3.4: Using Computation to Simulate Motion

In Section 3.1.1, Equation 3.1.1 we saw that under constant velocity motion we can describe a future position of an object using the current position, the velocity, and the amount of time to progress into the future.

$$x(t) = x_o + v_x t \tag{3.4.1}$$

We can model this motion on computer. We will use the Python language in this course, and in particular, we will use Visual Python so that we can apply physics to objects that visualize physical motion in 3D. To get started, we first need to define an object that will move through space. We will make a sphere call it `ball`. The ball object can beplaced at a position using a vector `pos=vec(x,y,z)`.

```
ball = sphere(pos=vec(0,0,0), color=color.green, radius=0.1)
```

We can define the velocity to be a vector along the x-direction also using the built-in vector function. `velocity=vec(vx, vy, vz)`.

```
velocity = vec(0.5, 0, 0)
```

Alternatively, it is possible to make the velocity one of the attributes of the ball object. We should be careful to only use this method when the attribute is a property of the object.

```
ball.vel = vec(0.5, 0, 0)
```

We can visualize the velocity with a vector arrow that remains attached to the ball's position and has a length that is the magnitude of the velocity. To do this we write the following code.

```
arrow(pos=ball.pos, axis=ball.vel, color=color.white)
```

Try, putting these lines of code in the trinket below and see what happens when you run the program.

trinket Code Run Help Share

main.py

Remix

Once we have objects, we can apply physical principles to make them move. One way to make the ball move is to repeatedly update the position using the equation above. For example, we could write lines of code that take the initial position x_o and move it forward with some amount $v_x t$. Since we are in a 3D world, we will do this with vectors where only the x-direction is changing from the description of position and velocity above. Suppose we want to take snapshots every second, i.e., `t = 0, 1, 2, 3,...`

```
r_0 = ball.pos
ball.pos = r_0 + ball.vel * 0
ball.pos = r_0 + ball.vel * 1
ball.pos = r_0 + ball.vel * 2
ball.pos = r_0 + ball.vel * 3
ball.pos = r_0 + ball.vel * 4
```

Copy the code above that creates the ball, and add these lines in the trinket below.

You will see that your program runs all of the lines instantly, and you do not see the motion. Slow the computation down by adding the command `rate(1)` between each line updating the ball's position. This will delay each line by 1 second. For example,

```
ball.pos = r_0 + ball.vel * 0
rate(1)
ball.pos = r_0 + ball.vel * 1
rate(1)
```

Hopefully it is clear to you this is an inefficient way to write a program, especially if we want to run many iterations of changing the ball's position. To make the process more streamlined, we use loops. In this case, we will use a while loop. The while loop runs "while" a condition is met. For example, we can run the ball simulation while the time is less than 10 seconds and update every 1 second. First, initialize the variable for time `t` to zero and the time increment `dt` to 1 second. Express the while loop as while `t<10:`, and in the while loop, update the position and time.

```
t = 0
dt = 1
```

```
while t<10:
    ball.pos = r_0 + ball.vel * t
    t = t + dt
```

Put all of this together, and your code will look like the following. Comments have been added to the code so that each line has a description. The comments after # are ignored when the program runs.

```
ball = sphere(pos=vec(0,0,0), color=color.green, radius=0.1)
ball.vel = vec(0.5, 0, 0)
arrow(pos=ball.pos, axis=ball.vel, color=color.white)
r_0 = ball.pos
t = 0
dt = 1
while t<10:
    rate(1)
    ball.pos = r_0 + ball.vel * t
    arrow(pos=ball.pos, axis=ball.vel, color=color.white)
    t = t + dt
```

Paste this code below and see the ball move at constant velocity.