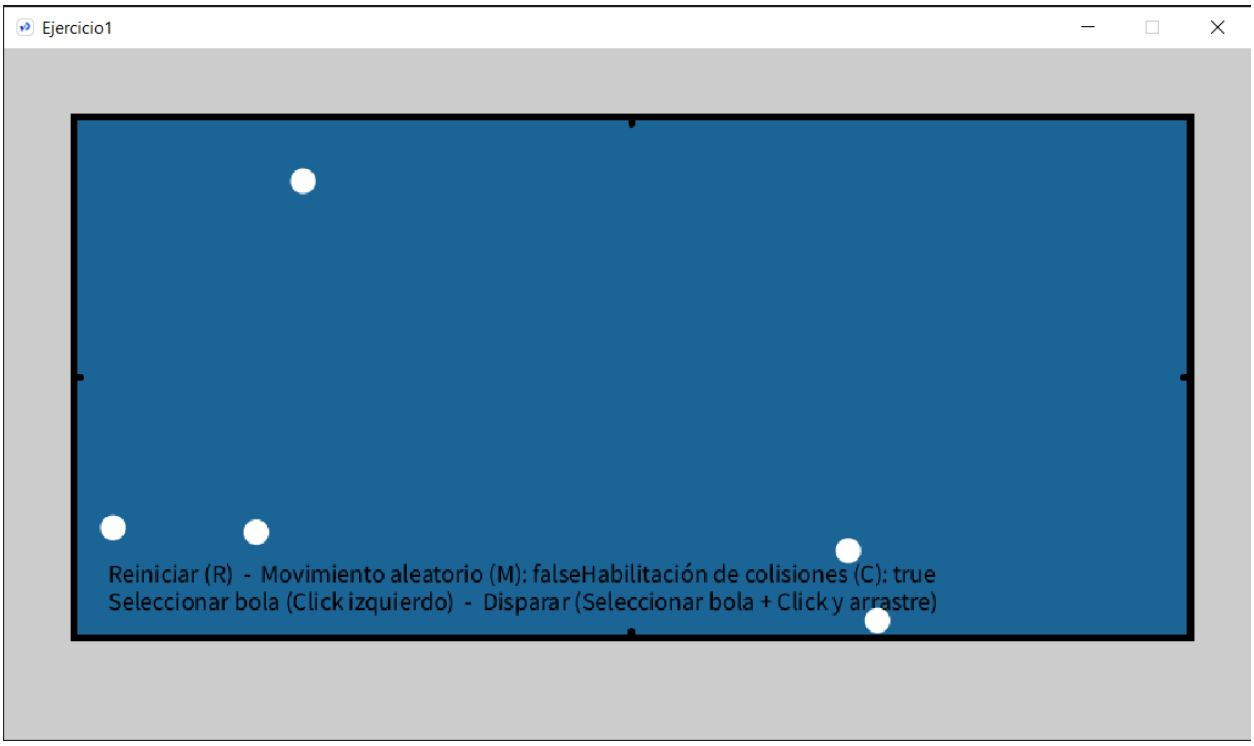
# P3 Modelos de colisión de partículas.

## Autores: Camilo Enguix y Pablo Gómez.

### Tarea 1 - Billar Francés.




Captura del resultado final.

En esta tarea 1 hemos implementado el modelo de velocidades para las colisiones entre partículas y las colisiones contra el plano.

Función calculo de colisión entre partículas:

```
void particleCollisionVelocityModel()
  {
    ArrayList<Particle> sistema = new ArrayList<Particle>(_ps.getParticleArray());
    for (int i = 0 ; i < _ps.getNumParticles(); i++){
```

```
      if(_id != i){ //Todas las partículas menos la nuestra
        Particle p = sistema.get(i);
        PVector vcol = PVector.sub(_s, p._s);
        float distance =vcol.mag();
        float minDist = p._radius + _radius;

        if(distance < minDist)
        {
           PVector dist = new PVector();
           dist.set(vcol);
           dist.normalize();

           //Vectores Normales
           PVector n1 = PVector.mult(dist, (_v.dot(vcol) / distance));
           PVector n2 = PVector.mult(dist, (p._v.dot(vcol) / distance));

           //Vectores Tangenciales
           PVector t1 = PVector.sub(_v, n1);
           PVector t2 = PVector.sub(p._v, n2);

           float L = (_radius + p._radius) - distance;

           PVector res = PVector.sub(n1, n2);
           float vrel = res.mag();


           PVector multN1 = PVector.mult(n1, -L/vrel);
           _s.add(multN1);

           PVector multN2 = PVector.mult(n2, -L/vrel);
           p._s.add(multN2);

           float u1 = n1.dot(vcol)/distance;
           float u2 = n2.dot(vcol)/distance;

           //Velocidades de salida
           float v1 = ((_m-_m)*u1 + 2*_m*u2) / (_m+_m);
           n1 = PVector.mult(dist, v1);

           float v2 = ((_m - _m)*u2 + 2*_m*u1) / (_m+_m);
           n2 = PVector.mult(dist, v2);

           _v = PVector.add(n1.mult(crB), t1);
           p._v = PVector.add(n2.mult(crB), t2);
        }
      }
    }
  }
```

Función calculo de colisión con el plano:

```
void planeCollision(ArrayList<PlaneSection> planes) //Colisión partícula-plano
  {
    //Comprobación
    for(PlaneSection pl : planes){
      float dist = pl.getDistance(_s); //Calculamos a cuánto está la partícula del plano

      if (dist < _radius){ //La distancia es menor que el radio de la partícula
        //Corrección de pos
        PVector normal; //Vector normal
        PVector delta; //Nueva posición
        float resti; //distancia de restitución

        normal = pl.getNormal();
        resti = _radius - dist;
        delta = normal.copy().mult(resti); //Posición nueva para quedar sobre el plano
        _s.add(delta);//Restitución

        //Correción de vel
        float nv; //nueva velocidad: la velocidad de colisión, redireccionada con la normal
        PVector vtan; //velocidad paralela al plano (tangencial) de la nv
        PVector vnorm; //velocidad perpendicular al plano (normal) de la nv

        nv = _v.dot(normal); //que descomponemos en:
        vnorm = normal.copy().mult(nv);
        vtan = PVector.sub(_v, vnorm);
        _v = PVector.add(vtan, vnorm.mult(crP));
      }
    }
  }
```

Para analizar el rendimiento de la aplicación desarrollada, hemos aumentado el numero de partículas de 5 a 100. El sistema se vuelve bastante inestable y la simulación no es muy correcta ya que no hay espacio suficiente en el tablero para recalcular los posicionamientos entre ellas.

Si no hay fricción en el sistema las bolas tardan muchísimo más en detenerse porque no hay nada que les vaya frenando.

No hemos añadido la función de atraer hacía una esquina todas las partículas pero suponemos que si hay una fuerza de atracción , la inestabilidad aumentara

considerablemente ya que la fuerza comienza a ser muy grande y los reposicionamientos se complican al encontrarse muy cerca.

## Tarea 2 - Simulación de fluido.

En esta tarea se ha creado un recipiente con planos(para aplicar el modelo de velocidades partícula-plano) para poder añadir un fluido dentro de el con un modelo de muelle entre las partículas que forman este fluido.

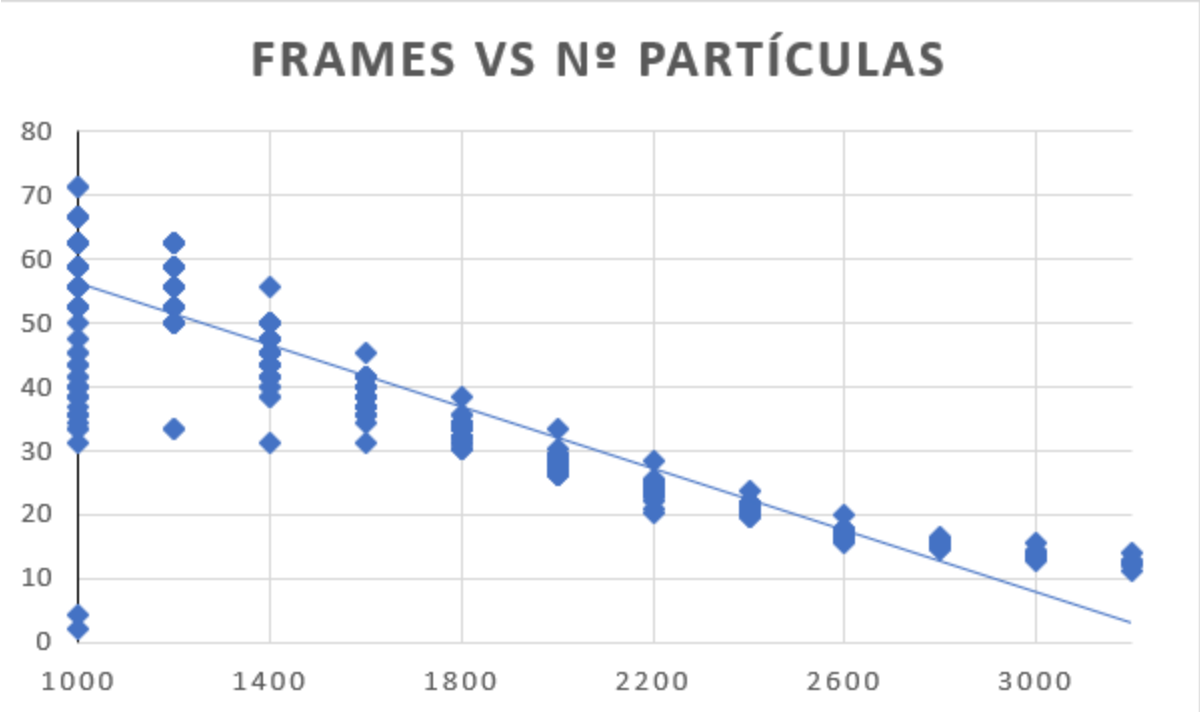
En nuestro caso hemos implementado correctamente la tecla P con la cual se abre la compuerta de abajo, mientras que, por culpa de un funcionamiento extraño del programa(al cual no le hemos sabido dar solución) no hemos podido implementar los diferentes tipos de fluidos.

Con tal de analizar el rendimiento hemos creado tres gráficas diferentes con los siguientes valores de variables:

```
float Kd = -4; //Constante de rozamiento
float crP = 0.6; //Perdida de energia sufrida
float Ke = 0.1; //Constante elástica
float m_bola=40.0;//Masa
float r_bola=5.0;//Radio de la particula
float dp = r_bola + 2;//Distancia entre las 2 particulas
```

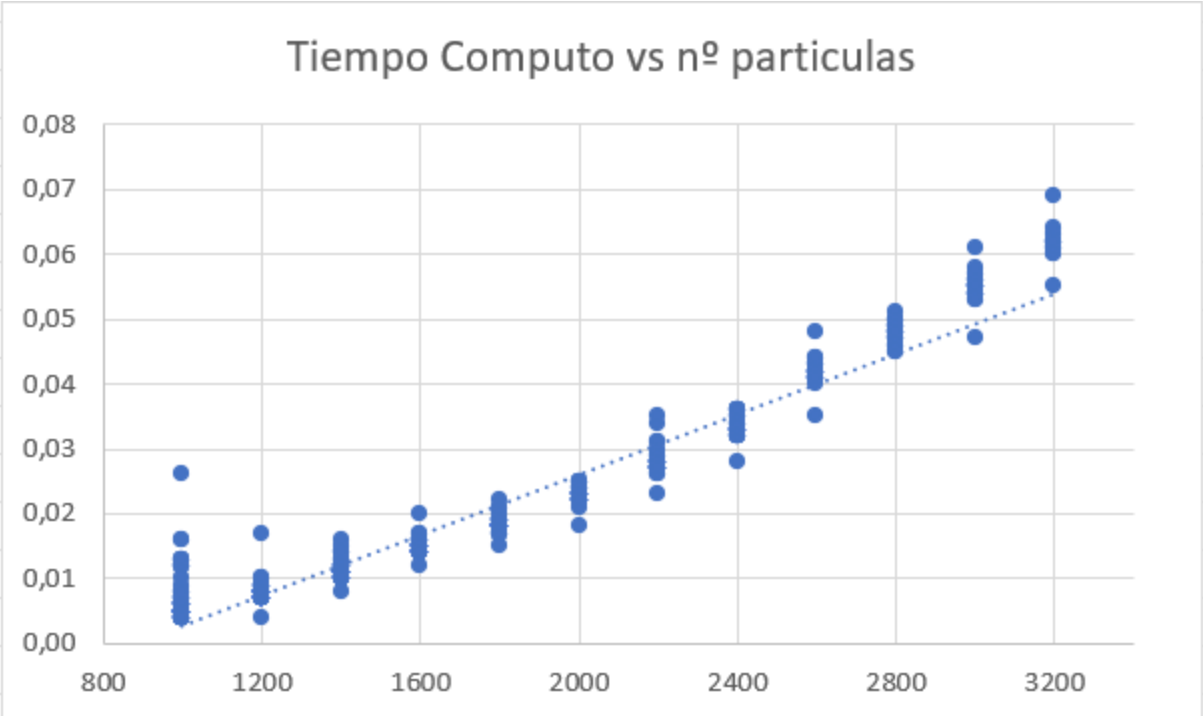
Pruebas de rendimiento:

-Frames por segundo:



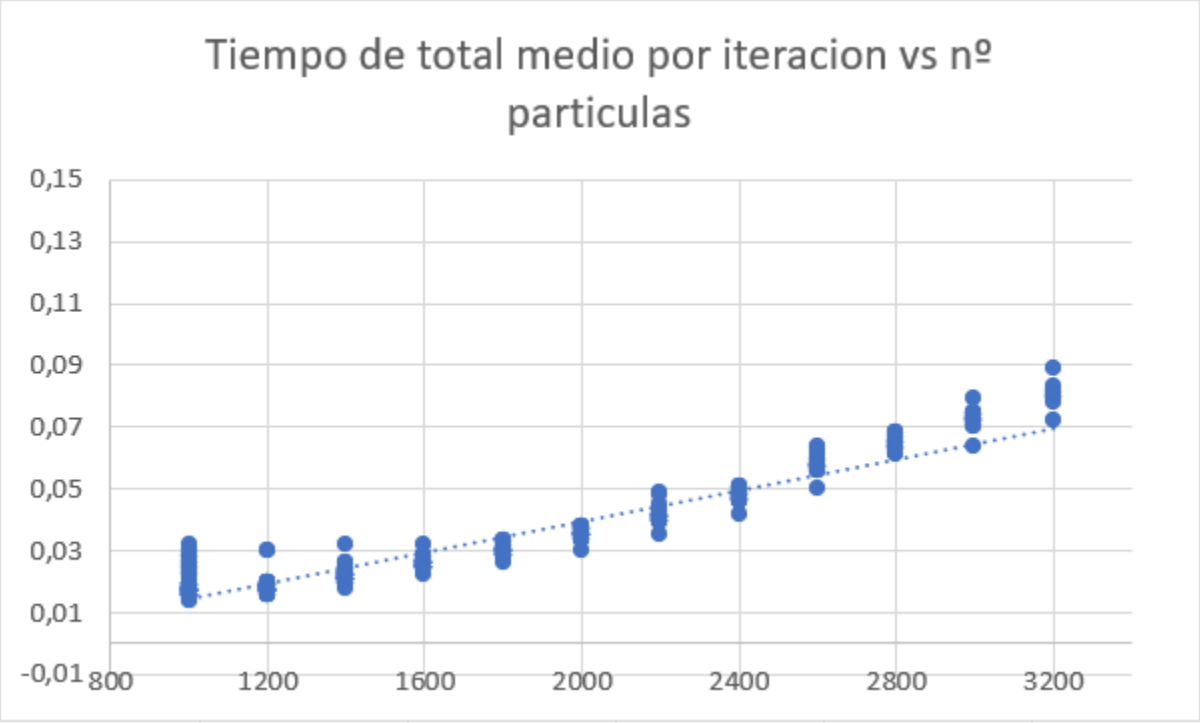

Como podemos observar, a cuantas mas partículas el frame rate se hará mucho mas pequeño.

-Tiempo de computación:



Como podemos observar en esta gráfica el tiempo va aumentando respecto va aumentando el número de partículas(coste de O(n^2)).

-Tiempo medio por iteración:




Al igual que la anterior, podemos observar en esta gráfica el tiempo va aumentando respecto va aumentando el número de partículas(coste de O(n^2)).

## Tarea 3 - Estructura de datos para Mejorar la Escabilidad.

Lo que sacamos en conclusión con esta tarea es que con una estructura Grid se pueden obtener resultados con una eficiencia ligeramente superior a lo que se puede obtener con las tablas Hash.

Pero en lo que se refiere a tiempo de computo lo que es la tabla Hash es superior al Grid. Esto sucede a que el Hash no necesita utilizar comparaciones continuamentes.