

Lab 2 - Visualizing with VPython

Thursday 7 September, 2006 - Due: Thursday 14 September

The aim of this lab is to get you started in thinking in terms of vectors (rather than in components of vectors) and thinking in 3D. (In addition, some programming ideas are marked with a ♣ symbol.) We will also generalize our harmonic motion example discussed in class to the case where the spring constants vary with coordinate direction. Please include in your writeup any sections of Python code you write.

1 bounce.py

Let's extend the VPython program that was presented in Lec. 2.

The following program is essentially the “complete VPython program” described on <http://vpython.org/vpythonprog.htm>.

```
from visual import *

floor = box(pos=(0,0,0), length=4, height=0.5, width=4, color=color.blue)
ball = sphere(pos=(0,4,0), radius=1, color=color.red)
ball.velocity = vector(0,-1,0)
dt = 0.01

while 1:
    rate(100)

    ball.pos = ball.pos + ball.velocity*dt

    if ball.y < ball.radius:
        ball.velocity.y = -ball.velocity.y
    else:
        ball.velocity.y = ball.velocity.y - 9.8*dt
```

To save yourself a lot of typing, you may be tempted to **select-copy-and-paste** the above code (or the code from the above URL) into the IDLE window. That's fine. However, you may have to manually correct any incorrect indenting or weird spacing!

Let's go through this program line by line. Hopefully, some of this will be a review. You might want to have the online documentation handy.

- `from visual import *` tells the Python interpreter to load the VPython module, historically called “visual”, which provides a lot of the 3D-visualization tools we'll be using.
- `box()` [on a line by itself] would create a box [via this “constructor”] with the default features described in <http://www.vpython.org/webdoc/visual/box.html>.

By specifying parameters, e.g.,

```
box(pos=(0,0,0), length=4, height=0.5, width=4, color=color.blue),
```

we create a box with different features.

By specifying a name [technically, a “reference”] `floor`, e.g.,

```
floor=box(pos=(0,0,0), length=4, height=0.5, width=4, color=color.blue),
```

we can refer to this box later on. For instance, we can change the color using

```
floor.color=color.yellow or floor.color=(1,1,0).
```

We can also print out the position of the floor using `print floor.pos`.

- Analogously, `ball = sphere(pos=(0,4,0), radius=1, color=color.red)` creates a red ball of radius=1, located at (0, 4, 0).
- `ball.velocity = vector(0,-1,0)` creates a new attribute `velocity` of the “object” named ball. Of course, Python doesn’t understand what velocity might mean physically. That’s your job. All Python knows is that this attribute is assigned a vector quantity `vector(0,-1,0)`.
- `dt = 0.01` assigns a floating-point value of 0.01 to a new variable `dt`.
- `while 1:` starts a “while loop block”. While the condition [here, “1”] is true, perform the following block of indented code:

```
    rate (100)

    ball.pos = ball.pos + ball.velocity*dt

    if ball.y < ball.radius:
        ball.velocity.y = -ball.velocity.y
    else:
        ball.velocity.y = ball.velocity.y - 9.8*dt
```

Since the condition “1” is always true, this block of code will be repeated forever [unless you somehow interrupt the program or break out of this while-loop].

Let’s go through this block of code.

- The line `rate(100)` limits the rate of screen updates to at most 100 frames per second, assuming that your hardware can support it. Without a `rate` statement, the `while` loop will be repeated as quickly as possible, which could inadvertently consume your much of your computer’s attention, leaving it unresponsive to your attempts to interrupt the program.
- The line `ball.pos = ball.pos + ball.velocity*dt` is the transcription of

$$\vec{s}_{new} = \vec{s}_{old} + \vec{v} dt.$$

- The line `if ball.y < ball.radius:` encodes a crude test if the ball making contact with the floor.

- * If true, `ball.velocity.y = -ball.velocity.y` reverses the direction of the y -component of the velocity.

★ *What is the physical reasoning and the set of assumptions underlying this line?*

- * If false (i.e., `else:`),
`ball.velocity.y = ball.velocity.y - 9.8*dt` is the transcription of

$$v_{y,new} = v_{y,old} + (-9.8) dt.$$

★ *What is the physical reasoning and the set of assumptions underlying this line?*

That's it! The ball bounces continually! in 3D!

- ★ Observe what happens when you right-click-and-drag your mouse.
- ★ Observe what happens when you middle-click-and-drag your mouse.

 autoscale

By default, the view changes to accommodate the entire visible scene. To prevent this from happening, you can insert the command `scene.autoscale=0` after the first line. The view then uses the default value of `scene.range=10`.

(Refer to <http://www.vpython.org/webdoc/visual/display.html>.)


Just for fun, include a coefficient of 0.90: `ball.velocity.y = -0.90*ball.velocity.y`

- ★ *What is the physical reasoning and the set of assumptions underlying this line?*

Now, here's a little puzzle: Instead of 0.90, use a slightly smaller number 0.85.

- ★ *What happens? and Why?*

(Hint for debugging: within the while loop, insert a line `print ball.pos` to see what is happening.)

 debugging
with print
statements

★ *Is there a better test that one should use? If so, what? Think physically!*
Hint: you can logically combine tests with Boolean operators (`and`, `or`, `not`) and parentheses, if necessary. Here's a silly example:

```
if ball.y < ball.radius and 1+2>4:.
```

Note, that this more complex condition is always false since “ $1 + 2 > 4$ ” is false, and, logically, (anything `and` false) is false.

2 bounce.py revised

Let's consider the original `bounce.py` program again. While this program is fairly simple and efficient, some modifications could be made to make it easier to simulate a more general situation. *Take advantage of VPython's implementation of the vector data type!*

Let's take care of the obvious part first.

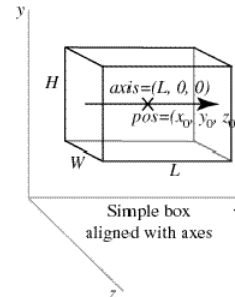
From what was presented in Lec. 2, you can probably improve the “freefall” statement `ball.velocity.y = ball.velocity.y - 9.8*dt` by writing it in terms of **vector** variables (rather than ordinary scalar variables).

★ *Modify the program to use an acceleration vector.*

Now, let's consider how the “floor” is being modeled. Conceptually, it's a section of plane parallel to the xz-plane. A convenient visualization is a **box** with a small dimension in the y-direction.

Use this figure from

<http://www.vpython.org/webdoc/visual/box.html>



to compare the original line [which I have spaced out for comparison]

```
floor=box(pos=(0,0,0),                length=4, height=0.5, width=4, color=color.blue)
```

with this proposed improvement

```
floor=box(pos=(0,0,0), axis=(0,1,0), length=0.5, height=4, width=4, color=color.blue).
```

In the animation window, it looks identical to the original. Why is this better? Before, with `axis` unspecified [in the constructor], `floor.axis` was set at its default value `vector(1,0,0)`, which has no special physical meaning. But, now, `floor.axis` is `vector(0,1,0)`, which has physical meaning as \hat{y} , the unit normal-vector to that plane. Note that the “thickness” of the floor (now along the direction of `floor.axis`) is now associated with `floor.length`. With this improvement, one can now conveniently orient the plane by simply modifying the `floor.axis`.

★ *Draw a closed box centered at the origin, with six square faces of edge length 8. (See the next paragraphs for hints.)*

Let me help you get started, as well as show you some ideas for developing visualizations.

```
from visual import *
```

```
#draw two vertical walls
```

```
wallR=box(pos=( 4,0,0), axis=(-1,0,0), length=0.3, height=8, width=8)
```

```
wallL=box(pos=(-4,0,0), axis=( 1,0,0), length=0.3, height=8, width=8)
```

- First, you should note that the lines that have a # signify “comments” in Python. Anything on the line that appears after a # will not be executed. So, you can use this to write notes to explain what you are doing. (You can also use this is to “comment out code”, which is very useful for debugging.)

- Using the method described above, we draw two vertical walls, with the `axis` pointing along their inward normal vectors.
- Hopefully, you realize that the 8s and the 4s aren't unrelated parameters. They, of course, are related to the required edge length of 8. For clarity and for ease of modification (for example, if one now wants an edge length of 12), it's convenient to introduce a new variable `edge=8`, as in:

```
#draw two vertical walls
edge=8
thk=0.3
wallR=box(pos=( edge/2. ,0,0), axis=(-1,0,0),
           length=thk, height=edge, width=edge)
wallL=box(pos=(-edge/2. ,0,0), axis=( 1,0,0),
           length=thk, height=edge, width=edge)
```

where I have gone ahead and introduced another variable `thk` for convenience.

In addition, I have made good use of spacing to line up analogous statements. Long Python lines can be split up by just hitting [ENTER], then inserting enough spaces before the remainder of the line to place it in a convenient position. Python is smart enough to know that the unfinished command continues on the next line

♣ use spaces

That should give you enough of a start to finish the rest of the box. You might wish to color-code the opposite sides of the box.

[OPTIONAL] Include a capture of your animation window in your Word document.

- Make the animation window the active-window. Use your mouse to pose your view. Then, while depressing [ALT], hit [PRNT SCRN].
- Make your Word document the active-window. Then, while depressing [CTRL], hit [V].
- Since you'll probably want to print this out, let's try to save on ink! First, make the background of the animation window white by inserting this line `scene.background=color.white`. Then, do your capture.

♣ screen capture in Windows

3 bounce2.py - your version

In your Python subfolder `C:\Python24\Lib\site-packages\visual\examples`, you'll find `bounce2.py`, which visualizes a ball bouncing in a 3D box (with one face not drawn so that you can look inside). Unlike `bounce.py`, the ball travels inertially in a straight line with constant velocity between collisions with the walls.

I've included the important lines of `bounce2.py` here:

```
from visual import *

side = 4.0
thk = 0.3
```

```

s2 = 2*side - thk
s3 = 2*side + thk
wallR = box(pos=( side, 0, 0),length=thk,height=s2, width=s3,
            color=color.red)
wallL = box(pos=(-side, 0, 0),length=thk,height=s2, width=s3,
            color=color.red)
wallB = box(pos=(0, -side, 0),length=s3, height=thk,width=s3,
            color=color.blue)
wallT = box(pos=(0, side, 0),length=s3, height=thk,width=s3,
            color=color.blue)
wallBK= box(pos=(0, 0, -side),length=s2, height=s2, width=thk,
            color=(0.7,0.7,0.7))

ball = sphere (color = color.green, radius = 0.4)
ball.mass = 1.0
ball.p = vector (-0.15, -0.23, +0.27)

side = side - thk*0.5 - ball.radius

dt = 0.5
t=0.0
while 1:
    rate(100)
    t = t + dt
    ball.pos = ball.pos + (ball.p/ball.mass)*dt
    if not (side > ball.x > -side):
        ball.p.x = -ball.p.x
    if not (side > ball.y > -side):
        ball.p.y = -ball.p.y
    if not (side > ball.z > -side):
        ball.p.z = -ball.p.z

```

★ Save this as new Python file also called `bounce2.py` in your home directory (I: drive). (Don't overwrite the original `bounce2.py`.) Run this program to see what it does.

★ In a new file, replace the walls in `bounce2.py` with the corresponding five walls from the closed box you drew in the previous section.

You can modify your `bounce2.py` and save it as a new file with a new name, while keeping your personal `bounce2.py` around for comparison. Notice that the use of the `axis` attribute simplifies the code for drawing box walls as the thickness is now always specified by the `length` parameter, independent of orientation.

I'll leave it up to you to decide if you want to account for the wall thicknesses. You can do so by following the details of the original `bounce2.py` box drawing code.

★ Explain the logical tests used to decide when the ball collides with a wall. Explain the corresponding command executed when the test is true.

4 Lissajou figures

Finally, we will modify the 2D harmonic oscillator code to handle situations in which the restoring force is different in the x and y directions (for simplicity we will continue to set $F_z = 0$). Cut/paste the code below into a Python file window taking care to preserve indentation.

```
from visual import *

scene.autoscale=0
scene.range=10.0

def force(pos,vel,t):
    result = vector(0,0,0)
    kx=1.0
    ky=1.0
    result.x=-kx*pos.x
    result.y=-ky*pos.y
    result.z=0
    return result

ball=sphere(radius=0.5,pos=vector(0,0,0),
            track=curve(radius=0.05),mass=1.0)
ball.vel=vector(5,5,0)
t=0
dt=0.01

while true:
    rate(100)
    t=t+dt
    ball.pos=ball.pos+ball.vel*dt
    ball.vel=ball.vel+force(ball.pos,ball.vel,t)*dt
    ball.track.append(pos=ball.pos)
```

★ *Keeping $k_y = 1.0$ change k_x from 1.0 to 16.0 in steps of unity. Sketch/do a screen capture of what you see*

★ *Can you come up with a rule for the ratio k_x/k_y that describes when the orbit of the particle is particularly simple (when the orbit repeats)? In these cases the orbit is called a Lissajou figure. Interpret this condition on the ratio in terms of the natural frequencies of oscillation in the x and y directions*

★ *Experiment also with $k_x < 1$*

[OPTIONAL] If you are completely done and have some time to play, try adding these lines, a set at a time.

- before the while loop, `box(axis=(0,0,0.03),width=15,height=15,color=color.blue)`
- before the while loop, `scene.forward=vector(1,0,0)`
`scene.up=vector(0,0,1)`
- within the while block, after the append statement `scene.forward=ball.vel`
`scene.center=ball.pos+vector(0,0,2)`