y-_cellSize));
   GhostParticle S = new GhostParticle(new PVector(s.x,s.y+_cellSize));
   GhostParticle E = new GhostParticle(new PVector(s.x+_cellSize,s.y));
   GhostParticle W = new GhostParticle(new PVector(s.x-_cellSize,s.y));
   GhostParticle SELF = new GhostParticle(new PVector(s.x,s.y));

   neighbors.addAll(_table.get(getIndex(SELF)));
   neighbors.addAll(_table.get(getIndex(N)));
   neighbors.addAll(_table.get(getIndex(S)));
   neighbors.addAll(_table.get(getIndex(E)));
   neighbors.addAll(_table.get(getIndex(W)));

   return neighbors;
}
```

3. Ejercicio 1: Billar Francés

Esta primera tarea consiste en implementar una simulación de un billar francés en el que se simulan las colisiones entre las bolas y las paredes del recipiente. El modelo de colisiones sera el de velocidades. A demás, se permite interactuar con las bolas mediante el click Izquierdo del ratón, y añadir mas bolas con el click Derecho.

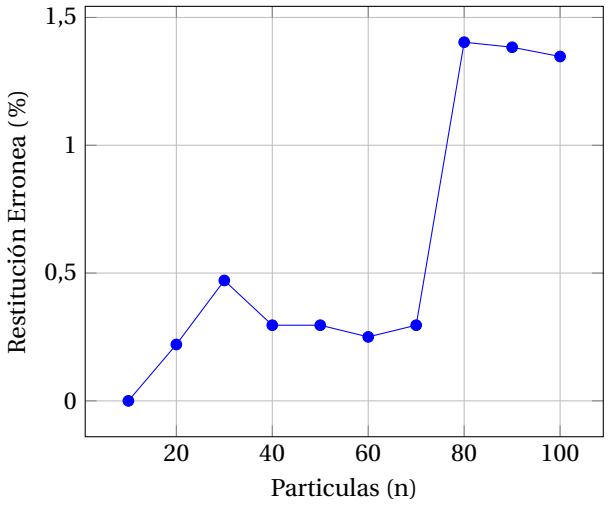
Para simular un rozamiento, se utiliza una constante de rozamiento Kd. Para que las colisiones queden realistas, se utiliza un coeficiente de restitución Cr1 para las colisiones con las paredes y un coeficiente de restitución Cr2 para las colisiones entre bolas.

3.1. Resultados y Análisis

En esta sección se presentan los resultados de la simulación y se analiza el comportamiento del sistema cuando se le somete a diferentes condiciones.

3.1.1. Estabilidad del sistema

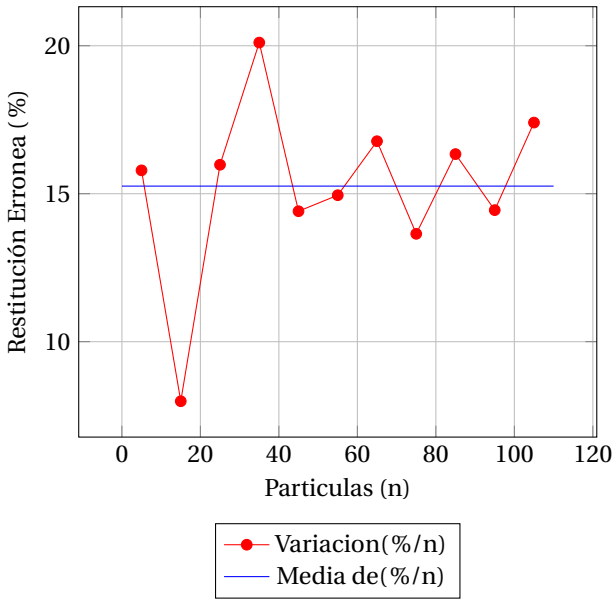
Para analizar la estabilidad del sistema se va a incrementar el numero de bolas de 10 en 10 y hacerlas rebotar entre ellas y con las paredes. Si las particulas no se restituyen como deben, se considerará que el sistema se desestabiliza.



(a) Porcentaje de restitución erronea en función del número de partículas

De lo anterior se deduce que el sistema se desestabiliza de manera mas significativa a partir de 80 particulas. Al aumentar el numero de colisiones tambien aumenta el error de restitución aunque el máximo sigue siendo del 2 %.

Para observar esto de manera más clara, se ha realizado una simulación con una fuerza hacia una esquina para obigar a las particulas a colisionar entre ellas y con las paredes. Los resultados se muestran a continuación:



(a) Porcentaje de restitución erronea en función del número de partículas con una fuerza hacia una esquina

Ahora, el 15,25 % de las colisiones tienen un error de restitución. Al obligar a las particulas a colisionar entre ellas de manera constante, aunque se restituyan correctamente, en el siguiente paso de integración vuelve a haber colisión y el error se acumula.

4. Ejercicio 2: Fluido Biscoso

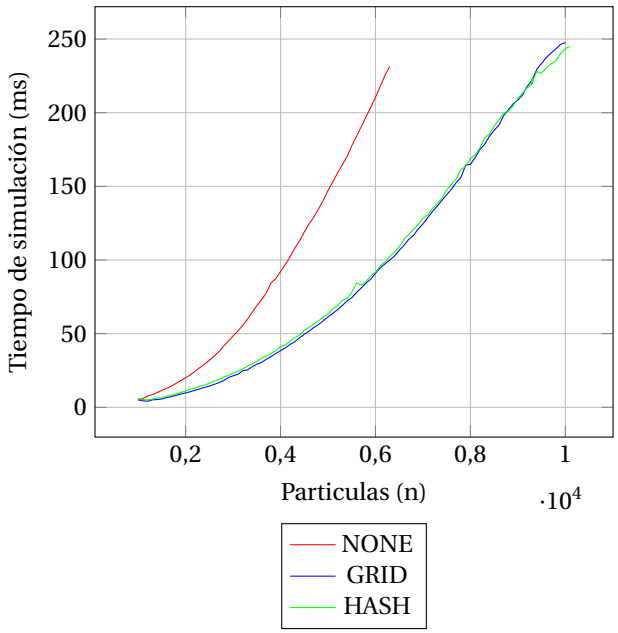
En esta segunda tarea se trata de simular un fluido viscoso en el que las partículas se ven afectadas por la gravedad. Para ello se utiliza el modelo basado en muelles para las colisiones entre partículas. En este sistema se pretende añadir un numero elevado de partículas. Para ello, se utilizan las estructuras de datos grid, hash y fuerza bruta. A demás, con el click Izquierdo del ratón se puede añadir particulas y con la Q se puede eliminar el plano de abajo y hacer se salgan todas fuera del recipiente.

4.1. Resultados y Análisis

En esta sección se presentan los resultados de la simulación y se analiza el comportamiento del sistema cuando se le somete a diferentes condiciones.

4.1.1. Tiempos de cálculo de las colisiones

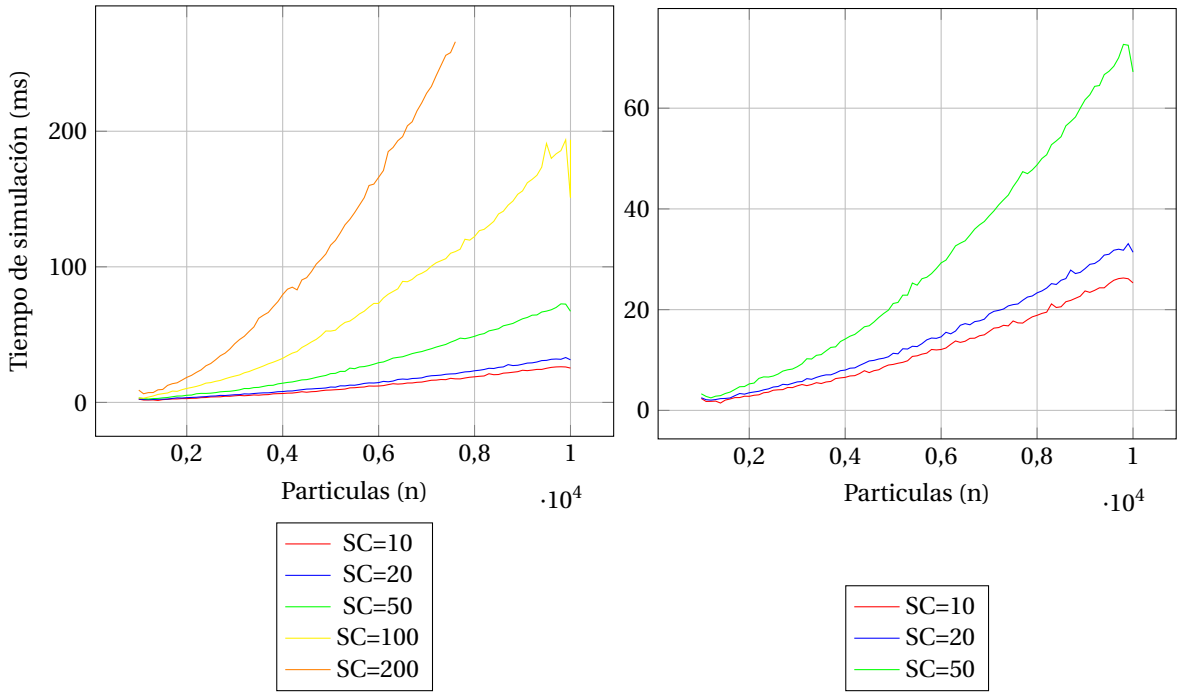
Para analizar el tiempo de cálculo de las colisiones se va a incrementar el numero de particulas de 100 en 100 y se va a medir el tiempo de cálculo de las colisiones con las tres estructuras de datos. Se va a sacar el tiempo promedio en función del número de particulas.



(a) Tiempo de cálculo de las colisiones en función del número de partículas

4.1.2. Tiempos de cálculo de las colisiones en el GRID

Para analizar el tiempo de cálculo de las colisiones en el GRID se va a incrementar el tamaño de las celdas. De esta forma se busca encontrar el tamaño de celda optimo para el sistema.



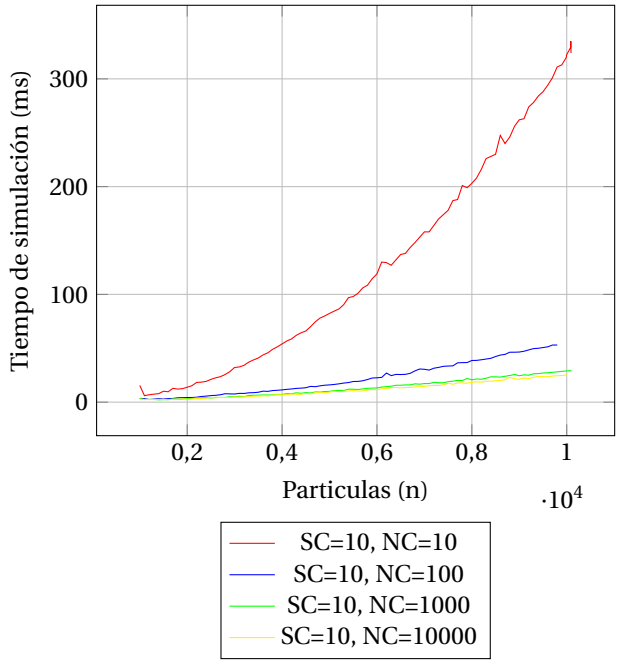
(a) Tiempo de cálculo de las colisiones en función del número de partículas

(b) Tiempo de cálculo de las colisiones en función del número de partículas

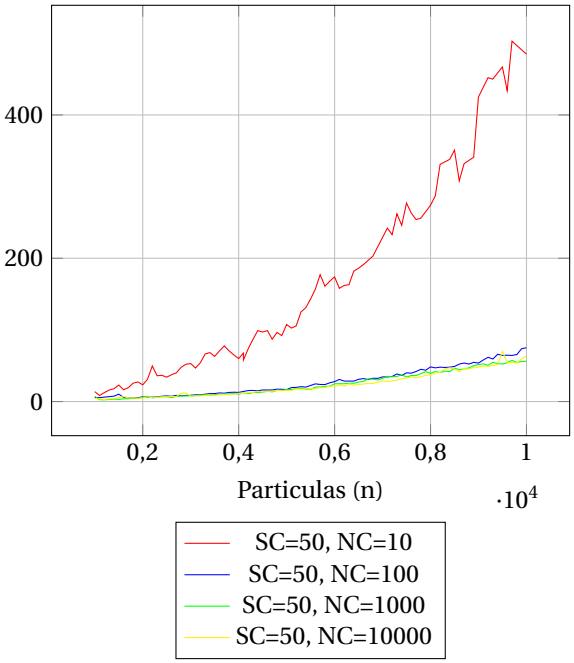
4.2. Tiempos de cálculo de las colisiones en el HASH

Para analizar el tiempo de cálculo de las colisiones en el HASH se va a incrementar el tamaño de la tabla hash a la vez que se cambia el tamaño de las celdas. De esta forma se busca encontrar el tamaño de celda y

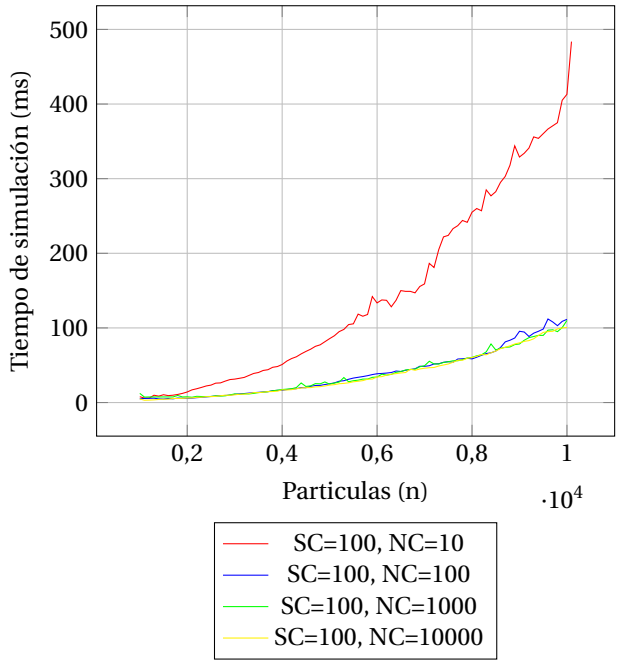
de tabla óptimo para el sistema. El procedimiento de incremento de partículas es el mismo que en el GRID.



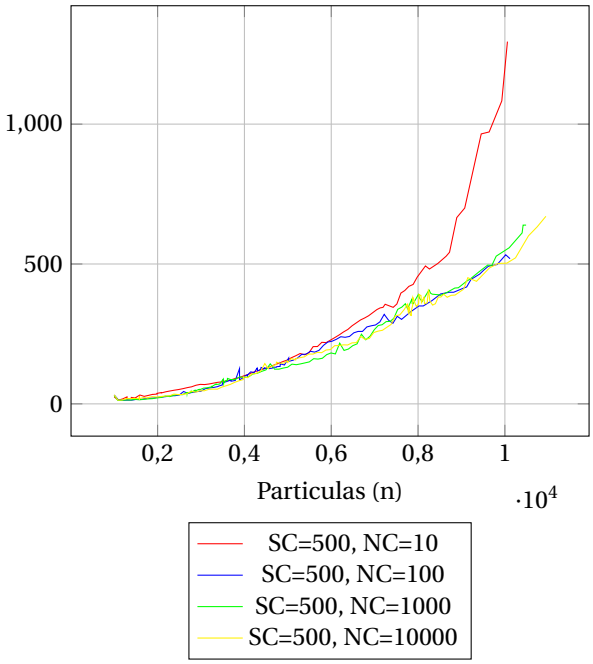
(a) Tiempo de cálculo de las colisiones en función del número de partículas



(b) Tiempo de cálculo de las colisiones en función del número de partículas



(c) Tiempo de cálculo de las colisiones en función del número de partículas



(d) Tiempo de cálculo de las colisiones en función del número de partículas

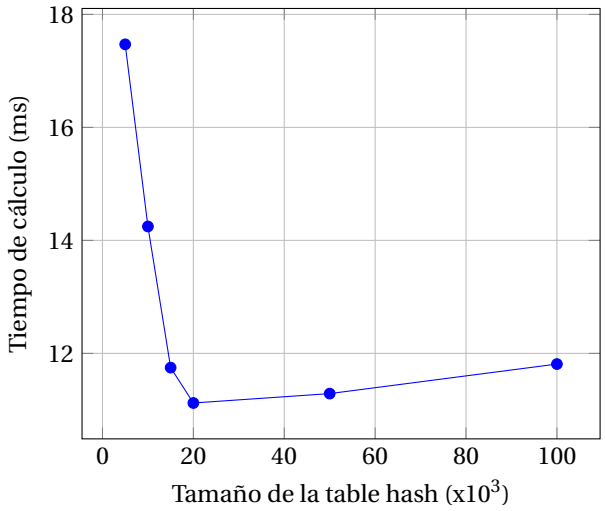
Para analizar mejor el resultado, vamos comparar los tiempos medios totales de cálculo de las colisiones de los mejores casos en cada gráfica.

SC/NC	t (ms)
10/1000	19.0475
10/10000	17.4699
50/100	28.1352
50/1000	24.0232
50/10000	23.7031
100/100	39.1920
100/1000	39.0576
100/10000	36.5332
500/100	175.255
500/1000	176.3743
500/10000	213.5961

Cuadro 1: Parametros de la simulación.

Como se observa, la estructura Hash se beneficia de tener un tamaño de tabla mayor, según la teoría este tamaño deberia del doble de número de partículas. Y tiene sentido, porque así, la cantidad de partículas en cada celda es menor. Por otro lado, el tamaño de celda también afecta al tiempo de cálculo por lo mismo, las partículas no están lo suficientemente distribuidas en la tabla. Si solo existen 4 celdas, todas las partículas irán a los mismos 4 índices independientemente del tamaño de la tabla.

Si se repite la simulación para los parámetros SC=10 y NC=20000, se obtiene que el coste medio es de 11.1217 ms. Como se esperaba, el tiempo de cálculo disminuye al aumentar el tamaño de la tabla hash al número óptimo. Pero, ¿Qué pasa si seguimos aumentando el tamaño de la tabla?



(a) Porcentaje de restitución erronea en función del número de partículas

5. Conclusiones

El sistema de partículas implementado es capaz de simular una hoguera con humo de forma realista. Se ha comprobado que el sistema alcanza un equilibrio segun los parámetros que se le pasen. Se ha comprobado que el tiempo de cálculo aumenta de forma lineal con el número de partículas, y que la diferencia entre el tiempo máximo y mínimo de cálculo también aumenta.