____

Name

Recitation Assignment # 2

October 3, 2007

Today we're going to do an actual simulation of a very simple physical system. It will be your first real introduction to numerical coding, I suspect, and so perhaps this will be a bit of a trial by fire. Don't worry, Travis is here to help, and you'll get the hang of it in no time.

You may complete this in class. However, if you are unable to do so, it is expected that you complete this for recitation next time.

Where a box appears, call over Travis to check over your progress. When the sheet is complete, you will hand it in. Also, you are expected to email your final programs to Travis: *hoppe@drexel.edu* .

Today we're going to make a model of projectile motion: a ball thrown through the air. We will use the following new concepts:

Programming:

- Importing an external library
- Creating geometric figures in VPython
- The print statement
- "While" loops
- Vector math

Physics:

- A particle moving under constant acceleration.
- $\Delta\vec{p} = \vec{F}\Delta t$
- $\Delta\vec{r} = \frac{\vec{p}}{m}\Delta t$

Each of these concepts will be important in later programs, so make sure you master them here. You will find it very useful to have an open copy of the VPython webpage (vpython.org), while you work.

1. Log into a workstation, pull up firefox, make yourself comfortable, etc.
2. Open up a "terminal" or "shell," and, if you have not already done so, create a directory called contemporary1. e.g.:
   ```
   % mkdir contemporary1
   ```
   change to the new directory as you learned last week.

3. Now you are ready to start! Run
   ```
   % emacs prog2.py &
   ```
   This will pull up the emacs file editor. (The "&" allows you to keep using the shell while emacs is running).

☐ 4. Write the following simple program and run it (by typing "python prog2.py" on the command line after you've saved the program in emacs):

   ```
   from visual import *
   xaxis=box(length=10,height=0.1,width=3)
   yaxis=box(length=0.1,height=10,width=3)
   scene.autoscale=0
   ```

   What does this do? Let's go over it line-by-line.

   - `from visual import *` – This line opens up an external library called "visual." It basically tells python to make all of the functions in visual (including those which make boxes and spheres and so on) available to you. **Every** program you write in this class will begin with this line.
   - `xaxis=box(length=10,height=0.1,width=3)` – This is your first visual command. Basically, it creates a long, flat box, and calls it "xaxis". The elements in it should be self-explanatory. The box is 10 units long (call them meters, if you like), 0.1 m high, and 3 meters wide. It is also centered at the origin, by default, though in later exercises, we may change that. If you look at the vpython webpage, you can see all of the other options you can specify for a "box", including the color.

     Why do we call it "xaxis"? Well, we may want to refer to this object later in the program, and we have to call it something. Let's say you forgot how tall the xaxis was, and you wanted to know. You could add the line (to the end of your program):

     ```
     print xaxis.height
     ```

     The print statement prints things to the terminal. Note that this command must appear *after* you've created the "xaxis" object, or the program won't know what you are talking about.

     Note also the weird syntax. "xaxis.height", literally means "the height of the xaxis." How do you suppose you'd print out the width of the xaxis?
   - `yaxis=box(length=0.1,height=10,width=3)`

     This command is almost identical to the xaxis, but creates (you guessed it) a yaxis.
   - `scene.autoscale=0`

     VPython tries to fit everything on the screen, and therefore it autoscales continuously. That can get annoying if you are going to move things around (like the ball later on), and don't want your perspective to keep shifting. This line prevents that from happening. Make sure you put this line in *after* you've created all of your objects.

     Oh, and if you don't believe me that it can get annoying to have your perspective shift in the middle of your program, try deleting this line later, and see how it looks.

5. Play around with the mouse. Hold the left, right, or both buttons while moving in the image window. See how your perspective changes.

6. Now it's time to add the ball. Create a sphere called "ball" and put it at the origin.

   Okay, here's a start:

   ```
   ball=sphere(pos=vector(0,0,0))
   ```

   This does exactly what you'd expect. Note that "pos" is a special name and VPython knows that it represents the position of the object. Notice that, unlike width or height, "pos" is a vector, not just a number. We'll see how vectors work later.

   Also, give it another parameter, m=5. Now, you and I know that "m" probably stands for mass, but Python doesn't. We'll have to add the physics later. This can be done in two ways.

   1. `ball=sphere(pos=vector(0,0,0),m=5)`

   or

   2. `ball=sphere(pos=vector(0,0,0))`

   `ball.m=5`

   Either one works.

   Additionally:

   This ball is way too big! Look at the documentation, and figure out how to give the ball a radius of 0.2.

   Make the ball red colored.

   Give the ball another attribute called "p" (which we will make the momentum), and set it to vector(20,20).

7. We need to set a bunch of numbers to be used in the program. I am going to give them names I like. You should be aware that you can call them whatever you want and the program will still work. However, they need to be *declared* before you use them!

   ```
   dt=0.1
   tmax=10.
   t=0.
   g=9.8
   ```

   All I've done is define 4 numbers. What are they going to stand for? *dt* will represent the amount of time between frames of our simulation. *tmax* tells the simulation how long to run. $t = 0$ tells the program what time to start at. $g = 9.8$ will be used to put gravity into the code.

   But does this mean that there is physics? No. The computer has no idea how gravity works. We're going to need to tell it.

☐ 8. Now we'll create a clock. First, you'll need a "while command".

   ```
   while (t < tmax) :
   ```

   What this does is it runs every command underneath again and again until whatever is in the parentheses is satisfied. Note that only the *indented* commands under the while will be run. For example, try putting in the following:

```
while (t < tmax) :
      print t

print ‘‘Now I am done.’’
```

Run it. See a bunch of 0's? Why won't the program end?? Because $t = 0$, and $tmax = 10$, and 0 is always less than 10.

If the program keeps running forever, you may want to hit ctrl-c to stop it.

If, instead, you change your loop to:

```
while (t < tmax) :
      t=t+dt
      print t

print ‘‘Now I am done.’’
```

You will note that in each step, the clock advances 0.1 until it hits 10, and then it tells you that it's done.

But wait! This loop was much faster than 10 seconds, wasn't it? If you want the clock to run in real time, you need the "rate" command, which will tell the loop the maximum number of times to run each second. Add:

`rate(1./dt)`

to your code right under the start of the while loop (but indented). Think about it: if dt=0.1, then 1./dt=10, or the loop should go 10 times per second, which is what we want!

☐ 9. Time to add some physics. Before the loop, I want you to define a vector with the x and z parts equal to 0, and the y part equal to -g. You should call this vector "a". Note: you must create this vector *after* you've told the program what "g" is.

Next, within the loop, update the position of the ball each time-step. The equation can be written as:

$$\vec{r}_{new} = \vec{r}_{old} + \frac{\vec{p}}{m}\Delta t$$

What does this mean? Well, we are creating a movie, and this equation tells us where the ball should be in each successive frame. Now, in computer programming, the "new" and "old" positions are held in the same variable position: `ball.pos`, which updates the position of the ball on the screen each time it is changed.

Likewise, the momentum of the ball is : `ball.p`

and so on. Turn the equation above into a single command, and run the program.

What happens? Well, the ball moves, but it doesn't fall, as you might expect. That's because the momentum doesn't change. You also need:

$$\vec{p}_{new} = \vec{p}_{old} + \vec{F}\Delta t$$

where

$$\vec{F} = m\vec{a}$$

as you may have heard.

You should write two lines (under the loop) to update the momentum of the ball.

10. It's easy to lose track of the ball. We may want to add a "trail" to indicate where it's been. Above the loop, add the line:

    `trail=curve(color=color.red)`

    This does nothing initially. However, every time you execute the command `trail.append(ball.pos)`

    a new line segment is added, terminating at the current position of the ball, creating a continuous curve. Add the above line below the loop.

11. Some physics. Using a pen and paper, and keeping $p_y = 20kg\ m/s$, compute p_x such that the ball will travel exactly 5m (to the edge of the x-axis) before hitting the ground. Put in your test value into the code and see if theory and numerics agree.

12. **E.C.** See if you can edit your code so that the loop ends the moment that the ball crosses the x-axis (e.g. when it hits the ground).

13. **E.E.C.** See if you can edit your code so that the ball *bounces* when it hits the ground!