Eric Roberts
CSCI 121

Handout #15
September 19, 2018

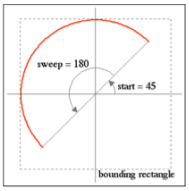
Graphical Structures

Graphical Structures

Eric Roberts
CS 121
September 19, 2018

The `GArc` Class

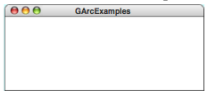
- The `GArc` class represents an arc formed by taking a section from the perimeter of an oval.
- Conceptually, the steps necessary to define an arc are:
  - Specify the coordinates and size of the bounding rectangle.
  - Specify the ***start angle***, which is the angle at which the arc begins.
  - Specify the ***sweep angle***, which indicates how far the arc extends.
- The geometry used by the `GArc` class is shown in the diagram on the right.
- In keeping with the graphics model, angles are measured in degrees starting at the +*x* axis (the 3:00 o'clock position) and increasing counterclockwise.
- Negative values for the *start* and *sweep* angles signify a clockwise direction.



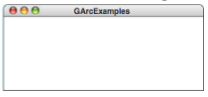
Exercise: `GArc` Geometry

Suppose that the variables `cx` and `cy` contain the coordinates of the center of the window and that the variable `d` is 0.8 times the screen height. Sketch the arcs that result from each of the following code sequences:

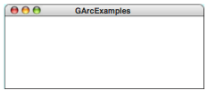
```
a1 = GArc(d, d, 0, 90)
gw.add(a1, cx - d / 2, cy - d / 2)
```



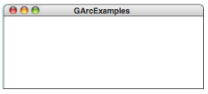

```
a2 = GArc(d, d, 45, 270)
gw.add(a2, cx - d / 2, cy - d / 2)
```




```
a3 = GArc(d, d, -90, 45)
gw.add(a3, cx - d / 2, cy - d / 2)
```




```
a4 = GArc(d, d, 0, -180)
gw.add(a4, cx - d / 2, cy - d / 2)
```



Filled Arcs

- The `GArc` class implements the methods `setFilled` and `setFilledColor`.
- A filled `GArc` is displayed as the pie-shaped wedge formed by the center and the endpoints of the arc, as follows:

```
def FilledEllipticalArc():
   gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
   arc = GArc(0, 0, gw.getWidth(), gw.getHeight(), 0, 90)
   arc.setFilled(True)
   gw.add(arc)
}
```

Additional Methods for `GArc`

`setStartAngle(`*start*`)`	Sets the start angle for the arc
`getSweepAngle()`	Returns the start angle for the arc
`setSweepAngle(`*sweep*`)`	Sets the sweep angle for the arc
`getSweepAngle()`	Returns the sweep angle
`setFrameRectangle(`*x, y, width, height*`)`	Resets the bounds for the frame

- These methods allow you to animate the appearance of an arc.
- The `setStartAngle` and `setSweepAngle` methods make it possible to change the starting position and the extent of the arc dynamically.
- The `setFrameRectangle` method changes the bounds of the rectangle circumscribing the oval from which the arc is taken.

Exercise: PacMan

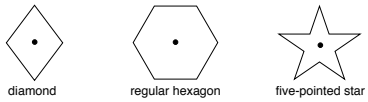
- Write a program that uses the `GArc` class to display a PacMan figure at the left edge of the graphics window.
- Add the necessary timer animation so that PacMan moves to the right edge of the window. As it moves, your program should change the start and sweep angles of the arc so that the mouth appears to open and close.

Questions about the PacMan Problem

- We're going to divide into groups and spend the next five minutes discussing important questions you would need to answer while solving the PacMan problem. Each group will discuss one of the following four questions:
  1. How would you create the initial PacMan object at the left of the window?
  2. What needs to happen on each time step?
  3. How do you get the program to stop?
  4. How would you design milestones that would allow you to test the program in pieces?

The `GPolygon` Class

- The `GPolygon` class is used to represent graphical objects bound by line segments. In mathematics, such figures are called ***polygons*** and consist of a set of ***vertices*** connected by ***edges***. The following figures are examples of polygons:



- Unlike the other shape classes, that location of a polygon is not fixed at the upper left corner. What you do instead is pick a ***reference point*** that is convenient for that particular shape and then position the vertices relative to that reference point.
- The most convenient reference point is usually the geometric center of the object.

Constructing a `GPolygon` Object

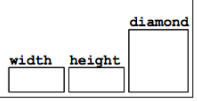
- The `GPolygon` function creates an empty polygon. Once you have the empty polygon, you then add each vertex to the polygon, one at a time, until the entire polygon is complete.
- The most straightforward way to create a `GPolygon` is to call the method `addVertex(`*x*`, `*y*`)`, which adds a new vertex to the polygon. The *x* and *y* values are measured relative to the reference point for the polygon rather than the origin.
- When you start to build up the polygon, it always makes sense to use `addVertex(`*x*`, `*y*`)` to add the first vertex. Once you have added the first vertex, you can call any of the following methods to add the remaining ones:
  - `addVertex(`*x*`, `*y*`)` adds a new vertex relative to the reference point
  - `addEdge(`*dx*`, `*dy*`)` adds a new vertex relative to the preceding one
  - `addPolarEdge(`*r*`, `*theta*`)` adds a new vertex using polar coordinates

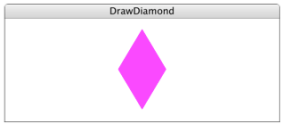
Using `addVertex` and `addEdge`

- The `addVertex` and `addEdge` methods each add one new vertex to a `GPolygon` object. The only difference is in how you specify the coordinates. The `addVertex` method uses coordinates relative to the reference point, while the `addEdge` method indicates displacements from the previous vertex.
- Your decision about which of these methods to use is based on what information you have readily at hand. If you can easily calculate the coordinates of the vertices, `addVertex` is probably the right choice. If, however, it is easier to describe each edge, `addEdge` is probably a better strategy.
- No matter which of these methods you use, the `GPolygon` class closes the polygon before displaying it by adding an edge from the last vertex back to the first one, if necessary.

`DrawDiamond` using `addVertex`

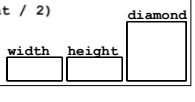
```
def createDiamond(width, height):
    diamond = GPolygon()
    diamond.addVertex(-width / 2, 0)
    diamond.addVertex(0, height / 2)
    diamond.addVertex(width / 2, 0)
    diamond.addVertex(0, -height / 2)
    return diamond
```

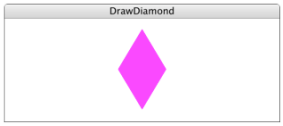




`DrawDiamond` using `addEdge`

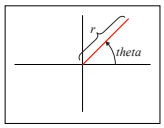
```
def createDiamond(width, height):
    diamond = GPolygon()
    diamond.addVertex(-width / 2, 0)
    diamond.addEdge(width / 2, -height / 2)
    diamond.addEdge(width / 2, height / 2)
    diamond.addEdge(-width / 2, height / 2)
    diamond.addEdge(-width / 2, -height / 2)
    return diamond
```





Using `addPolarEdge`

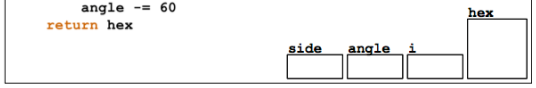
- In many cases, you can determine the length and direction of a polygon edge more easily than you can compute its *x* and *y* coordinates. In such situations, the best strategy for building up the polygon outline is to call `addPolarEdge(`*r*`, `*theta*`)`, which adds an edge of length *r* at an angle that extends *theta* degrees counterclockwise from the +*x* axis, as illustrated by the following diagram:

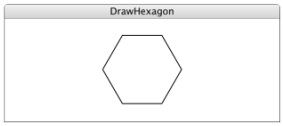


- The name of the method reflects the fact that `addPolarEdge` uses what mathematicians call ***polar coordinates***.

The `DrawHexagon` Program

```
def createHexagon(side):
    hex = GPolygon()
    hex.addVertex(-side, 0)
    angle = 60
    for i in range(6):
        hex.addPolarEdge(side, angle)
        angle -= 60
    return hex
```





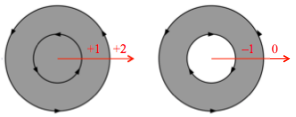
Determining the Interior of a Region

> For a simple path, it is intuitively clear what region lies inside. However, for a more complex path—for example, a path that intersects itself or has one subpath that encloses another—the interpretation of "inside" is not always obvious. . . .
>
> The *nonzero winding number rule* determines whether a given point is inside a path by conceptually drawing a ray from that point to infinity in any direction and then examining the places where a segment of the path crosses the ray. Starting with a count of 0, the rule adds 1 each time a path segment crosses the ray from left to right and subtracts 1 each time a segment crosses from right to left. After counting all the crossings, if the result is 0 then the point is outside the path; otherwise it is inside.

— *PostScript Language Reference Manual,* Adobe Systems Incorporated, 1999.

The Problem of Complex Paths

> The more interesting cases are those involving complex or self-intersecting paths. . . . For a path composed of two concentric circles, the areas enclosed by both circles are considered to be inside, provided that both are drawn in the same direction. If the circles are drawn in opposite directions, only the "doughnut" shape between them is inside, according to the rule; the "doughnut hole" is outside.



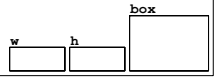
— *PostScript Language Reference Manual,* Adobe Systems Incorporated, 1999.

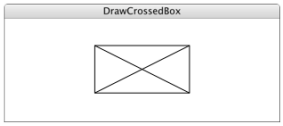
Creating Compound Objects

- The `GCompound` class in the graphics library makes it possible to combine several graphical objects so that the resulting structure behaves as a single `GObject`.
- The easiest way to think about the `GCompound` class is as a combination of a `GWindow` and a `GObject`. A `GCompound` is like a `GWindow` in that you can add objects to it, but it is also like a `GObject` in that you can add it to the graphics window.
- As was true in the case of the `GPolygon` class, a `GCompound` object has its own coordinate system that is expressed relative to a ***reference point***. When you add new objects to the `GCompound`, you use the local coordinate system based on the reference point. When you add the `GCompound` to the graphics window, all you have to do is set the location of the reference point; the individual components will automatically appear in the right locations relative to that point.

The `DrawCrossedBox` Program

```
def createCrossedBox(w, h):
    box = GCompound()
    box.add(GRect(-w / 2, -h / 2, w, h))
    box.add(GLine(-w / 2, -h / 2, w / 2, h / 2))
    box.add(GLine(-w / 2, h / 2, w / 2, -h / 2))
    return box
```





National Cancer Institute in Panama

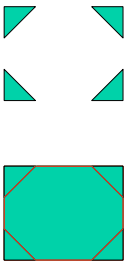
- Although the Therac-25 case is by far the best documented case of a software failure in a medical system, there are other cases that had greater costs in terms of loss of life.
- In November 2000, 28 patients at the *Instituto Oncológico Nacional* (ION) in Panama City received significant overdoses resulting from data entry errors on the institute's Treatment Planning System. By 2004, 21 of those patients had died.
- In addition to being more deadly in its effects, the incident in Panama is fascinating because understanding the details of the software problem requires familiarity with the subtleties of graphical algorithms for recognizing the interior of a polygon.
- I learned about this incident a few years ago when one of my students, Andrew Cooper, wrote his paper on this topic.

Shielding Blocks

- The purpose of the Treatment Planning System (TPS) is to help doctors determine the radiation doses and treatment times appropriate to each cancer case, which depends, of course, on the size and location of the cancer mass in the body.
- As with many other cancer-treatment systems, the Panama City TPS also allowed doctors—by drawing on a display screen—to specify the location of shielding blocks that would be placed on the body to protect vital organs from radiation.
- The TPS at this facility allowed a maximum of four blocks, but the doctors often needed more. They discovered that it was possible to get the picture they wanted on the screen by tracing the outline of a large block and then drawing a second block inside it that would remove part of that area. Because it was faster, doctors began to use this technique even for four blocks.

Illustration of Block Placement

- Suppose that you want to specify the block pattern shown at the right.
- One approach is simply to draw the four triangles.
- The optimization the doctors found (which requires fewer blocks and often allowed them to circumvent the four-block limit) was to perform the following steps:
  - Draw a rectangle over the full area.
  - "Erase" the interior octagonal region.
- The success of this shortcut procedure depends on the ***direction*** in which the inner region is drawn.



Findings of the Investigative Report

> From August 2000 onwards, data for multiple blocks were entered for a number of cases using the new method when calculating exposures of the pelvic region, even when a fifth shielding block was not required. Treatments of other regions of the body which required blocks were still calculated by digitizing each block separately.
>
> Since the procedure was not put in writing, the shortcut was apparently used in a slightly different way for some patients. In these cases, the blocks were digitized by following the inner boundaries of the blocks in one direction, and the outer boundaries in the opposite direction. It was later found that this method of data entry did not lead to an incorrect treatment time. The treatment times calculated with this method later turned out to be essentially correct.

— *Investigation of an Accidental Exposure of Radiotherapy Patients in Panama,* International Atomic Energy Agency, 2001.