P2 Sistemas de partículas

Autores	Camilo Enguix Nadal, Pablo Gómez
Fecha	@March 24, 2022
Asignatura	L1, Simulación

1 Diseño de la física del sistema

1.1 Tipos de palmeras

Diseñar distintos tipos de palmeras. Es decir, distintos tipos de formas de palmeras (circulares, en racimo, con formas pintorescas como corazones, elipses...). Esto se logra configurando adecuadamente las velocidades iniciales del vector que forma el sistema de partículas. En la memoria se debe explicar qué tipos de palmeras se han diseñado y sus ecuaciones.

Círculo

```
/* Para conseguir un círculo, sólo tenemos que multiplicar la velocidad x con el
coseno de un ángulo al azar (comprendido entre 0 y 2*PI radianes), y la y con el
seno de un ángulo al azar. Multiplicando estos valores conseguiremos ampliar o
reducir la figura, según interese. El mismo procedimiento, con distinta fórmula
se seguirá en el resto de formas.
*/
case CIRCULO:
    for (int i = 0; i < 350; i++){
      float angulo = random(0,TWO_PI);
      v = new PVector(random(5,6),random(5,6));
      v.x = 40*v.x*cos(angulo);
      v.y = 40*v.y*sin(angulo);

      _particles.add(new Particle(ParticleType.REGULAR_PARTICLE, _explosionPoint, v, m, ttl, _color));

      _numParticles ++;
    }
    break;
```

Espiral

```
case ESPIRAL:
    for (int i = 0; i < 350; i++){
      float angulo = random(0,TWO_PI);
      v = new PVector(2*angulo,2*angulo);
      v.x = 40*v.x*cos(angulo);
      v.y = 40*v.y*sin(angulo);

      _particles.add(new Particle(ParticleType.REGULAR_PARTICLE, _explosionPoint, v, m, ttl, _color));

      _numParticles ++;
    }
    break;
```

Corazón

```
case CORAZON:
    for (int i = 0; i < 350; i++){
      float angulo = random(0,2*TWO_PI);
      v.x = -80*(3*sin(angulo)-1*sin(3*angulo));
      v.y = -80*(3.25*cos(angulo)-1.25*cos(2*angulo)-0.5*cos(3*angulo)-0.25*cos(4*angulo));

      _particles.add(new Particle(ParticleType.REGULAR_PARTICLE, _explosionPoint, v, m, ttl, _color));

      _numParticles ++;
    }
    break;
```

Capturas de la escena

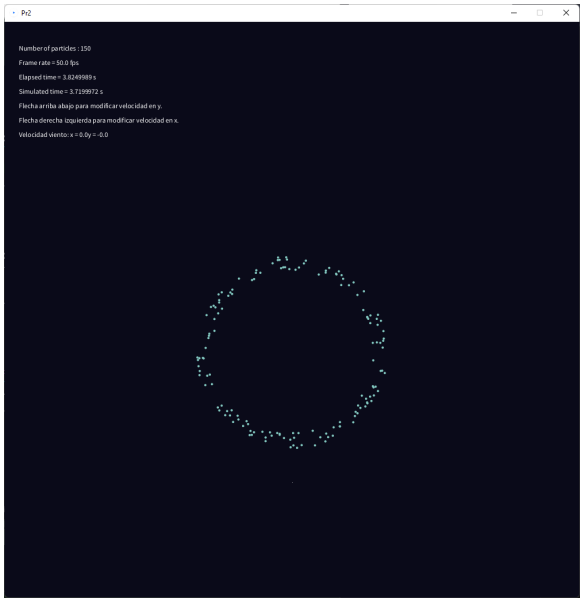


Figura 1. Cohete-círculo

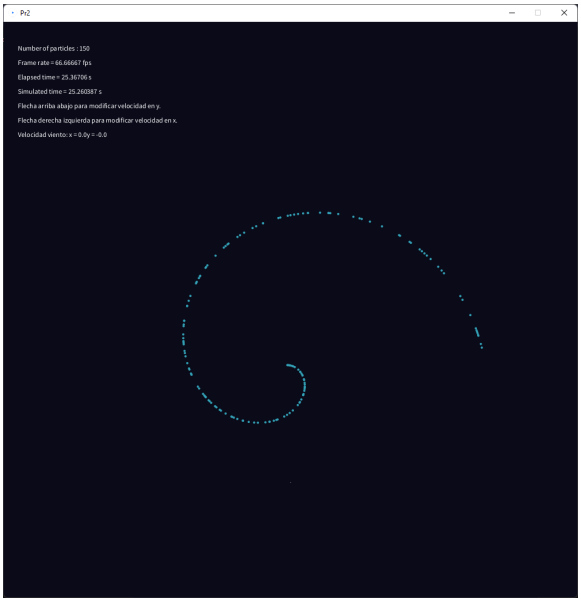
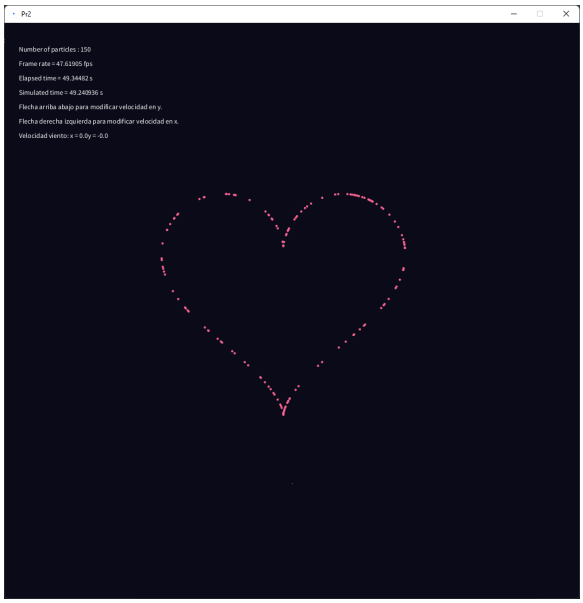


Figura 2. Cohete-espiral



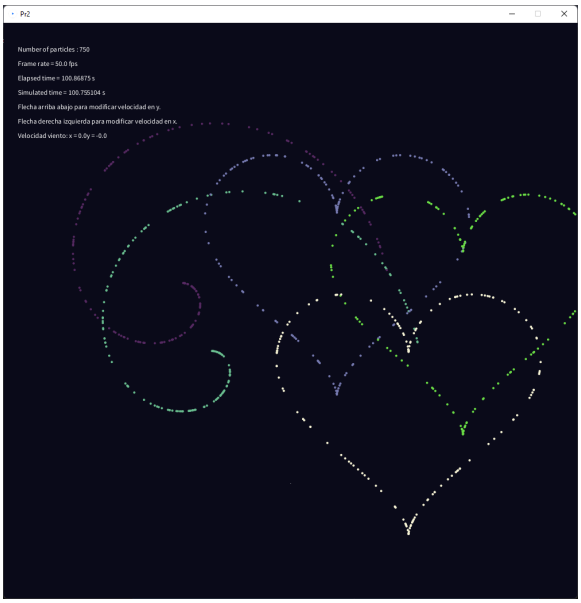


Figura 3. Cohete-corazón

Figura 4. Grupo de cohetes

1.2 Modelo de viento utilizado

Regular el efecto del viento mediante una interfaz (visual o a través de teclado) para configurar la velocidad y la dirección del viento. Las teclas o acciones de ratón usadas deben ser explicadas con textos en la pantalla.

El viento en sí

```
/*Para calcular la deriva que produce el viento en cada partícula, sumamos a la
fuerza de la gravedad, otra (la del viento) formada por una constante WIND_CONSTANT,
multiplicando la dirección del viento y la potencia con la que se mueve.
*/
void updateForce()
  {
    // Código para calcular la fuerza que actua sobre la partícula
    PVector Fw = PVector.mult(G, _m);

    PVector FViento = new PVector();
    PVector magViento = new PVector();

    magViento = PVector.sub(_windVelocity, _v);
    FViento = PVector.mult(magViento, WIND_CONSTANT);

    _F = PVector.add(Fw, FViento);
  }
```

La brújula

```
/* Se dibuja una línea blanca entre el punto que hemos definido como el centro de la
brújula y el punto que representa el incremento de velocidad en ambos ejes, señalando así,
la magnitud del viento.
*/
void drawWind()
{
  // Código para dibujar el vector que representa el viento
  fill(255);
  text("Velocidad viento: x = "+ _windVelocity.x + ", y = " + -1*_windVelocity.y, width*0.025, height*0.2);
  //stroke(126);
  PVector velocidad = new PVector();
  velocidad = _windVelocity.copy();
  velocidad.normalize();
  velocidad.mult(30);

  PVector centroBrujula = new PVector(width/2, height*0.8);
  stroke(255);
  line(centroBrujula.x, centroBrujula.y,_windVelocity.x+centroBrujula.x, _windVelocity.y+centroBrujula.y);
  noStroke();
}
```

Capturas de la escena

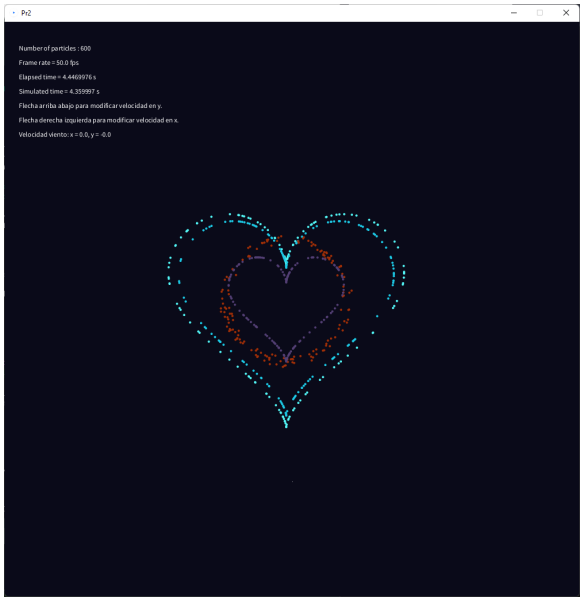


Figura 5. Escena sin viento.

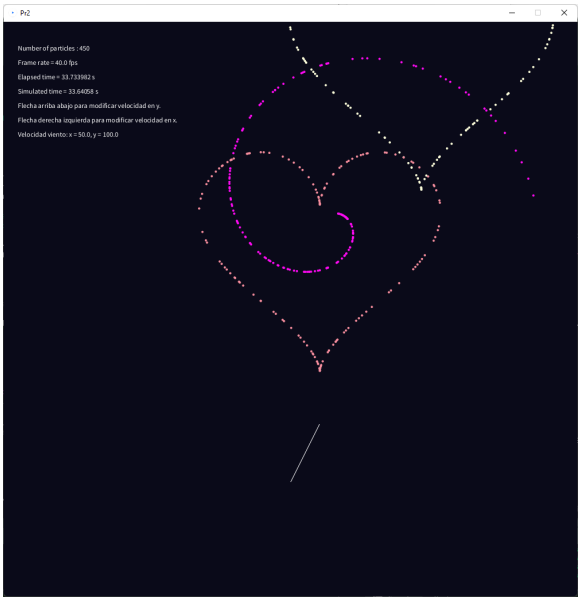


Figura 6. Viento con dirección (100, 50)m/s

2 Resultados obtenidos, análisis e interpretación

Realiza un estudio del rendimiento del sistema haciendo una gráfica donde se represente el tiempo medio que cuesta simular cada paso de simulación (Ts) frente al número de partículas (n) presentes en el sistema.

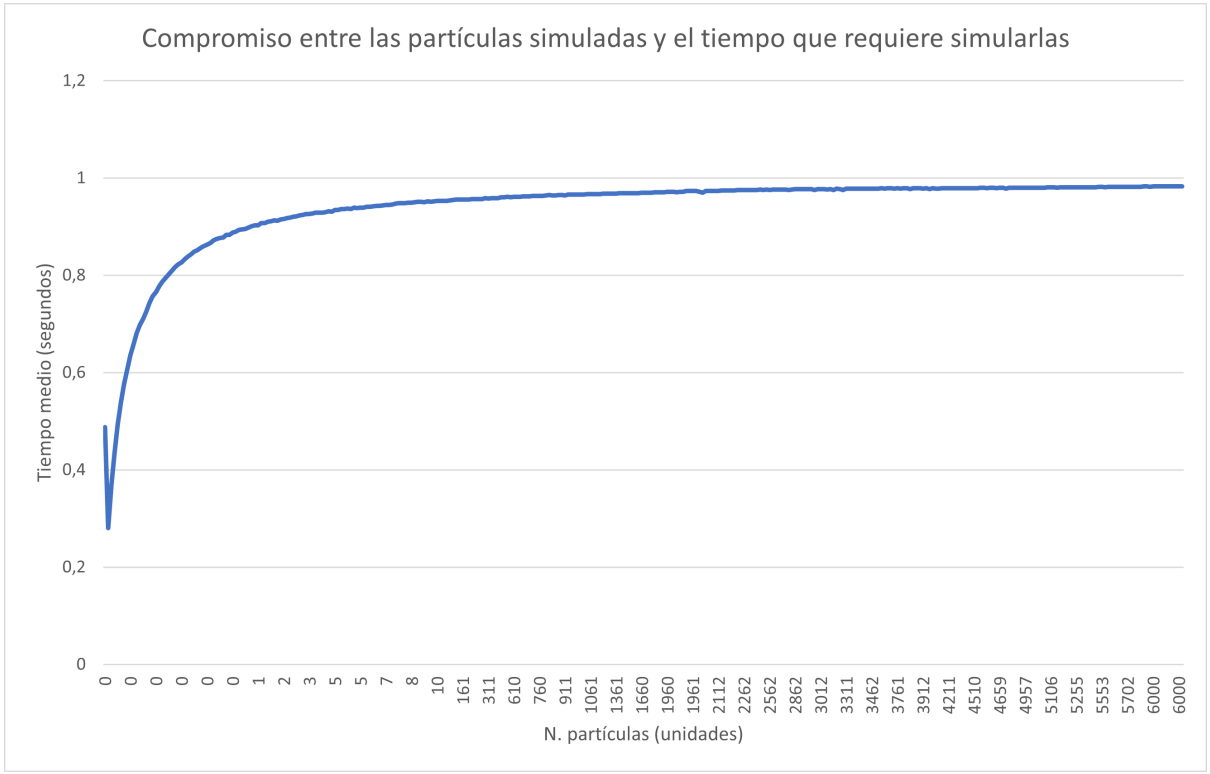


Figura 7. Estudio de rendimiento en segundos

> Conclusión:
> Al aumentar el número de partículas, aumentará el tiempo necesario para simularlas, pero en ordenadores potentes, como es el caso, el tiempo necesario acabará siendo estable ante un aumento lineal (incluso aunque la simulación se haga con java).

Repite el estudio anterior, pero comparando esta vez el frame rate real de la aplicación frente al número de partículas (n) presentes en cada paso de dibujado. Para hacerlo bien se deben tomar varias medidas y promediar. ¿Existe alguna diferencia reseñable con la gráfica del apartado anterior? ¿Por qué?

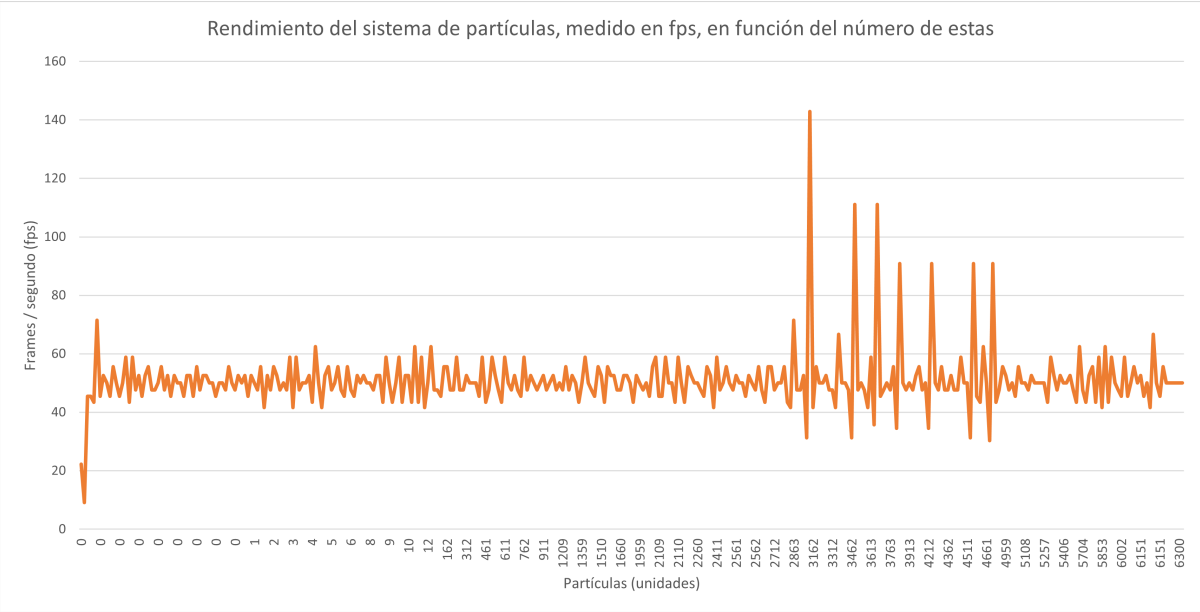


Figura 8. Estudio de rendimiento en fps

> Conclusión:
> La gráfica dista mucho de la anterior, ya que por lo general, el número de frames por segundo es estable, salvo en contados momentos donde se dispara o se reduce, debido a que el sistema acaba, colapsando puntualmente.