Divide, Conquer, and Glue with Pictures

Recall big idea #3: Divide, conquer & glue (DCG)

Divide
problem P into subproblems.

Conquer
each of the subproblems, &

Glue (combine)
the solutions to the subproblems
into a solution S for P.

DCG to make cool pictures
Today we’ll see how to use DCG to make complex and interesting pictures in a simple way.

But first it will help to define a more abstract notion of “picture” starting with cs1graphics.

Recall big idea number 1: Abstraction

User / Client

Implementer / Designer

Contract / API
Peter Henderson’s Picture Language

Henderson's Functional Geometry paper:
http://eprints.soton.ac.uk/257577/1/funcgeo2.pdf

DCG with Pics 5

Picture abstraction on top of cs1graphics

- Inspired by Peter Henderson’s picture language.
- We define a “picture” as a cs1graphics Drawable object that’s 200x200 centered at reference point (0,0).
- By fixing the size of a picture, we never have to worry about pictures of different sizes.
- Rotations and flips of pictures by angles that are multiples of 90 degrees results in another picture!
- We will define many functions that take picture(s) as inputs and return a picture as an output.

DCG with Pics 6

Some Primitive Pictures

bp (blue patch)

rp (red patch)

gw (green wedge)

Sample primitives defined in Python [1]

```python
def patch(color):
    pic = Square(200)
    pic.setFillColor(color)
    pic.setBorderColor('black')
    return pic

rp = patch('red')

def wedge(color):
    pic = Polygon(Point(100, 0),
                   Point(100,100),
                   Point(-100, 100))
    #Shift reference point from (100,0) to (0,0)
    pic.adjustReference(-100,0)
    pic.setFillColor(color)
    pic.setBorderColor(color)  # no border!
    return pic

gw = wedge('green')
```

DCG with Pics 7

DCG with Pics 8
Sample primitives defined in Python [2]

def checkmark(downColor, upColor):
    pic = Layer()
    downstroke = Path(Point(-100,0), Point(0,100))
    downstroke.setBorderColor(downColor)
    pic.add(downstroke)
    upstroke = Path(Point(0,100), Point(100,-100))
    upstroke.setBorderColor(upColor)
    pic.add(upstroke)
    return pic

mark = checkmark('red', 'blue')

def empty():
    return Layer()

empty = empty()

Clockwise rotations of pictures

clockwise90(p): # Returns new picture that's p rotated 90° clockwise

clockwise180(p): # Returns new picture that's p rotated 180° clockwise

clockwise270(p): # Returns new picture that's p rotated 270° clockwise

Clockwise90(gw)
clockwise180(gw)
clockwise270(gw)

Displaying Pictures

def displayPic(pic):
    '''Display picture in 600x600 canvas'''
    frame = Canvas(600, 600, 'white', 'Picture Frame')
    # Clone pic before changing it; otherwise
    # it wouldn't be a picture anymore!
    framedPic = pic.clone()
    framedPic.scale(3) # scale by 3 to fill 600x600 canvas
    framedPic.moveTo(300, 300) # move to center of canvas
    frame.add(framedPic)

* The definition of displayPic is a tad more complex than shown here to allow
  for a closeAllPics() function that closes all canvases created by displayPic.

Defining picture rotations in Python

def clockwisePic(pic, angle):
    newPic = pic.clone() # create new pic by cloning it.
    newPic.rotate(angle) # if angle is a multiple of 90,
                          # result still satisfies
                          # definition of picture.
    return newPic

def clockwise90(pic):
    return clockwisePic(pic, 90)

def clockwise180(pic):
    return clockwisePic(pic, 180)

def clockwise270(pic):
    return clockwisePic(pic, 270)
Flipping pictures

- `flipAcrossVert(p)` returns a new picture that’s `p` flipped across the vertical axis.
- `flipAcrossHoriz(p)` returns a new picture that’s `p` flipped across the horizontal axis.
- `flipAcrossDiag(p)` returns a new picture that’s `p` flipped across a 45-degree axis.

```
def flipPic(pic, angle):
    newPic = pic.clone()  # create new pic by cloning it.
    newPic.flip(angle)    # if angle is a multiple of 45, result still satisfies definition of picture.
    return newPic
```

```
def flipAcrossVert(pic):
    return flipPic(pic, 0)

def flipAcrossHoriz(pic):
    return flipPic(pic, 90)

def flipAcrossDiag(pic):
    return flipPic(pic, 45)
```

Overlaying pictures

```
def overlay(pic1, pic2):
    '''Returns a new pic in which pic1 appears on top of pic2.'''
    newPic = Layer()
    newPic.add(pic2)  # bottom pic goes first
    newPic.add(pic1)  # top pic goes last
    return newPic
```

```
def overlay(mark, gl):
    overlay(gl, mark)
```

fourPics: Combining four pictures

```
def fourPics(a, b, c, d):
    newPic = Layer()
    aHalf = a.clone()  # half a
    aHalf.scale(0.5)
    bHalf = b.clone()  # half b
    bHalf.scale(0.5)
    cHalf = c.clone()  # half c
    cHalf.scale(0.5)
    dHalf = d.clone()  # half d
    dHalf.scale(0.5)
    aHalf.move(-50, -50)
    bHalf.move(50, -50)
    cHalf.move(-50, 50)
    dHalf.move(50, 50)
    newPic.add(aHalf)
    newPic.add(bHalf)
    newPic.add(cHalf)
    newPic.add(dHalf)
    return newPic
```

```
def fourPics(bp, gw, mark, rp):
    fourPics(bp, gw, mark, rp)
```
**fourSame: Combining four copies of one picture**

```python
def fourSame(pic):
    return fourPics(pic, pic, pic, pic)
```

**Repeated tiling**

```python
def tiling(pic):
    return fourSame(fourSame(fourSame(fourSame(pic)))))
```

**How to make a checkerboard?**

```python
def checkerboard(color1, color2):
    return fourSame(fourSame(fourPics(patch(color1), patch(color2),
                                patch(color2), patch(color1))))
```

**DCG on checkerboard**

```python
def checkerboard(color1, color2):
    return ???
```
Combining four rotations of a picture

```python
def rotations(pic):
    return fourPics(clockwise270(pic), pic, clockwise180(pic), clockwise90(pic))

def rotations2(pic):
    return fourPics(pic, clockwise90(pic), clockwise180(pic), clockwise270(pic))
```

```
rotations(gw)
rotations2(gw)
```

A simple recipe for complexity

```python
def wallpaper(pic):
    return rotations(rotations(rotations(rotations(pic)))))

def design(pic):
    return rotations2(rotations2(rotations2(rotations2(pic))))
```

```
wallerp(mark)
design(mark)
```

A quilt problem

How do we build this complex quilt …

… from simple primitive pictures like this?

```python
triangles('green', 'blue')
patch('red')
```

Divide the quilt into subproblems
Conquer the subproblems using “wishful thinking”

clockwise270(quadrant())  quadrant()
clockwise180(quadrant())  clockwise90(quadrant())

Glue the subsolutions to solve the problem

def quilt():
    return fourPics(clockwise270(quadrant()),
                    quadrant(),
                    clockwise180(quadrant()),
                    clockwise90(quadrant()))

Abstracting over the glue

def quilt():
    return rotations(quadrant())
def rotations(pic):
    # picture function from before
    return fourPics(clockwise270(pic), pic,
                    clockwise180(pic), clockwise90(pic))

Subproblem: quadrant()

def quadrant():
    return corner(star('yellow', 'red', 'blue'),
                  star('red', 'green', 'blue'))
def corner(llPic, outerPic):
    return ?
Continue the descent …

```
def star(innerColor, middleColor, outerColor):
    return rotations(starQuadrant(innerColor, middleColor, outerColor))
```

```
def starQuadrant('red', 'green', 'blue'):
```

```
```

And descend some more …

```
def starQuadrant(squareColor, llTriColor, urTriColor):
    return ?
```

```
```

… until we reach primitives

```
def quilt():
    return rotations(quadrant())
```

```
def quadrant():
    return corner(star('yellow', 'red', 'blue'),
                  star('red', 'green', 'blue'))
```

```
def star(innerColor, middleColor, outerColor):
    return rotations(starQuadrant(innerColor, middleColor, outerColor))
```

```
def corner(llPic, outerPic):
    return fourPics(outerPic, outerPic, llPic, outerPic)
```

```
def starQuadrant(squareColor, llTriColor, urTriColor):
    return corner(patch(squareColor),
                   triangles(llTriColor, urTriColor))
```

```
def patch('red')
    return
```

```
def triangles('green', 'blue')
    return
```

All together now

```
def quilt():
    return rotations(quadrant())
```

```
def quadrant():
    return corner(star('yellow', 'red', 'blue'),
                  star('red', 'green', 'blue'))
```

```
def star(innerColor, middleColor, outerColor):
    return rotations(starQuadrant(innerColor, middleColor, outerColor))
```

```
def corner(llPic, outerPic):
    return fourPics(outerPic, outerPic, llPic, outerPic)
```

```
def starQuadrant(squareColor, llTriColor, urTriColor):
    return corner(patch(squareColor),
                   triangles(llTriColor, urTriColor))
```
Abstracting over quilt colors

How to generalize \texttt{quilt} to define \texttt{quiltColors}?

\begin{itemize}
  \item \texttt{quiltColors('yellow', 'red', 'green', 'blue')}
  \item \texttt{quiltColors('red', 'blue', 'magenta', 'cyan')}
\end{itemize}

Challenge

How do we build this picture …

… starting with this primitive?

\begin{itemize}
  \item \texttt{autumnLeaves()}
  \item \texttt{twoLeaves('orange')}
\end{itemize}