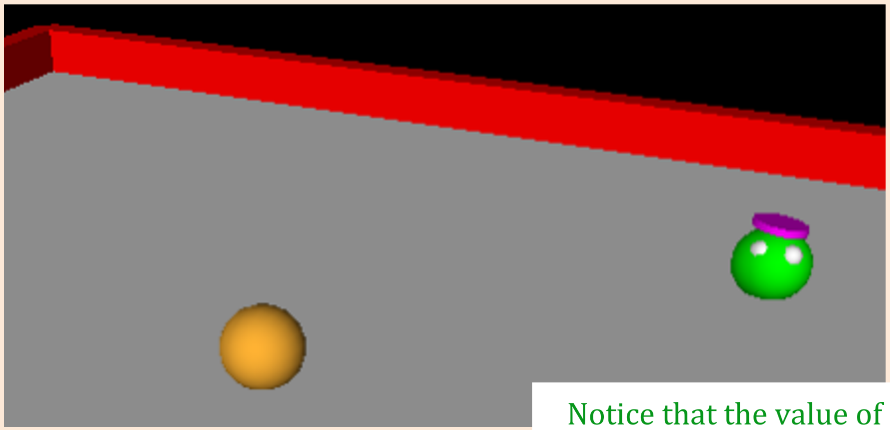
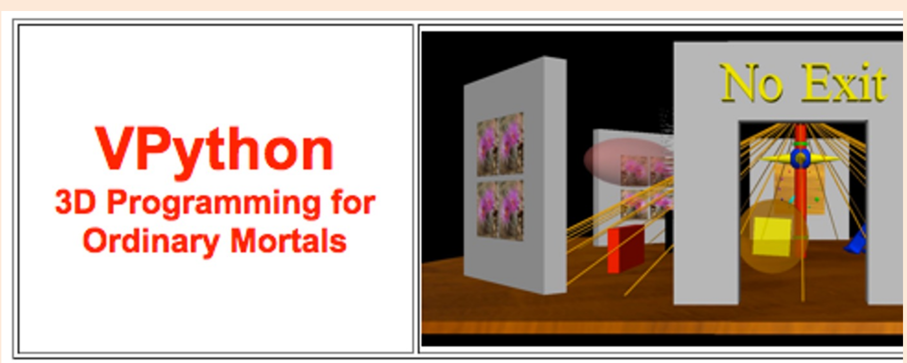
This week's `classes`...

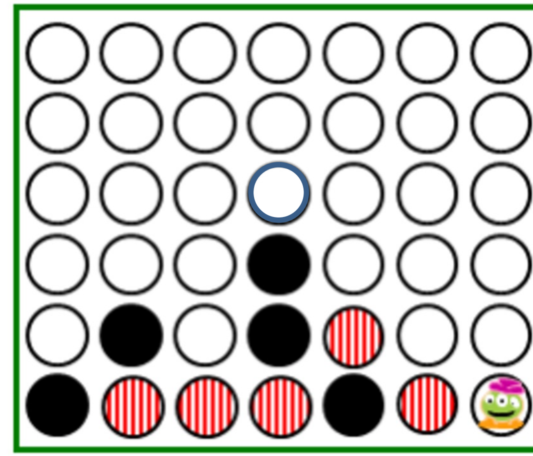


Notice that the value of **(dimension + eyes)** is conserved ~ at **5**!

Three-eyed? This week, we're ***3d***'ed!



Homework #11, due 4/16



Connect 4

`aiMove`

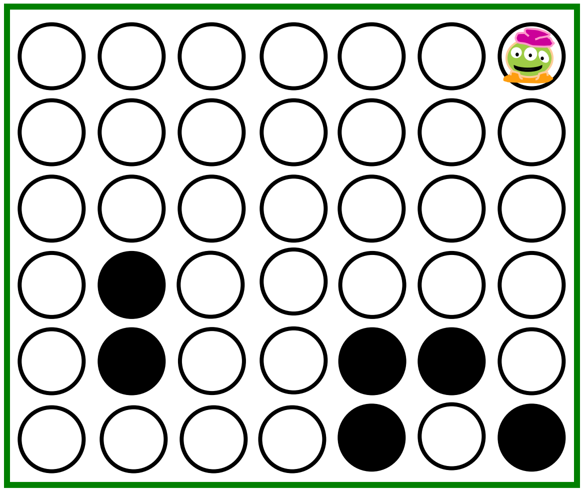
whether it's black's move or red's, they're eye-ing the same column!

Connect **4**, Part **2**

hw11pr2.py

Covering on Thursday!

b



```
| | | | | | | |
| | | | | | | |
| | | | | | | |
| |X|O| | | | |
| |X|O| |X|X| |
|O|O|O| |X|O|X|
---------------
 0 1 2 3 4 5 6
```

what methods will help?

`colsToWin( self, ox )`

`b.colsToWin('O')`

`b.colsToWin('X')`

what methods will help?

`aiMove( self, ox )`

`b.aiMove('O')`

`b.aiMove('X')`

`hostGame( self )`

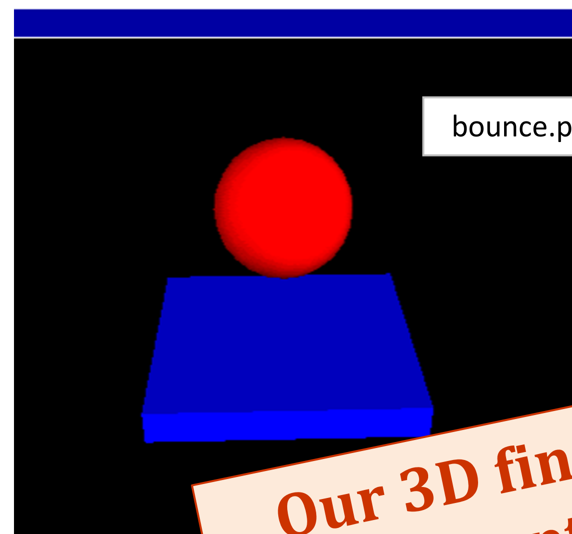
VPython ~ GlowScript!

https://vpython.org/index.html

stonehenge.py

built *by* and *for* physicists to simplify 3d simulations

Try this out in lab on Friday!

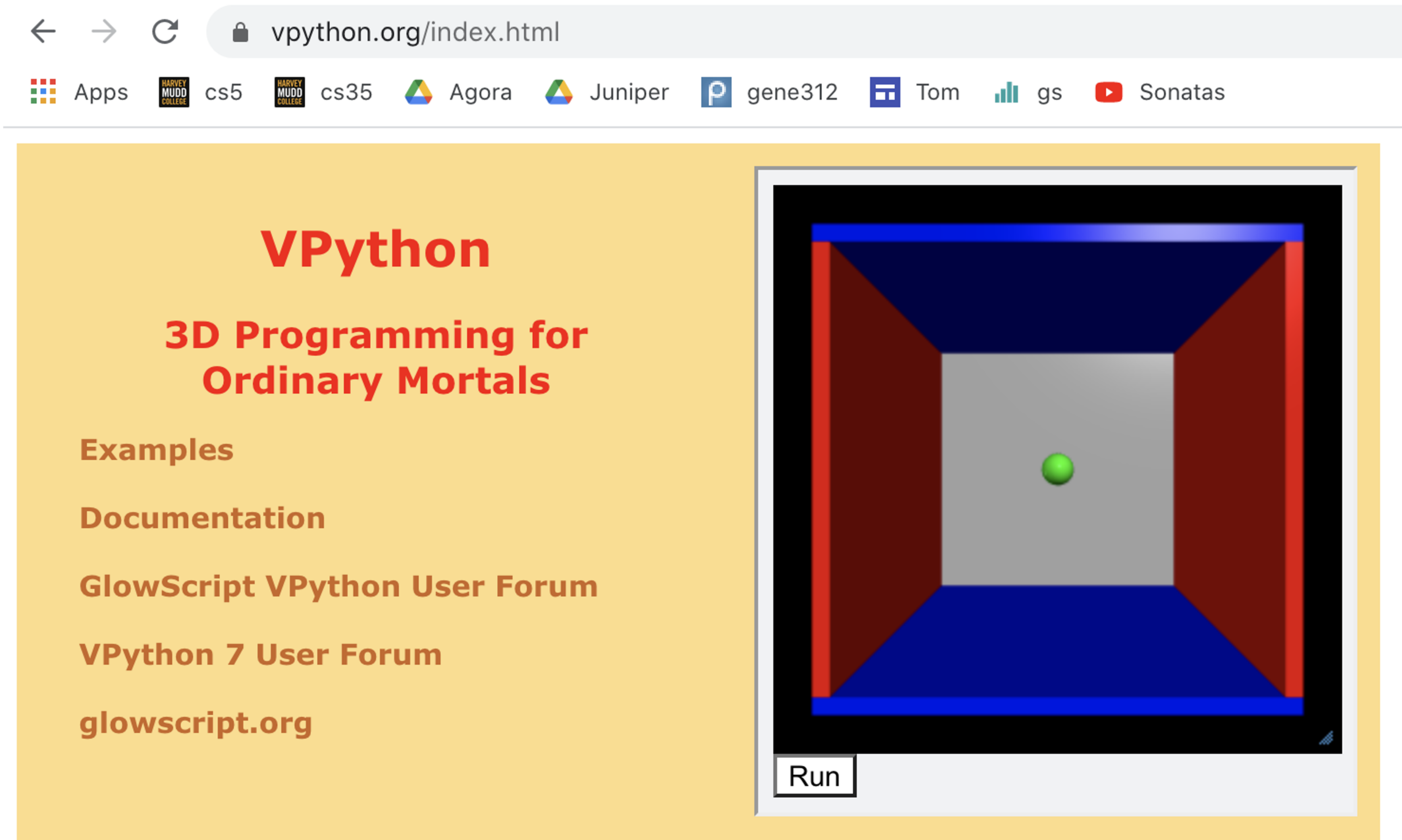


bounce.py

Our 3D final-project option

VPython ~ GlowScript!

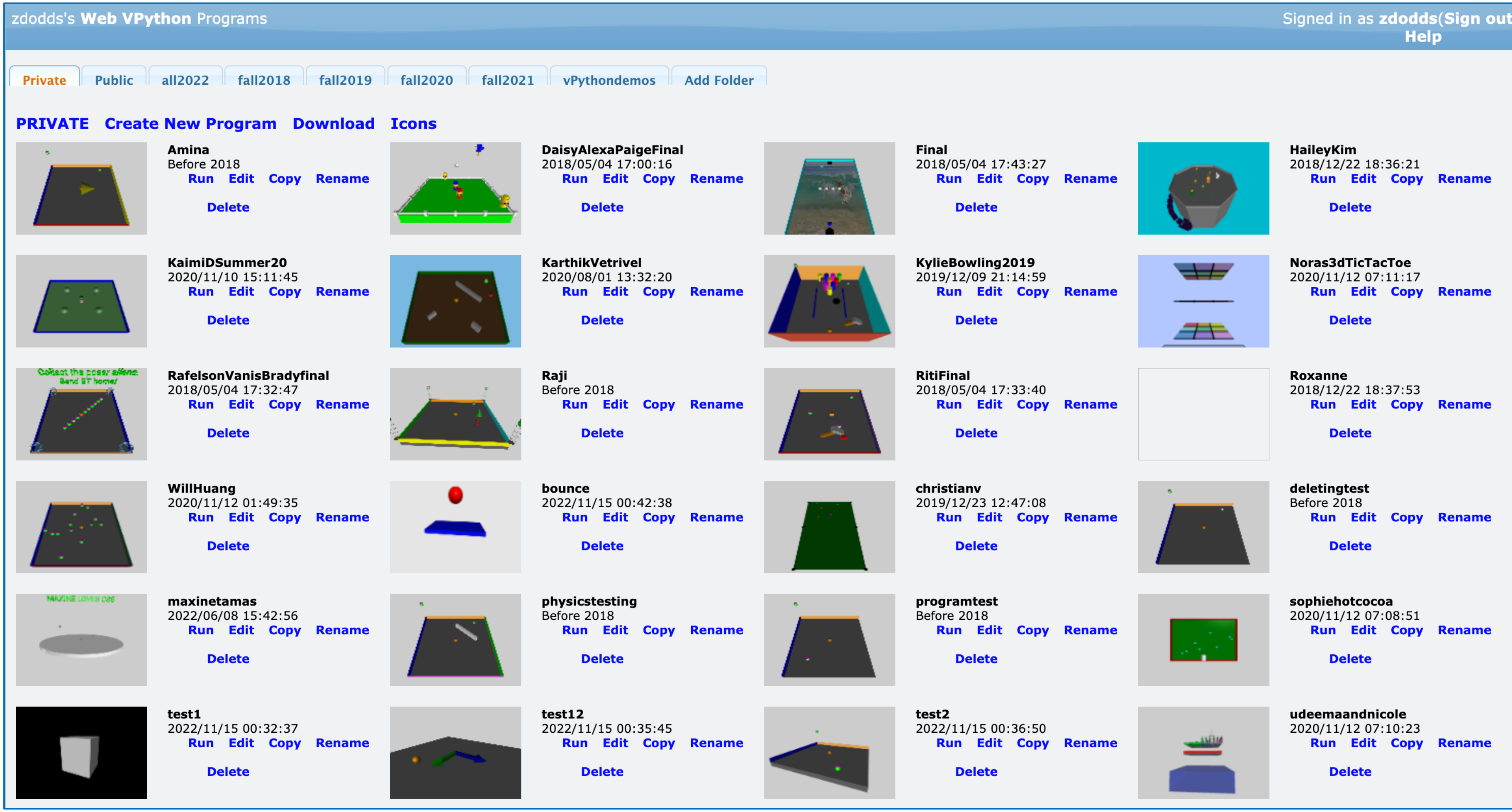
https://vpython.org/index.html



Try it! (See if you can Zoom / Rotate...)

VPython ~ GlowScript!

https://vpython.org/index.html



Let's try an example...

Python features, *motivated* by VPython…

Python features, *motivated* by VPython…

Tuples

default and named inputs

Python features, *motivated* by VPython...

Tuples

```
T = (4,2)        x = (1,0,0)
```

```
def f(x=3, y=17):
    return 10*x + y
```

default and named inputs

Python features, *motivated* by VPython...

Tuples are similar to lists, but they're parenthesized:

```
T = (4,2)        x = (1,0,0)
```

example of a two-element ***tuple*** named T and a three-element tuple named x

not vectors!

```
def f(x=3, y=17):
    return 10*x + y
```

examples of ***default and named inputs*** in a function definition

Python features, *motivated* by VPython...

Tuples are similar to lists, but they're parenthesized:

```
T = (4,2)        x = (1,0,0)
```

example of a two-element ***tuple*** named T and a three-element tuple named x

Etymology [edit]

The term originated as an abstraction of the sequence: single, double, triple, quadruple, quintuple, sextuple, septuple, octuple, ..., *n*-tuple, ..., where the prefixes are taken from the Latin names of the numerals. The unique 0-tuple is called the null tuple.

Tuples!

Lists that use parentheses are called ***tuples***:

```
T = ( 4, 2 )

T
(4, 2)

T[0]
4

T[0] = 42
Error!

T = ('a',2,'z')
```

Tuples are <u>immutable</u> lists: you can't change their elements...

...but you can always redefine the whole variable, if you want!

+ Tuples are more memory + time efficient
+ Tuples ***can*** be dictionary keys: ***lists can't***
  - ***But***, *you* ***can't change*** *tuples' elements!*

Tuple surprises...

Creating 0- and 1-tuples
would seem like a problem!

A bug from last week's **`Board`** class:

```
W = 4 # for example
s = " ",
for col in range(W):
    s += str(col), " "
```

trying for

```
" 0 1 2 3 "
```

yields a surprising result for **`s`**

Tuple surprises...

Creating 0- and 1-tuples would seem like a problem!

A bug from last week's **`Board`** class:

```
W = 4 # for example
s = " ",
for col in range(W):
    s += str(col), " "
```

trying for

~~`" 0 1 2 3 "`~~

yields a surprising result for **`s`**

actually creates a tuple:

```
(' ','0',' ','1',' ','2',' ','3',' ')
```

Default – *and named* – inputs!

Functions can have ***default input values*** and can take ***named inputs***

function def'n

```
def f(x=3, y=17):
    return 10*x + y
```

example of ***default*** input values for x and y

function CALL

```
f(4,2)
```

inputs in order!

Calling functions

Functions can have ***default input values*** and can take ***named inputs***

function def'n

```
def f(x   , y   ):
    return 10*x + y
```

function CALL

```
f(4,2)
```

inputs <u>in order</u>!

Function-call inputs *look like* tuples, but they're not quite the same…

Named inputs!

Functions can have ***default input values*** and can take ***named inputs***

function def'n

```
def f(x   , y   ):
    return 10*x + y
```

function CALL

```
f(x=4,y=2)
```

example of ***named*** input values for x and y

inputs by name!

Inputs by name *override* inputs by order

```
f(y=2,x=4)
```

Default inputs!

Functions can have ***default input values*** and can take ***named inputs***

function def'n

```
def f(x=3, y=17):
      return 10*x + y
```

example of ***default*** input values for x and y

function CALL

```
f(x=4,y=2)
```

example of ***named*** input values for x and y

inputs by name!

Default inputs fill in *only where there are gaps* `f(y=2)`

Default – *and named* – inputs!

Functions can have ***default input values*** and can take ***named inputs***

```
def f(x=3, y=17):
    return 10*x + y
```

example of an ordinary function call – totally OK

```
f(4,2)
```

example of *default inputs*

```
f()
```

example using only one *default input*

```
f(1)
```

example of a *named input*

```
f(y=1)
```

This is a *different function*, **f**:

```
def f(x=2, y=11):
    return x + 3*y
```

Input name(s) = ____________________

Named inputs

`f(3,1)` →	6
`f(3)` →	36
`f()` →	35
`f(y=4,x=2)` →	14

What will the above function calls return?

Not one of the above is 42!

but they all share a factor with it! - Eli B. '17

What is the ***shortest*** call to **f** returning 42? → f(9)

it's only four characters, too!

What call to **f** returns the string **'Lalalalala'** ? → f("Lala", "la")

you can pass strings into **f**!

These are tuples! They work like lists: What is **f**((), (1,0)) ? → (1,0,1,0,1,0)

you can pass tuples into **f**!

Mind Muddler:

Extra! *W*hat does this return? **y = 60; x = -6; f(y=x,x=y)** → 42

This is a *different function*, **f**:

```
def f(x=2, y=11):
    return x + 3*y
```

~ Solutions ~

Named inputs

`f(3,1)` → 6

`f()` → 35

`f(3)` → 36

`f(y=4,x=2)` →

What will the above function calls return?

Not one of the above is 42!

but they all share a factor with it! - Eli B. '17

What is the ***shortest*** call to **f** returning 42? → `f(9)`

it's only four characters, too!

What call to **f** returns the string 'Lalalalala' ? → `f('Lala','la')`

you can pass strings into **f**!

These are tuples! They work like lists: What is `f( (), (1,0) )` ? → `(1,0,1,0,1,0)`

you can pass tuples into **f**!

Mind Muddler:

Extra! *W*hat does this return? y = 60; x = -6; f(y=x,x=y) → 42

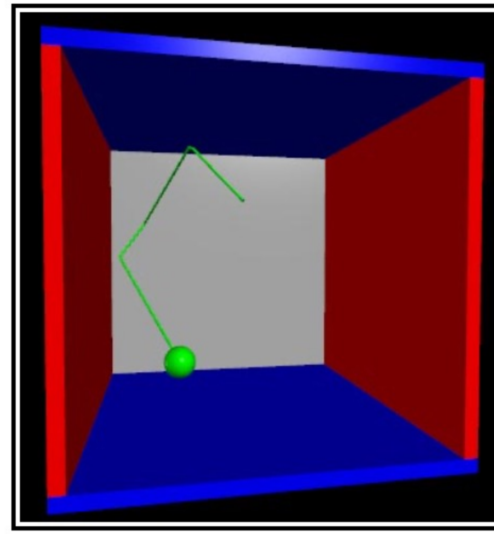
Using GlowScript / vPython...

www.glowscript.org/

Web VPython

Signed in as **Prof. Melissa**(**Sign out**)
Help

VPython is an easy-to-use, powerful environment for creating 3D animations. Here at glowscript.org (or **webvpython.org**, which takes you here), you can write and run VPython programs right in your browser, store them in the cloud for free, and easily share them with others. You can also use VPython with installed Python: see **vpython.org**.



The **Help** provides full documentation.
Welcome to VPython, a **Trinket** tutorial, is useful for anyone new to programming in VPython.

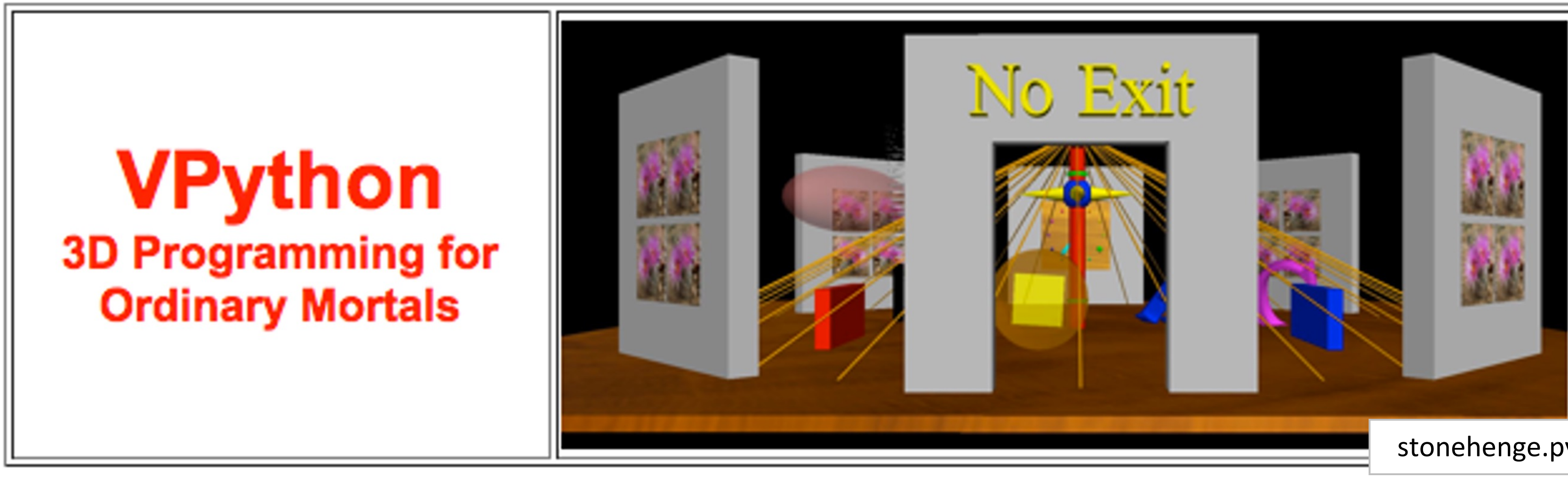
You are signed in as **Prof. Melissa** and your programs are **here**.
Your files will be saved here, but it is a good idea to backup your folders or individual files occasionally by using the download options that are provided.

Version 3.2

Example programs | **Forum**

VPython ~ GlowScript!

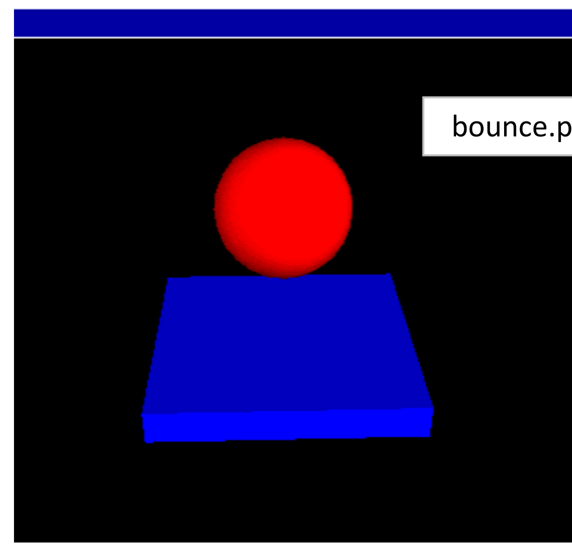
www.glowscript.org/



stonehenge.py

built *by* and *for* physicists to simplify 3d simulations

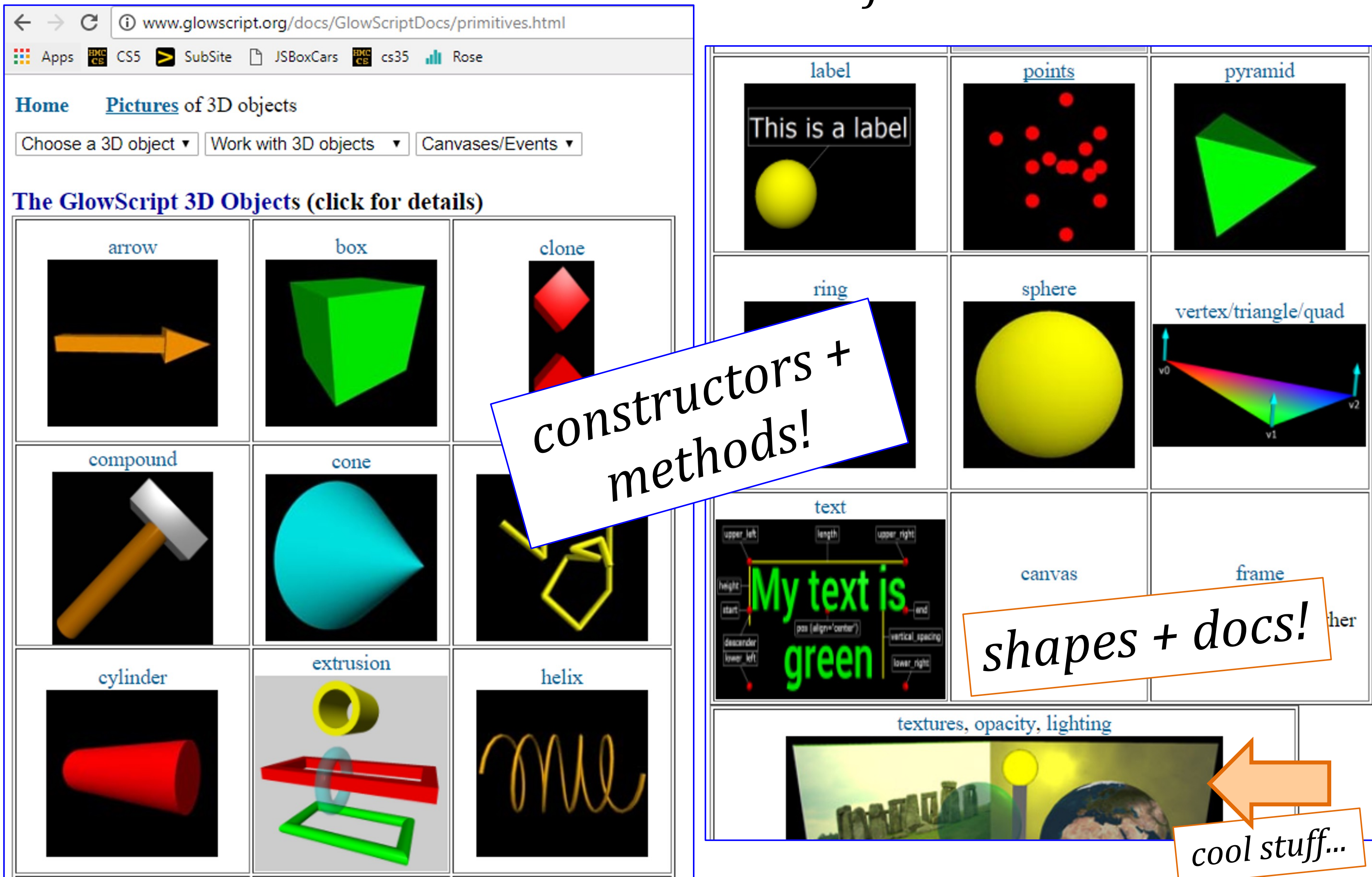
lots of available classes, objects and methods in its **API**



bounce.py

API

... stands for *Application Programming Interface*



API

... stands for *Application Programming Interface*

a ***programming*** description of how to access the functionality of a software library

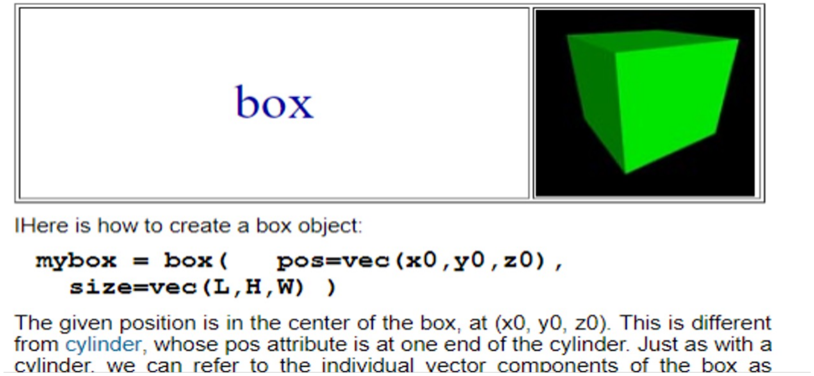
Classes!

Methods!

Conventions!

How do we learn an API?

Documentation


box

IHere is how to create a box object:

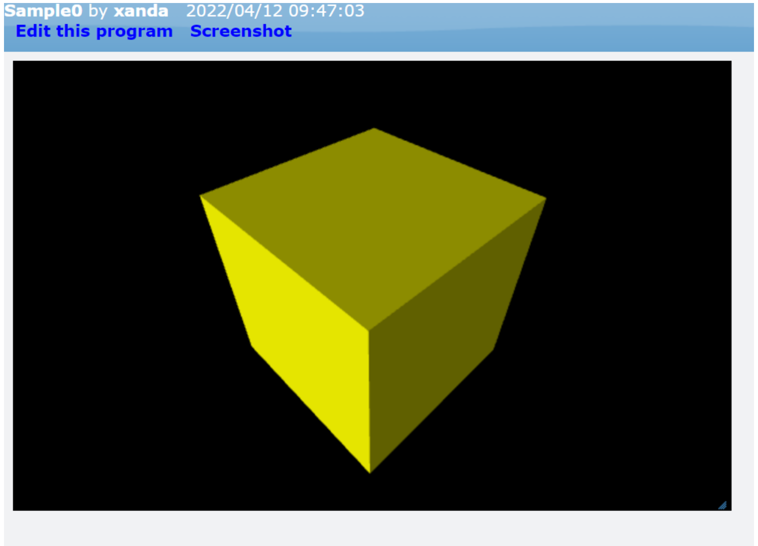
```
mybox = box(   pos=vec(x0,y0,z0),
   size=vec(L,H,W) )
```

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

Examples

```
# the simplest possible vpython program:
box( color = vector(1, 1, 0) )
```

Running things!



API

Examples

A demo of vPython's API:

```
# the simplest possible vpython program:
box( color = vector(1, 1, 0) )

# try changing the color: the components are
#   red, green, blue    each from 0.0 to 1.0

# then, add a second parameter:   size=vector(2.0,1.0,0.1)
#       the order of those three #s: Length, Height, Width

# then, a third parameter:  axis=vector(2,5,1)
#       the order of those three #s: x, y, z
```

What's **box**?
What's **color**?
What's **vector**?

vPython example API call(s)

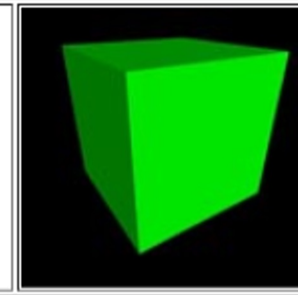
I'm hAPI
about APIs!

API

Documentation

constructor + default arguments; data!

box



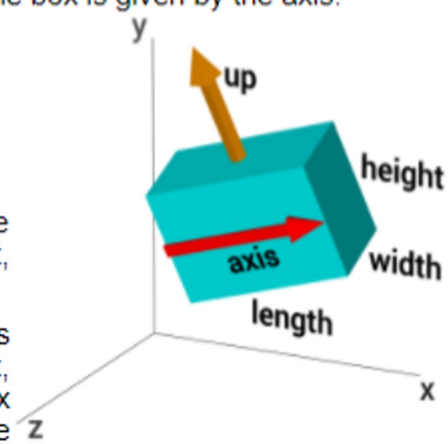
IHere is how to create a box object:

```
mybox = box(   pos=vec(x0,y0,z0),
   size=vec(L,H,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.pos.x, mybox.pos.y, and mybox.pos.z. For this box, we have mybox.axis = vec(1, 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:

```
mybox = box(
  pos=vec(x0,y0,z0),
  axis=vec(a,b,c),
  size=vec(L,H,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.

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

vectors

b.pos, b.vel,... are vectors

```
b.vel = vector(1,0,0)
```

vel.x
vel.y
vel.z

named components

```
b.pos = vector(0,0,0)
```

scalar multiplication

```
b.pos = b.pos + b.vel*0.2
```

component-by-component addition

compare with tuples...

vectors

act like 3D "*arrows*"

The vector Object

The vector object is not a displayable object but is a powerful aid to 3D computations.

```
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 <1 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].

www.glowscript.org/docs/GlowScriptDocs/vector.html

vectors!

lots of support!
(*don't write your own*)

Vector functions

The following functions are available for working with vectors:

mag(A) = A.mag = |A|, the magnitude of a vector

mag2(A) = A.mag2 = |A|*|A|, the vector's magnitude squared

norm(A) = A.norm() = A/|A|, a unit vector in the direction of the vector

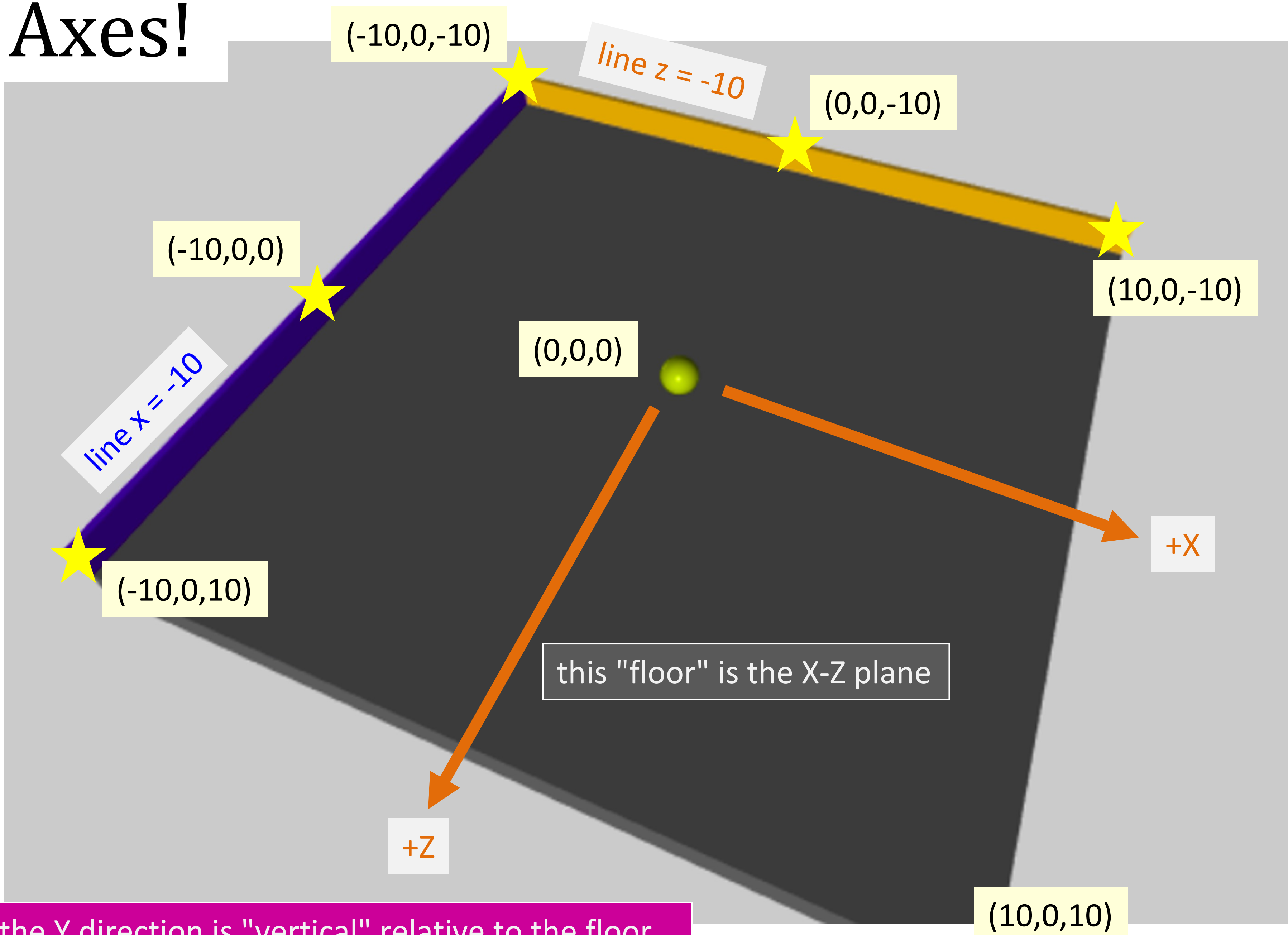
hat(A) = A.hat = A/|A|, a unit vector in the direction of the vector; an alternative to A.norm(), based on the fact that unit vectors are customarily written in the form **ĉ**, with a "hat" over the vector

dot(A,B) = A.dot(B) = A dot B, the scalar dot product between two vectors

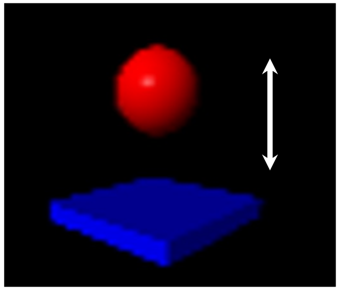
cross(A,B) = A.cross(B), the vector cross product between two vectors

diff_angle(A,B) = A.diff_angle(B), the angle between two vectors, in radians

Axes!



the Y direction is "vertical" relative to the floor...



vPython!

Look over this VPython program to determine:

(1) How many distinct vPython <u>classes</u> are here? ______
(2) How many <u>named inputs</u> are here? ____________________
(3) Tricky! How many vPython <u>objects</u> are here? _________
(4) What lines of code handle ***collisions*** ?
(5) How does ***"physics"*** work? Where is it?
(6) *Wind!* Add a line to create a *<u>horizontal acceleration</u>* ...

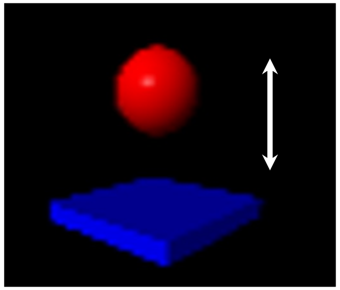
```
floor = box(length=4, width=4, height=0.5, color=vector(0,0,1))

ball = sphere(pos=vector(0,4.2,0), radius=1, color=vector(1,0,0))
ball.vel = vector(0,-1,0)  # this is the velocity

RATE = 30
dt = 1.0/RATE

while True:
    rate(RATE)

    ball.pos = ball.pos + ball.vel*dt

    if ball.pos.y < ball.radius:
        ball.vel.y *= -1.0
    else:
        ball.vel.y += -9.8*dt
```

Let's run this first...

what is this doing? (ball.pos = ball.pos + ball.vel*dt)

what is the if doing?

what is the else doing?



vPython

Look over this VPython program to determine:

(1) How many distinct vPython classes are here? 3

(2) How many named inputs are here? 7

(3) Tricky! How many vPython objects are here? 6 (or 7 once A)

(4) What lines of code handle ***collisions*** ?

(5) How does ***"physics"*** work? Where is it?

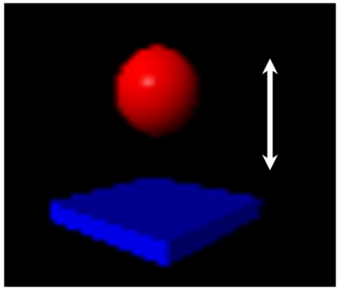
(6) Wind! Add a line to create a *horizontal acceleration* ...

```
floor = box(length=4, width=4, height=0.5, color=vector(0,0,1))

ball = sphere(pos=vector(0,4.2,0), radius=1, color=vector(1,0,0))
ball.vel = vector(0,-1,0)  # this is the velocity

RATE = 30
dt = 1.0/RATE

while True:
    rate(RATE)

    ball.pos = ball.pos + ball.vel*dt

    if ball.pos.y < ball.radius:
        ball.vel.y *= -1.0
    else:
        ball.vel.y += -9.8*dt
```

what is this doing?

what is the if doing?

what is the else doing?



vPython

Look over this VPython program to determine:

(1) How many distinct vPython classes are here? ___3___
(2) How many named inputs are here? ___7___
(3) Tricky! How many vPython objects are here? ___6___
(4) What lines of code handle ***collisions*** ?
(5) How does ***"physics"*** work? Where is it?
(6) Wind! Add a line to create a *horizontal acceleration* ...

```
floor = box(length=4, width=4, height=0.5, color=vector(0,0,1))

ball = sphere(pos=vector(0,4.2,0), radius=1, color=vector(1,0,0))
ball.vel = vector(0,-1,0)  # this is the velocity

RATE = 30
dt = 1.0/RATE

while True:
    rate(RATE)

    ball.pos = ball.pos + ball.vel*dt

    if ball.pos.y < ball.radius:
        ball.vel.y *= -1.0
    else:
        ball.vel.y += -9.8*dt
```

named inputs

objects

classes

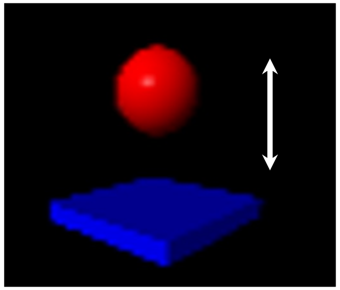
PHYSICS!

COLLISIONS!

GRAVITY!

Wind!

```
ball.vel.x += .5*dt
```



What makes things go?

```
floor = box(length=4, width=4, height=0.5, color=vector(0,0,1))

ball = sphere(pos=vector(0,4.2,0), radius=1, color=vector(1,0,0))
ball.vel = vector(0,-1,0)  # this is the velocity

RATE = 30
dt = 1.0/RATE

while True:
    rate(RATE)

    ball.pos = ball.pos + ball.vel*dt

    if ball.pos.y < ball.radius:
        ball.vel.y *= -1.0
    else:
        ball.vel.y += -9.8*dt
```

rate tells us the loops per second!

dt is the duration of *one* iteration (1/rate)

Computing **dt** and updating **pos** are *our* responsibility!

Lab goals

(0) Try out VPython: Get your bearings (*axes!*)

(1) Make guided changes to the starter code...

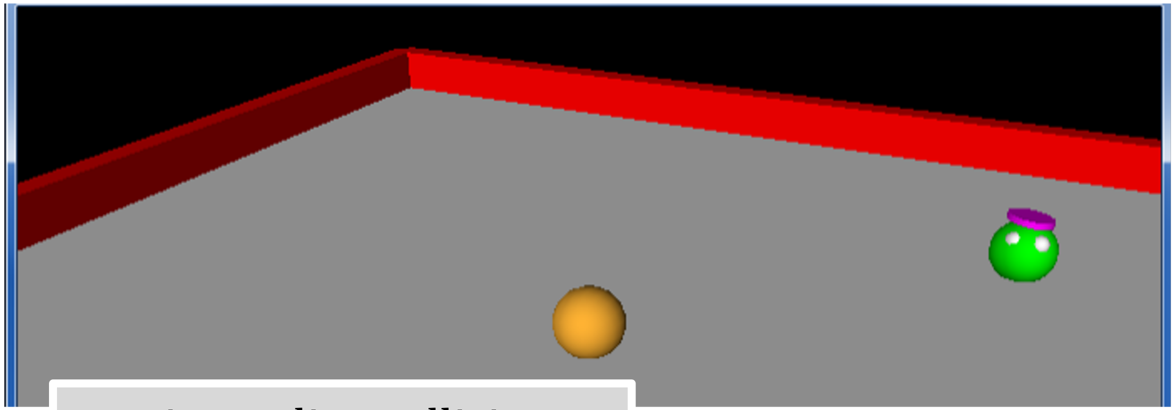
(2) Expand your ***walls*** and ***wall-collisions***…

(3) Improve your interaction/game!

(4) *Optional*: add scoring, enemies, or a moving target, hoops, traps, holes, etc. ~ ***final project...***

can expand to become a final project...

Collisions…



point-to-line collisions

Idea:

When the **ball** hits a **wall**,

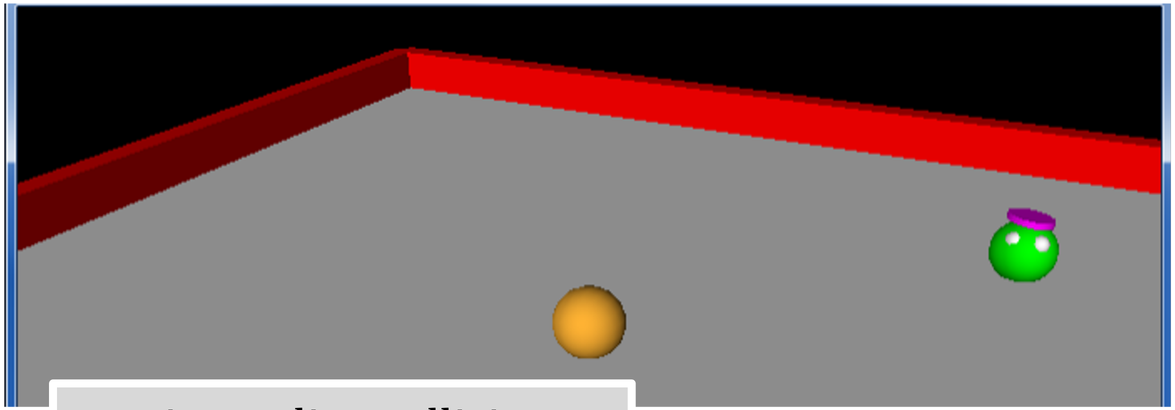
boundary collisions — the **ball** should **bounce**

When the **ball** hits the **alien**,

point-to-point collisions — the **alien** should **ascend**

How do we operationalize these?

Collisions...



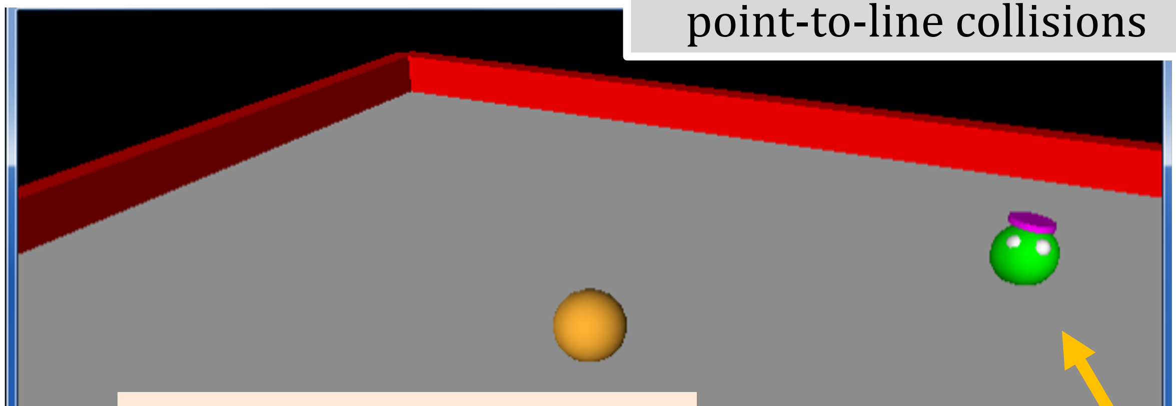
point-to-line collisions

```
# if the ball hits wallA
if ball.pos.z < wallA.pos.z:      # hit - check for z
    ball.pos.z = wallA.pos.z       # bring back into bounds
    ball.vel.z *= -1.0                    # reverse the z velocity

# if the ball hits wallB
if ball.pos.x < wallB.pos.x:       # hit - check for x
    ball.pos.x = wallB.pos.x        # bring back into bounds
    ball.vel.x *= -1.0                      # reverse the x velocity

# if the ball collides with the alien, give a vertical velocity
if mag( ball.pos - alien.pos ) < 1.0:
    print("To infinity and beyond!")
    alien.vel = vector(0,1,0)
```

point-to-point collisions

Demo!

point-to-line collisions



point-to-point collisions

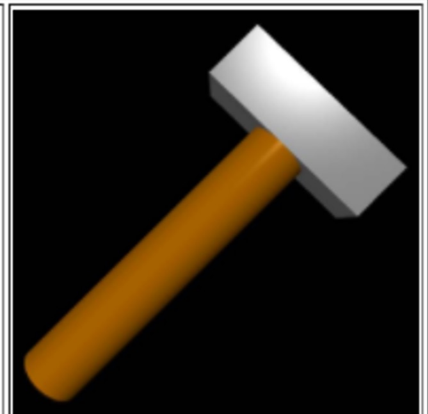
Why does this alien only have two eyes?

compound

Home   Pictures of 3D objects

Choose a 3D object ▾ | Work with 3D objects ▾ | Canvases/Events ▾

compound



The **compound** object lets you group objects together and manage them as though they were one object, by specifying in the usual way pos, color, size (and length, width, height), axis, up, opacity, shininess, emissive, and texture. Moreover, the display of a complicated compound object is faster than displaying the individual objects one at a time. (In GlowScript version 2.1 the **details were somewhat different**.)

The object shown above is a compound of a cylinder and a box:

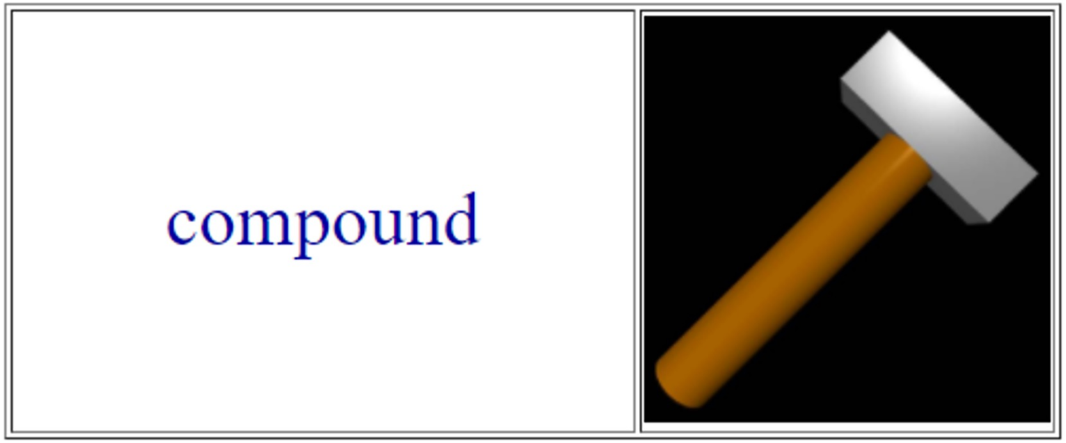
```
handle = cylinder( size=vec(1,.2,.2),
                   color=vec(0.72,0.42,0) )

head = box( size=vec(.2,.6,.2),
            pos=vec(1.1,0,0),
            color=color.gray(.6) )

hammer = compound([handle, head])
hammer.axis = vec(1,1,0)
```

The size of the object: After creating the compound named "hammer", **hammer.size** represents the size of the bounding box of the object.

compound

Home   Pictures of 3D objects

Choose a 3D object ▾ | Work with 3D objects ▾ | Canvases/Events ▾

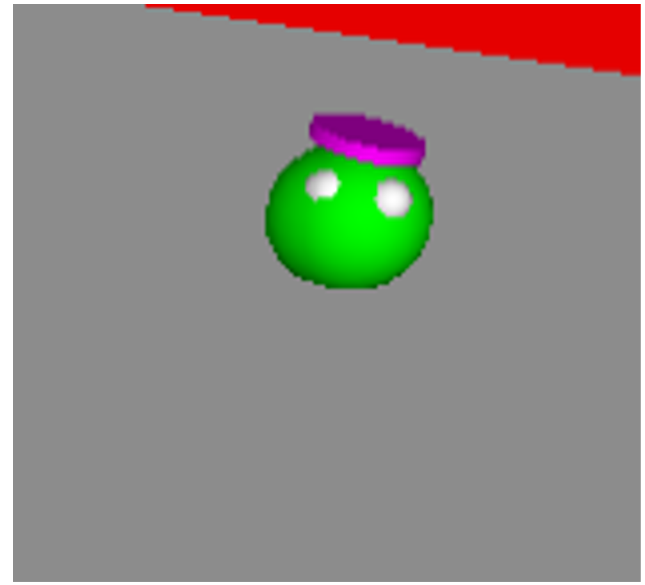


The **compound** object lets you group objects together and manage them as though they were one object, by specifying in the usual way pos, color, size (and length, width, height), axis, up, opacity, shininess, emissive, and texture. Moreover, the display of a complicated compound object is faster than displaying the individual objects one at a time. (In GlowScript version 2.1 the **details were somewhat different**.)

The object shown above is a compound of a cylinder and a box:

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

head = box( size=vec(.2,.6,.2),
             pos=vec(1.1,0,0),
             color=color.gray(.6) )

hammer = compound([handle, head])
hammer.axis = vec(1,1,0)
```

The size of the object: After creating the compound named "hammer", **hammer.size** represents the size of the bounding box of the object.

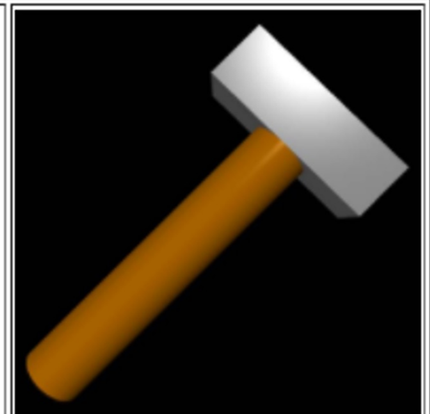


compound

Home   Pictures of 3D objects

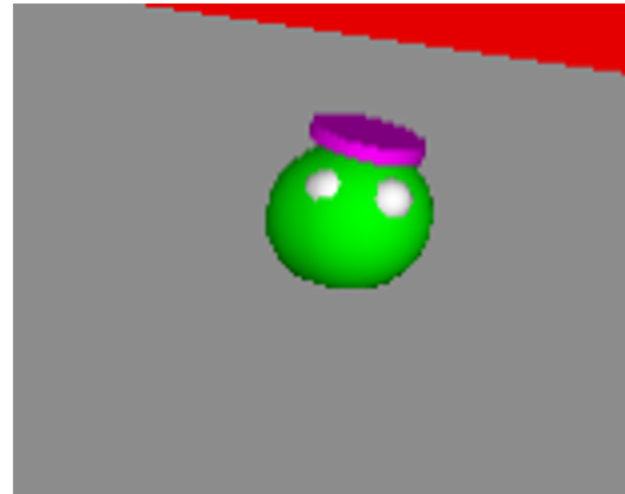
Choose a 3D object ▾   Work with 3D objects ▾   Canvases/Events ▾

compound



The **compound** object lets you group objects together and manage them as though they were one object, by specifying in the usual way pos, color, size (and length, width, height), axis, up, opacity, shininess, emissive, and texture. Moreover, the display of a complicated compound object is faster than displaying the individual objects one at a time. (In GlowScript version 2.1 the details were somewhat different.)

The object shown above is a compound of a cylinder and a box:



What's what here?

```
alien_body = sphere( size=1.0*vector(1,1,1), pos=vector(0,0,0), color=color.green )
alien_eye1 = sphere( size=0.3*vector(1,1,1), pos=.42*vector(.7,.5,.2), color=color.white )
alien_eye2 = sphere( size=0.3*vector(1,1,1), pos=.42*vector(.2,.5,.7), color=color.white )
alien_hat = cylinder( pos=0.42*vector(0,.9,-.2), axis=vector(.02,.2,-.02),
                                               size=vector(0.2,0.7,0.7), color=color.magenta)
alien_objects = [alien_body, alien_eye1, alien_eye2, alien_hat]

com_alien = compound( alien_objects, pos=starting_position )
```

Idea:

When the user presses:

the **ball** should accelerate:

up, W	**away from us (-z)**
left, A	**left (-x)**
down, S	**towards us (+z)**
right, D	**right (+x)**

key presses...

key presses...

random change of the sphere's color

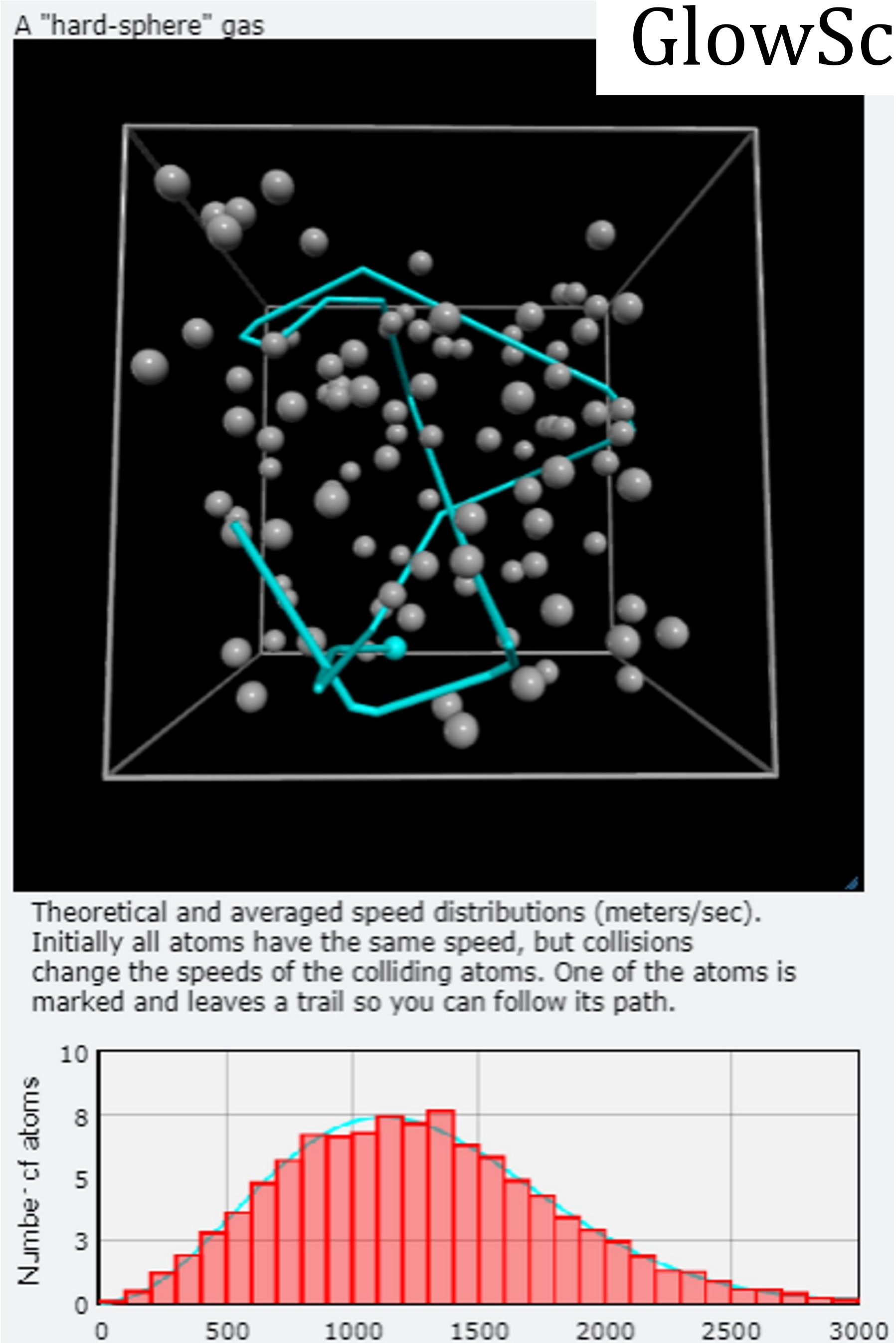
```
# +++ start of EVENT_HANDLING section -                          ..
#

def keydown_fun(event):
    """This function is called each time a key is pressed."""
    # ball.color = randcolor() # This turns out to be very distracting!
    key = event.key
    ri = randint(0, 10)
    print("key:", key, ri)

    amount = 0.42 # "Strength" of the keypress's velocity changes
    if key == 'up' or key in 'wWiI':
        ball.vel = ball.vel + vec(0, 0, -amount)
    elif key == 'left' or key in 'aAjJ':
        ball.vel = ball.vel + vec(-amount, 0, 0)
    elif key == 'down' or key in 'sSkK':
        ball.vel = ball.vel + vec(0, 0, amount)
    elif key == 'right' or key in "dDlL":
        ball.vel = ball.vel + vec(amount, 0, 0)
    elif key in ' rR':
        ball.vel = vec(0, 0, 0) # Reset! via R or the spacebar, " "
        ball.pos = vec(0, 0, 0)
```

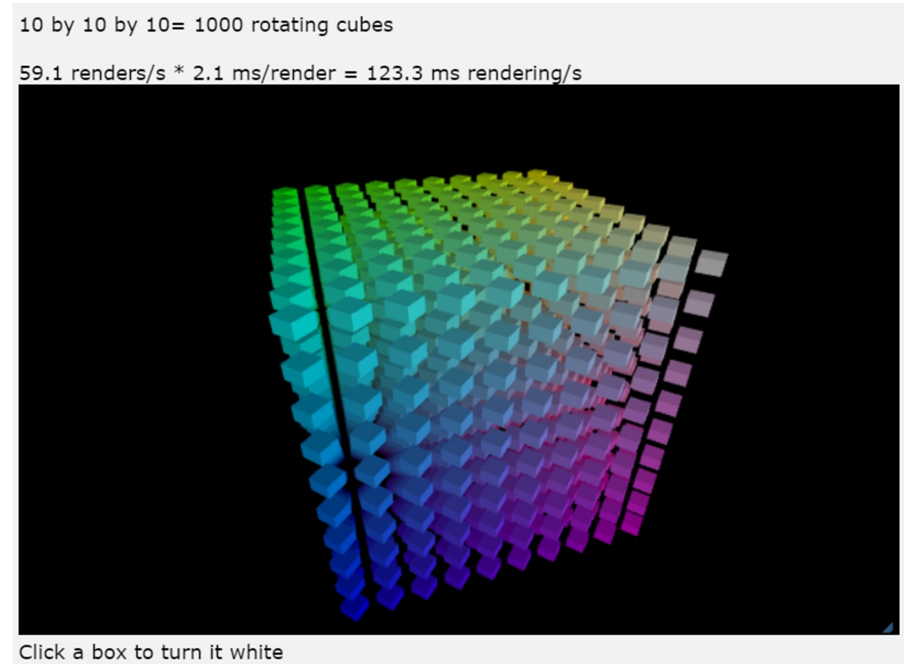
printing is great for debugging!

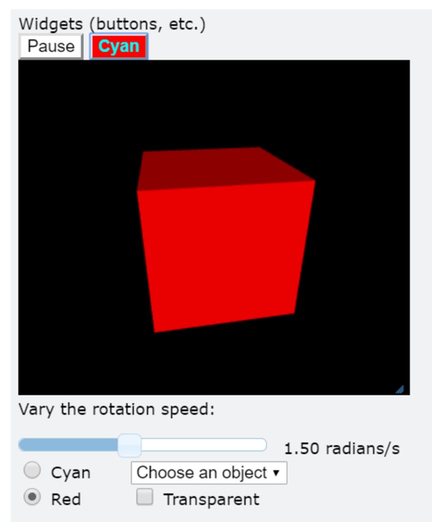
variables make it easy to change behavior across many lines of code (here, all four motion directions)

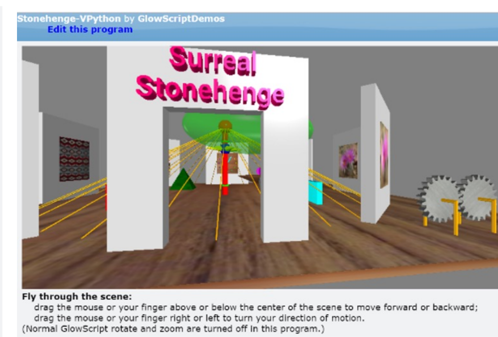
have shortcuts to make your game easier -- or to reset it!

GlowScript / vPython examples…



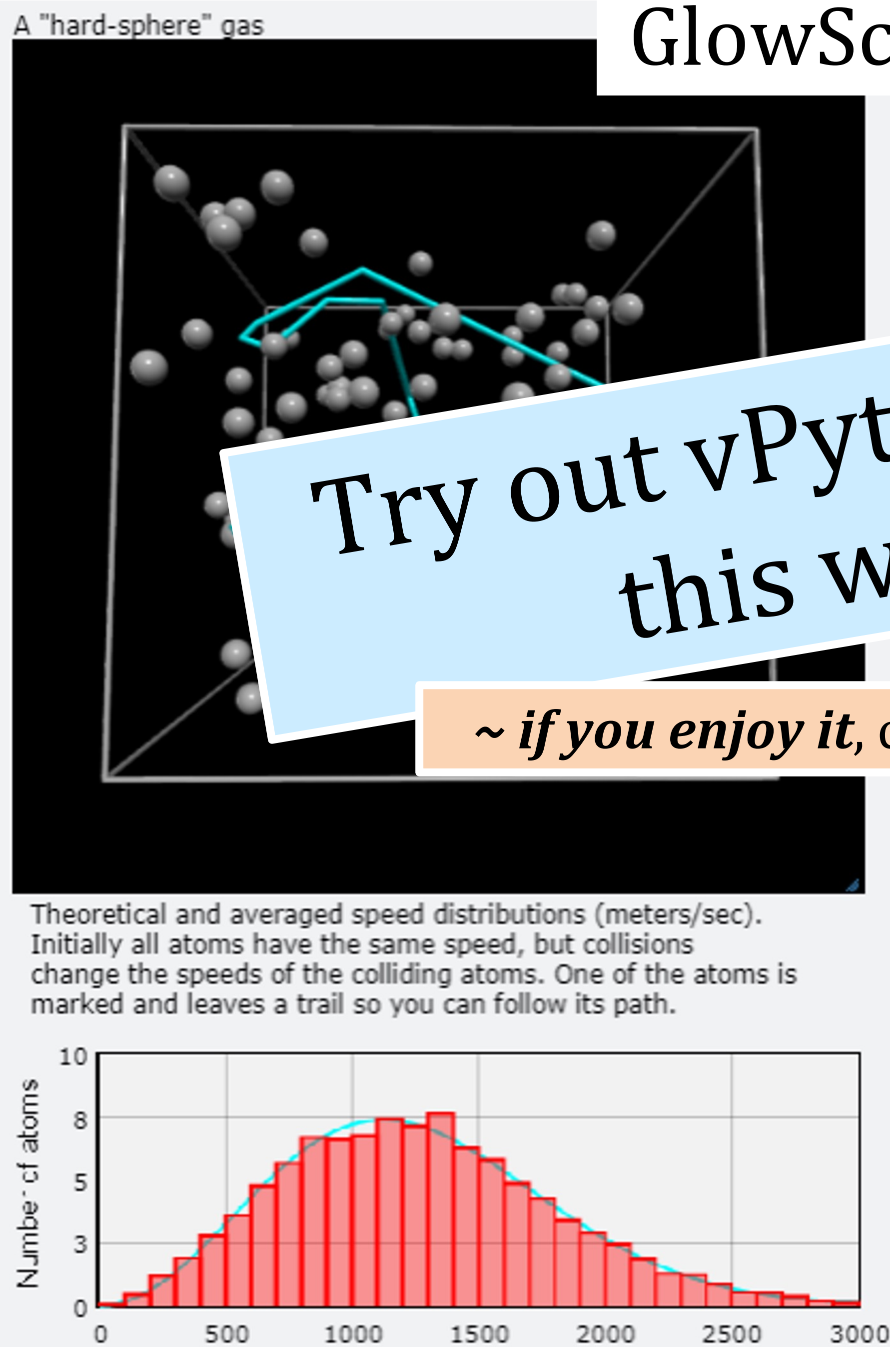


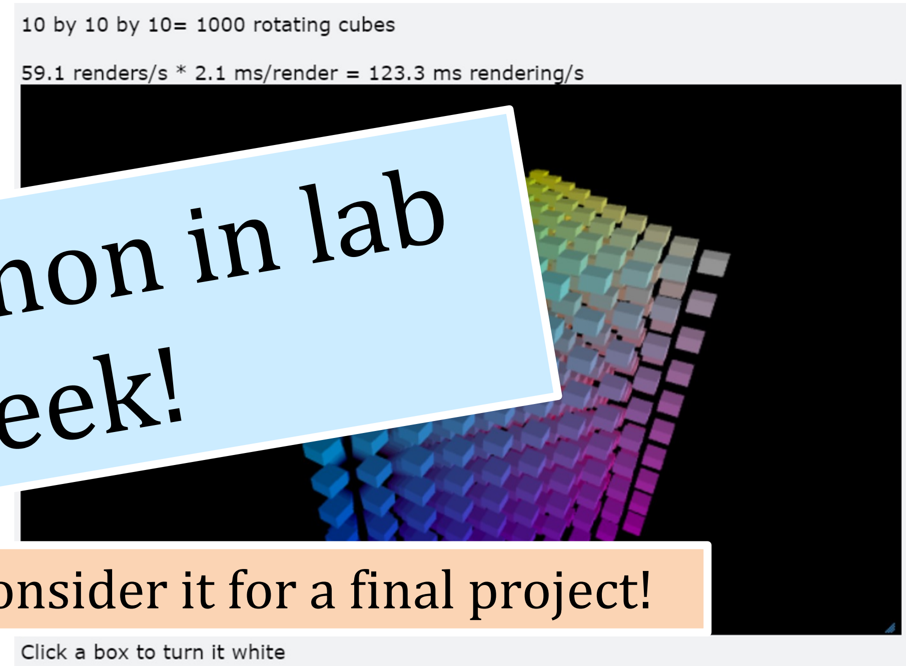




Why do I feel cornered?

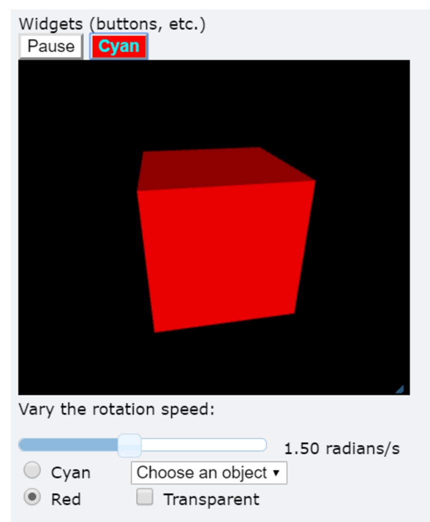
GlowScript / vPython examples…

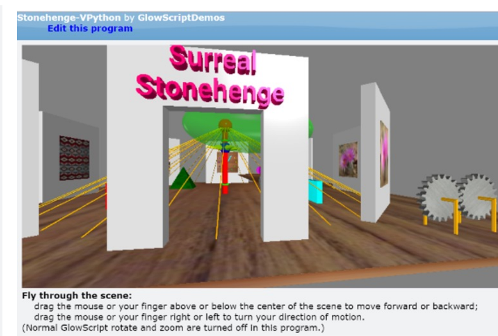




Try out vPython in lab this week!

~ ***if you enjoy it***, consider it for a final project!





Why do I feel cornered?

Looking further ahead…

How can we write a program that plays with **optimal strategy** for Connect 4?

Game AI...

Deep Blue (chess computer)

From Wikipedia, the free encyclopedia

Deep Blue was a chess-playing computer developed by IBM. On May 11, 1997, the machine, with human intervention between games, won the second six-game match against world champion Garry Kasparov by two wins to one with three draws.[1] Kasparov accused IBM of cheating and demanded a rematch, but IBM refused and dismantled Deep Blue.[2] Kasparov had beaten a previous version of Deep Blue in 1996.

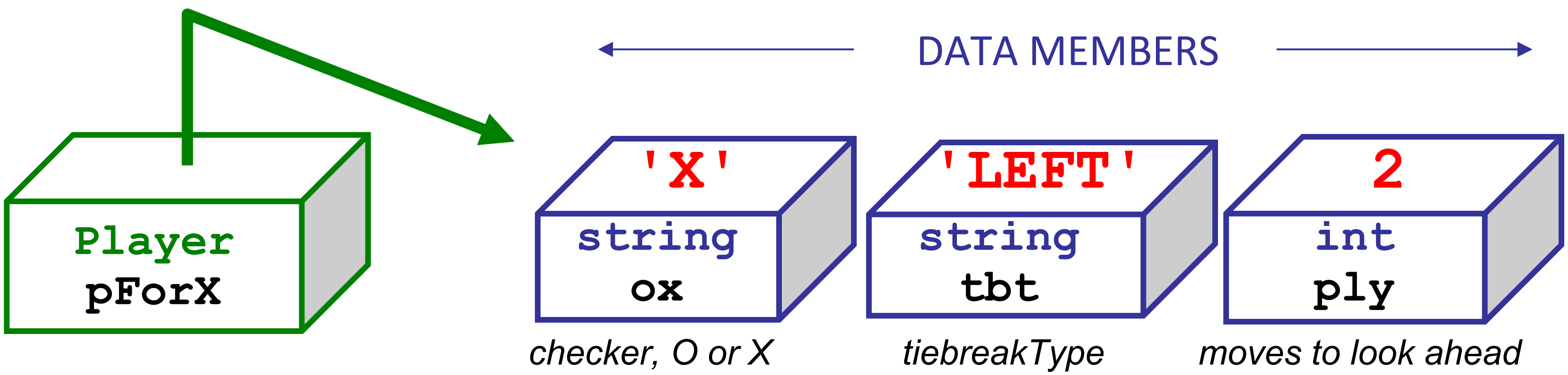
Deep Blue

Contents [hide]

Origins [edit]

The `Player` class (Final project)

What **data** does a computer AI player need?

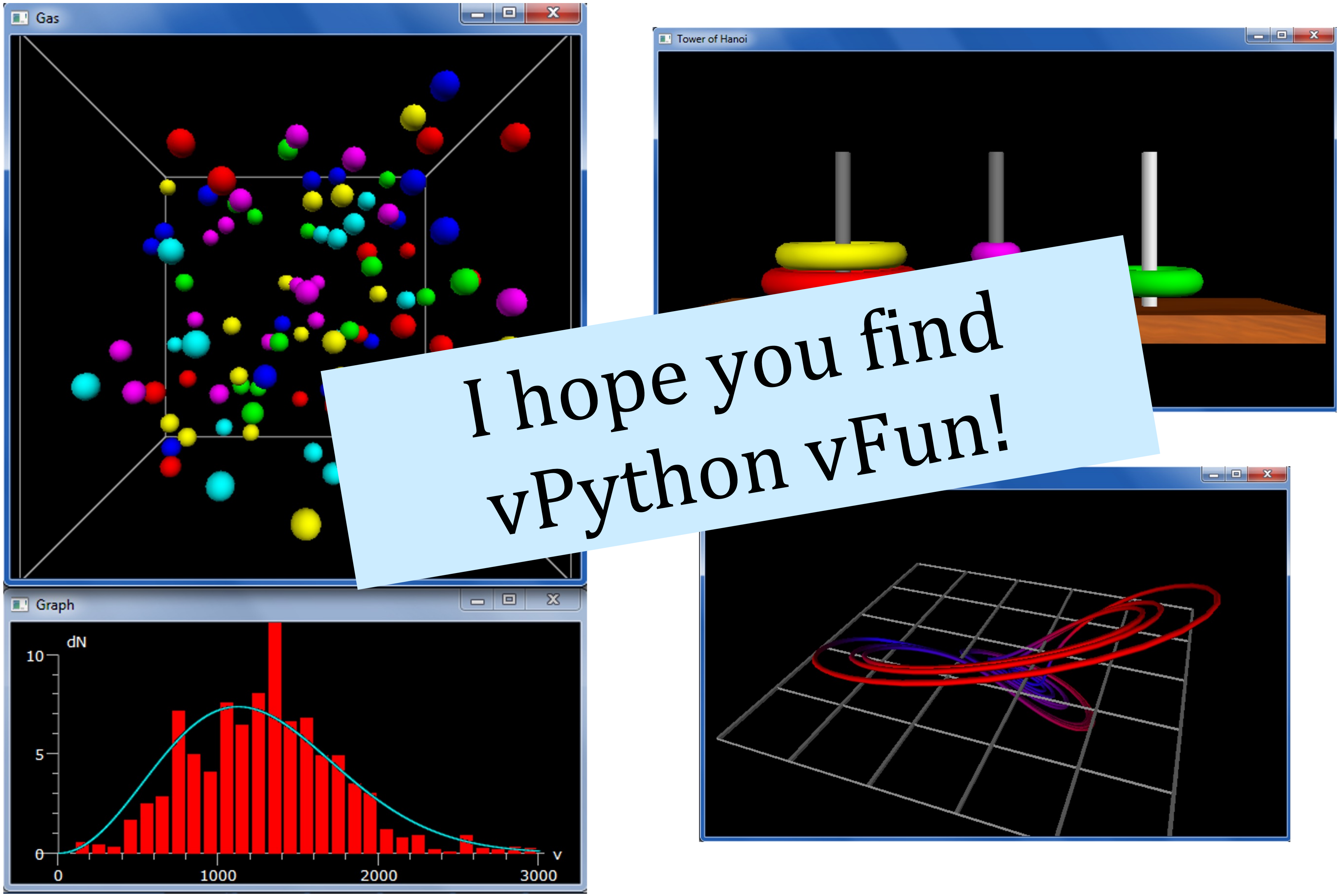


ox? tbt? ply?

x = Player('X', 'LEFT', 42)
x0rn
o0rn
b.playGame(x0rn, o0rn)

... perhaps *surprisingly, not so much.*

vPython examples…



I hope you find vPython vFun!