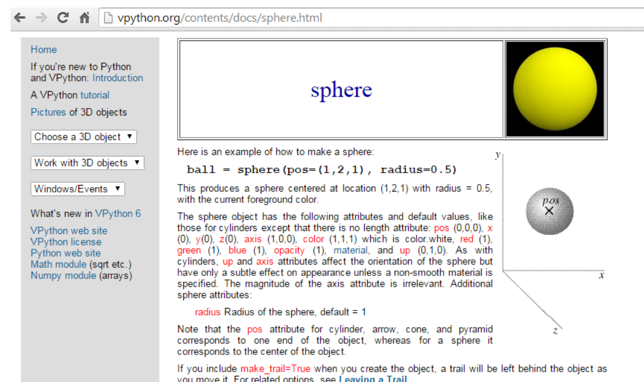
Visual Python

Visual Python is a library of 3D objects you can program in Python to do all kinds of cool stuff using the tools you've learned. Download and install the version for your computer at Vpython.org. The documentation on the Vpython website is excellent. Check it out to learn how to create and manipulate objects and see the examples.

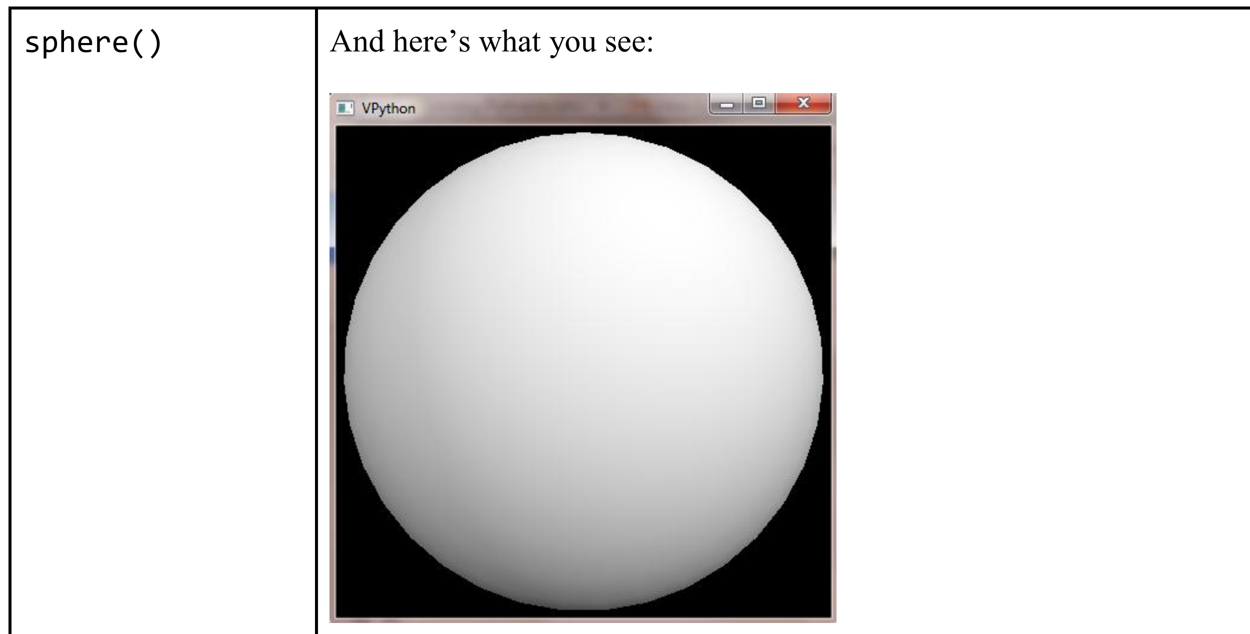


Open "VIDLE," the editor it puts on your desktop.

You always start off by importing the visual module this way:

```
from visual import *
```

Then you can create an object like a sphere or a box. A sphere is easy:

`sphere()`	And here's what you see: 

You can rotate the camera view by moving your mouse while holding down the right-mouse button. You can zoom in and out of the scene by moving the mouse while holding down both mouse buttons.

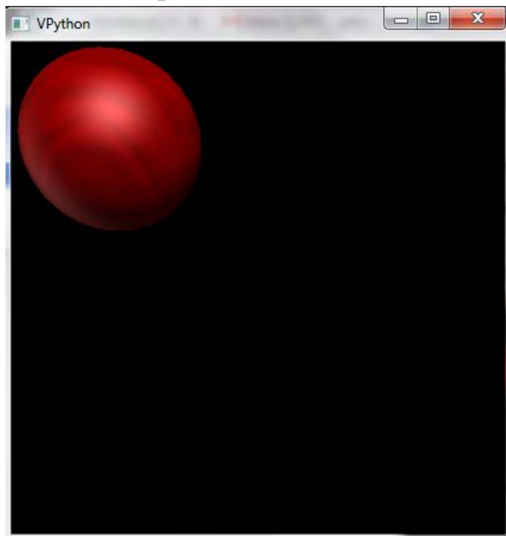
You can give a sphere attributes like position, radius, color and material:

```
sphere(pos = (-5, 5, 2),
      radius = 3,
      color = color.red,
      material = materials.wood)
```

It's best to name your objects so if you want to change one, you can point right to it.

```
ball = sphere(pos = (-5, 5, 2),
      radius = 3,
      color = color.red,
      material = materials.wood)
```

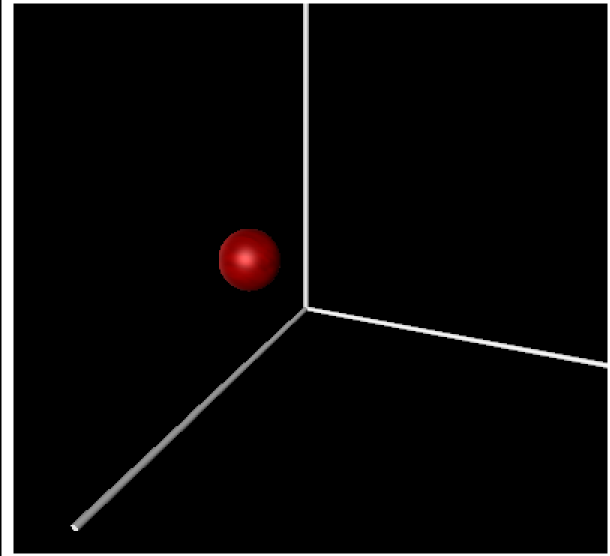
Here's the sphere now:



Program: Creating a Grid

Let's make the x-y-z axes using cylinders. As you can see on the Vpython documentation pages, to make a cylinder all you need is position, axis and radius. Position and radius are easy to understand, but the axis is the direction and the length of the cylinder. For my axes I'm making the cylinders long and thin. Add this code under the ball code.

```
xaxis = cylinder(pos=(0,0,0),
                 axis=(30,0,0),
                 radius = 0.25)
yaxis = cylinder(pos=(0,0,0),
                 axis=(0,30,0),
                 radius = 0.25)
zaxis = cylinder(pos=(0,0,0),
                 axis=(0,0,30),
                 radius = 0.25)
```



Vectors

Vectors are extremely useful tools in mathematics and VPython defines lots of things as vectors. The position of an object is already a vector. So if we wanted to move our red ball down 5 units, we could just add a vector:

```
ball.pos += vector(0,-5,0)
```

Run it and the ball will move 5 units downward. We could put that line of code in a loop and make the ball keep moving downwards:

```
for i in range(10):
    ball.pos += vector(0,-5,0)
```

But that might make it move too quickly to see. You can put a "rate" line at the beginning of the loop to slow it down:

```
for i in range(10):
    rate(20) #slows down the frame rate
    ball.pos += vector(0,-5,0)
```

You can change the position of objects using vectors or by making the position a function of a variable such as t for time.

Here's a good use of Trigonometry: make the ball move in a circular path. You can use sines and cosines. Change the loop to:

```
t = 0 #starting t
dt = 0.1 #time steps
for i in range(10):
    rate(10) #slows down the frame rate
    ball.pos = (5 + 10*sin(t),0,0) #make ball oscillate horizontally
    t += dt #increment time
```

Not much happening? Let's add a vertical oscillation, too. Change the ball.pos line to:

```
ball.pos = (5 + 10*sin(t),5 + 10*cos(t),0)
```

Now you can see the ball traveling in a circular path.

Falling Objects

Using Visual Python and vectors you can answer questions on how long it takes an object to reach the ground, like this one:

If an object is dropped from a height of 100 meters, how long will it take to reach the ground? (Neglect air resistance.)

We'll use the code from the above example and change the initial position:

```
ball = sphere(pos = (0, 100, 0), #ball is 100 meters up
```

Change the length of the y-axis:

```
yaxis = cylinder(pos=(0,0,0),
                axis=(0,100,0),
                radius = 0.25)
```

The only force operating on an object is gravity. We'll create a vector for the force of gravity:

```
fgrav = vector(0,-9.8,0)
```

We'll also create a vector for the velocity of the ball. Its initial velocity is 0.

```
ballv = vector(0,0,0) #initial velocity of the ball
```

We'll be running a loop and we want to know how long it takes for the ball to hit the ground. That means we need a variable for time, and a variable for the increment of time.

```
t = 0
```

```
dt = 0.1
```

Here’s the loop. We only need the loop to continue until the ball’s y-coordinate is 0.

```
while ball.y > 0:
    rate(10) #slows down the frame rate
    t += dt #make the time tick forward a fraction of a second
```

Next we’ll update the ball’s velocity by the acceleration due to gravity:

```
    ballv += fgrav*dt
```

Then we’ll update the ball’s position by its velocity. Notice everything is multiplied by dt since we’re dealing with a fraction of a second here. If the ball’s velocity is -5 meters per second, we have to realize it’s only going to move the ball a tenth of that distance in a tenth of a second.

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

Finally the ball will reach the ground and the loop will terminate. Outside the loop, we want to print the time t.

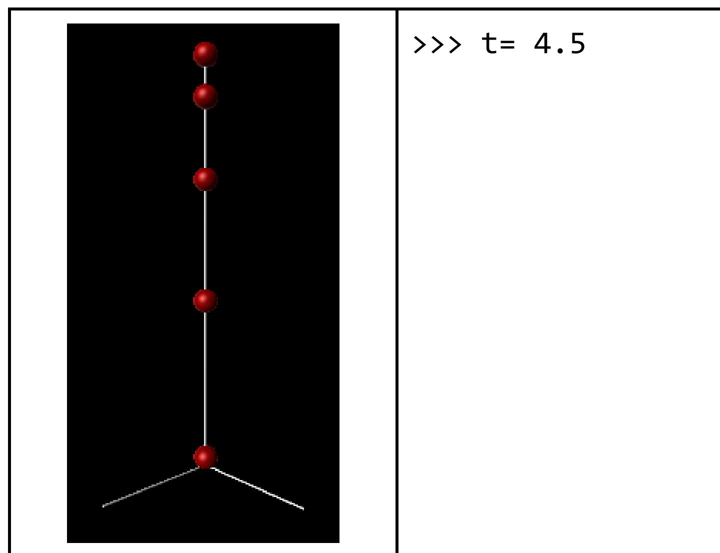
```
print “t=”,t
```

Here’s the entire code:

```
from visual import *

scene = display(center = (0,50,0)) #Use the whole screen!
ball = sphere(pos = (0, 100, 0),
      radius = 3,
      color = color.red,
      material = materials.wood)

xaxis = cylinder(pos=(0,0,0),
                 axis=(30,0,0),
                 radius = 0.25)
yaxis = cylinder(pos=(0,0,0),
                 axis=(0,100,0),
                 radius = 0.25)
zaxis = cylinder(pos=(0,0,0),
                 axis=(0,0,30),
                 radius = 0.25)
```

```
fgrav = vector(0,-9.8,0) #acceleration due to gravity
ballv = vector(0,0,0) #initial velocity of the ball

t = 0
dt = 0.1

while ball.y > 0:
    t += dt
    rate(10) #slows down the frame rate
    ballv += fgrav*dt #update ball’s velocity by gravity
    ball.pos += ballv*dt #update ball’s position by its velocity

print “t=”,t
```

Run it, and you’ll see the ball fall. You’ll also get a printout:



Is that right? Let’s check it algebraically. According to the position formula, the height is calculated as

$$h = h_0 + V_0 t - 4.9t^2$$

Plugging in our values, we get

$$h = 100 + (0)t - 4.9t^2$$

Plugging that into our quadratic solver (Chapter 3), we get

```
>>> quad(-4.9,0,100)
-4.51753951453 4.51753951453
```

We're right! About 4.5 seconds to reach the ground. To get more accurate results, just change dt to a smaller number, like 0.01. You might want to make the rate higher, too!

```
>>> t= 4.52
```

It's easy to change the ball's position vector to drop it from different heights. But you can even throw it up or down by changing the velocity vector:

```
ballv = vector(0,20,0) #ball is thrown upwards
```

Run the program again and you'll see the ball go upwards, slow down, then fall downwards as before. When it hits the ground, the time says `t = 7.0`
According to the positive value returned by our quadratic solver, that's pretty close:

```
>>> quad(-4.9,20,100)
-2.91630930673 6.99794195979
```

You can also get the velocity when the ball hits the ground by adding the velocity to the print statement:

```
print "t=",t, ", v=",ballv
```

The printout will be

```
>>> t= 7.0 , v= <0, -48.6, 0>
```

There's no horizontal velocity, just up and down velocity (the y-value of the vector). When it hit the ground it was traveling 48.6 meters per second (the negative is for downward).

The Solar System Model

You can even use vectors is if the force is changing. For example, Newton said the force of attraction between two bodies is proportional to the product of their masses and inversely proportional to the square of the distance between them:

$$\vec{F} = -\frac{GmM}{r^2}\hat{r}$$

$\vec{F}$ is the vector of the force of attraction between the Sun and Earth.
G is the Gravitational Constant. It's really small.
m is the mass of the earth. It's really big.
M is the mass of the Sun. It's really, really big.
r is the distance between the two bodies
$\hat{r}$ is the Unit Radial Vector, the one-unit long vector going from the Sun to the Earth. The negative sign means the force of gravity is going in the opposite direction, from the Earth towards the Sun.

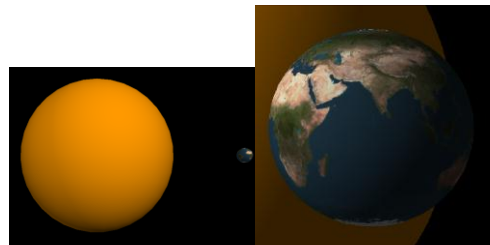
Let's create some spheres to be our Sun and Earth. We'll also create some vectors to represent the Earth's motion and the gravitational force pulling it into the Sun.

```
from visual import *

#Set up the display window:
scene = display(width = 1000, height = 1000)

#Create our objects:
sun = sphere(pos=(0,0,0), radius = 100, color = color.orange,
                                material = materials.emissive)
```

You can use the actual values from an astronomical data chart if you like, but I just want it to look realistic. Speaking of realistic, you can just choose a color for the Earth (blue? green?) but run it and check out the "earth" material:

```
earth = sphere(pos=(200,0,0),
      radius = 10,
      material = materials.earth,
      make_trail = True)
```

Pretty realistic!



Now we'll create a vector for the Earth's initial velocity:

```
#Earth's initial velocity
earth.v = vector(0,-1,-8)
```

And we'll just start the loop:

```
for i in range (3000):
     rate(50) #Slows down the speed
```

Now we need to calculate the distance between the Sun and Earth:

```
     #Earth-Sun distance (Pythagorean formula in 3D)
     dist = (earth.x**2 + earth.y**2 + earth.z**2)**0.5
```

We need to calculate the Unit Radial Vector. Or I should say we'll get VPython to calculate it:

```
     #Calculate Unit Radial Vector
     unitRadialVector = (earth.pos – sun.pos)/dist
```

The gravitational constant G is really really small and the masses of the Earth and Sun are really, really big. But they're all constants so I just replaced them all with one number. You can play around and see what number gives you the best paths!

```
     #Calculate Force of Gravity:
     Fgrav = -10000*unitRadialVector/dist**2
```

Now we'll move the Earth around:

```
     #Update Earth’s velocity by Force of Gravity:
     earth.v += Fgrav
     #Update Earth’s position by its velocity:
     earth.pos += earth.v
     if dist <= sun.radius: break #If the Earth hits the Sun, stop!
```

Run this and you should see a perfect elliptical orbit! Change the "make_trail = True" to "make_trail=False" to view it without the Earth's trail.

Lights

If you're a stickler for realism, you can turn off the ambient light and create a local light source at the Sun's position, the origin. Change the "display" code to this:

```
scene = display(width = 1000, height = 1000,
                      lights = [])
lamp = local_light(pos=(0,0,0))
```

Now you can say you proved Newton right!