Graphics Transformations and Layers

CS11 Computer Programming
Department of Computer Science
Wellesley College

A closer look at cs1graphics objects:
- An object is a data value that has state and behaviors. Examples of objects are cs1graphics canvases, circles, squares, texts, images, points, and layers.

Examples of object state:
- `paper = Canvas(800, 500, 'cyan', 'ocean')`
- `circ = Circle(50, Point(200, 100))`
- `rect = Rectangle(75, 250)`

<table>
<thead>
<tr>
<th>Width</th>
<th>Height</th>
<th>Background Color</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>500</td>
<td>'cyan'</td>
<td>'ocean'</td>
</tr>
</tbody>
</table>

Tools: graphicsState & cs1graphicsHelper

```python
from cs1graphics import *
from graphicsState import *

example = Canvas(300,200,'azure3')
dot = Circle(25,Point(0,0))
example.add(dot)
printState(dot)
dot.setFillColor('magenta')
printState(dot)
dot.moveTo(200,75)
printState(dot)
```

Cloning

We can make a copy of a drawable object using the `clone` method.

```python
wedge = Polygon(Point(0,100), Point(0,0), Point(200,100))
wedge.setFillColor('green')
wedge.moveTo(100,200)
wedge2 = wedge.clone()
wedge2.moveTo(525,200)
wedge3 = wedge.clone()
wedge3.moveTo(100,425)
wedge4 = wedge.clone()
wedge4.moveTo(525,425)
wedge4.setFillColor('orange')
```

* On this and subsequent slides, to avoid clutter we omit invocations of the Canvas add() method for each of the Polygon objects, e.g. `paper.add(wedge), paper.add(wedge2), etc.`
Scaling

We can change the size of a drawable object using the `scale` method.

```
wedge = Polygon(Point(0,100), Point(0,0), Point(200,100))
wedge.setFillColor('green')
wedge.moveTo(100,200)
```

```
wedge2 = wedge.clone()
wedge2.moveTo(525,200)
wedge2.scale(-1)
```

```
wedge3 = wedge.clone()
wedge3.moveTo(100,425)
wedge3.scale(2)
wedge4 = wedge.clone()
wedge4.moveTo(525,425)
wedge4.scale(0.5)
```

Rotating

We can rotate a drawable object using the `rotate` method.

```
```

```
wedge = Polygon(Point(0,100), Point(0,0), Point(200,100))
wedge.setFillColor('green')
wedge.moveTo(100,200)
```

```
wedge2 = wedge.clone()
wedge2.moveTo(525,200)
wedge2.rotate(180)
```

```
wedge3 = wedge.clone()
wedge3.moveTo(100,425)
wedge3.rotate(30)
```

```
wedge4 = wedge.clone()
wedge4.moveTo(525,425)
wedge4.rotate(-45)
```

Flipping

We can flip a drawable object around an axis using the `flip` method.

```
```

```
wedge = Polygon(Point(0,100), Point(0,0), Point(200,100))
wedge.setFillColor('green')
wedge.moveTo(100,200)
```

```
wedge2 = wedge.clone()
wedge2.moveTo(525,200)
wedge2.flip(0)
```

```
wedge3 = wedge.clone()
wedge3.moveTo(100,425)
wedge3.flip(90)
```

```
wedge4 = wedge.clone()
wedge4.moveTo(525,425)
wedge4.flip(45)
```

Uranium Flag

`uraniumFlag.py` creates the following picture:

Modify the file so that it creates this picture:
A better way: Abstraction!

Layers capture patterns

Drawing the three parts (yellow body, green fin, and black eye) of every fish over and over again would be tedious, and getting the coordinates right for all the scalings, rotations, and flippings would be extremely challenging!

Fortunately, there is a better way! We can abstract over the notion of a fish pattern by creating a `Layer` object, which you can think of as a mini-canvas for grouping together other shapes.

We can add items to a Layer just like we can add items to a canvas. The "push pin" reference point of a Layer is \((0,0)\) in its own coordinate system.

```
# fish = Layer()

# yellow body of the fish
body = Ellipse(100, 50, Point(0, 0))
body.setFillColor('yellow')
fish.add(body)

# green tail of the fish
tail = Polygon(Point(-50, 0),
                Point(-75, 25),
                Point(-75, -25))
tail.setFillColor('green')
fish.add(tail)

# black eye of the fish
eye = Circle(5, Point(25, -5))
eye.setFillColor('black')
fish.add(eye)
```
A tank with one fish

```ruby
# Add fish to a canvas
tank = Canvas(600, 400, 'skyblue', 'Where is Dory?')
tank.add(fish)
fish.moveTo(100, 50)
```

Cloning a Layer: a second fish

```ruby
# Add fish to a canvas
tank = Canvas(600, 400, 'skyblue', 'Where is Dory?')
tank.add(fish)
fish.moveTo(100, 50)
```

```ruby
# Add a second, bigger fish
fish2 = fish.clone()
tank.add(fish2)
fish2.moveTo(350, 100)
fish2.scale(2)
```

Populating the tank

```ruby
# Add fish to a canvas
tank = Canvas(600, 400, 'skyblue', 'Where is Dory?')
tank.add(fish)
fish.moveTo(100, 50)
```

```ruby
# Add a second, bigger fish
fish2 = fish.clone()
tank.add(fish2)
fish2.moveTo(350, 100)
fish2.scale(2)
```

```ruby
# Third fish
fish3 = fish.clone()
tank.add(fish3)
fish3.moveTo(550, 75)
fish3.scale(0.5)
```

```ruby
# Fourth fish
fish4 = fish.clone()
tank.add(fish4)
fish4.moveTo(125, 225)
fish4.scale(2)
fish4.flip(0)
```

```ruby
# Fifth fish
fish5 = fish.clone()
tank.add(fish5)
fish5.moveTo(250, 350)
fish5.rotate(30)
```

```ruby
# Sixth fish
fish6 = fish.clone()
tank.add(fish6)
fish6.moveTo(450, 275)
fish6.scale(1.5)
fish6.rotate(-30)
fish6.flip(0)
```

The power of abstraction

Suppose we want to add pink hats to all of our fish.

Do we want to change each fish individually? No! We need only change the prototypical fish (the original fish prior to any cloning).

```ruby
# Add pink hat *before* any clones are made
hat = Polygon(Point(21, -21), Point(-10, -14), Point(-23, -36), Point(4, -30), Point(30, -46))
hat.setFillColor('pink')
fish.add(hat)
```

Now all our fishies have pink hats.

This illustrates the power of abstraction. When we abstract over a pattern, then a change to the pattern affects all instantiations of the pattern.
Drawbacks of Layers

Although Layers are powerful, they do not let us abstract over all the properties of our fish that we might want to change.

What if we want different fish to have different colors?

What if we want different fish to have larger or smaller eyes?

We cannot express these differences with Layers. Why not?

But we can express them with user-defined functions, a more powerful abstraction mechanism that we will study next time.

Digging Deeper

In this course we learn problem solving and happen to use Python to write our solutions.

Python has many details that initially seem daunting. Although you might not need to know them right now, you might need them eventually, so we want to make you aware of them. We’ll put them in slides marked “Digging Deeper”:

Come back to this material whenever you want to understand Python details you might not have cared about earlier.

Classes

- A class is a description of the shared characteristics (state and behaviors) of a set of objects.
- A class is like a mold for making objects.
- Example of classes include:
  - Canvas
  - Rectangle
  - Point
  - Polygon
  - Circle
  - Text
  - Image
  - Layer
- An object is made from a class by calling the constructor function with the same name as the class. E.g.,
  - Canvas(900, 600, 'cyan', 'sky')
  - Rectangle(75, 300)
  - Circle(50, Point(200, 100))
- Each objects made from a class is an instance of the class.

Contracts

- Every method/function has a contract or Application Program Interface (API) that specifies the behavior of the method/function.
- Every class has a contract/API for all of its methods.
- Any user of a method/function/class can expect that it will behave as described in the contract.
- Any implementer of the method/function/class must ensure that it fulfills the contract.
- Example of contracts include:
  - csilgraphics API
  - Python built-in functions API
  - math module API
The Canvas Contract

This is just the names of the methods and their arguments in the Canvas class contract. The full Contract specifies what each method does.

Inheritance

Some classes are generalizations of another class. We arrange classes in an inheritance hierarchy in which more specific subclasses below inherit state and behavior from more general superclasses above.