19
Перья

Класс Pen

Геометрические фигуры

Спирали

Драконовы кривые

Кривая Гильберта

Управление перьями

Как объяснялось в предыдущей главе, формы представляют образы на экране. Прямые линии можно создавать последовательным копированием одной формы в другую, в позиции, равномерно расположенные на отрезке между двумя заданными точками. Экземпляры класса Pen (Перо) предоставляют средства высокого уровня для рисования произвольных линий.

Класс Pen — подкласс класса BitBlt. Раз так, экземпляр этого класса содержит форму-источник и форму-цель. Форма-источник может окрашиваться в черный, белый или любой оттенок серого цвета и копироваться в форму-цель по различным правилами комбинирования с различными полутоновыми масками и с различными прямоугольниками отсечения. Форма-источник — это острие пера, которым оно пишет, форма-цель — поверхность, на которой перо пишет — обычно это форма, представляющая экран дисплея.

В дополнение к реализации, наследуемой из класса BitBlt, у пера есть переменная экземпляра — точка, которая определяет позицию пера на экране, и переменная экземпляра — число, которое показывает направление, в котором перо будет двигаться. Перо понимает сообщения, изменяющие его позицию на экране и направление движения. Когда изменяется позиция пера, перо может оставлять в прежней позиции копию формы-источника. Графическое изображение создается перемещением пера в различные позиции на экране и копированием формы-источника в одной или нескольких таких позициях.

Некоторые другие системы программирования также используют вышеописанный подход к рисованию линий. В этих системах устройство рисования обычно называют “черепашкой” (turtle), вслед за первой реализацией этой идеи в языке Лого в лаборатории MIT/BBN (Seymour Papert, *MindStorms: Children, Computer and Powerful Ideas,* Basic Books, 1980[1]; Harold Abelson and Andrea diSessa, *Turtle Geometry: The Computer as a Medium for Exploring Mathematics,* MIT Press, 1981). Протокол класса Pen поддерживает сообщения, аналогичные командам черепашки из языка Лого. Это команды, говорящие черепашке пройти некоторое расстояние, повернуться на некоторый угол, опустить или поднять перо. Когда перо опущено и черепашка двигается, по пути ее движения остается след. Соответствующие сообщения класса Pen - это go: distance, turn: amount, down и up.

Можно создать несколько перьев и так скоординировать их передвижение на экране, что станет привлекательным сам процесс создания графического изображения. Следующий раздел содержит описание протокола класса Pen. Последующие подразделы содержат примеры рисунков, которые можно создать, посылая сообщения экземплярам класса Pen.

Класс Pen

Экземпляры класса Pen создаются посылкой классу Pen сообщение new. Созданное таким образом перо может рисовать на всем экране; его начальная позиция — центр экрана, начальное направление — вверх. Перо установлено для рисования (опущено вниз) с формой-источником (острием пера), представляющей черную точку размером 1 × 1 пиксел.

Есть два способа изменить начальную форму-источник пера. Первый способ — послать перу сообщение defaultNib: widthInteger. Другой способ — послать перу сообщение, унаследованное из суперкласса BitBlt. Например, сообщение sourceForm: изменит форму-источник, а сообщение mask: изменит форму-полутон (маску), используемую при отображении формы-источника. (Заметим, что по умолчанию маска — черная.)

[1] Есть перевод этой книги на русский язык: С. Пейперт, Дети, компьютеры и плодотворные идеи, — М., “Педагогика”, 1989 г. — Примеч. перев.

Pen instance protocol

initialize-release

defaultNib: shape	Это сообщение — простейший способ установить перо по умолчанию. Форма получателя определяется как прямоугольная фигура с высотой и шириной равными: (1) аргументу shape, если он — целое число; (2) координатам аргумента shape, если он — точка.

Так, выражение

```
bic ← Pen new defaultNib: 2
```

создает перо с острием — черной формой в 2 бита шириной и 2 бита высотой.

Протокол класса Pen обеспечивает доступ к текущей позиции, направлению и к области рисования пера. Область рисования пера называется *рамкой пера* (frame).

Pen instance protocol

accessing

direction	Возвращает текущее направление получателя в градусах; 270 означает направление вверх.
location	Возвращает текущую позицию получателя.
frame	Возвращает прямоугольник, в котором получатель может рисовать.
frame: aRectangle	Устанавливает прямоугольник aRectangle, в котором получатель может рисовать.

Продолжим пример с пером bic и, предполагая, что экран дисплея имеет ширину 600 точек и высоту 800 точек, получим:

выражение	*результат*
bic direction	270
bic location	300 @ 400
bic frame: (50 @ 50 extent: 200 @ 200)	
bic location	300 @ 400

Заметим, что когда перо bic направлено к верху экрана, его угол равен 270 градусам. Заметим также, что сейчас перо находится вне своей рамки и его нужно переместить внутрь прямоугольника (50 @ 50 corner: 250 @ 250) прежде, чем можно будет увидеть его следы.

Следующие “черепашьи” команды изменяют состояние пера, его ориентацию и положение.

Pen instance protocol

moving

down	Устанавливает состояние получателя “вниз”, опуская перо, после чего оно оставляет след во время движения.
up	Устанавливает состояние получателя “вверх”, поднимая перо, после чего оно не оставляет след во время движения.
turn: degrees	Изменяет направление получателя на заданное число градусов degrees.
north	Устанавливает направление получателя к верху экрана.
go: distance	Перемещает получателя в текущем направлении на расстояние, равное целому числу distance точек. Если перо опущено, рисуется линия с использованием формы-источника, как кисти.

goto: aPoint	Перемещает получателя в точку экрана с координатами aPoint. Если перо опущено, рисуется линия от текущей позиции до точки aPoint. Направление получателя не изменяется.
place: aPoint	Перемещает получателя в точку экрана с координатами aPoint, но ничего не рисует.
home	Помещает получателя в центр области (рамки), в которой он может рисовать.

Итак, мы можем поместить перо bic в центр его рамки, выполняя выражение

```
bic home
```

Если после этого мы выполним выражение

```
bic location
```

ответом будет точка 150 @ 150.

Предположим, что мы нарисовали линию пером, а потом решили стереть ее. Если линия рисовалась черной формой, то стереть линию мы сможем, рисуя по ней белой формой по крайней мере того же размера. Таким образом, пусть

```
bic go: 100
```

нарисовало черную линию. После этого вычислим выражение

```
bic white
```

чтобы установить белую маску (сообщение white наследуется из протокола класса BitBlt) и, наконец, вычислим выражение

```
bic go: –100
```

и нарисуем линию поверх первоначальной, стирая ее.

Рассмотрим обычный для Лого пример создания различных фигур многоугольников, например, квадрата.

```
4 timesRepeat: [bic go: 100. bic turn: 90]
```

Следующее выражение создает любой правильный многоугольник, вычисляя угол поворота, как функцию числа сторон. Если nSides — число сторон требуемого многоугольника, то выражение

```
nSides timesRepeat: [bic go: 100. bic turn: 360 // nSides]
```

будет рисовать этот многоугольник. Мы можем создать класс Polygon (Многоугольник), экземпляры которого характеризуются числом сторон и длиной стороны. Кроме того, каждый многоугольник имеет собственное перо для рисования. В описании, которое следует далее, мы определяем, что многоугольник можно нарисовать на экране дисплея, если послать ему сообщения draw. Метод реализации этого сообщения фактически уже был описан ранее.

class name	Polygon
superclass	Object
instance variable names	drawingPen nSides length

class methods

instance creation

```
new
    ↑super new default
```

instance methods

```
drawing
    draw
        drawingPen black.
        nSides timesRepeat: [drawingPen go: length; turn: 360 // nSides]

accessing
    length: n
        length ← n

    sides: n
        nSides ← n

private
    default
        drawingPen ← Pen new.
        self length: 100.
        self sides: 4
```

Теперь, выполняя следующие выражения, мы создаем экземпляр класса Polygon и рисуем на экране последовательность различных многоугольников.

```
poly ← Polygon new.
3 to: 10 do: [ :sides | poly sides: sides. poly draw]
```

Результат показан на рис. 19.1.

Рис. 19.1.

Геометрические фигуры

Упомянутые ранее книги по языку Лого содержат множество примеров того, как применять команды рисования линий для создания изображений на экране компьютера. Мы приведем несколько примеров методов, которые можно добавить к классу Pen для того, чтобы любое перо могло создавать геометрические фигуры, подобные показанным на рис. 19.2–19.5. (Примечание: эти методы включены в систему, как часть описания класса Pen, поэтому пользователь может поупражняться в создании различных геометрических фигур.)

Спирали

Первая фигура называется спиралью. Спираль получается, когда перо рисует постепенно удлиняющиеся прямые, поворачиваясь после прорисовки каждой прямой на некоторый угол. Рисование спирали начинается с прямой линии длины 1 с последующим увеличением длины каждый раз на 1 до достижения длины, равной первому аргументу сообщения spiral:angle:. Второй аргумент этого сообщения — угол поворота пера после прорисовки каждой прямой.

```
    spiral: n angle: a
        1 to: n do:
            [:i | self go: i. self turn: a]
```

Каждая из прямых на рис. 19.2 была нарисована посылкой перу bic сообщения spiral:angle:.

```
bic spiral: 150 angle: 89
```

Рис. 19.2a.

```
bic spiral: 150 angle: 91
```

Рис. 19.2b.

```
bic spiral: 150 angle: 121
```

Рис. 19.2c.

```
bic home.
bic spiral: 150 angle: 89.
bic home.
bic spiral: 150 angle: 91
```

Рис. 19.2d.

Драконовы кривые

Рис. 19.3 изображает “драконову кривую” порядка 8, которая рисуется в середине экрана при выполнении следующих выражений:

```
bic ← Pen new defaultNib: 4.
bic dragon: 8
```

С сообщением dragon: в классе Pen связан следующий метод:

```
dragon: n
    n = 0
        ifTrue: [self go: 10]
        ifFalse:
            [n > 0
                ifTrue:
                    [self dragon: n – 1.
                     self turn: 90.
                     self dragon: 1 – n]
                ifFalse:
                    [self dragon: –1 – n.
                     self turn: –90.
                     self dragon: 1 + n]]
```

Рис. 19.3.

Драконовы кривые обсуждались Мартином Гарднером (Martin Gardner) в его колонке математических игр в *Scientific American* (March 1967, p.124, April 1967, p.119). Другое описание драконовых кривых можно найти в статье Donald Knuth, Chandler Davis, “Number Representation and Dragon Curves”, *Journal of Recreation Mathematics,* Vol. 3, 1970, pp. 66–81 and 133–149.

Кривая Гильберта

На рис. 19.4 изображена заполняющая плоскость кривая, связанная с именем математика Давида Гильберта (David Hilbert). Заполняющая плоскость кривая имеет индекс; когда индекс стремиться к бесконечности, кривая стремиться покрыть все точки плоскости. Пример такой кривой - результат выполнения выражения

```
Pen new hilbert: 5 side: 8
```

Рис. 19.4.

Индекс кривой в примере равен 5; в каждой точке рисуется линия длиной в 8 пикселов. Ниже приводиться метод для сообщения hilbert:side:.

```
hilbert: n side: s
    | a m |
    n = 0 ifTrue: [↑self turn: 180].
    n > 0 ifTrue: [a ← 90.
                   m ← n –1]
          ifFalse: [a ← 90.
                   m ← n + 1].
    self turn: a.
    self hilbert: 0 – m side: s.
    self turn: a.
    self go: s.
    self hilbert: m side: s.
    self turn: 0 – a.
    self go: s.
    self turn: 0 – a.
    self hilbert: m side: s.
    self go: s.
    self turn: a.
    self hilbert: 0 – m side: s.
    self turn: a
```

Вывод на экран кривой Гильберта может давать очень красивый эффект, если в качестве формы-источника брать различные фигуры. Предположим, что форма-источник с именем dots — системный курсор, представляющий собой строку из трех точек. Рис. 19.5 создан в результате выполнения выражений:

```
bic ← Pen new sourceForm: Cursor dots.
bic combinationRule: Form under.
bic hilbert: 4 side: 16
```

Рис. 19.5

Выражения Cursor dots и Form under определяют, соответственно, форму и правило комбинирования, которые являются системными константами и известны названным классам. Другие такие константы перечислены в следующей главе. Сообщения sourceForm: и combinationRule: наследуются классом Pen из суперкласса BitBlt.

Управление перьями

Следующий пример приведен на рис. 19.6. Хотя мы не можем показать здесь полностью процесс создания полученного изображения, это хороший пример того, что читателю можно пробовать воспроизвести самому. Основная идея состоит в том, чтобы создать объект, который мог бы управлять несколькими перьями и координировать их действия при рисовании. Класс таких объектов мы назовем именем Commander (Командир). Экземпляр класса Commander — массив перьев. Перьям, управляемым экземпляром класса Commander, можно задать направления, перечисляя перья и для каждого из них выполняя блок, содержащий команды перьям. Например, если все перья “командира” должны выполнить сообщение go: 100, то ему можно послать сообщение

```
do: [:eachPen | eachPen go: 100]
```

Класс Commander также поддерживает сообщения, размещающие перья его экземпляра так, чтобы они могли создавать фигуры на основе симметрии. Два таких сообщения приведены ниже в описании класса Commander. Это сообщения fanOut и lineUpFrom: startPoint to: endPoint. Первое сообщение размещает перья так, что их направления равномерно распределяются на окружности. Второе сообщение располагает перья равномерно вдоль прямого отрезка, соединяющего точки, заданные как аргументы сообщения.

Далее следует описание класса Commander. Сообщение new: переопределяется, поскольку массив должен в каждом своем элементе хранить перо.

```
class name                    Commander
superclass                    Array

class methods

instance creation
    new: numberOfPens
        | newCommander |
        newCommander ← super new: numberOfPens.
        1 to: numberOfPens do:
            [:index | newCommander at: index put: Pen new].
        ↑newCommander

instance methods

distributing
    fanOut
        1 to: self size do:
            [:index |
                (self at: index) turn: (index – 1) * (360 / self size)]

    lineUpFrom: startPoint to: endPoint
        1 to: self size do:
            [:index | (self at: index)
                place: startPoint +
                    (stopPoint – startPoint * (index – 1) / (self size – 1))]
```

Приведенные методы — хорошие примеры посылок сообщений экземплярам класса Point. Изображение на рис. 19.6 получилось при выполнении выражений:

```
bic ← Commander new: 4.
bic fanOut.
bic do: [:eachPen | eachPen up. eachPen go: 100. eachPen down].
bic do: [:eachPen | eachPen spiral: 200 angle: 121]
```

Рис. 19.6.

Сообщение do: экземпляру класса Commander наследуется им из его суперкласса Collection.

Другой результат использования класса Commander приведен на рис. 19.7. Это изображение создано при помощи сообщения lineUpFrom:to:. Оно представляет собой простую последовательность спиралей, размещаемых вдоль вертикального отрезка прямой под некоторым углом.

```
bic ← Commander new: 6.
bic lineUpFrom: (300 @ 150) to: (300 @ 500).
bic do: [:eachPen | eachPen spiral: 200 angle: 121]
```

☐ *Дополнительный протокол класса* Commander Расширенное описание класса Commander добавляет к первоначальному протоколу те сообщение из протокола класса Pen, которые изменяют позицию или ориентацию перьев. Дополнительный протокол позволяет посылать сообщения из протокола класса Pen экземплярам класса Commander. Каждое такое сообщение реализуется как передача сообщения всем элементам набора. Таким образом, сообщения экземпляру класса Commander такие же, что и сообщения любому перу, а не сообщение do:. При таком описании класса Commander, процесс формирования изображения экземпляром этого класса оказывается более похожим на рисование сразу несколькими перьями. Кроме того, все перья, управляемые экземпляром класса Commander рисуют свои линии параллельно; например, все спирали на рис. 19.6 и 19.7 будут разворачиваться одновременно.

```
down
    self do: [:each | each down)

up
    self do: [:each | each up]

turn: degrees
    self do: [:each | each turn: degrees]

north
    self do: [:each | each north]

go: distance
    self do: [:each | each go: distance]

goto: aPoint
    self do: [:each | each goto: aPoint]

place: aPoint
    self do: [:each | each place: aPoint]

home
    self do: [:each | each home]

spiral: n angle: a
    1 to: n do:
        [:i | self go: i. self turn: a]
```

Рис. 19.7.

Используя этот дополнительный протоколом, рисунок 19.6 можно получить как результат выполнения выражений

```
bic ← Commander new: 4.
bic fanOut.
bic up.
bic go: 100.
bic down.
bic spiral: 200 angle: 121
```

а рисунок 19.7 как результат выполнения выражений

```
bic ← Commander new: 6.
bic lineUpFrom: (300 @ 750) to: (300 @ 500).
bic spiral: 200 angle: 121
```