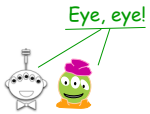
Final projects

Final CS hw
- Open-ended
- Comprehensive
- Same projects for Black/Gold
- Five choices…



Working in teams of 1-3 is OK

Teams need to work together—*at the same time*—and need to share the work equally

Teaming is extra-encouraged on the final project!

When?!

4/19
- "Start"—part of HW 11

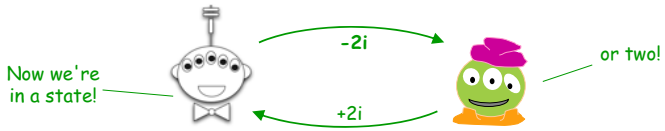
Mon. 4/26
- "Milestone" is part of HW 12
- Project-specific tasks to help with progress...

Mon. 5/3
- Final project & short reflection on how to run it and how it went.
- Due at 11:59 PM
- Euros OK; grutoring tapers.

Final ***lab*** days…

Labs **will** meet the last 2 weeks of class, *but...*

- They're "***extra-optional***": no lab problem *no sign-in*
- Lab time: ***final projects*** assistance + progress
- There are ***theocomp*** problems, too (hw12)
- Plus, hw12 has up to +50 pts of extra credit!



TextGame

Varying from the C4 homework

[1] Should have a "Board": some visible game state
Doesn't really need to be a board: Jotto, Nim, Hangman all OK

[2] Should have multiple turns (per game)
Jotto, Nim, Hangman all fit this, but RPS does not (that's the starter code)

[3] Should track the human/machine rivalry
A starting point for this is provided; use or vary it as you like

[4] *Should have an AI of some sort*
The "I" need not be sophisticated: *three or more heuristics*
Key: You should be able to play vs. the machine (more than randomly)

Life+1

Building from Week 9's lab

[1] Should have a **`Life`** class: similar to C4's **`Board`**

Enable methods for analysis & members for data storage
Also, you need to visualize your code with the *Pyglet* 2D library

[2] Should allow *any* "Life-like" rules

Python dictionaries, e.g. **`{'B': [3], 'S':[2, 3]}`**

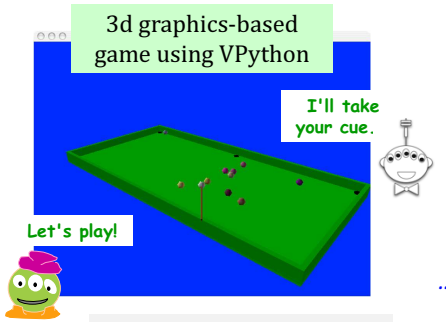
[3] Should track generations' *evolution*

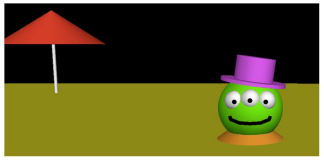
Grow? Fade? What percent of the world is alive?

[4] Should create & explore your own variation(s)

Can follow more Birth/Survival rule sets, add more states, or something completely different…

vPool





… it's not really very constrained at all!

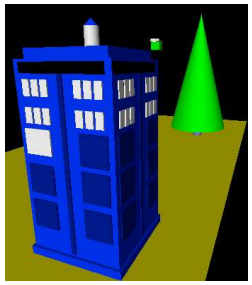
A few constraints…

Need ≥4 physically interacting objects

Allow the user to direct 1+ objects, either by keyboard or mouse or both

Needs a game goal + be winnable!

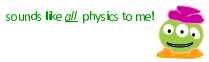
Must detect **some** "linear" and some "spherical" collisions, and implement their results on the motion of the objects



The vPool project

VPython was designed to make 3d physics simulations simpler to program – as a result, the library itself is physics-free!

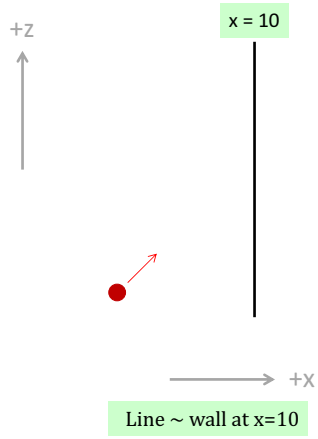
⇒ Phunky Physics is welcome!



- ***Linear collisions*** should be somewhere ("walls")
- ***Spherical collisions*** should be somewhere ("points")
- You need "pockets"—*or some other game objective*
- You need **user control** of at least one object (mouse or keyboard)

vPython: Linear collisions

At least some of the game needs to be about *detecting collisions* and *changing velocities*



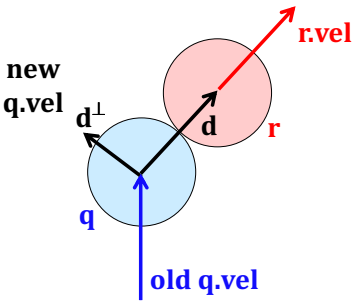

```
ball.vel = vector(1,0,1)
while True:
    ball.pos += dt*ball.vel
    if ball.pos.x > 10:
        ball.vel.x
        ball.pos.x
```

Line ~ wall at x=10

How to **bounce**?

What else to do?

Spherical collisions



1 First approximation:

Stop **q**. *Undo any overlap.*

Make **r**.vel = **q**.vel.

2 Second approximation:

Stop **q**. *Undo any overlap.*

Compute **d** = **r**.pos – **q**.pos

Make **r**.vel = **d**

3 Third approximation:

Same as **second**, but

Make **q**.vel = **d**$^{\perp}$, at 90° from **d**

Reality is just one eye away!

the TextID project

Big ideas:

(1) Build ***lexical*** models of bodies of text…

(2) *Use a similarity score to measure*

Rowlingness vs. ***Shakepearity***

NYTimes-iness vs. ***WSJournalicity***

Big Bang Theory vs. ***Modern Family***

Even better*: your own choice of two or more comparisons…*

```
I like poptarts and 42 and spam. Spamful poptarts are
like poptartful spams -- and are liked by all!
Will _Thanksgiving_ bring spam poptarts?
```
File

`class TextModel`

… contains at least five Python dictionaries, e.g.,

count of each word

{'and': 3, 'poptartful': 1, 'liked': 1, 'spamful': 1, 'like': 2, '': 1, 'spam': 2, 'i': 1, '42': 1, 'all': 1, 'thanksgiving': 1, 'will': 1, 'bring': 1, 'poptarts': 3, 'spams': 1, 'by': 1, 'are': 2}

{0: 1, 1: 1, 2: 2, 3: 6, 4: 5, 5: 3, 7: 1, 8: 3, 10: 1, 12: 1} ???

{'and': 3, '': 1, 'all': 1, 'like': 3, 'thanksgiv': 1, 'spam': 4, 'i': 1, '42': 1, 'by': 1, 'will': 1, 'bring': 1, 'are': 2, 'poptart': 4} ???

{12: 1, 5: 1, 7: 1} ???

??? {'!': 1, '-': 2, '?': 1, '_': 2, '.': 1}

What are these four other dictionaries counting?!

TextID's *building blocks…*

This looks familiar…

(1) Get text from file…

(2) Split up the text into words (first pass)

(3) Model punctuation marks (optional)

(4) Model sentence lengths (using '.!?')

Text with punctuation preserved

(5) Take a breather…

(6) Clean up the words (second pass)

(7) Model words and word lengths

(8) Stem words and model those stems

Text with punctuation removed

(9) You're ready to score against your model!

TextID's *library resources*

7.1. `string` — Common string operations

5.6.1. String Methods

`string.` **`punctuation`**
String of ASCII characters which are considered punctuation

`str.` **`replace`**(*old*, *new*[, *count*])
Return a copy of the string with all occurrences of substring *old* replaced by *new*.

`str.` **`lower`**()
Return a copy of the string with all the cased characters [4] converted to lowercase.

`str.` **`split`**([*sep*[, *maxsplit*]])
Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

Stemming

An algorithm that outputs the **root** of the input word.

`stem('parties')` *ends in ies*	⇨	`'parti'`
`stem('love')` *ends in vow + cons + e*	⇨	`'lov'`
`stem('swerving')` *ends in vow + cons + ing*	⇨	`'swerv'`
`stem('quickly')` *ends in vow + cons + ly*	⇨	`'quick'`
`stem('slowest')` *ends in vow + cons + est*	⇨	`'slow'`
`stem('stems')` *ends in cons + s*	⇨	`'stem'`
`stem('stemming')` *ends in vow + 2*cons + ing*	⇨	`'stem'`

these don't have to be words, just **stems**...

but, they ***can*** also be words

Either way, they ***all*** have exceptions! ⇨

Model *matching*

Suppose we have two ***trained models***:

WS: { "love": 50,
"spell": 8,
"thou": 42 }

JKR: { "love": 25,
"spell": 275,
"potter": 700 }

normalize for size

WS: { "love": 0.50,
"spell": 0.08,
"thou": 0.42 }

JKR: { "love": 0.025,
"spell": 0.275,
"potter": 0.700 }

Unknown text: { "love": 3,
"thou": 1,
"potter": 2,
"spam": 4 }

Model *matching*

Suppose we have two ***normalized models***:

WS: { "love": 0.50,
"spell": 0.08,
"thou": 0.42 }

JKR: { "love": 0.025,
"spell": 0.275,
"potter": 0.700 }

The ***WS***-based probability of each word in ***Unknown text***

For missing words, use **half** the smallest value – *across both normalized models!*

3*log(.50) + log(.42) + 2*log(.012) + 4*log(.012) = -29.48

love love love thou potter potter spam spam spam spam

3 1 2 4

OK!

Take logs of everything!

Unknown text: { "love": 3, "potter": 2,
"thou": 1, "spam": 4 }

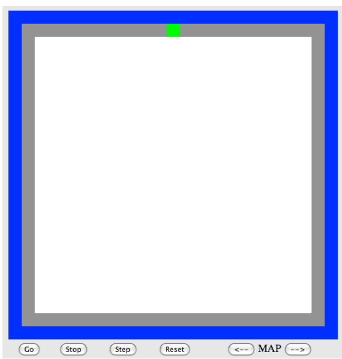
half of ϵ !

The Picobot project

Big ideas

(1) Implement Picobot in Python

(2) *Train Python to write successful Picobot programs!*



Talk about going *full circle...*

Picobot's classes

```
class Program:
```

What in Python could most usefully hold all of these *rules*?

What type should `self.rules` be?

Python dictionary

```
0 xxxx -> N 0
0 Nxxx -> W 0
0 NxWx -> S 0
0 xxWx -> S 0
0 xxWS -> E 0
0 xxxS -> E 0
0 xExS -> N 0
0 xExx -> N 0
0 NExx -> S 1
1 xxxx -> S 1
1 Nxxx -> E 1
1 NxWx -> E 1
1 xxWx -> N 1
1 xxWS -> N 1
1 xxxS -> W 1
1 xExS -> W 1
1 xExx -> S 1
1 NExx -> W 0
```

both *tuples*

key value

```
self.rules[ (1,"NExx") ] = ("W",0)
```

Picobot's classes

What type in Python could most usefully hold the *environment*?

```
class World:
```

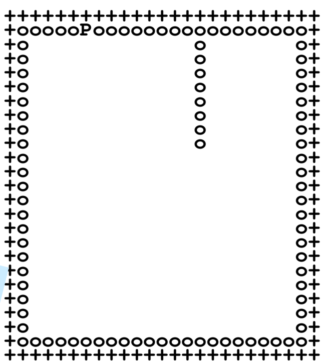
```
class Board
```

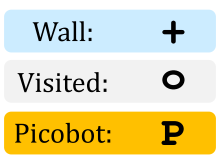
What class that you've already written will be most similar to Picobot's **World**?

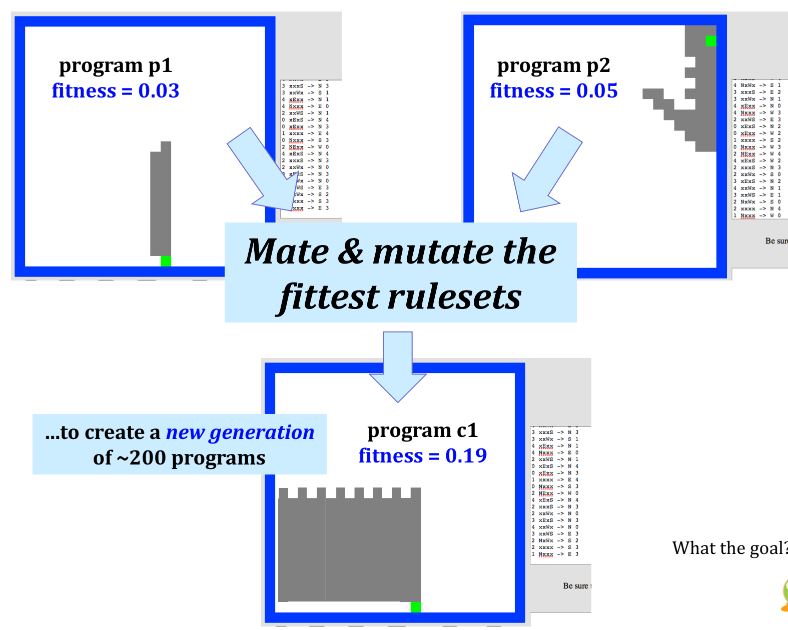
What will `self.room` be?

The same as the Connect-Four board's `self.data`!

A list-of-lists-of-one-character-strings....







What the goal?