Engineering at Alberta

# **VPython** – Visual Python

Weichen Qiu

*July 2020*

## Contents

## Introduction

VPython is an easy-to-use, powerful environment for creating 3D animations and publishing them on the web. It has a multitude of built-in objects and shapes, as well as, graphing capabilities.

Interactive programs can be built using widgets. The process to make the program interactive starts with first creating the objects and assigning variables to it, then creating a widget that the user can interact with (ie. slider,textbox…etc.). Immediately above that, declaring a new function for the widget to *bind* to that will process the user's input and act on the objects.

This document starts with an introduction on how to create 3D and 2D objects. Followed by all the parameters that can be applied to these objects and how to modify these parameters. Widgets are then introduced that allow user interactions. After that, how to make graphs and the different types of graphs are covered. Finally, animations are explained which rely on modifying parameters and can be integrated with graphs.

## Starting a New Program

Go to https://www.glowscript.org/ and sign in using your Google account.

Once signed in, go to your programs by clicking here.

Click **Create New Program** and enter a program name.

A blank text editor should now open.

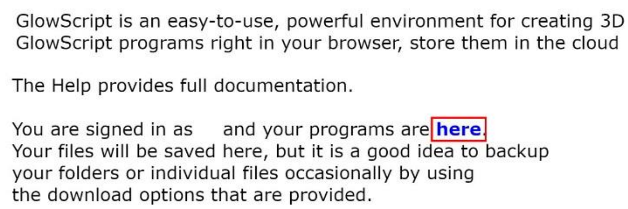



**Figure 1 Press here to go to MyPrograms**


**Figure 2 Press Create New Program**

## Basics

See VPython Documentation here.

Please note Python is **case** sensitive and **indent** sensitive.

Pressing **Run this program** ( or ctrl + 1 ) will run the code and pressing **Edit this program** will return to the code editor.

VPython uses `vector(x,y,z)` as vectors to specify x,y,z values in 3D space such as position and size, it is similar to a list of 3 values in python `[x,y,z]`.

To print to the output window, use `print()`, this will be useful in debugging code.

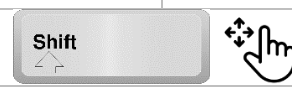
## Moving in 3D space

To rotate in 3D space, **right-click** and drag or hold down **ctrl** and drag.

To zoom in and out, **scroll** in and out or hold down **alt** and drag.

To pan, hold down **shift** and drag.

**Table 1 Movement controls**

| | | |
|---|---|---|
| Rotate |  |  Ctrl |
| Zoom |  |  Alt |
| Pan |  Shift | |

## Shapes

### Box

To create a simple box, type `box()` and run the program.

The `box()` function can take parameters to modify the box.

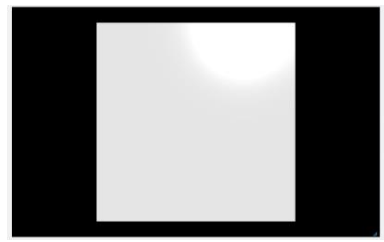


**Figure 3 box()**

#### Position

Use the **pos** parameter to <u>position</u> the box. `box(pos=vector(1,2,3))` will position the box at $x = 1,\ y = 2,\ z = 3$.

#### Size

Use the **size** parameter to <u>size</u> the box, where `size=vector(`*`width,height,depth`*`)`. For instance, `box(pos=vector(1,2,3), size=vector(4,5,6))` will set the $\text{width} = 4, \text{height} = 5, \text{depth} = 6$.

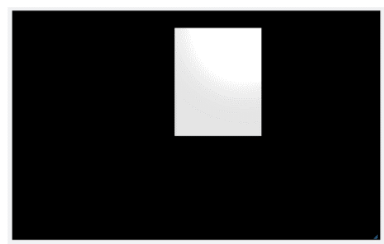


**Figure 4 box(pos=vector(1,2,3), size=vector(4,5,6))**

#### Color

Use the **color** parameter to color the box. `color=color.`*`<colorName>`*, where *colorName* is one of the colors in table 2.

`box(pos=vector(1,2,3), size=vector(4,5,6), color=color.red)` will create a red box at position (1,2,3) and of size (4,5,6).

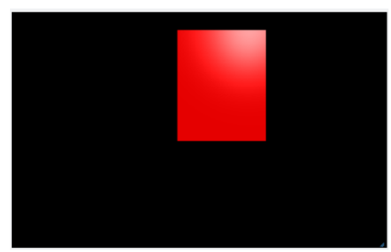


**Figure 5 box(pos=vector(1,2,3),size=vector(4,5,6),color=color.red)**

**Table 2 Colors**

| | |
|---|---|
| | color.red |
| | color.green |
| | color.blue |
| | color.yellow |
| | color.orange |
| | color.cyan |
| | color.magenta |
| | color.black |
| | color.white |

Color can also be represented by the RGB scale, `color=vector(red,green,blue)` where the range is between 0 to 1, for instance, `color=vector(1,0,1)`.

## Sphere

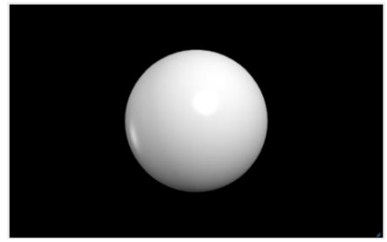


Figure 6 sphere(radius=2)

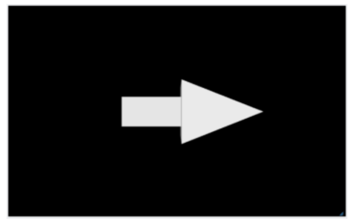
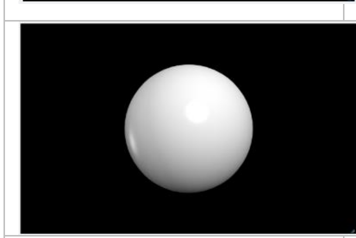
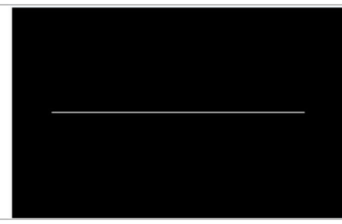
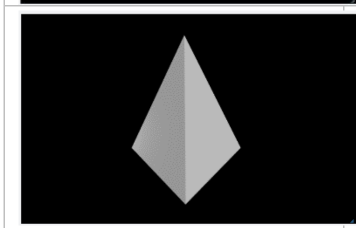
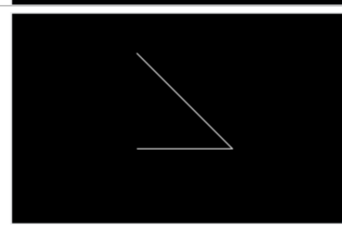
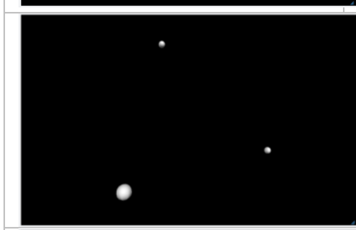
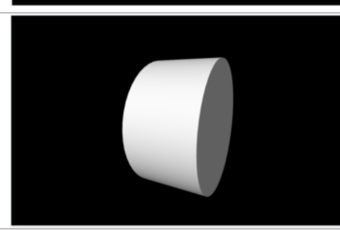
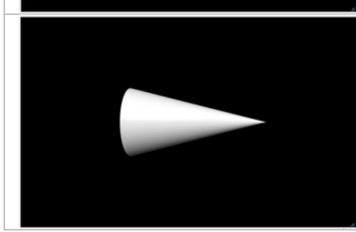
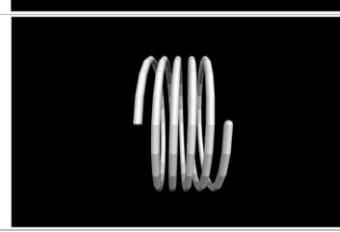
Use `sphere()` to create a sphere, which is similar to box( ) except that size is replaced by **radius**.

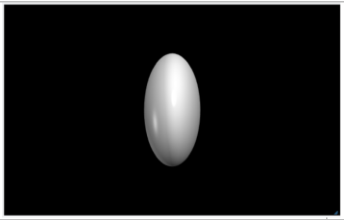
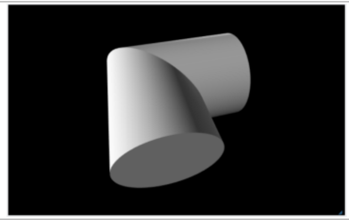
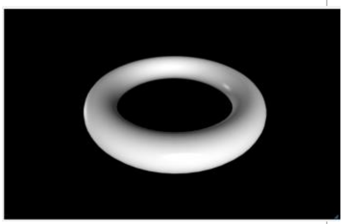
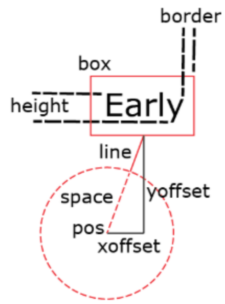
`sphere(radius=2)` creates a sphere of radius of 2.

## 3D Shapes

Documentation for 3D shapes [here]

**Table 3 3D shapes**

| | | | |
|---|---|---|---|
|  | **box** ( size=vector(1,1,1) ) |  | **arrow** (shaftwidth = 1, headwidth = 2, headlength = 3, axis=vector(5,0,0)) |
|  | **sphere** ( radius=1 ) |  | **curve** (pos= [vector(0,0,0), vector(1,0,0)]) |
|  | **pyramid** (axis=vector(0,1,1)) |  | **curve** (pos=[vector(0,0,0), vector(1,0,0), vector(0,1,0)]) |
|  | **points** (pos= [vector(1,0,0), vector(0,1,0), vector(0,0,1)], radius=1) |  | **cylinder** (axis=vector(1,0,0)) |
|  | **cone** (axis=vector(4,0,0) ,radius=1) |  | **helix** (axis=vector(1,0,0)) |

| | | | |
|---|---|---|---|
|  | **ellipsoid** (length=1, height=2, width=3) |  | **extrusion**(path= [vector(0,0,0), vector(0,0,1), vector(1,0,1)] , shape= shapes.circle(radius=0.5)) |
|  | **ring** (axis=vector(0,1,0), radius=0.5, thickness=0.1) |  | **label** (pos=obj.pos, text='Early', xoffset=20, yoffset=50, space=30, height=16, border=4, font='sans') |
|  | **text** (text='text here', align='center') | | Note: label always point forwards |

## 2D Shapes

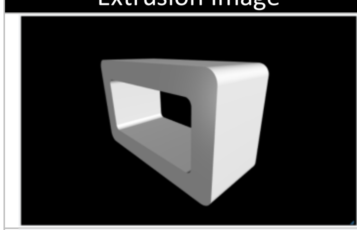
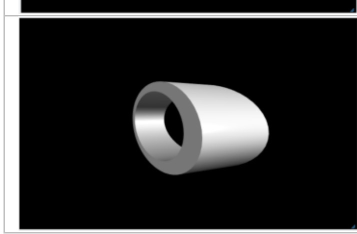
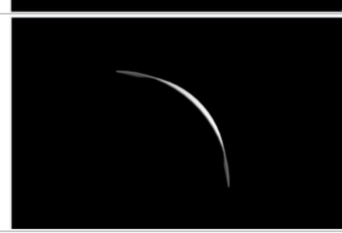
Documentation on 2D shapes here.

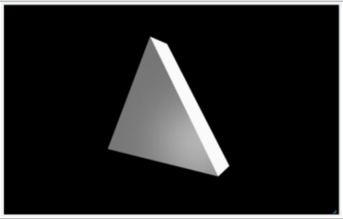
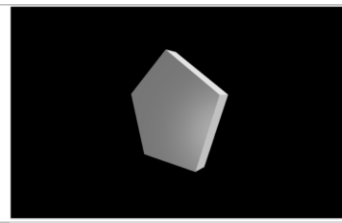
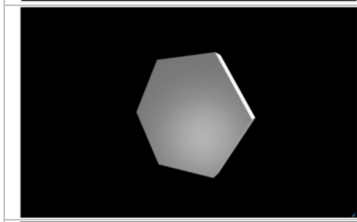
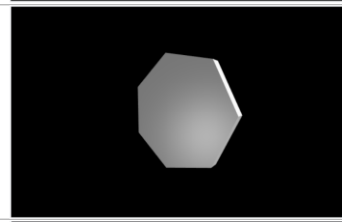
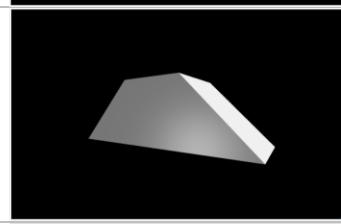
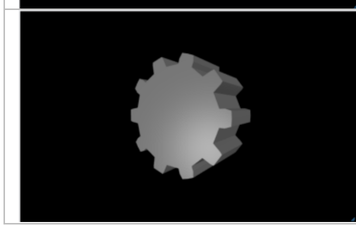
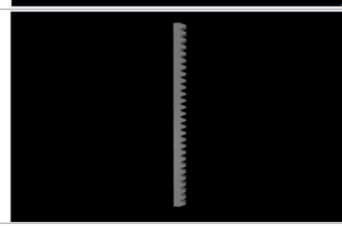
2D shapes are to be used as the `shape=` in 3D extrusions.

```
extrusion(path=[vector(a,b,c),vector(e,f,g)] ,shape=shapes.<2D shape>() )
```

Remove the thickness parameter to eliminate the hole in the shape.

**Table 4 2D Shapes**

| Extrusion Image | *<2D shape>* | Extrusion Image | *<2D shape>* |
|---|---|---|---|
|  | shapes.**rectangle** (width=5, height=3, rotate=pi, roundness=0.1, thickness=0.2) |  | shapes.**circle** (radius=2, thickness=0.2) |
|  | shapes.**ellipse** (width=5, height=3, thickness=0.2) |  | shapes.**arc** (radius=2, angle1=0, angle2=pi/2) |

| | | | |
|---|---|---|---|
|  | shapes.**triangle**<br>(length=5) |  | shapes.**pentagon**<br>(length=5) |
|  | shapes.**hexagon**<br>(length=5) |  | shapes.**ngon**<br>(np=7,<br>length=5) |
|  | shapes.**star**<br>(n=5) |  | shapes.**trapezoid**<br>(width=3,<br>top=1,<br>height=1) |
|  | shapes.**gear**<br>(n=10) |  | shapes.**rackgear**() |

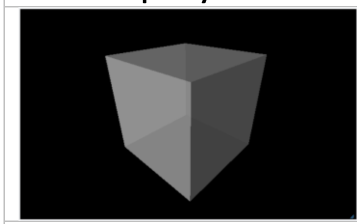
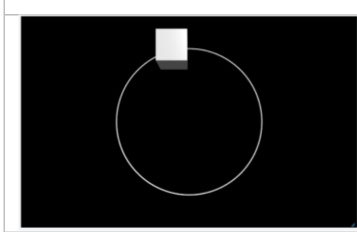
# Parameters

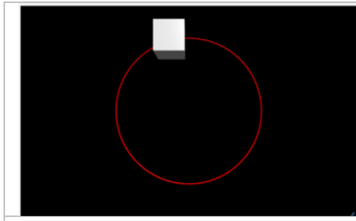
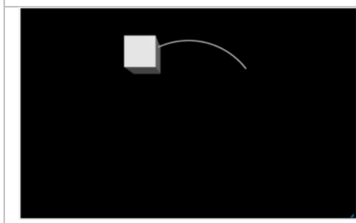
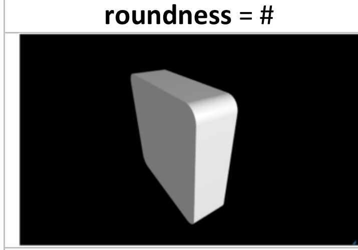
Parameters for 2D shapes documentation here.

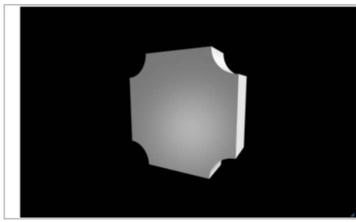
In the table below, “#” is used as a placeholder for a *number*.

**Table 5 Parameters**

| Code | Parameter Name | Info |
|---|---|---|
| **pos** = vector ( #,#,# ) | Position | x,y,z positions of an object in 3D space |
| **pos** = [vector ( #,#,# ),<br>vector ( #,#,# ) , …] | Position List | List of points for making lines, curves, and points |
| **size** = vector ( #,#,# ) | Size | Width, height, depth of the box |
| **radius** = # | Radius | The radius of circular objects |
| **color** = color.<color><br>or<br>**color** = vector ( #,#,# ) | Color | Color of shape. Insert name of color at <color> |
| <object>.**rotate**<br>(axis=vector(#,#,#), angle=#) | Rotate <object> by an angle along the axis | Rotate a 3D object. Specify the axis to rotate along and the angle to rotate by |
| **visible** = True/False | Visible | Turn object on or off<br>(visible or not visible) |

| | | |
|---|---|---|
| **axis** = vector ( #,#,# ) | Axis<br>(For pointy shapes) | Direction and length to point at |
| **path** =<br>[vector ( #,#,# ),<br>vector ( #,#,# ),<br>vector ( #,#,# ) , … ] | Path (Extrusion only) | Path that extrusion takes |
| **shape** =<br>shapes.<shape>() | Shape (Extrusion only) | The shape of extrusion’s cross-section |
| **up** = vector( #,#,# ) | Up axis | Direction of object’s “up” as a vector |
| **opacity** = #<br> | Opacity<br>[0,1] | The opacity of an object. Must be between 0 and 1, where 1 is solid and 0 is invisible |
| **texture** = “*<url>/image.jpg*”<br> | Texture | Giving objects a texture. Set texture equal to URL of the image. |
| **shininess** = #<br> | Shininess<br>[0,1] | How reflective or shiny the object is, where 0 is dull and 1 is shiny |
| **make_trail** = True/False<br> | Make Trail | Leaves a trail behind moving object |
| **trail_color** = color.<color><br>or<br>**trail_color** = vector ( #,#,# ) | Trail Color<br>( make_trail must be True ) | Color of the trail left behind moving object |

| | | |
|---|---|---|
|  | | |
| **retain** = #  | Retain number of points in the trail<br>( make_trail must be True ) | The trail disappears after a specified number of points to retain. Tail-like appearance. |
| | 2D shapes only | |
| **length** = # | Length | Length of 2D shapes (pentagon, hexagon, ngon) |
| **width** = # | Width | Width of 2D shapes (rectangle, ellipse) |
| **height** = # | Height | Height of 2D shapes (rectangle, ellipse) |
| **rotate** = # | Rotate | The rotational angle in *radians*<br>(+) is CCW and (-) is CW |
| **np** = # | Number of sides | Number of sides of a polygon |
| **n** = # | Number of beams | Number of outward-going beams on a star or gear |
| **scale** = # | Scale | Resize object in both x and y-direction |
| **xscale** = # | X-scale | Scale in x-direction |
| **yscale** = # | Y-scale | Scale in y-direction |
| **thickness** = #  | Thickness | Used to create hollow objects |
| **roundness** = #  | Roundness | Round sharp corners |
| **invert** = True/False | Invert rounding | |

| | | |
|---|---|---|
|  | ( roundness ≠ 0 ) | Used with roundness, a circular chamfer is created, instead of rounded corner |

## Changing Parameters using Variables

These parameters can be changed by assigning the object to a variable.

### For 3D objects:

```
<variable> = <object>(parameter=value)

Ex. my_box = box(pos=vector(0,0,0), color=color.red, opacity=1)
```

These parameters can be changed by first calling it using the variable and setting it equal to a new value.

```
<variable>.<parameter> = new_value

Ex. my_box.pos = vector(1,2,3)          # change position to (1,2,3)

    my_box.color = color.blue           # change color to blue

    my_box.opacity = 0.5                # change opacity to 50%
```

For parameters equal to a vector, such as pos and size, each individual number in the vector can be changed as follows:

```
<variable>.<parameter>.x/y/z = new_value

Ex. my_box.pos.x = 1                    # change x position to 1

    my_box.pos.y = 2                    # change y position to 2

    my_box.pos.z = 3                    # change z position to 3
```

Where `x/y/z` corresponds to the x,y,z values in `vector(x,y,z)`.

3D objects can be made to **rotate** as follows:

```
<variable>.rotate(axis=vector(#,#,#), angle=#)

Ex. my_box.rotate(axis=vector(1,0,0), angle=pi)  # rotate by pi radians
```

Rotate has an optional `origin` attribute to rotate about an axis relative to an origin:

```
my_box.rotate(axis=vector(1,0,0), angle=pi, origin=vector(1,1,1))
```

For 2D objects:

```
<variable> = extrusion(path=[vector(a,b,c),vector(e,f,g)],

                       shape=shapes.<2D shape>())
```

The 2D shape can be rotated as in the following example:

```
my_circle = extrusion(path=[vector(0,0,0),vector(1,1,1)],
                     shape=shapes.circle(radius=pi))
my_circle.rotate(axis=vector(1,0,0), angle=pi) # rotate by pi along axis
```

## Widgets

Interactive Widgets Documentation [here]

VPython has a number of built-in widgets that allow users to control object parameters.

Widgets are displayed under the model in the order that it appears in the code, use **`scene.append_to_caption('\n\n')`** to add spaces between widgets.

Use `<variable>.delete()`, where `<variable> = <widget>(bind=function)` to delete it.

### Bind Parameter

All widgets have the **bind** parameter that assigns it to a *function* that is called when clicked.

First, define the python function that will control the object parameters using input from the widget.

```
def f(x):                          # define the function
     actions here
```

The keyword **def** declares a function in python, and the input parameter, **x** refers to the widget. Therefore, to access parameters from the widget use `x.<parameter>`.

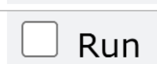
Note that name of the function does *not* have to be `f` and the parameter name does *not* have to be `x`.

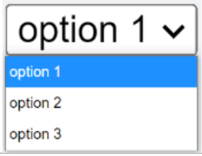
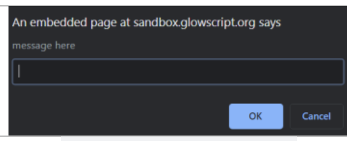
Next, select the widget to use and bind it to the function using the **bind** parameter.

```
<widget>(bind=f)                   # bind widget to function
```

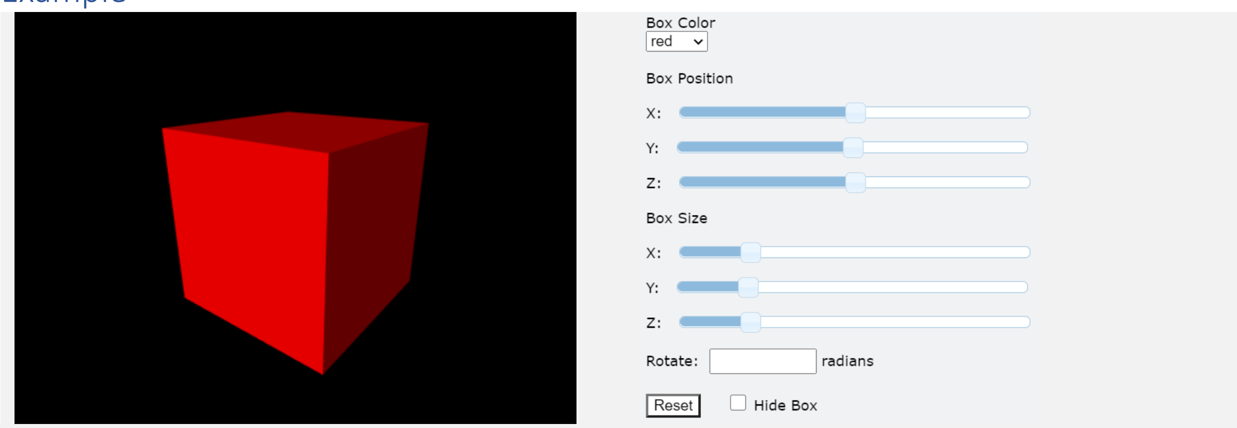
*Note. All widget parameters are optional except for *bind.*

**Table 6 Widgets**

| Widget | | Function Code | Widget Code |
|---|---|---|---|
| Button |   | def f(x):<br>*action* | **button**(bind=f,<br>text="Run",<br>color=color.green,<br>background=color.red) |
| Radio |   | def f(x):<br>print (x.**checked**) | **radio** (bind=f,<br>text="Run") |
| CheckBox |   | def f(x):<br>print (x.**checked**) | **checkbox**(bind=f,<br>text="Run") |
| Slider |  | def f(x):<br>print (x.**value**) | **slider**(bind=f,<br>vertical=False,<br>min=0, max=10,step=1,<br>value=5,<br>length=200,width=10) |

| | | | |
|---|---|---|---|
| Menu |  | def f(x):<br>print (x.**selected**)<br>print (x.**index**) | **menu**(bind=f,<br>choices=['option 1',<br>'option 2','option 3']) |
| Number Input |  | def f(x):<br>print (x.**number**) | **winput**(bind=f,<br>prompt="Enter here",<br>type="*numeric*",<br>width=100, height=20) |
| Text Input | | def f(x):<br>print (x.**text**) | **winput**(bind=f,<br>prompt="Enter here",<br>type="*string*",<br>width=100, height=20) |
| Pop up Window |  | print (x) | x = **input**("message here") |
| Plain Text | text here | | **wtext**(text="text here") |

## Example




```
my_box = box(pos=vector(0,0,0), size=vector(1,1,1), color=color.red)

# Dropdown Menu - Color Selection
scene.append_to_caption('Box Color\n')
def Menu(m):
    if m.selected == "red":
        my_box.color = color.red
    elif m.selected == "blue":
        my_box.color = color.blue
    else:
        my_box.color = color.green
menu( choices=['red' ,'blue', 'green'], bind=Menu )
scene.append_to_caption('\n\n')

# Sliders - Box Position
```

```
scene.append_to_caption('Box Position\n\n')
def Pos_x(x):
    my_box.pos.x = x.value
scene.append_to_caption('X: ')
slider( bind=Pos_x, min=-1, max=1, step=0.001, value=0)
scene.append_to_caption('\n\n')

def Pos_y(y):
    my_box.pos.y = y.value
scene.append_to_caption('Y: ')
slider( bind=Pos_y ,min=-1, max=1, step=0.001, value=0)
scene.append_to_caption('\n\n')

def Pos_z(z):
    my_box.pos.z = z.value
scene.append_to_caption('Z: ')
slider( bind=Pos_z ,min=-1, max=1, step=0.001, value=0)
scene.append_to_caption('\n\n')

# Sliders – Box Size
scene.append_to_caption('Box Size\n\n')
def Size_x(x):
    my_box.size.x = x.value
scene.append_to_caption('X: ')
slider( bind=Size_x, min=0, max=5, step=0.001, value=1)
scene.append_to_caption('\n\n')

def Size_y(y):
    my_box.size.y = y.value
scene.append_to_caption('Y: ')
slider( bind=Size_y ,min=0, max=5, step=0.001, value=1)
scene.append_to_caption('\n\n')

def Size_z(z):
    my_box.size.z = z.value
scene.append_to_caption('Z: ')
slider( bind=Size_z ,min=0, max=5, step=0.001, value=1)
scene.append_to_caption('\n\n')

# Number input - Rotate
def Rotate(x):
    my_box.rotate(axis=vector(1,0,0),angle=x.number)
winput(bind=Rotate, prompt="Rotate: ",type="numeric")
wtext(text=" radians")
scene.append_to_caption('\n\n')

# Button - Reset
def Reset():
    my_box.pos=vector(0,0,0)
    my_box.size=vector(1,1,1)
    my_box.color=color.red
```

```
button(bind=Reset, text="Reset")

scene.append_to_caption('      ')

# Checkbox - Visible
def Checkbox(c):
    my_box.visible = not c.checked # alternates
checkbox(bind=Checkbox, text='Hide Box') # text to right of checkbox
```

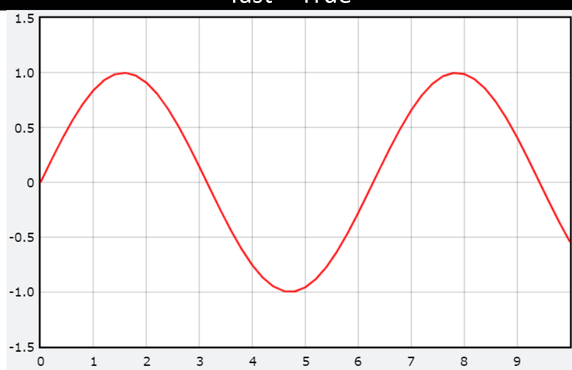
## Graphs

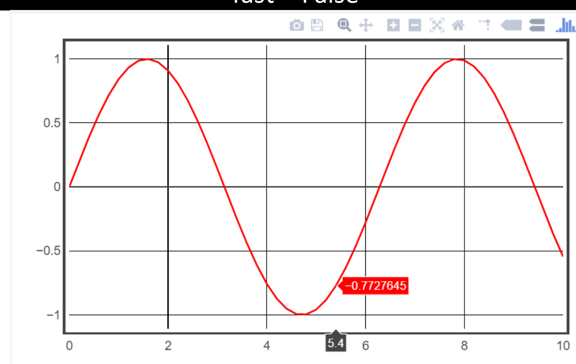
Graph Documentation here.

There are 2 graph options: (default is `fast=True`)

1. `fast=False` based on Plotly with interactive capabilities such as panning and zooming
2. `fast=True` based on Flot and is not interactive

| fast = True | fast = False |
|---|---|




First define the graph window and specify the graph size, title, and labels.

```
g = graph (width=600,height=400,title='title here',xtitle='x',ytitle='y',
           foreground=color.black, background=color.white,   # optional
           xmin=0, xmax=10, ymin=-15, ymax=15,fast=False)    # optional
```

Then declare the type of plot and set the graph parameter equal to the graph `g` declared above.

```
plot1 = <type>(graph=g, color=color.red)
```

The type of graph `<type>` is either `gcurve`, `gdots`, `gvbars` or `ghbars`

There are 3 ways to add data to the graph:

1. Declare the data in the plot declaration as a list of data points `[x,y]`
   a. `plot1 = <type>(graph=g,color=color.red,data=[[1,2],[3,4],… ])`
2. Use the plot function
   a. `plot1.plot([1,2])          # adding a single point to plot`

```
b. plot1.plot([1,2],[3,4]…)      # adding multiple points to plot
```

3. Use plot in a for loop with a function

```
a. for x in range(10):
       plot1.plot(x, sin(x))     # plot points(x,y),where y=sin(x)
```

The plot's data can be accessed by `plot1.data`.
To change the entire dataset use `plot1.data=[[#,#],[#,#]…]`.

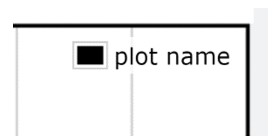
## Multiple Plots

To add multiple plots to the same graph, create a new plot with the same *`graph=`* parameter

```
plot2 = <type>(graph=g,color=color.green,data=[[10,11],[12,13],… ])
```

Note that the type of plots do *not* have to be the same.

**Figure 7 Graph Legend**




## Legend

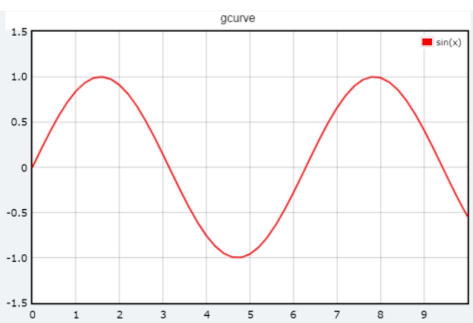
To add a legend, use the `label=` parameter when making a new plot

```
plot2 = <type>(graph=g,data=[[1,2],[3,4],… ],label="plot name")
```

Aside on **for loops**:

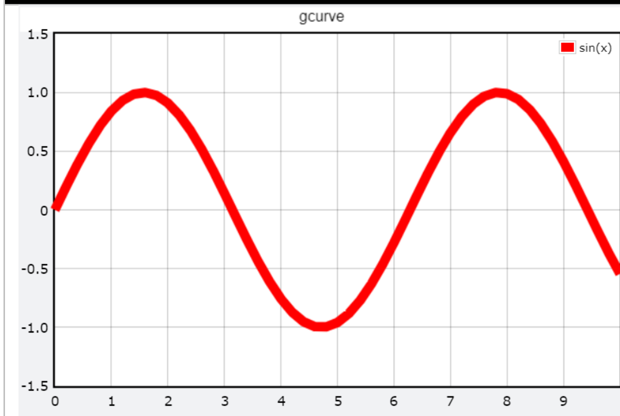
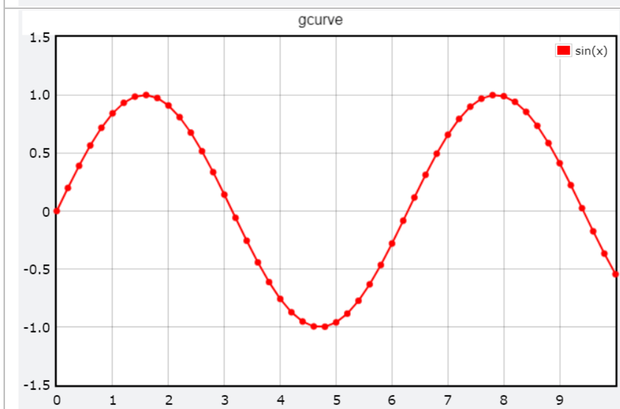
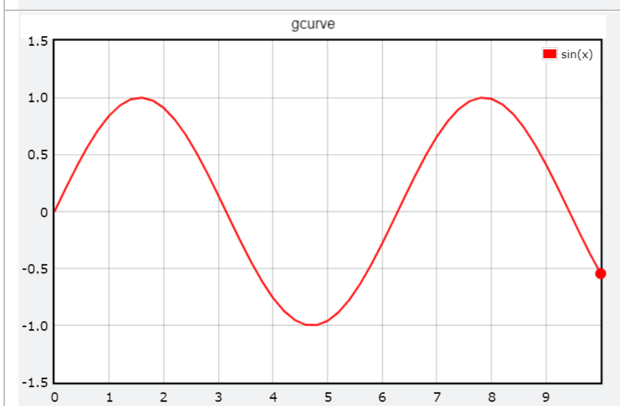
```
for i in range(start,stop):          for i in range(start, stop, step size):
for i in range(0,10):                for i in range(0,10,2):
 - Default is step size of 1           - Specify the step size
```

## gcurve – Line Plot

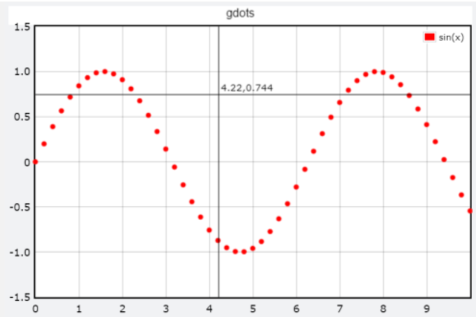



```
g = graph(title='gcruve graph')
plot1 = gcurve(graph=g, color=color.red, label="sin(x)")
# range is list of numbers from 0 to 10 with step of 0.2
for x in range(0,10,0.2):
    plot1.plot(x, sin(x))
```

**Table 7 GCurve Parameters**

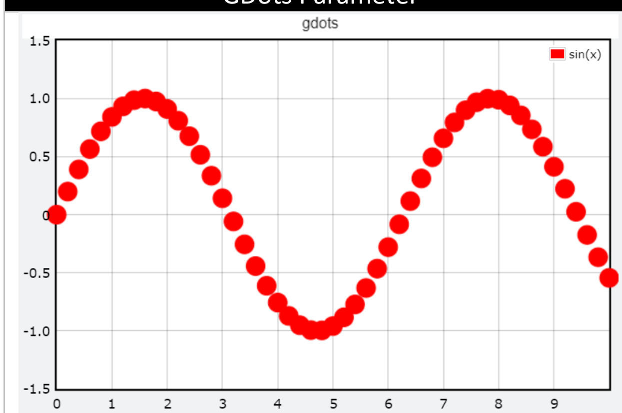
<table>
<tr><th>GCurve Parameter</th><th colspan="3">Code</th></tr>
<tr><td rowspan="2">
</td><td colspan="3">Line Width</td></tr>
<tr><td colspan="3">width = #</td></tr>
<tr><td rowspan="4">
</td><td colspan="3">Markers</td></tr>
<tr><td colspan="3">markers = True</td></tr>
<tr><td rowspan="2">Optional Marker Parameters</td><td>radius = #</td><td>Radius of marker</td></tr>
<tr><td>marker_color = color.<color></td><td>Marker color</td></tr>
<tr><td rowspan="4">
</td><td colspan="3">Single Dot at current plotting point</td></tr>
<tr><td colspan="3">dot = True</td></tr>
<tr><td rowspan="2">Optional Dot Parameters</td><td>dot_radius = #</td><td>Dot radius</td></tr>
<tr><td>dot_color = color.<color></td><td>Dot color</td></tr>
</table>

## gdots – Scatter Plot

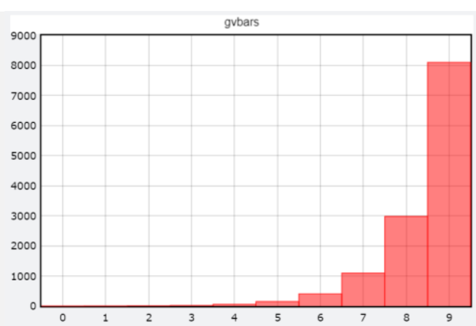



```
g = graph(title='gdots')
plot1 = gdots(graph=g, color=color.red, label="sin(x)")
for x in range(0,10,0.2):
    plot1.plot(x, sin(x))
```

**Table 8 GDots Parameters**

| GDots Parameter | Code |
| --- | --- |
|   | **Dot Radius** |
| | radius = # |

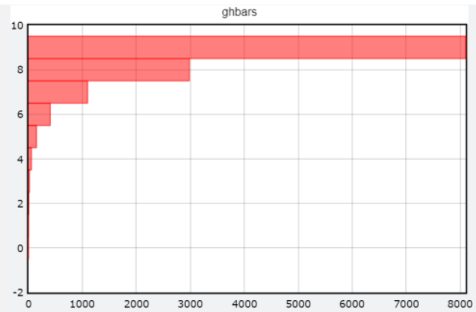
## gvbars – Vertical Bars



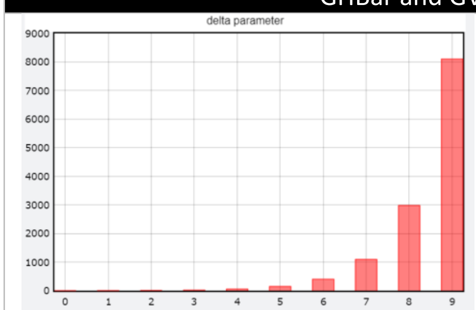
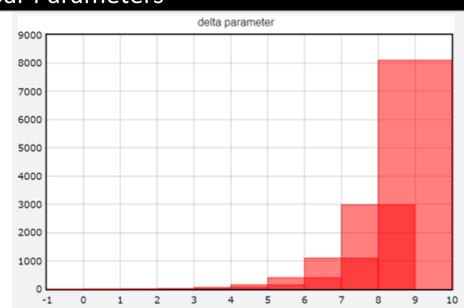

```
g = graph(title='gvbars')
plot1 = gvbars(graph=g, color=color.red)
```

```
for x in range(10):
    plot1.plot(x,exp(x))
```

## ghbars – Horizontal Bars

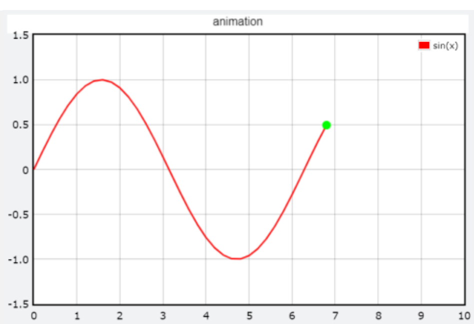



```
g = graph(title='ghbars')
plot1 = ghbars(graph=g, color=color.red)
for y in range(10):
    plot1.plot(exp(y),y)
```

<table>
<tr><th colspan="2">GHBar and GVBar Parameters</th><th>Code</th></tr>
<tr><td rowspan="2"><br>delta = 0.5</td><td rowspan="2"><br>delta = 2</td><td>Bar Width</td></tr>
<tr><td>delta = #</td></tr>
</table>

## Moving Graph

To create a plotting animation, use `rate(#)` to delay the plotting. Also, set `scroll=True` and specify `xmin, xmax` in `graph()`.




```
g = graph(title='animation', scroll=True, xmin=0, xmax=10)
```

```
plot1 = gcurve(graph=g, color=color.red, label="sin(x)",
               dot=True, dot_color=color.green)
for x in range(0,20,0.2):
    rate(10)
    plot1.plot(x, sin(x))
```

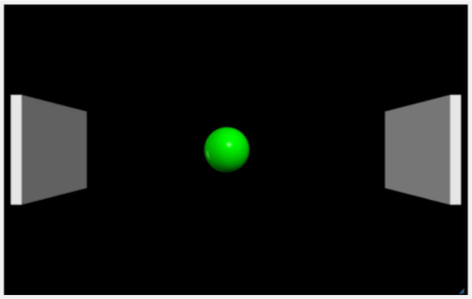
## Animations

As seen in Moving Graphs, the key behind Animations is **`rate(#)`**, which allows an action to take place over a period of time.

Use a `while True:` loop to have the animation play infinitely or a `for i in range(#):` loop to play for a finite time.

Tip: if your program crashes, check that your loop has **`rate(#)`**.

### Bouncing Ball Example



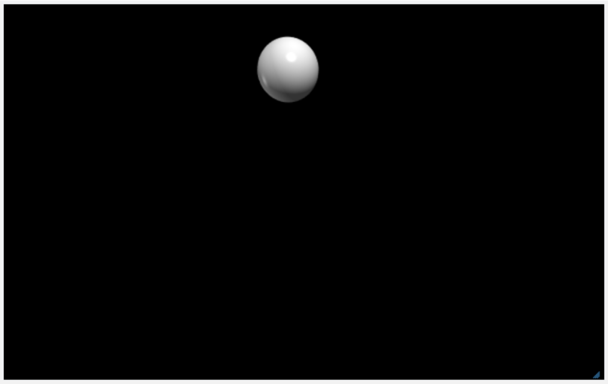
```
# make sphere and 2 wall objects
my_sphere = sphere(pos=vector(0,0,0), radius=0.25, color=color.green )
wall1 = box(pos=vector(2,0,0), size=vector(0.1,1,1), color=color.white)
wall2 = box(pos=vector(-2,0,0), size=vector(0.1,1,1), color=color.white)
# define the inner edge that ball hits
edge1 = wall1.pos.x - wall1.size.x/2
edge2 = wall2.pos.x - wall2.size.x/2
# while loop does the animation
dx = 0.01              # dx is distance ball moves for each loop
while True:
    rate(100)          # delay between each loop
    if (my_sphere.pos.x + my_sphere.radius >= edge1) or
       (my_sphere.pos.x – my_sphere.radius <= edge2):
```

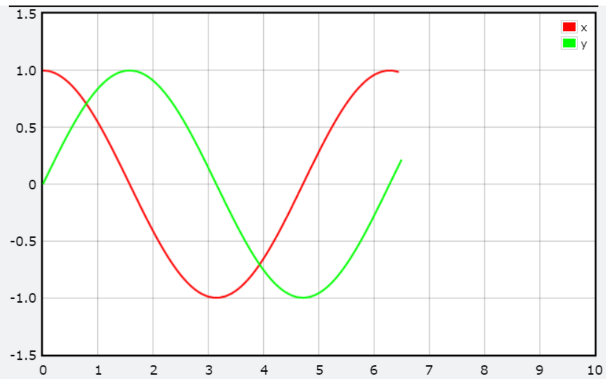
```
            dx = -dx          # if sphere at wall, reverse direction
    my_sphere.pos.x = my_sphere.pos.x+dx  # move sphere's x position by dx
```

To run the animation for a finite time, say 100 loops, replace `while True:` with:

```
for i in range(100):
```

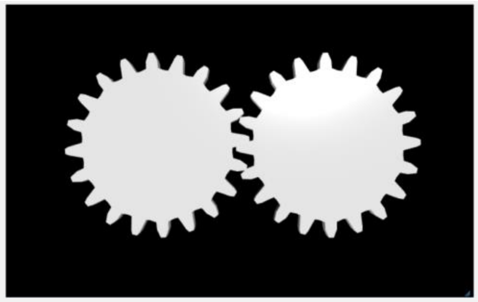
## Animation with Moving Graph Example






```
# create sphere object
my_sphere = sphere(pos=vector(1,0,0), size=vector(0.5,0.5,0.5))
# create graph
g = graph(scroll=True, xmin=0, xmax=10)
x_plot = gcurve(graph=g, color=color.red, label="x")
y_plot = gcurve(graph=g, color=color.green, label="y")
# animation
time = 0
while True:
    rate(100)
    # move sphere x an y using cos and sin
    my_sphere.pos.x = cos(time)
    my_sphere.pos.y = sin(time)
    # plot new x and y
    x_plot.plot(time,cos(time))
    y_plot.plot(time,sin(time))
    time += 0.01                     # increase time by 0.01
```

## Rotating Gear Example – Animating 2D Shapes



```
# create gear shape
g = shapes.gear(n=20)
gear1 = extrusion(path=[vector(0,0,0), vector(0,0,0.1)],
                  shape=g, pos=vector(2,0,0))
gear2 = extrusion(path=[vector(0,0,0), vector(0,0,0.1)],
                  shape=g, pos=vector(0,0,0))
# rotate gear animation
gear1.rotate(axis=vector(0,0,1), angle=0.1)
while True:
    rate(10)
    gear1.rotate(axis=vector(0,0,1), angle=0.05)   # rotate CCW
    gear2.rotate(axis=vector(0,0,1), angle=-0.05)  # rotate CW
```

# Math Functions

Built in math functions documentation here

| | | | | | | |
|---|---|---|---|---|---|---|
| | **abs(x)** | **sqrt(x)** | **sin(x)** | **atan(x)** | **cos(x)** | **min(x,y,z)** |
| **atan2(y,x)** | **sqrt(x)** | **exp(x)** | **log(x)** | **pow(x,y)** | **pi** | **min(a,b,c,..)** |
| **sign(x)** | **round(x)** | **tan(x)** | **ceil(x)** | **random()** | **factorial(x)** | **max(x,y,z)** |
| **acos(x)** | **asin(x)** | **floor(x)** | **degrees()** | **radians()** | **combin(x,y)** | **max(a,b,c,..)** |

Vector Operations documentation here. Let `a` and `b` be 2 different vectors.

| | | |
|---|---|---|
| | $\boldsymbol{a} \cdot \boldsymbol{b}$<br>`dot(a,b) = a.dot(b)` | *scalar project a along b*<br>`comp(a,b) = a.comp(b) = dot(a,norm(b))` |
| $\lvert\boldsymbol{a}\rvert$<br>`mag(a) = a.mag` | $\boldsymbol{a} \times \boldsymbol{b}$<br>`cross(a,b) = a.cross(b)` | $\boldsymbol{a} == \boldsymbol{b}$<br>`a.equals(b)` |
| $\lvert\boldsymbol{a}\rvert^2$<br>`mag2(a) = a.mag2` | $\boldsymbol{a}\angle\boldsymbol{b}$<br>`diff_angle(a,b) = a.diff_angle(b)` | `vector.random()` |
| $\frac{\boldsymbol{a}}{\lvert\boldsymbol{a}\rvert}$<br>`norm(a) = a.norm()` | *vector project a along n*<br>`proj(a,b) = a.proj(b) = dot(a,norm(b))*norm(b)` | `a.x = a['x']`<br>`a.y = a['y']`<br>`a.z = a['z']` |

## Mouse and Keyboard Events

VPython can take mouse and keyboard inputs.

Mouse events documentation here. Keyboard events documentation here.

**Table 9 Events**

| | | |
|---|---|---|
| Keyboard events | keydown | Pressing key event |
| | keyup | Releasing key event |
| Keyboard function | keysdown() | List of all keys currently pressed |
| Keyboard event results | ev.event | Name of the event |
| | ev.key | String name of the pressed key |
| | ev.which | Numerical code of the pressed key |
| Mouse events | click | On mouse click |
| | mousedown | Pressing down mouse event |
| | mouseup | Releasing mouse event |
| | mousemove | Mouse moving event |
| | mouseenter | Mouse enters canvas event |
| | mouseleave | Mouse leaves canvas event |
| Mouse event results | ev.event | Name of the event |
| | ev.pos | Position of the event |
| Scene.mouse | scene.mouse.**pos** | Current 3D position of mouse |
| | obj = scene.mouse.**pick** | The object pointed to by the mouse |

Events are actions that a user can take using the mouse or keyboard.

Note that `<event>` below can be one event or a list of events separated by spaces.

There are 2 ways to listen for user events:

1. `ev = scene.waitfor (' <event> ')`

This method will only wait for the event *once*.

```
box()

ev = scene.waitfor('click')

print(ev.event, ev.pos)
```

Use a `while True`: loop to listen to event(s) more than once:

```
box()
while True:
    ev = scene.waitfor('keydown')
    print(ev.key, ev.which)        # name and numeric code of key
    print(keysdown())              # list of all pressed keys
```

2. `scene.`**`bind`** `(' `*`<event>`*` ', `*`function`*`)` (recommended)

This method binds the scene (canvas) to the callback function and will execute the function on the event. The function should take `ev` as input.

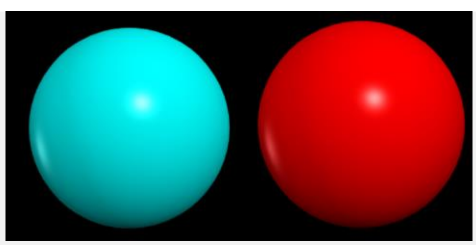
```
box()
def callback(ev):
    print(ev.event)   # name of event
# call function on any one of these events
scene.bind('mouseup mousedown',callback)
```

## Picking an Object

Use `obj = scene.mouse.pick` to get object that mouse is pointing to. Obj=None if no object selected.

```
b = box()
def callback(ev):
    obj = scene.mouse.pick             # pick the object
    if obj != None:                    # check obj is not None
        obj.pos = scene.mouse.pos      # set obj to mouse position
scene.bind('click',callback)   # call function on any one of these events
```

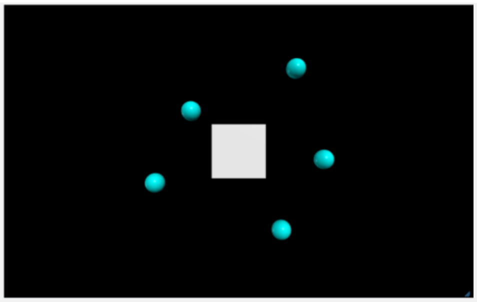
## Changing Colors Example



```
s = sphere(color=color.cyan)
def change():
    if s.color.equals(color.cyan):
        s.color = color.red
```

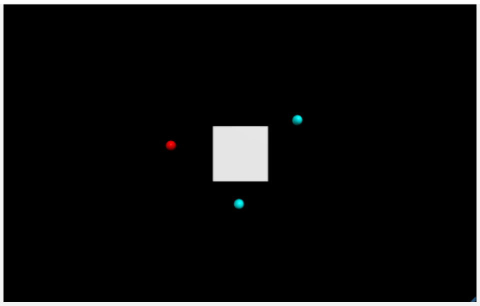
```
    else:
        s.color = color.cyan
scene.bind('click', change)
```

## Create Sphere on Click Example



```
scene.range = 3
box()
def createSphere(ev):
    loc = ev.pos
    sphere(pos=loc, radius=0.2, color=color.cyan)
scene.bind('click', createSphere)
```

## Create and Drag Sphere Example



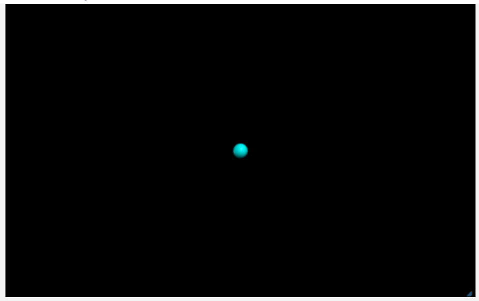
```
scene.range = 3
box()
drag = False
s = None # declare s to be used below
def down():
    global drag, s
    s = sphere(pos=scene.mouse.pos,
        color=color.red,
```

```
        size=0.2*vec(1,1,1))
    drag = True
def move():
    global drag, s
    if drag: # mouse button is down
        s.pos = scene.mouse.pos
def up():
    global drag, s
    s.color = color.cyan
    drag = False
scene.bind("mousedown", down)
scene.bind("mousemove", move)
scene.bind("mouseup", up)
```

## Typing Text into Label Example


```
prose = label() # initially blank text
def keyInput(evt):
    s = evt.key
    if len(s) == 1: # includes enter ('\n')
        prose.text += s # append new character
    elif s == 'delete' and len(prose.text) > 0:
        prose.text = prose.text[:-1] # erase letter
scene.bind('keydown', keyInput)
```

## Moving Sphere Using Arrow Keys



```
scene.range = 20
ball = sphere(color=color.cyan)
v = vec(0,0,0)
dv = 0.2
dt = 0.01
while True:
    rate(30)
    k = keysdown() # a list of keys that are down
    if 'left' in k: v.x -= dv
    if 'right' in k: v.x += dv
    if 'down' in k: v.y -= dv
    if 'up' in k: v.y += dv
    ball.pos += v*dt
```

# Miscellaneous

Other Documentation here.

## LaTex

Latex in VPython Documentation here

To render Latex, insert the following code:

```
MathJax.Hub.Queue(["Typeset",MathJax.Hub])
```

All Latex backslashes ( ie. \ ) must be replaced by double backslashes ( ie. \\ ).

All Latex statements must be enclosed by **\\(** <latex here> **\\)** or **$** <latex here> **$** or **$$** <latex here> **$$**, where $$ moves the equation to a new line.

$$\text{Final kinetic energy} = \frac{1}{2} m v_i^2 + \int_i^f \vec{F} \circ d\vec{r}$$

```
box()

scene.caption = "Final kinetic energy = \\( \\dfrac {1} {2}mv_i^{2}+\\int _{i}^{f}\\vec{F}\\circ d \\vec{r} \\)"

MathJax.Hub.Queue(["Typeset",MathJax.Hub])
```

$$\dfrac{5}{7}$$

$$a^b$$

$$\begin{bmatrix} 1 & 2 & 3\end{bmatrix}$$

```
box()

scene.caption = "$\\dfrac {5} {7}$"

scene.append_to_caption("$$a^b$$")

scene.append_to_caption("\\begin{bmatrix} 1 & 2 & 3\\end{bmatrix}")

MathJax.Hub.Queue(["Typeset",MathJax.Hub])

MathJax.Hub.Queue(["Typeset",MathJax.Hub])
```

To dynamically change text, `MathJax.Hub.Queue(["Typeset",MathJax.Hub])` must be called after every update to re-render the latex.

```
box()

scene.title = "\\(\\dfrac {5} {7} \\)"

def latex():

    scene.title = "\\(\\dfrac {3y} {4x} \\)"

    MathJax.Hub.Queue(["Typeset",MathJax.Hub])     # re-render latex

button(bind=latex,text='change')
```

## Cloning

Use `copyObj = obj.clone()` to clone/copy the obj. Cloning documentation here.

```
b = box(pos=vector(1,1,0))

bcopy = b.clone(pos=vector(1,-1,0))
```

## Compound

Compound documentation here.

Use `newObj = compound([obj1, obj2])` to combine `obj1` and `obj2` into one object, `newObj`. Both objects can now be controlled by just calling `newObj`.

```
handle = cylinder( size=vector(1,.2,.2), color=vector(0.72,0.42,0) )
```

```
head = box(size=vector(.2,.6,.2),pos=vector(1.1,0,0),color=color.gray(.6))
hammer = compound([handle, head])
```

## Camera

The camera's position and axis can be controlled using *scene.camera.*

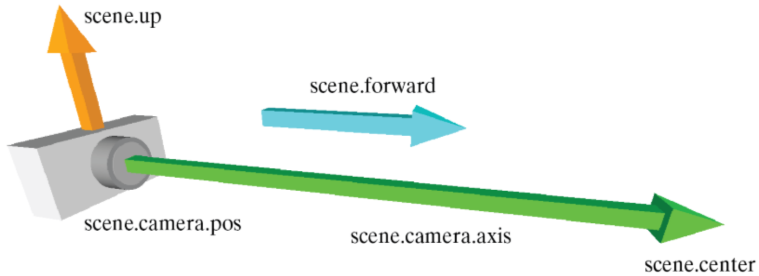



**Figure 8 Camera controls**

### Camera Follow

The camera can be made to follow a moving object.

Declare `scene.camera.follow(obj)` immediate after creating the object.

```
ball = sphere()
scene.camera.follow(ball)
```

## Camera Control

Use `scene.camera.pos` to get the camera's position and to control the camera's position:

```
scene.camera.pos = vector(#,#,#)
```

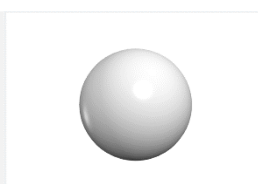
Use `scene.camera.axis` to get camera's direction and to control direction:

```
scene.camera.axis = vector(#,#,#)
```

Use `scene.range = #` to zoom out and establish a wider range.

## Canvas

The canvas is the window that displays the 3D objects and can be customized.



```
canvas(width=700,height=500,background=color.white)
sphere()
```

**Table 10 Canvas Parameters**

| Code | Parameter | Info |
|---|---|---|
| **width** = # | Width | Width of the canvas window |

| | | |
|---|---|---|
| **height** = # | Height | Height of the canvas window |
| **background** = color.<color> | Background Color | The background color of the window |
| **resizable** = True/False | Resizable | Whether the window can be resized (default=True) |
| **align** = "left"/"right" | Align canvas | Alignment of canvas on the page (default="left") |
| **ambient** = color.<color> | Ambient Light Color | Color of the light (default=color.gray(0.2)) |
| **lights** = [distant_light( direction=vec( 0.22, 0.44, 0.88), color=color.gray(0.8))] | Adding new light source | Add a light source with its light direction and light color |

### Multiple Canvases

There can be more than one canvases, assign each of the canvases to a variable. When using multiple canvases, all the objects must have the **`canvas=`** parameter to assign each object to a canvas variable.

```
canvas1 = canvas()                  # canvas 1
canvas2 = canvas()                  # canvas 2
box(canvas=canvas1)                 # assign box to canvas 1
sphere(canvas=canvas2)              # assign sphere to canvas 2
```

## Scene Text

To add *text* above the 3D window, use `scene.append_to_title ("text here")`.

To add *text* below the 3D window, use `scene.append_to_caption ("text here")`.

To add *widgets* above the 3D window, use the pos attribute, `pos = scene.title_anchor`.

To add *widgets* below the 3D window, use the pos attribute, `pos = scene.caption_anchor`.

To add *widgets* to the print box area at the bottom of webpage, use `pos = print_anchor`.

## Delete Object

```
my_box = box()
my_box.visible = False          # makes invisible
del my_box                      # deletes from program memory
```

## Bounding Box

Bounding Box returns coordinates of all 8 corners of the object as a list. Each vector coordinate can be accessed using `my_box.bounding_box()[#]` and each value in the coordinate can be accessed using `my_box.bounding_box()[#].x/y/z`. Documentation here.

```
my_box = box()
my_box.bounding_box()
# returns:[<0,0,-1.5>, <0,0,0>, <0,2,-1.5>, <0,2,0>, <1,0,-1.5>, <1,0,0>,
# <1,2,-1.5>, <1,2,0>]
```

```
my_box.bounding_box()[0]          # returns 1 st coordinate: <0, 0, -1.5>
my_box.bounding_box()[0].x        # returns x value: 0
```

## Sharing


**Figure 9 Share button at the top**

To share the program, in edit mode, click **Share or export this program** at the top. There are 3 options:

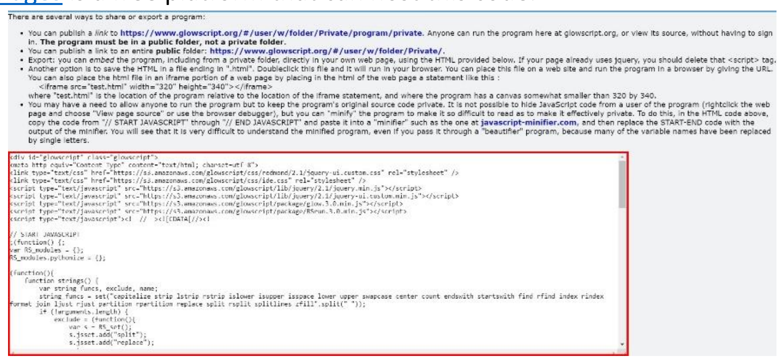
1. Copy the HTML and Javascript code in the textbox at the bottom and paste it into your website code. Github Pages is a free platform that can host this code.
   a. 

   **Figure 10 HTML/Javascript code to share**
2. Go back to **Edit this Program** and **run** the program, if the program is in a **public** folder, the *URL* of the page running the program can be shared directly.
3. The page running the program can be embedded directly into your website using HTML's *iframe*:
   a.
   ```
   <iframe src="glowscript.org/<program url>" width="320"
   height="340"></iframe>
   ```

## Backup

The programs linked to your Google account will be stored on Glowscript's servers but the python code can also be backed up just in case.


**Figure 11 Download .py file button**

To back up the program on your computer or the cloud, go to edit the program page and click the **Download** button at the top, this will download the python file (.py). Safely store this python file.

To reupload a .py file back onto glowscript.org, copy the .py code into a blank Glowscript program. Then remove the `from vpython import *` and remove the `#` in front of `GlowScript 3.0 VPython`.

## Examples

Glowscript examples here

## References

[1] Glowscript, "glowscript.org," [Online]. Available: https://www.glowscript.org/docs/VPythonDocs/index.html. [Accessed July 2020].

[2] L. C. Physics, "VPython for beginners," [Online]. Available: https://www.youtube.com/playlist?list=PLdCdV2GBGyXOnMaPS1BgO7IOU_00ApuMo. [Accessed July 2020].