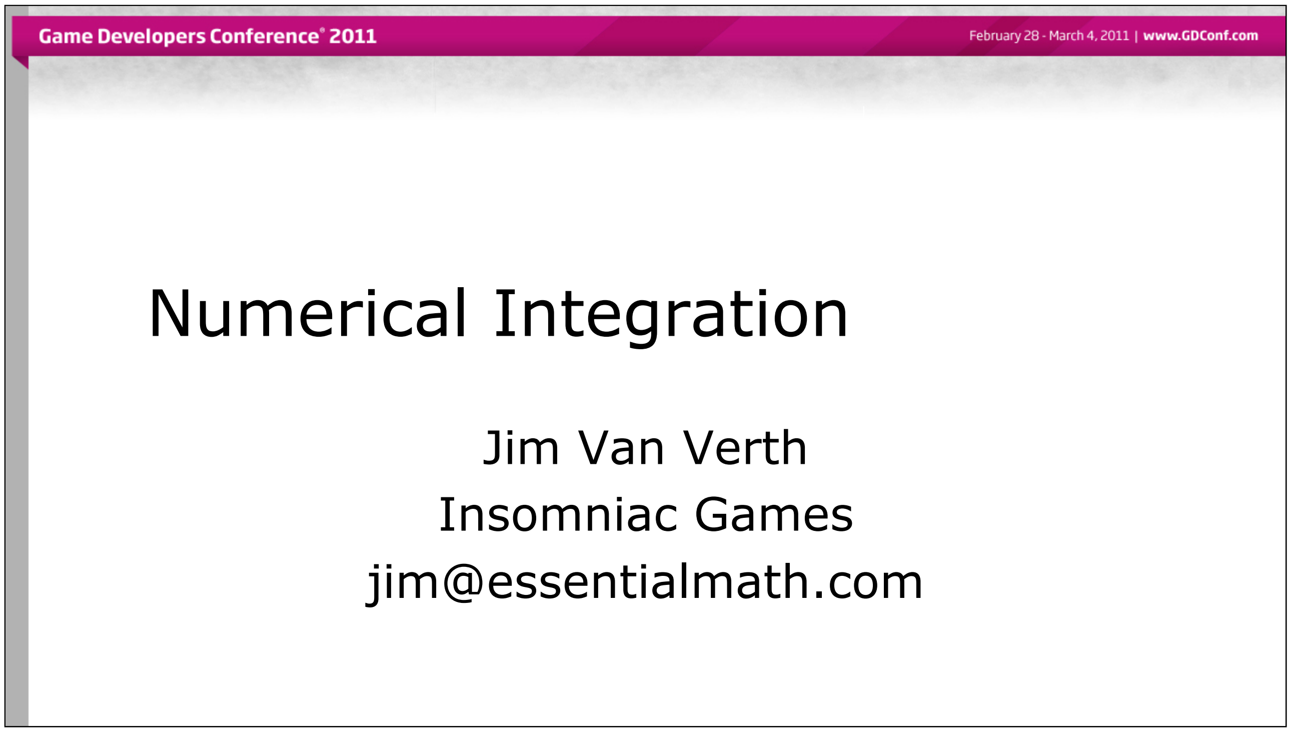
Game Developers Conference®

February 28 - March 4, 2011
Moscone Center, San Francisco

www.GDConf.com



Numerical Integration

Jim Van Verth
Insomniac Games
jim@essentialmath.com

Essential mathematics for games and interactive applications

Talk Summary

- Going to talk about:
  - Euler’s method subject to errors
  - Implicit methods help, but complicated
  - Verlet methods help, but velocity out of step
  - Symplectic methods can be good for both

Game Developers Conference® 2011

February 28 - March 4, 2011 | www.GDConf.com

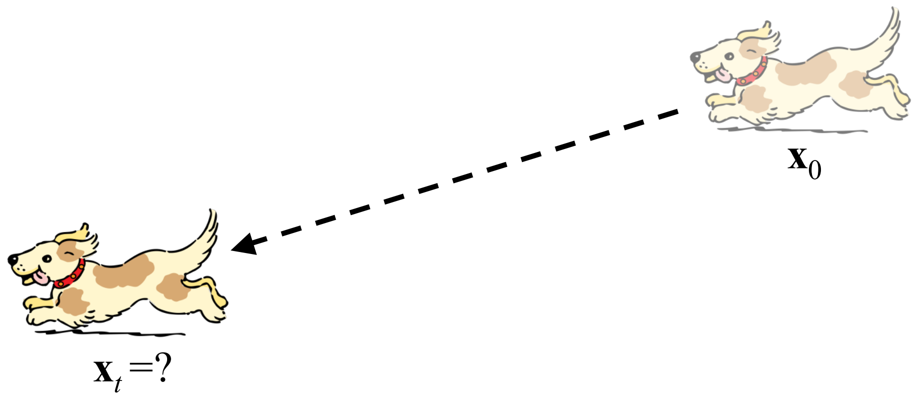
Our Test Case

So let's begin our discussion of integration methods with an overall definition of what I mean by numerical integration.

Our Test Case

Suppose that Mariano Rivera is out playing fetch with his dog.

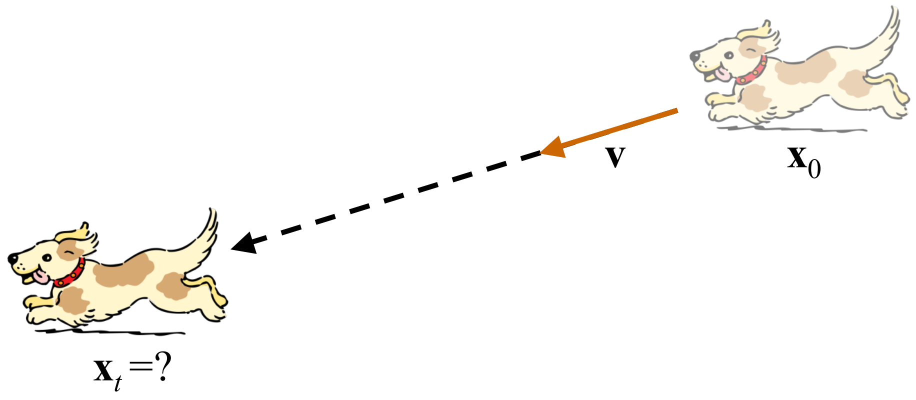
Our Test Case



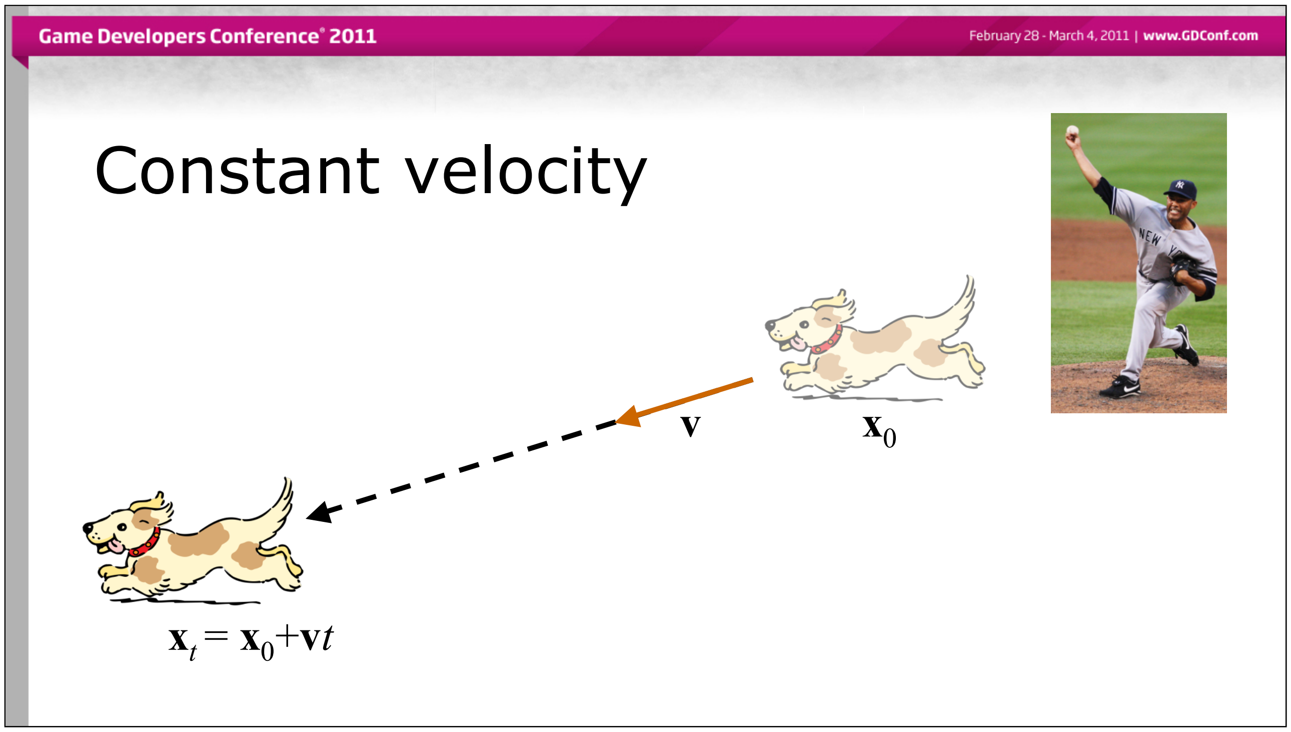
The dog likes to run and catch the ball in mid-air, and so Mariano wants to throw the ball where the dog will be after a certain time,

say at time t. We'll call the dog's original position x0, and the throw location x_t. How can he know where x_t will be?

Constant velocity



If the dog always runs straight with some velocity **v**, it’s simple enough.



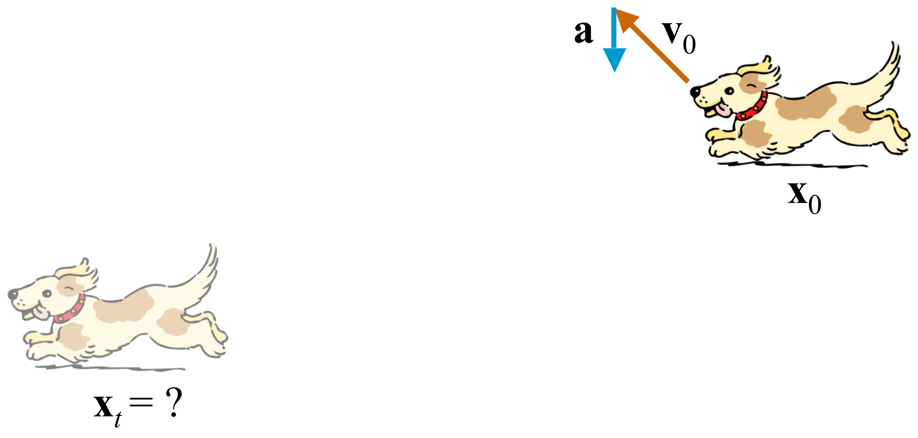
Constant velocity

$\mathbf{x}_t = \mathbf{x}_0 + \mathbf{v}t$

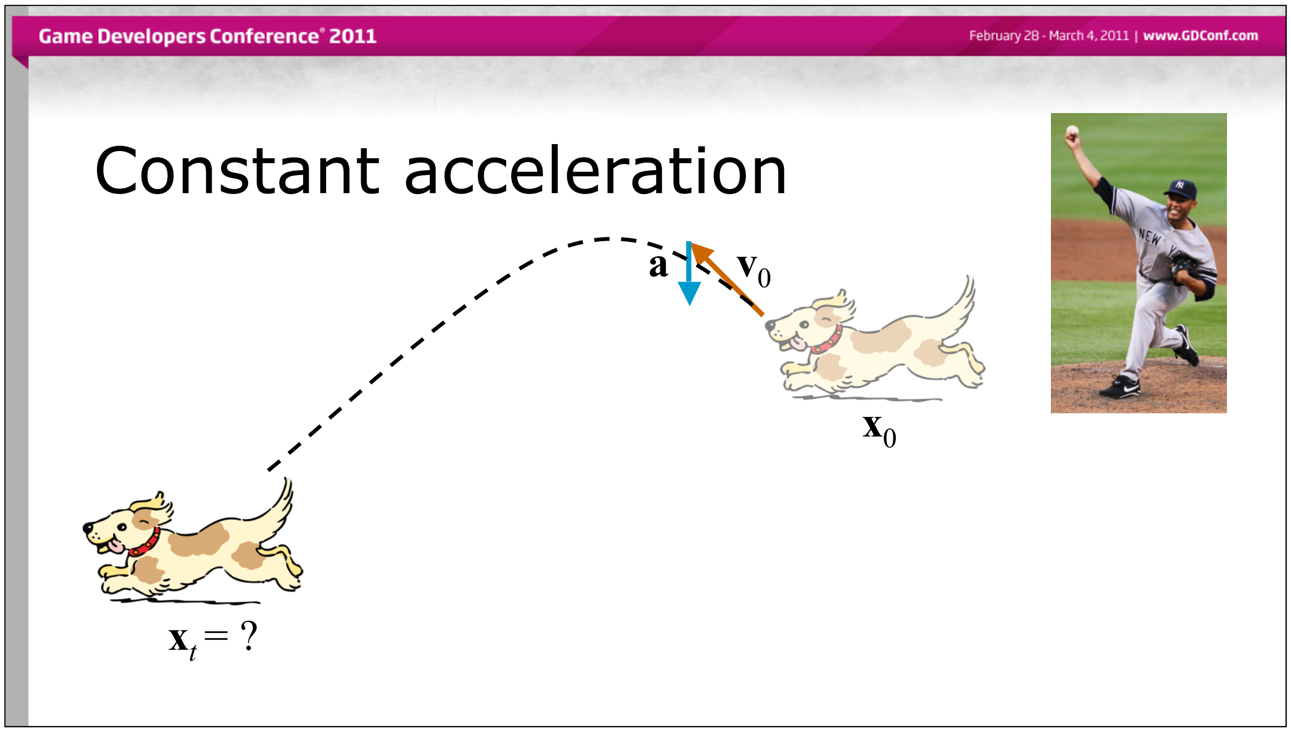
We just multiply the velocity by the time and add to the current position. We can represent this by the function x_t = x0 + vt.

Game Developers Conference® 2011 February 28 - March 4, 2011 | www.GDConf.com

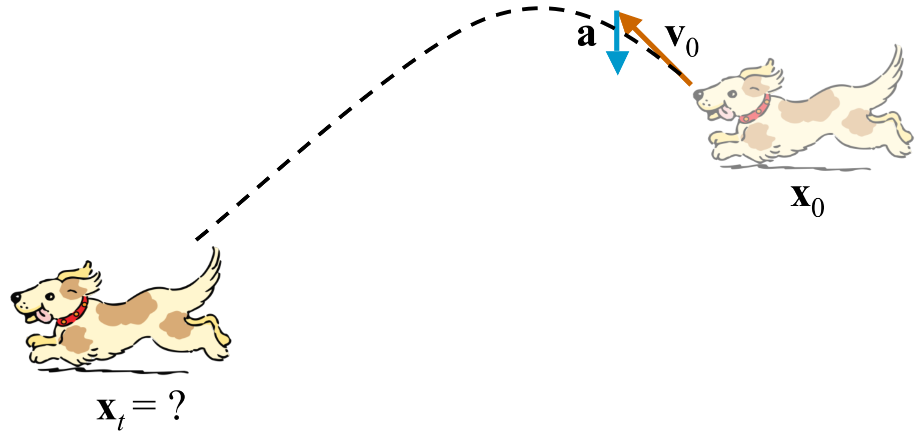
Constant acceleration



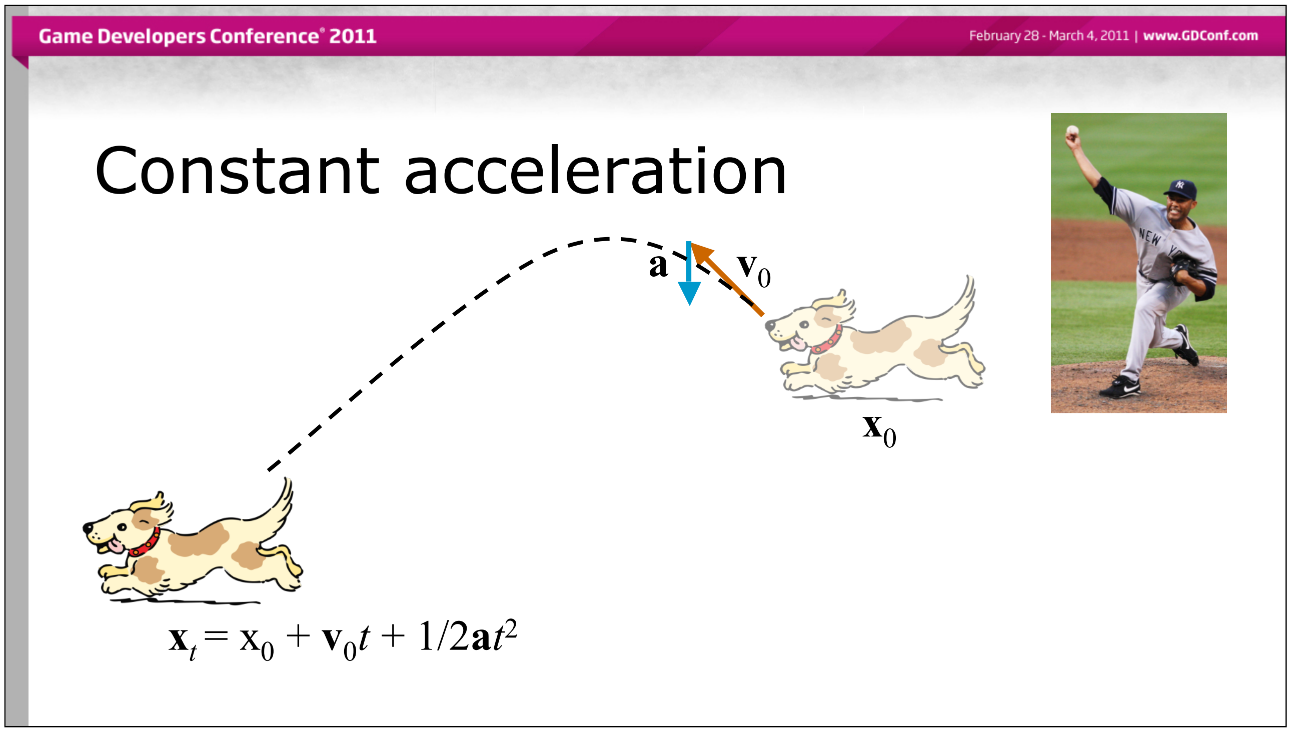
Now let’s make it more interesting. Suppose the dog’s velocity is changing constantly over time, based on some acceleration A.



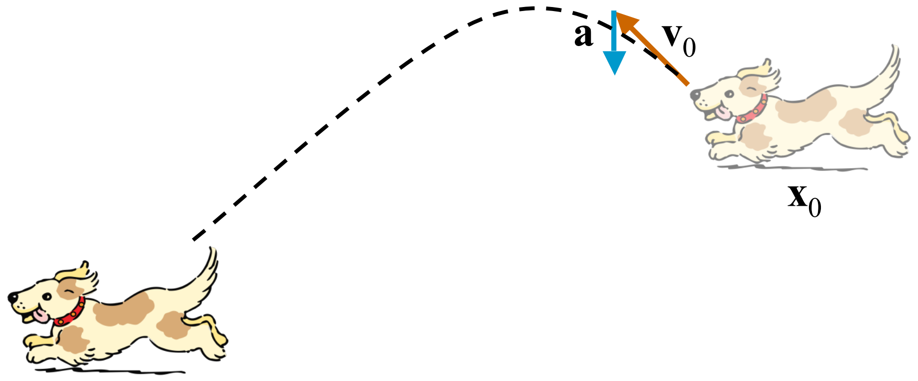
Constant acceleration



In this case, the path the dog takes is a parabola…



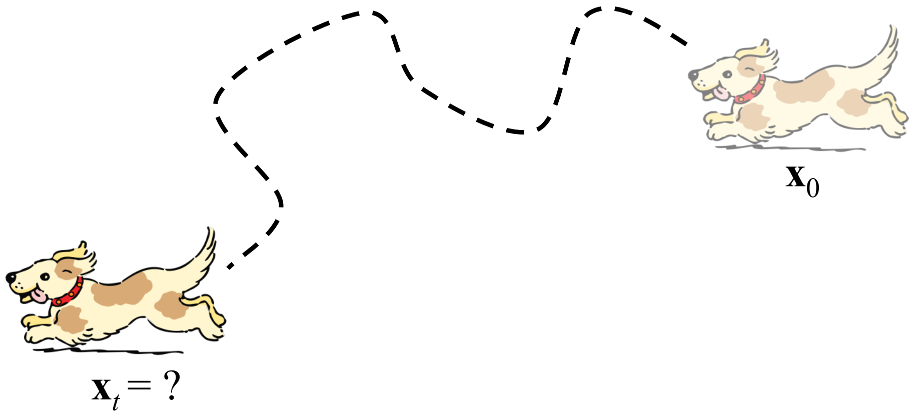
Constant acceleration



$$\mathbf{x}_t = x_0 + \mathbf{v}_0 t + 1/2\mathbf{a}t^2$$

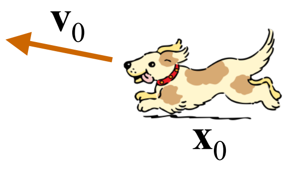
Represented by this quadratic function.

Variable acceleration



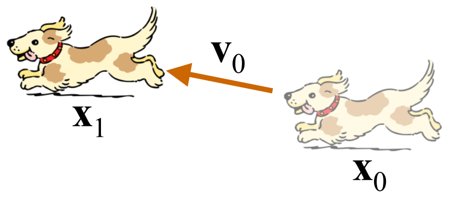
Now suppose the dog's path doesn't have these nice properties. The velocity changes at a non-constant rate, i.e. the acceleration may change dependent on time, position or velocity. Now, it's possible that we could derive a formula for this case like we did the others, I.e. integrate from acceleration to get a formula for the velocity function, and then integrate from velocity to get a function for position but a) it's not likely, and b) it may not be a very efficient formula. So what do we do?

Euler’s method

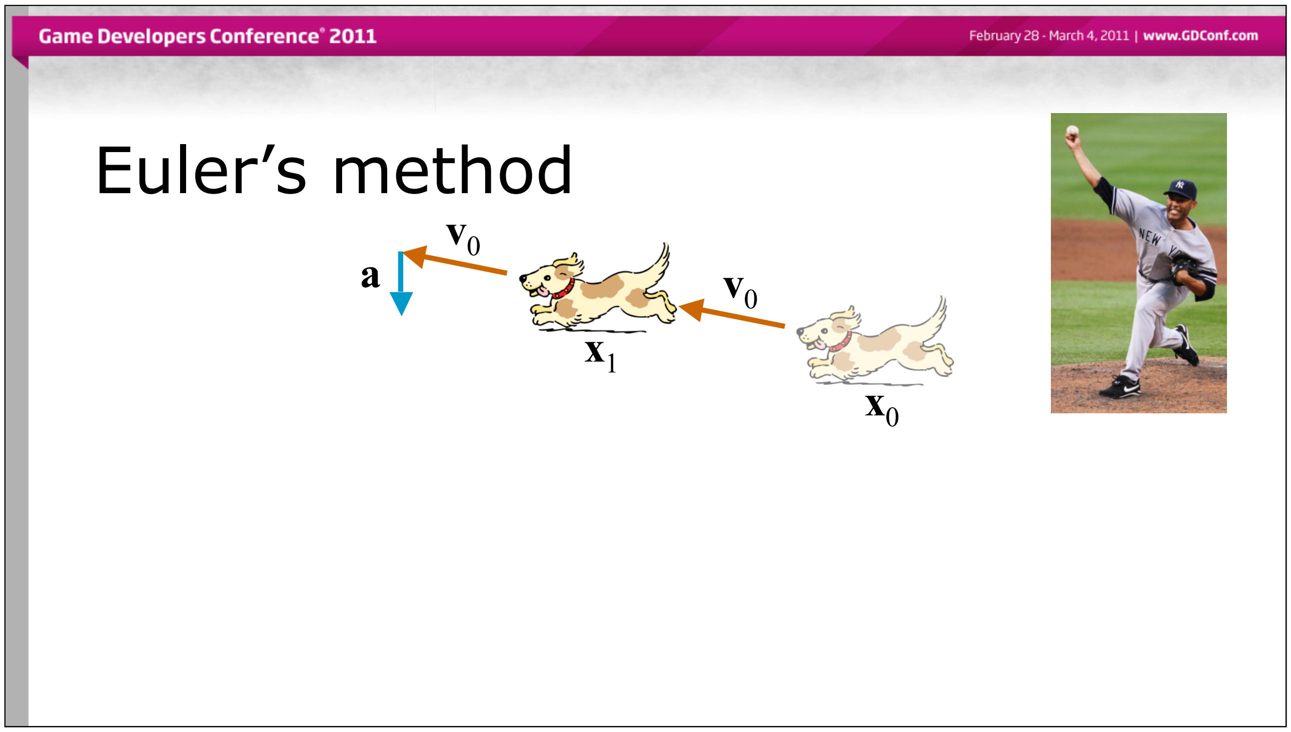


One possibility is to use our first approach and break our path into linear pieces. We’ll step a little bit in the direction of the velocity to update our position, and do the same to update our velocity. For the sake of simplicity I’ll use the parabola again, but in principle this works in any case.

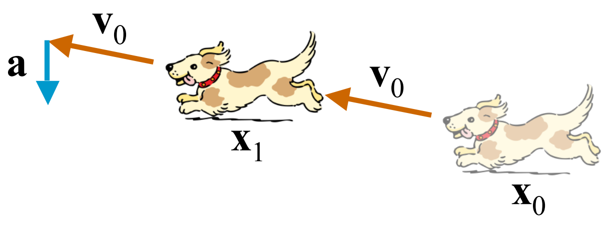
Euler's method



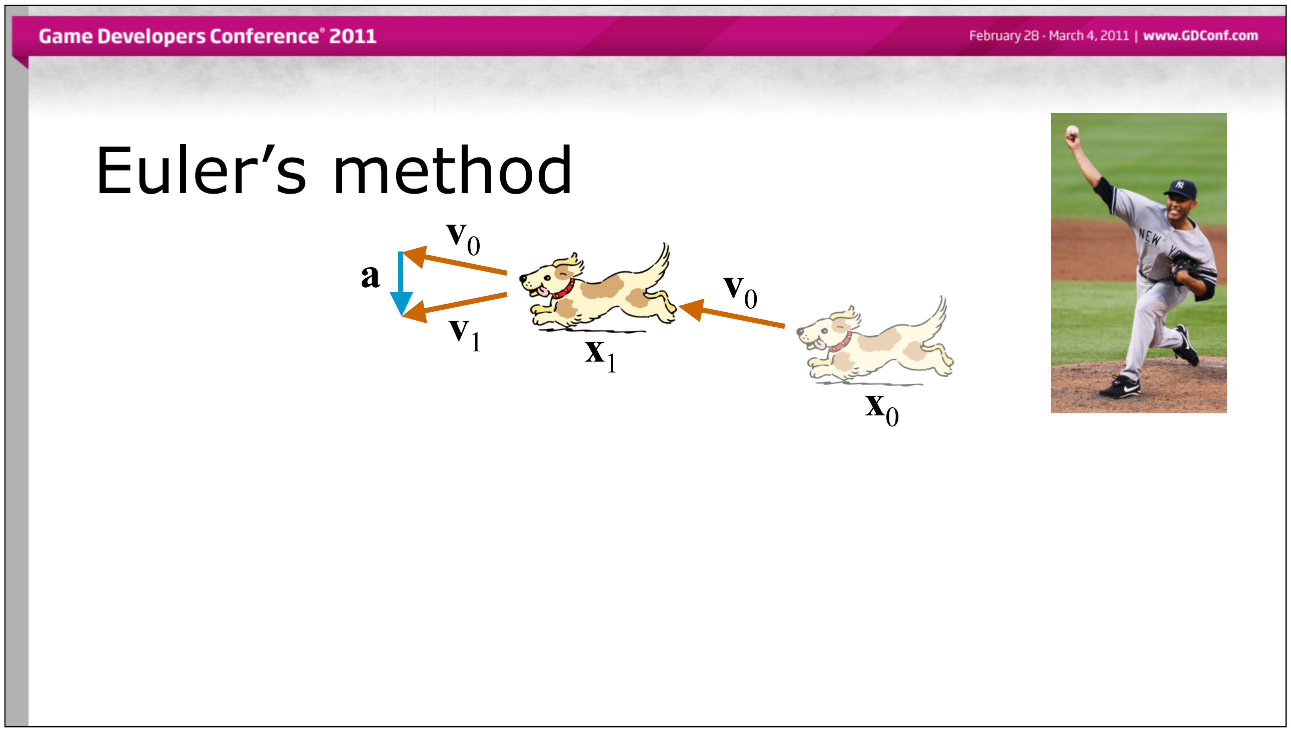
So first we step along the current velocity…



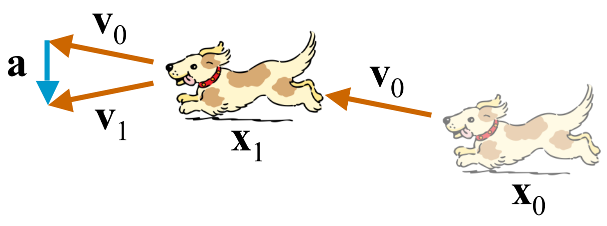
Euler’s method



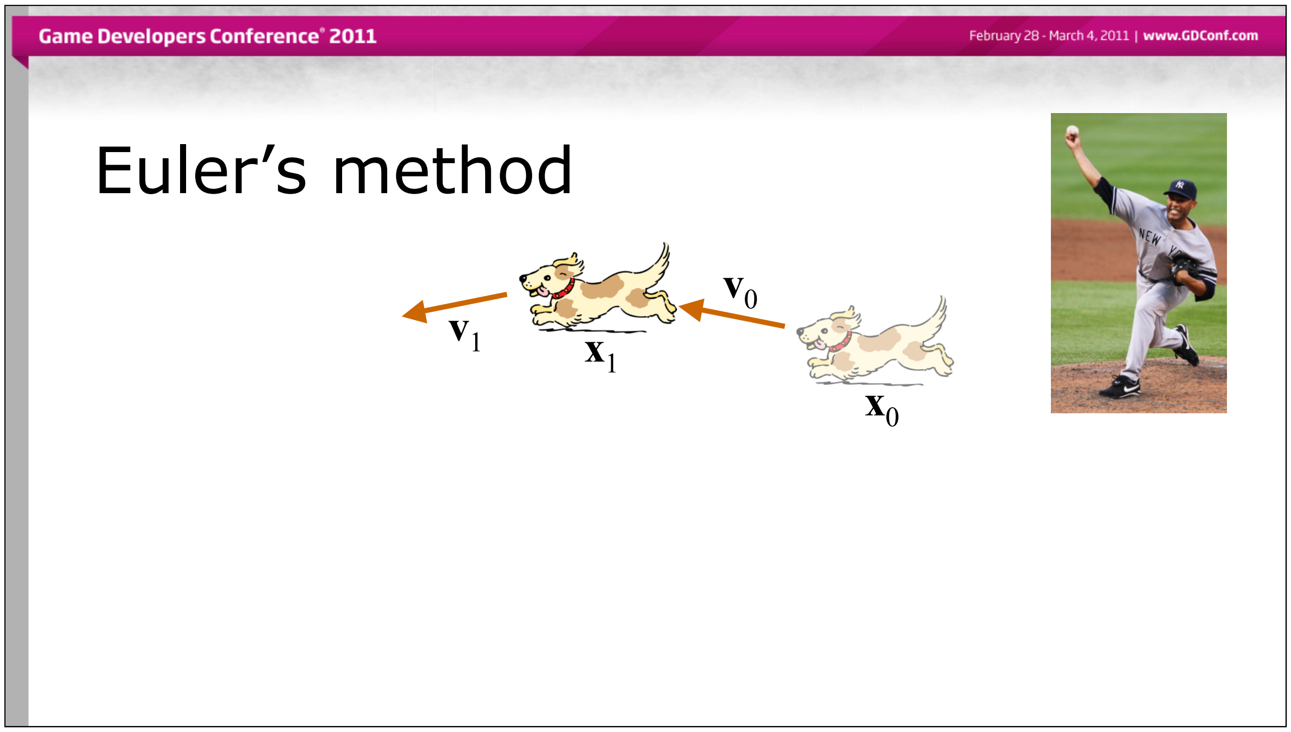
Then update the velocity based on the current acceleration



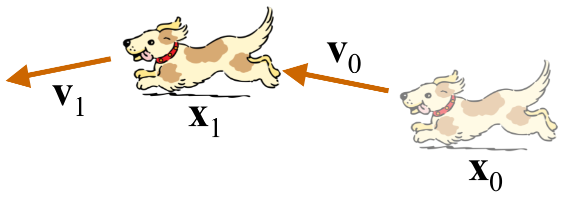
Euler’s method



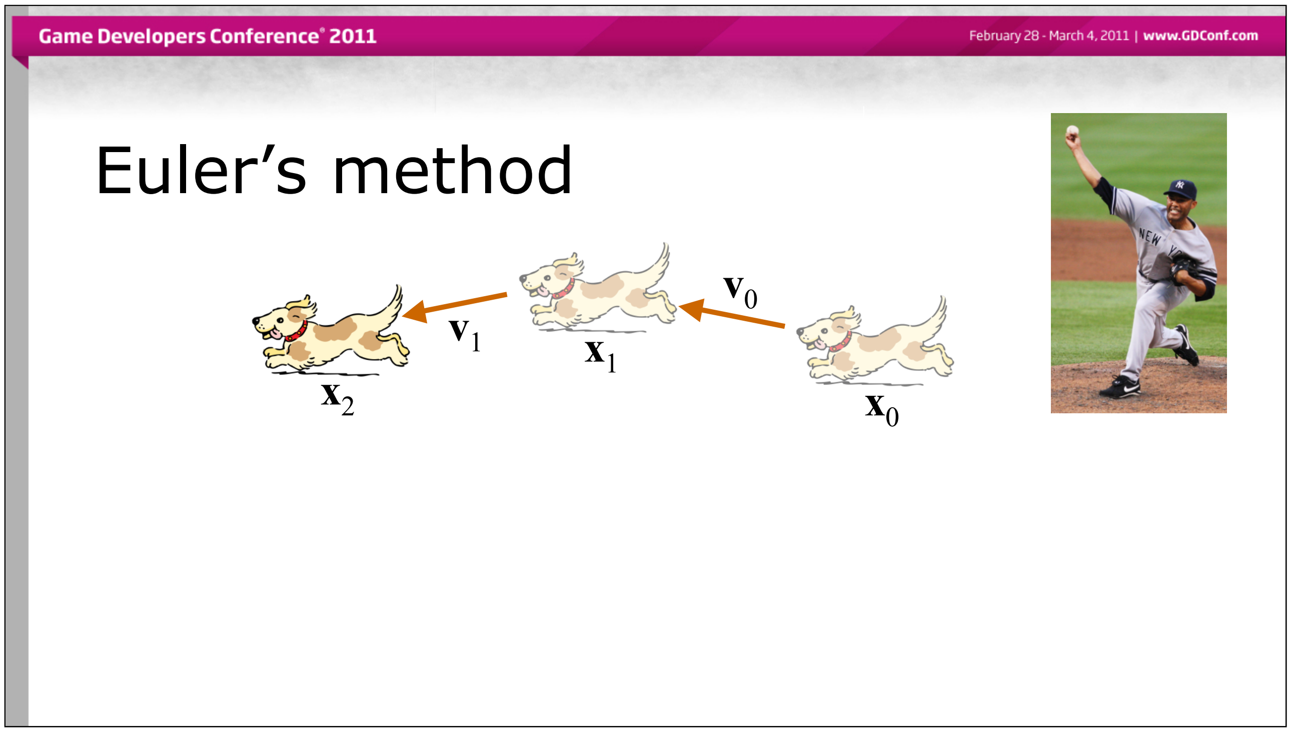
Like so



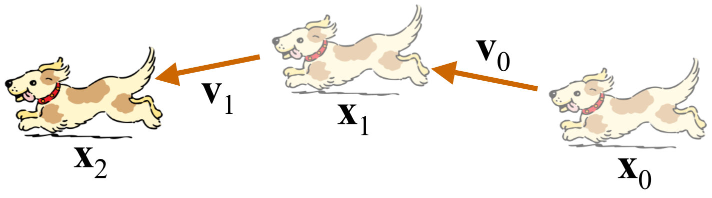
Euler's method



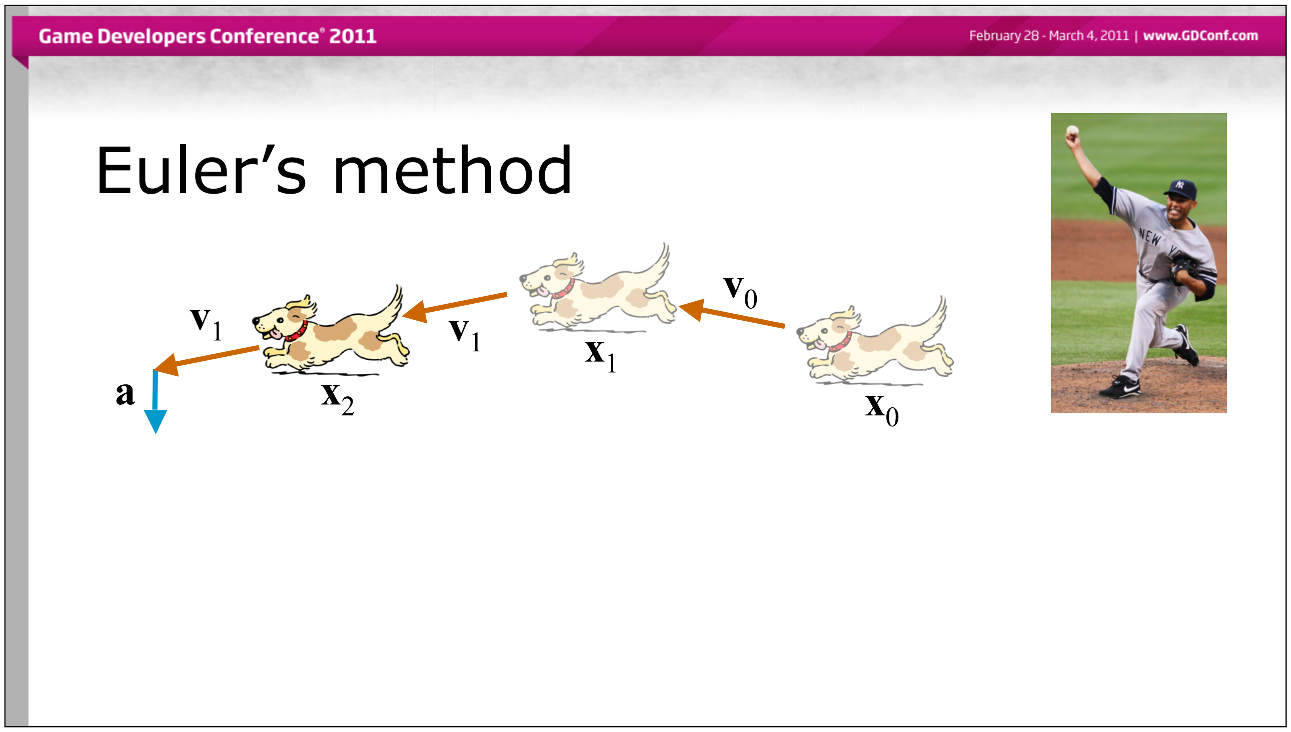
Then we're ready for the next position step



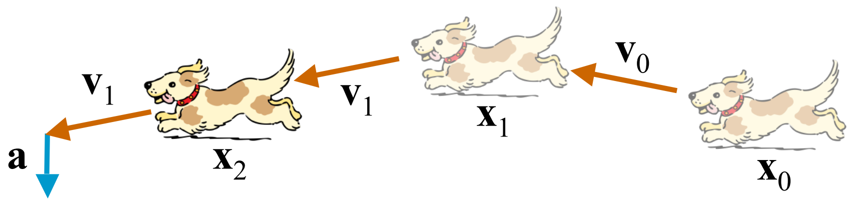
Euler’s method



Then we’re ready for the next position step

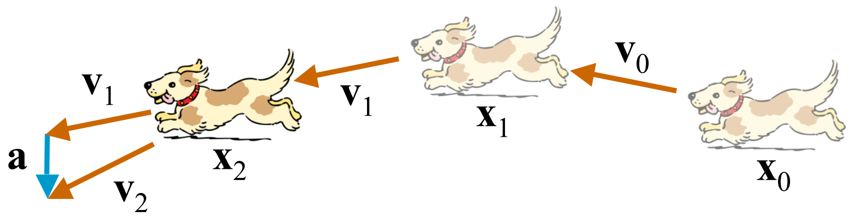


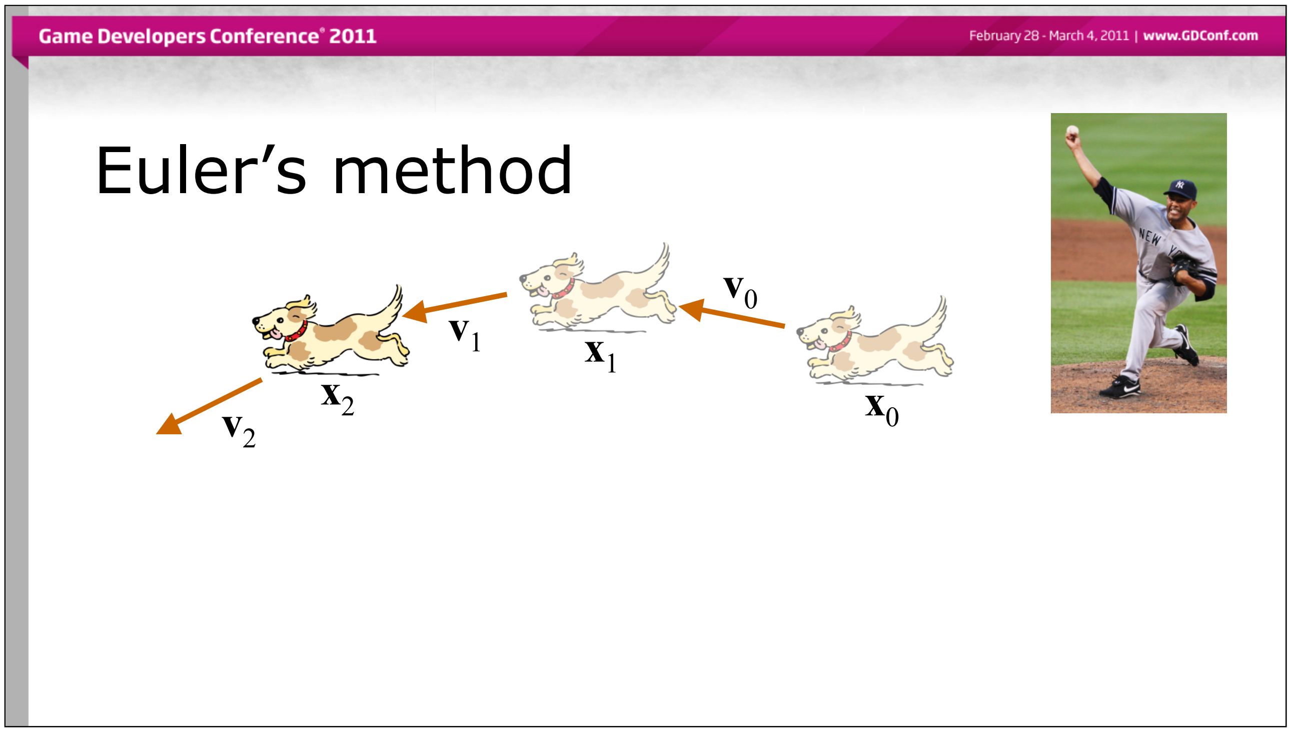
Euler’s method



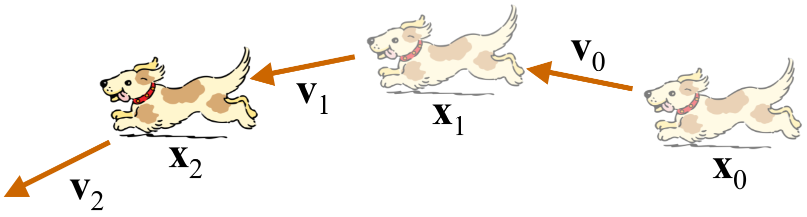
And update velocity

Euler’s method

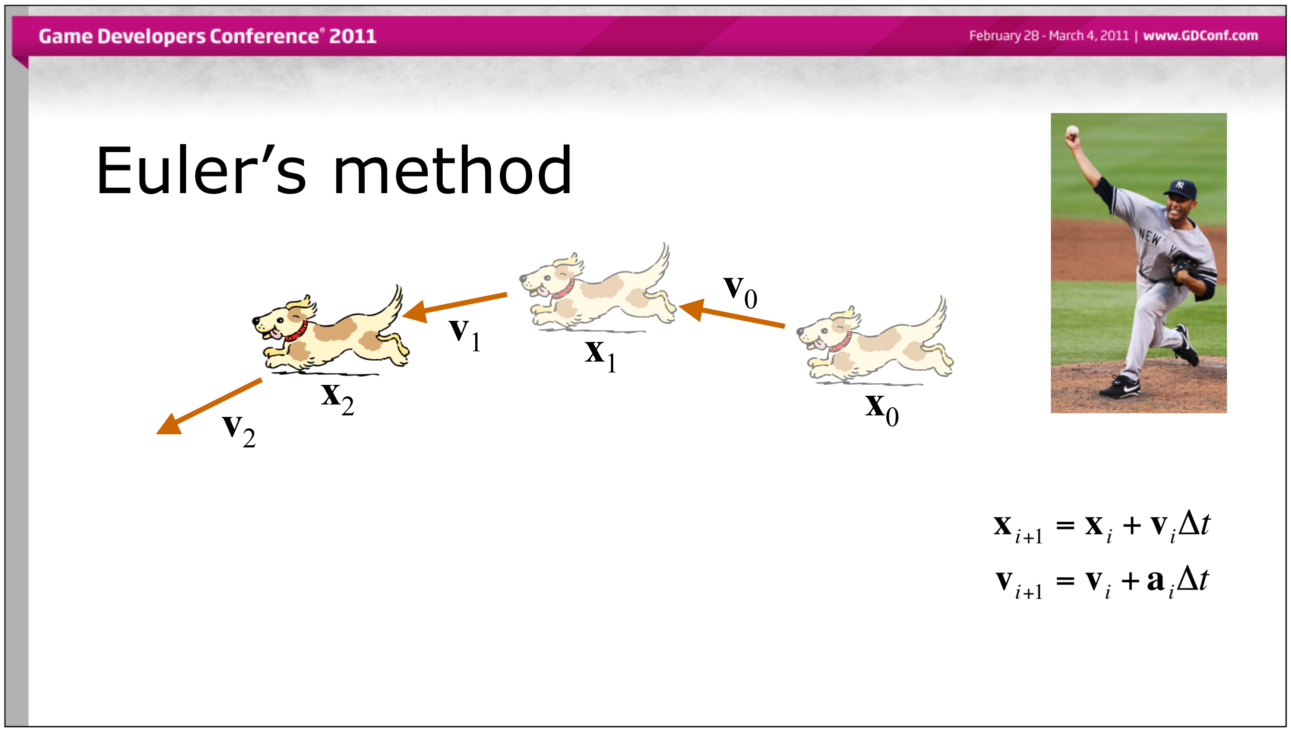




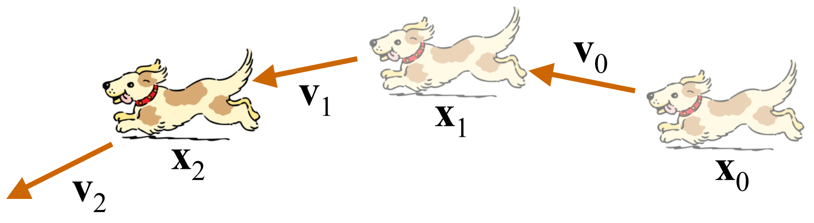
Euler's method



And there we go.



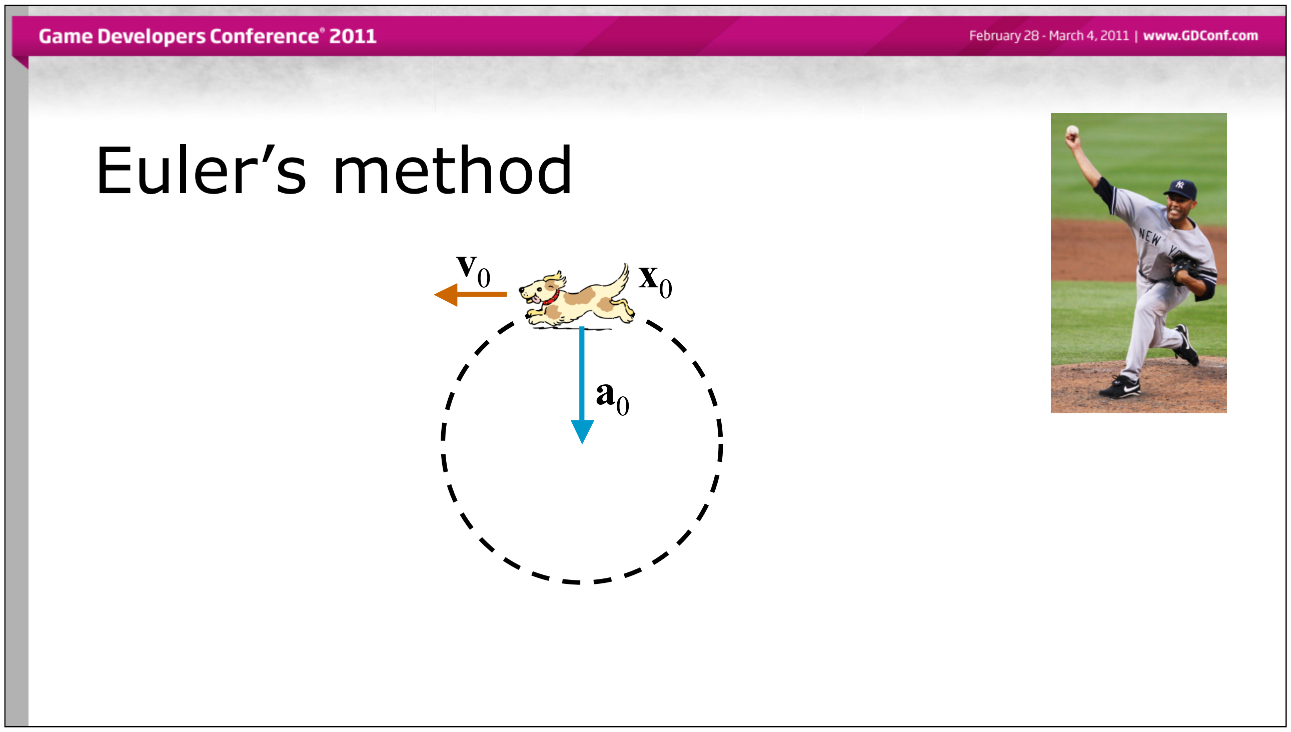
Euler’s method



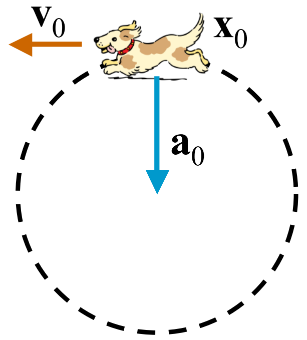
$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_i \Delta t$$
$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i \Delta t$$

We can represent this algebraically as follows: we update the position with its current derivative times the time step, and the velocity with its current derivative times the time step. Pretty simple.

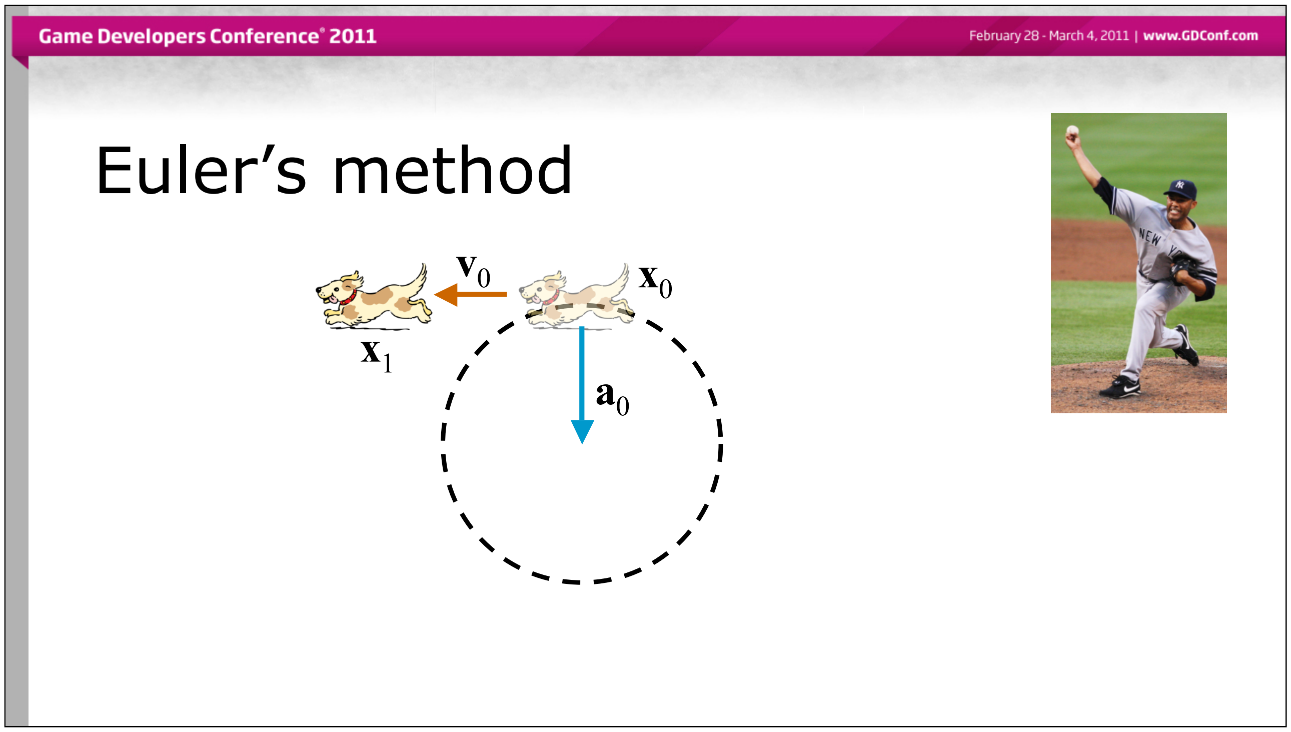
So that’s Euler’s method. While it seems straightforward, it has problems. Let’s consider another case.



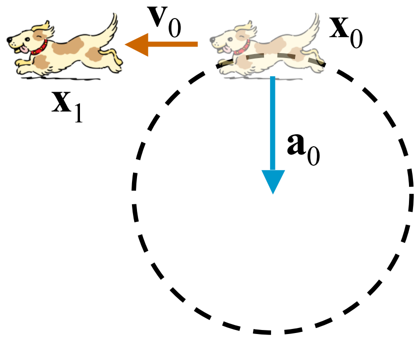
Euler's method



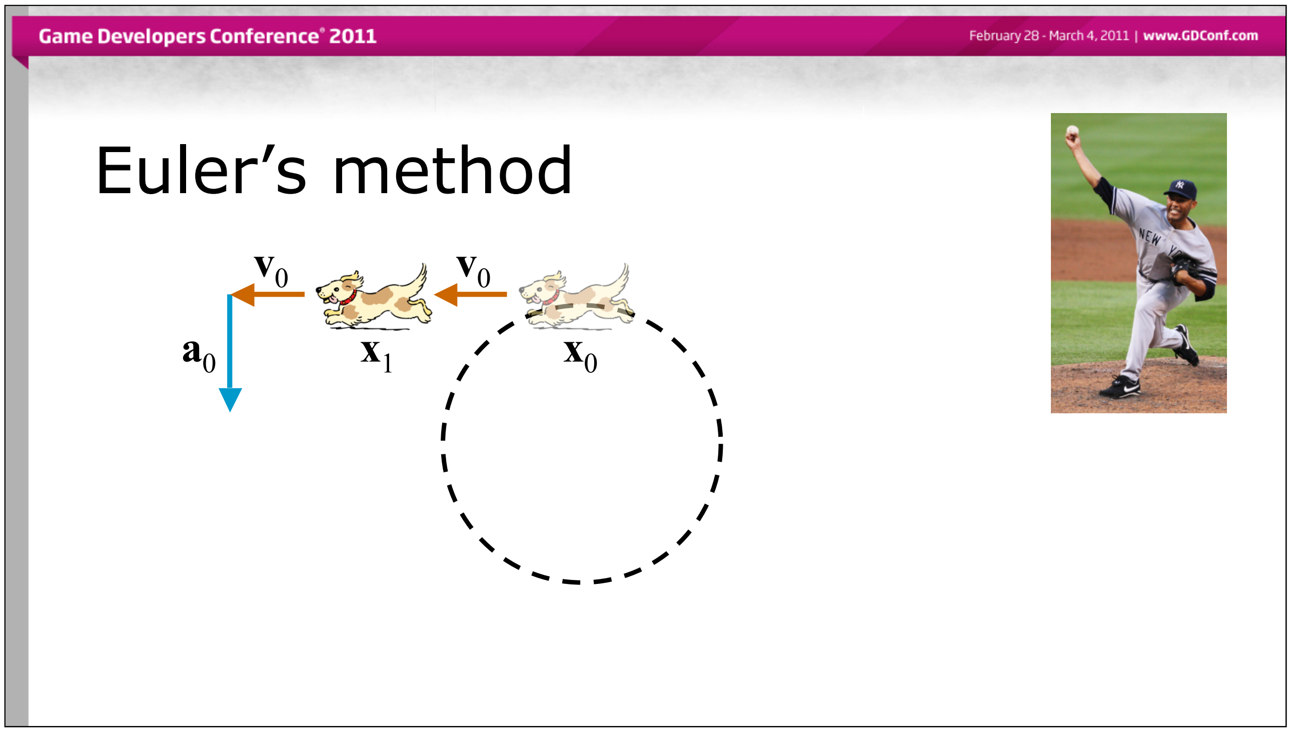
Suppose Mariano has attached his dog to a tree with a fixed rod, so the dog can only run in a circle. Hey, he's a member of the Yankees, so by definition he's evil, right? Now he wants to know where the dog will be at time t. Let's see what Euler's method will do.



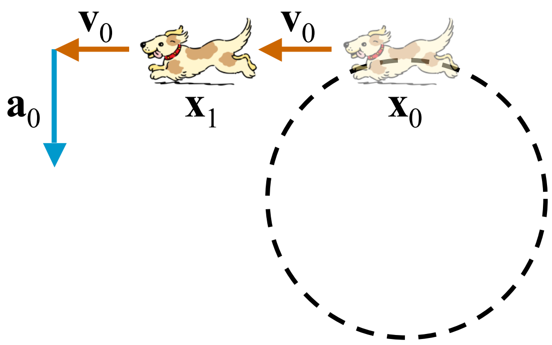
Euler’s method



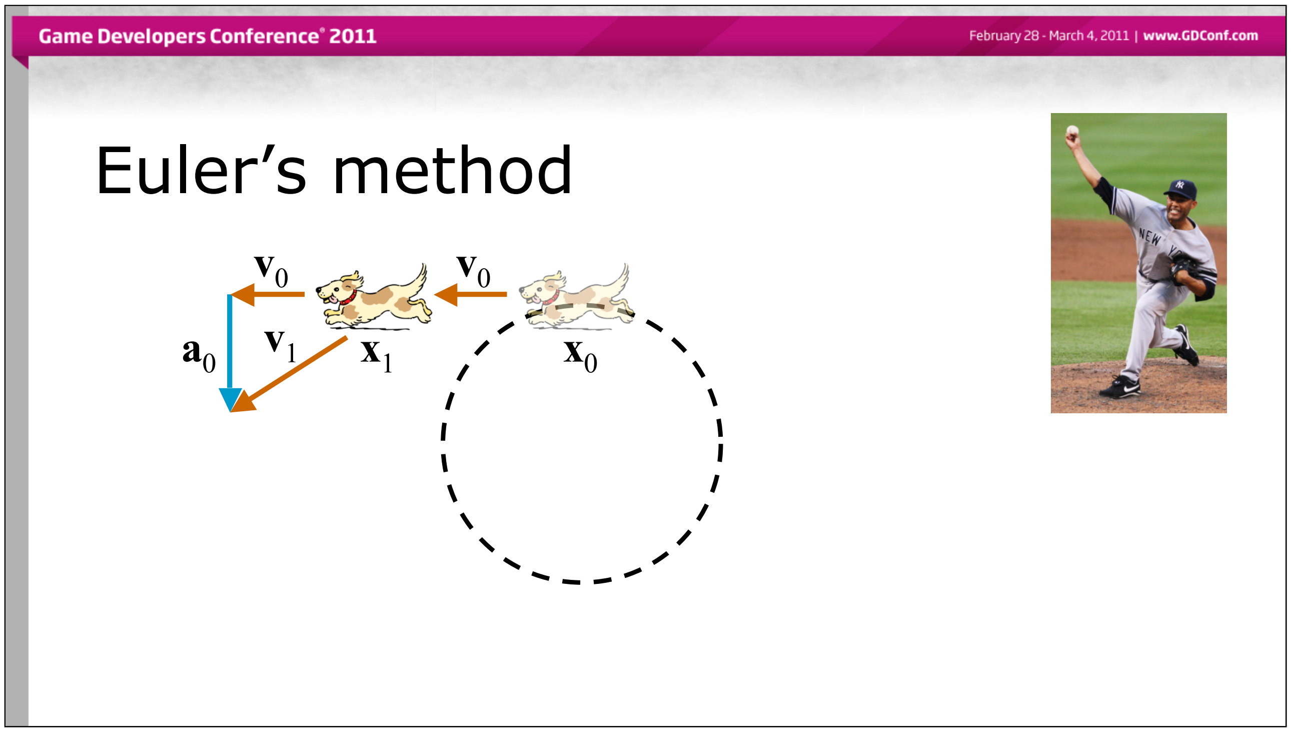
So we step position



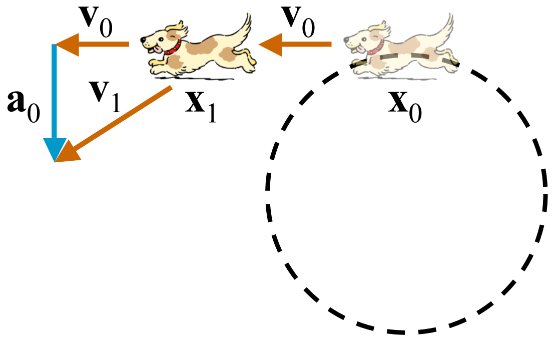
Euler’s method



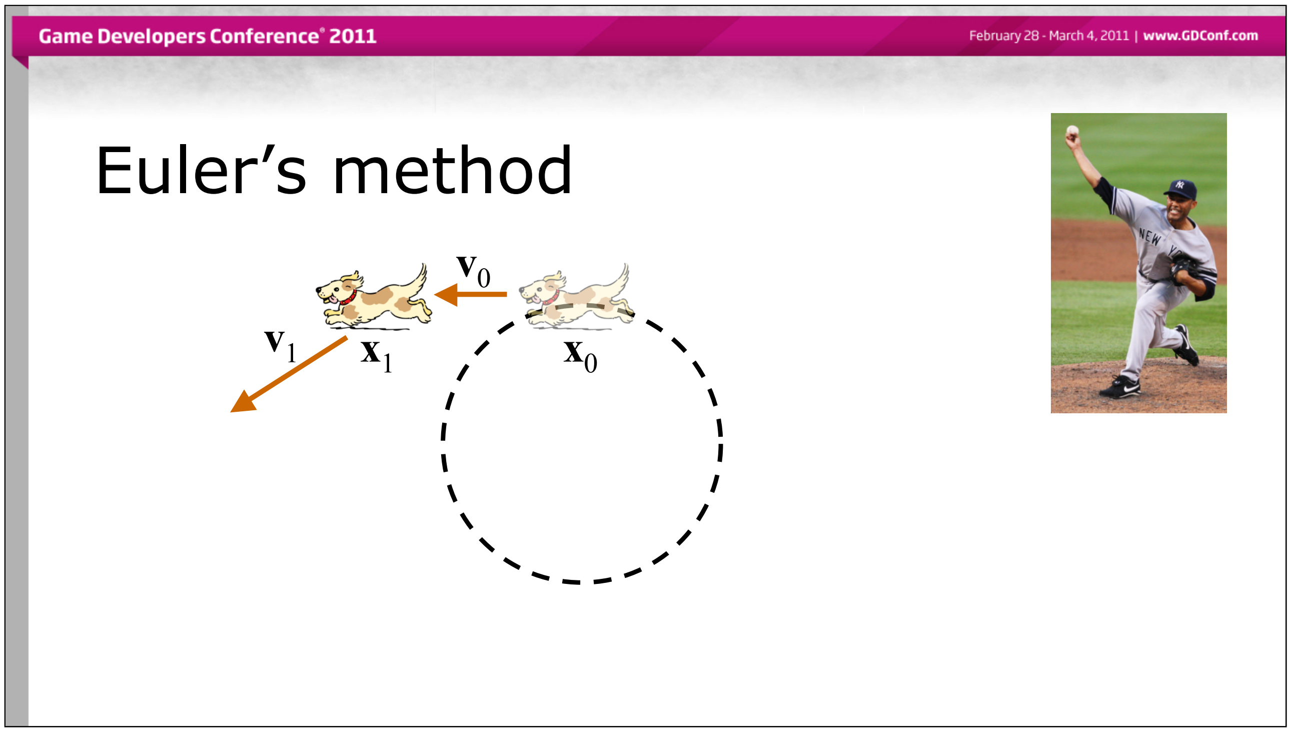
And velocity



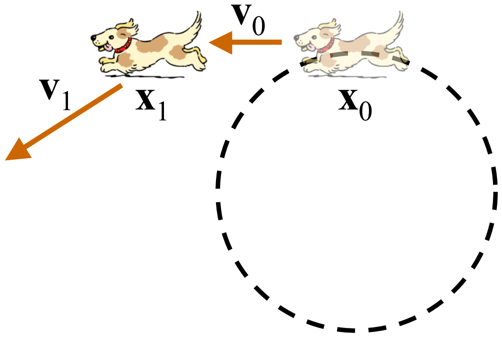
Euler's method



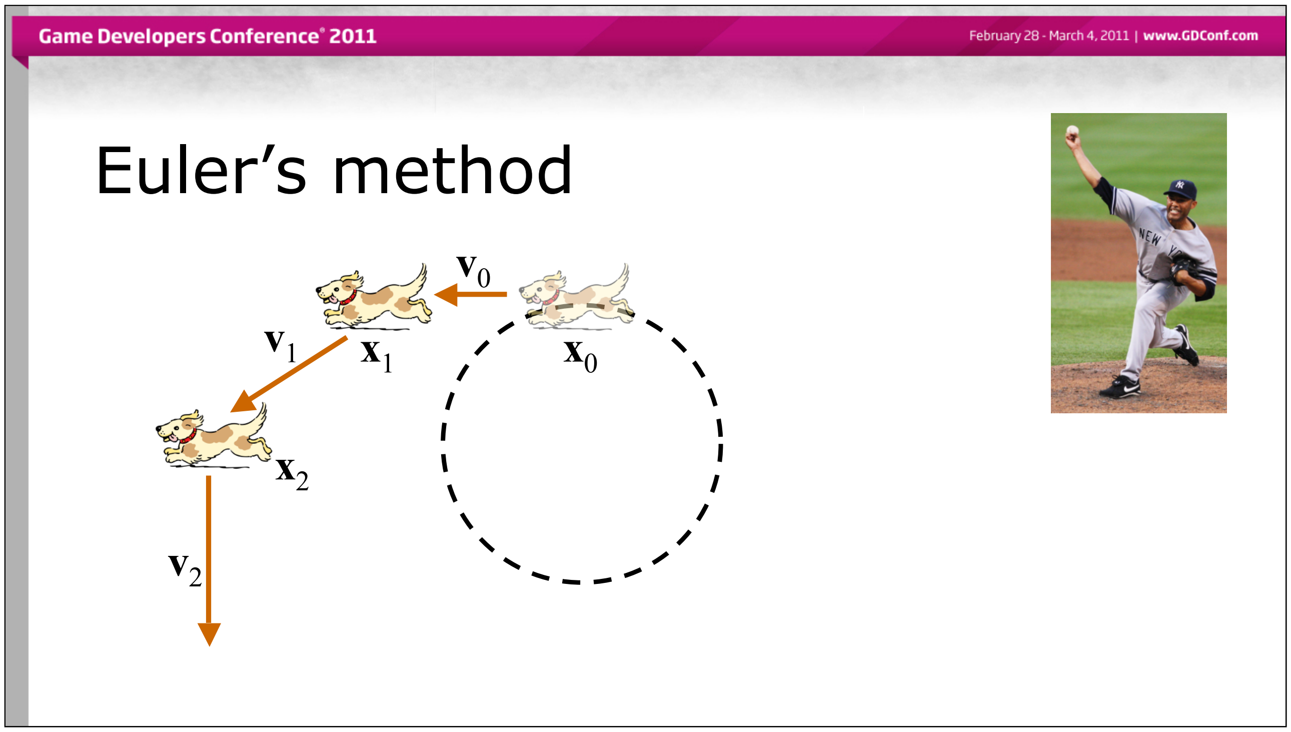
And velocity



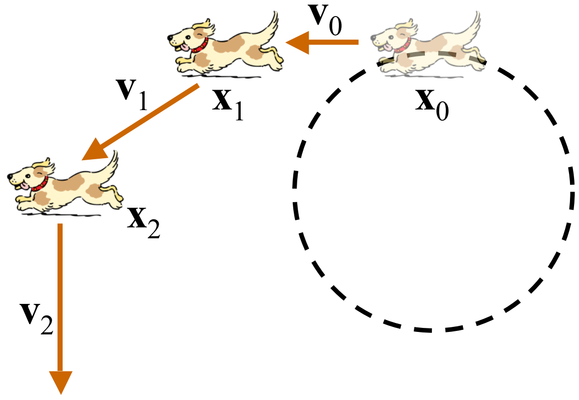
Euler’s method



So… that result is not so good. Let’s do one more iteration, just looking at velocity.

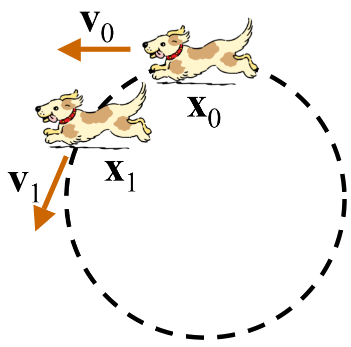


Euler's method



The end result? We are spiraling away from the actual path of the dog, and our velocities are getting larger. What is going on here?

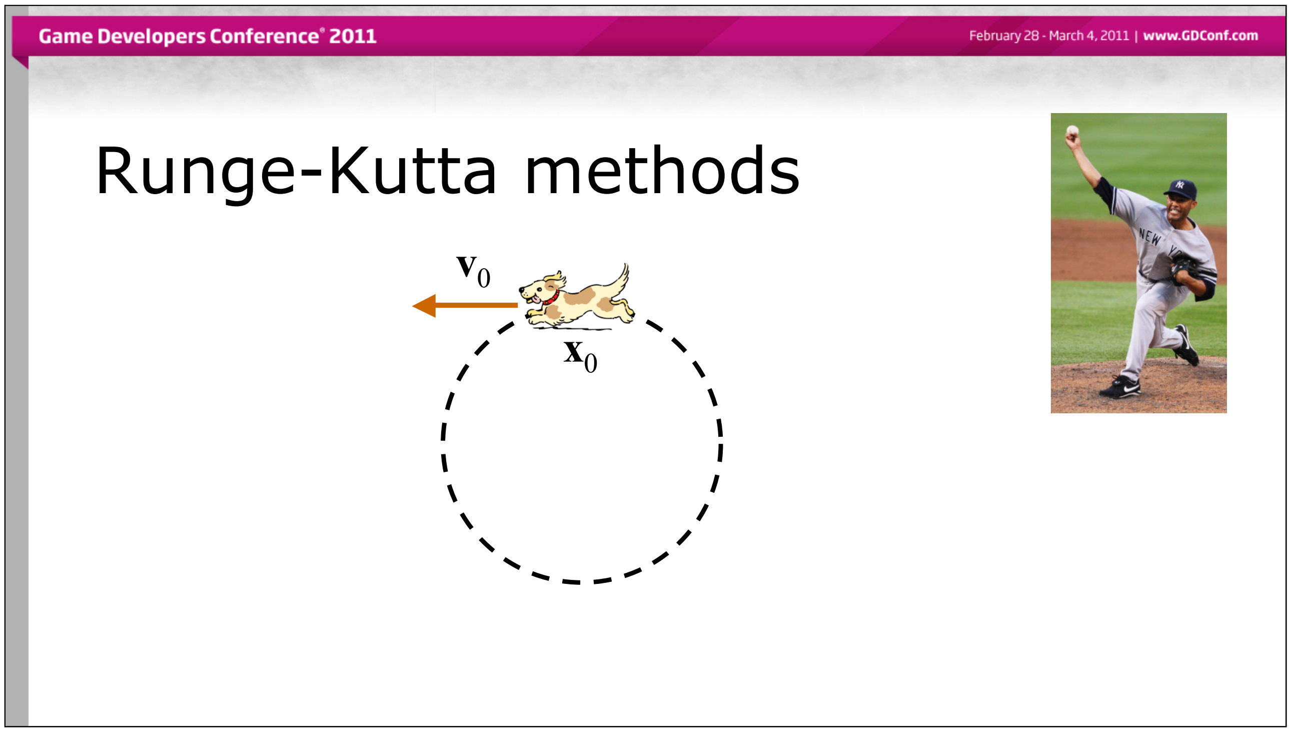
Euler’s method



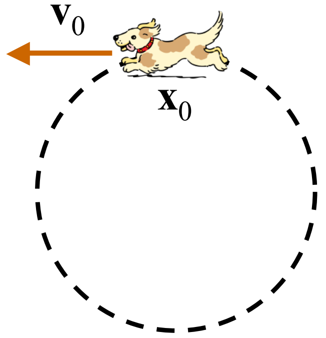
One problem is that the velocity varies a lot during the time step. We are assuming that the initial velocity is a good estimate of the average velocity across the interval, but in this case, it clearly is not. This introduces error into the simulation, and in the case of using Euler’s method with oscillating systems like the orbit, and springs, that error accumulates in a way that adds energy. Decreasing how far we step will decrease the error, but ultimately we will still have problems with energy gain.

Euler

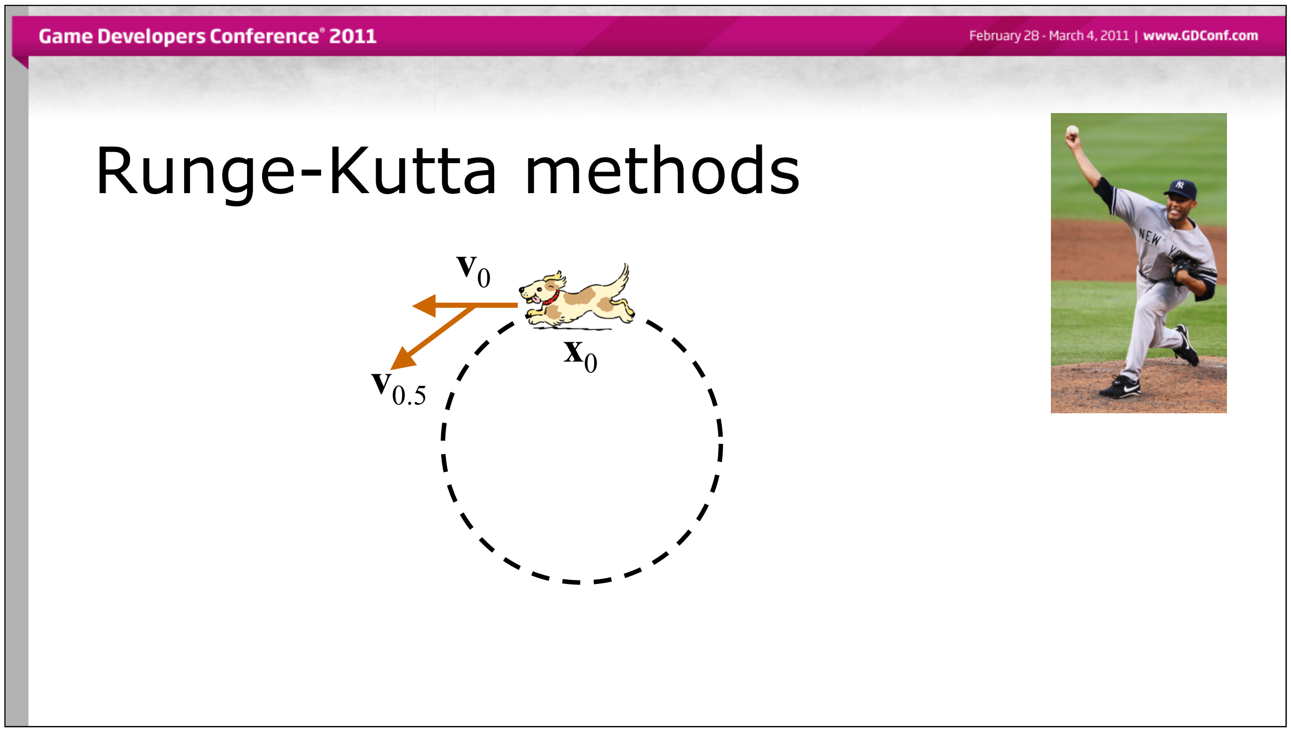
- Okay for non-oscillating systems
- Explodes with oscillating systems
- Adds energy! Very bad!



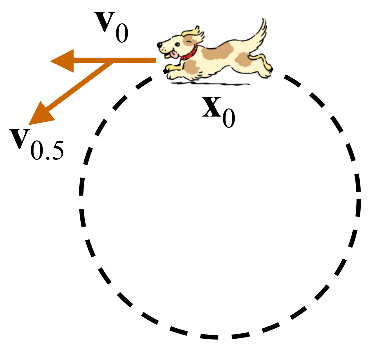
Runge-Kutta methods



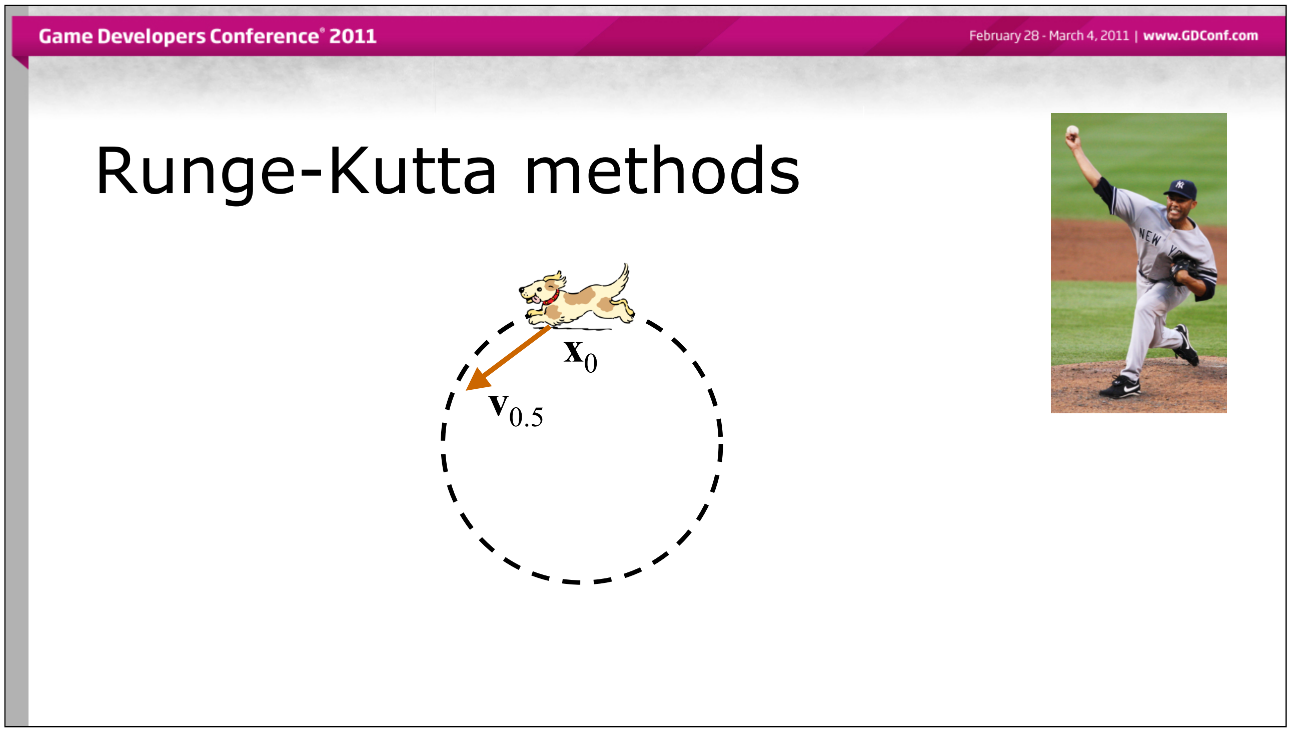
So remember that our current velocity wasn't a very good estimate for the average velocity during the time step. One solution is to take estimates of the velocity across the interval and use those to get a better velocity for the Euler step. For example, we might step halfway,



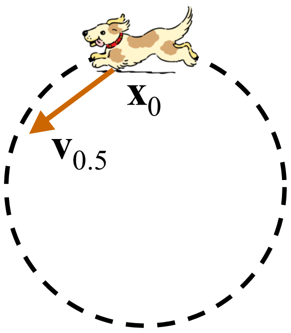
Runge-Kutta methods



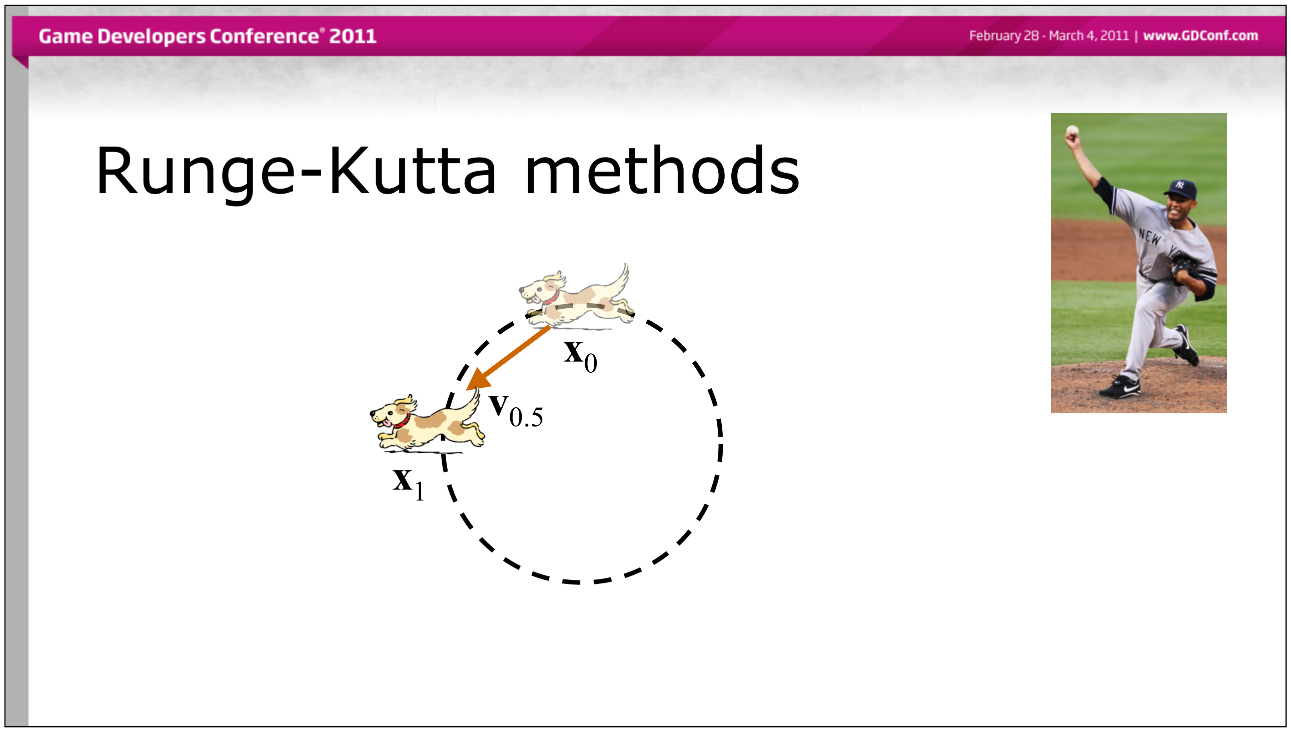
And use the position and velocity there to compute a new velocity and acceleration. Here I'm just showing the velocity to keep the diagram simple.



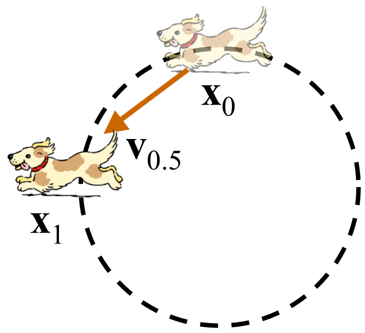
Runge-Kutta methods



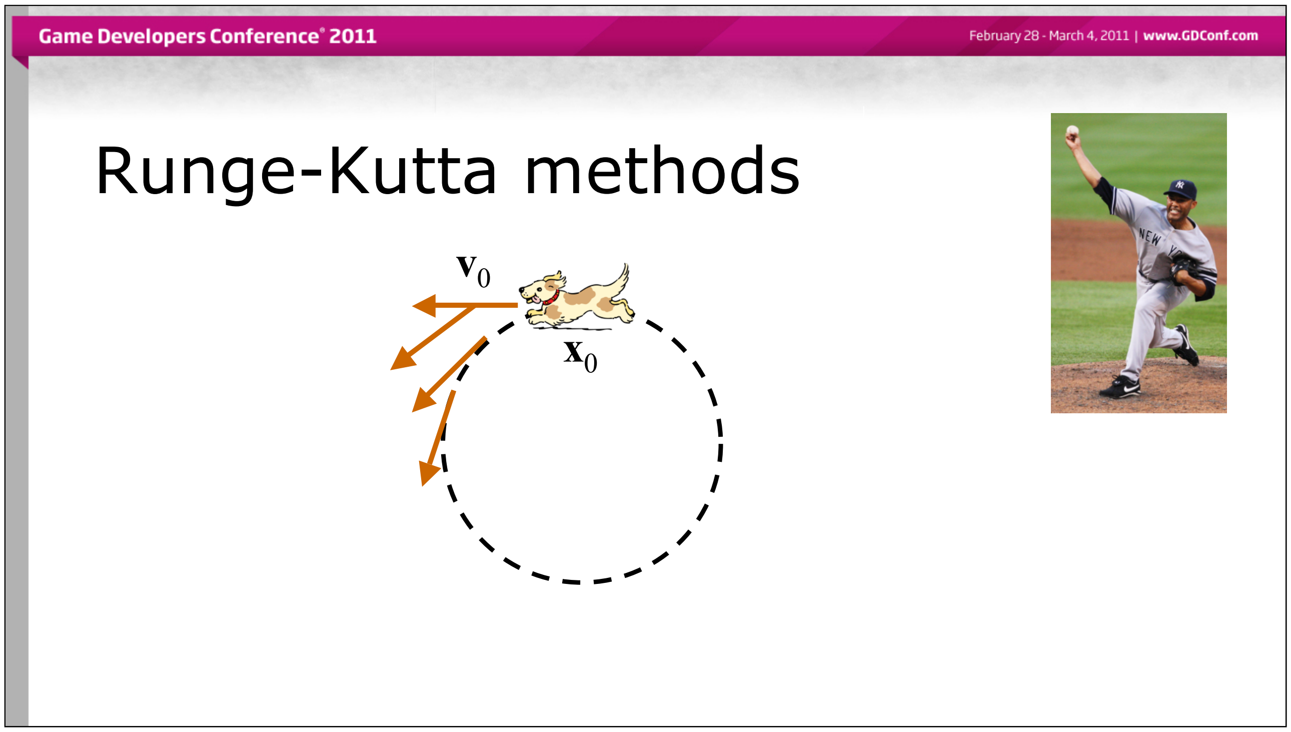
And then take the full time step with the newly calculated acceleration and velocity.



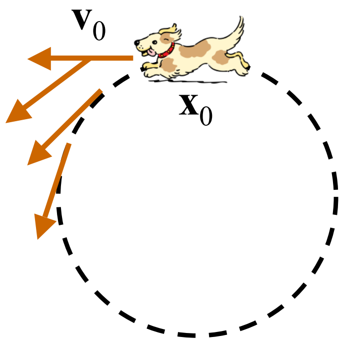
Runge-Kutta methods



And this is known as Midpoint method. As you can see, we can get better results this way, at least for this example.



Runge-Kutta methods

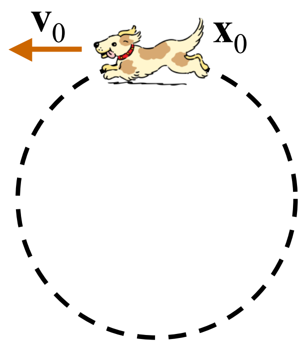


The most commonly known of these takes a weighted average of the original velocity and three estimates and is known as Runge-Kutta 4, or just RK4. The "4" in this case means that the order of the error is the time step to the 4th power. Midpoint method is an order 2 method. Euler's method is a Runge-Kutta method as well, just order 1.

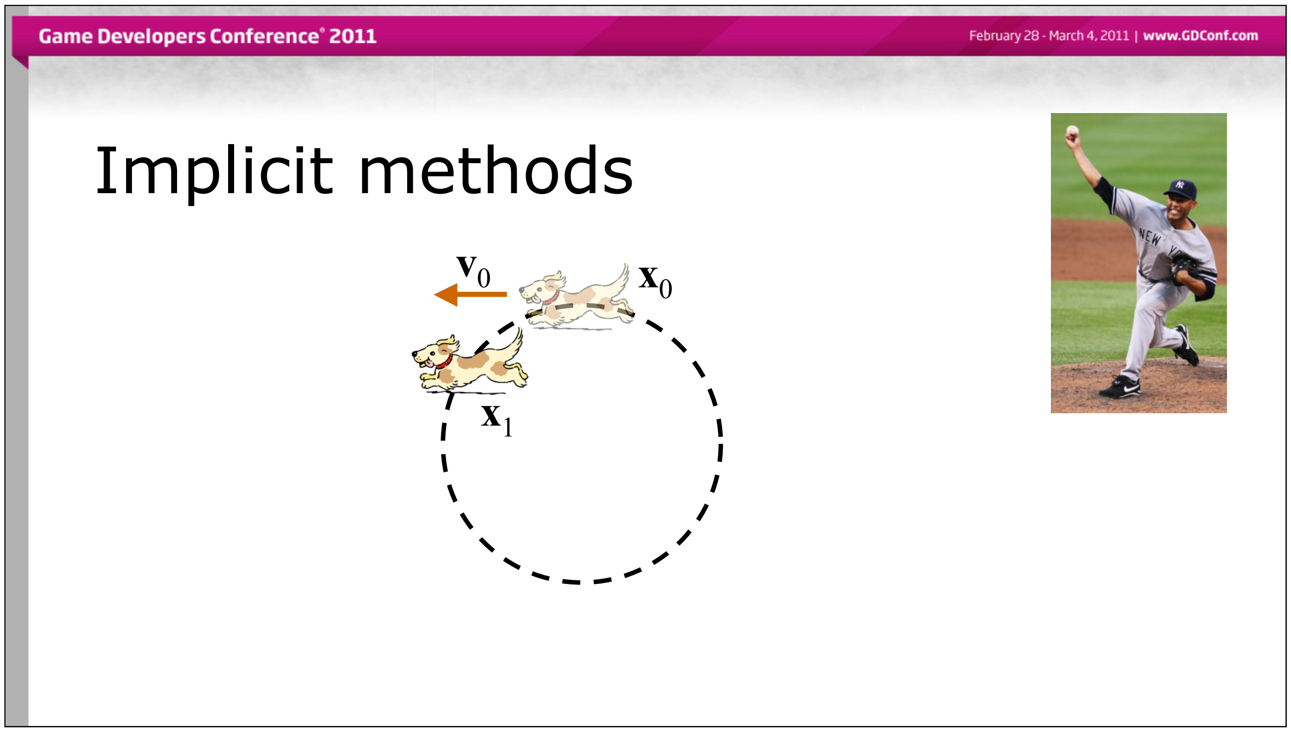
Runge-Kutta 4

- Very stable and accurate
- Conserves energy well
- But expensive: four evaluations of derivative

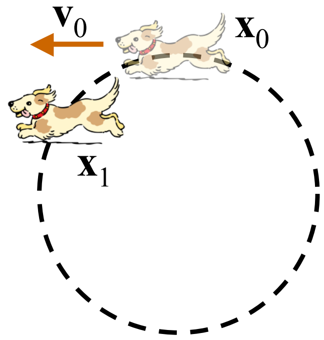
Implicit methods



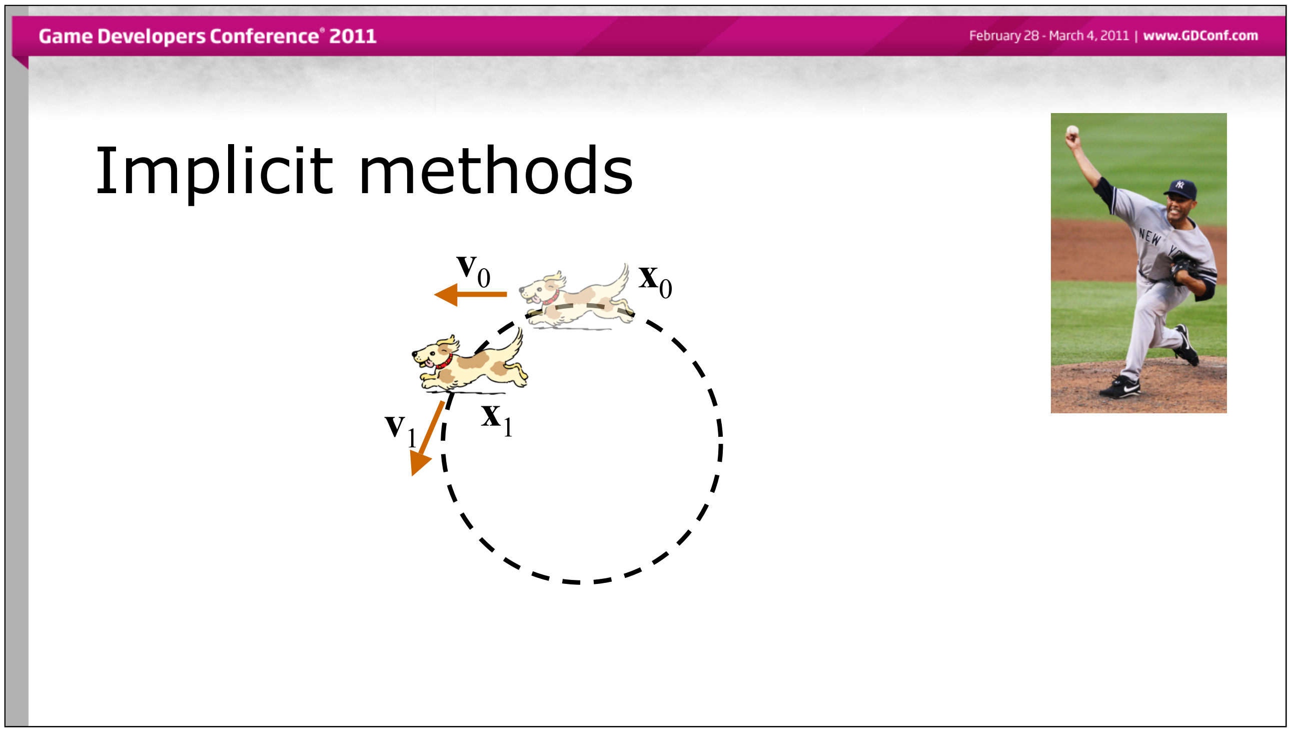
While RK4 is a very good solver, we would like something a little faster, that only takes one evaluation. One idea is rather than starting with known values, I.e. with as with the explicit solver we saw before, we use future values instead. This is known as an implicit solver. For example, we could take as our derivative the velocity at the end of the interval instead of the beginning. That’s pretty straightforward.



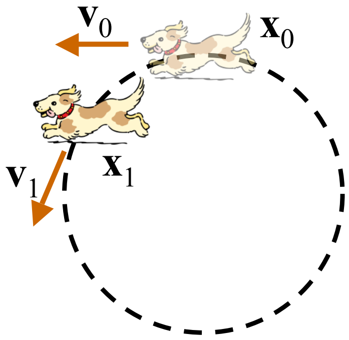
Implicit methods



We just jump to our future position,

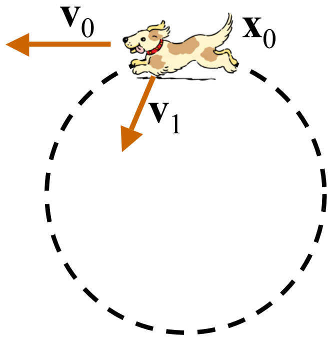


Implicit methods



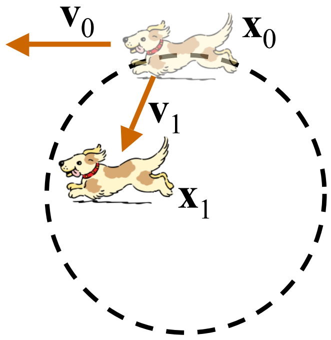
Calculate the derivative,

Implicit methods

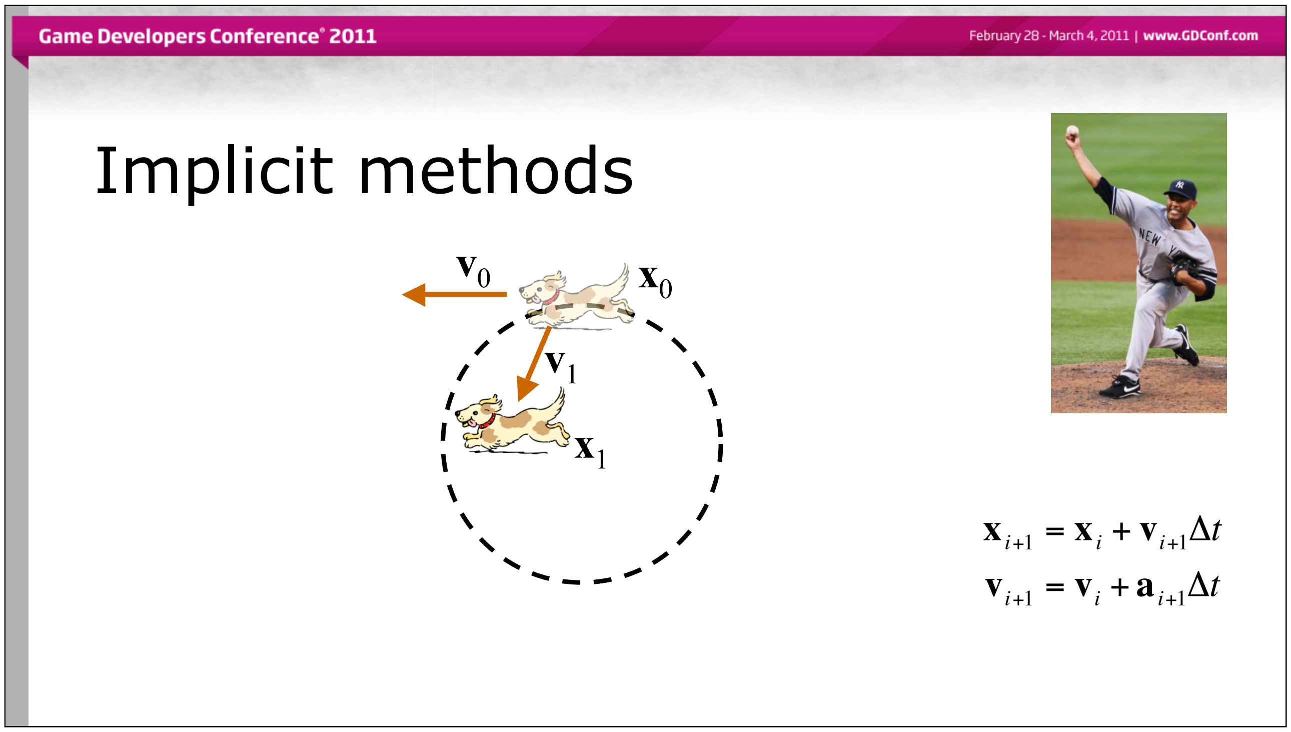


Then jump back and use that derivative

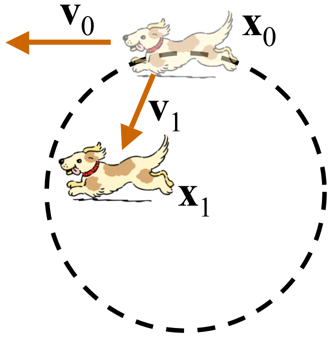
Implicit methods



To um, calculate our future position.

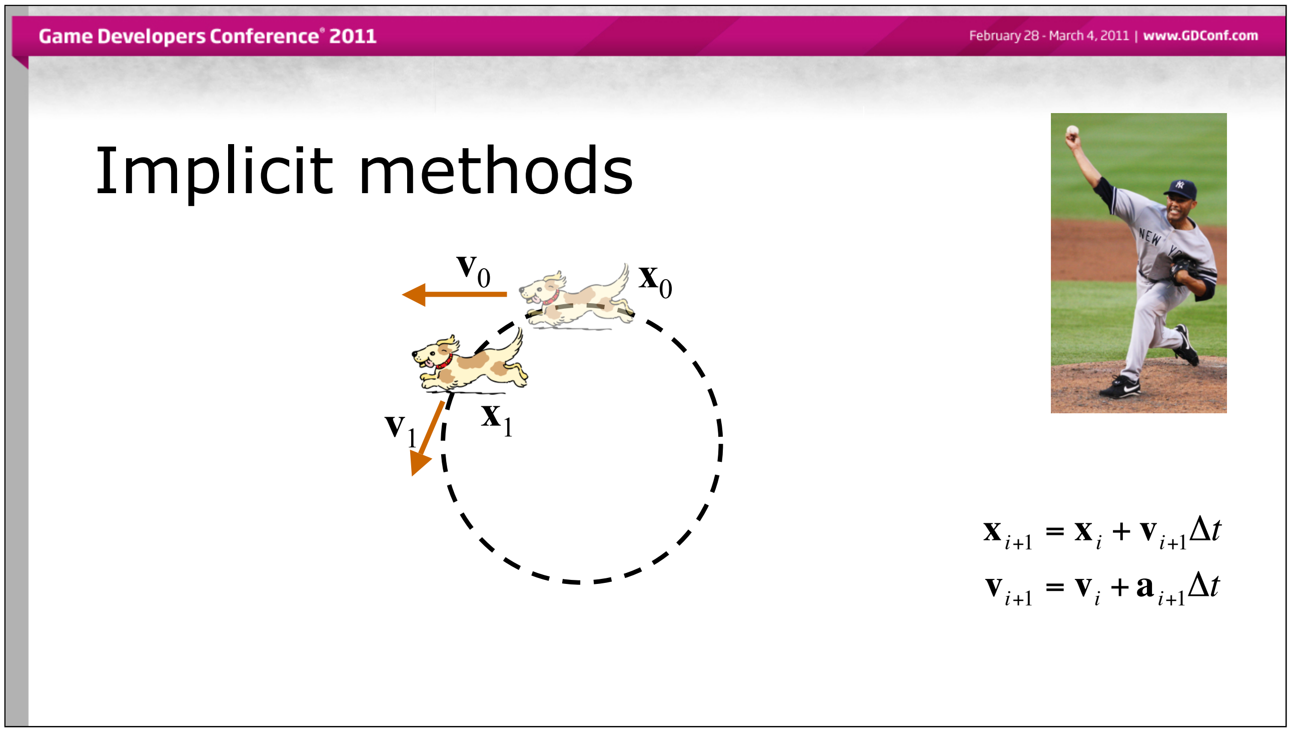


Implicit methods

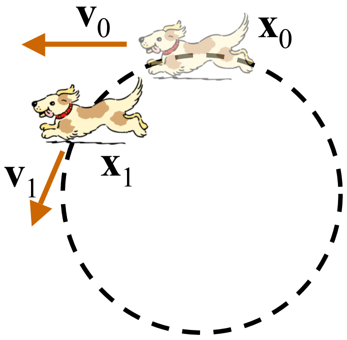


$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_{i+1}\Delta t$$
$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_{i+1}\Delta t$$

And we can represent that algebraically like this. This is known as Backward Euler. So… clearly there are some problems here.

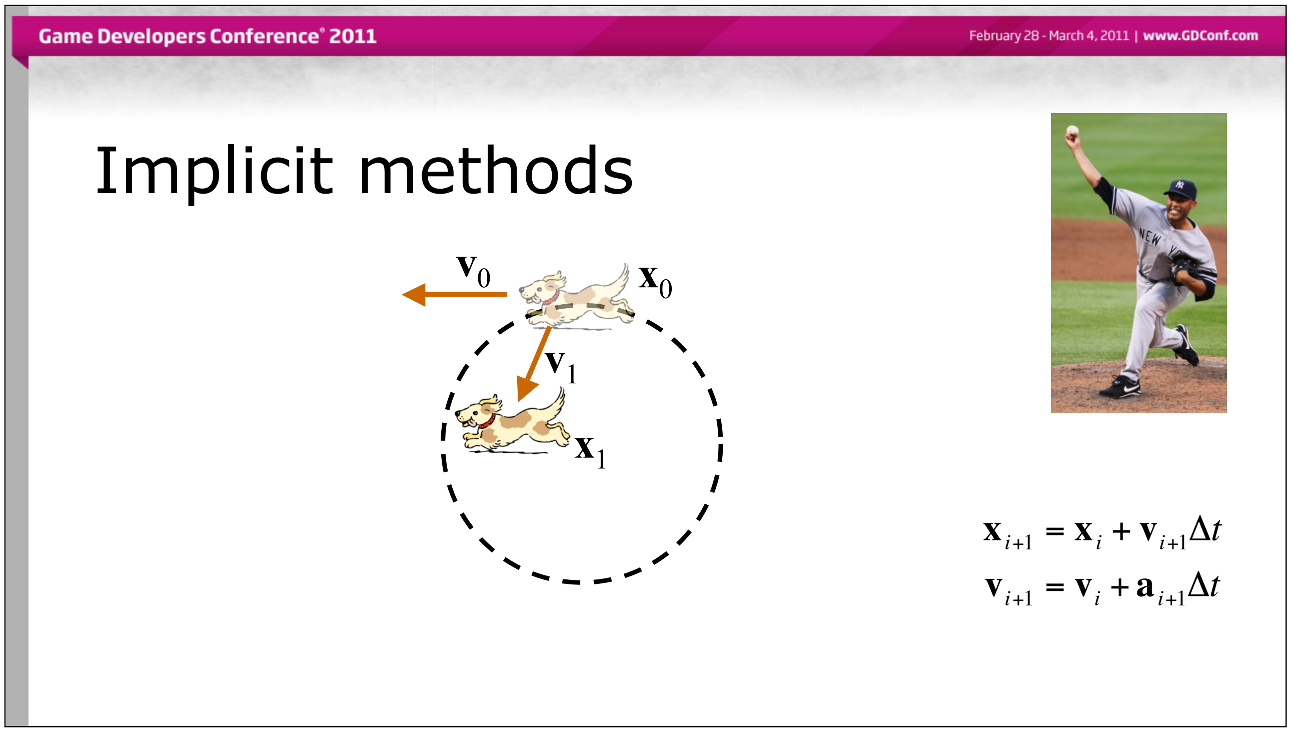


Implicit methods

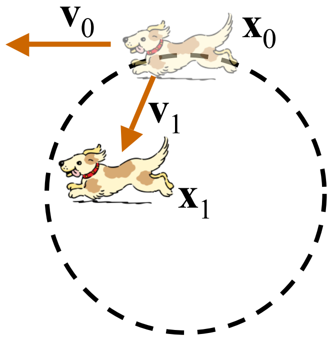


$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_{i+1}\Delta t$$
$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_{i+1}\Delta t$$

The first is, how can we know what the future values are? There are a few ways: first, if we know the system, we can try to derive an analytical solution. That's not so helpful in the general case. We can also solve this by building a sparse linear system and solving for it -- but that can be slow. Or we can use an explicit solver to step ahead, grab values there, then feed that into the implicit equation. That's known as a predictor-corrector solver: we predict a value with an implicit method, then correct with an implicit method. However, now we're back to taking at least two evaluations.



Implicit methods



$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_{i+1}\Delta t$$
$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_{i+1}\Delta t$$

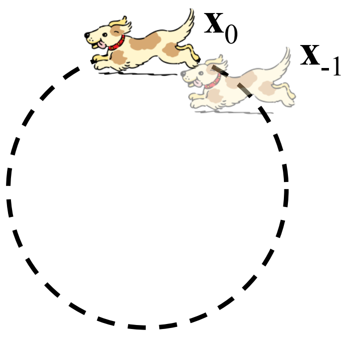
The second issue is that implicit methods don't add energy, they remove it. If I were to continue simulating the dog's movement, it would spiral into the center. This can be nice if we want to have a dampening effect, and it will not diverge, but not so good if we want to conserve energy as much as possible.

Backward Euler

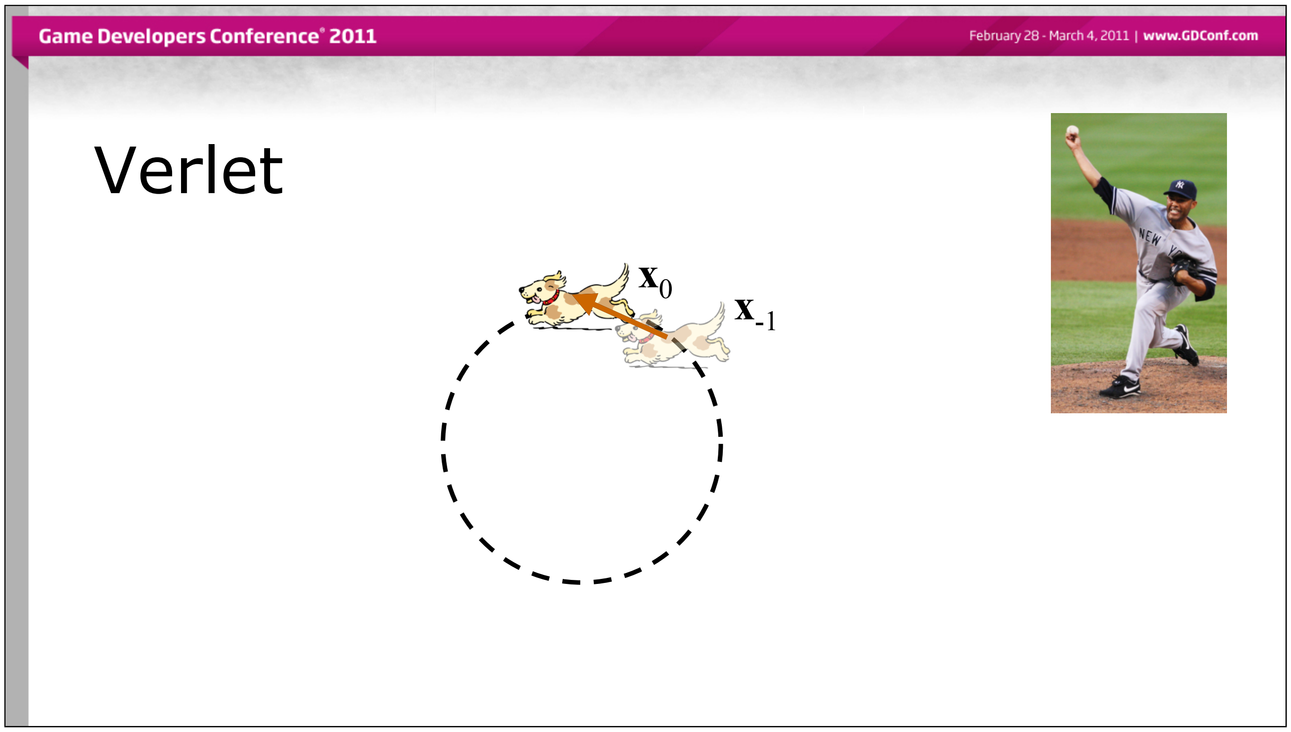
- Not easy to get implicit values
- More expensive than Euler

- But tends to converge: better but not ideal

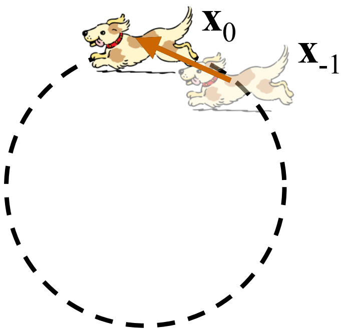
Verlet



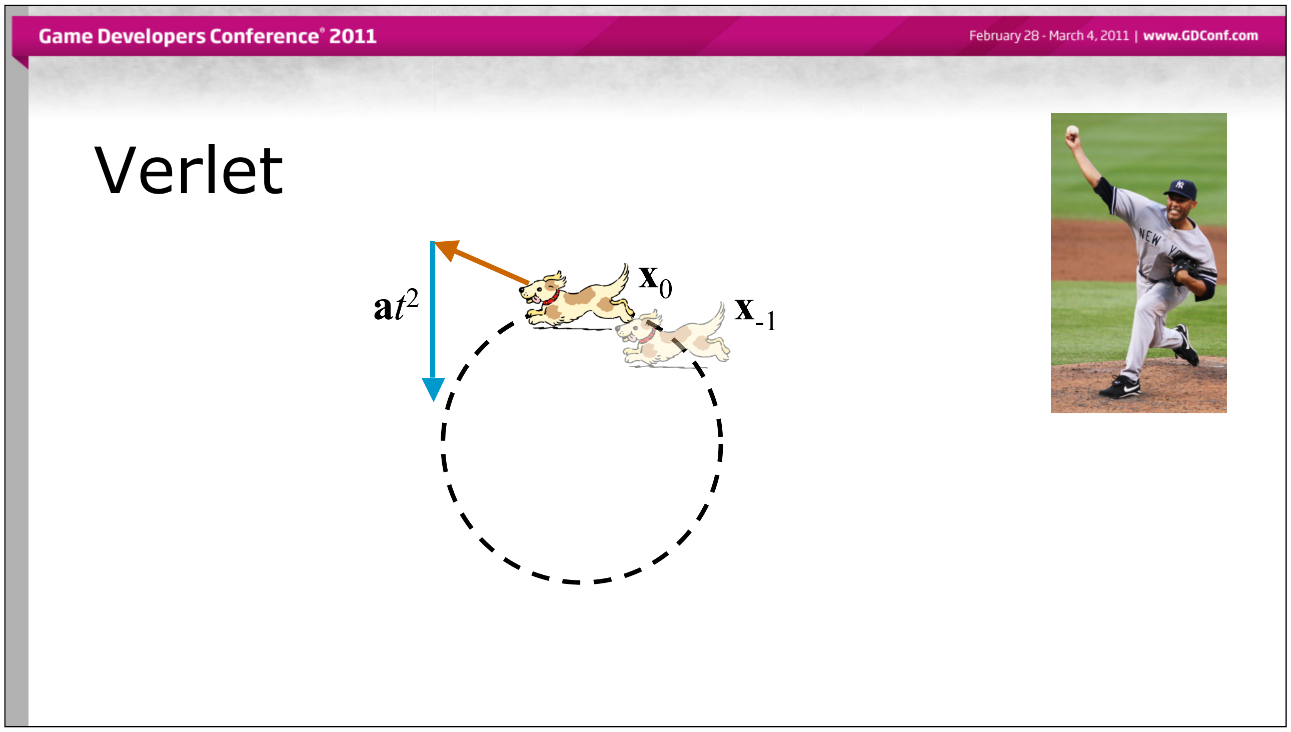
Calculating a decent velocity seems to be causing us problems, so let’s take it out of the equation. Suppose we have have a previous position, and we’ve generated our current position by running a stable solver like RK4.



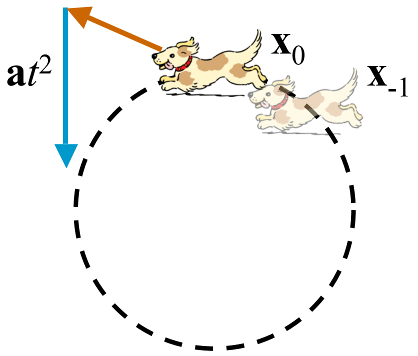
Verlet



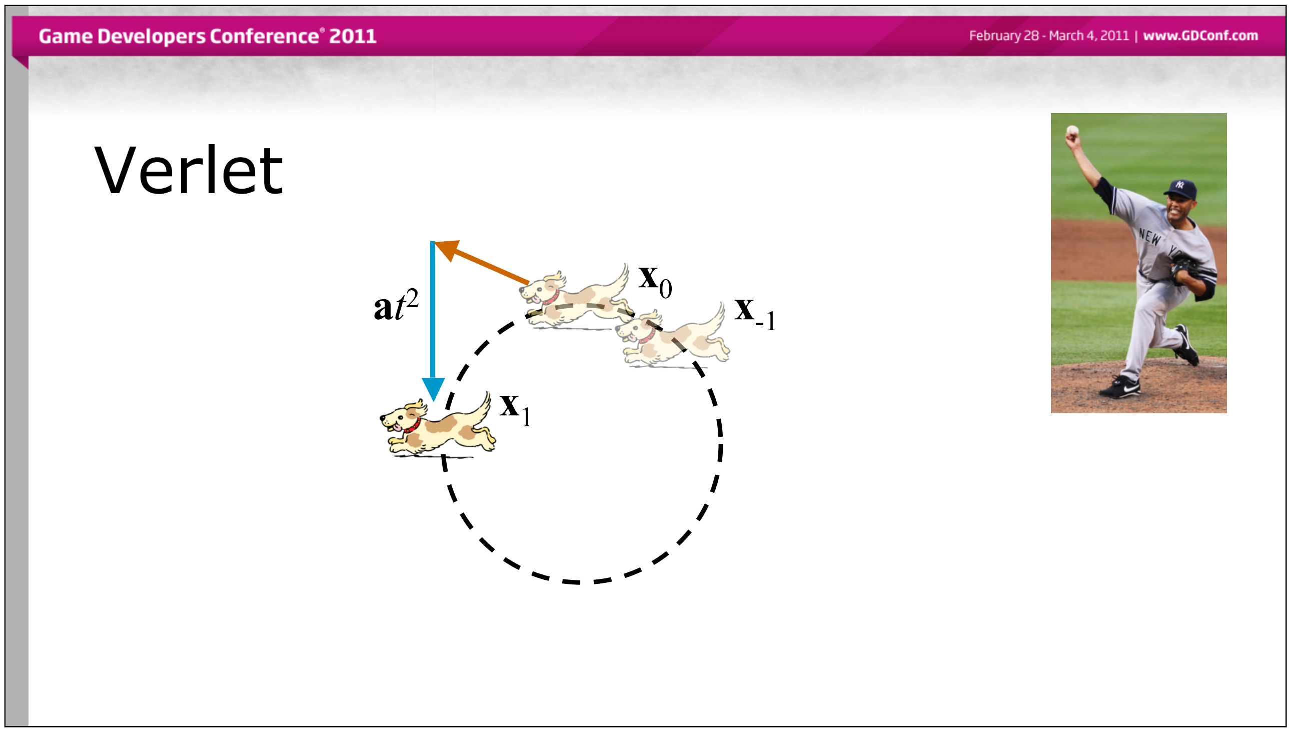
We can subtract the previous position from the current position to get an approximation to velocity.



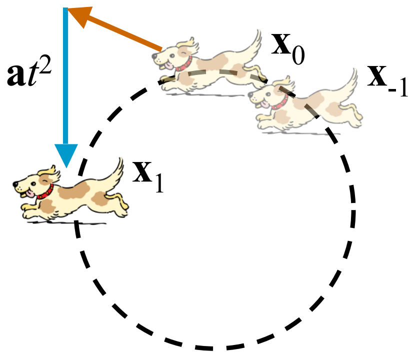
Verlet



Then add in an acceleration term.

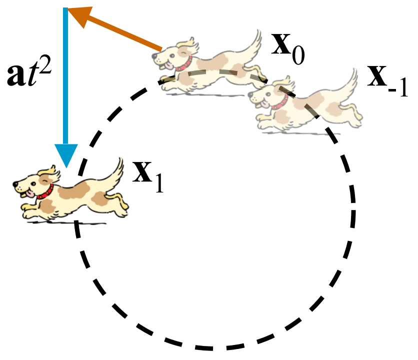


Verlet



And step.

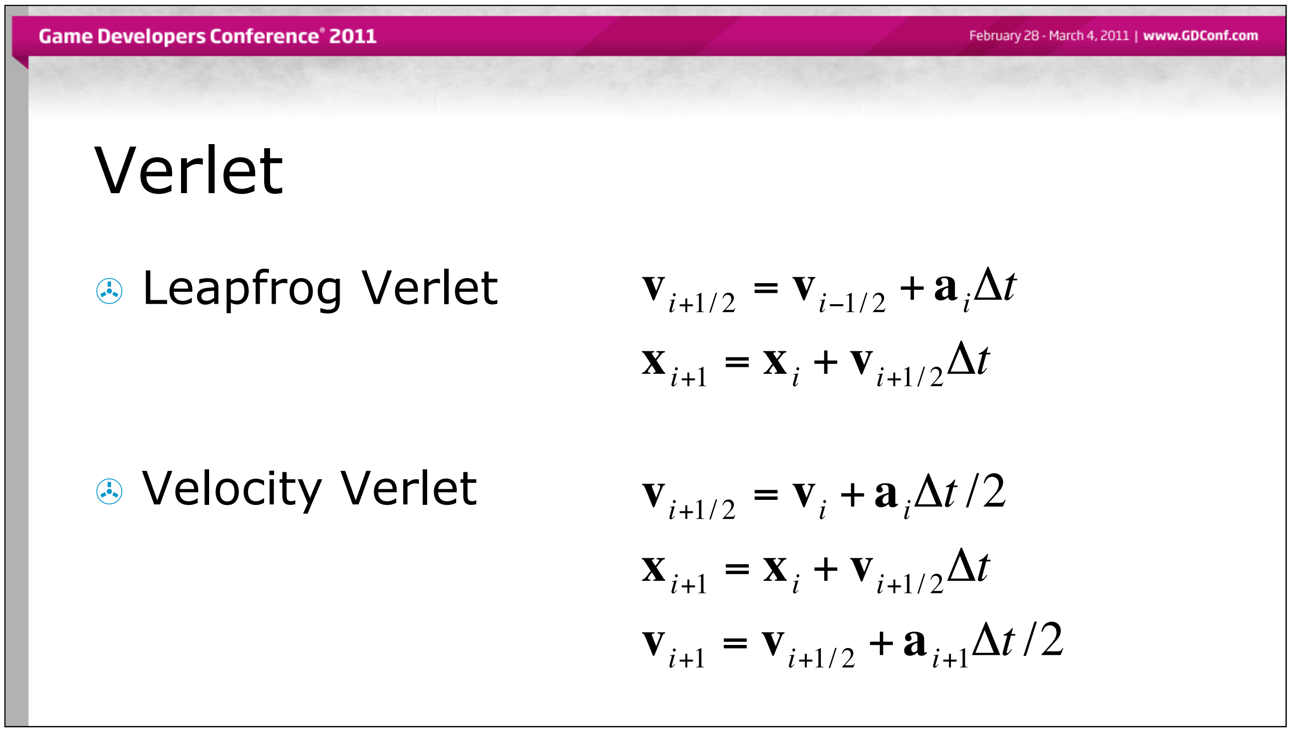
Verlet



$$\mathbf{x}_{i+1} = 2\mathbf{x}_i - \mathbf{x}_{i-1} + \mathbf{a}\Delta t^2$$

And the formula for that is this. This is known as Verlet integration.

Now I've exaggerated the result here for effect, and honestly, my scale is a bit off, but this is a very stable method. The problem is that we are approximating our velocity. If we want to do an impulse-based constraint system, where we instantaneously change velocity based on an impulse -- say for a collision, or to keep a dog attached to a pole -- we have nothing to modify. Thomas Jacobsen -- who introduced the game development community to this method -- has done some work on modifying positions in response to collision and constraint, but in my mind I find it difficult to work with.



Verlet

- Leapfrog Verlet

$$\mathbf{v}_{i+1/2} = \mathbf{v}_{i-1/2} + \mathbf{a}_i \Delta t$$
$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_{i+1/2} \Delta t$$

- Velocity Verlet

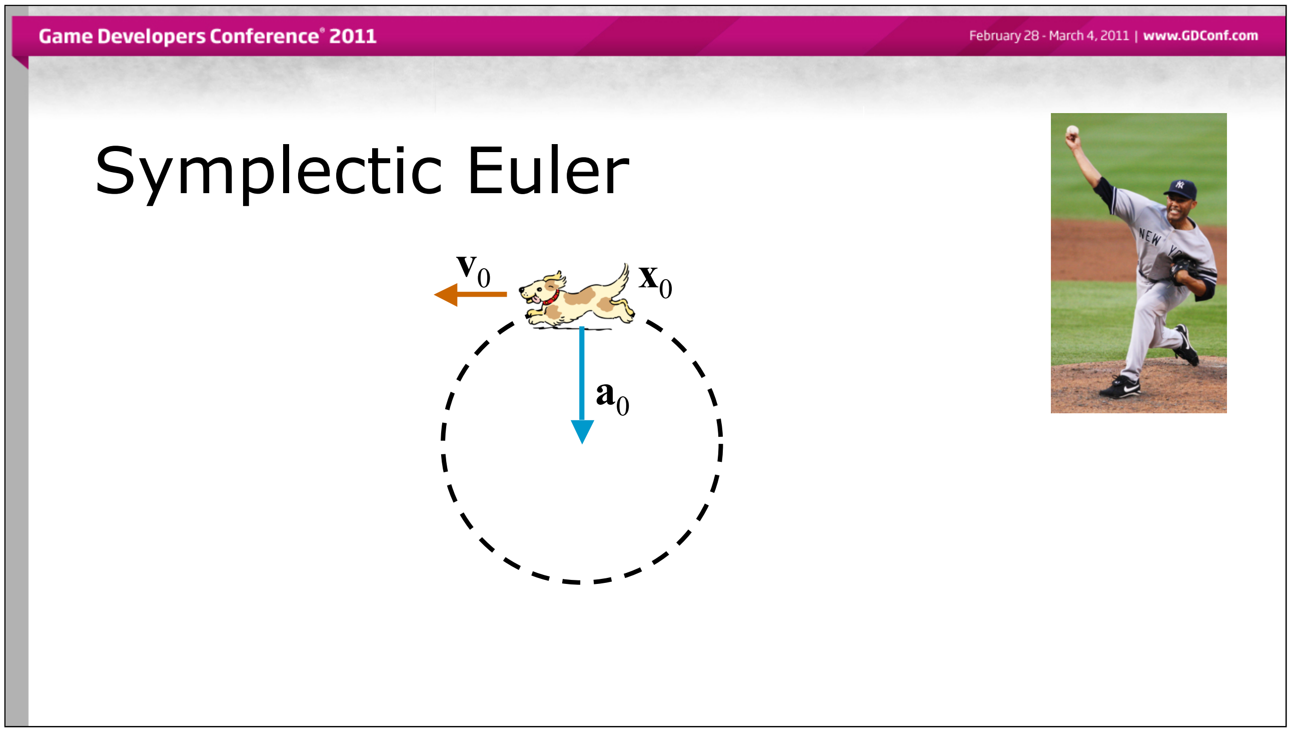
$$\mathbf{v}_{i+1/2} = \mathbf{v}_i + \mathbf{a}_i \Delta t / 2$$
$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_{i+1/2} \Delta t$$
$$\mathbf{v}_{i+1} = \mathbf{v}_{i+1/2} + \mathbf{a}_{i+1} \Delta t / 2$$

That said, there are more advanced Verlet methods that do make use of velocity, but they use a half-velocity, I.e. you start by stepping half the interval to get your velocity, and then step full intervals after that. So your velocity is out of sync with your position. The other issue is that the best of those -- Velocity Verlet -- uses two evaluations, which we're trying to avoid.

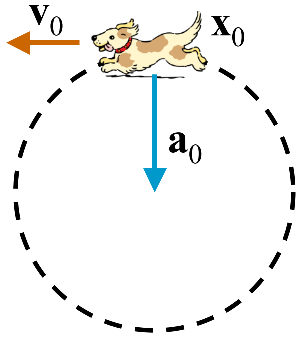
Verlet

- Very stable
- Cheap

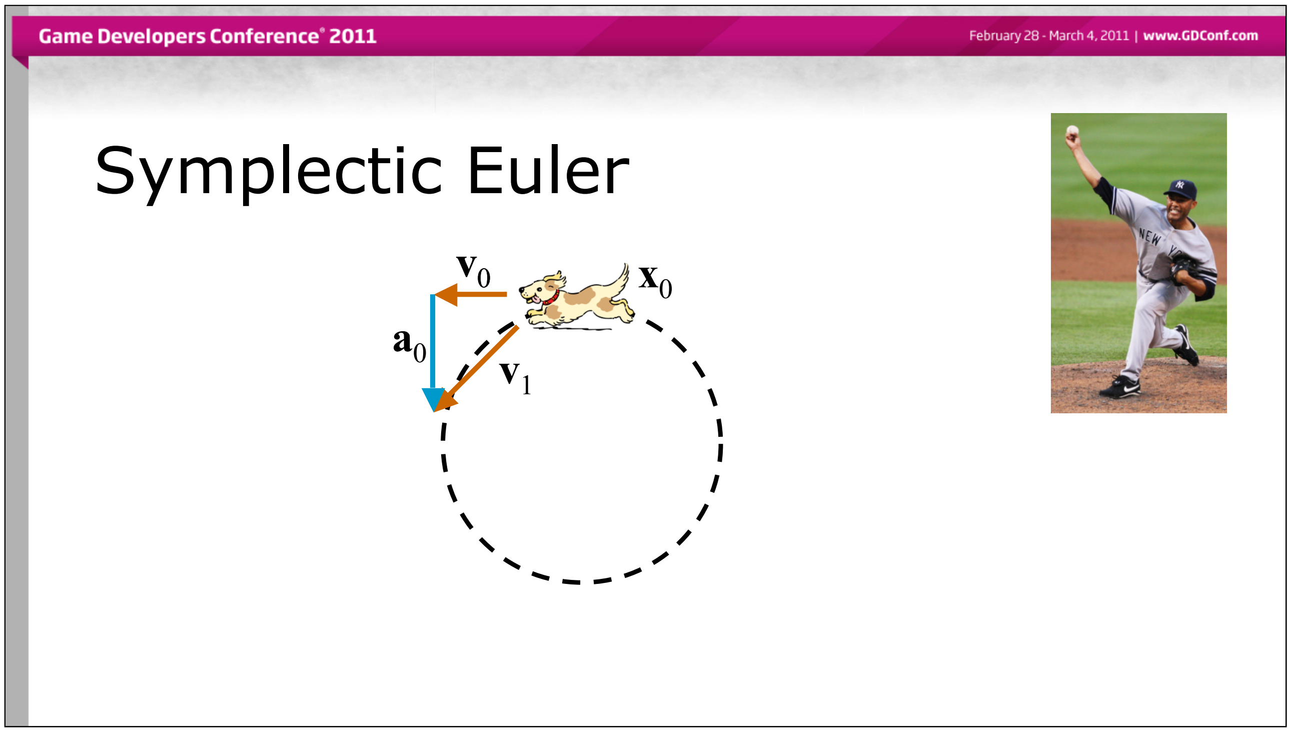
- Not too bad, but have estimated velocity



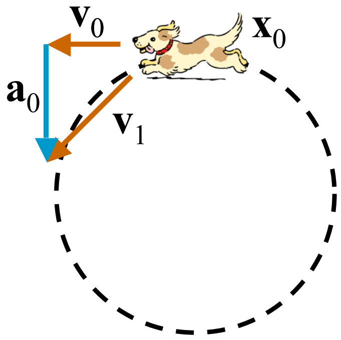
Symplectic Euler



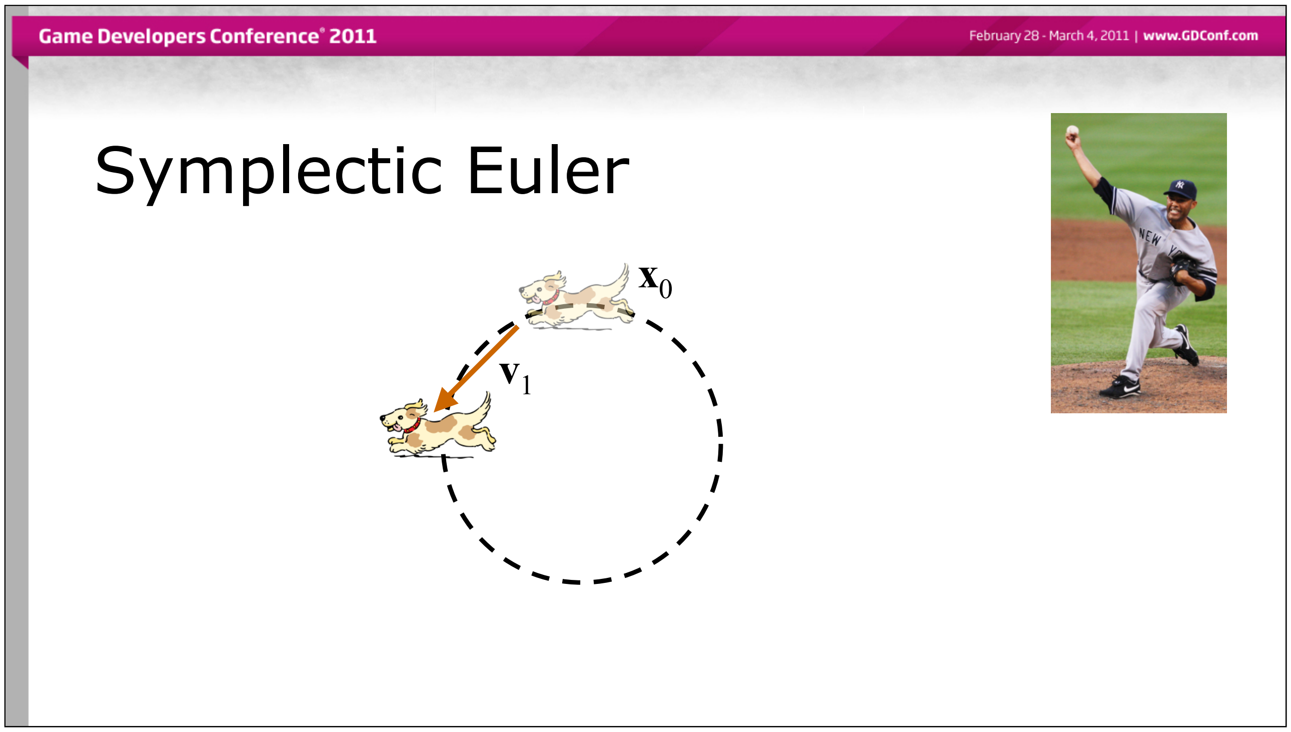
So let's go back to our Euler example again. Suppose we were to use a mix of explicit and implicit methods. That is, we will use an explicit method for solving for velocity, but use an implict method for solving for position. So we'll update our velocity first…



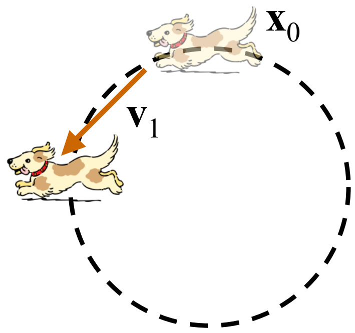
Symplectic Euler



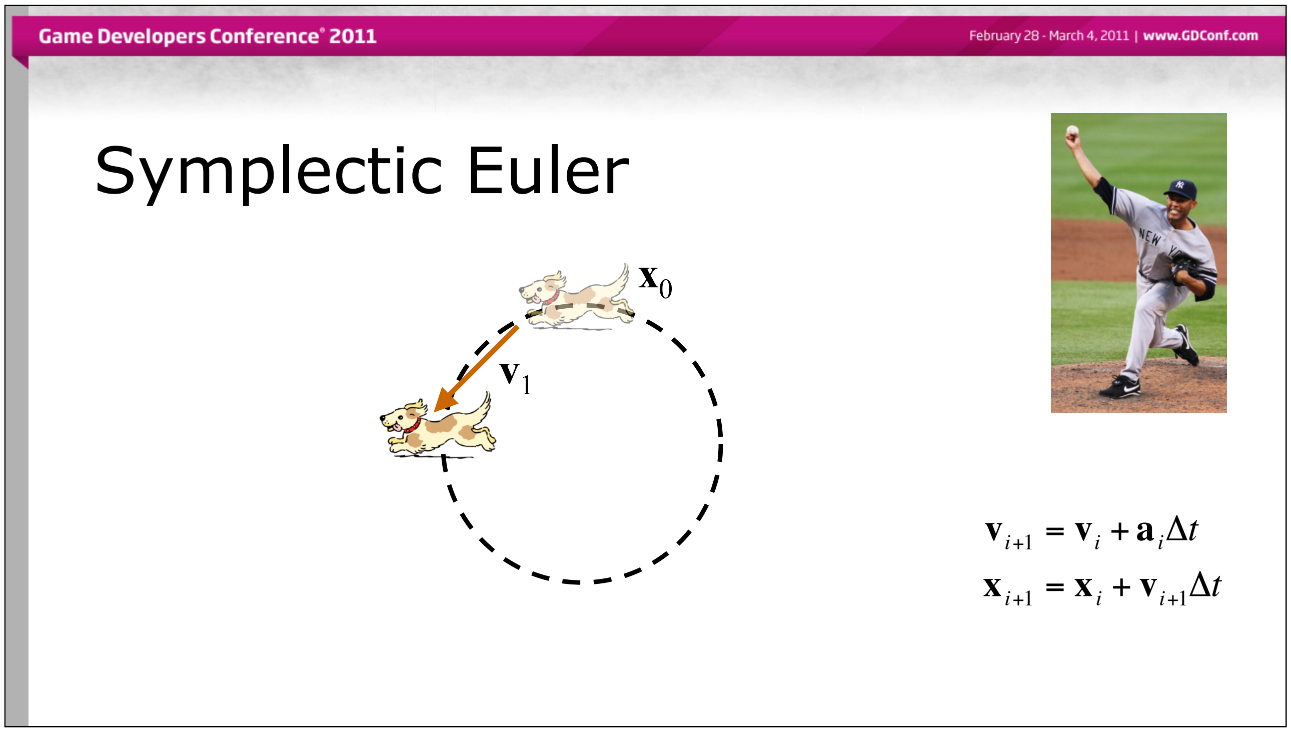
Then use that updated velocity to update position, like so:



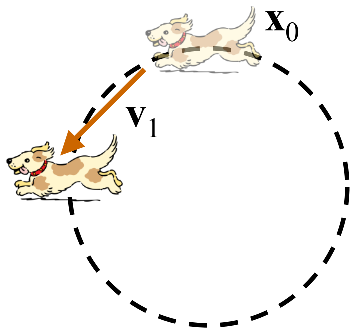
Symplectic Euler



And the formulas are:

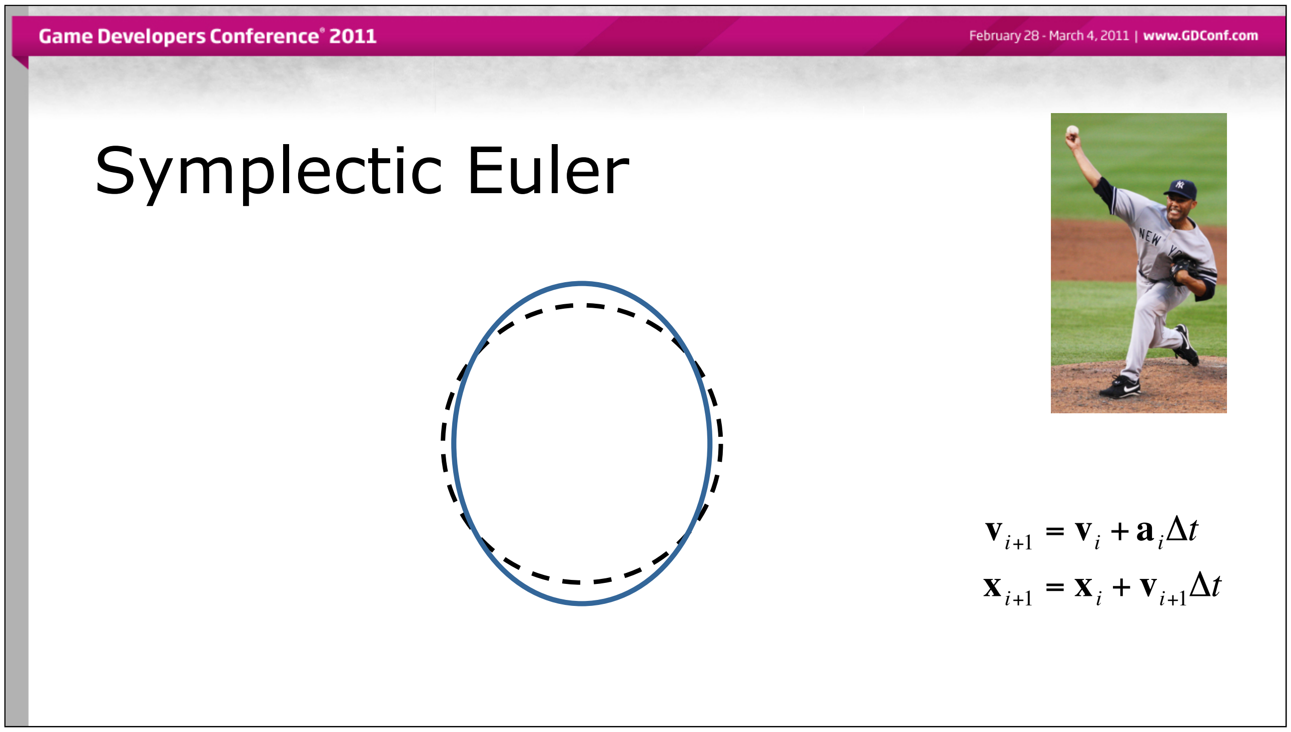


Symplectic Euler

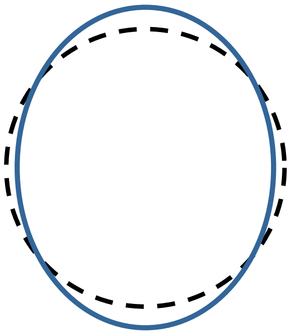


$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i \Delta t$$
$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_{i+1} \Delta t$$

That is symplectic Euler, and because we are using an explicit method for velocity and an implicit method for position, it's known as a semi-implicit method. Verlet is another example of a semi-implicit method -- in fact, symplectic Euler and Verlet are just variants of each other. The advantage of symplectic Euler is that we now have a velocity to use for impulses.

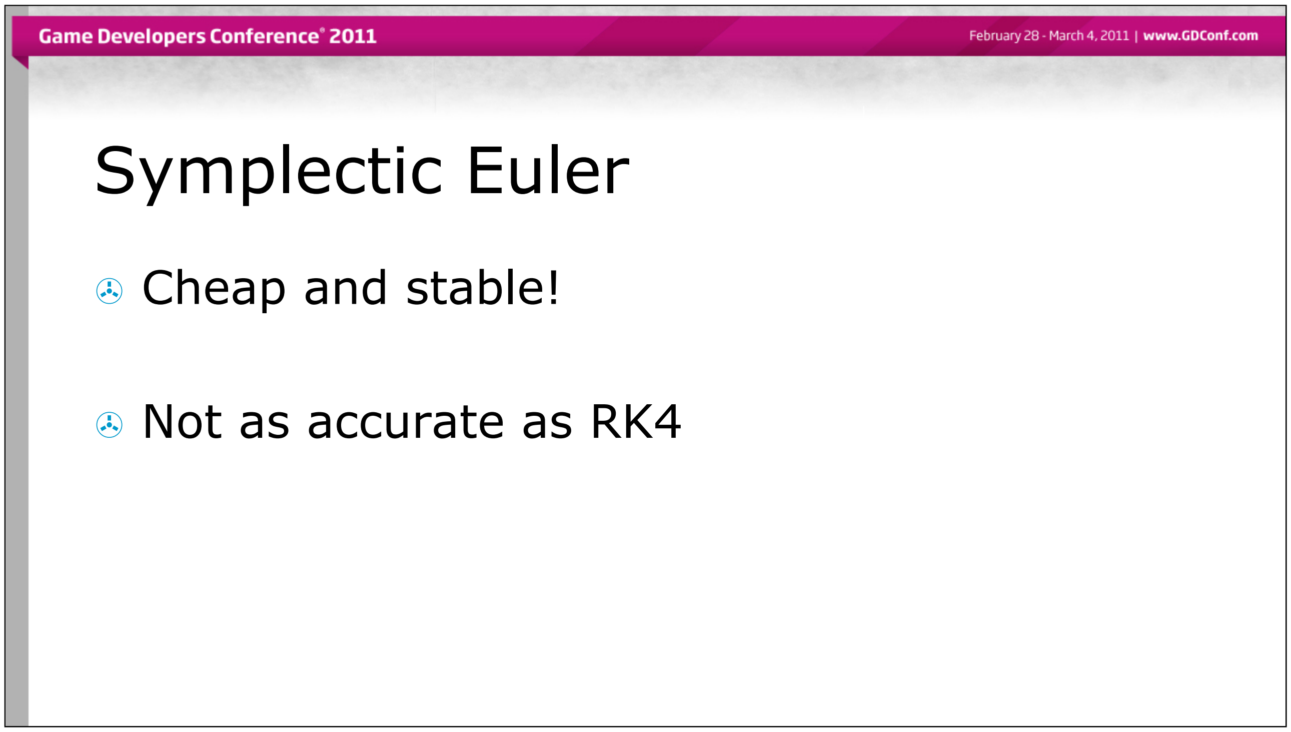


Symplectic Euler



$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i \Delta t$$
$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v}_{i+1} \Delta t$$

How does this work? Well, Symplectic Euler still has error, but it accumulates in a way that maintains stability with oscillating functions -- it accumulates in periodic fashion as well, adding and subtracting energy over time. So in our orbit example, we see the desired circular path as the dashed line. Sympletic Euler takes -- exaggerated here for effect-- an elliptical path. So while it may not be as accurate as higher-order methods, it's extremely stable. And in many cases, that's all we need.



Game Developers Conference® 2011 February 28 - March 4, 2011 | www.GDConf.com

Symplectic Euler

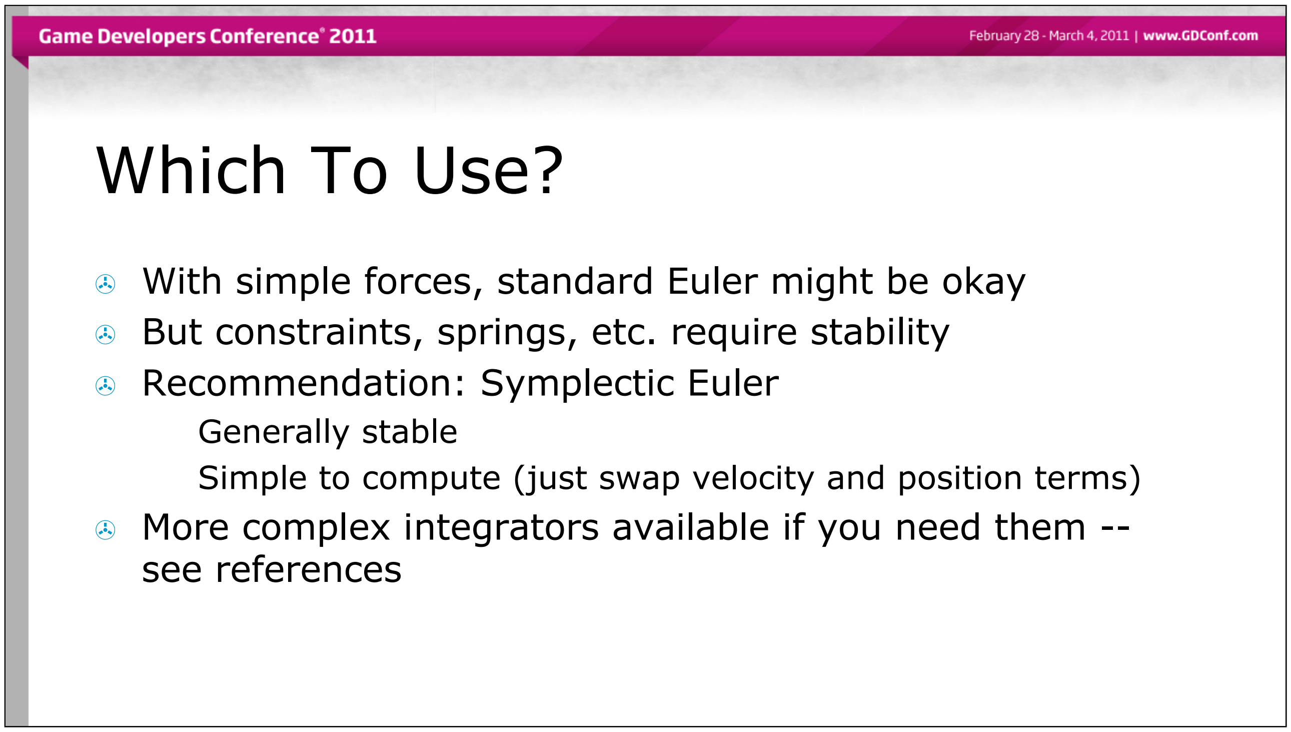
- Cheap and stable!
- Not as accurate as RK4

Strictly speaking, Symplectic Euler can still diverge for large time steps or stiff equations, but it's more stable than many other methods under common conditions.

Symplectic Euler

- Cheap and stable!
- Not as accurate as RK4 - but hey, it's a game

Demo Time



Which To Use?

- With simple forces, standard Euler might be okay
- But constraints, springs, etc. require stability
- Recommendation: Symplectic Euler
  - Generally stable
  - Simple to compute (just swap velocity and position terms)
- More complex integrators available if you need them -- see references

Implicit Euler is best if you want strict stability and don't mind the dampening effect.

References

- Burden, Richard L. and J. Douglas Faires, *Numerical Analysis*, PWS Publishing Company, Boston, MA, 1993.
- Witken, Andrew, David Baraff, Michael Kass, SIGGRAPH Course Notes, *Physically Based Modelling*, SIGGRAPH 2002.
- Eberly, David, *Game Physics*, Morgan Kaufmann, 2003.

References

- Hairer, et al, "Geometric Numerical Integration Illustrated by the Störmer/Verlet method," *Acta Numerica* (2003), pp 1-51.
- Robert Bridson, Notes from CPSC 533d: Animation Physics, University of BC.