Simple 3D Programming Using VPython

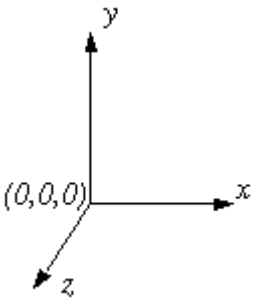
I. VPython: the Python/ Visual / IDLE environment

The interactive development environment you will use is called "IDLE."

The Display window

When using VPython the display window shows objects in 3D.

(0,0,0) is in the center of the display window . The +x axis runs to the right, the +y axis runs up, and the +z axis points out of the screen, toward you.



x, y, and z are measured in whatever units you choose; the display is automatically scaled appropriately. (You could, for example, create a sphere with a radius of 1E-15 m to represent a nucleus, or a sphere with a radius of 1E6 m to represent a planet, though it wouldn't make sense to put both of these objects in the same display!)

The Output window

The output of any -print- statements you execute in your program goes to the Output window, which is a scrolling text window. You can use this window to print values of variables, print lists, print messages, etc. Place it where you can see messages in it.

The Code window

If you type the following simple program into the code window in IDLE and run it (press F5, or use the Run menu), you will see a display like the one shown in the figure below.

```
from visual import *
redbox=box(pos=vector(4,2,3),
     size=(8.,4.,6.),color=color.red)
greenball=sphere(pos=vector(4,7,3), radius=2,
     color=color.green)
```

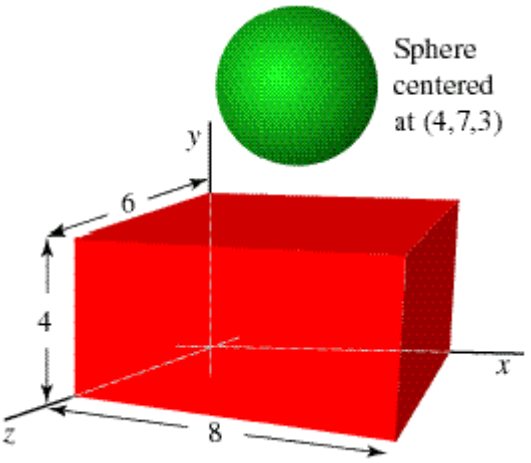
Visual is the name of the 3D graphics module used with the Python programming language. VPython is the name of the combination of the Python programming language, the Visual module, and the development environment IDLE.

Viewing the scene

In the display window, click and drag with the right mouse button (hold down the shift key on a Macintosh). Drag left or right, and you rotate around the scene. To rotate around a horizontal axis, drag up or down with the right mouse button. Click and drag up or down with the middle mouse button to move closer to the scene or farther away (on a 2-button mouse, hold down the left and right buttons; on a 1-button mouse, hold down the CTRL key).

II. Visual Entities

Objects, names, and attributes



The graphical objects you create, such as spheres, boxes, and curves, continue to exist for the duration of your program, and the Visual 3D graphics module will continue to display them, wherever they are. You must give each object a name (such as redbox or greenball in the example above) if you wish to refer to it again later in your program. All objects have attributes: properties like greenball.pos (the position of the sphere), greenball.color, and radius or other size parameter. If you change an attribute of an object, such as its position or color, Visual will automatically display the object in its new location, or with its new color.

You can set the values of attributes in the "constructor" (the code used to create the object), and you can also modify attributes later:

```
greenball.radius = 2.2
```

In addition to the built-in set of attributes, you may create new attributes. For example, you might create a sphere named moon; in addition to its radius and location, you might give it attributes such as mass (moon.mass) and momentum (moon.momentum).

Vectors

Not all objects in Visual are visible objects. For example, Visual allows you to create 3D vector quantities, and to perform vector operations on them. If you create a vector quantity called ***a***, you may refer to its components as ***a.x***, ***a.y***, and ***a.z***. To add two vectors, ***a*** and ***b***, however, you do not need to add the components one by one; Visual will do the vector addition for you:

```
a = vector(1.,2.,3.)
b = vector(4.,5.,6.)
c=a+b
```

If you print ***c*** , you will find that it is a vector with components (5.,7.,9.).

Scalar multiplication

```
d = 3.*a # d is a vector with components (3., 6., 9.)
```

Vector magnitude

```
d = mag(c) # d is a scalar
z = mag(c)**2 # you can't square a vector; just its
     magnitude
```

Vector products

```
f = cross(a,b) # cross product
g = dot(a,b) # dot product
h = norm(a) # normalized (unit) vector
```

The attributes of Visual objects can be vectors, such as velocity or momentum.

III. Simple Python Programming

Importing the 3D Graphics Module (Visual)

The first line of your program must be:

```
from visual import *
```

Comments

A comment in a Python program starts with "#"

```
# this line is a comment
```

Variables

Variables can be created anywhere in a Python program, simply by assigning a variable name to a value. The type of the variable is determined by the assignment statement.

```
a = 3 # an integer
b = -2. # a floating-point number
c = vector(0.4, 3e3, -1e1) # a vector
Earth = sphere(pos=(0,0,0), radius=6.4e6) # a Visual
     object
bodies = [ship, Earth, Moon] # a list of objects
```

Basic Visual objects such as sphere() and box() have a set of attributes such as color, and you can define additional attributes such as mass or momentum. Other objects, such as vector(), have built-in attributes but you cannot create additional attributes. Numbers, lists, and other built-in objects do not have attributes at all.

Warning about division

Division of integers will not come out the way you may expect, since the result is rounded down to the nearest integer. Thus:

```
a = 2/3
print a # a is 0
```

To avoid this, place a decimal point after every number, like this:

```
b = 2./3.
print b # b is 0.6666667, as expected
```

We recommend putting the following statement at the start of your program, in which case 2/3 will be 0.6666667; there are two underscores before and after the word "future":

```
from __future__ import division
```

Exponentiation

```
x**2 # Not x^2
10**-2 gives an error; use 10.**-2
```

Logical Tests

If, elif ("else if"), else:

```
if a == b: # see table of logical expressions below
    c = 3.5 # indented code executed if test is true
elif a < b:
    c = 0. # c will be set to zero only if a < b
else:
    c = -23.2
```

Logical expressions

==	equal
!=	not equal (also <>)
<	less than
>	greater than
<=	less than or equal
>=	greater or equal
or	logical or
and	logical and
in	member of a sequence
not in	not sequence member

Lists

A list is an ordered sequence of any kind of object. It is delimited by square brackets.

```
moons = [Io, Europa, Ganymede, Callisto]
```

The function "arange" (short for "arrayrange") creates a sequence of numbers:

```
angles = arange (0., 2.*pi, pi/100.)
# numbers from 0. to 2.*pi-(pi/100.) in steps of
     (pi/100.)

numbers = arange(10) # integer argument -> integers
print numbers # [0,1,2,3,4,5,6,7,8,9]
```

Loops

The simplest loop in Python is a "while" loop. The loop continues as long as the specified logical expression is true:

```
while x < 23:
    x = x + vx*dt
```

To write an infinite loop, just use a logical expression that will always be true:

```
while 1==1:
    ball.pos = ball.pos + (ball.momentum/ball.mass)*dt
```

Since the value assigned to a true logical expression is 1, the following also produces an infinite loop:

```
while 1:
    a = b+c
```

Infinite loops are ok, because you can always interrupt the program by choosing "Stop Program" from the Run menu in IDLE.

It is also possible to loop over the members of a sequence:

```
moons = [Io, Europa, Ganymede, Callisto]
for a in moons:
    r = a.pos - Jupiter.pos

for x in arange(10):
# see "lists" above
...

for theta in arange(0., 2.*pi, pi/100.):
# see "lists" above
```

You can restart a loop, or terminate the loop prematurely:

```
if a == b: continue # go back to the start of the loop
if a > b: break # exit the loop
```

Printing results

To print a number, a vector, a list, or anything else, use the "print" command:

```
print Europa.momentum
```

To print a text message, enclose it in quotes:

```
print "We have just crashed on the Moon with speed", v,
     "m/s."
```

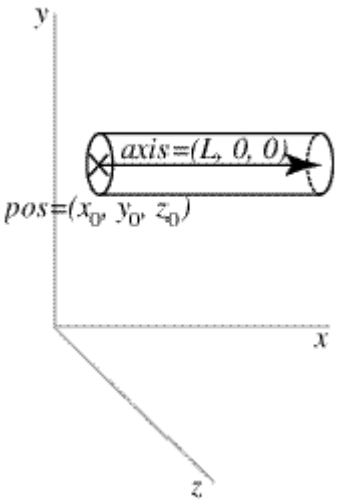
More Information about Python

See The Visual Module of VPython for detailed descriptions of all aspects of Visual.

We have summarized a very small subset of the Python programming language. Extensive Python Documentation is provided here, and there is additional information at the Python website, but much of this information assumes that you already have lots of programming experience in other languages. We recommend the following book to

those who want to learn more about Python: *Learning Python*, by Mark Lutz & David Ascher, 1999, O'Reilly.

The cylinder Object



Here is an example of how to make a cylinder, naming it "rod" for future reference:

```
rod = cylinder(pos=(0,2,1),
      axis=(5,0,0), radius=1)
```

The center of one end of this cylinder is at x=0, y=2, and z=1. Its axis lies along the x axis, with length 5, so that the other end of the cylinder is at (5,2,1), as shown in the accompanying diagram.

You can modify the position of the cylinder after it has been created, which has the effect of moving it immediately to the new position:

```
rod.pos = (15,11,9) # change position
      (x,y,z)
rod.x = 15 # only change pos.x
```

If you create an object such as a cylinder but without giving it a name such as rod, you can't refer to it later. This doesn't matter if you never intend to modify the object.

Since we didn't specify a color, the cylinder will be the current "foreground" color (see Controlling One or More Visual Display Windows). The default foreground color is white. After creating the cylinder, you can change its color:

```
rod.color = (0,0,1) # make rod be blue
```

This will make the cylinder suddenly turn blue, using the so-called RGB system for specifying colors in terms of fractions of red, green, and blue. (For details on choosing colors, see Specifying Colors.) You can set individual amounts of red, green, and blue like this:

```
rod.red = 0.4
rod.green = 0.7
rod.blue = 0.8
```

The cylinder object can be created with other, optional attributes, which can be listed in any order. Here is a full list of attributes, most of which also apply to other objects:

pos Position: the center of one end of the cylinder; default = (0,0,0)
A triple, in parentheses, such as (3,2,5)

axis The axis points from pos to the other end of the cylinder, default = (1,0,0)

x, y, z Essentially the same as pos.x, pos.y, pos.z, defaults are all 0

radius The radius of the cylinder, default = 1

length Length of axis; if not specified, axis determines the length, default = 1
If length is specified, it overrides the length given by axis

color Color of object, as a red-green-blue (RGB) triple: (1,0,0) is pure red, default = (1,1,1), which is color.white

red, green, blue (can set these color attributes individually), defaults are all 1

up Which side of the cylinder is "up"; this has only a subtle effect on the 3D appearance of the cylinder, default (0,1,0)

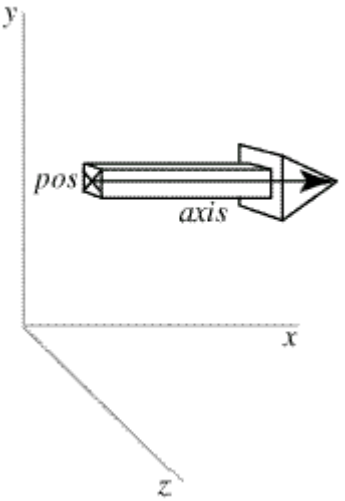
Note that the pos attribute for cylinder, arrow, cone, and pyramid corresponds to one end of the object, whereas for a box, sphere, or ring it corresponds to the center of the object.

When you start a VPython program, for convenience Visual creates a display window and names it scene. By default, objects that you create go into that display window. See Controlling One or More Visual Display Windows later in this reference for how you can create additional display windows and place objects in them.

The arrow Object

The arrow object has a straight box-shaped shaft with an arrowhead at one end. The following statement will display an arrow pointing parallel to the x axis:

```
pointer = arrow(pos=(0,2,1),
      axis=(5,0,0), shaftwidth=1)
```



The arrow object has the following attributes and default values, like those for cylinders: pos (0,0,0), x (0), y(0), z(0), axis (1,0,0), length (1), color (1,1,1) which is color.white, red (1), green (1), blue (1), and up (0,1,0). The up attribute is significant for arrow because the shaft and head have square cross sections, and setting the up attribute rotates the arrow about its axis. Additional arrow attributes:

shaftwidth By default, shaftwidth = 0.1*(length of arrow)

headwidth By default, headwidth = 2*shaftwidth

headlength By default, headlength = 3*shaftwidth

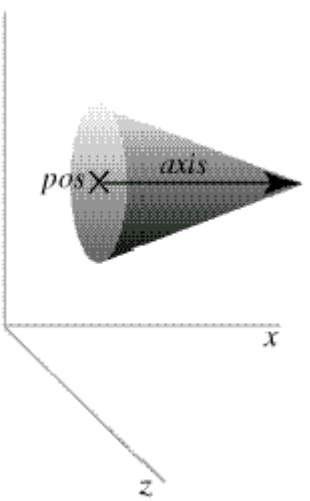
Assigning any of these attributes to 0 makes it use defaults based on the size of the arrow. If headlength becomes larger than half the length of the arrow, or the shaft becomes thinner than 1/50 the length, the entire arrow is scaled accordingly.

This default behavior makes the widths of very short arrows shrink, and the widths of very long arrows grow (while displaying the correct total length). If you prefer that shaftwidth and headwidth not change as the arrow gets very short or very long, set fixedwidth = 1. In this case the only adjustment that is made is that headlength is adjusted so that it never gets longer than half the total length, so that the total length of the arrow is correct. This means that very short, thick arrows look similar to a thumbtack, with a nearly flat head.

Note that the pos attribute for cylinder, arrow, cone, and pyramid corresponds to one end of the object, whereas for a box, sphere, or ring it corresponds to the center of the object.

The cone Object

The cone object has a circular cross section and tapers to a point. The following statement will display a cone with the center of its circular base at (5,2,0), pointing parallel to the x axis with length 12; the wide end of the cone has radius 1:




```
cone(pos=(5,2,0), axis=(12,0,0),
     radius=1)
```

The cone object has the following attributes and default values, like those for cylinders: pos (0,0,0), x (0), y(0), z(0), axis (1,0,0), length (1), color (1,1,1) which is color.white, red (1), green (1), blue (1), and up (0,1,0). As with cylinders, up has a subtle effect on the 3D appearance of a cone. Additional cone attribute:

radius Radius of the wide end of the cone, default = 1

Note that the pos attribute for cylinder, arrow, cone, and pyramid corresponds to one end of the object, whereas for a box, sphere, or ring it corresponds to the center of the object.

The pyramid Object

The pyramid object has a rectangular cross section and tapers to a point. The following statement will display a pyramid with the center of the rectangular base at (5,2,0), pointing parallel to the x axis with a base that is 6 high (in y), 4 wide (in z), and with a length 12 from base to tip:

```
pyramid(pos=(5,2,0), size=(12,6,4))
```

The pyramid object has the following attributes and default values, like those for cylinders: pos which is the center of the rectangular base (0,0,0), x (0), y(0), z(0), axis (1,0,0), length (1), color (1,1,1) which is color.white, red (1), green (1), blue (1), and up (0,1,0). Additional pyramid attributes:

height In the y direction in the simple case, default is 1

width In the z direction in the simple case, default is 1

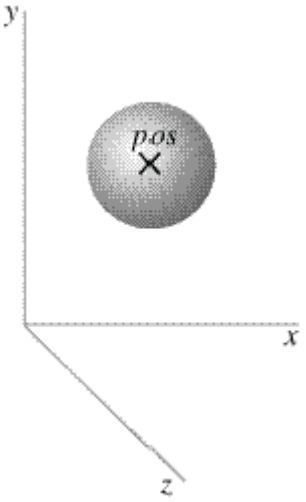
size (length, height, width), default is (1,1,1)
mypyramid.size=(20,10,12) sets length=20, height=10, width=12

Note that the pos attribute for cylinder, arrow, cone, and pyramid corresponds to one end of the object, whereas for a box, sphere, or ring it corresponds to the center of the object.

The sphere Object

Here is an example of how to make a sphere:

```
ball = sphere(pos=(1,2,1), radius=0.5)
```



This produces a sphere centered at location (1,2,1) with radius = 0.5, with the current foreground color.

The sphere object has the following attributes and default values, like those for cylinders except that there is no length attribute: pos (0,0,0), x (0), y(0), z(0), axis (1,0,0), color (1,1,1) which is color.white, red (1), green (1), blue (1), and up (0,1,0). As with cylinders, up has a subtle effect on the 3D appearance of a sphere. The axis attribute only affects the orientation of the sphere and has a subtle effect on appearance; the magnitude of the axis attribute is irrelevant. Additional sphere attributes:

radius Radius of the sphere, default = 1

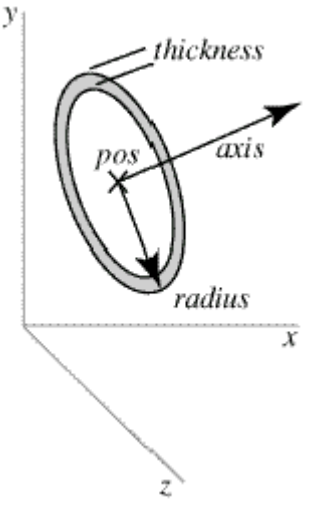
Note that the pos attribute for cylinder, arrow, cone, and pyramid corresponds to one end of the object, whereas for a sphere it corresponds to the center of the object.

Originally there was a label attribute for the sphere object, but this has been superceded by the label object.

The ring Object

The ring object is circular, with a specified outer radius and thickness, and with its center given by the pos attribute:

```
ring(pos=(1,1,1), axis=(0,1,0),
     radius=0.5, thickness=0.1)
```



The ring object has the following attributes and default values, like those for cylinders: pos (0,0,0), x (0), y(0), z(0), axis (1,0,0), length (1), color (1,1,1) which is color.white, red (1), green (1), blue (1), and up (0,1,0). As with cylinders, up has a subtle effect on the 3D appearance of a ring. The axis attribute only affects the orientation of the ring; the magnitude of the axis attribute is irrelevant. Additional ring attributes:

radius Outer radius of the ring, default = 1

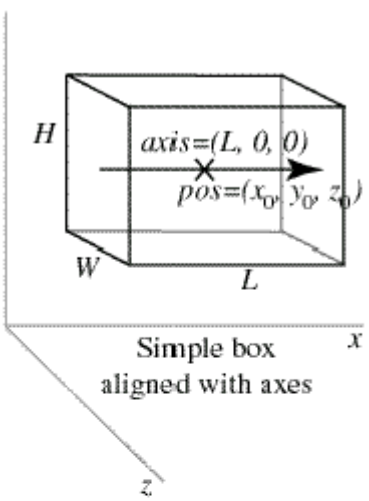
thickness Thickness of ring (1/10th of radius if not specified)

Note that the pos attribute for cylinder, arrow, cone, and pyramid corresponds to one end of the object, whereas for a ring, sphere, and box it corresponds to the center of the object.

The box Object

In the first diagram we show a simple example of a box object:

```
mybox = box(pos=(x0,y0,z0),
      length=L, height=H, width=W)
```

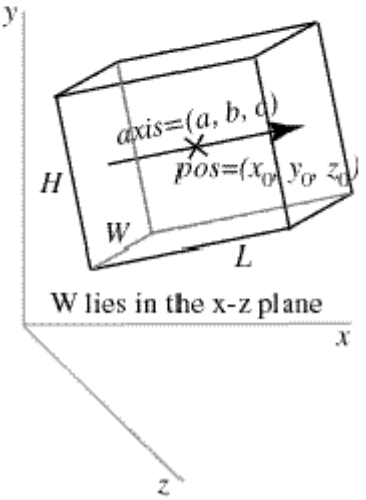


Simple box aligned with axes

The given position is in the center of the box, at (x0, y0, z0). This is different from cylinder, whose pos attribute is at one end of the cylinder. Just as with a cylinder, we can refer to the individual vector components of the box as mybox.x, mybox.y, and mybox.z. The length (along the x axis) is L , the height (along the y axis) is H , and the width is W (along the z axis). For this box, we have mybox.axis = (L, 0, 0) . Note that the axis of a box is just like the axis of a cylinder.

For a box that isn't aligned with the coordinate axes, additional issues come into play. The orientation of the length of the box is given by the axis (see second diagram):

```
mybox = box(pos=(x0,y0,z0),
      axis=(a,b,c), length=L,
      height=H, width=W)
```



W lies in the x-z plane

The axis attribute gives a direction for the length of the box, and the length, height, and width of the box are given as before (if a length attribute is not given, the length is set to the magnitude of the axis vector).

There remains the issue of how to orient the box rotationally around the specified axis. The rule that Visual uses is to orient the width to lie in a plane perpendicular to the display "up" direction, which by default is the y axis. Therefore in the diagram you see that the width lies parallel to the x-z plane. The height of the box is oriented perpendicular to the width, and to the specified axis of the box. It helps to think of length initially as going along the x axis, height along the y axis, and width along the z axis, and when the axis is tipped the width stays in the x-z plane.

You can rotate the box around its own axis by changing which way is "up" for the box, by specifying an up attribute for the box that is different from the up vector of the coordinate system:

```
mybox = box(pos=(x0,y0,z0), axis=(a,b,c), length=L,
      height=H, width=W, up=(q,r,s))
```

With this statement, the width of the box will lie in a plane perpendicular to the (q,r,s) vector, and the height of the box will be perpendicular to the width and to the (a,b,c) vector.

The box object has the following attributes and default values, like those for cylinders: pos (0,0,0), x (0), y(0), z(0), axis (1,0,0), length (1), color (1,1,1) which is color.white, red (1), green (1), blue (1), and up (0,1,0). Additional box attributes:

height In the y direction in the simple case, default is 1

width In the z direction in the simple case, default is 1

size (length, height, width), default is (1,1,1)
mybox.size=(20,10,12) sets length=20, height=10, width=12

Note that the pos attribute for cylinder, arrow, cone, and pyramid corresponds to one end of the object, whereas for a box, sphere, or ring it corresponds to the center of the object.

The ellipsoid Object

A long ellipsoid object looks like a cigar; a short one looks like somewhat like a pill. Its cross sections are circles or ellipses. The elllipsoid object has the same attributes as the box object and it can be thought of as fitting inside a box of the same dimensions:

```
myell = ellipsoid(pos=(x0,y0,z0), length=L, height=H,
      width=W)
```

The given position is in the center of the ellipsoid, at (x0, y0, z0). This is different from cylinder, whose pos attribute is at one end of the cylinder. Just as with a cylinder, we can refer to the individual vector components of the ellipsoid as myell.x, myell.y, and myell.z. The length from end to end (along the x axis) is L , the height (along the y axis) is H , and the width is W (along the z axis). For this ellipsoid, we have myell.axis = (L, 0, 0) . Note that the axis of an ellipsoid is just like the axis of a cylinder.

For an ellipsoid that isn't aligned with the coordinate axes, additional issues come into play. The orientation of the length of the ellipsoid is given by the axis (see diagrams shown with the documentation on the box object):

```
myell = ellipsoid(pos=(x0,y0,z0), axis=(a,b,c),
      length=L,
      height=H, width=W)
```

The axis attribute gives a direction for the length of the ellipsoid, and the length, height, and width of the ellipsoid are given as before (if a length attribute is not given, the length is set to the magnitude of the axis vector).

The ellipsoid object has the following attributes and default values, like those for cylinders: pos (0,0,0), x (0), y(0), z(0), axis (1,0,0), length (1), color (1,1,1) which is color.white, red (1), green (1), blue (1), and up (0,1,0). Additional box attributes:

height In the y direction in the simple case, default is 1

width In the z direction in the simple case, default is 1

size (length, height, width), default is (1,1,1)
myell.size=(20,10,12) sets length=20, height=10, width=12

Note that the pos attribute for cylinder, arrow, cone, and pyramid corresponds to one end of the object, whereas for an ellipsoid, box, sphere, or ring it corresponds to the center of the object.

The curve Object

The curve object displays straight lines between points, and if the points are sufficiently close together you get the appearance of a smooth curve. In addition to its basic use for displaying curves, the curve object has powerful capabilities for other uses, such as efficient plotting of functions.

Some attributes, such as pos and color, can be different for each point in the curve. These attributes are stored as Numeric arrays. The Numeric module for Python provides powerful array processing capabilities; for example, two entire arrays can be added together. Numeric arrays can be accessed using standard Python rules for referring to the nth item in a sequence (that is, seq[0] is the first item in seq, seq[1] is the second, seq[2] is the third, etc). For example, anycurve.pos[0] is the position of the first point in anycurve.

You can give curve an explicit list of coordinates enclosed in brackets, like all Python sequences. Here is an example of a 2D square:

```
square = curve(pos=[(0,0),(0,1),(1,1),(1,0),(0,0)])
```

Essentially, (1,1) is shorthand for (1,1,0). However, you cannot mix 2D and 3D points in one list.

Curves can have thickness, specified by the radius of a cross section of the curve (the curve has a thickness or diameter that is twice this radius):

```
curve(pos=[(0,0,0), (1,0,0), (2,1,0)], radius=0.05)
```

The default radius is 0, which draws a thin curve. A nonzero radius makes a "thick" curve, but a very small radius may make a curve that is too thin to see.

In the following example, the arange() function (provided by the Python Numeric module, which is imported by the Visual module, gives a sequence of values from 0 to 20 in steps of 0.1 (not including the last value, 20).

```
c = curve( x = arange(0,20,0.1) ) # Draw a helix
c.y = sin( 2.0*c.x )
c.z = cos( 2.0*c.x )
```

The x, y, and z attributes allow curves to be used to graph functions easily:

```
curve( x=arange(100), y=arange(100)**0.5,
     color=color.red)
```

A function grapher looks like this (a complete program!):

```
eqn = raw_input('Equation in x: ')
x = arange( 0, 10, 0.1 )
```

```
curve( x=x, y=eval(eqn) )
```

Parametric graphing is also easy:

```
t = arange(0, 10, 0.1)
curve( x = sin(t), y = 1.0/(1+t), z = t**0.5,
     red = cos(t), green = 0, blue = 0.5*(1-cos(t)) )
```

Here are the curve attributes:

pos[] Array of position of points in the curve: pos[0], pos[1], pos[2]....
The current number of points is given by len(curve.pos)

x[ ], y[ ], z[ ] Components of pos; each defaults to [0,0,0,0,...]

color[ ] Color of points in the curve

red[ ], green[ ], blue[ ] Color components of points in the curve

radius Radius of cross-section of curve
The default radius=0 makes a thin curve

Adding more points to a curve

Curves can be created incrementally with the append() function. A new point by default shares the characteristics of the last point.

```
helix = curve( color = color.cyan )
for t in arange(0, 2*pi, 0.1):
    helix.append( pos=(t,sin(t),cos(t)) )
```

One of the many uses of curves is to leave a trail behind a moving object. For example, if ball is a moving sphere, this will add a point to its trail:

```
trail = curve()
ball = sphere()
...# Every time you update the position of the ball:
trail.append(pos=ball.pos)
```

Interpolation

The curve machinery interpolates from one point to the next. For example, suppose the first three points are red but the fourth point is blue, as in the following example. The lines connecting the first three points are all red, but the line going from the third point (red) to the fourth point (blue) is displayed with a blend going from red to blue.

```
c = curve( pos=[(0,0,0), (1,0,0)], color=color.red )
c.append( pos=(1,1,0) ) # add a red point
c.append( pos=(0,1,0), color=color.blue) # add blue
     point
```

If you want an abrupt change in color or thickness, simply add another point at the same location. In the following example, adding a blue point at the same location as the third (red) point makes the final line be purely blue.

```
c = curve( pos=[(0,0,0), (1,0,0)], color=color.red )
c.append( pos=(1,1,0) ) # add a red point
c.append( pos=(1,1,0), color=color.blue) # same point,
     blue
c.append( pos=(0,1,0) ) # add blue point
```

The helix Object

The following statement will display a helix that is parallel to the x axis:

```
spring = helix(pos=(0,2,1), axis=(5,0,0), radius=0.5)
```

The helix object has the following attributes and default values: pos (0,0,0), x (0), y(0), z(0), axis (1,0,0), length (1), radius (0.2), coils (5), thickness (radius/20), color (1,1,1) which is color.white, red (1), green (1), blue (1), and up (0,1,0).

Note that the pos attribute for cylinder, arrow, cone, pyramid, and helix corresponds to one end of the object, whereas for a box, sphere, or ring it corresponds to the center of the object.

The convex Object

The convex object takes a list of points for pos, like the curve object. An object is generated that is everywhere convex (that is, bulges outward). Any points that would make a portion of the object concave (bulge inward) are discarded. If all the points lie in a plane, the object is a flat surface.

The label Object

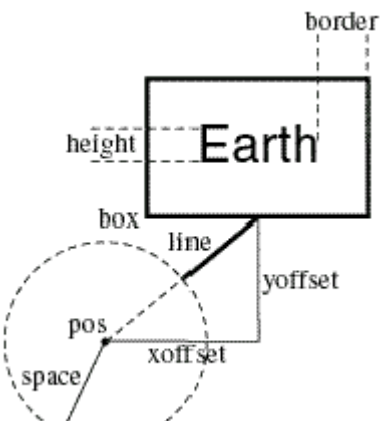
With the label object you can display text in a box. Here are simple examples (in the second label statement, note the standard Python scheme for formatting numerical values, where 1.5f means 1 figure before the decimal point and 5 after):

```
box(pos=(0,0,0), color=color.red)

label(pos=(0,0.25,0), text='This is a box')

label(pos=(0,-0.25,0), text='pi = %1.5f' % pi)
```

There are many additional label options. In the accompanying diagram, a sphere representing the Earth (whose center is at earth.pos) has an associated label carrying the text "Earth" in a box, connected to the sphere by a line which stops at the surface of the sphere:



```
earthlabel = label(pos=earth.pos, text='Earth',
     xoffset=20, yoffset=12, space=earth.radius,
     height=10, border=6)
```

A unique feature of the label object is that several attributes are given in terms of screen pixels instead of the usual "world-space" coordinates. For example, the height of the text is given in pixels, with the result that the text remains readable even when the sphere object is moved far away. Other pixel-oriented attributes include xoffset, yoffset, and border. Here are the label attributes:

pos; x,y,z The point in world space being labeled

xoffset, yoffset The x and y components of the line, in pixels (see diagram)

text The text to be displayed, such as 'Earth'
(Line breaks can be included as \n, as in label.text = "Three\nlines\nof text")

height Height of the font in pixels

color, red, green, blue Color of the text

opacity Opacity of the background of the box, default 0.66
(0 transparent, 1 opaque, for objects behind the box)

border Distance in pixels from the text to the surrounding box

box 1 if the box should be drawn (default), else 0

line 1 if the line from the box to pos should be drawn (default), else 0

linecolor Color of the line and box

space World-space radius of a sphere surrounding pos,
into which the connecting line does not go

font Optional name of the font, such as `helvetica'

Composite Objects with frame

You can group objects together to make a composite object that can be moved and rotated as though it were a single object. Create a frame object, and associate objects with that frame:

```
f = frame()
cylinder(frame=f, pos=(0,0,0), radius=0.1, length=1,
     color=color.cyan)
sphere(frame=f, pos=(1,0,0), radius=0.2,
     color=color.red)
f.axis = (0,1,0)
f.pos = (-1,0,0)
```

By default, frame() has a position of (0,0,0) and axis in the x direction (1,0,0). The cylinder and sphere are created within the frame. When any of the frame attributes are changed (pos, x, y, z, axis, or up), the composite object is reoriented and repositioned.

Another frame attribute is objects, which is a list of currently visible objects contained in the frame (the list does not include objects that are currently invisible). If you want to make all the objects in a frame be red, do the following (assume the frame is named f):

```
for obj in f.objects:
    obj.color = color.red
```

If you use this method to make all the objects invisible, the f.objects list will be empty. If you need a list containing all the objects, both visible and invisible, you need to maintain your own list of objects.

If ball is an object in a frame, ball.pos is the position local to the frame, not the actual position in "world space". Here is a routine that will calculate the position of a vector such as ball.pos in world space:

```
def world_space_pos(frame, local):
    """Returns the position of local in world space."""
    x_axis = norm(frame.axis)
    z_axis = norm(cross(frame.axis, frame.up))
    y_axis = norm(cross(z_axis, x_axis))
    return
    frame.pos+local.x*x_axis+local.y*y_axis+local.z*z_a
    xis
```

The faces Object

The "faces" primitive takes a list of triangles (position, color, and normal for each vertex). This is useful for writing routines in Python to import 3D models made with other 3D modeling tools. You would still need to do lots of calculations of normals and so on, but you would not need to do C coding to import an arbitrary model file.

The faces object is an array primitive (like curve, convex, etc), so you have to use a frame to move it around. It consists of a set of one-sided triangles with user-specified vertices, colors, and normals. The pos, color, and normal attributes look like this:

```
pos = [ t0v0, t0v1, t0v2, t1v0, t1v1, t1v2, t2v0, t2v1,
    t2v2, ... ]
```

where t0v0 is the position of vertex 0 of triangle 0, t0v1 is vertex 1 of triangle 0, etc.

Each face is a one-sided surface. Which side is illuminated is determined by the "winding" order of the face. When you are looking at a face, it is illuminated if the order of the vertices in the pos list goes counter-clockwise. If you need the triangle to be visible from either side, you must create another triangle with the opposite winding order.

If you don't specify normals at the vertices, the face is illuminated only by "ambient" light. In order for the main lighting to affect the appearance, you must specify normals to the surface at the vertices. In the simplest case, a normal at a vertex is perpendicular to the face, and adjoining faces have a hard edge where they join. A soft edge can be

produced by averaging the normals to the two faces at their common vertices. The brightness of a face is proportional to the cosine of the angle between the normal and the light.

If you specify different colors at the vertices of one triangular face, VPython interpolates across the face, in which case the face is not all one color. There is a similar interpolation for normals if there are different normals at the vertices, in which case the face is not all one brightness.

The faces object is intended to help with writing model importers and other new primitives in Python, not for direct manipulation by normal programs. It is considerably lower-level than any of the other objects in Visual (although it is not necessarily any faster, at least right now). It is flexible enough to implement smooth or facet shading, per-vertex coloration, two-sided or one-sided lighting, etc, but all of these calculations must be made by the programmer (when setting up pos, color, normal).

For examples of the use of the faces object, see the faces demo programs.

Additional Attributes

The following attributes apply to all VPython objects:

visible If false (0), object is not displayed; e.g. ball.visible = 0
Use ball.visible = 1 to make the ball visible again.

frame Place this object into a specified frame, as in ball = sphere(frame = f1)

display When you start a VPython program, for convenience Visual creates a display window and names it scene. By default, objects you create go into that display window. You can choose to put an object in a different display like this:

```
scene2 = display( title = "Act IV, Scene 2" )
rod = cylinder( display = scene2 )
```

The function display.get_selected() returns a reference to the display in which objects are currently being created.

__class__ Name of the class of object. For example, ball.__class__ is sphere is true if ball is a sphere object. There are two underscores before and after the word class. In a list of visible objects provided by scene.objects, if obj is in this list you can determine the class of the object with obj.__class__.

__copy()__ Makes a copy of an object. There are two underscores before and after the copy(). Without any arguments, this results in creating a second object in the exact same position as the first, which is probably not what you want. The __copy__() function takes a list of keyword=value argument pairs which are applied to the new object before making it visible. For example, to clone an object from one display to another, you would execute: new_object = old_object.__copy__(display=new_display). Restriction: If the original object is within a frame, and the new object is on a different display, you must supply both a new display and a new frame for the new object (the new frame may be None). This is due to the restriction that an object may not be located

within a frame that is in a separate display. The attribute __members__ used to give a list of all the object's attributes but is no longer available in VPython. Its main use was in copying objects.

Here is an example that uses the __copy()__ function. The following routine copies all of the Visual objects currently existing in one display into a previously defined second display:

```
def clone_universe( new_display, old_display):
    # Create a dictionary of frames in the old display
    # to the corresponding frames in the new display.
    frames = dict()
    # Initialize the lookup dictionary
    for obj in old_display.objects:
        if obj.__class__ == frame:
            frames[obj] = obj.__copy__( frame=None, display=new_display)
    # For each old reference frame that was within another reference frame,
    # place the new reference frame within the appropriate frame in the new
    # display. Here old is an object and new is its frame in the new display.
    for old, new in frames.iteritems():
        if old.frame:
            new.frame = frames[old.frame]
    # Copy over the universe.
    for obj in old_display.objects:
        if obj.__class__ == frame:
            # Already taken care of above.
            continue
        if obj.frame:
            # Then initialize with the corresponding reference frame in the new
            # display.
            obj.__copy__( display=new_display, frame=frames[obj.frame])
        else:
            # Don't need to care about the frame attribute, since it is None.
            obj.__copy__( display=new_display)
```

See Controlling One or More Visual Display Windows for more information on creating and manipulating display objects.

Convenient Defaults

Objects can be specified with convenient defaults:

cylinder() is equivalent to cylinder(pos=(0,0,0), axis=(1,0,0), radius=1)

arrow() is equivalent to arrow(pos=(0,0,0), axis=(1,0,0), radius=1)

cone() is equivalent to cone(pos=(0,0,0), axis=(1,0,0), radius=1)

pyramid() is equivalent to pyramid(pos=(0,0,0), size=(1,1,1), axis=(1,0,0))

sphere() is equivalent to sphere(pos=(0,0,0), radius=1)

ring() is equivalent to ring(pos=(0,0,0), axis=(1,0,0), radius=1)

box() is equivalent to box(pos=(0,0,0), size=(1,1,1))

curve() establishes an "empty" curve to which points can be appended

convex() establishes an "empty" object to which points can be appended

frame() establishes a frame with pos=(0,0,0) and axis=(1,0,0)

Rotating an Object

The cylinder, arrow, cone, pyramid, sphere, ring, box, and ellipsoid objects (but not curve or convex) can be rotated about a specified origin:

```
object.rotate(angle=pi/4., axis=axis, origin=pos)
```

The rotate function applies a transformation to the specified object (sphere, box, etc.). The transformation is a rotation of angle radians, counterclockwise around the line defined by origin and origin+axis. By default, rotations are around the object's own pos and axis.

Specifying Colors

In the RGB color system, you specify a color in terms of fractions of red, green, and blue, corresponding to how strongly glowing are the tiny red, green, and blue dots of the computer screen. In the RGB scheme, white is the color with a maximum of red, blue, and green (1, 1, 1). Black has minimum amounts (0, 0, 0). The brightest red is represented by (1, 0, 0); that is, it has the full amount of red, no green, and no blue.

Here are some examples of RGB colors, with names you can use in Visual:

(1,0,0) color.red	(1,1,0) color.yellow	(0,0,0) color.black
(0,1,0) color.green	(1,0.5,0) color.orange	(1,1,1) color.white
(0,0,1) color.blue	(0,1,1) color.cyan	
	(1,0,1) color.magenta	

You can also create your own colors, such as these:

(0.5, 0.5, 0.5) a rather dark grey

(1,0.7,0.2) a coppery color

Colors may appear differently on different computers, and under different 3D lighting conditions. The named colors above are most likely to display appropriately, because RGB values of 0 or 1 are unaffected by differing color corrections ("gamma" corrections).

The VPython demo program colorsliders.py lets you adjust RGB sliders to visualize colors and print color triples that you copy into your program. It also provides HSV sliders to adjust hue, saturation (how much white is added to dilute the hue), and value (brightness), which is an alternative way to describe colors.

Currently Visual only accepts RGB color descriptions, but there are functions for converting color triples between RGB and HSV:

```
c = (1,1,0)
c2 = color.rgb_to_hsv(c) # convert RGB to HSV
print hsv # (0.16667, 1, 1)
c3 = color.hsv_to_rgb(c2) # convert back to RGB
print c3 # (1, 1, 0)
```

Another example: **sphere(radius=2, color=hsv_to_rgb((0.5,1,0.8))**

Deleting an Object

To delete a Visual object just make it invisible: ball.visible = 0

Technical detail: If you later re-use the name ball, for example by creating a new object and naming it ball, Python will be free to release the memory used by the object formerly named ball (assuming no other names currently refer to that object).

Limiting the Animation Rate

```
rate( frequency )
```

Halts computations until 1.0/frequency seconds after the previous call to rate().

For example, rate(50) will halt computations long enough to make sure that at least 1.0/50.0 second has elapsed (if this much time has already elapsed, no halt is performed). If you place rate(50) inside a computational loop, the loop will execute at a maximum of 50 times per second, even if the computer can run faster than this. This makes animations look about the same on computers of different speeds, as long as the computers are capable of carrying out 50 computations per second.

Floating Division

By default, Python performs integer division with truncation, so that 3/4 is 0, not 0.75. This is inconvenient when doing scientific computations, and can lead to hard-to-find bugs in programs. You can write 3./4., which is 0.75 by the rules of "floating-point" division.

Starting with Python 2.2, you can change the default so that 3/4 is treated as 0.75. Place this at the start of your program:

```
from __future__ import division
```

There are two underscores ("_" and "_") before "future" and two after.

The Visual module converts integers to floating-point numbers for you when specifying attributes of objects:

object.pos = (1,2,3) is equivalent to object.pos = (1.,2.,3.)

A related issue in versions of Python preceding Python 2.2 is that raising an integer to a negative power, as in 10**-2, gives an error. Instead, use 10. (a floating-point number) and write the expression as 10.**-2 in order to obtain the desired result (0.01 in this case). This is not a problem in Python 2.2 and later versions.

The vector Object

The vector object is not a displayable object but is a powerful aid to 3D computations. Its properties are similar to vectors used in science and engineering. It can be used together with Numeric arrays. (Numeric is a module added to Python to provide high-speed computational capability through optimized array processing. The Numeric module is imported automatically by Visual.)

```
vector(x,y,z)
```

Returns a vector object with the given components, which are made to be floating-point (that is, 3 is converted to 3.0).

Vectors can be added or subtracted from each other, or multiplied by an ordinary number. For example,

```
v1 = vector(1,2,3)
v2 = vector(10,20,30)
print v1+v2 # displays (11 22 33)
print 2*v1 # displays (2 4 6)
```

You can refer to individual components of a vector:

v2.x is 10, v2.y is 20, v2.z is 30

It is okay to make a vector from a vector: vector(v2) is still vector(10,20,30).

The form vector(10,12) is shorthand for vector(10,12,0).

A vector is a Python sequence, so v2.x is the same as v2[0], v2.y is the same as v2[1], and v2.z is the same as v2[2].

```
mag( vector ) # calculates length of vector

mag(vector(1,1,1)) # is equal to sqrt(3)

mag2(vector(1,1,1)) # is equal to 3, the magnitude
     squared
```

You can also obtain the magnitude in the form v2.mag. and the square of the magnitude as v2.mag2.

It is possible to reset the magnitude or the magnitude squared of a vector:

```
v2.mag = 5 # sets magnitude of v2 to 5

v2.mag2 = 2.7 # sets squared magnitude of v2 to 2.7
```

You can reset the magnitude to 1 with norm():

```
norm( vector ) # normalized; magnitude of 1

norm(vector(1,1,1)) equals vector(1,1,1)/sqrt(3)
```

You can also write v1.norm(). Since norm(v1) = v1/mag(v1), it is not possible to normalize a zero-length vector: norm(vector(0,0,0)) gives an error, since division by zero is involved.

```
vector1.diff_angle(vector2)
```

Calculates the angle between two vectors (the "difference" of the angles of the two vectors)..

```
cross( vector1, vector2 )
```

Creates the cross product of two vectors, which is a vector perpendicular to the plane defined by vector1 and vector2, in a direction defined by the right-hand rule: if the fingers of the right hand bend from vector1 toward vector 2, the thumb points in the direction of the cross product. The magnitude of this vector is equal to the product of the magnitudes of vector1 and vector2, times the sine of the angle between the two vectors.

```
dot( vector1, vector2 )
```

Creates the dot product of two vectors, which is an ordinary number equal to the product of the magnitudes of vector1 and vector2, times the cosine of the angle between the two vectors. If the two vectors are normalized, the dot product gives the cosine of the angle between the vectors, which is often useful.

Rotating a vector

```
v2 = rotate(v1, angle=theta, axis=(1,1,1))
```

The default axis is (0,0,1), for a rotation in the xy plane around the z axis. There is no origin for rotating a vector. Notice too that rotating a vector involves a function, v = rotate(), as is the case with other vector manipulations such as dot() or cross(), whereas rotation of graphics objects involves attributes, in the form object.rotate().

Convenient conversion

For convenience Visual automatically converts (a,b,c) into vector(a,b,c), with floating-point values, when creating Visual objects: sphere.pos=(1,2,3) is equivalent to sphere.pos=vector(1.,2.,3.). However, using the form (a,b,c) directly in vector computations will give errors, because (a,b,c) isn't a vector; write vector(a,b,c) instead.

You can convert a vector vec1 to a Python tuple (a,b,c) by tuple(vec1) or by the much faster option vec1.as_tuple().

Graph Plotting

In this section we describe features for plotting graphs with tick marks and labels. Here is a simple example of how to plot a graph:

```
from visual.graph import * # import graphing features


funct1 = gcurve(color=color.cyan) # a connected curve
     object


for x in arange(0., 8.1, 0.1): # x goes from 0 to 8
    funct1.plot(pos=(x,5.*cos(2.*x)*exp(-0.2*x))) # plot
```

Importing from visual.graph makes available all Visual objects plus the graph plotting module. The graph is autoscaled to display all the data in the window.

A connected curve (gcurve) is just one of several kinds of graph plotting objects. Other options are disconnected dots (gdots), vertical bars (gvbars), horizontal bars (ghbars), and binned data displayed as vertical bars (ghistogram; see later discussion). When creating one of these objects, you can specify a color attribute. For gvbars and ghbars you can also specify a delta attribute, which specifies the width of the bar (the default is delta=1.).

You can plot more than one thing on the same graph:

```
funct1 = gcurve(color=color.cyan)
funct2 = gvbars(delta=0.05, color=color.blue)
for x in arange(0., 8.1, 0.1):
    funct1.plot(pos=(x,5.*cos(2.*x)*exp(-0.2*x))) #
     curve
    funct2.plot(pos=(x,4.*cos(0.5*x)*exp(-0.1*x))) #
     vbars
```

In a plot operation you can specify a different color to override the original setting:

```
mydots.plot(pos=(x1,y1), color=color.green)
```

When you create a gcurve, gdots, gvbars, or ghbars object, you can provide a list of points to be plotted, just as is the case with the ordinary curve object:

```
points = [(1,2), (3,4), (-5,2), (-5,-3)]
data = gdots(pos=points, color=color.blue)
```

This list option is available only when creating the gdots object.

Overall gdisplay options

You can establish a gdisplay to set the size, position, and title for the title bar of the graph window, specify titles for the x and y axes, and specify maximum values for each axis, before creating gcurve or other kind of graph plotting object:

```
graph1 = gdisplay(x=0, y=0, width=600, height=150,
          title='N vs. t', xtitle='t', ytitle='N',
          xmax=50., xmin=-20., ymax=5E3, ymin=-2E3,
          foreground=color.black,
      background=color.white)
```

In this example, the graph window will be located at (0,0), with a size of 600 by 150 pixels, and the title bar will say 'N vs. t'. The graph will have a title 't' on the horizontal axis and 'N' on the vertical axis. Instead of autoscaling the graph to display all the data, the graph will have fixed limits. The horizontal axis will extend from -20. to +50., and the vertical axis will extend from -2000. to +5000. (xmin and ymin must be negative; xmax and ymax must be positive.) The foreground color (white by default) is black, and the background color (black by default) is white. If you simply say gdisplay(), the defaults are x=0, y=0, width=800, height=400, no titles, fully autoscaled.

Every gdisplay has the attribute display, so you can manipulate basic display aspects of the graphing window:

```
graph1.display.visible = 0 # make the display invisible
```

You can have more than one graph window: just create another gdisplay. By default, any graphing objects created following a gdisplay belong to that window. You can also specify which window a new object belongs to:

```
energy = gdots(gdisplay=graph1, color=color.blue)
```

Histograms (sorted, binned data)

The purpose of ghistogram is to sort data into bins and display the distribution. Suppose you have a list of the ages of a group of people, such as [5, 37, 12, 21, 8, 63, 52, 75, 7]. You want to sort these data into bins 20 years wide and display the numbers in each bin in the form of vertical bars. The first bin (0 to 20) contains 4 people [5, 12, 8, 7], the second bin (20 to 40) contains 2 people [21, 37], the third bin (40 to 60) contains 1 person [52], and the fourth bin (60-80) contains 2 people [63, 75]. Here is how you could make this display:

```
from visual.graph import *
.....
```

```
agelist1 = [5, 37, 12, 21, 8, 63, 52, 75, 7]
ages = ghistogram(bins=arange(0, 80, 20),
     color=color.red)
ages.plot(data=agelist1) # plot the age distribution
.....
ages.plot(data=agelist2) # plot a different distribution
```

You specify a list (bins) into which data will be sorted. In the example given here, bins goes from 0 to 80 by 20's. By default, if you later say

```
ages.plot(data=agelist2)
```

the new distribution replaces the old one. If on the other hand you say

```
ages.plot(data=agelist2, accumulate=1)
```

the new data are added to the old data.

If you say the following,

```
ghistogram(bins=arange(0,50,0.1), accumulate=1,
     average=1)
```

each plot operation will accumulate the data and average the accumulated data. The default is no accumulation and no averaging.

gdisplay vs. display

A gdisplay window is closely related to a display window. The main difference is that a gdisplay is essentially two-dimensional and has nonuniform x and y scale factors. When you create a gdisplay (either explicitly, or implicitly with the first gcurve or other graphing object), the current display is saved and restored, so that later creation of ordinary Visual objects such as sphere or box will correctly be associated with a previous display, not the more recent gdisplay.

Controlling One or More Visual Display Windows

Initially, there is one Visual display window named scene. Display objects do not create windows on the screen unless they are used, so if you immediately create your own display object early in your program you will not need to worry about scene. If you simply begin creating objects such as sphere they will go into scene.

display() Creates a display with the specified attributes, makes it the selected display, and returns it. For example, the following creates another Visual display window 600 by 200, with 'Graph of position' in the title bar, centered on (5,0,0) and with a background color of cyan filling the window.

```
scene2 = display(title='Graph of position',
     width=600, height=200,
     center=(5,0,0), background=(0,1,1))
```

General-purpose options

select() Makes the specified display the "selected display", so that objects will be drawn into this display by default; e.g. scene.select()

The function display._selected() returns a reference to the display in which objects are currently being created (which will be None if no display has been created yet).

foreground Set color to be used by default in creating new objects such as sphere; default is white. Example: scene.foreground = (1,0,0)

background Set color to be used to fill the display window; default is black.

stereo Stereoscopic option; scene.stereo = 'redcyan' will generate a scene for the left eye and a scene for the right eye, to be viewed with red-cyan glasses, with the red lens over the left eye. (There are also 'redblue' and 'yellowblue' options; note that objects that were not originally white may be somewhat dim.)

Setting scene.stereo = 'crosseyed' produces side-by-side images which if small enough can be seen in 3D by crossing your eyes but focusing on the screen (this takes some practice). Setting scene.stereo = 'passive' produces side-by-side images which if small enough can be seen in 3D by looking "wall-eyed", looking into the far distance but focusing on the screen (this too takes some practice).

scene.stereo = 'active' will render alternating left eye/right eye images for viewing through shutter glasses if the graphics system supports quad buffered stereo. If stereo equipment is not available, setting the option has no effect, and scene.stereo will have the value 'nostereo'. You can also use scene.stereo = 'passive' with quad buffered stereo for display using two polarized projectors (for stereo viewing using simple passive polarized glasses). (Quad buffered 'active' stereo is only available on specialised graphics systems that have the necessary hardware and shutter glass connector, such as PCs with nVidia Quadro graphics cards. It generates the illusion of depth by rendering each frame twice from slightly different viewpoints corresponding to the left and right eyes. Special shutter glasses are synchronised with the alternating images so that each eye sees only the matching frame, and your brain does the rest. It's called 'quad buffered' because there is an OpenGL buffer per eye, both double-buffered for smooth updating. 'Passive' stereo requires a video card that can drive two monitors, or two projectors.)

stereodepth By default, the front of the scene is located at the location of the physical screen, which reduces eye strain. Setting scene.stereodepth = 1 moves the center of the scene to the location of the physical screen, with the front half of the scene seeming to stick dramatically out of the screen. scene.stereodepth = 2 moves the scene fully in front of the physical screen, for maximally dramatic stereo effect.

ambient Amount of nondirectional ("ambient") lighting. Default is 0.2. Also see the following lights attribute.

lights List of vectors representing directions from the origin to the lights. The magnitude of the vector is the intensity. For example, scene.lights =

[vector(1,0,0)] with scene.ambient = 0 will light the scene from the right side, with no ambient lighting on the left. By default there are two lights in the list: (0.17, 0.35, 0.70), magnitude 0.8, and (-0.26, -0.07, -0.13), magnitude 0.3. The attributes lights and ambient must be used with some care, because if the total lighting intensity exceeds 1 anywhere in the scene the results are unpredictable.

cursor.visible By setting scene.cursor.visible = 0, the mouse cursor becomes invisible. This is often appropriate while dragging an object using the mouse. Restore the cursor with scene.cursor.visible = 1.

objects A list of all the visible objects in the display; invisible objects are not listed. For example, this makes all boxes in the scene red:

```
for obj in scene2.objects:
    if obj.__class__ == box # can say either box or
 'box'
        obj.color = color.red
```

To obtain camera position, see Mouse Interactions.

Controlling the window

x, y Position of the window on the screen (pixels from upper left)

width, height Width and height of the display area in pixels: scene.height = 200

title Text in the window's title bar: scene.title = 'Planetary Orbit'

visible Make sure the display is visible; scene2.visible = 1 makes the display named scene2 visible. This is automatically called when new primitives are added to the display, or the mouse is referenced. Setting visible to 0 hides the display.

fullscreen Full screen option; scene2.fullscreen = 1 makes the display named scene2 take up the entire screen. In this case there is no close box visible; press Escape to exit.
There is currently a bug in the fullscreen option for Linux/Unix/Mac OSX. Keyset input is not recognized, including the Escape key. If you use the fullscreen option on these systems, be sure to program a mouse input for quitting the program.

exit If sceneb.exit = 0, the program does not quit when the close box of the sceneb display is clicked. The default is sceneb.exit = 1, in which case clicking the close box does make the program quit.

Controlling the view

center Location at which the camera continually looks, even as the user rotates the position of the camera. If you change center, the camera moves to continue to look in the same "compass" direction toward the new center, unless you also change forward (see next attribute). Default (0,0,0).

autocenter scene.center is continuously updated to be the center of the smallest axis-aligned box containing the scene. This means that if your program moves the entire scene, the center of that scene will continue to be centered in the window.

forward Vector pointing in the same direction as the camera looks (that is, from the current camera location, given by scene.mouse.camera, toward scene.center). The user rotation controls, when active, will change this vector continuously. When forward is changed, the camera position changes to continue looking at center. Default (0,0,-1).

fov Field of view of the camera in radians. This is defined as the maximum of the horizontal and vertical fields of view. You can think of it as the angular size of an object of size range, or as the angular size of the longer axis of the window as seen by the user. Default pi/3.0 radians (60 degrees).

range The extent of the region of interest away from center along each axis. This is always 1.0/scale, so use either range or scale depending on which makes the most sense in your program. Default (10,10,10) or set by autoscale.

scale A scaling factor which scales the region of interest into the sphere with unit radius. This is always 1.0/range, so use either range or scale depending on which makes the most sense in your program. Default (0.1,0.1,0.1) or set by autoscale.

uniform = 0 each axis has different units and scales

autoscale will scale axes independently

the x and y axes will be scaled by the aspect ratio of the window

uniform = 1 each axis has the same scale

autoscale scales axes together

the aspect ratio of the window does not affect scaling

up A vector representing world-space up. This vector will always project to a vertical line on the screen (think of the camera as having a "plumb bob" that keeps the top of the screen oriented toward up). The camera also rotates around this axis when the user rotates "horizontally". By default the y axis is the up vector.

There is an interaction between up and forward, the direction that the camera is pointing. By default, the camera points in the -z direction (0,0,-1). In this case, you can make the x or y axes (or anything between) be the up vector, but you cannot make the z axis be the up vector, because this is the axis about which the camera rotates when you set the up attribute. If you want the z axis to point up, first set forward to something other than the -z axis, for example (1,0,0).

autoscale = 0 no automatic scaling (set range or scale explicitly)

autoscale = 1 automatic scaling (default)

It is often useful to let Visual make an initial display with autoscaling, then turn autoscaling off to prevent further automated changes.

userzoom = 0 user cannot zoom in and out of the scene

userzoom = 1 user can zoom (default)

userspin = 0 user cannot rotate the scene

userspin = 1 user can rotate (default)

Mouse Interactions

Introduction

Mouse objects are obtained from the mouse attribute of a display object such as scene. For example, to obtain mouse input from the default window created by Visual, refer to scene.mouse. For basic examples of mouse handling, see Click example and Drag example.

A mouse object has a group of attributes corresponding to the current state of the mouse. It also has functions getevent() and getclick(), which return an object with similar attributes corresponding to the state of the mouse when the user last did something with the mouse buttons. If the user has not already done something with the mouse buttons, getevent() and getclick() will stop program execution until this happens.

Different kinds of mouse

The mouse routines can handle a three-button mouse, with "left", "right", and "middle" buttons. For systems with a two-button mouse, the "middle" button consists of the left and right buttons pressed together. For systems with a one button mouse, the right button is invoked by holding down the SHIFT key, and the middle button is invoked by holding down the CTRL key.

Current state of mouse

pos The current 3D position of the mouse cursor; scene.mouse.pos. Visual always chooses a point in the plane parallel to the screen and passing through display.center. (See Projecting mouse information onto a given plane for other options.)

button = None (no buttons pressed), 'left', 'right', 'middle', or 'wheel' (scroll wheel pressed on some Windows mouses). Example: scene.mouse.button == 'left' is true if the left button is currently down.

pick The nearest object in the scene which falls under the cursor, or None. At present only spheres, boxes, cylinders, and convex can be picked. The picked object is scene.mouse.pick.

pickpos The 3D point on the surface of the picked object which falls under the cursor, or None; scene.mouse.pickpos.

camera The read-only current position of the camera as positioned by the user, scene.mouse.camera. For example, mag(scene.mouse.camera-scene.center) is the distance from the center of the scene to the current position of the camera. If you want to set the camera position and direction by program, use scene.forward and scene.center, described in Controlling Windows.

ray A unit vector pointing from camera in the direction of the mouse cursor. The points under the mouse cursor are exactly { camera + t*ray for t>0}.

The camera and ray attributes together define all of the 3D points under the mouse cursor.

project() Projects position onto a plane. See Projecting mouse position onto a given plane.

alt = 1 if the ALT key is down, otherwise 0

ctrl = 1 if the CTRL key is down, otherwise 0 (for a one-button mouse, meaningful only if mouse buttons up)

shift = 1 if the SHIFT key is down, otherwise 0 (for a one-button mouse, meaningful only if mouse buttons up)

Note that programs that depend on modifying a mouse event with CTRL or SHIFT will not work properly with a one-button mouse (e.g. most Macintosh systems), since CTRL invokes "middle" button, and SHIFT invokes "right" button.

Getting events

There are four kinds of mouse events: press, click, drag, and drop:

A press event occurs when a mouse button is depressed.

A click event occurs when all mouse buttons are released with no or very slight movement of the mouse.
Note that a click event happens when the mouse button is *released*. See Click example.

A drag event occurs when the mouse is moved slightly after a press event, with mouse buttons still down.
This can be used to signal the beginning of dragging an object. See Drag example.

A drop event occurs when the mouse buttons are released after a drag event.

Between a drag event (start of dragging) and a drop event (end of dragging), there are no mouse events but you can examine the continuously updated position of the mouse indicated by scene.mouse.pos. Here is how to tell that an event has happened, and to get information about that event:

events The number of events (press, click, drag, or drop) which have been queued; e.g. scene.mouse.events.
scene.mouse.events = 0 may be used to discard all input. No value other than zero can be assigned.

getevent() Obtains the earliest mouse event and removes it from the input queue. If no events are waiting in the queue (that is, if scene.mouse.events is zero), getevent() waits until the user enters a mouse event (press, click, drag, or drop). getevent() returns an object with attributes similar to a mouse object: pos, button, pick, pickpos, camera, ray, project(), alt, ctrl, and shift. These attributes correspond to the state of the mouse when the event took place. For example, after executing mm = scene.mouse.getevent() you can look at the various properties of this event, such as mm.pos, mm.pick, mm.drag (see below), etc.

If you are interested in every type of event (press, click, drag, and drop), you must use events and getevent(). If you are only interested in left click events (left button down and up without significant mouse movement), you can use clicked and getclick():

clicked The number of left clicks which have been queued; e.g. scene.mouse.clicked.
This does not include a count of nonclick events (press, drag, or drop).

getclick() Obtains the earliest mouse left click event (pressing the left button and releasing it in nearly the same position) and removes it from the input queue, discarding any earlier press, drag, or drop events. If no clicks are waiting in the queue (that is, if scene.mouse.clicked is zero), getclick() waits until the user clicks. Otherwise getclick() is just like getevent().

It is a useful debugging technique to insert scene.mouse.getclick() into your program at a point where you would like to stop temporarily to examine the scene. Then just click to proceed.

Additional information obtained with getevent() or getclick()

In addition to the information available with scene.mouse, getevent() and getclick() furnish this additional information:

press = 'left' or 'right' or 'middle' for a press event, or None

click = 'left' or 'right' or 'middle' for a click event, or None; in this case pos and other attributes correspond to the state of the mouse at the time of the original press event, so as not to lose initial position information. See Click example.

drag = 'left' or 'right' or 'middle' for a drag event, or None; in this case pos and other attributes correspond to the state of the mouse at the time of the original press event, so as not to lose initial position information. See Drag example.

drop = 'left' or 'right' or 'middle' for a drop event, or None

release = 'left' or 'right' or 'middle' following click and drop events, indicating which button was released, or None

Normally, dragging with right or middle button represents spin or zoom, and is handled automatically by Visual, so you can check for left-button drag or drop events simply by checking whether drag or drop is true (in Python, a nonempty string such as 'left' is true, None is false). Unless you disable user zoom (scene.userzoom = 0), click, drag, and drop with the middle button are invisible to your program. Unless you disable user spin (scene.userspin = 0), click, drag, and drop with the right button are invisible to your program.

Projecting mouse position onto a given plane

Here is a way to get the mouse position relative to a particular plane in space:

```
temp = scene.mouse.project(normal=(0,1,0),
     point=(0,3,0))
if temp: # temp is None if no intersection with plane
    ball.pos = temp
```

This projects the mouse cursor onto a plane that is perpendicular to the specified normal. If point is not specified, the plane passes through the origin. It returns a 3D position, or None if the projection of the mouse misses the plane.

In the example shown above, the user of your program will be able to use the mouse to place balls in a plane parallel to the xy plane, a height of 3 above the xy plane, no matter how the user has rotated the point of view.

You can instead specify a perpendicular distance d from the origin to the plane that is perpendicular to the specified normal. The example above is equivalent to

```
temp = scene.mouse.project(normal=(0,1,0), d=3)
```

Keyboard Interactions

If scene.kb.keys is nonzero, one or more keyboard events have been stored, waiting to be processed.

Executing key = scene.kb.getkey() obtains a keyboard input and removes it from the input queue. If there are no events waiting to be processed, getkey() waits until a key is pressed.

If len(key) == 1, the input is a single printable character such as 'b' or 'B' or new line ('\n') or tab ('\t'). Otherwise key is a multicharacter string such as 'escape' or 'backspace' or 'f3'. For such inputs, the ctrl, alt, and shift keys are prepended to the key name. For example, if you hold down the shift key and press F3, key will be the character string 'shift+f3', which you can test for explicitly. If you hold down all three modifier keys, you get 'ctrl+alt+shift+f3'; the order is always ctrl, alt, shift.

Here is a test routine that let's you type text into a label:

```
prose = label() # initially blank text
while 1:
    if scene.kb.keys: # is there an event waiting to be
     processed?
        s = scene.kb.getkey() # obtain keyboard
     information
        if len(s) == 1:
            prose.text += s # append new character
        elif (s == 'backspace' or s == 'delete') and
     len(prose.text) > 0:
            prose.text = prose.text[:-1] # erase one
     letter
        elif s == 'shift+delete':
            prose.text = '' # erase all the text
```

Note that mouse events also provide information about the ctrl, alt, and shift keys, which may be used to modify mouse actions.

You can also input a line of text from the keyboard by highlighting the output window (where print statements are displayed), using the standard Python function raw_input(). The statement text = raw_input() accepts a line of typing ending with Enter and sets text to the input, not including the end-of-line. The statement text = raw_input('Type something: ') prompts for the input before you type.

Controls: buttons, sliders, toggles, and menus

You can create buttons, sliders, toggle switches, and pull-down menus to control your program. You import these capabilities with this statement:

```
from visual.controls import *
```

Importing from visual.controls makes available all Visual objects plus the controls module. To use the control features, you create a special controls window and add control objects to that window, specifying what actions should take place when the controls are manipulated. For example, an action associated with a button might be the execution of a function to change the color of a Visual object. After creating the controls, you repeatedly call an interact routine to check for user manipulation of the controls, which trigger actions. For a detailed example, see the VPython demo program controlstest.py.

Here is a small example. All it does is change the button text when you click the button. The Python construction "lambda:" is required for the controls module to have the correct context ("namespace") for calling the specified routine.

```
from visual.controls import *

def change(): # Called by controls when button is
     clicked
    if b.text == 'Click me':
        b.text = 'Try again'
    else:
        b.text = 'Click me'

c = controls() # Create controls window
# Create a button in the controls window:
b = button( pos=(0,0), width=60, height=60,
              text='Click me', action=lambda: change() )
while 1:
    c.interact() # Check for mouse events and drive
     specified actions
```

Controls window

controls() Creates a controls window with the specified attributes, and returns it. For example, the following creates a controls window 300 by 300, located at (0,400) with respect to the upper left corner of the screen, with 'Controlling the Scene' in the title bar, and a range of 50 (window coordinates from -50 to +50 in x and y):

```
c = controls(title='Controlling the Scene',
     x=0, y=400, width=300, height=300, range=50)
```

Controls window parameters

x, y Position of the window on the screen (pixels from upper left)

width, height Width and height of the display area in pixels.

title Text in the control window's title bar.

range The extent of the region of interest away from the center along each axis. The default is 100. The center of a controls window is always (0,0).

Control objects

After creating a controls window, you can create the following control objects that will appear in that window:

button A button to click.

slider Drag a slider to enter a numeric value graphically.

toggle Click on the handle to flip a toggle switch.

menu A pull-down menu of options.

Control objects have the following attributes:

pos Position of the control (center of button or toggle, one end of slider, upper left corner of menu title)

color Gray by default

width Width of button, toggle, or menu

height Height of button, toggle, or menu

axis Axis for slider, pointing from pos to other end (as for cylinder or arrow)

length Length of slider (in direction of axis)

min, max Minimum and maximum values for a slider

value Value of toggle (0 or 1) or slider (depends on slider min and max). The value of a toggle or slider can be set as well as read. If you set the value of a toggle or slider, the control moves to the position that corresponds to that value.

text Text to display on a button, or menu title

text0 Text to display below a toggle switch (associated with toggle value = 0)

text1 Text to display above a toggle switch (associated with toggle value = 1)

action Specify Python statement to be executed when a control is manipulated

items For menus only, list of menu items to choose from. Here is how to add a menu item to a menu named m1:

```
m1.items.append( ('Red', lambda:
cubecolor(color.red)) )
```

This adds to the pull-down menu an item 'Red' which when chosen will pass the value color.red to the subroutine cubecolor(). The Python construction "lambda:" is required for the controls module to have the correct context ("namespace") for calling the specified routine.

The factorial and combin Functions

```
from visual import *

from visual.factorial import *

factorial(N) = N!

combin(a,b) = a!/(b!*(a-b)!)
```

*Note: To avoid confusion between the module named "factorial" and the function named "factorial", import the factorial module **after** importing the visual module itself.*

A major use of these functions is in calculating the number of ways of arranging a group of objects. For example, if there are 5 numbered balls in a sack, there are factorial(5) = 5! = 5*4*3*2*1 = 120 ways of taking them sequentially out of the sack (5 possibilities for the first ball, 4 for the next, and so on).

If on the other hand the 5 balls are not numbered, but 2 are green and 3 are red, of the 120 ways of picking the balls there are 2! indistinguishable ways of arranging the green balls and 3! ways of arranging the red balls, so the number of different arrangements of the balls is combin(5,2) = 5!/(3!*2!) = 10.

Logically, the combin function is just a combination of factorial functions. However, cancellations in the numerator and denominator make it possible to evaluate the combin function for values of its arguments that would overflow the factorial function, due to the limited size of floating-point numbers. For example, combin(5,2) = 5!/(3!*2!) = (5*4)/2 = 10, and we didn't have to evaluate 5! fully.