CS425: Game Programming 1
Lecture 10: Particle System
October 10, 2018

Lecturer: Jyh-Ming Lien

1 Introduction

1.1 Physics Engine

Physics engine is usually used in game to simulate the motion of objects, such as particles. The following simple procedule is the core of a physics engine (using a particle system as an example).

1. Get states of particles (positions and velocities)
2. Get forces F applied to each particles
3. Compute derivatives from the forces F (**last lecture**)
4. **Using ODE solvers to compute the new positions and velocities** (today)
5. Set the states back to the particles

2 Initial value problem

Remember that a state of a particle is $s = \begin{bmatrix} x \\ \dot{x} \end{bmatrix} = \begin{bmatrix} x \\ v \end{bmatrix}$, where x is the position and v is the velocity. Given an initial condition, ex: starting position and velocity, our goal is to find a sequence of positions and velocities in the future.

This is usually stated as an ODE: $\dot{x} = f(x, t)$, where $\dot{x}$ is the derivative of x. Note both x and $\dot{x}$ can be either scalars or vectors. Standard differential equations courses concerns about finding the solution x symbolically. For example, given $\dot{x} = -kx$, the answer is $x = e^{-kt}$.

In our physics engine, we have unknown variables as $s = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$, therefore, our ODE looks like this: $\dot{s} = f(s, t) = f(\begin{bmatrix} x \\ v \end{bmatrix}, t) = \begin{bmatrix} v \\ \frac{F}{m} \end{bmatrix}$, where F is the combined force acting on the given particle at time t and m is the mass of the particle. In addition, we know two things: the initial condition s_{t_0} and the function f, which states how forces are generated based on the state s_t at time t.

Since s, the state of a particle changes over time, we can think s as a function of t, i.e. $s(t)$. The shape of the function $s(t)$ is usually too complex to be described as a simple function. In addition, we only need to know the value of s at discrete times, i.e., $s(t_0)$, $s(t_0 + h)$, $s(t_0 + 2h)$, $\cdots$, etc.

3 ODE solvers

We assume that the particle's state $s(t)$ is an analytical function. Therefore, given a known state $s(t)$ at time t, we can obtain the new state $s(t+h)$ via *Taylor series expansion*. That is,

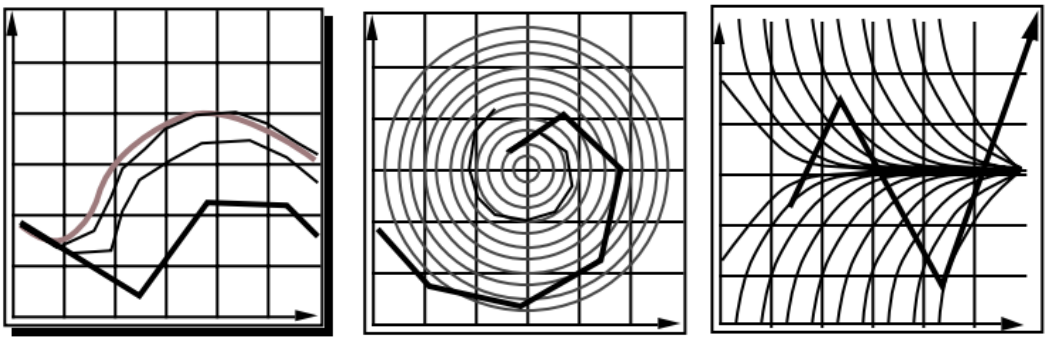
$$s(t+h) = s(t) + hs'(t) + \frac{h^2}{2}s''(t) + \frac{h^3}{3!}s'''(t) + \cdots = \sum_{n=0}^{\infty}\frac{h^n}{n!}s^{(n)}(t) \ .$$

All the numerical methods that we will see below approximate this representation.

3.1 Euler method

The new state is obtained as: $s(t+h) = s(t) + hs'(t) = \begin{bmatrix} x \\ v \end{bmatrix} + h \begin{bmatrix} v \\ \frac{F}{m} \end{bmatrix}$.

Error The error of Euler method is $\frac{h^2}{2}s''(t) + \frac{h^3}{3!}s'''(t) + \cdots = O(h^2)$ for each step h. When your particle is simulated for a second, you will take $\frac{1}{h}$ steps (iterations). This means your error accumulates to $O(h^2 \cdot \frac{1}{h}) = O(h)$ per second. This means your error is linear to the step size. Smaller h means smaller error. See the figures below.



n-th order method We call an ODE solver an n-th order method if it has per step (not per second) error as $O(h^{n+1})$. Therefore, Euler method is called first order ODE solver. Note that h should be alway less than one, so $h << h^{n+1}$ is n is large (enough). This means, the higher-order ODE is more accurate than the lower-order ODE given that h is fixed. Or, if the same amount of error can be tolerated in the simulation, we can say that higher-order ODE is more efficient as larger h can be used.

Implementation Complete these functions:
//compute the derivative of the state (x, v) and store the results in (dx, dv)
//n is number of particles
DERIVE(int n, double x[], double v[], double dx[], double dv[])

//n is number of particles, h is the delta time, store results in (nx, nv)
EULER(int n, double h, double x[], double v[], double dx[], double dv[], double nx[], double nv[])

More about errors To have accurate approximation of $s(t + h)$ of a complex system, h is usually very small. Small h means that you will have to call ODE solver many many times in order to estimate the motion of the particle for even merely over a period of a second. This usually means inefficiency. Even if efficiency is not an issue, there is also numerical error when h is too small, such as round-off errors. That is when you multiply two very small numbers, the result may not be what you expected due to insufficient precision in our computer.

3.2 Midpoint method

Basic Idea Stop at the midpoint and evaluate the derivatives. Let us look at the position component $x(t+h)$ of $s(t+h)$ first:

$$x(t+h) = x(t) + h \cdot (\text{velocity at midpoint}) \tag{1}$$

velocity at midpoint The velocity at midpoint between time t and $t+h$ can be approximated as

$$v(t+\frac{h}{2}) = v(t) + \frac{h}{2} \cdot \dot{v} = v(t) + \frac{hF(t+\frac{h}{2})}{2m} ,$$

where $F(t+\frac{h}{2})$ is the force applied to the particle at time $t+\frac{h}{2}$. Therefore, for the position component, we can rewrite Equation 1 as:

$$x(t+h) = x(t) + h \cdot (\text{velocity at midpoint}) = x(t) + h \cdot v(t) + \frac{h^2 F(t+\frac{h}{2})}{2m}$$

The velocity component The velocity component $v(t+h)$ of $s(t+h)$ is actually easier.

$$v(t+h) = v(t) + h \cdot (\text{acceleration at midpoint}) = v(t) + \frac{hF(t+\frac{h}{2})}{2m}$$

midpoint method Let summarize.

$$s(t+h) = \begin{bmatrix} x \\ v \end{bmatrix} + h \cdot \begin{bmatrix} \text{velocity at midpoint} \\ \text{acceleration at midpoint} \end{bmatrix} = \begin{bmatrix} x \\ v \end{bmatrix} + h \begin{bmatrix} v(t) + \frac{hF(t+\frac{h}{2})}{2m} \\ \frac{F(t+\frac{h}{2})}{2m} \end{bmatrix}$$

Question How do you get $F(t+\frac{h}{2})$?

Implementation You can implement the midpoint method by calling Euler method twice. How?

Error Compare Taylor serier expansion $s(t+h) = s(t) + hs'(t) + \frac{h^2}{2}s''(t) + \frac{h^3}{3!}s'''(t) + \cdots$ and our midpoint method is $s(t) + hv(t) + \frac{h^2}{2}F(t)/m$, we can see that terms after $\frac{h^3}{3!}s'''(t)$ are missing. Therefore the error is $O(n^3)$ per step and $O(n^2)$ per second. Therefore, the midpoint method is a second order method.

3.3 3rd and 4th Order Runge-Kutta methods

Runge-Kutta methods are a set of related methods. They were proposed by two German mathematicians in 19 century. Midpoint method can be though as a 2nd order Runge-Kutta method. The basic idea of Runge-Kutta methods are based on valuation of derivatives multiple times between t and $t + h$.

3rd order Runge-Kutta Here is the formula for the 3rd order Runge-Kutta method:

$$\begin{array}{lcl} k_1 & = & \text{start velocity \& acceleration} \\ k_2 & = & \text{midpoint velocity \& acceleration} \\ k_3 & = & \text{end velocity \& acceleration (using } k_2) \\ s(t+h) & = & s(t) + \frac{h}{6}\left(k_1 + 4k_2 + k_3\right) \end{array}$$

The error of the 3rd order Runge-Kutta method is $O(h^4)$ per step.

Implementation You can implement the midpoint method by calling Euler method three times. How?

4th order Runge-Kutta Known as RK4. The most popular ODE solver due to its balance in efficiency and numerical stability. Here is the formula for the 4th order Runge-Kutta method:

$$
\begin{array}{lcll}
k_1 & = & \text{start velocity \& acceleration} & = \begin{bmatrix} v(t) \\ F(t)/m \end{bmatrix} \\
k_2 & = & \text{midpoint velocity \& acceleration} & \\
k_3 & = & \text{midpoint velocity \& acceleration (using } k_2) & \\
k_4 & = & \text{end velocity \& acceleration (using } k_3) & \\
s(t+h) & = & s(t) + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) &
\end{array}
$$

The error of the 4th order Runge-Kutta method is $O(h^5)$ per step.

Implementation