Progettazione e sviluppo di un simulatore del Sistema Solare

Mattia Gregnanin

21 maggio 2023

Indice

1 Introduzione e obiettivi

Lo scopo di questo documento è quello d'illustrare la progettazione di un motore fisico e l'implementazione di un motore grafico per un simulatore del Sistema Solare con la caratteristica di essere *dinamico*, ovverosia capace di studiare l'evoluzione di un qualsiasi numero di pianeti e oggetti celesti.

Ad esempio, si potrebbe studiare il comportamento di un satellite attorno alla Terra con la sola Terra in gioco, oppure considerando anche gli effetti dovuti alla presenza della Luna e del Sole. Tutti i corpi possono essere caricati tramite un file JSON che ne specifica posizione e velocità iniziali e, inoltre, eventualmente dotati di una rotazione propria.

Si noti però che tutti i corpi sono rappresentati in forma perfettamente sferica, con l'intera massa concentrata al centro, e si trascurano effetti dovuti al fatto che i corpi rotanti sono schiacciati ai poli o altri fenomeni di perturbazione.

1.1 Linguaggi e librerie utilizzati

Per questo progetto è stato utilizzato il linguaggio di programmazione `TypeScript`, derivato dal più noto `JavaScript` e che introduce una maggiore tipizzazione[3]. Ciò ha permesso

al simulatore di essere una "web-app" e di poter essere eseguito da qualsiasi dispositivo dotato di un browser che esegue codice `JavaScript`, semplicemente visitando la pagina `https://mattiagre.github.io/public`.

Come motore grafico è stata utilizzata la libreria open source `Three.js`, disponibile al sito `https://threejs.org` e che mette già a disposizione strumenti matematici utili come le classi `Vector3`, `Matrix3`, `Quaternion`, ecc....

Infine, per l'interfaccia grafica viene utilizzata la libreria `dat.GUI`, resa pubblica all'indirizzo `https://github.com/dataarts/dat.gui`.

2 Descrizione del motore fisico

2.1 Problema degli N corpi

L'obiettivo principale del motore fisico è quello di risolvere numericamente il cosiddetto "problema degli N corpi". Esso è costituito[5] da un numero N qualsiasi di punti materiali che si attraggono con una forza centrale data dall'interazione gravitazionale pari a

$$\mathbf{F}_{ji} = -\frac{Gm_i m_j}{r_{ij}^3}\mathbf{r}_{ij}, \tag{1}$$

dove $\mathbf{F}_{ji}$ è la forza che il corpo i esercita sul corpo j, mentre $\mathbf{r}_{ij}$ è il vettore che congiunge i a j, ovvero $\mathbf{r}_j - \mathbf{r}_i$, e r_{ij} il suo modulo.

Trascurando qualsiasi altra forza (ad esempio, la pressione di radiazione dovuta al Sole), il generico corpo m_k è soggetto a una forza risultante pari a

$$\mathbf{R}_k = \sum_{\substack{i=1 \\ i\neq k}}^{N} \mathbf{F}_{ki} = \sum_{\substack{i=1 \\ i\neq k}}^{N} -\frac{Gm_i m_k}{r_{ik}^2}\hat{\mathbf{r}}_{ik}, \quad k = 1, 2, \dots, N. \tag{2}$$

Il motore fisico deve quindi risolvere le N equazioni

$$m_k \ddot{\mathbf{r}}_k = \mathbf{R}_k, \quad k = 1, 2, \dots, N,$$

ovvero, esplicitando le accelerazioni

$$\ddot{\mathbf{r}}_k = \frac{\mathbf{R}_k}{m_k}, \quad k = 1, 2, \dots, N. \tag{3}$$

2.2 Calcolo della forza risultante

L'algoritmo che calcola le N accelerazioni e, quindi, le N forze risultanti, utilizza il semplice metodo esaustivo che ha complessità $O(N^2)$. Esso è implementato nella seguente maniera:

```
static applyGravity(bodies: Body[]) {
  for (let i = 0; i < bodies.length; i++) {
    for (let j = i + 1; j < bodies.length; j++) {
      const displacement = bodies[i].position.clone().sub(bodies[j].position);
      if (displacement.lengthSq() === 0)
        console.error('Two bodies have the same position. Cannot compute the
        ↪ force between them.');
      const force = displacement.divideScalar(Math.pow(displacement.lengthSq(),
       ↪ 1.5)).multiplyScalar(- AstroSystem.G_COSTANT * bodies[i].mass *
       ↪ bodies[j].mass);
      bodies[i].applyForce(force);
      bodies[j].applyForce(force.negate());
    }
  }
}
```

La terza riga è composta dall'istruzione `let j = i + 1` invece che da `let j = 0`. Questo serve a sfruttare la relazione $\mathbf{F}_{ij} = -\mathbf{F}_{ji}$ data dalla terza legge della dinamica e a dimezzare le operazioni necessarie.

Si noti inoltre che alla riga cinque è presente una condizione che verifica che due corpi diversi non abbiano la stessa posizione. Ciò significherebbe che i due corpi si sono collisi e ciò non viene risolto da questo motore fisico, che invece mostra a console un errore.

2.3 Algoritmi d'integrazione

Compito del risolutore è quella di trovare le posizioni dei corpi dopo un piccolo tempo Δt (che in questa simulazione *non* è costante) a partire dall'eq. (3). Ciò non è possibile farlo analiticamente in quanto nell'equazione differenziale

$$\frac{d^2\mathbf{r}_k}{dt^2} = \frac{\mathbf{R}_k}{m_k} = -\sum_{\substack{i=1,\\ i\neq k}}^{N} \frac{Gm_i}{r_{ik}^3}\mathbf{r}_{ik} = -\sum_{\substack{i=1,\\ i\neq k}}^{N} \frac{Gm_i}{\|\mathbf{r}_k - \mathbf{r}_i\|^3}(\mathbf{r}_k - \mathbf{r}_i), \quad k = 1, 2, \dots, N$$

il termine a destra dipende dalle posizioni istantanee dei corpi e non può essere integrato semplicemente.

Nascono allora vari algoritmi che permettono di approssimare le posizioni con un sufficiente grado di precisione. Nel simulatore sono implementati i tre che seguono, ma è facilmente possibile aggiungerne altri.

2.3.1 Algoritmo di Störmer-Verlet

L'algoritmo di Verlet[1] permette calcolare la posizione $\mathbf{r}(t + \Delta t)$ di un corpo a partire da $\mathbf{r}(t)$ e $\mathbf{r}(t - \Delta t)$ con un errore locale pari a $O(\Delta t^4)$. Esso si basa sull'approssimazione di $\mathbf{r}(t + \Delta t)$ e $\mathbf{r}(t - \Delta t)$ attraverso un'espansione in serie di Taylor come:

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2 + \frac{1}{6}\mathbf{b}(t)\Delta t^3 + \mathbf{O}(\Delta t^4) \tag{4}$$

$$\mathbf{r}(t - \Delta t) = \mathbf{r}(t) - \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2 - \frac{1}{6}\mathbf{b}(t)\Delta t^3 + \mathbf{O}(\Delta t^4) \tag{5}$$

da cui, sommando, si ottiene

$$\mathbf{r}(t + \Delta t) = 2\mathbf{r}(t) - \mathbf{r}(t - \Delta t) + \mathbf{a}(t)\Delta t^2 + O(\Delta t^4).$$

Algoritmo di Störmer-Verlet

Detti $\mathbf{r}^{(0)} = \mathbf{r}(t_0)$, $\mathbf{r}^{(1)} = \mathbf{r}(t_0 + \Delta t)$, $\mathbf{r}^{(2)} = \mathbf{r}(t_0 + 2\Delta t)$, $\dots$, $\mathbf{r}^{(k)} = \mathbf{r}(t_0 + k\Delta t)$ e denotata l'accelerazione con $\mathbf{a}^{(k)} = \mathbf{a}(\mathbf{r}(t_0 + k\Delta t))$, il metodo di Störmer-Verlet si riduce a calcolare

$$\begin{cases} \mathbf{r}^{(1)} = \mathbf{r}^{(0)} + \mathbf{v}^{(0)}\Delta t + \frac{1}{2}\mathbf{a}^{(0)}\Delta t^2 \\ \mathbf{r}^{(k+1)} = 2\mathbf{r}^{(k)} - \mathbf{r}^{(k-1)} + \mathbf{a}^{(k)}\Delta t^2 \end{cases} \tag{6}$$

Il metodo di Störmer-Verlet è di secondo ordine. Questo perché, nonostante l'errore locale sia di quarto grado, quello globale è di secondo. Noti esattamente $\mathbf{r}^{(0)}$ e $\mathbf{r}^{(1)}$, ovvero $\delta\mathbf{r}^{(0)} = \delta\mathbf{r}^1 = \mathbf{0}$, si può dimostrare per induzione che

$$\begin{aligned} \delta\mathbf{r}^{(2)} &= \mathbf{O}(\Delta t^4) \\ \delta\mathbf{r}^{(3)} &= 2\delta\mathbf{r}^{(2)} - \delta\mathbf{r}^{(1)} + \delta\mathbf{a}^{(2)}\Delta t^2 + \mathbf{O}(\Delta t^4) = 3\mathbf{O}(\Delta t^4) \\ \delta\mathbf{r}^{(4)} &= 2\delta\mathbf{r}^{(3)} - \delta\mathbf{r}^{(2)} + \delta\mathbf{a}^{(3)}\Delta t^2 + \mathbf{O}(\Delta t^4) = 6\mathbf{O}(\Delta t^4) \\ &\vdots \\ \delta\mathbf{r}^{(k+1)} &= 2\delta\mathbf{r}^{(k)} - \delta\mathbf{r}^{(k-1)} + \delta\mathbf{a}^{(k)}\Delta t^2 + \mathbf{O}(\Delta t^4) = \frac{k(k+1)}{2}\mathbf{O}(\Delta t^4) \end{aligned}$$

L'errore globale dopo un periodo $T = n\Delta t$ è dunque

$$\delta \mathbf{r}^{(n)} = \frac{n^2 - n}{2}\mathbf{O}(\Delta t^4) = \left(\frac{T^2}{2\Delta t^2} - \frac{T}{2\Delta t}\right)\mathbf{O}(\Delta t^4) = \mathbf{O}(\Delta t^2),$$

ovvero è un integratore di secondo ordine, come già affermato.

2.3.2 Algoritmo di Störmer-Verlet modificato

Il passaggio effettuato dalle eq. (4) e (5) per cancellare i termini con esponente dispari necessita che l'intervallo temporale Δt rimanga costante nel tempo. Nella simulazione esso però varia in funzione del *frame rate*, che può dipendere da vari fattori[1]

Inoltre per ciascun corpo si tiene in memoria solo la posizione e velocità correnti, perciò sarebbe preferibile utilizzare la velocità $\mathbf{v}^{(k)}$ invece della posizione precedente $\mathbf{r}^{(k-1)}$.

Si utilizza allora una versione differente dell'algoritmo, che si ottiene dalle due espansioni in serie di Taylor:

$$\begin{aligned}\mathbf{r}^{(k+1)} &= \mathbf{r}^{(k)} + \mathbf{v}^{(k)}\Delta t^{(k)} + \frac{1}{2}\mathbf{a}^{(k)}\Delta t^{(k)^2} + \mathbf{O}(\Delta t^{(k)^3}) \\ \mathbf{r}^{(k-1)} &= \mathbf{r}^{(k)} - \mathbf{v}^{(k)}\Delta t^{(k-1)} + \frac{1}{2}\mathbf{a}^{(k)}\Delta t^{(k-1)^2} + \mathbf{O}(\Delta t^{(k-1)^3})\end{aligned}$$

da cui, moltiplicando la prima per Δt_{k-1} e la seconda per Δt_k e sommando si ottiene

$$\mathbf{r}^{(k+1)}\Delta t^{(k-1)} \approx (\Delta t^{(k-1)} + \Delta t^{(k)})\mathbf{r}_k - \mathbf{r}^{(k-1)}\Delta t^{(k)} + \frac{1}{2}\Delta t^{(k-1)}\Delta t^{(k)}(\Delta t^{(k-1)} + \Delta t^{(k)})\mathbf{a}^{(k)},$$

ovvero, dividendo per Δt_{k-1}

$$\mathbf{r}^{(k+1)} \approx \mathbf{r}^{(k)} + \frac{\mathbf{r}^{(k)} - \mathbf{r}^{(k-1)}}{\Delta t^{(k-1)}}\Delta t^{(k)} + \mathbf{a}^{(k)}\frac{\Delta t^{(k-1)} + \Delta t^{(k)}}{2}\Delta t^{(k)}.$$

Si può infine denotare il rapporto $\dfrac{\mathbf{r}^{(k)} - \mathbf{r}^{(k-1)}}{\Delta t^{(k-1)}}$ come $\mathbf{v}^{(k)}$, notando che si tratta della sua approssimazione al primo ordine.

Algoritmo di Störmer-Verlet modificato

Posto l'istante iniziale $t^{(0)}$ e denotati con $t^{(1)} = t^{(1)} + \Delta t^{(0)}, t^{(2)} = t^{(1)} + \Delta t^{(1)}, \ldots, t^{(k+1)} = t^{(k)} + \Delta t^{(k)}$ gli istanti di tempo, si ha

$$\begin{cases}\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} + \mathbf{v}^{(k)}\Delta t^{(k)} + \mathbf{a}^{(k)}\dfrac{\Delta t^{(k-1)} + \Delta t^{(k)}}{2}\Delta t^{(k)} \\ \mathbf{v}^{(k+1)} = \dfrac{\mathbf{r}^{(k+1)} - \mathbf{r}^{(k)}}{\Delta t^{(k)}}\end{cases} \tag{7}$$

Esso viene implementato nel file `solver.ts` come

```
class Verlet implements Integrator {
  /**
   * Previous time interval.
   */
  #prevDt: number;

  constructor(private gravityApplier: GravityApplierCallback) { }
```

[1]La web app chiama il metodo `window.requestAnimationFrame()` per aggiornare sia la fisica che la scena. Il tempo tra una chiamata e l'altra dipende, in generale, dal *refresh rate* del dispositivo, ma anche dalle impostazioni di risparmio energetico[2].

```
  updatePositions(bodies: Body[], dt: number): void {
    // Initialize the previous time interval
    if (this.#prevDt === undefined)
      this.#prevDt = dt;

    // Apply gravity
    this.gravityApplier(bodies);

    bodies.forEach(body => {
      // Copy the body previous position
      const prevPosition = body.position.clone();
      // Compute the new position
      body.position.add(body.velocity.clone().multiplyScalar(dt))
        .add(body.acceleration.multiplyScalar(dt * (dt + this.#prevDt) / 2));
      // Update the velocity
      body.velocity.copy(prevPosition.sub(body.position).divideScalar(-dt));
      // Set the acceleration back to zero
      body.acceleration.set(0, 0, 0);
    });

    // Update the previous time interval
    this.#prevDt = dt;
  }
}

/**
 * Uses the Stormer-Verlet method, with a local truncation error of O(dt^4),
  ↪  but global O(dt^2) error. Supports changing time interval.
 */
export const VERLET_INTEGRATOR: Integrator = new Verlet(Solver.applyGravity);
```

2.3.3 Algoritmo Leapfrog

L'algoritmo Leapfrog[4], chiamato così perché calcola posizione e velocità a intervalli separati con una struttura che riprende il gioco della cavallina, *leapfrog* appunto, è un metodo d'integrazione di secondo ordine con la caratteristica di mantenere costanti certe proprietà del sistema fisico, come l'energia. È quindi possibile tornare anche indietro nel tempo senza alterare il sistema.

Algoritmo Leapfrog (kick-drift-kick)

Detta $\mathbf{v}^{(k+1/2)}$ la velocità nel passaggio intermedio e, per semplicità, $\Delta t = \Delta t^{(k)}$, si determinano $\mathbf{r}^{(k+1)}$ e $\mathbf{v}^{(k+1)}$ come

$$\begin{cases} \mathbf{v}^{(k+1/2)} = \mathbf{v}^{(k)} + \mathbf{a}^{(k)} \dfrac{\Delta t}{2} \\ \mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} + \mathbf{v}^{(k+1/2)} \Delta t \\ \mathbf{v}^{(k+1)} = \mathbf{v}^{(k)} + \mathbf{a}^{(k+1)} \dfrac{\Delta t}{2} \end{cases} \tag{8}$$

Lo svantaggio rispetto al metodo di Störmer-Verlet è che richiede la valutazione dell'accelerazione una volta in più, anche se può essere evitata all'inizio. Per questa simulazione, potendo cambiare l'integratore al volo, tale stratagemma non viene implementato:

```
class Leapfrog implements Integrator {
  constructor(private gravityApplier: GravityApplierCallback) { }
```

```
3
4     updatePositions(bodies: Body[], dt: number): void {
5       // Apply gravity
6       this.gravityApplier(bodies);
7
8       // Compute v_{i + 1/2} and r_{i + 1}
9       bodies.forEach(body => {
10        body.velocity.add(body.acceleration.multiplyScalar(dt / 2));
11        body.position.add(body.velocity.clone().multiplyScalar(dt));
12        // Set the acceleration to zero
13        body.acceleration.set(0, 0, 0);
14      });
15
16      // Apply gravity for the second time
17      this.gravityApplier(bodies);
18
19      // Compute v_{i + 1}
20      bodies.forEach(body => {
21        body.velocity.add(body.acceleration.multiplyScalar(dt / 2));
22        // Set the acceleration to zero
23        body.acceleration.set(0, 0, 0);
24      });
25    }
26  }
27
28  /**
29   * Uses the Leapfrog method, with a global O(dt^2) error. Its symplectic nature
       ↪  keeps the mechanical energy constant.
30   */
31  export const LEAPFROG_INTEGRATOR: Integrator = new
     ↪  Leapfrog(Solver.applyGravity);
```

2.3.4 Algoritmo di Yoshida

Il professore Haruo Yoshida, nell'articolo "Construction of higher order symplectic integrators" del 1990[6], riuscì ad ottenere una classe d'integratori simplettici di ordine maggiore di due. Quello più utilizzato è di quarto ordine, detto comunemente *algoritmo di Yoshida*:

Algoritmo di Yoshida (di quarto ordine)

Dette $\mathbf{r}_1^{(k)}, \mathbf{r}_2^{(k)}, \ldots$, le posizioni intedermedie alla k-esima iterazione e, in ugual modo, con $\mathbf{v}_1^{(k)}, \mathbf{v}_2^{(k)}, \ldots$ le velocità intermedie, si ha

$$\begin{cases} \mathbf{r}_1^{(k)} = \mathbf{r}^{(k)} + c_1 \mathbf{v}^{(k)} \Delta t \\ \mathbf{v}_1^{(k)} = \mathbf{v}^{(k)} + d_1 \mathbf{a}(\mathbf{x}_1^{(k)}) \Delta t \\ \mathbf{r}_2^{(k)} = \mathbf{r}_1^{(k)} + c_2 \mathbf{v}_1^{(k)} \Delta t \\ \mathbf{v}_2^{(k)} = \mathbf{v}_1^{(k)} + d_2 \mathbf{a}(\mathbf{x}_2^{(k)}) \Delta t \\ \mathbf{r}_3^{(k)} = \mathbf{r}_2^{(k)} + c_3 \mathbf{v}_2^{(k)} \Delta t \\ \mathbf{v}_3^{(k)} = \mathbf{v}_2^{(k)} + d_3 \mathbf{a}(\mathbf{x}_3^{(k)}) \Delta t \\ \mathbf{r}^{(k+1)} \equiv \mathbf{r}_4^{(k)} = \mathbf{r}_3^{(k)} + c_4 \mathbf{v}_3^{(k)} \Delta t \\ \mathbf{v}^{(k+1)} \equiv \mathbf{v}_4^{(k)} = \mathbf{v}_3^{(k)} \end{cases} \tag{9}$$

dove $c_1, c_2, c_3, c_4, d_1, d_2, d_3$ sono dei coefficienti il cui valore è dato da

$$c_1 = c_4 \equiv \frac{x_1}{2}, \quad c_2 = c_3 \equiv \frac{x_0 + x_1}{2}$$
$$d_1 = d_3 \equiv x_1, \quad d_2 \equiv x_0$$

con

$$x_0 \equiv -\frac{\sqrt[3]{2}}{2 - \sqrt[3]{2}} \quad x_1 \equiv \frac{1}{2 - \sqrt[3]{2}}.$$

Si nota immediatamente che l'algoritmo richiede la computazione dell'accelerazione tre volte a ogni iterazione. Ciò potrebbe risultare abbastanza lento nel caso in cui l'accelerazione venga calcolata con il metodo esaustivo e N sia molto grande.

Esso è implementato come segue.

```
class Yoshida implements Integrator {

  static readonly X0 = -Math.cbrt(2) / (2 - Math.cbrt(2));
  static readonly X1 = 1 / (2 - Math.cbrt(2));
  static readonly C1 = this.X1 / 2;
  static readonly C2 = (this.X0 + this.X1) / 2;
  static readonly C3 = this.C2;
  static readonly C4 = this.C1;
  static readonly D1 = this.X1;
  static readonly D2 = this.X0;
  static readonly D3 = this.X1;

  constructor(private gravityApplier: GravityApplierCallback) { }

  updatePositions(bodies: Body[], dt: number): void {
    // First iteration
    bodies.forEach(body => {
      body.position.add(body.velocity.clone().multiplyScalar(Yoshida.C1 * dt));
    });
    this.gravityApplier(bodies);
    // Second iteration
    bodies.forEach(body => {
      body.velocity.add(body.acceleration.multiplyScalar(Yoshida.D1 * dt));
      body.acceleration.set(0, 0, 0);
      body.position.add(body.velocity.clone().multiplyScalar(Yoshida.C2 * dt));
    });
    this.gravityApplier(bodies);
    // Third iteration
    bodies.forEach(body => {
      body.velocity.add(body.acceleration.multiplyScalar(Yoshida.D2 * dt));
      body.acceleration.set(0, 0, 0);
      body.position.add(body.velocity.clone().multiplyScalar(Yoshida.C3 * dt));
    });
    this.gravityApplier(bodies);
    // Fourth iteration
    bodies.forEach(body => {
      body.velocity.add(body.acceleration.multiplyScalar(Yoshida.D3 * dt));
      body.acceleration.set(0, 0, 0);
      body.position.add(body.velocity.clone().multiplyScalar(Yoshida.C4 * dt));
    });
```

```
  }
}

/**
 * Uses the fourth order simplectic integrator by Prof. Haruo Yoshida.
 */
export const YOSHIDA_INTEGRATOR: Integrator = new Yoshida(Solver.applyGravity);
```

2.4 Unità di misura e costanti utilizzate

Per quanto concerne le unità di misura che vengono utilizzate durante i calcoli, si è deciso di optare per il sistema astronomico che misura le distanze in au, i tempi in giorni e le masse in relazione alla massa della Terra. Questo viene fatto per normalizzare le distanze a valori molto vicini all'unità che, con i chilometri, non sarebbe stato possibile.

Le costanti utilizzate nella simulazione sono definite nel file `astro-system.ts` come segue

```
/**
 * Defines constants used by the astronomical system of units.
 */
export module AstroSystem {
  /**
   * Meters in an astronomical unit.
   */
  export const AU = 149597870707;
  /**
   * Seconds in a day.
   */
  export const DAY = 86400;
  /**
   * Seconds in a sidereal day.
   */
  export const SIDEREAL_DAY = 86164.0905;
  /**
   * The mass of the Earth (in kg) as revised May 9, 2022, by NASA JPL
    ↪ (https://ssd.jpl.nasa.gov).
   */
  export const EARTH_MASS = 5.97219e24;
  /**
   * The universal gravitational interaction constant in the astronomical
    ↪ system of units.
   */
  export const G_COSTANT = 6.67428e-11 * EARTH_MASS * DAY ** 2 / AU ** 3;
}
```

3 Descrizione e uploading degli oggetti celesti

Come già affermato, nel sistema non sono codificati direttamente i pianeti o altri corpi, ma vengono invece caricati da un file JSON che può essere liberamente modificato.

3.1 La classe Body e le sue sottoclassi

Tutti i corpi celesti sono rappresentati nella simulazione come oggetti della classe `Body`. Essa permette di maneggiare proprietà intrinseche di essi, come massa, posizione, velocità, accelerazione e rotazione propria. Quest'ultima è identificata come un oggetto della classe `BodyRotation`, codificata come segue.

```ts
/**
 * Contains useful informations about the rotation of a body.
 */
export class BodyRotation {
  /**
   * The angle (in degrees) that the rotation axis makes with the perpendicular to the ecliptic.
   */
  readonly obliquity: number;
  /**
   * The rotational period, in seconds.
   */
  readonly period: number;
  /**
   * The axis of rotation. Its length is guaranteed to be one.
   */
  readonly axis: THREE.Vector3;

  constructor(obliquity: number, period: number) {
    this.obliquity = obliquity;
    this.period = period;
    this.axis = new THREE.Vector3(Math.sin(obliquity * Math.PI / 180), Math.cos(obliquity * Math.PI / 180), 0);
  }
}
```

Essa contiene quindi tre proprietà: l'inclinazione (rispetto alla perpendicolare all'eclittica), il periodo e infine l'asse di rotazione normalizzato. Nella classe `Body` il metodo `rotate()`, chiamato a ogni nuovo rendering della scena, utilizza l'asse per ruotare il corpo attorno per l'intervallo di tempo specificato.

3.2 Serializzazione dei corpi celesti

Il metodo `loadBodiesFromJSON()` della classe `SolarSystem` carica nella scena i corpi contenuti nel file `bodies.json` come oggetti della classe `Body`. Ad esempio, la struttura dell'oggetto che rappresenta la Terra è

```json
{
  "name": "Earth",
  "mass": 5.97219E24,
  "radius": 6371,
  "texture": "earth.jpeg",
  "orbit": {
    "color": "0x228B22",
    "thickness": 3
  },
  "position": {
    "x": -1.771478822733411E-01,
    "y": 9.672393401210591E-01,
    "z": -4.085203551709195E-06
  },
  "velocity": {
    "x": -1.720758492418282E-02,
    "y": -3.159006535002814E-03,
    "z": 1.050908513009135E-07
```

```
  },
  "rotation": {
    "obliquity": 23.4392911,
    "period": 86164.0905
  }
}
```

La massa è specificata in kg, così come il raggio in km e il periodo di rotazione in secondi. Verranno poi convertiti nelle unità di misura utilizzate dal simulatore da `loadBodiesFromJSON()`.

Le posizioni e le velocità sono invece caricate, rispettivamente, in au e au/d. Questo perché tali informazioni sono direttamente disponibili dall'applicativo "Horizons System" di JPL, disponibile all'indirizzo `https://ssd.jpl.nasa.gov/horizons/app.html#/`.

Si noti infine che le dimensioni dei corpi non risultano in scala $1:1$ con le distanze, ma in scala $30\,000:1$. Questo perché altrimenti essi non sarebbero distinguibili nello spazio.

3.3 Orbite dei corpi in movimento

Particolare attenzione va posta al problema della rappresentazione delle orbite, ovvero della traiettoria percorsa da ciascun corpo in movimento nello spazio. Poiché gli oggetti non si muovono su percorsi predefiniti (ad esempio, si pensi a un semplice sistema solare che utilizza i parametri orbitali per determinare la posizione sull'orbita ellittica dato il tempo), non è possibile prevedere la posizione di esso nel futuro senza calcolarla.

L'orbita è quindi raffigurata da una linea che passa per gli ultimi punti attraverso cui il corpo è passato. È previsto un numero massimo di punti, in modo da evitare errori di allocazione di memoria e conseguentemente `SEGFAULT`. Quando il numero di punti viene superato, vengono eliminati i punti più vecchi e aggiunti quelli nuovi alla fine dell'array.

Il codice che aggiorna le orbite è dato dal metodo `updateOrbit()` della classe `Body`.

4 Esempi di simulazione

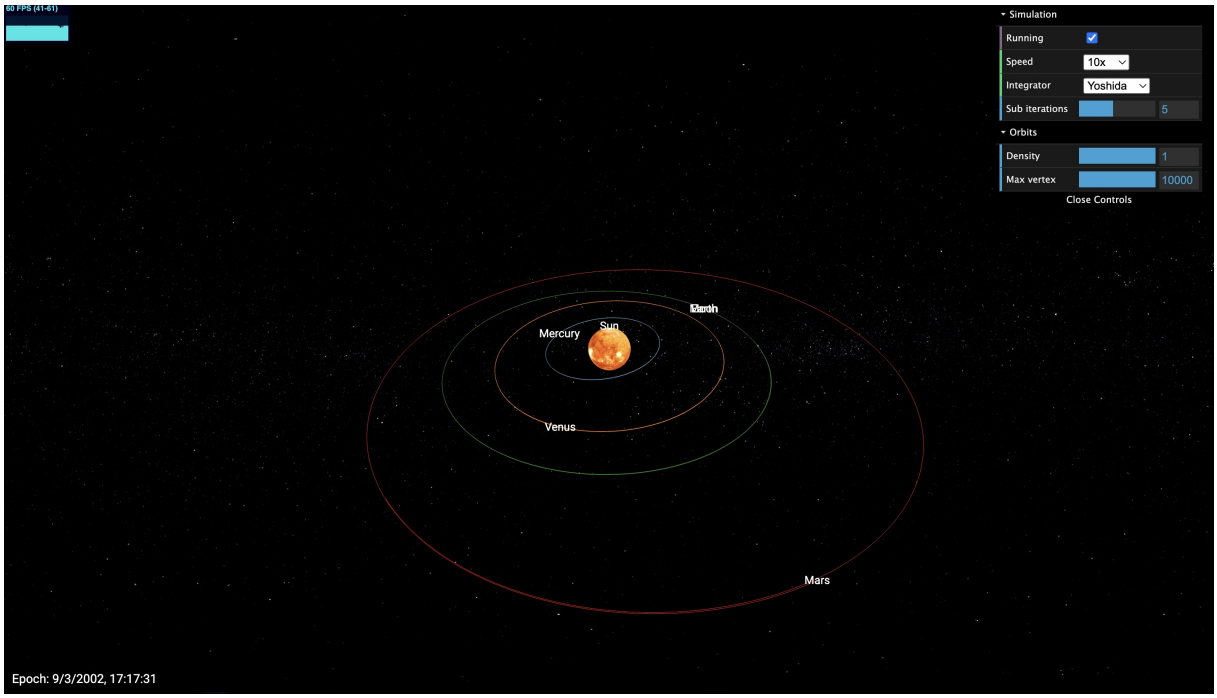

Figura 1: Simulazione dei pianeti interni e della Luna.

5 Sviluppi futuri

Il codice postato in questo documento è soggetto a revisione costante e potrebbe cambiare da un momento all'altro, per risolvere bug o aggiungere funzionalità.

Tra le modifiche più rilevanti che nel futuro verranno implementate vi è l'algoritmo di Barnes-Hut, che permette di calcolare le accelerazioni dei corpi con una complessità pari a $O(N \log N)$, invece del corrente $O(N^2)$.

Risulterebbe anche interessante poter lanciare dei satelliti da un pianeta direttamente dalla simulazione, selezionando latitudine e longitudine del punto di lancio e poi il beta angle.

Infine, poiché la simulazione inizia sempre il 1 gennaio 2000, si può aggiungere la possibilità di cambiare data, sia avanti che indietro nel tempo, senza dover attendere che la simulazione la raggiunga.

Riferimenti bibliografici

[1] James Schloss. *Verlet Integration*. `https://www.algorithm-archive.org/contents/verlet_integration/verlet_integration.html`. Visitato il 20-05-2023.

[2] MDN Web Docs. *Window: requestAnimationFrame() method*. `https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame`. Visitato il 20-05-2023. 2023.

[3] Microsoft. *TypeScript is JavaScript with syntax for types*. `https://www.typescriptlang.org`. Visitato il 20-05-2023. 2023.

[4] Wikipedia contributors. *Leapfrog integration*. `https://en.wikipedia.org/wiki/Leapfrog_integration`. Visitato il 20-05-2023. 2023.

[5] Wikipedia contributors. *n-body problem*. `https://en.wikipedia.org/wiki/N-body_problem`. Visitato il 20-05-2023. 2023.

[6] Haruo Yoshida. «Construction of higher order symplectic integrators». In: *Physics Letters A* 150 (1990), pp. 262–268.