Goals:
Type in the starter code (no cut and paste) to acclimate to Python and glowscript.
Modify the starter code in the following ways:

- Add in additional charges
- Allow some or all of the charges to move

Ready to code? Might as well dive in…
Go to http://vpython.org/ and read the page where it says "How to Get Started".

Following the instructions, I went to glowscript.org. I clicked sign in and it let me sign in with google id. I don't particular like having to use a google Id but I figured I wanted this done so I did it. I understand if you are morally opposed. Once I got signed in I first wrote the cheesiest program ever:

```
box()
```

I then ran the code. Once you hit run, try swiveling the point of view by holding down the right mouse button and moving the mouse. Try zooming by holding both mouse buttons and moving the mouse. Try changing the size of the black window by dragging on the bottom right corner. Note: you might also click on the HELP link in the upper right corner of the glowscript page. This should link you to a page with all kinds of documentation on different types of objects.

Starter Code
The Coulomb force between two charged balls is very similar to the gravitational force.

Gravitational	Coulomb
$\vec{F}_{1on2} = \dfrac{Gm_1m_2}{r_{12}^2}(-\hat{r}_{1to2})$	$\vec{F}_{1on2} = \dfrac{kq_1q_2}{r_{12}^2}(\hat{r}_{1to2})$

The Coulomb constant is $k = 8.99\frac{\text{N}\cdot\text{m}^2}{\text{C}^2}$ and the charges q_1 and q_2 are given in Coulombs (C).

On the following page is an example of some code (**PointChargeForces01NoMotion**) I wrote up. The code draws two point charges at positions specified in the code. It chooses colors for the charges based on the sign of the charges in the code. It also determines the Coulomb force acting on one of the charges and draws an arrow. A scale factor is included so you can adjust the size of the arrow as desired.

INTRO VID: To make this handout easier to understand, watch this video before going on.
https://www.youtube.com/watch?v=CNxTrfLr6sw&feature=youtu.be

Type in this code without using cut and paste. For those of you less familiar with coding, the # is used for making comments in the code. These lines of code are not used by the computer. They are there to help humans understand as they read/edit the code. You can shorten the comments if desired to save time.
Tip: I would hit run every few lines while you are initially typing. This makes it much easier to debug.
Tip: to comment sections of code in bulk in glowscript, highlight all desired lines then use "CTRL /".

Once the code is typed and working, try editing the charges and positions to verify the code makes sense to you. Try both positive charges, both negative, and mixed sign cases.

- Try larger charges versus smaller.
- Try closer versus farther apart.
- Try putting both charges on the horizontal axis.
- Then change the position to put the charges on a vertical axis.

CODE BELOW: PointChargeForces01NoMotion (there is a space between import and * in the first line of code)

```
GlowScript 2.7 VPython
from visual import *

#scale_arrow is a number that will allow you to change the size of force arrows
scale_arrow=1

#q's are charges in coulombs
q1 = 1.0e-4   #0.1 mC
q2 = -2.0e-4  #0.2 mC

#m's are masses in kilograms
m1 = 1
m2 = 2000

#the coulomb constant is k
k = 8.99e9

#r's are initial locations of charges
r1 = vec(5, 5, 0)
r2 = vec(0, 0, 0)

#p's are initial momentums of charges
p1 = vec(3.3, -3.3, 0)
p2 = vec(0, 0, 0)

#make positive charges red and negative charges blue-green
if q1>0:
   c1=vec(1, 0, 0)
else:
   c1=vec(0, 0.6, 1)
ball1 = sphere(pos=r1, radius=0.3, color=c1)

if q2>0:
   c2=vec(1, 0, 0)
else:
   c2=vec(0, 0.6, 1)
ball2 = sphere(pos=r2, radius=1, color=c2)

#determine the coulomb force q2 exerts ON q1 (the small ball)
r_2to1=ball1.pos-ball2.pos   #final position minus initial position
F_2on1 = k*q1*q2*r_2to1/mag(r_2to1)**3
print(F_2on1)   #this print statement allows you to check the math

#draw an arrow representing the force, scale_arrow changes size of all force arrows
F_arrow_2on1 = arrow(pos=r1, axis=scale_arrow*F_2on1, color=c2)
```

Verify the code works

- On a scrap of paper, do a quick force calculation using the parameters you chose for the charges and positions. Make sure the components of force printed at the output match your calculation on paper.
- If you change the sign of the charge at the origin (q2), we expect the color of the ball AND the color of the force arrow to change. Furthermore, we expect the force between like charges should be repulsive. Verify your code responds as expected upon changing the sign of q2.
- If charge q1 is twice as far away, at position vector r1=vec(10,10, 0), how should the size of the force arrow change? Verify the code works as expected. **Afterwards, reset the position of q1 to r1=vec(5, 5, 0).**

Make a copy of this code

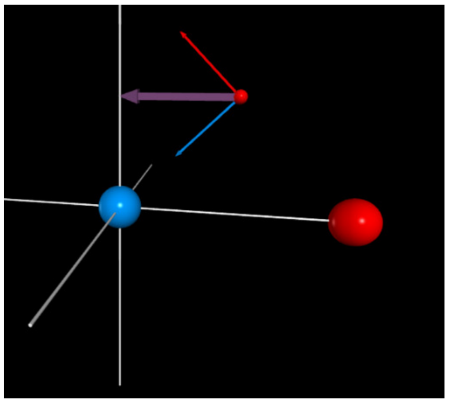
Click on your name somewhere near the top of the screen. You should see a screen shot of your code at some stage of its development. Next to the screenshot is a copy button. Make a copy. Why? As we move on, we want to make changes. If you make a mistake during the changes, you will have a back-up of your code that still works.

Modify it!

First add in a coordinate system using the lines of code shown below:

```
#draw a set of coordinate axes
line_x=cylinder(pos=vec(-10, 0, 0), axis=vec(20, 0, 0), radius=0.05)
line_y=cylinder(pos=vec(0, -10, 0), axis=vec(0, 20, 0), radius=0.05)
line_z=cylinder(pos=vec(0, 0, -10), axis=vec(0, 0, 20), radius=0.05)
```

Note: if you wish to add axis labels it is probably possible. In glowscript, go to the upper right corner and click “help”. A new window opens. In this new window click on the drop down box labeled “Choose a 3D object”. Look near the bottom of the options to find “text”. You are on your own from there…



Now show you know what you are doing by adding a third charge.
I let q3=+2.0e-4. I made r3=vec(10, 0, 0).
You will now need to draw *two* force arrows on q1.
Add a *third* arrow (different color) showing the NET force on q1.
Hopefully you can make something that looks like the screen shot at right.
Notice the force of 2 (the blue charge) on 1 is attractive because q2 has *opposite* sign of charge as q1.
Notice the force of 3 (big red charge) on 1 is repulsive because q3 has *same* sign of charge as q1.

For the following questions, take a guess BEFORE you run the code. Then use the code to see if you guessed correctly.

1) What should happen to the resultant force arrow if q2=q3=-2.0e-4?
2) What should happen to the resultant force arrow if q2=q3=+2.0e-4?
3) What should happen to the resultant force arrow if q2=q3=+1.0e-4?
4) What should happen to *each* force arrow if q3 is moved to r3=vec(13, 0, 0)?
5) What position vector is needed for q1 if you want it to be the top of an equilateral triangle of side length 10? Assume r2=vec(0, 0, 0) & r3=vec(10, 0, 0).

If desired, add in a fourth charge. **Why?** So you can simulate all the workbook questions which use four charges.

SAVE YOUR CODE BY MAKING A NEW COPY

How to make decent computer models of motion:
The Euler-Cromer method is a variation on the Euler method of numerical integration used in calc classes.
Do the following:

- Define values of constants like g or k (spring constant)
- Specify initial conditions (initial mass, position, velocity, etc)
- Specify a tiny time interval Δt. This time interval is supposed to be small enough such that acceleration, force, and velocity are approximately constant for that tiny amount of time. In practice you guess initially and refine your guess based on how the program runs.
- Create an iterative loop that does the following:
    - Calculate net force vector based on *current* position
    - Update the momentum of the system using $\vec{p}_f = \vec{p}_i + \vec{F}_{net}\Delta t$
    - Compute current avg velocity using $\vec{v}_{avg} \approx \frac{\vec{p}_f}{m}$
    - Update the position using $\vec{r}_f = \vec{r}_i + \vec{v}_{avg}\Delta t$

Nothing here will ever be exactly perfect. That said, you can keep slicing up using a smaller Δt to get higher quality *approximate* answers.

Now try writing the following code. Pay close attention: capitalization, punctuation, and indentation mistakes *may* cause the code to fail. If using glowscript, remember it is easier to debug if you run every few lines of code.

MAKE A NEW CODE CALLED MovingBall01
DO NOT ADD THIS CODE TO YOUR EXISTING CODE

```
from visual import *
ball=sphere(pos=vector(1, 2, 3), color=color.orange, radius=0.5)
velocity=vector(1, 3, 1)
my_arrow=arrow(pos=ball.pos, axis=velocity, color=color.green)

deltat=0.1
t=0
while t<80:
   rate(10)
   ball.pos=ball.pos+velocity*deltat
   t=t+deltat
```

Run it and see what happens.

For after class: from a book called Matter and Interaction more video tutorials can be found here:
www.Vpython.org/video01.html
www.Vpython.org/video02.html
www.Vpython.org/video03.html
www.vpython.org/video04.html
I started watching the first vid and it seemed pretty instructive and easy to follow.

Putting it all together

I wrote some code for a two charge system which allows q1 to move.

Make a copy of your two charge code (not the three charge code).
Add in the snippet below so charge q1 will orbit q2.
Note: if you change sim_speed to 2 the code runs at double speed (1/2 gives you slo-mo).

CODE BELOW: Additional snippet

```
#loop uses NET force ON ball1 to update position and momentum of ball1
#for now assume ball2 remains motionless as m2 >> m1
#WARNING: code produces unphysical results if ball1 moves inside ball2
t = 0
delta_t = 0.1
sim_speed = 1
while t<30:
   rate(sim_speed/delta_t)
   r_2to1=ball1.pos-ball2.pos                          #update vector between 1 & 2
   F_2on1 = k*q1*q2*r_2to1/mag(r_2to1)**3              #update force 2 exerts on 1
   p1 = p1 + F_2on1*delta_t                            #momentum update equation
   ball1.pos = ball1.pos + (p1/m1)*delta_t             #position update equation
   F_arrow_2on1.pos = ball1.pos                        #update F_2on1 arrow pos
   F_arrow_2on1.axis = F_2on1                          #update F_2on1 arrow size
   t = t + delta_t                                     #increment the time
```

For mad flair

Try adding a trail on the moving ball.
Do not ADD this code.
Look for the spot in the code where you referenced ball1 in the two charge code.
Add in the extra parts shown below:
Modify the original code with something like this:

```
        ball1 = sphere(pos=r1, radius=0.4, color=c1,
                        make_trail = True, trail_type = "points", interval = 5, retain = 50)
```

Now make the three ball code fully operational

Make an extra copy of the three charge code so you can modify without worry of losing your previous work.
Modify the code so the final product does the following:

a) Get rid of the arrows for F_2on1 & F_3on1.
b) Draw *net* force arrow *on each* ball. Hint: by Newton's 3rd law F_1on2=-F_2on1.
c) Use net force on each ball to update the momentum and, in turn, the position of each ball.

Use m1=1, m2=2000 & m3=2000 & positions r1=vec(5, 5, 0), r2=vec(-5, 0, 0) & r3=vec(15, 0, 0). **Run the code.**

Now try this set of initial conditions (**before hitting run,** predict what should happen based on parameters):

• r1=vec(5, 2, 0)	r2=vec(-5, 0, 0)	r3=vec(15, 0, 0)	
• q1=-1.0e-4	q2=+1.0e-4	q3=+1.0e-4	
• p1=vec(0, 0, 0)	p2=vec(0, 0, 0)	p3=vec(0, 0, 0)	…this means "starting from rest"
• m1=1	m2=1e6	m3=1e6	…this fixes 2 & 3 in place due to large mass

Now slowly decrease the size of m2 & m3 until you notice them start to move. Or try r1=vec(5.5, 0, 0).

Going Further: Create a code to determine electric *field* (not force) in 3D space. Start with a single positive 1nC charge at the origin. For 11 points. Use 5 to the left of the origin, 1 at the origin, and 5 to the right of the origin. I am assuming everyone will use an increment size of 1 between arrows. Determine the electric field and draw an appropriate sized arrow at each point.

Notice the field at the origin blows up to infinity. You'll want to put in an if statement to handle this situation.

Note: Please introduce either a saturation value (fields greater than some max value have arrows draw at fixed saturation value). Be sure to include a print statement indicating to the user each time the saturation value has been exceeded.

Note: Some arrows may be too small to be seen. Once you have a saturation value in the code, try increasing the charge to 10 μC. Hopefully the arrows closest to the charge will exceed you saturation value and all look the same size. Meanwhile, hopefully you can finally see the arrows farthest from the charge.

Make sure your code acts correctly if you change the sign of the charge to negative! Do the arrows flip direction?

Now try to do the same thing for 121 points arranged in all four quadrants of the xy-plane. Tip: compare your code to the PhET entitled charges and fields t verify it is working properly.

Now try adding in a second charge.

Finally, you could try to go up to 3D, you might need to use a much smaller number of points to prevent excessively long run times. Perhaps try a 5 by 5 by 5 array. Try making a line of charge (or a ring)…