

The Visual Module of VPython - Reference Manual

par

Date de publication : 4 février 2009

Dernière mise à jour :

VPython is the Python programming language plus a 3D graphics module called "Visual" originated by David Scherer in 2000. This documentation describes all of the Visual capabilities. This translation is consistent with the version 5.0.3 of VPython.

Simple 3D Programming Using VPython.....	3
I - VPython: the Python/ Visual / IDLE environment.....	3
II - Visual Entities.....	3
III - Simple Python Programming.....	4
3D objects.....	8
arrow.....	8
box.....	9
cone.....	10
convex.....	11
curve.....	11
cylinder.....	14
ellipsoid.....	15
faces.....	16
frame.....	17
helix.....	17
label.....	18
points.....	19
pyramid.....	20
ring.....	20
sphere.....	21
Working with objects.....	22
Color and Opacity.....	22
Lighting.....	22
Materials and Textures.....	25
Defaults.....	28
Animation Speed.....	28
Rotations.....	28
Additional Options.....	29
Delete an Object.....	30
3/4 = 0 ?.....	30
Windows/Events/Files.....	30
Windows.....	30
Mouse Events.....	32
Mouse Click.....	34
Mouse Drag.....	35
Keyboard Events.....	36
Button and Sliders.....	37
Reading/Writing Files.....	38
Vector operations.....	39
Graphs.....	41
factorial/combin.....	43
Visual license.....	43

Simple 3D Programming Using VPython

I - VPython: the Python/ Visual / IDLE environment

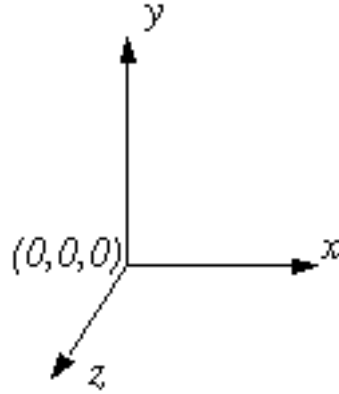
The interactive development environment you will use is called "IDLE."

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

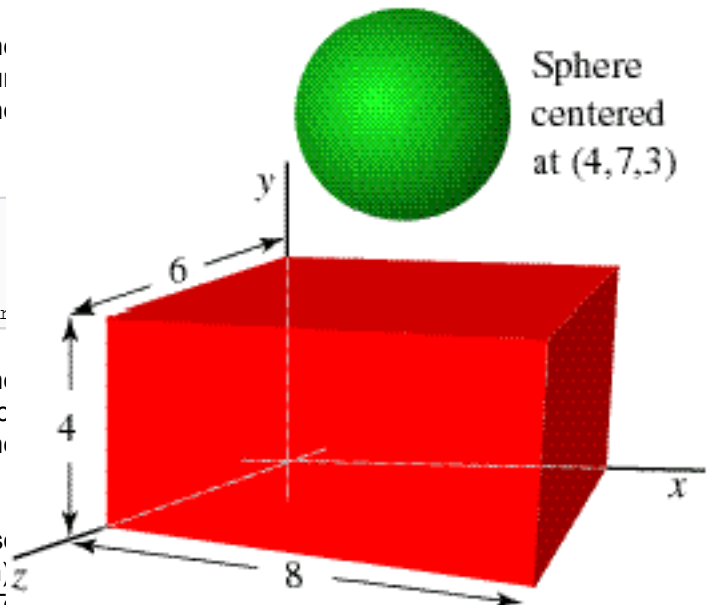
If you type or copy the following simple program into the code window in IDLE and run it (press F5, or use the Run menu), you will see a display like the one shown in the figure.

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

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 command key on a Macintosh), Drag left or right, and you rotate around the scene. To rotate around a horizontal axis, drag up or down. 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 Option 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 **ball** in the example above) if you wish to refer to it again later in your program. All objects have attributes: properties like **ball.pos** (the position of the sphere), **ball.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:

```
ball.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

```
s = mag(c) # s is a scalar
z = mag(c)**2 # you can square the magnitude of a vector
```

Vector products

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

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) # an 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 velocity. Other objects, such as `vector()`, have built-in attributes but you cannot create additional attributes.

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 = 3/4
print a # a is 0
```

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

```
b = 3./4.
print b # b is 0.75, as expected
```

We recommend putting the following statement as the first line of your program, in which case `3/4` will be `0.75`; there are two underscores before the word "future" and two after the word "future":

```
from __future__ import division
```

Exponentiation

```
x**2 # Not x^2, which is a bit operation in Python
```

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
<	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 an array of numbers:

```
angles = arange(0., 2*pi, pi/100)
# 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
```

You can also use the Python symbols True or False:

```
while True:
    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" option:

```
print Europa.momentum
```


To print a text message, enclose it in quotes:

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

Python also offers a formatted print capability. Here price will be printed with 3 digits before the decimal place and 2 digits after, and num will be printed as an integer:

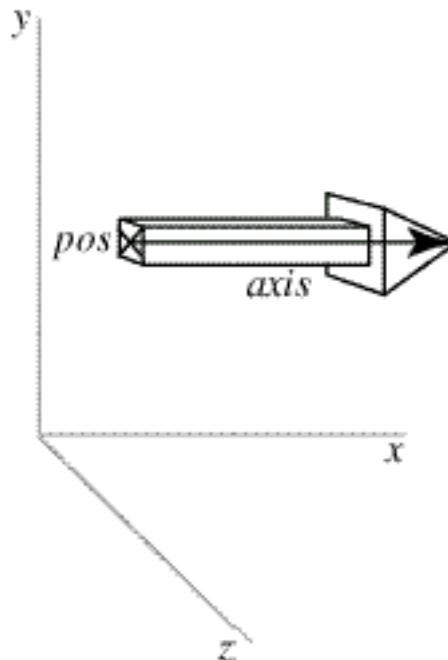
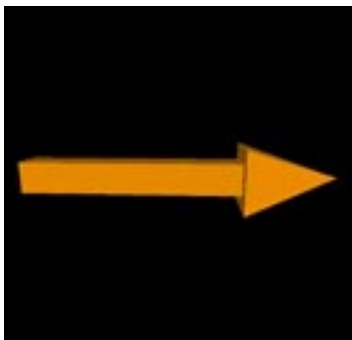
```
print "It's $%3.2f dollars for %d copies" % (price,num)
```

More Information about Python

We have summarized a small but important subset of the Python programming language. Extensive Python documentation is provided on the Help menu in IDLE, 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, and about programming in general: *Python Programming: An Introduction to Computer Science* by John M. Zelle (Franklin Beedle & Associates, 2003).

3D objects

arrow



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), **opacity** (1), **material**, and **up** (0,1,0). As with **box**, 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

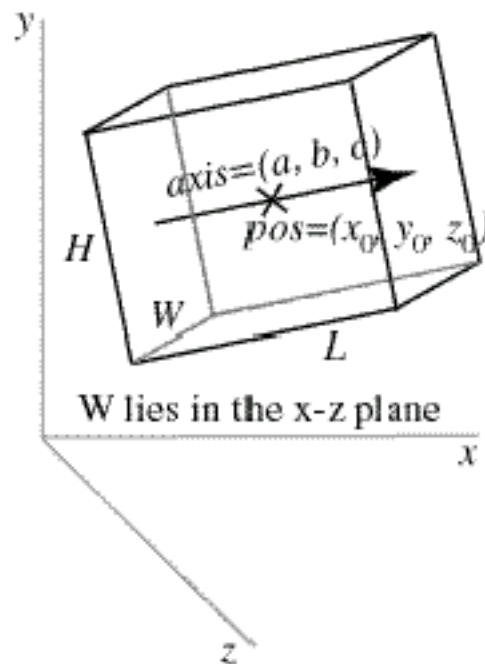
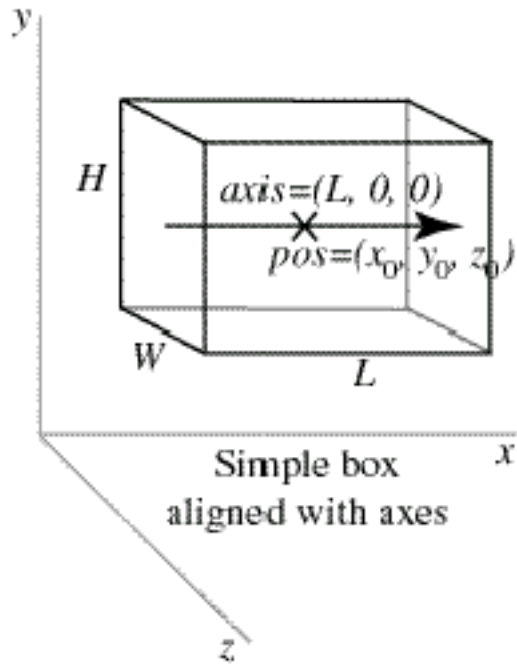
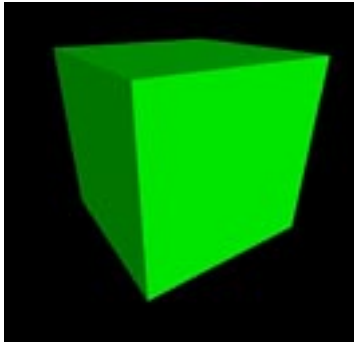
Assigning any of these attributes to 0 makes it use defaults based on the size of the arrow. If **headlength** becomes larger than half the length of the arrow, or the shaft becomes thinner than 1/50 the length, the entire arrow is scaled accordingly.

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.

See description of **Additional Attributes** available for all 3D display objects.

box



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)
```

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)
```

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), **opacity** (1), **material**, 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

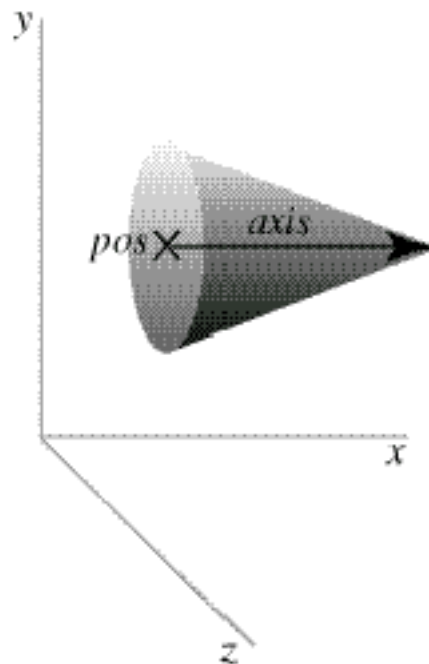
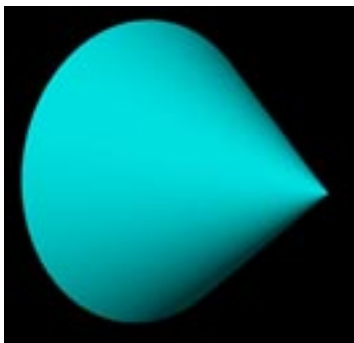
size (length, height, width), default is (1,1,1)

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.

See description of **Additional Attributes** available for all 3D display objects.

cone



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)
```

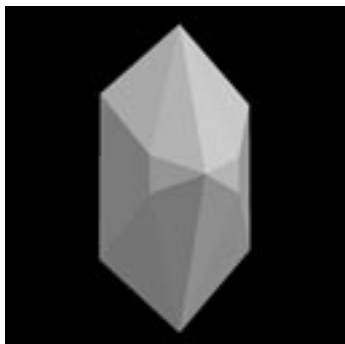
The cone object has the following attributes and default values, like those for cylinders: **pos** (0,0,0), **x** (0), **y**(0), **z**(0), **axis** (1,0,0), **length** (1), **color** (1,1,1) which is color.white, **red** (1), **green** (1), **blue** (1), **opacity** (1), **material**, and **up** (0,1,0).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

See description of **Additional Attributes** available for all 3D display objects.

convex



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.

Currently it is not possible to specify the opacity of a convex object.

See description of **Additional Attributes** available for all 3D display objects.

curve



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 numpy arrays. The numpy module for Python provides powerful array processing capabilities; for example, two entire arrays can be added together. Numpy 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 numpy 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!), where "raw_input" is a Python function that accepts input typed in the Python Shell window:

```
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

material Material for a thick curve; see **Materials** for currently available options

Currently it is not possible to specify the opacity of a curve object.

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.

```
spiral = curve( color = color.cyan )
for t in arange(0, 2*pi, 0.1):
```

```
spiral.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)
```

When appending to a curve, you can optionally choose to retain only the last N points, including the one you're adding:

```
trail.append(pos=ball.pos, retain=50) # last 50 points
```

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
```

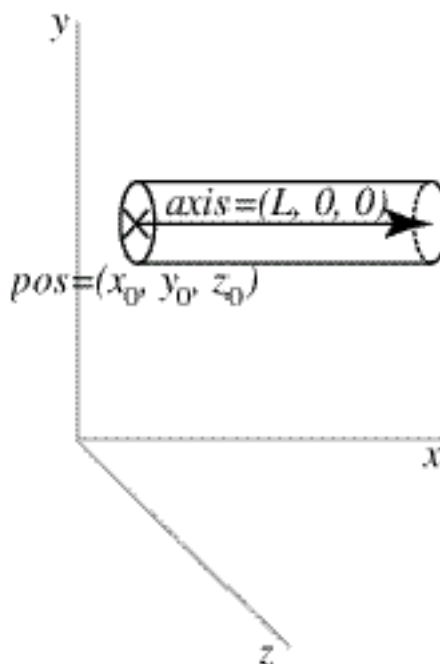
If you want an abrupt change in color or thickness, add another point at the same location. In the following example, adding a blue point at the same location as the third (red) point makes the final line be purely blue.

```
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
c.append( pos=(0,1,0) ) # add blue point
```

Technical note: No matter how many points are in a curve, only 1000 are displayed, selected evenly over the full set of points, in order that the display of a very long curve doesn't slow down unacceptably.

See description of **Additional Attributes** available for all 3D display objects.

cylinder



Studying this description of the cylinder object provides an overview of important aspects common to all of the Visual 3D objects, box, sphere, pyramid, etc.

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 (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

opacity Opacity of object, default = 1; 0 is completely transparent

material Material of object; see **Materials** for currently available options

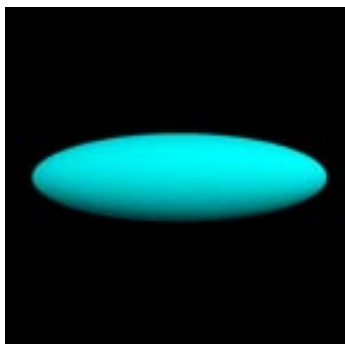
up Which side of the cylinder is "up"; this has only a subtle effect on the 3D appearance of the cylinder unless a non-smooth material is specified; default (0,1,0)

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.

See description of **Additional Attributes** available for all 3D display objects.

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.

ellipsoid



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 ellipsoid 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), **opacity** (1), **material**, and **up** (0,1,0). Additional attributes, similar to those for a box:

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.

See description of **Additional Attributes** available for all 3D display objects.

faces



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**).

You can specify a **material**, but currently you can not specify opacity for faces.

See description of **Additional Attributes** available for all 3D display objects.

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

frame

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) # change orientation of both objects
f.pos = (-1,0,0) # change position of both objects
```

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.

You can make all the objects in a frame invisible or visible by setting the frame's **visible** attribute.

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, not lights, which are found in `scene.lights`). 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_axis
```

helix



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

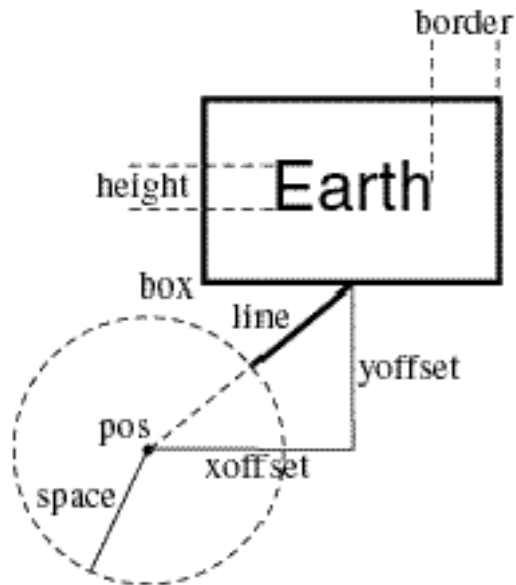
```
spring = helix(pos=(0,2,1), axis=(5,0,0), radius=0.5)
```

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** (1), **coils** (5), **thickness** (radius/20), **color** (1,1,1) which is color.white, **red** (1), **green** (1), **blue** (1), **material**, 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.

Currently it is not possible to specify the opacity of a helix object, which is based on the curve object. See description of **Additional Attributes** available for all 3D display objects.

label



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 with 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, font='sans')
```

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. If there are no offsets (see diagram), the center of the text is at **pos**
xoffset, yoffset The x and y components of the line, in pixels (see diagram). You can **left justify** text by setting **xoffset** = 1 and **line** = 0 (so the 1-pixel line doesn't show), or **right-justify** text by setting **xoffset** = -1 and **line** = 0. **text** The text to be displayed, such as 'Earth'

(Line breaks can be included as \n, as in label.text = "Three\nlines\nof text")

font Name of the desired font; for example, 'sans', or 'serif', or 'monospace' (fixed-width)

Python Unicode strings are supported.

height Height of the font in pixels; default is 13 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; default is 5 pixels

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

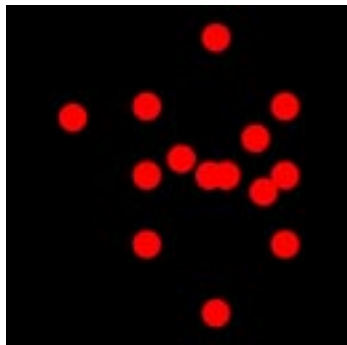
line 1 if the line from the box to pos should be drawn (default), else 0

linecolor Color of the line and box

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

See description of **Additional Attributes** available for all 3D display objects.

points



The points object takes a list of points for **pos**, like the curve object. The following statement will display two points, each of radius 50 pixels:

```
points(pos=[(-1,0,0), (1,0,0)], size=50, color=color.red)
```

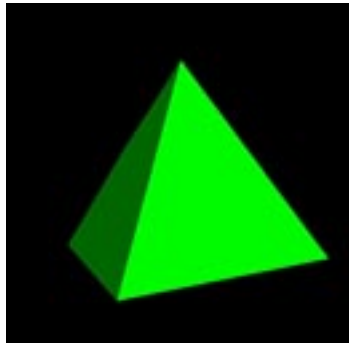
A new points object is similar to a curve, but with disconnected points. As with curve, the pos attribute is an array of points, and color can also be an array. If you say **shape="round"**, the points are round, which is the default; **shape="square"** makes square points. The size of the points is specified by **size**, and the default size is 5 (meaning a square 5 by 5, or a circular disk bounded by a 5 by 5 square). The size attribute is in screen pixels if **size_units="pixels"** (the default), but if **size_units="world"**, the size is in the usual coordinates. The maximum size of a point is about 50 by 50 pixels; specifying a larger size than the maximum does not increase the size.

Lighting does not affect the appearance, which is determined solely by the color. You cannot specify a material for points, and currently it is not possible to specify the opacity of a points object.

See description of **Additional Attributes** available for all 3D display objects.

Technical caveat: size_units="world" may not work on very old video drivers which do not support OpenGL 1.4 or the ARB_POINT_PARAMETERS extension. If you have problems, upgrade your video driver.

pyramid



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), **opacity** (1), **material**, 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

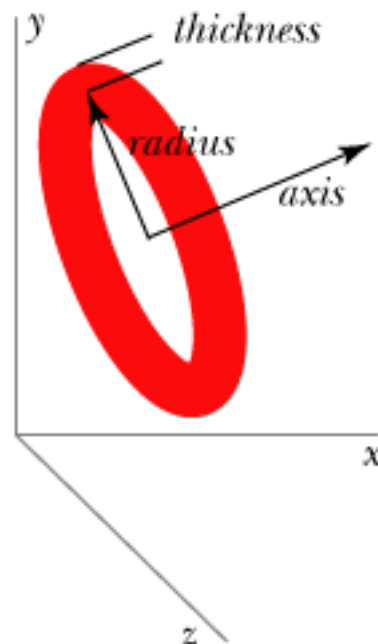
size (length, height, width), default is (1,1,1)

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.

See description of **Additional Attributes** available for all 3D display objects.

ring



The ring object is circular, with a specified 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), **opacity** (1), **material**, and **up** (0,1,0). As with cylinders, up has a subtle effect on the 3D appearance of a ring unless a non-smooth material is specified. The **axis** attribute only affects the orientation of the ring; the magnitude of the **axis** attribute is irrelevant. Additional ring attributes:

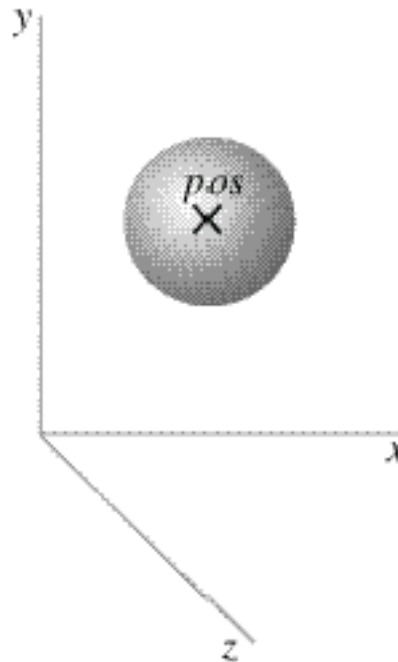
radius Radius of the central part of the ring, default = 1, so
 outer radius = radius+thickness
 inner radius = radius-thickness

thickness The radius of the cross section of the ring (1/10th of radius if not specified), not the full diameter as you might expect.

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.

See description of **Additional Attributes** available for all 3D display objects.

sphere



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), **opacity** (1), **material**, and **up** (0,1,0). As with cylinders, **up** and **axis** attributes affect the orientation of the sphere but have only a subtle effect on appearance unless a non-smooth material is specified. 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.

See description of **Additional Attributes** available for all 3D display objects.

Working with objects

Color and Opacity

Color

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; or you can say **color=color.gray(0.5)** to mean (0.5,0.5,0.5)

(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.

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 # (0.166667, 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))**

Opacity

You can make most objects be transparent by specifying a value from 0-1 inclusive for the attribute "opacity". For example, **box(color=color.red, opacity=0.8)** is slightly transparent. An opacity value of 0 means totally transparent, and 1 means totally opaque. Currently curve, convex, faces, points, and helix objects do not allow transparency.

You may see incorrect rendering any time there is a translucent object (opacity < 1.0) which is not convex (e.g. ring), or two translucent objects which overlap on the screen and also in their depth extents (distances from the camera to the nearest and farthest planes perpendicular to scene.forward which intersect the object). The objects need not actually overlap in 3D space to have problems. The incorrect rendering will usually have the effect of making the more distant object disappear (fail to show through the nearer object). Accurate rendering of ad hoc scenes with translucency is difficult and expensive, and we did not want to wait for a perfect solution before introducing this useful enhancement.

Lighting

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 its upper left corner at the upper left corner of the screen (y is measured down from the top of the screen), with 'Examples of Tetrahedrons' in the title bar, centered on location (5,0,0), and with a background color of cyan filling the window.

```
scene2 = display(title='Examples of Tetrahedrons',
                 x=0, y=0, 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()**

Executing **myscene = display.get_selected()** returns a reference to the display in which objects are currently being created.

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.

ambient Color of nondirectional ("ambient") lighting. Default is color.gray(0.2); for compatibility with earlier versions of Visual, this can be expressed as **scene.ambient=0.2**. Also see the following **lights** attribute.

lights List of light objects created for this display. By default, a display has two distant lights:

distant_light(direction=(0.22, 0.44, 0.88), color=color.gray(0.8)) and **distant_light(direction=(-0.88, -0.22, -0.44), color=color.gray(0.3))**.

These are equivalent to the default lights in Visual prior to version 5. You can get rid of these default lights with **scene.lights = []**. The color of light objects and the amount of **scene.ambient** must be specified with some care, because if the total lighting intensity exceeds 1 anywhere in the scene the results are unpredictable. **scene.lights** is a list of any lights you have created.

You can create lights that are local, near other objects. The following statement creates a yellow light positioned at (x,y,z), and if you continually update **lamp.pos**, the light will move. You may wish to place a sphere or box with **material=materials.emissive** at the same location so that the lamp looks like a glowing lamp.

```
lamp = local_light(pos=(x,y,z), color=color.yellow)
```

A distant red light located in the *direction* (x,y,z) is created like this:

```
distant_light(direction=(x,y,z), color=color.red)
```

Previous to Visual version 5, you set up a light by specifying a vector in the direction to the light from the origin, and the magnitude of the vector was the intensity. For example, **scene.lights = [vector(1,0,0)]** with **scene.ambient = 0** will light the scene with full intensity from the right side, with no ambient lighting on the left. In Visual version 5 and later, this scheme for specifying lights still works, but it is preferable to create light objects.

To obtain **camera position**, see **Mouse Interactions**.

objects A list of all the visible objects in the display; invisible objects and lights are not listed (**scene.lights** is a list of existing lights). For example, the following makes all visible boxes in the scene red:

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

show_rendertime If you set **scene.show_rendertime = 1**, in the lower left corner of the display you will see something like "cycle: 27 render: 5", meaning 27 milliseconds between renderings of the scene, taking 5 milliseconds to render, in which case 22 out of 27 milliseconds were devoted to executing your Python statements.

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 = 'cross-eyed'** 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 CRT displays and 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.)

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**. **NOT YET IMPLEMENTED IN VISUAL 5.**

Controlling the window

The window attributes **x**, **y**, **width**, **height**, **title**, and **fullscreen** cannot be changed while a window is active; they are used to create a window, not to change one. If you want to modify any of these window attributes, first make the window invisible, make the changes, and then make the window visible again. This creates a new window with the new attributes; all existing objects are still part of the new 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` (includes title bar).

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

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; the Escape key has no effect. If you use the fullscreen option on Linux, be sure to program a mouse or keyboard input for quitting the program.)

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.

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 the inverse of scale, so use either **range** or **scale** depending on which makes the most sense in your program. Setting range to 10 is the same as setting it to (10,10,10). Setting range to (10,0,0) means that `scene.center+scene.range` will be at the right edge of a square window. A sphere of radius 10 will fill the window. A cubical box whose half-width is 10 will overfill the window, because the front of the box in 3D appears larger than the xy plane passing through `scene.center`, unless the field of view is very small.

scale A scaling factor which scales the region of interest into the sphere with unit radius. This is the inverse of range, so use either **range** or **scale** depending on which makes the most sense in your program. Setting scale to 0.1 is the same as setting it to (0.1,0.1,0.1) or setting range to (10,10,10).

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.

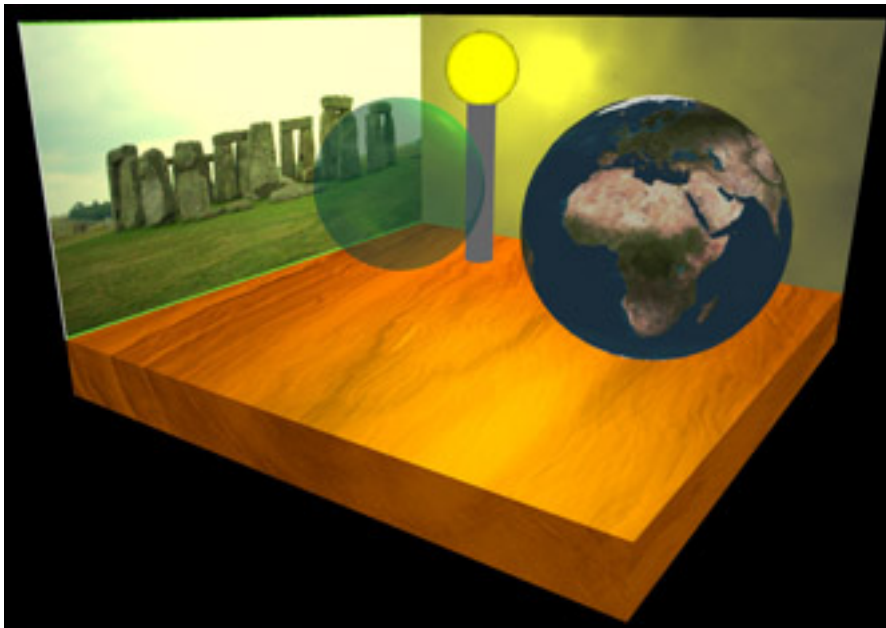
userzoom = 0 user cannot zoom in and out of the scene

userzoom = 1 user can zoom (default)

userspin = 0 user cannot rotate the scene

userspin = 1 user can rotate (default)

Materials and Textures



You can specify a material such as wood for any object other than a points object:

```
sphere(color=color.orange, material=materials.wood)
```

The materials that are currently available include these:

- ***materials.wood***
- ***materials.rough***
- ***materials.marble***
- ***materials.plastic***
- ***materials.earth***
- ***materials.diffuse***
- ***materials.emissive (looks like it glows)***
- ***materials.unshaded (unaffected by lighting)***

The example program ***material_test.py*** displays all of these materials. The ***emissive*** material is particularly appropriate for simulating the appearance of a light glowing with the specified color. This apparent light has no lighting effect on other objects, but you may wish to place a `local_light` at the same location, as is done with the swinging light in the example program ***texture_and_lighting.py***. The appearance of the ***unshaded*** material is unaffected by lighting and is useful when you want to display an object whose appearance is determined solely by its own attributes.

Materials will work with graphics cards that support Pixel Shader 3.0 ("PS 3.0"). For details, see http://en.wikipedia.org/wiki/Pixel_shader#Hardware. Some materials may work with graphics cards that support PS 2.0, but other materials may need to be manually disabled; see instructions in the ***site-settings.py*** module in the Visual package in your site-packages folder. If the graphics hardware does not support pixel shaders, the material property

is ignored. If you think you should be able to use materials but have trouble with their display or performance, we highly recommend upgrading your video card drivers to the latest version.

Some materials such as wood are oriented to the specified axis. For example, a wood box with default axis = (1,0,0) shows tree rings on its yz surfaces and stripes on the other faces. Changing the axis changes which face you see the tree rings on.

Creating your own texture

You can create a texture object and then apply it to the surface of an object. A surface texture is an M by N array of slots containing 1, 2, 3, or 4 numerical values. M and N must be powers of 2 (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, etc.). The numerical values can represent color, luminance (brightness or shades of gray), or opacity.

Here are the possibilities for each slot in the array:

- 1 value: luminance by default, or specify **channels=["opacity"]** to represent opacity
- 2 values: luminance and opacity
- 3 values: red,green,blue
- 4 values: red,green,blue,opacity

Here is an example program in which a 4 by 4 by 1 checkerboard texture is created and applied to a box:

```
from visual import *
checkerboard = ( (0,1,0,1),
                 (1,0,1,0),
                 (0,1,0,1),
                 (1,0,1,0) )
tex = materials.texture(data=checkerboard,
                        mapping="rectangular",
                        interpolate=False)
box(axis=(0,0,1), color=color.cyan, material=tex)
```

The example above uses a **rectangular** mapping, which places the texture on two opposing faces of a box, with stripes along the sides. By default, one of the faces is in the (1,0,0) direction, but this can be changed by specifying a different axis for the box, as was done in the example above. A **sign** mapping is similar to rectangular but is unaffected by the color of the object and appears on only one face of a box (determined by the axis of the box). A **spherical** mapping wraps around the entire object. In the example program **texture_and_lighting.py** you can find a creation of a beach ball using spherical mapping.

By default **interpolate** is True, but to get a sharply defined checkerboard in the example above, it was set to False. You can save the texture data in a file for later use:

```
materials.saveTGA("checks", checkerboard)
```

This saves the checkboard pattern in a file "checks.tga", a targa file which many graphics applications can display. In later programs you can use this data without recreating it:

```
data = materials.loadTGA("checks")
```

More generally, any targa file whose width and height are both powers of 2 can be read as data using **materials.loadTGA(filename)**. If the actual file name is "checks.tga" you can give the full file name or just "checks". One way to create a pattern is to start by creating a numpy array of zeros, then assign values to individual slots:

```
pattern = zeros((4,8,3)) # 4 by 8 by 3 numpy array of 0's
pattern[0][0] = (1,.5,.7) # assign first rgb triple
```

Another example

Here is an example of placing a "sign" on one face of a box, consisting of a 2 by 2 by 3 grid of color components:

```

from visual import *
grid = ( (color.red, (1, 0.7, 0)),
         ((0, 1, 0.3), color.magenta) )
tgrid = materials.texture(data=grid,
                          mapping="sign",
                          interpolate=False)
box(axis=(0,0,1), material=tgrid)
    
```

Making a texture from a photo

A texture can be created from a targa file, and various graphics applications can convert photos in jpeg or other formats to targa files. One tool for doing this is PIL, the Python Imaging Library, which can be downloaded and installed (you can find it with a web search). Here is an example of PIL code which converts a jpeg photo into a targa file which can be used to create a texture for displaying the image, as in the example program **stonehenge.py**.

```

from visual import *
import Image # Must install PIL
name = "flower"
width = 128 # must be power of 2
height = 128 # must be power of 2
im = Image.open(name+".jpg")
#print im.size # optionally, see size of image
# Optional cropping:
#im = im.crop((x1,y1,x2,y2)) # (0,0) is upper left
im = im.resize((width,height), Image.ANTIALIAS)
materials.saveTGA(name,im)
    
```

At a later time you can say **data = materials.loadTGA(name)** to retrieve the image data from the targa file. As a convenience, a texture can also be created directly from the PIL image data, like this:

```

tex = materials.texture(data=im, mapping="sign")
    
```

Efficiency issues

Normally you create a data pattern containing values in the range from 0.0 to 1.0, the standard range of color components and opacity in Visual. However, the underlying graphics machinery works with values in the range of 0 to 255, which can be expressed in one 8-bit byte of computer memory. If you are dealing with large textures and time is critical, you should avoid conversions from the range 0-1 to the range 0-255 by constructing the texture data from a numpy array of unsigned 8-bit bytes. An unsigned byte is referred to as **ubyte**. Here is a simple example:

```

checkers = array( ( (0,255,0,255),
                   (255,0,255,0),
                   (0,255,0,255),
                   (255,0,255,0) ), ubyte)
    
```

The array function converts a sequence of values into a numpy array. In this case the values are 8-bit bytes.

Channels

Data "channels" are a part of the definition of a texture. For the most part, these channels are assigned automatically for you, like this:

- 1 value: **channels=["luminance"]** by default, **channels=["opacity"]** to represent opacity
- 2 values: **channels=["luminance","opacity"]**
- 3 values: **channels=["red","green","blue"]**
- 4 values: **channels=["red","green","blue","opacity"]**

Except for specifying that a pattern represents opacity rather than luminance (brightness, or shade of gray), it isn't necessary to specify channels when constructing a texture because the channel options shown above are currently the only valid sets of channels. However, it is expected that in the future there may be additional channels available, such as glossiness.

mipmap

When an object in the scene is small and far away, there is no need to display its texture in full detail. With the default **mipmap=True**, Visual prepares a set of smaller textures to use when appropriate. These additional textures take some time to prepare for later use, and required storage space is one-third larger, but they can speed up the rendering of a scene. It should rarely be the case that you would need to set **mipmap=False**.

Creating your own materials

Creating your own materials (in contrast to creating textures) is technically somewhat challenging. The program **materials.py**, a component of the Visual module, contains the shader models for wood and other materials, and it also contains instructions on how to build your own materials. Shader models are written in a C-like language, GLSL (OpenGL Shader Language).

Defaults

Convenient Defaults

Objects can be specified with convenient defaults:

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

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

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

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

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

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

`ellipsoid()` is equivalent to `ellipsoid(pos=(0,0,0), size=(1,1,1))`

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

`helix()` is equivalent to `helix(pos=(0,0,0), axis=(1,0,0), radius=1, thickness=0.05, coils=5)`

`points()` establishes an "empty" set of points to which points can be appended

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

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

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

Animation Speed

Limiting the Animation Rate

rate(frequency)

Halts computations until $1.0/\text{frequency}$ 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.

Rotations

Rotating an Object

Objects other than curve, convex, faces, and points can be rotated about a specified origin (to rotate these other objects, put them in a frame and rotate the frame).

```
obj.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**.

Also see the rotation function available for **vectors**.

Additional Options

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 )
```

Executing **myscene = display.get_selected()** returns a reference to the display in which objects are currently being created. Given a specific display named scene2, **scene2.select()** makes scene2 be the "selected display", so that objects will be drawn into scene2 by default.

There is a **rotate function** for all objects other than curve, convex, faces, and points (which can be put into a frame and the frame rotated).

__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 **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. 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, as long as there are no nested frames (frames within frames):

```
def clone_universe( new_display, old_display):
    # Create a dictionary of frames in old display to
    # the corresponding frames in the new display.
    frames = {} # create empty dictionary
    for obj in old_display.objects:
        if obj.__class__ == frame:
            frames[obj] = obj.__copy__( frame=None,
                                       display=new_display)
    # For each old frame within another reference frame,
    # place the new frame in appropriate frame in 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.
            pass
        elif obj.frame:
            # Initialize with the corresponding frame
            # in the new display:
            obj.__copy__( display=new_display,
                         frame=frames[obj.frame])
        else:
            # No frame issue;
            obj.__copy__( display=new_display)
```

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

Delete an Object

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).

3/4 = 0 ?

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.

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.)***

Windows/Events/Files

Windows

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 its upper left corner at the upper left corner of the screen (y is measured down from the top of the screen), with 'Examples of Tetrahedrons' in the title bar, centered on location (5,0,0), and with a background color of cyan filling the window.

```
scene2 = display(title='Examples of Tetrahedrons',
                x=0, y=0, 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()***

Executing ***myscene = display.get_selected()*** returns a reference to the display in which objects are currently being created.

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.

ambient Color of nondirectional ("ambient") lighting. Default is color.gray(0.2); for compatibility with earlier versions of Visual, this can be expressed as ***scene.ambient=0.2***. Also see the following ***lights*** attribute.

lights List of light objects created for this display. By default, a display has two distant lights:

distant_light(direction=(0.22, 0.44, 0.88), color=color.gray(0.8)) and **distant_light(direction=(-0.88, -0.22, -0.44), color=color.gray(0.3))**.

These are equivalent to the default lights in Visual prior to version 5. You can get rid of these default lights with **scene.lights = []**. The color of light objects and the amount of **scene.ambient** must be specified with some care, because if the total lighting intensity exceeds 1 anywhere in the scene the results are unpredictable. **scene.lights** is a list of any lights you have created.

You can create lights that are local, near other objects. The following statement creates a yellow light positioned at (x,y,z), and if you continually update lamp.pos, the light will move. You may wish to place a sphere or box with **material=materials.emissive** at the same location so that the lamp looks like a glowing lamp.

```
lamp = local_light(pos=(x,y,z), color=color.yellow)
```

A distant red light located in the *direction* (x,y,z) is created like this:

```
distant_light(direction=(x,y,z), color=color.red)
```

Previous to Visual version 5, you set up a light by specifying a vector in the direction to the light from the origin, and the magnitude of the vector was the intensity. For example, **scene.lights = [vector(1,0,0)]** with **scene.ambient = 0** will light the scene with full intensity from the right side, with no ambient lighting on the left. In Visual version 5 and later, this scheme for specifying lights still works, but it is preferable to create light objects.

To obtain **camera position**, see [Mouse Interactions](#).

objects A list of all the visible objects in the display; invisible objects and lights are not listed (scene.lights is a list of existing lights). For example, the following makes all visible boxes in the scene red:

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

show_rendertime If you set **scene.show_rendertime = 1**, in the lower left corner of the display you will see something like "cycle: 27 render: 5", meaning 27 milliseconds between renderings of the scene, taking 5 milliseconds to render, in which case 22 out of 27 milliseconds were devoted to executing your Python statements.

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 = 'cross-eyed'** 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 CRT displays and 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.)

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**. **NOT YET IMPLEMENTED IN VISUAL 5.**

Controlling the window

The window attributes **x**, **y**, **width**, **height**, **title**, and **fullscreen** cannot be changed while a window is active; they are used to create a window, not to change one. If you want to modify any of these window attributes, first make the

window invisible, make the changes, and then make the window visible again. This creates a new window with the new attributes; all existing objects are still part of the new 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` (includes title bar).

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

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; the Escape key has no effect. If you use the fullscreen option on Linux, be sure to program a mouse or keyboard input for quitting the program.)

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.

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 the inverse of scale, so use either **range** or **scale** depending on which makes the most sense in your program. Setting range to 10 is the same as setting it to (10,10,10). Setting range to (10,0,0) means that `scene.center+scene.range` will be at the right edge of a square window. A sphere of radius 10 will fill the window. A cubical box whose half-width is 10 will overfill the window, because the front of the box in 3D appears larger than the xy plane passing through `scene.center`, unless the field of view is very small.

scale A scaling factor which scales the region of interest into the sphere with unit radius. This is the inverse of range, so use either **range** or **scale** depending on which makes the most sense in your program. Setting scale to 0.1 is the same as setting it to (0.1,0.1,0.1) or setting range to (10,10,10).

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 Events

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 Command key, and the middle button is invoked by holding down the Option 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 mice). 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 $\{ \text{camera} + t \cdot \text{ray} \text{ for } t > 0 \}$.

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)

Getting events

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

- 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.
- A release event occurs when the mouse buttons are released after a click or 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' for a press event, or None ('right' or 'middle' not currently implemented)

click = 'left' for a click event, or None ('right' or 'middle' not currently implemented); 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**.

See **Click example**.

drag = 'left' for a drag event, or None ('right' or 'middle' not currently implemented); 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**.

See **Drag example**.

drop = 'left' for a drop event, or None ('right' or 'middle' not currently implemented)

release = 'left' following click and drop events, indicating which button was released, or None ('right' or 'middle' not currently implemented)

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)
```

Mouse Click

Click Example

This program displays a box (which automatically creates a window referred to as **scene**), then repeatedly waits for a mouse left click, prints the mouse position in the Python shell window, and displays a cyan sphere. A mouse click is defined as pressing and releasing the left mouse button with almost no motion of the mouse, so the sphere appears when you release the mouse button.

```
from visual import *
scene.range = 4
box() # display a box for context
while 1:
    if scene.mouse.clicked:
        m = scene.mouse.getclick()
        loc = m.pos
```

```
print loc
sphere(pos=loc, radius=0.2, color=color.cyan)
```

Copy this program into an IDLE window and run the program. Click outside the box and a cyan sphere appears where you click. If you click inside the box, nothing seems to happen. This is because the mouse click is in the xy plane, and the sphere is buried inside the box. If you rotate the scene and then click, you'll see that the spheres go into the new plane parallel to the screen and passing through **scene.center**. If you want *all* of the spheres to go into the xy plane, perpendicular to the z axis, change the latter part of the program like this:

```
loc = m.project(normal=(0,0,1))
# loc is None if no intersection with plane
if loc:
    print loc
    sphere(pos=loc, radius=0.2, color=color.cyan)
```

Here is **general mouse documentation**.

Mouse Drag

Drag example

Here is the sequence of mouse events involved in dragging something:

- 1) m1.press is true when you depress the mouse button (it is 'left' if left button; any quantity that is nonzero is considered true in Python).
 - 2) m1.drag is true when the mouse coordinates change from what they were at the time of m1.press.
- At the time of the drag event, the mouse position is reported to be what it was at the time of the press event, so that the dragging can start at the place where the user first clicked. If the mouse is in motion at the time of the press event, it is quite possible that the next position seen by the computer, at the time of the drag event, could be quite far from the click position. This is why the position of the drag event is reported as though it occurred at the press location.
- 3) No events occur while dragging; you continually use scene.mouse.pos to update what you're dragging.
 - 4) m1.drop is true when you release the mouse button.

You can program dragging with the mouse simply by continually reading the current value of **scene.mouse.pos**. Here is a complete routine for dragging a sphere with the left button down. Copy this into an edit window and try it!

```
from visual import *
scene.range = 5 # fixed size, no autoscaling
ball = sphere(pos=(-3,0,0), color=color.cyan)
cube = box(pos=(+3,0,0), size=(2,2,2), color=color.red)
pick = None # no object picked out of the scene yet
while 1:
    if scene.mouse.events:
        m1 = scene.mouse.getevent() # get event
        if m1.drag and m1.pick == ball: # if touched ball
            drag_pos = m1.pickpos # where on the ball
            pick = m1.pick # pick now true (not None)
        elif m1.drop: # released at end of drag
            pick = None # end dragging (None is false)
    if pick:
        # project onto xy plane, even if scene rotated:
        new_pos = scene.mouse.project(normal=(0,0,1))
        if new_pos != drag_pos: # if mouse has moved
            # offset for where the ball was clicked:
            pick.pos += new_pos - drag_pos
            drag_pos = new_pos # update drag position
```

If you do a lot of processing of each mouse movement, or you are leaving a trail behind the moving object, you may need to check whether the "new" mouse position is in fact different from the previous position before processing the "move", as is done in the example above. For example, a trail drawn with a curve object that contains a huge number of points all at the same location may not display properly.

Most VPython objects can be "picked" by clicking them. Here is a more general routine which lets you drag either the tail or the tip of an arrow. Copy this into an edit window and try it!

```

from visual import *
scene.range = 8 # fixed size, no autoscaling
arr = arrow(pos=(2,0,0),axis=(0,5,0))
by = 0.3 # click this close to tail or tip
drag = None # have not selected tail or tip of arrow
while 1:
    if scene.mouse.events:
        m1 = scene.mouse.getevent() # obtain event
        if m1.press:
            if mag(arr.pos-m1.pos) <= by:
                drag = 'tail' # near tail of arrow
            elif mag((arr.pos+arr.axis)-m1.pos) <= by:
                drag = 'tip' # near tip of arrow
            drag_pos = m1.pos # save press location
        elif m1.drop: # released at end of drag
            drag = None # end dragging (None is False)
    if drag:
        new_pos = scene.mouse.pos
        if new_pos != drag_pos: # if mouse has moved
            displace = new_pos - drag_pos # moved how far
            drag_pos = new_pos # update drag position
            if drag == 'tail':
                arr.pos += displace # displace the tail
            else:
                arr.axis += displace # displace the tip
    
```

Here is **general mouse documentation**.

Keyboard Events

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.

Multicharacter names include delete, backspace, page up, page down, home, end, left, up, right, down, numlock, scrlck, f1, f2, f3, f4, f5, f6, f7, f8. Windows and Linux also have f9, f11, f12, insert.

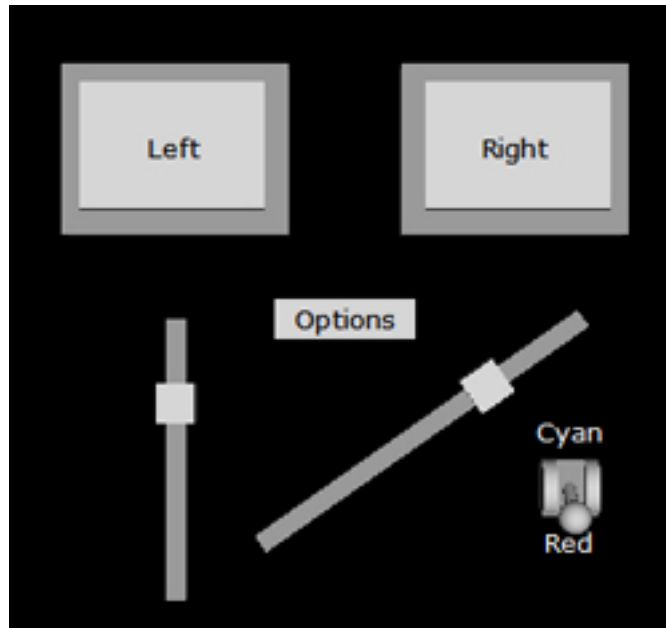
Here is a test routine that lets you type text into a label:

```

from visual import *
prose = label() # initially blank text
while 1:
    if scene.kb.keys: # event waiting to be processed?
        s = scene.kb.getkey() # get keyboard info
        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 letter
        elif s == 'shift+delete':
            prose.text = '' # erase all text
    
```

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

Button and Sliders



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 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; drive 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).

display Every controls window has the attribute **display**; sphere(display=c.display) will place a sphere in the controls window named c.

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), slider (depends on slider min and max), or menu (the text just selected on the menu).

The value of a toggle or slider (but not a menu) 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 the header at the top of a menu

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)

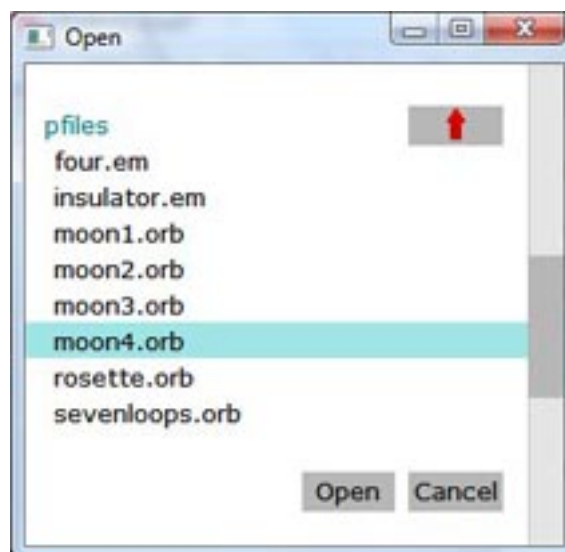
action Specify Python statement to be executed when a control is manipulated

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.

Reading/Writing Files



Reading and Writing Files

A simple file dialog package is provided in the module **visual.filedialog**.

Here is how to get a file dialog display to choose a file to read, and then display the contents. The `get_file()` routine lets you choose a file, and it returns a file descriptor, a pointer to the chosen file (here the file descriptor has been named `fd`). If you cancel the file dialog display, `get_file()` returns `None`, which you should check for (the statements just after the "if `fd`:" will be executed only if `fd` is not `None`). Using the file descriptor you can read the entire file as one long string, or with `readlines()` you can read a list of lines of text, each ending with an end-of-line character (`\n`).

```
from visual.filedialog import get_file
fd = get_file()
if fd:
    data = fd.read() # or fd.readlines()
    fd.close() # close the file (we're through with it)
    print data
```

To choose a file and write data to the chosen file, do this:

```
from visual.filedialog import save_file
fd = save_file()
if fd:
    fd.write("This is a test.\nThis is only a test.")
    fd.close() # close the file (we're through with it)
```

There are other file descriptor functions besides `read()`, `readlines()`, `write()`, and `close()`; see Python documentation. For example, `fd.name()` is the name of the file associated with the file descriptor.

The examples shown above are sufficient for many tasks, but you can customize the file dialog display by specifying parameters for the `get_file()` and `save_file()` functions, as shown here with their default values:

```
save_file(file_extensions=None, x=100, y=100,
          title="Save", mode='w', maxfiles=20)

get_file(file_extensions=None, x=100, y=100,
         title="Open", mode='rU', maxfiles=20)
```

For example, **`get_file(file_extensions=['.txt.', '.py'])`** will display only files ending in these extensions. The parameter **`maxfiles`** specifies the maximum number of files to show on one page, which determines the height of the file dialog display. The default "universal" file reading mode is `'rU'` which converts different kinds of end-of-line markers for Windows, Mac, and Linux files to the same standard character `\n`. The parameters `x` and `y` specify the pixel location of the upper left corner of the file dialog display, measured from the upper left corner of the computer screen. The title is displayed at the top of the window.

Vector operations

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()**. For convenience, **norm(vector(0,0,0))** is calculated to be **vector(0,0,0)**.

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

```
v1.diff_angle(v2)
```

You can also write **v1.diff_angle(v1,v2)**. For convenience, if either of the vectors has zero magnitude, the difference of the angles is calculated to be zero.

There is a function for 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.

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

There is a function for 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.

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

You can also say **vector1.cross(vector2)** or **vector1.dot(vector2)**.

Rotating a vector

There is a function for 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. You can also write **v2 = v1.rotate(angle=theta, axis=(1,1,1))**. There is also a **rotate capability for objects**.

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.astuple()**.

Graphs

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 (arange creates a numeric array running from 0 to 8, stopping short of 8.1):

```
from visual.graph import * # import graphing features
funct1 = gcurve(color=color.cyan) # a graphics curve
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 specify a **delta** attribute, which specifies the width of the bar (the default is **delta=1**). For **gdots** you can specify a **shape** attribute "round" or "square" (default is **shape="round"**) and a **size** attribute, which specifies the width of the dot in pixels (default is **size=5**).

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:

```
values = [(1,2), (3,4), (-5,2), (-5,-3)]
data = gdots(pos=values, 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 -200. 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 place additional labels or manipulate the graphing window. The only objects that you can place in the graphing window are labels, curves, faces, and points.

```
graph1 = gdisplay()
label(display=graph1.display, pos=(3,2), text="P")
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, or you can specify which window a new object belongs to:

```
energy = gdots(gdisplay=graph2.display, 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.

factorial/combin

The factorial and combin Functions

```
from visual import *
from visual.factorial import * # import after visual
print factorial(4) # gives 24
print combin(10,2) # gives 45
```

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

The factorial function factorial(N) is N!; 4! is (4)(3)(2)(1) = 24, and 0! is defined to be 1.

The combin function is $\text{combin}(a,b) = a!/(b!(a-b)!)$.

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.

Visual license

The Visual library is Copyright (c) 2000 by David Scherer.

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of David Scherer not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

DAVID SCHERER DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL DAVID SCHERER BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

The following notice applies to the CXX library, which is linked into cvisual.dll:

*** Legal Notice for all LLNL-contributed files ***

Copyright (c) 1996. The Regents of the University of California. All rights reserved.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

This work was produced at the University of California, Lawrence Livermore National Laboratory under contract no. W-7405-ENG-48 between the U.S. Department of Energy and The Regents of the University of California for the operation of UC LLNL.

DISCLAIMER

This software was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark,

manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.