Physics 231 – Lab 1
Introduction to VPython

Names: ________________
____
____

Equipment: computers with current VPython, whiteboards and pens

In this lab, you will get acquainted with VPython and write your first simulations.

Objective: To learn the basics of programming in VPython.

Background

VPython is a programming environment which will allow you to:

- Visualize vector quantities like position, momentum, and force in 3D
- Do calculations based on fundamental principles to predict the motion of interacting objects and animate the predicted motion

Overview of Computer Programs

- A computer program consists of a sequence of instructions.
- The computer carries out the instructions one by one, in the order in which they appear, and stops when it reaches the end.
- Each instruction must be entered exactly correctly (as if it were an instruction to your calculator).
- If the computer encounters an error in an instruction (such as a typing error), it will stop running and print a red error message.

A typical program in this class has four sections:

- Setup statements
- Definitions of constants
- Creation of objects
- Calculations to predict motion or move objects (these may be repeated)

0. Vector Refresher

Even before we start writing code in Python, refresh your memory of how vectors work and are visualized.

- Use a whiteboard to work problem 1.X.69 from the text.

1. Create a Program using the VIDLE program editor

On the screen desktop there should be an icon called “IDLE for Python.” Double click it to starts IDLE, which is the editing environment for VPython.

1.1 Starting a Program: Setup Statements

- Enter the following two statements in the IDLE editor window:

```
from __future__ import division, print_function
from visual import *
```

Every VPython program begins with these setup statements. The second statement tells the program to use the 3D module (called “visual”). The asterisk means “Add to Python all of the features available in the visual module.” The first statement (from *space* underscore underscore future underscore underscore *space* division, print_function) tells older versions of the Python language (prior to Python 3.0) to treat 1/2 as 0.5 (instead of 0!) and to use the new print function. Later versions of Python (who already know the print function and to treat ½ as 0.5) will simply ignore this line.

Before we write any more, name and save the program.

- From the "File" menu, select "Save." Save it on a disk, and give it the name "vectors.py". YOU MUST TYPE the ".py" file extension because the editor will NOT automatically add it. Without the ".py" file extension, the editor recognizes they type of file and also colorizes your program's statements in helpful ways.

1.2 Comments

Along with the code that instructs the computer (to define constants, create objects, perform calculations, etc.) you can also write comments that are just there to help you make sense of your code when you're reading over it. Comment lines start with a # (pound sign).

A comment line can be a note to yourself or others who read your code, such as:

```
# objects created in the following lines
```

You can also put a comment at the end of a line. For example, adding the following line to your code would tell the computer to create a sphere and would remind you, via the comment, that that sphere is going to represent a tennis ball:

```
Sphere() # represents the tennis ball.
```

Or you can "comment out" a line of code by putting a # (pound sign) at its beginning. This makes the computer ignore that line's instructions without your completely erasing the line. For example, putting the # in front of Sphere (),

```
# Sphere ()
```

would prevent your program from creating the sphere. This can be a handy trick when you're trying out tentative changes to your code or trouble shooting it.

- On the line below "from visual import *", label your program by typing "#" followed by your names. Then resave your program.

2. 3D Objects

In the next section of your program you will create 3D objects.

- *View Video 1. 3D Objects* [*http://www.youtube.com/VPythonVideos*](http://www.youtube.com/VPythonVideos) . This will give you an idea of what you're going to be doing and what it looks like.

- On the line after your two "from…" lines, type:

```
sphere()
```

This line tells the computer to create a sphere object as in the video.

- Run the program by pressing F5 on the keyboard to see the sphere get rendered.

Two new windows appear in addition to the editing window. One of them is the 3-D graphics window, which now contains a sphere.

2.1 The Python Shell / Text-Output Window and Error Debugging

When you ran the program, two new windows were created – one displaying your sphere and one displaying some text. The text-output window, also called the "Shell" window, displays any test you tell your program to print as well as displays any error statements that help you to find mistakes in your code. We'll get more into both of these functions shortly.

IMPORTANT: Arrange the windows on your screen so the Shell window is always visible (so you can notice when errors are identified.) DO NOT close the Shell window.

To KILL the program (stop its running), you can close the graphics window (not the Shell window.)

2.2 The 3D Graphics Scene

The other window that was generated when you ran the program displays any graphics associated with your program, the sphere in this case. By default, the sphere is at the origin <0,0,0>, and the the "camera" the "camera" (that is, your point of view) is looking directly at the origin.

- Hold down both buttons and move the mouse up and down to make the camera move closer or farther away from the center of the scene. (On a Macintosh, hold down the Options key and the mouse button while moving the mouse.)
- Hold down the right mouse button alone and move the mouse to make the camera "revolve" around the scene, while always looking at the center. (On a Macintosh, hold down the Apple Command key and the mouse button while moving the mouse.)

When you first run the program, the coordinate system has the positive x direction to the right, the positive y direction pointing up, and the positive z direction coming out of the screen toward you. You can then rotate the camera view to make these axes point in other directions.

2.2.1 Autoscaling and Units

VPython automatically "zooms" the camera in or out so that all objects appear in the window. Because of this "autoscaling", the numbers for the "`pos`" and "`radius`" could be in any consistent set of units, like meters, centimeters, inches, etc. For example, this could represent a sphere with a radius 0.1 m and a position vector of < 2, 4, 0 > m. In this course, we will always use SI units in our programs ("Systeme International", the system of units based on meters, kilograms, and seconds).

View VPython Instruction Videos: A. Debugging Syntax Errors
http://www.youtube.com/VPythonVideos which discusses common syntax errors you may encounter.

To see an example of an error message for yourself, try intentionally making a spelling mistake.

- Change the first line of the program to the following:

```
from visual export *
```

- Run the program. Notice you get a message in red text in the output window. The message gives the filename, the line where the error occurred, and a description of the error.
- Correct the error in the first line.

Whenever your program fails to run properly, look for a red error message in the text output window.

2.3 Changing Attributes of an Object

As discussed in the video, the objects have characteristic attributes such as the sphere's position, radius and color. If you don't explicitly give them values, default values will be set (thus the gray sphere of radius 1 sitting at the origin.) Of course, in most of your programs these objects will represent real things (like a tennis ball) and will move according to the laws of physics, so you'll be changing their attributes. To get a little practice, change the appearance of the sphere:

- Change the last line of the program to the following:

```
sphere(pos=vector(-2,4,0), radius=0.2, color=color.yellow)
```

- Run the program.

As is probably self-evident, this line of code the sphere defines the sphere object's three "attributes":

(1) `pos`: the position vector of the center of the sphere, relative to the origin at the center of the screen
(2) `radius`: the sphere's radius
(3) `color`: the sphere's color. Color values are written as "`color.aaa`", where "`aaa`" can be red, blue, green, cyan, magenta, yellow, orange, black, or white.

2.4 Creating a Second Sphere

We can have the program to create additional objects. Add a new sphere with its own position, radius, and color:

- Type the following on a new line, then run the program:

```
sphere(pos=vector(-3,-1,0), radius=0.3, color=color.white)
```

When you run the program now, you should see two spheres. In later exercises, the yellow sphere will represent a tennis ball and the white sphere will represent a baseball. (The radii are exaggerated for visibility.)

2.5 Creating Arrows

We often use arrow objects in VPython to represent vector quantities such as displacement, force, and velocity. Add arrows to your programs.

- Type the following on a new line, then run the program:

```
arrow(pos=vector(2,-3,0), axis=vector(3,4,0), color=color.cyan)
```

This line tells VPython to create an arrow object with three attributes:

(1) `pos`: the position vector of the tail of the arrow. In this case, the tail of the arrow is at the position $< 2, -3 ,0 >$ m.
(2) `axis`: the components of the arrow vector; that is, the vector measured from the tail to the tip of the arrow. In this case, the arrow vector is $< 3, 4, 0 >$ m.
(3) `color`: the arrow's color.

To demonstrate the difference between "`pos`" and "`axis`," make a second arrow with a different "`pos`" but same "`axis`."

- Type the following on a new line, then run the program:

```
arrow(pos=vector(3,2,0), axis=vector(3,4,0), color=color.red)
```

Note the red arrow starts at a different point than the cyan arrow, but has the same magnitude and direction. This is because they have the same "`axis`," but different values of "`pos`."

2.5.1 Scaling an Arrow's Axis

Often the simulations that you create will help you to visualize the motions of objects through space, represented by spheres, while overlaying arrows that represent vector properties. You'll often want to scale those arrows to fit in the scenes. Since the axis of an arrow is a vector, we can achieve this by performing scalar multiplication on it.

- Change the axis of the red arrow to be half as long and point in the opposite direction by multiplying by -0.5 so the last line of code reads:

```
arrow(pos=vector(3,2,0), axis=-0.5*vector(3,4,0), color=color.red)
```

- Run the program.

Multiplying an axis vector by a scalar will change the length of the arrow, because it changes the magnitude of the axis vector. The arrow will point in the same direction if the scalar is positive, and in the opposite direction if the scalar is negative.

For the next section, we will only need one arrow, so make the computer ignore one of the arrow commands by "commenting it out" using a number sign (#).

- Change the last line (the red arrow) to the following:

```
#arrow(pos=vector(3,2,0), axis=-0.5*vector(3,4,0), color=color.red)
```

- Run the program. There should now only be one arrow on the screen.

2.5.2 Representing Position Vectors with Arrows

Over this semester, we'll use arrows to represent a variety of vector quantities. We'll start with a simple relative position vectors. The relative position vector that starts at a position $\vec{A}$ and ends at a position $\vec{B}$ can be found by "final minus initial," or $\vec{B} - \vec{A}$.

Now you'll make an arrow represent the relative position vector of the tennis ball with respect to the baseball. The arrow's tail should be at the position of the baseball (the white sphere), and the tip should be at the position of the tennis ball (the yellow sphere). To do that you'll need to address these two questions:

(1) What would be the "`pos`" of this arrow, whose tail is on the baseball (the white sphere)?
(2) What would be the "`axis`" so that the tip is on the tennis ball (the yellow sphere)?

- Using these values of "pos" and "axis", change the last line of the program to make the red arrow point from the white baseball to the yellow tennis ball.

- Run the program and check that the appearance of the arrow is correct.

2.6 Variable Assignment: Naming Objects and Referencing Object Attributes

If you assign names to the objects (spheres and arrows) that you create, then you can use those names in subsequent calculations to, for example, make an arrow automatically stretch from one ball to the other, no matter how you move the balls.

View Video 2. Variables http://www.youtube.com/vpythonvideos

To see that the arrows currently *don't* automatically follow the balls,

- Change the position of the baseball (the white sphere) so that it has a *z*-component:

  ```
  sphere(pos=vector(-3,-1,3.5), radius=0.3, color=color.white)
  ```

- Run the program.

Note that the relative position arrow still points in its original direction even though the ball's no longer there. We want this arrow to automatically always point toward the tennis ball no matter what position we give the tennis ball. To do this, we will have to refer to the tennis ball's position *symbolically*. Since there is more than one sphere, we need to give the objects names to tell them apart.

- Name the spheres:

```
tennisball = sphere(pos=vector(2,4,0), radius=0.20, color=color.yellow)
baseball = sphere(pos=vector(-3,-1,3.5),radius=0.15, color=color.white)
```

Now, we can use these names later in the program to refer to each sphere individually. In addition, we can refer to the attributes of each object by writing, for example, "`tennisball.pos`" to refer to the tennis ball's position attribute, or "`baseball.color`" to refer to the baseball's color attribute. To see how this works, do the following exercise.

- Give the red arrow the name b2t, since it goes from the baseball to the tennis ball:

  ```
  b2t = arrow(...
  ```

Since we can refer to the attributes of objects symbolically, we want to write symbolic expressions for the "`axis`" and "`pos`" of the arrow `b2t`. The expressions should use general attribute names in symbolic form, like "`tennisball.pos`" and "`baseball.pos`", *not* specific numerical vector values such as "`vector(-3,-1,0)`." This way, if the positions of the tennis ball or baseball are changed, the arrow will still point from baseball to tennis ball.

- In symbols (letters, not numbers), what should be the "axis" of the red arrow that points from the baseball to the tennis ball?
- Change the arrow statement so that it uses appropriate symbolic expressions for "`pos`" and "`axis`".

- Run the program. Examine the 3D display closely to make sure that the red arrow still points from the baseball to the tennis ball. If it doesn't, correct your program, still using no numbers.
- Change the "`pos`" of the baseball to (-4, -2, 5). Change the "`pos`" of the tennis ball to (3, 1, -2). Run the program. Examine the 3D display closely to make sure that the red arrow still points from the baseball to the tennis ball. If it doesn't, correct your program (still using no numbers).

2.6.1 Printing Variable Values

The Shell Window is also used to display any text that you instruct the program to print. Often you'll want to print a variable's value. So here's a little practice

- Add the following line at the end of your program and run the program:

  ```
  print ( tennisball.pos )
  ```

  (note: the parenthesis are required for Python 3.x)
- Look at the text output window; the printed vector should be the tennis ball's position.

To make the output a little nicer (and make it easier to remember what the number that just got spewed out represents), you can add some text in the print statement.

- Modify the print statement to read

  ```
  print (‘position of tennis ball:’, tennisball.pos, “m.”)
  ```

2.7 While Loops

Another programming tool you will use in every program for this course is the loop. A loop is a set of instructions in a program that are executed over and over again, until some condition is met. For example, you may simulate constant-velocity motion of a car down a track from the start line to the finish line, so you'd calculate the new position over and over again, until the finish line is reached. There are several ways to create a loop, but usually in this course we will use the "while" statement to make loops.

- **Before you go on the next part, save your program (vectors.py), log on to WebAssign, and upload it via the Lab 1 assignment.**

Throughout the labs in this course, this icon will flag activities to be done using WebAssign. Today, WebAssign's only getting used for uploading code; however, in most subsequent labs, it's also used for answering questions. WebAssign gives you 5 shots at answering a lab question. These WebAssign activites count toward your lab grade.

3. Exercises

The previous pages of this lab have walked you through some essential elements of writing a VPython program. Now you'll get a little practice applying what you've learned to write a couple new programs. You'll probably want to keep a copy of your previous program open for easy reference.

For each of these parts, create a new program by going to the "File" menu and selecting "New window." Type the setup lines and a commented line with your names, then save your program with an appropriate name.

3.1 A Static Model of the Inner Planets

The first program that you write will make a model of the Sun and a few planets, let's call it "planets.py" (remember, you have to add on the ".py" by hand.) The distances are given in scientific notation. In VPython, to write numbers in scientific notation, use the letter "e"; for example, the number 2×10^7 is written as 2e7 in a VPython program.

Remember, after the "from…import…" lines, write a comment in which you name yourselves.

Create a model of the Sun and three of the inner planets (Mercury, Venus, and Earth). The distances from the Sun to each of the planets are given by the following:

Mercury:	5.8×10^{10} m from the sun
Venus:	1.1×10^{11} m from the sun
Earth:	1.5×10^{11} m from the sun

The inner planets all orbit the sun in roughly the same plane, so we'll place them in the x-y plane. Place the Sun at the origin, Mercury at $< 5.8\times10^{10}, 0, 0 >$ m, Venus at $< -1.1\times10^{11}, 0, 0 >$ m, and Earth at $< 0, 1.5\times10^{11}, 0 >$ m.

If you used the real radii of the Sun and the planets in your model, they would be too small for you to see! So use these values:

Radius of Sun:	7.0×10^9 m
Radius of Mercury:	2.4×10^9 m
Radius of Venus:	6.0×10^9 m
Radius of Earth:	6.4×10^9 m

The radius of the Sun in this program is ten times larger than the real radius, while the radii of the planets in this program are 1000 times larger than the real radii.

Finally an arrow representing the relative position of Mercury with respect to the Earth. That is, the arrow's tail should be on Earth, and the arrow's tip should be on Mercury. using symbolic values for the "pos" and "axis" attributes (no numerical data).

- **Before you go on, save your program (planets.py) and upload via WebAssign.**

3.2 Self-Check

It is important that you know the basic features of VPython and can work independently. You will be asked on quizzes and tests to write programs or portions of programs without being able to look back at old ones. In preparation for that, you'll write the next simple program not only <u>on your own</u>, but with a minimum of peaking at the programs you've just written and the instructions in the previous sections of the lab (sure, you can peak if you've got to.)

- Call the program selfcheck.py.
- Remember the lines that you need to start every program.
- Create a cyan sphere whose center is at <-4,-3,9 > m and whose radius is 0.7 m and give it the name `S1`.
- Create a red sphere whose center is at < 5,3,2 > m and whose radius is 0.6 m and give it the name `S2`.
- Create a green arrow whose tail is at the location of `S1` and whose head is at the location of `S2`. Do not use numbers for pos and axis; instead use appropriate attributes of `S1` and `S2`.

- **Save your program (selfcheck.py) and upload it via WebAssign.**
- **If you're not using your own laptop, email yourself copies of selfcheck.py and vectors.py.**