

Description of Objects in Visual

Visual is a 3D graphics facility developed by David Scherer to be used with the Python programming language. This reference document describes each of the Visual objects. Be sure to study the discussion of the cylinder object in detail, because much of what is said there applies to other objects as well.

The cylinder Object

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

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

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

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

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

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

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

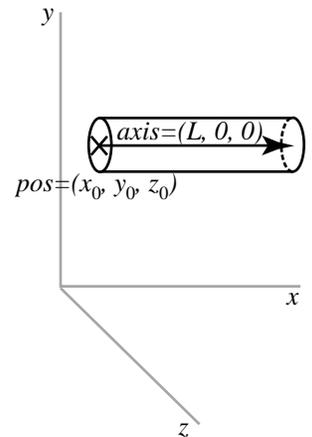
This will make the cylinder suddenly turn blue, using the so-called RGB system for specifying colors in terms of fractions of red, green, and blue. (For details on choosing colors, see "Specifying Colors" later in this reference.) You can set individual amounts of red, green, and blue like this:

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

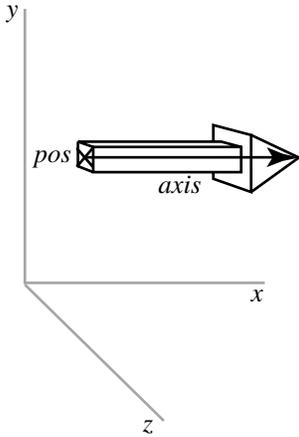
If you create an object such as a cylinder and don't give it a name such as `rod`, you won't be able to refer to it later. This doesn't matter if you never intend to modify the object.

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:

<code>pos</code>	Position: the center of one end of the cylinder A triple, in parentheses, such as $(3,2,5)$
<code>axis</code>	The axis points from <code>pos</code> to the other end of the cylinder
<code>x, y, z</code>	Essentially the same as <code>pos.x</code> , <code>pos.y</code> , <code>pos.z</code>
<code>radius</code>	The radius of the cylinder
<code>length</code>	Length of axis; if not specified, axis determines the length If length is specified, it overrides the length given by axis
<code>color</code>	Color of object, as a red-green-blue (RGB) triple: $(1,0,0)$ is pure red
<code>red, green, blue</code>	(can set these color attributes individually)
<code>up</code>	Which side of the cylinder is "up"; this has only a subtle effect on the 3D appearance of the cylinder



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



The arrow Object

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

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

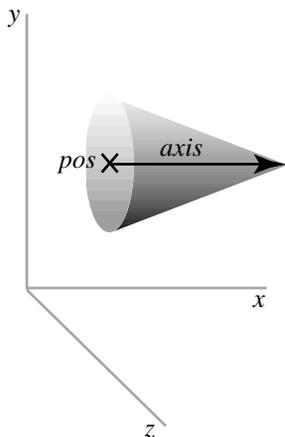
The arrow object has attributes `pos`, `x`, `y`, `z`, `axis`, `length`, `color`, `red`, `green`, `blue`, and `up` like those for cylinders. The `up` attribute is significant for arrow because the shaft and head have square cross sections, and setting the `up` attribute rotates the arrow about its axis. 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.



The cone Object

The cone object has a circular cross section and tapers to a point. The following statement will display a cone pointing parallel to the x axis; the wide end of the cone has the specified radius:

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

The cone object has attributes `pos`, `x`, `y`, `z`, `axis`, `length`, `color`, `red`, `green`, `blue`, and `up` like those for cylinders. As with cylinders, `up` has a subtle effect on the 3D appearance of a cone. Additional cone attribute:

`radius` Radius of the wide end of the cone

The sphere Object

Here is an example of how to make a sphere:

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

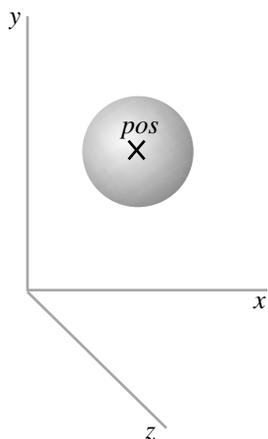
This produces a sphere centered at location (1,2,1) with radius = 0.5, with the current foreground color.

The sphere object has attributes `pos`, `x`, `y`, `z`, `axis`, `color`, `red`, `green`, `blue`, and `up` like those for cylinders. As with cylinders, `up` has a subtle effect on the 3D appearance of a sphere. The `axis` attribute only affects the orientation of the sphere and has a subtle effect on appearance; the magnitude of the `axis` attribute is irrelevant. Additional sphere attributes:

`radius` Radius of the sphere

`label` *Experimental:* `ball.label='Sun'` attaches 'Sun' to sphere

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



The ring Object

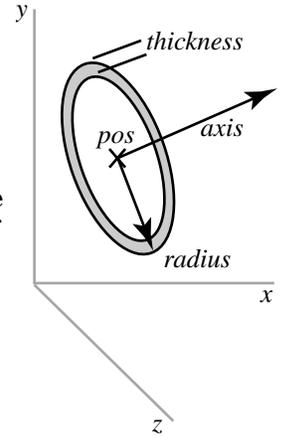
The ring object is circular, with a specified outer radius and thickness, and with its center given by the `pos` attribute:

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

The ring object has attributes `pos`, `axis`, `x`, `y`, `z`, `color`, `red`, `green`, `blue`, and `up` like those for cylinders. As with cylinders, `up` has a subtle effect on the 3D appearance of a ring. The `axis` attribute only affects the orientation of the ring; the magnitude of the `axis` attribute is irrelevant. Additional ring attributes:

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

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

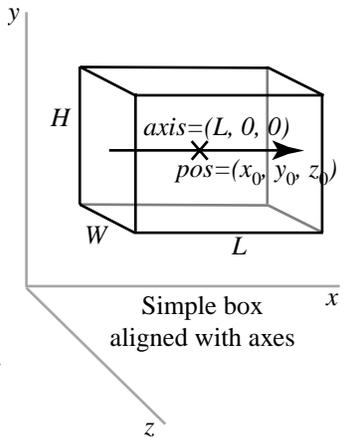


The box Object

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

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

The given position is in the center of the box, at (x_0, y_0, z_0) . 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.

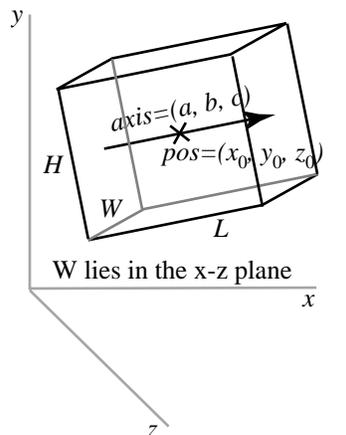


For a box that isn't aligned with the coordinate axes, additional issues come into play. The orientation of the length of the box is given by the `axis` (see second diagram):

```
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 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 attributes `pos`, `x`, `y`, `z`, `axis`, `length`, `color`, `red`, `green`, `blue`, and `up` like those for cylinders. Additional box attributes:

<code>height</code>	In the y direction in the simple case
<code>width</code>	In the z direction in the simple case
<code>size</code>	(length, height, width)

`mybox.size=(20,10,12)` sets length=20, height=10, width=12

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

The curve Object

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Parametric graphing is also easy:

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

Here are the curve attributes:

```
pos[]      Array of position of points in the curve: pos[0], pos[1], pos[2]...
           The current number of points is given by len(curve.pos)
x[ ], y[ ], z[ ] Components of pos; each defaults to [0,0,0,0,...]
color[ ]   Color of points in the curve
red[ ], green[ ], blue[ ] Color components of points in the curve
radius     Radius of cross-section of curve, for all elements of the curve
           The default radius=0 makes a thin curve
```

Adding more points to a curve

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

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

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

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

Interpolation

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

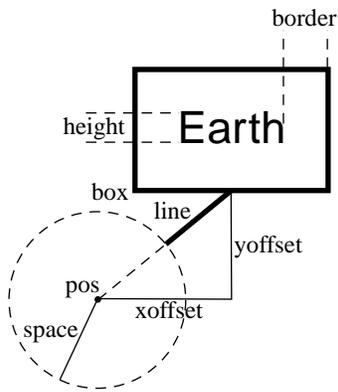
```
c = curve( pos=[(0,0,0), (1,0,0)], color=color.red )
c.append( pos=(1,1,0) ) # add a red point
c.append( pos=(0,1,0), color=color.blue) # add blue point
```

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

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

The convex Object

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



The label Object

With the label object you can display text in a box, linked by a line to a specified point. In the accompanying diagram, a sphere representing the Earth (whose center is at `earth.pos`) has an associated label carrying the text “Earth” in a box, connected to the sphere by a line which stops at the surface of the sphere:

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

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

<code>pos; x,y,z</code>	The point in world space being labeled
<code>xoffset, yoffset</code>	The x and y components of the line, in pixels (see diagram)
<code>text</code>	The text to be displayed, such as ‘Earth’ (Line breaks are not yet supported)
<code>height</code>	Height of the font in pixels
<code>color, red, green, blue</code>	Color of the text
<code>opacity</code>	Opacity of the background of the box, default 0.75 (0 transparent, 1 opaque, for objects behind the box)
<code>border</code>	Distance in pixels from the text to the surrounding box
<code>box</code>	1 if the box should be drawn (default), else 0
<code>line</code>	1 if the line from the box to <code>pos</code> should be drawn (default), else 0
<code>linecolor</code>	Color of the line and box
<code>space</code>	World-space radius of a sphere surrounding <code>pos</code> , into which the connecting line does not go
<code>font</code>	Optional name of the font, such as ‘helvetica’

Composite Objects with frame

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

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

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

Convenient Defaults

Objects can be specified with convenient defaults:

cylinder() is equivalent to cylinder(pos=(0,0,0), axis=(1,0,0), radius=1)
arrow() is equivalent to arrow(pos=(0,0,0), axis=(1,0,0), radius=1)
cone() is equivalent to cone(pos=(0,0,0), axis=(1,0,0), radius=1)
sphere() is equivalent to sphere(pos=(0,0,0), radius=1)
ring() is equivalent to ring(pos=(0,0,0), axis=(1,0,0), radius=1)
box() is equivalent to box(pos=(0,0,0), length=1, height=1, width=1)
curve() establishes an “empty” curve to which points can be appended
convex() establishes an “empty” object to which points can be appended
frame() establishes a frame with pos=(0,0,0) and axis=(1,0,0)

Rotations

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

```
object.rotate(angle=None, 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`.

Additional Options for Objects

The following attributes apply to all Visual 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 `frame=f1`

display When you start a Visual 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 )
```

See "Controlling One or More Visual Display Windows" later in this reference for more information on creating and manipulating display objects.

Specifying Colors

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

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

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

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

(0.5, 0.5, 0.5) a rather dark grey	(1,0.7,0.2) a coppery color
------------------------------------	-----------------------------

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

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

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

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

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

Limiting the Animation Rate

`rate(frequency)`

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

For example, `rate(50)` will halt computations for 1.0/50.0 second. If you place `rate(50)` inside a computational loop, the loop will execute only 50 times per second, even if the computer can run faster than this. This makes animations look about the same on computers of different speeds, as long as the computers are capable of carrying out 50 computations per second.

Floating Division

Standard 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. Alternatively, you can insert the following line, which makes `3/4` evaluate to 0.75 for statements in a Visual program on Windows (not yet available on other platforms, nor with standard Python):

```
import floatdivision
```

Also, Visual converts integers to floating-point numbers for you:

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

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. Numeric is imported 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 the length of the given vector.

`mag(vector(1,1,1))` is equal to the square root of 3

You can also obtain the magnitude in the form `v2.mag`.

`norm(vector)`

Produces a normalized form of the given vector; that is, its magnitude is 1.

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

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

`cross(vector1, vector2)`

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

`dot(vector1, vector2)`

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

Rotating a vector

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

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

Convenient conversion

For convenience Visual automatically converts (a,b,c) into `vector(a,b,c)`, with floating-point values, when creating Visual objects: `sphere.pos=(1,2,3)` is equivalent to `sphere.pos=vector(1.,2.,3.)`. However, using the form (a,b,c) directly in computations will give errors.

Example of Vector Computations

Here is an example of user attributes (“mass” and momentum “p”) added to display objects, with vector processing:

```
from visual import *

G = 6.7e-11                # gravitational constant
au = 1.5e11                # astronomical unit (sun-earth distance)
year = 365*24*60*60       # seconds in a year

# Use large radii in order to be able to see objects:
sun = sphere( pos=(0,0,0), radius=10*7e8, mass=2e30,
              color=(1,1,0) )
earth = sphere( pos=(au,0,0), radius=400*6.4e6,
               mass=6e24, color=(0,1,1) )

# Give the earth some momentum p:
earth.p = vector(0, earth.mass * pi * au / year, 0)

dt = 1e4                  # time interval per step
scene.autoscale = 0      # turn off autoscaling of display

while 1:
    r = mag( sun.pos - earth.pos )
    F = G*sun.mass*earth.mass*(sun.pos-earth.pos)/r**3
    earth.p = earth.p + F*dt
    earth.pos = earth.pos + (earth.p/earth.mass)*dt
    rate(100)             # compute 100 steps every second
```

Controlling One or More Visual Display Windows

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

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

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

General-purpose options

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

`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.

Controlling the window

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

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

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

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

`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).

`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 $\pi/3.0$ (60 degrees).
range	The extent of the region of interest away from center along each axis. This is always $1.0/\text{scale}$, so use either range or scale depending on which makes the most sense in your program. Default (10,10,10) or set by autoscale.
scale	A scaling factor which scales the region of interest into the sphere with unit radius. This is always $1.0/\text{range}$, so use either range or scale depending on which makes the most sense in your program. Default (0.1,0.1,0.1) or set by autoscale.
uniform	0 = each axis has different units and scales autoscale will scale axes independently the x and y axes will be scaled by the aspect ratio of the window 1 = each axis has the same scale autoscale scales axes together the aspect ratio of the window does not affect scaling
up	A vector representing world-space up. This vector will always project to a vertical line on the screen (think of the camera as having a “plumb bob” that keeps the top of the screen oriented toward up). The camera also rotates around this axis when the user rotates “horizontally”. By default the y axis is the up vector. There is an interaction between up and forward, the direction that the camera is pointing. By default, the camera points in the $-z$ direction (0,0,-1). In this case, you can make the x or y axes (or anything between) be the up vector, but you cannot make the z axis be the up vector, because this is the axis about which the camera rotates when you set the up attribute. If you want the z axis to point up, first set forward to something other than the $-z$ axis, for example (1,0,0).
autoscale	0 = no automatic scaling (set range or scale explicitly) 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 1 = user can zoom (default)
userspin	0 = user cannot rotate the scene 1 = user can rotate (default)

Mouse Objects

Introduction

Mouse objects are never created by the program; they are always obtained from the **mouse** attribute of a **display** object such as **scene**. For example, to obtain mouse input in the default window created by Visual, refer to **scene.mouse**.

A mouse object has a group of attributes corresponding to the current state of the mouse. It also has a function `getclick()`, which returns an object with similar attributes corresponding to the state of the mouse when the user last clicked. If the user has not already clicked the mouse, your program will stop executing until this happens.

Current state of mouse

<code>pos</code>	The current 3D position of the mouse cursor; <code>scene.mouse.pos</code> . Visual always chooses a point in the plane parallel to the screen and passing through <code>display.center</code> .
<code>pick</code>	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 <code>scene.mouse.pick</code> .
<code>pickpos</code>	The 3D point on the surface of the picked object which falls under the cursor, or None; <code>scene.mouse.pickpos</code> .
<code>camera</code>	The current position of the camera; e.g. <code>scene.mouse.camera</code> . The camera and ray attributes together define all of the 3D points under the mouse cursor.
<code>ray</code>	A unit vector pointing from camera in the direction of the mouse cursor. The points under the mouse cursor are exactly $\{ \text{camera} + t * \text{ray} \text{ for } t > 0 \}$.
<code>project()</code>	See the later section on “Alternative to camera and ray” for projecting mouse information onto a given plane.

Getting clicks

<code>clicked</code>	The number of clicks which have been queued; e.g. <code>scene.mouse.clicked</code> . <code>scene.mouse.clicked = 0</code> may be used to discard input. No value other than zero can be assigned.
<code>getclick()</code>	Removes a click from the input queue; <code>scene.mouse.getclick()</code> . If no clicks are waiting in the queue (that is, if <code>scene.mouse.clicked</code> is zero), <code>getclick()</code> waits until the user clicks. <code>getclick()</code> returns an object with attributes similar to the mouse: <i>pos</i> , <i>pick</i> , <i>pickpos</i> , <i>camera</i> , and <i>ray</i> . These attributes correspond to the state of the mouse when the click took place. For example, <code>scene.mouse.getclick().pos</code>

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.

Alternative to camera and ray

While `scene.mouse.camera` and `scene.mouse.ray` are powerful, they can be hard to use. Here is an alternative way to use mouse information:

```
temp = scene.mouse.project(normal=(0,1,0), d=0)
if temp <> None:
    ball.pos = temp
```

This projects the mouse cursor onto a plane that is a distance `d` from the origin of the scene (0,0,0); the plane is perpendicular to the specified normal. If `d` is not specified, its default value is zero and the plane passes through the origin. It returns a 3D position, or `None` if the projection of the mouse misses the plane.

For example, if you want the user of your program to be able to use the mouse to place balls in the `xy` plane, no matter how the user has rotated the point of view, you would use `temp = scene.mouse.project(normal=(0,0,1))`.

Mouse example

This program displays a sphere (which automatically creates a window referred to as `scene`), then repeatedly waits for a mouse click, prints the mouse position, and displays a small red sphere:

```
sphere()          # display a white sphere for context
while 1:
    if scene.mouse.clicked:
        m = scene.mouse.getclick()
        loc = m.pos
        print loc
        sphere(pos=loc, radius=0.1, color=(1,0,0))
```

Try running this program. A mouse click is defined as pressing and releasing the mouse button at the same location.

You will find that if you click inside the large white sphere, nothing seems to happen. This is because the mouse click is in the `x,y` plane, so the little red sphere is buried inside the large white sphere. If you rotate the scene and then click, you'll see that the little red spheres go into the new plane parallel to the screen and passing through `display.center`. If you want all the red spheres to go into the `xy` plane, do this:

```
loc = m.project(normal=(0,0,1))
if loc <> None:
    print loc
    sphere(pos=loc, radius=0.1, color=(1,0,0))
```

Graph Plotting in VPython

In this section we describe features for plotting graphs with tick marks and labels.

A simple plot

Here is a simple example of how to plot a graph:

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

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

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

Importing from `visual.graph` makes available all of `visual` 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 (`gdot`), vertical bars (`gvbar`), horizontal bars (`ghbar`), and binned data displayed as vertical bars (`ghistogram`; see later discussion). When creating one of these objects, you can specify a `color` attribute. For `gvbar` and `ghbar` you can also specify a `delta` attribute, which specifies the width of the bar (the default is `delta=1.`).

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

```
funct1 = gcurve(color=color.cyan)
funct2 = gvbar(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`, `gdot`, `gvbar`, or `ghbar` object, you can provide a list of points to be plotted, just as is the case with the ordinary curve object:

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

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)
```

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

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

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

Histograms (sorted, binned data)

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

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

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

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

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

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

the new data are added to the old data.

If you say the following,

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

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

gdisplay vs. display

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

This description of the Visual 3D graphics facility was produced by Ruth Chabay, David Scherer, and Bruce Sherwood, of Carnegie Mellon University.