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 by David Scherer while he was a 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, which allows you to enter computer code, try your program, and get information about your program. IDLE is currently not available on pre-OSX Macintosh.

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

2 Spinning and Zooming

> Start IDLE by double-clicking on the snake icon on your desktop. A window labeled "Untitled" should appear. Pick Open on the File menu and choose the program **randombox.py.** Press F5 to run this program.
(On Linux or Macintosh OSX, type **visual-demos** in a typescript to start IDLE in the demo folder. On pre-OSX Macintosh, drag the program **randombox.py** onto **PythonInterpreter**)

Hold down the right mouse button (shift key on Macintosh) and drag the mouse to spin the "camera" around the scene. Hold down the middle mouse button (left+right buttons on a two-button mouse; control key on Macintosh) and drag the mouse to zoom into and out of the scene. These navigation tools are built into all VPython programs unless specifically disabled by the author of the program.

> Close all the VPython windows to start over.

3 Your First Program

> Start IDLE by double-clicking on the snake icon on your desktop. A window labeled "Untitled" should appear. This is the window in which you will type your program. (On pre-OSX Macintosh, use an editor of your choice.)
(On Linux or Macintosh OSX, type **idle** in a typescript to start IDLE in the current folder.)

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

```
from visual import *
```

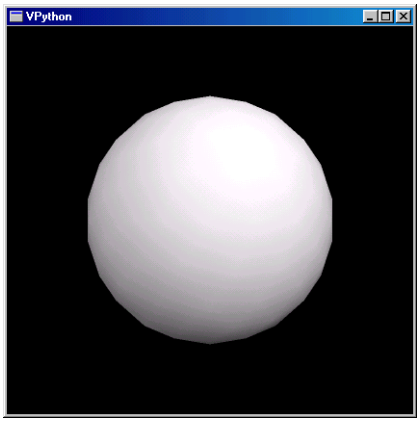
This statement instructs Python to use the Visual graphics module.

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

```
sphere()
```

4 Running the Program

> Now run your program by pressing F5 (or by choosing "Run program" from the "Run" menu). (On pre-OSX Macintosh, save your program. In the Finder, drag your program file onto **PythonInterpreter**.)



When you run the program, two new windows appear. There is a window titled "VPython," in which you should see a white sphere, and a window titled "Output". Move the Output 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 "middle" mouse button and move the mouse. You should see that you are able to zoom into and out of the scene. (On a 2-button mouse, hold down both left and right buttons; on a 1-button mouse, hold down the control key and the mouse button.)

> Now try holding down the right mouse button (hold down shift key with a one-button mouse). You should find that you are able to rotate around the sphere (you can tell that you are moving because the lighting changes).

To avoid confusion about motion in the scene, it is helpful to keep in mind the idea that you are moving a camera around the object. When you zoom and rotate you are moving the camera, not the object.

5 Stopping the Program

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

6 Save Your Program

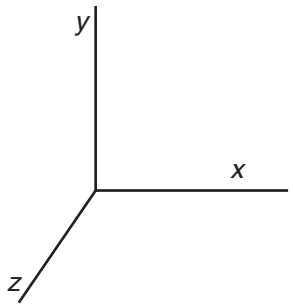
Save your program by pulling down the File menu and choosing Save As. Give the program a name ending in ".py," such as "MyProgram.py". You must type the ".py" extension; IDLE will not supply it. Every time you run your program, IDLE will save your code before running. You can undo changes to the program by pressing CTRL-z.

7 Modifying Your Program

A single white sphere isn't very interesting. Let's change the color, size, and position of the sphere.

> Change the second line of your program to read:

```
sphere(pos=(-5,0,0), radius=0.5, color=color.red)
```



What does this line of code do? To position objects in the display window we set their 3D cartesian 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. The assignment

```
pos=(-5,0,0)
```

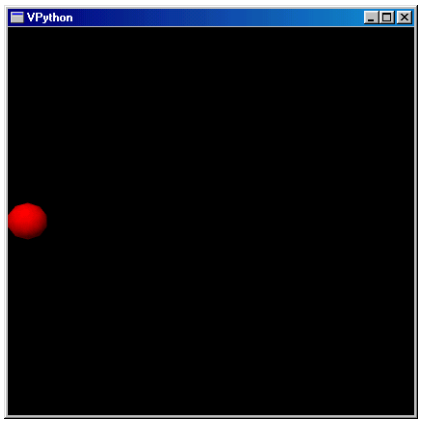
sets the position of the sphere by assigning values to the *x, y, and z* coordinates of the center of the sphere. The assignment

```
radius=0.5
```

gives the sphere a radius of 0.5 of the same units. Finally,

```
color = color.red
```

makes the sphere red (there are 8 colors easily accessible by `color.xxx`: red,green, blue, yellow, magenta, cyan, black, and white).



> Now press F5 to run your program. Note that VPython automatically makes the display window an appropriate size, so you can see the sphere in your display.

7.1 Objects and attributes

The sphere created by your program is an *object*. Properties like `pos`, `radius`, and `color` are called *attributes* of the object.

Now let's create another object. We'll make a wall, to the right of the sphere.

> Add the following line of code at the end of your program.

```
box(pos=(6,0,0), size=(0.2,4,4), color=color.green)
```

> Before running your program, try to predict what you will see in your display. Run your program and try it.

7.2 Naming objects

In order to refer to objects such as the sphere and the box, we need to give them names.

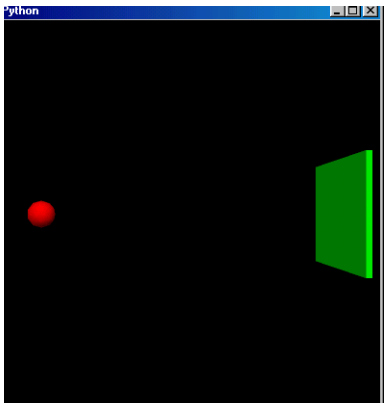
> To name the sphere "ball", change the statement that creates the red sphere to this:

```
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.red)
```

> Similarly, give the box the name "wallR" (for right wall).

Your program should now look like this:

```
from visual import *
sphere(pos=(-5,0,0), radius=0.5, color=color.red)
wallR = box(pos=(6,0,0),size=(0.2,4,4),color=color.green)
```



If you run your program you should find that it runs as it did before.

8 Order of Execution

You may already know that when a computer program runs, the computer starts at the beginning of the program and executes each statement in the order in which it is encountered. In Python, each new statement begins on a new line. Thus, in your program the computer first draws a red sphere, then draws a green box (of course, this happens fast enough that it appears to you as if everything was done simultaneously). When we add more statements to the program, we'll add them in the order in which we want them to be executed. Occasionally we'll go back and insert statements near the beginning of the program because we want them to be executed before statements that come later.

9 Animating the Ball

We would like to make the red 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 a time interval between "snapshots." We'll call this very short time interval "dt". In the context of the program, we are talking about virtual time (i.e. time in our virtual world); a virtual time interval of 1 second may take much less than one second on a fast computer.

> Type this line at the end of your program:

```
dt = 0.05
```

We also 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 will 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.

> Type this line at the end of your program:

```
ball.velocity = vector(2,0,0)
```

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.

Since distance = speed*time, we can calculate how far the ball moves (the displacement of the ball) in time dt by multiplying ball.velocity by dt. To find the ball's new position, we add the displacement to its old position:

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

Note that just as velocity is a vector, so is the position of the ball.

9.3 The meaning of the equal sign

If you are new to programming, the statement

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

may look very odd indeed. Your math teachers would surely have objected had you written `a = a + 2`.

In a Python program (and in many other computer languages), the equal sign means "assign a value to this variable." When you write:

```
a = a + 2
```

you are really giving the following instructions: Find the location in memory where the value of the variable *a* is stored. Read up that value, add 2 to it, and store the result in the location in memory where the value of *a* is stored. So the statement:

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

really means: "Find the location in memory where the position of the object *ball* is stored. Read up this value, and add to it the result of the vector expression `ball.velocity*dt`. Store the result back in the location in memory where the position of the object *ball* is stored.

9.4 Running the animation

Your program should now look like this:

```
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.red)
wallR = box(pos=(6,0,0), size=(0.2,4,4), color=color.green)
dt = 0.05
ball.velocity = vector(0.2,0,0)
ball.pos = ball.pos + ball.velocity*dt
```

> 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.

> Delete the last line of your program. Now type:

```
while (1==1):
```

Don't forget the colon! Notice that when you press return, the cursor appears at an indented location after the while. 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 number 1 is equal to 1, or forever. We can stop the loop by quitting the program.

> Indented under the while statement, type this:

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

(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 simply press TAB to indent a line.) All indented statements after a `while` statement will be executed every time the loop is executed.

Your program should now look like this:

```
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.red)
wallR = box(pos=(6,0,0), size=(0.2,4,4), color=color.green)
dt = 0.05
ball.velocity = vector(2,0,0)
while (1==1):
   ball.pos = ball.pos + ball.velocity*dt
```

> Run your program.

You should observe that the ball moves to the right, quite rapidly.

> To slow it down, insert the following statement inside the loop (after the `while` statement):

```
rate(100)
```

This specifies that the `while` loop will not be executed more than 100 times per 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 we told it to do.

10 Making the ball bounce: Logical tests

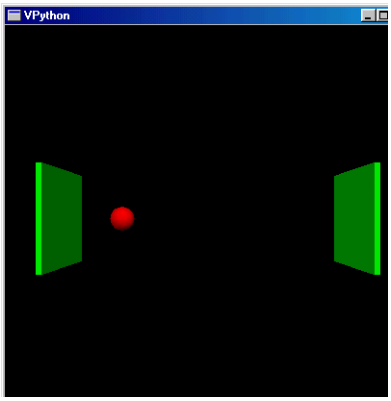
To make the ball bounce off the wall, we need to detect a collision between the ball and the wall. A simple approach is to compare the *x* coordinate of the ball to the *x* coordinate of the wall, and reverse the *x* component of the ball's velocity if the ball has moved too far to the right. We can use a logical test to do this:

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

The indented line after the "if" statement will be executed 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.

However, since we want this logical test to be performed every time the ball is moved, we need to indent both of these lines, so they are inside the while loop. Insert tabs before the lines, or select the lines and use the "Indent region" option on the "Format" menu to indent the lines. Your program should now look like this:

```
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.red)
wallR = box(pos=(6,0,0), size=(0.2,4,4), color=color.green)
dt = 0.05
ball.velocity = vector(2,0,0)
while (1==1):
   rate(100)
   ball.pos = ball.pos + ball.velocity*dt
   if ball.x > wallR.x:
        ball.velocity.x = -ball.velocity.x
```



> 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.x is at the center of the ball and wallR.x is at the center of the wall, the ball appears to penetrate the wall slightly.

> Add another wall at the left side of the display, and make the ball bounce off that wall also.

Your program should now look something like the following:

```
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.red)
wallR = box(pos=(6,0,0), size=(0.2,4,4), color=color.green)
wallL = box(pos=(-6,0,0), size=(0.2,4,4), color=color.green)
dt = 0.05
ball.velocity = vector(2,0,0)
while (1==1):
   rate(100)
   ball.pos = ball.pos + ball.velocity*dt
   if ball.x > wallR.x:
        ball.velocity.x = -ball.velocity.x
   if ball.x < wallL.x:
        ball.velocity.x = -ball.velocity.x
```

Note that we inserted the statement to draw the left wall near the beginning of the program, before the while loop. If we had put the statement after the "while" (inside the loop), a new wall would be created every time the loop was executed. We'd end up with thousands of walls, 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.

11 Making the ball move at an angle

To make the program more interesting, let's make the ball move at an angle.

> Change the ball's velocity to make it move in the *y* direction, as well as in the *x* direction. Before reading further, try to do this on your own.

Since the ball's velocity is a vector, all we need to do is give it a nonzero y component. For example, we can change the statement:

```
ball.velocity = vector(2,0,0)
```
to
```
ball.velocity = vector(2,1.5,0)
```

Now the ball moves at an angle. Unfortunately, it now misses the wall! However, you can fix this later by extending the wall, or by adding a horizontal wall above the other walls.

12 Visualizing velocity

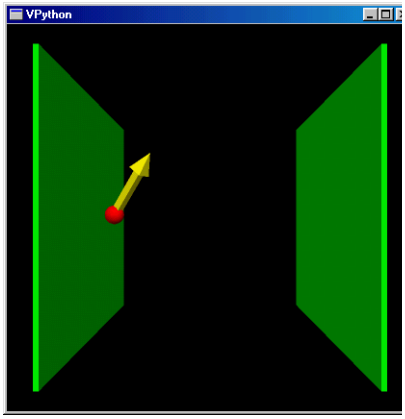
We will often want to visualize vector quantities, such as the ball's velocity. We can use an `arrow` to visualize the velocity of the ball. Before the `while` statement, but after the program statement setting the ball's velocity,

```
ball.velocity = vector(2,1.5,1)
```

we can create an arrow:

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

It's 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 thousands of arrows, all at the same location! This would make our 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, this arrow doesn't change when the ball moves. We need to update the position and axis of the velocity vector every time we move the ball.

> Inside the `while` loop, after the last line of code, add these lines:

```
bv.pos = ball.pos
bv.axis = ball.velocity
```

The first of these lines moves the tail of the arrow to the location of the center of the ball. The second aligns the arrow with the current velocity of the ball.

Let's also make the walls a bit bigger, by saying `size=(0.2,12,12)`, so the ball will hit them. Your program should now look like this:

```
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.red)
wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)
wallL = box(pos=(-6,0,0), size=(0.2,12,12), color=color.green)
dt = 0.05
ball.velocity = vector(2,1.5,1)
bv = arrow(pos=ball.pos, axis=ball.velocity, color=color.yellow)
while (1==1):
   rate(100)
   ball.pos = ball.pos + ball.velocity*dt
   if ball.x > wallR.x:
        ball.velocity.x = -ball.velocity.x
```

```
    if ball.x < wallL.x:
        ball.velocity.x = -ball.velocity.x
    bv.pos = ball.pos
    bv.axis = ball.velocity
```

> 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.

13 Leaving a trail

Sometimes 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 before the loop, and add a point to it every time we move the ball.

> After creating the ball, but before the loop, add the following line:

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

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

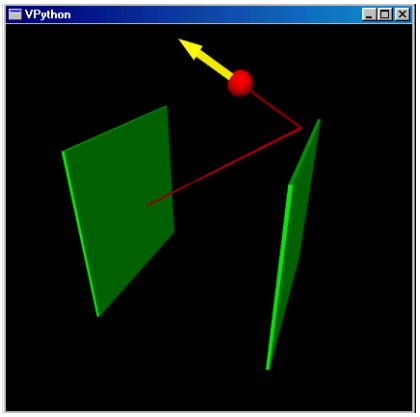
> At the end of the loop, add the following statement (indented):

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

This statement adds a point to the trail. The position of the point is the same as the current 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.red)
wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)
wallL = box(pos=(-6,0,0), size=(0.2,12,12), color=color.green)
dt = 0.05
ball.velocity = vector(2,1.5,1)
bv = arrow(pos=ball.pos, axis=ball.velocity, color=color.yellow)
ball.trail = curve(color=ball.color)

while (1==1):
    rate(100)
    ball.pos = ball.pos + ball.velocity*dt
    if ball.x > wallR.x:
        ball.velocity.x = -ball.velocity.x
    if ball.x < wallL.x:
        ball.velocity.x = -ball.velocity.x
    bv.pos = ball.pos
    bv.axis = ball.velocity
    ball.trail.append(pos=ball.pos)
```



> Run your program.

You should see a red trail behind the ball.

14 Optional: 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 gets information about the attributes of your objects, such as ball.pos and ball.radius. This program does the necessary computations to figure out how to draw your scene on the computer screen, and instructs the computer to draw this picture. This happens many times per second, so the animation appears continuous.

15 Making the ball bounce around inside a large box

> You are now at a point where you can try the following modifications to your program.

- Add top and bottom walls.

- Expand all the walls so they touch, forming part of a large box.
- Add a back wall, and an "invisible" front wall. That is, 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, and make it bounce off the back and front walls.

The demo program "bounce2.py", which you can find in the VPython demo folder, is an example of a program which does this.