Introduction to 3D computer modeling

OBJECTIVES

In this course you will construct computer models to:

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

To do this you will use a 3D programming environment called *VPython*.

In this activity you will learn:

- How to use IDLE, the interactive editor for *VPython*
- How to structure a simple computer program
- How to create 3D objects such as spheres and arrows
- How to use vectors in *VPython*

Some of this reviews what we've already done in class, but students have found handy to have everything you need for a nice introduction to *VPython* in one place.

GROUP ROLES

Before you begin, you should agree on the responsibilities of the Manager, the Recorder, and the Skeptic for this part of the lab. The recorder will need to make sure everyone has your group's answers to the QUESTIONS sections; you will need them later.

OVERVIEW OF A COMPUTER PROGRAM

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 has four sections:

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

I. Using *IDLE* to create a program

Find an icon called "VPython Starter". This starts *IDLE*, the control program for Python. VPython opens in editor mode, where you can write your program. (You can also enter commands in a command-line mode, by going to the Run menu and selecting "Python Shell". From the prompt ">>>" you can enter *Python* commands.)

1. Starting a program: Setup statements

Enter the following two statements in the IDLE editor window:

```
from __future__ import division
from visual import *
```

Every *VPython* program begins with these setup statements.
The first statement (from *space* underscore underscore future underscore underscore *space* import *space* *) tells the *Python* language to treat 1/2 as 0.5. Without the first statement, the *Python* programming language does integer division with truncation and 1/2 is zero!
The second statement tells the program to use the 3D module (called "visual").

Before we write any more, let's save the program:

- **In the *IDLE* editor, from the "File" menu, select "Save." Browse to a location where you can save the file (probably the desktop), and give it the name "vectors.py". YOU MUST TYPE the ".py" file extension --*IDLE* will NOT automatically add it. Without the ".py" file extension IDLE won't colorize your program statements in a helpful way.**

2. Creating an object

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

- **Now tell *VPython* to make a sphere. On the next line, type:**

```
sphere()
```

This statement tells the computer to create a sphere object. Run the program by pressing fn-F5 on the keyboard. Two new windows appear in addition to the editing window. One of them is the 3-D graphics window, which now contains a sphere.

3. The 3-D graphics scene

By default the sphere is at the center of the scene, and the "camera" (that is, your point of view) is looking directly at the center.

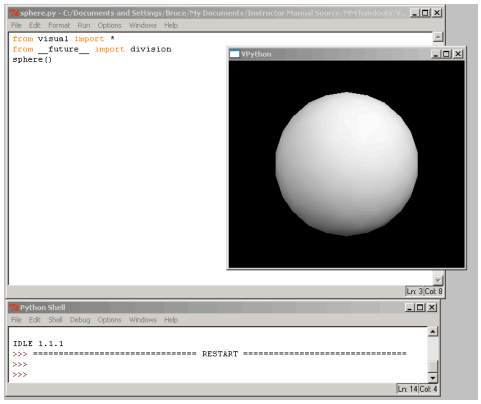
- **PC: Hold down both buttons and move the mouse up and down to make the camera move closer or farther away from the center of the scene. Mac: Hold down the option key to do the same thing. (This may differ on Linux machines.)**
- **PC: Hold down the right mouse button alone and move the mouse to make the camera "revolve" around the scene, while always looking at the center. Mac: Hold down the Apple key to do the same thing. (This may differ on Linux machines.)**

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.

4. The *Python* Shell window is important -- Error messages appear here

IMPORTANT: Arrange the windows on your screen so the Shell window is always visible.
DO NOT CLOSE THE SHELL WINDOW.
KILL the program by closing only the graphic display window.

The second new window that opened when you ran the program is the Python Shell window. If you include lines in the program that tell the computer to print text, the text will appear in this window.



- **Use the mouse to make the *Python* Shell window smaller, and move it to the lower part of the screen. Keep it open when you are writing and running programs so you can easily spot error messages, which appear in this window.**
- Make your edit window small enough that you can see both the edit window and the Python Shell window at all times.
- **Do not expand the edit window to fill the whole screen. You will lose important information if you do!**
- **To kill the program, close only the graphics window. Do not close the *Python* Shell window.**

5. Error messages: Making and fixing an error

To see an example of an error message, let's try making a spelling mistake:

- **Change the second statement of the program to the following:**

```
from bisual import *
```

- **Run the program.**

Notice you get a message in red text in the *Python* Shell window. The message gives the filename, the line where the error occurred, and a description of the error (in this case "ImportError: No module named bisual").

Read error messages from the bottom up: The bottom line contains the information about the location of the error.

- **Correct the error in the program.**

Whenever your program fails to run properly, look for a red error message in the *Python* Shell window.

Even if you don't understand the error message, it is important to be able to see it, in order to find out that there is an error in your code. This helps you distinguish between a typing or coding mistake, and a program that runs correctly but does something other than what you intended.

6. Changing "attributes" (position, size, color, shape, etc.) of an object:

Now let's give the sphere a different position in space and a radius.

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

```
sphere(pos=vector(-5,2,-3), radius=0.40, color=color.red)
```

- **Run the program. Experiment with other changes to "pos", "radius", and "color", running the program each time you change something. Find answers to the following questions:**

QUESTIONS TO ANSWER ABOUT SPHERES:

What does changing the "pos" attribute of a sphere do?
What does changing the "radius" attribute of a sphere do?
What does changing the "color" attribute of a sphere do? What colors can you use?

7. 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 0.20 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).

8. Creating a second object

- **We can tell the program to create additional objects. On a new line, after the statement that creates the red sphere, create a green sphere whose center is at < -3, -1, 0> and whose radius is 0.15, then run the program.**

You should now see the original red sphere and a new green sphere. In later exercises, the red sphere will represent a baseball and the green sphere will represent a tennis ball. (The radii are exaggerated for visibility.)

9. Creating an arrow object

We often use arrow objects in *VPython* to depict vector quantities. We next add arrows to our 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)
```

- **Experiment with other changes to "pos", "axis", and "color", running the program each time you change something. Change one thing at a time so you can tell what effect your changes have. For example, try making a second arrow with the same "pos" but a different axis (and different color.)**
- **Find answers to the following questions:**

QUESTIONS TO ANSWER ABOUT ARROWS:

Is the "pos" of an arrow the location of its tail, its tip, its middle, or somewhere else?
Is the "axis" of an arrow the position of its tail, the position of its tip, or the position of the tip relative to the tail (that is, a vector pointing from tip to tail)?
To create a sphere whose center is at the tip of the cyan arrow above, what should the "pos" of the sphere be?

10. Multiplying by a scalar: Scaling an arrow's axis

Since the axis of an arrow is a vector, we can perform scalar multiplication on it. Start with this arrow:

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

- **Modify the axis of the arrow by changing the statement to the following, and note what happens:**

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

QUESTIONS TO ANSWER ABOUT SCALING ARROWS:

To make an arrow 3 times as long without changing its direction, what scalar factor would you multiply "axis" by?
To reverse the direction of an arrow without changing its length, what scalar factor would you multiply "axis" by?

11. Comment lines (lines ignored by the computer)

Comment lines start with a # (pound sign).
A comment line can be a note to yourself, such as:

objects created in following lines

Or a comment can be used to remove a line of code temporarily, without erasing it.

- **Put a # at the beginning of the line creating your arrow, and run the program:**

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

Note the pound sign at the beginning of the line. The pound sign lets *VPython* know that anything after it is just a comment, not actual instructions. The statement will be skipped when the program is run.

- **Run the program. Explain what you see.**

12. Representing position vectors with arrows

- **Clean up your program so it contains only the following objects:**

A red sphere (representing a baseball) at location < -5, 2, -3 > , with radius 0.4
A green sphere (representing a tennis ball) at location < -3, -1, 0 >, with radius 0.15
A cyan arrow with its tail at < 2, -3, 0 > and its tip at < 2, -3, 0 > + < 3, 4, 0 >

We can use arrows to represent position vectors and relative position vectors. Remember that a 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 $\vec{B} - \vec{A}$.

We want to make an arrow represent the relative position vector of the tennis ball with respect to the baseball. That is, the arrow's tail should be at the position of the baseball (the *red* sphere), and the tip should be at the position of the tennis ball (the *green* sphere).

- **What would be the "pos" of this arrow, whose tail is on the baseball (the red sphere)?**
- **What would be the "axis" of this arrow, so that the tip is on the tennis ball (the green sphere)?**
- **Using these values of "pos" and "axis", change the last line of the program to make the cyan arrow point from the red baseball to the green tennis ball.**
- **Run the program.**

CHECKPOINT: Examine the 3D display carefully.
If the cyan arrow does not point from the red baseball to the green tennis ball, correct your program.

13. Naming objects; Using object names and attributes

- **Change the position of the green sphere to < -3, -1, 3.5 >**

- **Run the program.**

- **Note that the relative position arrow still points in its original direction. We want this arrow to 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. But first, since there is more than one sphere and we need to tell them apart, we need to give the objects names.

- **Give names to the spheres by changing the "sphere" statements in the program to the following:**

```
baseball = sphere(pos=vector(5, 2, -3), radius=0.40, color=color.red)
tennisball = sphere(pos=vector(-3,-1,3.5), radius=0.15, color=color.green)
```

We've now given names to the spheres. We can use these names later in the program to refer to each sphere individually. Furthermore, we can specifically 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.

- Start a new line at the end of your program and type:

```
print tennisball.pos
```

- Run the program.
- Look at the text output window. The printed vector should be the same as the tennis ball's position.

- **What do you expect to see if you add this line to your program?**

```
print tennisball.pos - baseball.pos
```

- **Run the program to see if your prediction was correct.**

Let's also give a name to the arrow.

- **Edit the last line of the program (the cyan arrow) to the following, to give the arrow a name:**

```
bt = arrow(pos= .... )
```

Since we can refer to the attributes of objects symbolically, we want to write symbolic expressions for the "axis" and "pos" of the arrow "bt". 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 "pos" of the cyan arrow that points from the baseball to the tennis ball? **Make sure that your expression doesn't contain any numbers.**

- In symbols (letters, not numbers), what should be the "axis" of the cyan arrow that points from the baseball to the tennis ball? (Remember that a relative position vector that starts at position $\vec{A}$ and ends at position $\vec{B}$ can be found by "final minus initial," or $\vec{B} - \vec{A}$.). **Make sure that your expression doesn't contain any numbers.**

- Change the last line of the program so that the arrow statement uses these symbolic expressions for "pos" and "axis".

- Run the program. Rotate the 3D display and examine it closely to make sure that the cyan 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 cyan arrow still points from the baseball to the tennis ball. If it doesn't, correct your program, still using no numbers in the statement creating the arrow.

CHECKPOINT: Does your program behave as it is supposed to? That is: if you change the positions of both balls, does the arrow automatically adjust so it still points from the baseball (red) to the tennis ball (green)?

Compare your work to the work of another group.

.

14. Turn in your program to *WebAssign*

There is a *WebAssign* assignment where the Recorder should turn in your final program. When you turn in a program to *WebAssign*, be sure to follow the instructions given there, which may sometimes ask you to change some of the parameters in your program. Everyone in the group should save a copy of the program, for future reference. (You can email the program to each other or use a flash drive, if necessary.)

15. Using *VPython* outside of class

You can download *VPython* from http://vpython.org and install it on your own computer.

16. Reference manual and programming help

There is an on-line reference manual for *VPython*. In the text editor (IDLE), on the Help menu choose "Visual". The first link, "Reference manual", gives detailed information on spheres, arrows, etc. In the text editor (*IDLE*), on the Help menu choose "Python Docs" to obtain detailed information on the *Python* programming language upon which *VPython* is based. We will use only a small subset of Python's extensive capabilities.