INTEGRADOR DE VERLET

Física II

MISIÓN #1

Autores:

Marc Pagès, Arnau Falgueras, Marc Ramos

Roger Promera, Pol Camacho, Carlos Redolar

Pablo Galve, Josep Sànchez, Silvino Medina

Centre de la Imatge i la Tecnologia Multimèdia

22/11/2019

ÍNDICE

1. Introducción

En este documento vamos a describir nuestro integrador de Verlet. Pero, antes de nada, para ponernos en contexto es necesario hacer una introducción para saber qué es el algoritmo de Verlet y qué es un integrador, para así poder entender qué significa cada concepto y el funcionamiento en su totalidad.

1.1 Integrador

Un integrador es un dispositivo que en su salida realiza la operación matemática de integración, es decir, una generalización de la suma de infinitos sumandos, infinitesimalmente pequeños.

1.2 Algoritmo de Verlet

El algoritmo de Verlet es un procedimiento para la integración numérica de ecuaciones diferenciales ordinarias de segundo orden con valores iniciales conocidos. Es especialmente utilizado en las situaciones en que la expresión de la segunda derivada sólo es en función de las variables, ya sea dependiente o independiente, sin participar la primera derivada.

A parte del algoritmo base de Verlet, existe otro basado en la velocidad, el cual es considerado mejor que el algoritmo original. Es bastante parecido al algoritmo Leapfrog, el cual está basado en actualizar posiciones y velocidades en puntos del tiempo intercalados, escalonados de tal manera que se saltan los unos a los otros. Pero, a diferencia de este otro algoritmo, la velocidad y la posición son calculados con el mismo valor que la variable de tiempo.

1.3 Historia

La primera persona en presentar la primera versión de la denominada Integración de Verlet fue el matemático francés Loup Verlet, que lo hizo en el año 1967 y se caracterizaba a la vez por su simplicidad, exactitud y estabilidad. La siguiente versión, propuesta en 1985, fue apodada como “Algoritmo de Verlet con velocidad”. Esta consistía en lo mismo que la anterior, con ligeras correcciones en la integración y con mejoras en la precisión y la estabilidad de las soluciones.

1.4 Objetivos

El objetivo principal del proyecto es implementar un integrador basado en el algoritmo de Verlet, el cual tenga un óptimo funcionamiento y que no solo esté basado en cálculos matemáticos y valores, sino que también disponga de una parte gráfica que proyecte todos los cálculos de una forma más visual y fácil de entender.

A parte, no queremos introducir sólo las ecuaciones básicas del algoritmo de Verlet, sino ir más allá e implementar las máximas ecuaciones posibles para que el integrador sea lo más completo posible.

Además, queremos que el usuario, a través de la consola, pueda determinar cuáles son los valores de las diferentes unidades de las fórmulas, para que el resultado que se refleje de forma gráfica sea el que la propia persona haya introducido en el menú inicial.

2. Desarrollo y código

Antes de empezar con las descripciones de todas las funciones es importante dejar claro que todos los valores que las funciones tienen que recibir para que cada una de ellas haga sus cálculos propios pueden provenir de diferentes sitios. Uno de ellos es que los proporcione el propio usuario a la hora de completar el menú. Otra, y la última, es la de que otra función le proporcione información que la función inicial necesite.

2.1 Menú

Aunque en la explicación de cada función se explica cuál es su funcionamiento, creemos que es necesario explicar específicamente cómo funciona y la forma en que lo ve el usuario, en este apartado.

Así pues, lo que hace el menú es, en primer lugar, hacer escoger al usuario entre cuatro opciones: calcular valores específicos, hacer una representación gráfica de la posición de la partícula, simularla en tiempo real o salir del menú y del integrador.

- En el caso de que el usuario escoja la primera opción, el programa le volverá a preguntar por si quiere calcular una posición en un momento concreto, el tiempo que tarda la partícula en alcanzar un punto en concreto o la velocidad terminal de la partícula. Entonces, en todas ellas el usuario deberá introducir valores que se piden en las tres opciones, pero habrá algunas preguntas concretas de cada uno, como, por ejemplo, en la primera opción se pide el momento concreto en que se quiere calcular la posición.
- En el segundo lugar, tenemos la opción de que se haga una representación gráfica de la particular. En este caso, se pide al usuario que introduzca una posición x y una posición y, lo mismo con la velocidad, una aceleración, que puede ser la gravedad a secas, o la gravedad teniendo en cuenta el rozamiento, la densidad de la partícula, etc. y la masa de esta. Una vez que todos los valores estén introducidos, él integrador procederá a hacer la representación gráfica del comportamiento de la partícula.
- Por último, en el caso de que el usuario escoja la opción de la simulación en tiempo real, se pedirá al usuario exactamente los mismo que en el caso anterior, pero a la hora de hacer la representación gráfica, esta será de forma real, es decir, el comportamiento exacto de la partícula momento a momento.

2.2 Main Verlet functions

2.2.1 Initial Situation

Esta función ha sido diseñada para el correcto funcionamiento de *VerletIntegration*, la cual explicaremos en el siguiente punto. Aun así es necesario empezar a explicarla, para así poder explicar bien cuál es la función de esta. La función *VerletIntegration*, lo que se hace es calcular la posición que tendrá la partícula en la siguiente iteración mediante la utilización de una posición actual y una posición previa, una aceleración y un tiempo. Así pues, en la primera iteración, la posición actual, la aceleración y el tiempo son proporcionados por el usuario o ya están estipulados por el integrador. En cambio, para poder calcular la posición previa, es necesario utilizar una ecuación aparte, y aquí es donde entra esta función.

Así pues, dado que esta ecuación sí que utiliza la velocidad y sumado a todos los valores que hemos dicho anteriormente, esta función es capaz de calcular la posición previa de la partícula en la primera iteración para poder proporcionarle a la función de Verlet Integration y pueda calcular la nueva posición de la partícula, tanto en la primera iteración como en las siguientes.

2.2.2 Verlet integration

$$\vec{x}_{n+1} = 2\vec{x}_n - \vec{x}_{n-1} + \vec{A}(\vec{x}_n)\,\Delta t^2.$$

Esta es la ecuación base de la integración de Verlet, la cual hemos utilizado en primer lugar y tiene como componentes la posición, la aceleración y el tiempo.

Para entender cuál es la implementación de código que hemos hecho en esta ecuación, antes de todo es necesario entender qué es lo que hace la ecuación en sí, los valores necesarios para calcularla y cuál es el resultado.

Antes que nada, decir que es una ecuación que lo que hace es calcular la siguiente posición de una partícula, partiendo de una posición actual sumado a la resta de la posición actual y la posición previa. Entonces se suma a la multiplicación de la aceleración por el tiempo al cuadrado.

2.2.3 Velocity Verlet

$$\vec{v}(t+\Delta t) = \vec{v}(t) + \frac{\vec{a}(t) + \vec{a}(t+\Delta t)}{2}\Delta t.$$

En este caso, nos encontramos delante de una ecuación parecida a la base del algoritmo de Verlet, pero en este caso, basado en la velocidad.

Lo que hace esta ecuación es calcular la nueva velocidad de una partícula en concreto, mediante el uso de una velocidad actual, una aceleración y un tiempo. Así pues, lo que hace la ecuación es coger la velocidad actual de la partícula, sumarla a la suma de la mitad de la aceleración previa y la aceleración actual y todo ello multiplicado por el tiempo.

2.2.4 Stormer Verlet

$$Vn+1 = (Xn+1-X) / dt$$

A parte de la ecuación anterior, la cual es utilizada para calcular la velocidad de la particular, hay otras formas para poder obtener esta velocidad, como es en este caso la ecuación de Stormer Verlet. El proceso de obtención de la velocidad se divide en cuatro casos en los que dependiendo del valor de la aceleración calcularemos el nuevo valor de la velocidad de una manera más o menos compleja. En el primer caso si la aceleración en los dos ejes es igual a cero simplemente tendremos que dividir el incremento de posición entre el diferencial de tiempo.

En el caso de que solo valga cero en el eje x calcularemos el nuevo valor de la velocidad en el eje x mediante el mismo método de antes y calcularemos su nuevo valor en el eje y calculando el incremento de posición, restando a este la mitad del producto de la aceleración por el diferencial de tiempo al cuadrado y dividiendo el resultado entre el diferencial de tiempo.

En el caso de que sea al revés y sea la aceleración en el eje y la que valga cero y en el eje x sea diferente de cero haremos el proceso inverso al de antes.

Por último si tenemos una aceleración en los dos ejes haremos el cálculo que he explicado en el segundo caso para el eje y pero en este caso para la velocidad en los dos ejes.

2.3 Cálculo de colisiones

2.3.1 Check Collision

Esta función solamente comprueba si existe una colisión entre la partícula y un objeto dentro del mundo desde todas las direcciones posibles (por arriba, por abajo, por la derecha y por la izquierda.

Empezamos creando un booleano llamado ret, y le asignamos el valor false. Este se volverá true cuando exista la colisión, y seguirá falso cuando no la haya.

Para que exista la colisión, se deben cumplir 4 condiciones que están incluidas dentro del if: La primera es si la posición x de la partícula más su radio es igual o superior a la posición x del objeto. La segunda, si la posición x de la partícula menos su radio es igual o menor a la posición x del objeto más su anchura. La tercera, si la posición y de la partícula menos su radio es igual o menor a la posición y del objeto más su altura. Y la cuarta, si la posición y de la partícula más su radio es igual o mayor a la posición y del objeto.

Si todos estos 4 casos se cumplen, entonces el *booleano ret* se vuelve *true* y la función retorna ese valor, ya que eso significa que está colisionando.

A continuación, existe otra función de *CheckCollision*, pero con la diferencia de que retorna todas las posibles colisiones que puedan existir entre la partícula y el objeto.

2.3.2 Calculate Collision Position

Una vez sabemos que se ha producido una colisión, entonces utilizamos esta función para saber qué posición ocupa la partícula justo antes de que esta colisione. Para eso, necesitaremos una variable float llamada time, la función Calculate Time (que se explicará más adelante) y dos booleanos (*col_x y col_y*) que representan las colisiones en x y en y.

Empezamos igualando la variable time a 0 y los booleanos a false, ya que solo cuando haya una colisión en ese eje se volverá true (eso lo determinaremos con la posición anterior a la colisión de la partícula) y calculamos la velocidad en la posición anterior para después poder calcular bien el tiempo con la función *StormerVerlet.*

Si la posición x previa de la partícula (no la actual) más su radio es más pequeña que la posición x del otro objeto significa que ha colisionado con un objeto a su derecha (eje x), entonces igualamos la variable time a la función *CalculateTime*, donde le pasamos la posición previa de la partícula, su velocidad en x, su aceleración en x y la posición x del objeto menos el radio de la partícula. El booleano *col_x* se iguala a true.

Si la posición x previa de la partícula menos su radio es superior a la posición x del objeto más su anchura significa que ha colisionado con un objeto a su izquierda (eje x), igualamos la variable time a la función *CalculateTime* y le pasamos las mismas variables que antes,

pero añadiendo la anchura del objeto y sumandole el radio en vez de restarle. El booleano *col_x* se iguala a true.

Si la posición y previa de la partícula más su radio es inferior a la posición y del otro objeto significa que ha colisionado con un objeto abajo (eje y), entonces igualamos la variable time a la función Calculate Time y le pasamos las mismas variables que en la primera condición, pero en el eje y. Ahora el booleano *col_y* se iguala a true.

Por último, si la posición y previa de la partícula menos su radio es superior a la posición y del objeto más su altura significa que ha colisionado con un objeto arriba (eje y), entonces igualamos la variable time a la función *CalculateTime*, y le pasamos las mismas variables que en la segunda condición, pero en el eje y y cambiando la anchura por la altura. El booleano *col_y* se iguala a true.

Una vez calculado el time, ya podemos calcular con exactitud la posición de la partícula en el momento del choque. Y lo hacemos basándonos en esta fórmula de cinemática:

$$x = x_o + V_x t - \frac{1}{2} a t^2$$

Donde x es la posición en el choque, x_0 es la posición previa a la colisión, V_x es la velocidad de la partícula (que hemos calculado al principio de esta función), t es la variable time previamente calculada, y a es la aceleración de la partícula.

Finalmente, calculamos la velocidad de la partícula durante la colisión y la reducimos multiplicandola por 0.9. Además, utilizamos los booleanos *col_x y col_y.* Si *col_x* es true, entonces la velocidad final en x será ella misma menos ella misma. Y si *col_y* es true, pasará lo mismo, pero con la velocidad en y.

2.3.3 Calculate Time

Esta función, como hemos dicho anteriormente, se utiliza siempre que la partícula colisiona. Con ella calculamos el tiempo total que tarda el objeto hasta el punto de colisión. Para ello se usa la posición previa del cuerpo, la posición nueva (justo en el momento anterior que empieza a colisionar), la velocidad inicial en ese intervalo de tiempo y la aceleración (todas estas variables solo recibimos el eje que nos interesa para la colisión, definido en la función *CalculateCollisionPosition*), también el intervalo de tiempo entre dos posiciones.

Definimos 4 variables float de tiempo: t*ime, time1, time2 y tpow*.

Si la aceleración es 0, aislamos el tiempo de la fórmula de cinemática:

$$1.\, x = x_o + V_x t$$

Y retornamos la variable time.

Si la aceleración no es 0, entonces calculamos el *tpow*, que no es más que aislar el tiempo de la fórmula del MRUA:

$$x = x_o + V_x t - \frac{1}{2} a t^2$$

Como al aislar el tiempo utilizamos una ecuación de segundo grado, el tiempo que aislado de esta manera:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

=*tpow*, siendo b la velocidad de la partícula, a la aceleración y c la resta de la posición inicial menos la final.

Como hay 2 soluciones posibles, el tiempo final se calcula con 2 variables, que son el time 1 y el time 2 respectivamente.

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Siendo *tpow*, como hemos dicho antes, la solución de la raíz cuadrada, el time 1 será la solución usando el signo positivo, mientras que time 2 será la solución usando el signo negativo.

Finalmente, ponemos 2 condiciones. Si el time 1 es más grande que 0 y más pequeño que el intervalo de tiempo, este representa el tiempo total que ha durado la colisión, y se iguala a time (ya que es la solución correcta). Por el contrario, si time 2 es superior a 0 y más pequeño que el intervalo de tiempo, será este el que se igualará a time (ya que es la solución correcta). Si ninguno de los 2 es superior a 0, significa que el tiempo de colisión será 0. Al ser el tiempo algo que nunca puede ser negativo, solo puede haber una solución adecuada en la fórmula anterior.

2.3.4 Calculate Collision Final Position

Esta función se encarga de calcular tanto la posición actual después de haber colisionado y rebotado como la posición anterior a la actual (siguiendo la misma dirección que ha obtenido después de colisionar) para así poder obtener las próximas posiciones de la partícula con Verlet. Este cálculo se efectúa después del rebote, cogiendo la posición actual (el intervalo de tiempo de la partícula menos el tiempo que ha necesitado para llegar hasta ese punto, calculado con la función *CalculateTime*) y la posición previa.

A esta función le pasamos todas las variables de la partícula y el tiempo, de esta manera tenemos los datos de las posiciones, las velocidades, aceleraciones de la partícula y el tiempo total. Por lo tanto con la fórmula básica deberíamos poder predecir su siguiente trayectoria.

time = particle.dt - time

*particle.pos = particle.prev_pos + particle.v * time + particle.a * 0.5 * (time * time)*

*particle.prev_pos=particle.pos - particle.v * particle.dt - particle.a * 0.5 *particle.dt * particle.dt*

2.4 Cálculo de posición y velocidad

2.4.1 Acceleration sum & Drag acceleration

No se puede entender la función de Acceleration Sum sin antes calcular la de *DragAcceleration.* Aquí asumimos que la aceleración de la partícula no es constante y varía según una serie de factores que veremos a continuación.

$$F_D = \frac{1}{2}\rho v^2 C_D A$$

En este caso, con la *DragAcceleration* calculamos la desaceleración de una partícula cuando existe una fuerza de rozamiento (pongamos por caso, con el aire) contraria al movimiento de la partícula. Esta aceleración, que puede ser en el eje x o y, depende de la densidad, del coeficiente de rozamiento, el área de la partícula, la velocidad (en x o y) y la masa de la partícula.

Por lo tanto, el resultado es la multiplicación de la densidad por el coeficiente de rozamiento por el área por ½, todo ello multiplicado por la velocidad al cuadrado (dependiendo de si queremos coger la aceleración en x o y tendremos que utilizar la velocidad en x o y) y todo lo anterior dividido entre la masa.

En el menú inicial le preguntamos al usuario si quiere tener una única aceleración, la gravedad. Si la respuesta es NO, entonces le pedimos que dé valores de gravedad, de coeficiente aerodinámico, de densidad y de masa. La velocidad ya está definida por el usuario anteriormente.

Cabe decir que esta desaceleración es en dirección opuesta al movimiento de la partícula, y por lo tanto se deberá restar a la aceleración final como veremos a continuación.

Con esto ya podemos calcular la Acceleration Sum, que no es más que el cálculo de la aceleración total en un momento determinado. Para ello, mediante un fPoint, creamos la suma de aceleraciones en x y en y, y las inicializamos en 0. Después, igualamos la suma a la aceleración de ese momento de la partícula, y para la suma en y, se le añade la gravedad. Finalmente, la suma total de la aceleración será ella misma menos la *DragAcceleration.*

2.4.2 Terminal Velocity

Esta función retorna la velocidad de la partícula en el momento en que la aceleración es constante. Cuando esta partícula cae con una gravedad y una aceleración, la velocidad aumenta. Pero llega un momento en el que la *DragAcceleration* y el peso de la partícula (su masa) se igualan, y la velocidad es constante.

2.4.3 Parachutist Acceleration

En el caso que la opción que elija el usuario en base a la aceleración sea únicamente la gravedad esta función no tiene sentida ser aplicada, pero en caso de que elija que existan fuerzas de rozamiento, densidad, área y demás si que se puede aplicar esta función.

Para calcular la aceleración, lo que hace la ecuación es multiplicar la masa negativa por la gravedad y sumarlo a dos veces la gravedad y a una constante. Finalmente, se divide todo esto a la masa de la partícula.

2.4.4 Freefall Speed

$$v_t = \sqrt{\frac{2mg}{\rho C A}}$$

El objetivo de esta función es retornar la velocidad de la partícula en el caso que el usuario elija qué cómo aceleración existan fuerzas de rozamiento, la densidad del aire, el área de la partícula, etc. Así pues, para calcular la velocidad se multiplica dos por la gravedad y por la masa. Entonces, se divide todo esto a la multiplicación de la densidad del aire, el área y la fricción del aire. Finalmente, se eleva al cuadrado toda la ecuación.

2.4.5 Freefall Acceleration

El objetivo de esta función es muy parecido que el de la función anterior, ya que lo que hace es devolver la aceleración de la partícula en el caso de que el usuario lo desee.

Las componentes que necesita la ecuación para poder calcular la aceleración son la masa, la gravedad y la fricción del aire. En primer lugar, se multiplica la masa por la gravedad y se le resta la masa por la gravedad y por la fricción del aire. Finalmente, se divide todo ello por la masa de la partícula.

2.5 Cálculo de posición

2.5.1 Classical Motion

*Xn+1 = X + V*dt + ½ a*dt^2*

En el caso de la función *ClassicalMotion*, su funcionamiento es prácticamente el mismo que en la función base de Verlet, la única diferencia es que en este caso sí que se utiliza la velocidad para calcular la nueva posición de la partícula, y en consecuencia, no necesita otra función que le calcule la posición previa. Así pues, lo que hace es esta ecuación es, por decirlo de alguna manera, sustituir la posición previa por la velocidad, para así poder calcular la posición.

Su funcionamiento pasa por diferentes valores que introduce el usuario al inicio, que son: la posición actual de la partícula, la velocidad y la aceleración, sin olvidarnos del tiempo que es establecido por nosotros. El calcula se realiza de la siguiente manera:

En primer lugar, se suma la posición actual a la velocidad multiplicada del tiempo y seguidamente, se multiplica a la multiplicación de la mitad de la aceleración y el tiempo al cuadrado.

2.5.2 Time to Position

El objetivo de esta función es devolver cuanto tiempo tarda la partícula en llegar a una posición en concreto. Para ello utiliza una función (*ClassicalMotion*)explicada anteriormente para poder adquirir variables necesarias para el cálculo.

El funcionamiento de la función es el siguiente:

- Lo primera que hace la función es calcular algunas variables que son necesarias para el cálculo del tiempo. Estas pueden ser dadas por el usuario en el menú o haciendo cálculos con otras funciones, como por ejemplo la función *ClassicalMotion,* la cual hemos explicado anteriormente.
- En el momento en el que el usuario elige si quiere realizar el cálculo en el axis x o en él y, la función acceda a un if o a otro. En ambos se realizan los mismos cálculos, pero cada uno en sus respectivos ejes.
- Así pues, los cálculos realizados por la función son, en primer lugar, un bucle el cual va iterando hasta que llegue al máximo de las iteraciones posibles (cuando ya ha llegado a la posición). En este bucle, en el caso que todavía no se haya llegado al máximo de las iteraciones se ejecuta una conmutación, es decir, se atribuye la posición de la partícula a una variable que se guarda. Seguidamente, mediante la función *VerletIntegration*, se calcula la nueva posición de la partícula y se le atribuye este valor a la posición de la partícula. Finalmente, se atribuye el valor inicial guardado a la posición previa para que en la siguiente iteración se pueda realizar el cálculo de la función *VerletIntegration* otra vez.

2.5.3 Position at Time

Esta función hace exactamente lo mismo que la función anterior, pero a la inversa. Es decir, calcula la posición de la partícula en un instante determinado.

Esta función tiene como variables la posición de la partícula, su velocidad y su aceleración en los 2 ejes y el tiempo total en el que se calculan las posiciones.

El procedimiento es similar, ya que, como en la Time To Position, se utiliza la función Classical Motion, que ya se ha explicado antes.

En primer lugar, creamos 2 fPoint: uno para la posición previa y otro que será un auxiliar. Un float llamado *time_passed* que lo igualaremos a dt, con valor de 1.0f. La posición previa la igualamos a la variable de posición que le hemos pasado en la función, y a su vez, está la igualamos a la función de Classical Motion. Luego hacemos un bucle while en el que se incremente el time passed en 1 en cada ciclo mientras este sea menor que el tiempo total. Igualamos la *aux_position* a la posición en ese momento, ya que esta actuará como posición previa, y calculamos la posición nueva de la partícula con la función de Verlet Integration. Una vez hecho, igualamos la *aux_position* a la *prev_position* e incrementamos en 1 el time passed. De esta forma, con cada ciclo, la posición de la partícula cambiará, siendo la posición nueva del anterior ciclo la posición antigua del nuevo, y así sucesivamente.

En el menú, si el usuario elige la opción 1 (Calculate Specific Values) y otra vez la 1 (Position at Certain Time), introducirá valores de posición, velocidad y aceleración en los 2 ejes, y por último el tiempo total para calcular dichas posiciones. En la consola, saldrán tantas “líneas” como ciclos tenga el bucle, y en la línea final, saldrá la posición de la partícula en el instante de tiempo que le ha dado el usuario.

2.5.4 Fight Time

Esta función calcula el tiempo de vuelo, es decir, el tiempo que tarda una partícula lanzada desde una posición inicial en alcanzar su posición final y la misma altura de lanzamiento.

A esta función deberemos pasarle la velocidad inicial de la partícula y la gravedad (y el ángulo en un caso especial).

Esta función tiene dos variantes. Debido a que necesitamos la velocidad inicial en el eje Y la función puede: aceptar una velocidad de tipo fPoint con sus dos componentes de la cual cogemos la v.y directamente, o bien, le podemos pasar el módulo de la velocidad en tipo float directamente. En el caso de pasarle la velocidad como fPoint tan solo necesitaremos su componente vertical, en caso contrario la fórmula para calcular el tiempo será la misma, pero al tener simplemente el módulo de la velocidad necesitaremos su ángulo de lanzamiento para determinar su componente en Y.

Aislando el tiempo de las ecuaciones generales de tiro parabólico obtenemos la siguiente fórmula: *t = (2 * vy)/g*

Que si sustituimos estas variables con lo explicado anteriormente obtenemos que tenemos que pasar esto al programa.

*time = (2 * vi * sin(angle * PI / 180)) / gravity*

*time = (2 * vi.y) / gravity*

3. Ejemplos

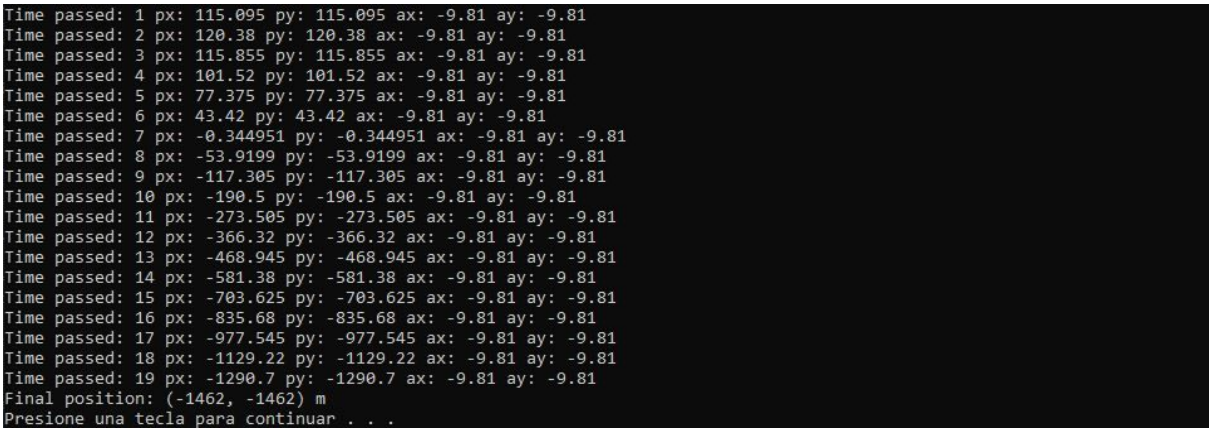
En este apartado vamos a mostrar algunos ejemplos de cómo se grafica la partícula en cada uno de los casos que podemos elegir en el menú inicial, el cual es rellenado por el usuario.

3.1 Calculate specific values

Valores introducidos

- Posición x e y: 100, 100 m
- Velocidad inicial x e y: 20, 20 m/s
- Aceleración x e y: 5, 5 m/s^2
- Tiempo: 20 s

Resultado

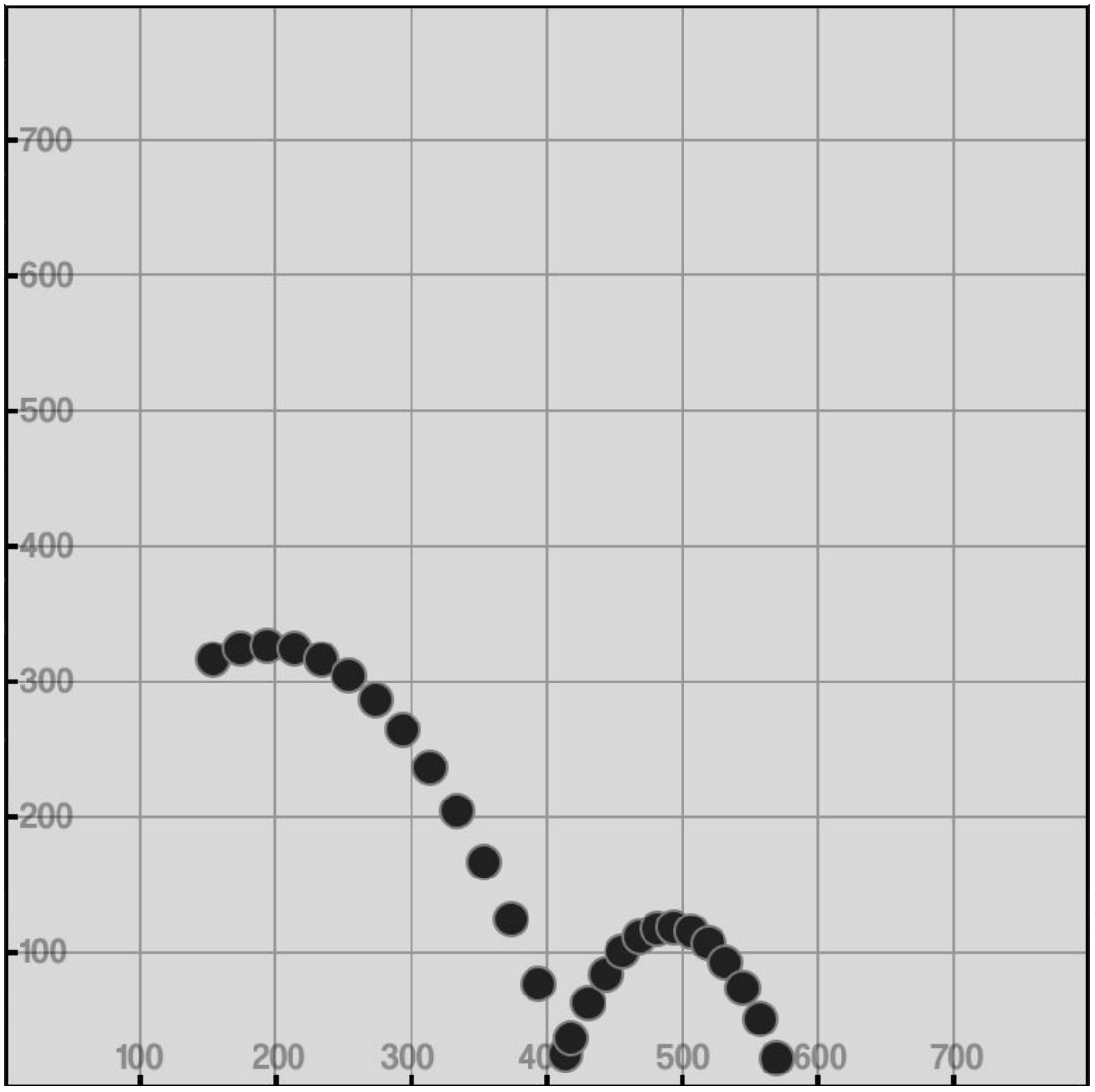


3.2 Do graphical presentation of the position

Valores introducidos

- Posición x e y: 100, 300 m
- Velocidad inicial x e y: 20, 20 m/s
- Gravedad: - 5 m/s^2
- Masa: 5 kg

Resultado



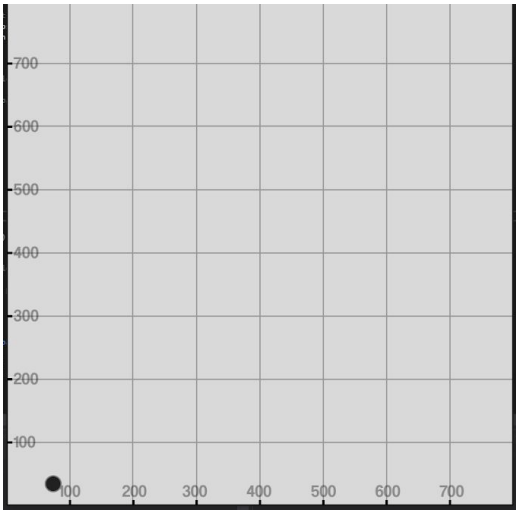
3.3 Simulate in real time

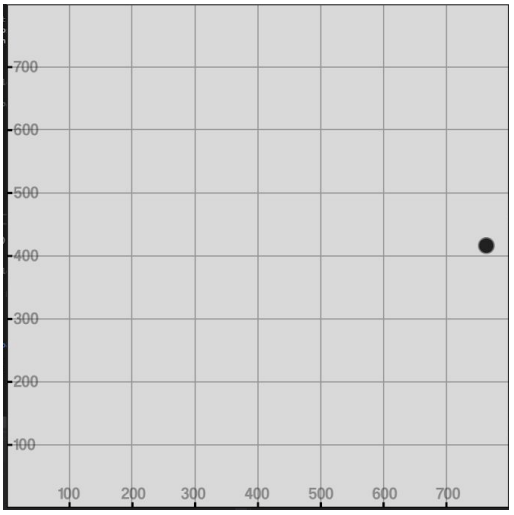
Valores introducidos

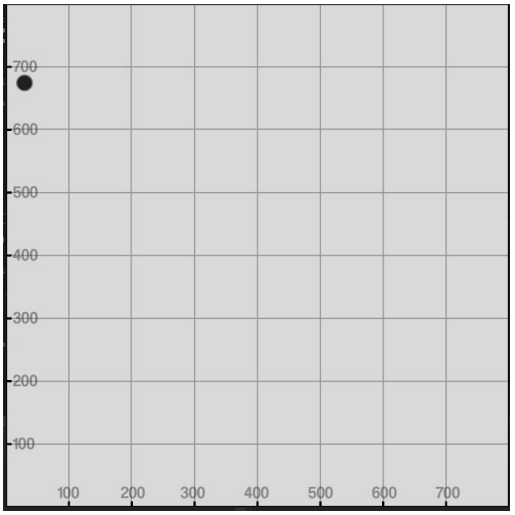
- Posición x e y: 10, 10 m
- Velocidad inicial x e y: 10, 10 m/s
- Gravedad: - 5 m/s^2
- Masa: 5 kg

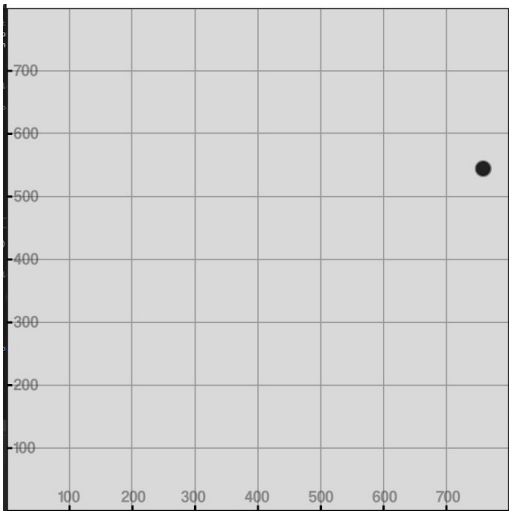
Resultado

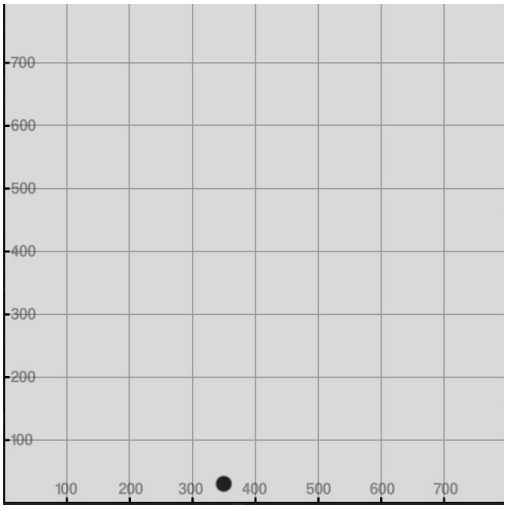
En este caso, para poder comprobar lo que realmente hace esta opción del menú es necesario ver la compilación del código en acción. Pero, para poder mostrarlo un poco en este documento, hemos introducido la posición inicial de la partícula en base a unos valores y su posición final.










700
600
500
400
300
200
100
100
200
300
400
500
600
700

4. Conclusiones

Una vez acabada esta tarea podemos estar orgullosos del trabajo realizado y los resultados obtenidos, estas son las ideas generales de los principales puntos que hemos conseguido implementar:

- En primer lugar, hemos implementado un menú en el cual hacemos que el usuario escoja diferentes opciones según lo que quiera hacer en ese momento. Así permitimos que este defina una serie de variables y, según su elección, confeccionamos una simulación que sea representativa y acorde con sus exigencias.
- Hemos sabido implementar con éxito las funciones básicas del Integrador de Verlet, ya que, a partir de una posición actual y una posición previa, podemos calcular las posiciones siguientes y, por lo tanto, la trayectoria que hará.
- A partir de las ecuaciones generales de la posición, velocidad y aceleración hemos conseguido implementar funciones que nos calculen ciertas variables como el tiempo total de vuelo, calcular el tiempo que tarda en llegar a cierta posición, o prever la posición anterior o posterior de una partícula sabiendo otras variables.
- También hemos incluido unas funciones que calculan la aceleración contando con rozamientos con el aire o funciones que determinan la velocidad terminal de un objeto en caída libre.
- Hemos añadido un sistema de colisiones en el cual podemos calcular si la partícula ha colisionado con el suelo, paredes o techo (que son los límites de la ventana) y a partir de ahí saber su velocidad, su aceleración y el tiempo total de colisión.
- Aunque simples las ecuaciones de integración de Verlet i todas las que hemos podido desarrollar a partir de estas han tenido un nivel de precisión considerable y a nivel de rendimiento han ofrecido un buen resultado.
- En último lugar, pero no por ello menos importante, hemos implementado un sistema gráfico (a parte de la consola de Windows) en el que se representan todos los cambios que experimenta esta partícula de una forma más visual y más fácil de entender para todo el mundo.

El código está abierto y visible para todo el mundo en el repositorio de GitHub del siguiente enlace:

https://github.com/Team-Cell/Verlet-Integrator

Además del código en la pestaña releases se puede descargar un .zip con una versión en que se puede ejecutar directamente el programa.