Lecture V: The game-engine loop & Time Integration
The Game-Engine Loop

1. Forces $\vec{F}(t)$
2. Integrate velocities and positions
   - Previous state: $(\vec{v}(t), \vec{\omega}(t)), (\vec{p}(t), q(t))$
   - Per-body change $(\vec{v}(t + \Delta t), \vec{\omega}(t + \Delta t)), (\vec{p}(t + \Delta t), q(t + \Delta t))$
3. Resolve Interpenetrations
4. Position correction $(\vec{v}(t + \Delta t), \vec{\omega}(t + \Delta t)), (\vec{p}(t + \Delta t), q'(t + \Delta t))$
5. Resolve Collisions
6. Velocity correction $(\vec{v}(t + \Delta t), \vec{\omega}(t + \Delta t)), (\vec{p}(t + \Delta t), q'(t + \Delta t))$
Challenges

- Kinematics: continuous motion in continuous time.
  - Events are local and always valid.
- Computer simulation:
  - Discrete time steps $\Delta t$.
  - Discrete Space (mesh, particles, grids)
Updating Position

- **Force** induces acceleration.

- When **mass** is constant:
  \[ F(p, t) = m \cdot a(p, t) \]

- Derivatives: \( v'(t) = a(t) \) and \( p'(t) = v(t) \)

- Thus: \( F(p, t) = m \cdot p''(t) \).

- A **differential equation**.
  - Often impossible to solve analytically.
  - More often, \( a = a(v, t) \) (velocity dependence).

- **Discretization**: **stability** and **convergence** issues.
Taylor Approximation

• A function at $t + \Delta t$ can be approximated by the function at $t$ with derivatives:

\[ p(t + \Delta t) \]
\[ \approx p(t) + \Delta t \cdot p'(t) + \frac{\Delta t^2}{2} p''(t) + \cdots + \frac{\Delta t^n}{n!} p^{(n)}(t) \]

• We do not usually use (or have) more than second derivatives.
First-Order Approximation

- If $\Delta t$ is small enough, we approximate linearly:

$$ p_o(t + \Delta t) \approx p_o(t) + \Delta t \cdot p_o'(t) $$

- Euler’s Method: approximating forward both velocity and position within the same step:

$$ v(t + \Delta t) = v(t) + a(t)\Delta t = v(t) + \frac{F(t)}{m}\Delta t $$

$$ p_o(t + \Delta t) = p_o(t) + v(t)\Delta t $$

Unknown for next time step

Assumed known for this time step
Euler’s Method

• This is known as Euler’s method

\[ v(t + \Delta t) = v(t) + a(t)\Delta t \]

\[ p_o(t + \Delta t) = p_o(t) + v(t)\Delta t \]
Euler’s Method

• **Note:** we approximate the velocity as **constant** between frames.
  
  • We compute the acceleration of the object from the net force applied on it:
    
    \[
    a(t) = \frac{F(t)}{m}
    \]
  
  • We compute the velocity from the acceleration:
    
    \[
    v(t + \Delta t) = v(t) + a(t)\Delta t
    \]
  
  • We compute the position from the velocity:
    
    \[
    p_o(t + \Delta t) = p_o(t) + v(t)\Delta t
    \]
Issues with Linear Dynamics

• A mere sequence of instants.
  • Without the precise instant of bouncing.

• Trajectories are piecewise-linear.
  • Constant velocity and acceleration in-between frames.
Time Step

• When $\Delta t \to 0$, we converge to $p(t) = \int_0^t v(s) ds$

• Is possible solution: reducing $\Delta t$ as much as possible?

• First-order method, piecewise-constant velocity: not very stable.

• Purpose: make the most with every $\Delta t$ you get.
Time Step

• **First-order assumption:** the slope at $t$ as a good estimate for the slope over the entire interval $\Delta t$.
• The approximation can **drift off** the function.
• Farther drifting $\Leftrightarrow$ tangent approximation worse.
Midpoint Method

• Estimating tangent in Half step:
  \[ v \left( t + \frac{\Delta t}{2} \right) = v(t) + a(t, v) \frac{\Delta t}{2} \]

• Full step:
  \[ v(t + \Delta t) = v(t) + a \left( t + \frac{\Delta t}{2}, v \left( t + \frac{\Delta t}{2} \right) \right) \Delta t \]

• Second-order approximation.
• Compute position similarly with \( v \).
Midpoint Method

• Approximating the tangent in mid-interval.
• Applying it to initial point across the entire interval.
• Error order: the square of the time step $O(\Delta t^2)$. Better than Euler’s method ($O(\Delta t)$) when $\Delta t < 1$.
• Approximating with a quadratic curve instead of a line.
• …can still drift off the function.
Improved Euler’s Method

• Considers the tangent lines to the solution curve at both ends of the interval.

• Velocity to the first point (Euler’s prediction): 
  \[ v_1 = v(t) + \Delta t \cdot a(t, v) \]

• Velocity to the second point (correction point): 
  \[ v_2 = v(t) + \Delta t \cdot a(t + \Delta t, v_1) \]

• Improved Euler’s velocity 
  \[ v(t + \Delta t) = \frac{v_1 + v_2}{2} \]

• Compute position similarly with \( v_1, v_2 \) instead of \( a \).
Improved Euler’s Method

The order of the error is $O(\Delta t^2)$. The final derivative is still inaccurate.
Runge-Kutta Method

- There are methods that provide better than quadratic error.

- The Runge-Kutta order-four method (RK4) is \( O(\Delta t^4) \).

- A combination of the midpoint and modified Euler’s methods, with higher weights to the midpoint tangents than to the endpoints tangents.
Computing the four following tangents (note dependence of acceleration on velocity):
\[ v_1 = \Delta t \cdot a(t, v(t)) \]
\[ v_2 = \Delta t \cdot a \left( t + \frac{\Delta t}{2}, v(t) + \frac{1}{2} v_1 \right) \]
\[ v_3 = \Delta t \cdot a \left( t + \frac{\Delta t}{2}, v(t) + \frac{1}{2} v_2 \right) \]
\[ v_4 = \Delta t \cdot a(t + \Delta t, v(t) + v_3) \]
Blend as follows:
\[ v(t + \Delta t) = v(t) + \frac{v_1 + 2v_2 + 2v_3 + v_4}{6} \]
Compute position similarly with \( v \) values.
\[ v_1 = \Delta t \cdot a(t, v(t)) \]
\[ v_2 = \Delta t \cdot a\left(t + \frac{\Delta t}{2}, v(t) + \frac{1}{2} v_1\right) \]
\[ v_3 = \Delta t \cdot a\left(t + \frac{\Delta t}{2}, v(t) + \frac{1}{2} v_2\right) \]
\[ v_4 = \Delta t \cdot a(t + \Delta t, v(t) + v_3) \]
Integrazione Leapfrog
(“a cavallina”)
Integrazione Leapfrog
Integrazione Leapfrog
first step

\[ a = f(p_0, ...) \]
\[ \vec{v}_{0.5} = \vec{v}_0 + \vec{a} \cdot dt / 2 \]
Integrazione Leapfrog

\[
\begin{align*}
p_1 &= p_0 + \vec{v}_{0.5} \cdot dt \\
p_2 &= p_1 + \vec{v}_{1.5} \cdot dt \\
p_3 &= p_2 + \vec{v}_{2.5} \cdot dt
\end{align*}
\]

\[
\begin{align*}
a &= f(p_1, \ldots) \\
a &= f(p_2, \ldots)
\end{align*}
\]

\[
\begin{align*}
\vec{v}_{1.5} &= \vec{v}_{0.5} + a \cdot dt \\
\vec{v}_{2.5} &= \vec{v}_{1.5} + a \cdot dt
\end{align*}
\]
Leapfrog Method

- More accurate than Euler-based methods
  - Residue of $O(\Delta t^3)$
- But at the same cost as Euler’s method!
- Major advantage: fully reversible!
Verlet integration

- Based on the Taylor expansion series of the previous time step and the next one:

\[
p_o(t + \Delta t) + p_o(t - \Delta t)
\approx p_o(t) + \Delta t \cdot p_o'(t) + \frac{\Delta t^2}{2} \cdot p_o''(t) + \ldots
\]

\[
+ p_o(t) - \Delta t \cdot p_o'(t) + \frac{\Delta t^2}{2} \cdot p_o''(t) - \ldots
\]

Cancels out!
Verlet integration

• Approximating without velocity:

\[ p_o(t + \Delta t) = 2p_o(t) - p_o(t - \Delta t) + \Delta t^2 p_o''(t) + O(\Delta t^4) \]
Verlet integration

- An $O(\Delta t^2)$ order of error.
- Very stable and fast without the need to estimate velocities.
- We need an estimation of the first $p_o(t - \Delta t)$
  - Usually obtained from one step of Euler’s or RK4 method.
- Difficult to manage velocity related forces such as drag or collision.
- Introduction to position-based dynamics.
Implicit methods

- So far: computing current position $p(t)$ and velocity $v(t)$ for the next position (forward).
  - Those are denoted as explicit methods.
- In implicit methods, we make use of the quantities from the next time step!
  $$p(t) = p(t + \Delta t) - \Delta t \cdot v(t + \Delta t)$$
  - This particular one: backward Euler.
- Computing in inverse:
- Finding position $p(t + \Delta t)$ which produces $p(t)$ if simulation is run backwards.
Implicit methods

• Not more accurate than explicit methods, but more stable.

• Especially for a damping of the position (e.g. drag force or kinetic friction).
Backward Euler

• How to compute the velocity from the future?

• Given the forces applied, extracting from the formula:
  • Example: a drag force $F_D = -b \cdot v$ is applied:
    $$\frac{v(t + \Delta t) - v(t)}{\Delta t} = -b \cdot v(t + \Delta t)$$
  • And therefore
    $$v(t + \Delta t) = \frac{v(t)}{1 + \Delta t \cdot b}$$
Backward Euler

- Often not knowing the forces in advance (likely case in a game).
- Or that the backward equation is not easy to solve.
- We use a predictor-corrector method:
  - one step of explicit Euler’s method
  - use the predicted position to calculate $v(t + \Delta t)$
- More accurate than explicit method but twice the amount of computation.
Semi-Implicit Method

- Combines the simplicity of explicit Euler and stability of implicit Euler.
- Runs an explicit Euler step for velocity and then an implicit Euler step for position:

\[ v(t + \Delta t) = v(t) + \Delta t \times a(t) = v(t) + \Delta t \times F(t)/m \]

\[ p(t + \Delta t) = p(t) + \Delta t \times v(t) = p(t) + \Delta t \times v(t + \Delta t) \]
Semi-Implicit Method

• The position update in the second step uses the next velocity in the implicit scheme.
  • Good for position-dependent forces.
  • Conserves energy over time, and thus stable.

• Not as accurate as RK4 (order of error is still $O(\Delta t)$), but cheaper and yet stable.

• Very popular choice for game physics engine.
Summary

• First-order methods
  • Implicit and Explicit Euler method, Semi-implicit Euler, Exponential Euler

• Second-order methods
  • Verlet integration, Velocity Verlet, Trapezoidal rule, Beeman’s algorithm, Midpoint method, Improved Euler’s method, Heun’s method, Newmark-beta method, Leapfrog integration.

• Higher-order methods
  • Runge-Kutta family methods, Linear multistep method.

• Position-based methods
  • Leapfrog, Verlet.
Concluding remarks

• Dimension
  • Integration methods shown for 1D variables.
  • Generalization using vector-based structures (e.g., Jacobians).

• Rotational motion
  • The integration methods work exactly the same for angular displacement $\theta$, velocity $\omega$ and acceleration $\alpha$.

• Evaluation of all dimensions and variables should be done for the same simulation time $\Delta t$. 