1 Overview

VPython is a programming language that is easy to learn and is well suited to creating 3D interactive models of physical systems. VPython has three components that you will deal with directly:

- Python, a programming language invented in 1990 by Guido van Rossem, a Dutch computer scientist. Python is a modern, object-oriented language which is easy to learn.
- Visual, a 3D graphics module for Python created in 2000 by David Scherer while he was an undergraduate student at Carnegie Mellon University. Visual allows you to create and animate 3D objects, and to navigate around in a 3D scene by spinning and zooming, using the mouse.
- IDLE, an interactive editing environment, written by van Rossem and modified by Scherer, and later by others, which allows you to enter computer code, try your program, and get information about your program.

This tutorial assumes that Python and Visual are installed on the computer you are using.

2 Your first program

> Start IDLE. On Windows, there may be an icon on your desktop, or look for "IDLE (Python GUI)" in the Python folder on the Programs menu. On Macintosh, doubleclick the icon in Applications/VPython. A window labeled "Untitled" should appear. If instead a non-empty "Python Shell" window appears, on the File menu choose "New Window".

> In the empty window, as the first line of your program, type the following statement:

```
from visual import *
```

This statement instructs Python to use the Visual graphics module.

> As the second line of your program, type the following statement:

```
sphere()
```

3 Running the program

> Now run your program by pressing F5 (or by choosing "Run program" from the "Run" menu).

When you run the program, there is a window titled "VPython," in which you should see a white sphere, and a window titled "Python Shell". Move the Shell window to the bottom of your screen where it is out of the way but you can still see it (you may make it smaller if you wish).

> In the VPython window, hold down the left+right buttons on a two-button mouse (or the option key on Macintosh) and move the mouse. You should see that you are able to zoom into and out of the scene.

> Now try holding down the right mouse button (hold down the Apple command key on Macintosh). You should find that you are able to rotate around the sphere (you can tell that you are moving because the lighting changes).

Keep in mind that you are moving a camera around the object. When you zoom and rotate you are moving the camera, not the object.

4 Stopping the program

Click the close box in the upper right of the display window (VPython window) to stop the program. Leave the Shell window open, because this is where you will receive error messages.

5 Save Your program

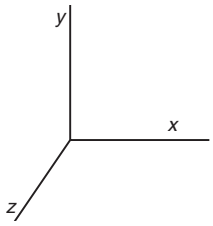
If you have not already saved your program, save it now by pulling down the File menu and choosing Save As. Give the program a name ending in ".py," such as "MyProgram.py". You must type the ".py" extension; IDLE will not supply it and if it is omitted IDLE will not colorize your program to identify various components, which is useful. Every time

you run your program, IDLE will save your code before running. You can undo changes to the program by pressing CTRL-z.

6 A ball in a box

The goal of the following activity is to give you experience in using the velocity to update the position of an object, to create a 3D animation. You will write a program to make a ball bounce around in a box, in 3D.

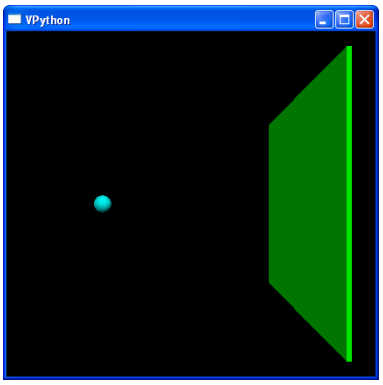
To position objects in the display window we use their 3D coordinates. The origin of the coordinate system is at the center of the display window. The positive *x* axis runs to the right, the positive *y* axis runs up, and the positive *z* axis comes out of the screen, toward you.



Start a new program like the following, which displays a ball and a wall (we've called it wallR because it is on the right side of the scene). *Read the program carefully and make sure you understand the relationship between these statements, the coordinate system, and the display generated by the program.*

```
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.cyan)
wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)
```

As with the `sphere` object, the `pos` attribute of the VPython `box` object positions the center of the box. The `size` attribute specifies the width (in x), height (in y), and depth (in z). Since the width has been specified to be only 0.2, the box displays as a thin wall. There are 9 colors easily accessible by `color.xxx`: red, green, blue, yellow, magenta, cyan, orange, black, and white. It is possible to design your own colors.



7 Updating the position of the ball

We are going to make the cyan ball move across the screen and bounce off of the green wall. We can think of this as displaying "snapshots" of the position of the ball at successive times as it moves across the screen. To specify how far the ball moves, we need to specify its velocity and how much time has elapsed.

We need to specify the velocity of the ball. We can make the velocity of the ball an attribute of the ball, by calling it `ball.velocity`. Since the ball can move in three dimensions, we must specify the x, y, and z components of the ball's velocity. We do this by making `ball.velocity` a vector.

> Add this statement to your program:

```
ball.velocity = vector(25,0,0)
```

We need to specify a time interval between "snapshots." We'll call this very short time interval "deltat" to represent Δt. In the context of the program, we are talking about virtual time (that is, time in our virtual world); a virtual time interval of 1 second may take much less than one second on a fast computer. To keep track of how much total time has elapsed, let's also set a "stopwatch" time `t` to start from zero.

> Add these statements to your program:

```
deltat = 0.005
t = 0
```

If you run the program, nothing will happen, because we have not yet given instructions on how to use the velocity to update the ball's position. We need to use the "position update" relationship among position, velocity, and time interval, that the final (vector) position is the initial position plus the average velocity times the time interval:

$$\vec{r}_f = \vec{r}_i + \vec{v}\Delta t$$

This relationship is valid if the velocity is nearly constant (both magnitude and direction) during the time interval Δt. If the velocity is changing (in magnitude and/or direction), you need to use a short enough time interval that the velocity at the start of the time interval is nearly the same as the average velocity during the whole time interval.

We need to translate the standard notation $\vec{r}_f = \vec{r}_i + \vec{v}\Delta t$ into the language of VPython:

```
ball.pos = ball.pos + ball.velocity*deltat
```

Just as velocity is a vector, so is the position of the ball, `ball.pos`. As in most programming languages, the equals sign means something different than it does in ordinary algebra notation. In VPython, the equals sign is used for *as-*

signment, not equality. That is, the line *assigns* the vector `ball.pos` a *new* value, which is the *current* value of `ball.pos` plus the displacement `ball.velocity*deltat` ($\vec{v}\Delta t$), the *change* in the position.

8 A 3D animation: Continuous updating of the position of the ball

Your program should now look like this:

```
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.cyan)
wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)
ball.velocity = vector(25,0,0)
deltat = 0.005
t = 0
ball.pos = ball.pos + ball.velocity*deltat
```

> Run your program.

Not much happens! The problem is that the program only took one time step; we need to take many steps. To accomplish this, we write a *while loop*. A `while` loop instructs the computer to keep executing a series of commands over and over again, until we tell it to stop.

> Insert the following line just before your position update statement (that is, just before the last statement):

```
while t < 3:
```

Don't forget the colon! Notice that when you press return, the cursor appears at an indented location after the `while` statement. The indented lines following a `while` statement are inside the loop; that is, they will be repeated over and over. In this case, they will be repeated as long as the "stopwatch" time `t` is less than 3 seconds.

> Indent your position update statement under the while statement:

```
    ball.pos = ball.pos + ball.velocity*deltat
```

Note that if you position your cursor at the end of the `while` statement, IDLE will automatically indent the next lines when you press ENTER. Alternatively, you can press TAB to indent a line. All indented statements after a `while` statement will be executed every time the loop is executed.

> Also update the stopwatch time by adding `deltat` to `t` for each passage through the loop:

```
    t = t + deltat
```

Your program should now look like this:

```
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.cyan)
wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)
ball.velocity = vector(25,0,0)
deltat = 0.005
t = 0
while t < 3:
    ball.pos = ball.pos + ball.velocity*deltat
    t = t + deltat
```

> Run your program.

Depending on the speed of your computer, the ball may have moved so fast that you saw only a flash! Moreover, VPython by default tries to keep all of the objects visible, so as the ball moves far away from the origin, VPython moves the "camera" back, and the wall recedes into the distance. This is called "autoscaling".

> To slow down the animation, insert the following statement inside the loop (just after the `while` statement, indented as usual):

```
    rate(100)
```

This specifies that the `while` loop will not be executed more than 100 times per second, even if your computer is capable of many more than 100 loops per second. (The way it works is that each time around the loop VPython checks

to see whether 1/100 second of real time has elapsed since the previous loop. If not, VPython waits until that much time has gone by. This ensures that there are no more than 100 loops performed in one second.)

> Run your program.

You should see the ball move to the right more slowly. However, it keeps on going right through the wall, off into empty space, because this is what it was told to do. VPython doesn't know any physics! You have to tell it what to do.

9 Making the ball bounce: Logical tests

To make the ball bounce off the wall, we need to detect a collision between the ball and the wall. A simple approach is to compare the *x* coordinate of the ball to the *x* coordinate of the wall, and reverse the *x* component of the ball's velocity if the ball has moved too far to the right. In VPython you can access the x, y, or z component of any vector by referring to the x, y, or z attribute of that vector. In the statements below, `ball.pos` is a vector, and `ball.pos.x` is the x component of that vector. Similarly, `ball.velocity` is a vector, and `ball.velocity.x` is the x component of that vector.

> Insert these statements into your while loop, just before your position update statement (press Enter at the end of the preceding statement, or insert tabs before the new statements, or select the new statements and use the "Indent region" option on the "Format" menu to indent the lines):

```
if ball.pos.x > wallR.pos.x:
    ball.velocity.x = -ball.velocity.x
```

The indented statement after the `if` statement will be executed (and reverse the x component of velocity) only if the logical test in the previous line gives `True` for the comparison. If the result of the logical test is `False` (that is, if the *x* coordinate of the ball is not greater than the *x* coordinate of the wall), the indented line will be skipped. We want this logical test to be performed every time the ball is moved, so we need to indent both of these lines, so they are inside the `while` loop.

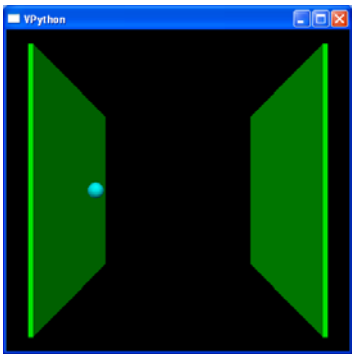
Your program should now look like this:

```
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.cyan)
wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)
ball.velocity = vector(25,0,0)
deltat = 0.005
t = 0
while t < 3:
    rate(100)
    if ball.pos.x > wallR.pos.x:
        ball.velocity.x = -ball.velocity.x
    ball.pos = ball.pos + ball.velocity*deltat
    t = t + deltat
```

> Run your program.

You should observe that the ball moves to the right, bounces off the wall, and then moves to the left, continuing off into space. Note that our test is not very sophisticated; because `ball.pos.x` is at the center of the ball and `wallR.pos.x` is at the center of the wall, if you look closely you can see that the ball appears to penetrate the wall slightly. You could if you wish make the test more precise by using the radius of the ball and the thickness of the wall.

> Add another wall at the left side of the display, and give it the name `wallL`. Make the ball bounce off that wall also.



You need to create the left wall near the beginning of the program, *before* the while loop. If you put the statement inside the loop, a new wall would be created every time the loop was executed. A very large number of walls would be created, all at the same location. While we wouldn't be able to see them, the computer would try to draw them, and this would slow the program down considerably.

> Next, before the while loop, change the initial velocity to have a nonzero y component:

```
ball.velocity = vector(25,5,0)
```

When you run the program, the ball bounces even where there is no wall! Again, the issue is that VPython doesn't know any physics. If we tell it to make the ball change direction when the ball's position is to the right of the right wall's position, VPython goes ahead and does that, whether that makes physics sense or not. We'll fix this later.

10 Visualizing velocity

We will often want to visualize vector quantities, such as the ball's velocity. We can use an `arrow` object to visualize the velocity of the ball.

> Before the `while` loop, but after the program statement that sets the ball's velocity, create an arrow, which you will use to visualize the velocity vector for the ball. The tail of the arrow is at the location given by `pos`, and we choose that to be the location of the ball. The tip of the arrow is at the location that is a vector displacement `axis` away from the tip (in this case, the location of the ball plus the velocity of the ball).

```
varr = arrow(pos=ball.pos, axis=ball.velocity, color=color.yellow)
```

It is important to create the arrow *before* the `while` loop. If we put this statement in the indented code after the `while`, we would create a new arrow in every iteration. We would soon have a large number of arrows, all at the same location. This would make the program run very slowly.

> Run your program.

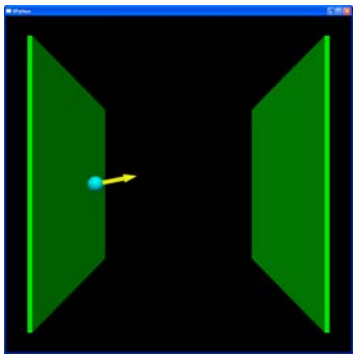
You should see a yellow arrow with its tail located at the ball's initial position, pointing in the direction of the ball's initial velocity. However, the arrow is huge, and it completely dominates the scene. The problem is that velocity in meters per second and position in meters are basically different kinds of quantities, and we need to scale the velocity in order to fit it into the diagram.

Let's scale down the size of the arrow, by multiplying by a "scalar", a single number. Multiplying a scalar times a vector changes the magnitude of a vector but not its direction, since all components are scaled equally.

> Change your arrow statement to use a scale factor to scale the axis, like this, then run the program:

```
vscale = 0.1
varr = arrow(pos=ball.pos, axis=vscale*ball.velocity, color=color.yellow)
```

Run the program. Now the arrow has a reasonable size, but it doesn't change when the ball moves. We need to update the position and axis of the arrow every time we move the ball.



> Inside the `while` loop, update the position and axis attributes of the arrow named `varr`, so that the tail of the arrow is always on the ball, and the axis of the arrow represents the current vector velocity. Remember to use the scale factor `vscale` to scale the axis of the arrow. Run the program.

The arrow representing the ball's velocity should move with the ball, and should change direction every time the ball collides with a wall.

11 Autoscaling

By default, VPython "autoscales" the scene, by continually moving the camera forwards or backwards so that all of the scene is in view. If desired, you can turn off autoscaling after creating the initial scene.

> Just before the `while` loop, after drawing the initial scene, turn off further autoscaling:

```
scene.autoscale = False
```

12 Leaving a trail

Often we are interested in the trajectory of a moving object, and would like to have it leave a trail. We can make a trail out of a `curve` object. A `curve` is an ordered list of points, which are connected by a line (actually a thin tube). We'll create the `curve` object before the loop, and append a point to it every time we move the ball.

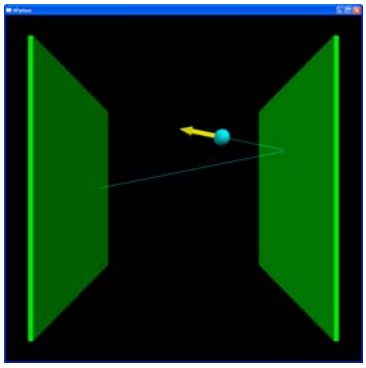
> After creating the ball, but before the `while` loop, insert the following statement to create a `curve` object

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

This creates a `curve` object, associated with the ball, whose color is the same as the color of the ball, but without any points in the curve as yet.

> Inside the while loop, after updating the ball's position, add this statement:

```
ball.trail.append(pos=ball.pos)
```



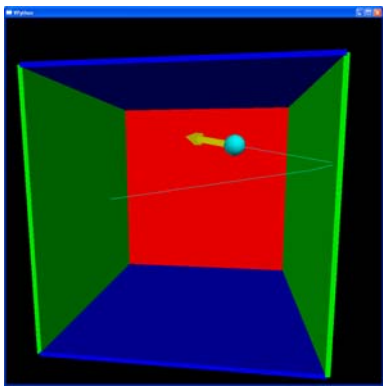
This statement appends a point to the trail. The position of the added point was chosen to be the same as the current position of the ball. Run your program. You should see a cyan trail behind the ball.

13 How your program works

In your `while` loop you continually change the position of the ball (`ball.pos`); you could also have changed its color or its radius. While your code is running, VPython runs another program (a "parallel thread") which periodically uses information about the current attributes of your objects, such as the attributes `ball.pos` and `varr.axis`, to draw a new scene. This additional program (the "rendering thread") does the necessary computations to figure out how to draw the current version of the scene on the computer screen, and instructs the computer to draw this picture. This happens many times per second, so the animation appears continuous.

14 Making the ball bounce around inside a box

With the program you've written so far, the ball escapes and bounces off nothing. To make a more realistic model of the motion, do the following:



- Add top and bottom walls, and make the ball bounce off these walls.
- Make the walls touch, forming part of a large box.
- Add a back wall, and an "invisible" front wall, and make the ball bounce off these walls. Do not draw a front wall, but include an `if` statement to prevent the ball from coming through the front.
- Give your ball a component of velocity in the z direction as well, so that it will bounce off the back and front walls. Make the initial velocity of the ball be this:

```
ball.velocity = vector(25,5,15)
```

The completed program should include these features:

- Complete box (except that the front is open).
- Correct initial velocity.
- Continuous display of velocity arrow that moves with the ball.
- Ball leaves a trail.

It is worth pointing out that this program represents a simple model of a gas in a container. The pressure on the walls of the container is due to the impacts of large numbers of gas molecules hitting the walls every second.

15 Playing around

Here are some suggestions of things you might like to play with after completing your program.

- You can make the program run much longer by changing to `while t < 3e4:` or you can make it run "forever" by changing to `while True:`, since the condition is always `True`.

- You might like to add some more balls, with different initial positions and initial velocities. Inside the loop you will need position updates for each ball.
- You could make a "Pong" game by moving walls around under mouse or keyboard control. See the sections on Mouse Interactions and Keyboard Interactions in the on-line Visual reference manual, accessible from the Help menu in IDLE.
- You can design your own colors using fractional values for the amount of red, green, and blue. For example, if you specify color=(1, 0.7, 0.7) you get a kind of pink (100% red, 70% green, 70% blue). The example program colorsliders.py lets you experiment with colors.
- You could change the color of the ball or the wall whenever there is a collision.
- After the statement from `visual import *`, add the statement `scene.stereo = 'redcyan'` (or use `'redblue'`) and your scene will be in stereo for use with red-cyan (or red-blue) glasses (red lens on left eye). You can also specify `'passive'` or `'crosseyed'` for walleyed or crosseyed stereo.

16 Example programs

If you quit VPython and restart it, you should find that choosing "Open" on the File menu gives you access to a large number of example programs written in VPython, which may suggest useful ideas.