Министерство общего и профессионального образования
Российской Федерации
РОСТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Ю.А. Кирютенко В.А. Савельев

Объектно-ориентированное программирование и язык Smalltalk. Графика в Smalltalk/V for Windows

Ростов-на-Дону
1998

Ю.А. КирютенкоВ.А. Савельев

Объектно-ориентированное программирование и язык Smalltalk.

Графика в Smalltalk/V for Windows

Аннотация

Методическая разработка посвящена современному направлению в программировании — объектно-ориентированной методологии программирования и языку Smalltalk и является продолжением вышедших ранее методических разработок по объектно-ориентированному программированию и языку Smalltalk, посвященных общим концепциям и синтаксису языка, интерфейсу пользователя и среде программирования, классам Collection, Magnitude, Stream и их подклассам, протоколам поддержки всех объектов и всех классов системы, классам ядра графики системы Smalltalk/V. В данной методической разработке описываются классы, обеспечивающие работу с графическими объектами в системах Smalltalk/V for Windows и SmalltalkExpress фирмы ParkPlace/Digitalk (США).

Методическая разработка предназначена для студентов 3—5 курсов механико-математического факультета и слушателей ФПК.

Печатается в соответствии с решением кафедры математического анализа Ростовского государственного университета, протокол № 9 от 29 июня 1998 года.

Настоящие методические указания набраны в системе LaTeX с использованием кириллических шрифтов семейства «Литературная» (AOParaGraph) и математических шрифтов семейств Computer Modern и Math Symbol (American Mathematical Society).

1 Введение

Пример графических возможностей языка Smalltalk/V можно увидеть, если, например, просмотреть демонстрационные примеры, пользуясь опцией Graphics-Demo в меню File окна Transcript. Здесь мы рассмотрим вопросы построения графических образов и анимацию более подробно.

Графика Smalltalk/V for Windows формируется на основе графических возможностей среды MS Windows. Основа языка графики в Windows — Интерфейс Графических Устройств (GDI — Graphics Device Interface, см. [4]). Smalltalk/V вызывает функции GDI для того, чтобы с их помощью реализовать графические операции языка Smalltalk.

Smalltalk/V поддерживает и растровую, и векторную графику. Изображение в векторной графике задаётся как набор объектов, характеризуемых цветом и математическим описанием линий, ограничивающих (окаймляющих) их. Любая графическая форма может представляться и преобразовываться в соответствии с абстрактным геометрическим описанием, не принимая во внимание разрешающую способность конкретного графического устройства. Эта особенность векторной графики реализует абстрактную графическую модель и позволяет строить изображения, независящие от графического устройства.

Основываясь на идее независимости от устройства, система Smalltalk/V, работающая с графическим языком, делает достаточно лёгким вывод графики в графическую среду, которой обычно является экран дисплея, но такой средой может быть и принтер, и графопостроитель или любое другое записывающее устройство. Преимущество независимости от устройства состоит в том, что качество — разрешающая способность и цвет — возникающего в результате графического образа является функцией устройства вывода, а не устройства, на котором создавался графический образ.

Некоторые устройства вывода, типа механического графопостроителя, реализуют векторную графику непосредственно, например, непрерывно перемещая перо из позиции A в позицию B. Но многие устройства вывода создают изображения, составленные из точек — линия состоит из каждой включаемой в неё точки на пути от точки A до точки B. Следовательно, образы векторной графики часто представляются растровым устройством вывода, как наилучшим из всего, что вообще можно сделать. Именно поэтому понимание графики в языке Smalltalk, векторной или растровой, лучше всего основывать на понимании основ растровой графики. Расширение растровых графических понятий до векторных возможностей затем происходит относительно просто.

Упомянутые выше растровые точки отображаются на вашем экране как пикселы (pixel) (этот термин является сокращением от английского выражения “picture elements” —«элементы изображения»), а составляемое из пикселов изобра-

жение сохраняется как растр — экземпляр класса **Bitmap**. Таким образом, растр — это просто матрица битов, со значением 1, представляющим белый пиксел (белую точку) или 0, представляющим чёрную точку. Чтобы сослаться на конкретную точку внутри растра, используются точки — экземпляры класса **Point**.

Напомним, каждая точка имеет две переменные экземпляра: **x** — горизонтальную координату, и **y** — вертикальную координату. Графические устройства, чтобы адресовать пикселы, должны иметь систему координат, налагаемую на устройство. Начало координат $(0, 0)$ может отображаться в любом его месте. По умолчанию, начало совпадает с верхним левым углом. Переменная x возрастает при движении слева на право, а переменная y — при движении сверху вниз (см. Рис. 1). Единицы измерения системы координат можно выбирать. Возможный выбор зависит от графического интерфейса.

Рис. 1: Система координат Smalltalk/V.

При перемещении группы точек из одного места в другое (или внутри того же самого растра или между различными растрами), для представления в системе координат включаемых в операцию прямоугольных областей используются прямоугольники — экземпляры класса **Rectangle**. Прямоугольники представляется двумя точками: **origin** (начало — верхняя левая точка), и **corner** (угол — нижняя правая точка). По этой информации **Smalltalk** может определить **extent** (экстент — размер) прямоугольника, то есть его ширину и высоту. Экстент вычисляется по формуле: **corner** − **origin**.

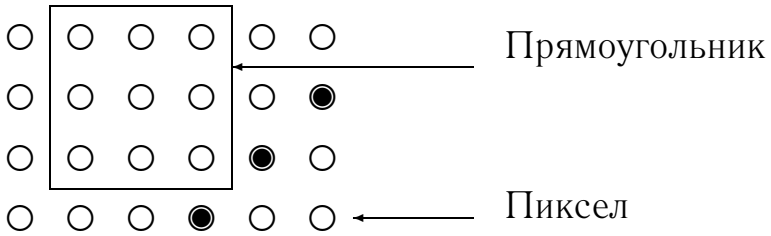

Рис. 2: Прямоугольник **1@0 corner: 4@3**

Прямоугольник включает в себя пикселы, расположенные внутри прямоугольника. Сам прямоугольник непосредственно как бы накладывается на промежутки (лакуны) между пикселами (см. Рис. 2). Поэтому (обратите на это внимание!) точка **origin** и пикселы верхней и левой границ прямоугольника включаются в прямоугольник, в то время как точка **corner** и пикселы нижней и правой границ — нет. Например, прямоугольник **1@0 corner: 4@3** содержит 9 пикселов.

Классы **Point** и **Rectangle** подробно описаны в [2]. Несложно изучить их основ-

ные свойства самостоятельно, пользуясь встроенным в систему окном просмотра иерархии классов. Мы же двинемся дальше и отметим другие классы графики системы Smalltalk.

Модель графики Smalltalk/V поддерживает метафору «перо и бумага». Графическая среда (холст) — нечто, способное к сохранению изображения, подобна бумаге, а графическое средство (или инструмент) — нечто, способное создавать изображение, подобно перу.

Графической средой может быть экран монитора, принтер, файл или часть компьютерной памяти. Графические среды представляется абстрактным классом GraphicsMedium и его подклассами Bitmap, Printer, Screen и StoredPicture, а также классом Window и его многочисленными подклассами. Экземпляр любого подкласса Window — это окно в смысле системы Windows.

Объекты, которые создают графические образы в графических средах, являются экземплярами подклассов класса GraphicsTool, которые образуют следующую иерархию.

```
GraphicsTool
  TextTool
    Pen
      RecordingPen
        Commander
```

Каждый графический класс имеет соответствующий набор переменных экземпляра и множество методов, реализующих требуемые графические операции. Если возникнет необходимость выйти за пределы графических возможностей ядра языка Smalltalk/V, следует обращаться к классам, реализующим низкоуровневый программный интерфейс с системой Windows (Windows Classes) и вызовам функций интерфейса прикладного программирования (API). Smalltalk/V позволяет вызвать любую функцию, описанную в Справочнике по Разработке Программного обеспечения MS Windows (Microsoft Windows Software Development Kit Reference). Любые необходимые функции, не определённые в основном образе системы Smalltalk могут быть добавлены в него.

Теперь рассмотрим подробно большинство из перечисленных графических классов и приведём примеры, демонстрирующие характерное поведение их экземпляров.

2 Графическая среда

Класс GraphicsMedium — абстрактный класс, который не создаёт экземпляров. Экземпляры создают его подклассы Bitmap, Printer, Screen и StoredPicture. Класс Bitmap (Растр), может иметь столько экземпляров, сколько необходимо. А классы Printer, Screen и StoredPicture должны иметь по одному экземпляру для каждого реально существующего материального устройства. Например, класс Printer может иметь экземпляры для `LPT1`, `LPT2`, и т.д. Глобальная пере-

менная Display — экземпляр класса Screen для экрана монитора. Каждый экземпляр класса StoredPicture связывается с графическим метафайлом Windows (на диске или в оперативной памяти).

Класс Window и его многочисленные подклассы — также графические среды. Несмотря на то, что они и не наследуют переменных и методов класса GraphicsMedium как подклассы, все подклассы класса Window реализуют полный протокол класса GraphicsMedium. Все, что можно делать с подклассами GraphicsMedium, можно делать и с подклассами Window. Они были «подняты» до состояния прямых подклассов класса Object для того, чтобы их было просто найти в иерархии и подчеркнуть фундаментальное значение этих объектов в прикладном программировании.

2.1 Класс GraphicsMedium

GraphicsMedium — абстрактный класс, который содержит общий код для своих подклассов. Он имеет две переменные экземпляра:

graphicsTool — содержит графическое средство, связанное со средой;

deviceContext — содержит контекст устройства среды, который содержит зависящую от устройств информацию.

Каждый раз, когда создаётся новый экземпляр среды, создаётся и связывается со средой графический инструмент. Именно к этому графическому инструменту мы обращались во всех предыдущих примерах и будем обращаться далее, посылая сообщение pen графической среде. Например:

```
(Bitmap width: 100 height: 100) pen.
Printer new pen.
Display pen.
```

2.2 Растры

Точка или прямоугольник задают позицию или область позиций. Объект, который может фактически содержать графическое изображение — экземпляр класса Bitmap. Основные возможности растровой графики реализуются, используя растровые структуры данных — массивы битов, которые представляют изображение.

Когда растровое изображение — простой одноцветный графический образ, для описания изображения необходим всего бит на пиксел. Цвет, использующий 4, 8 или 24 бита на пиксел, увеличивает уровень информативности и сложности растровых структур данных. Растры, используемые для вывода изображений на

экран, скрывают под своим простым интерфейсом более громоздкий и сложный механизм GDI-функции.

Каждый растр имеет три переменные экземпляра:

archive — используется как буфер, чтобы хранить биты растра, когда сохраняется образ.

bitmapHandle — содержит дескриптор растра, когда образ активен и растровые биты находятся в памяти.

bitmapInfo — содержит структуру, которая описывает ширину и высоту растра, количество битов и битовых плоскостей.

Для одноцветного растра, каждый бит в растре соответствует пикселу, когда тот отображается на экране. Для многоцветного растра, нужно более одного бита, чтобы представить на экране цветной пиксел. Есть два практически эквивалентных способа реализации цветных растров. Один из способов представления цвета в растре состоит в том, чтобы иметь множество плоскостей. Каждая плоскость будет состоять из последовательных битов в памяти. Биты в той же самой позиции каждой плоскости воспринимаются вместе, формируя цвет пиксела в данной позиции. Таким образом, шестнадцатицветный растр нуждается в четырёх плоскостях.

Другой способ реализации цветного растра представляет пиксел последовательными битами в памяти. Для растра из 16-ти цветов каждый пиксел представляется четырьмя последовательными битами.

Выбор способа представления осуществляется Windows, на основании установленного видеоадаптера и его драйвера. Поэтому при программировании лучше ориентироваться на высокоуровневые графические операции, меньше зависящие от применяемого метода реализации цветов.

Чтобы создать, например, новый одноцветный растр размером 100 на 100 пикселов, надо вычислить или выражение **Bitmap width: 100 height: 100** или выражение **Bitmap extent: 100 @ 100**.

Многоцветный растр можно создать, если вычислить любое из следующих двух выражений

```
Bitmap screenWidth: 100 height: 100.
Bitmap screenExtent: 100 @ 100.
```

Последние два выражения создают растр, который является совместимым с экранным форматом, или с помощью плоскостей или с помощью последовательности битов на один пиксел (число битов на один пиксел или число плоскостей определяется счётчиком битов). Все четыре выражения автоматически создают

перо — экземпляр класса Pen, связанное с экземпляром класса Bitmap. Это перо сохраняется в переменной экземпляра graphicsTool, и к нему можно обращаться посредством сообщения pen. Если читатель знаком с графикой системы Smalltalk/V, то класс Bitmap системы Smalltalk/V for Windows — это «улучшенный» вариант класса Form (см. [3])

Рассмотрим несколько примеров. В первом примере

```
| bitmap |
bitmap := Bitmap screenExtent: 100 @ 100.
bitmap pen home;
        fill: ClrRed; foreColor: ClrGreen;
        ellipse: 40 minor: 20; displayText: 'Hi'.
bitmap displayAt: 10 @ 100 with: Display pen.
bitmap release
```

создаётся растр размером 100 на 100 пикселов с атрибутами используемой видеосистемы (Bitmap screenExtent: 100 @ 100) и в нем собственным пером растра прямо в памяти, действуя подобно физическому устройству, создаётся рисунок, состоящий из эллипса и строки. Более подробно, в этом растре будет сделано следующее:

bitmap pen home — перо растра устанавливается в центр растра;

fill: ClrRed — прямоугольник заполняется красным цветом;

foreColor: ClrGreen — цвет инструмента становится зелёным;

ellipse: 40 minor: 20 — рисуется зелёный эллипс на красном фоне;

displayText: 'Hi' — в текущей позиции пера выводится текст.

Затем, используется перо среды Display и метод displayAt:, на экран в точке 10 @ 100, с которой будет совпадать левый верхний угол растра, отображается созданный в памяти растр, после чего (последняя строка метода) освобождается память, связанная с растром. Надо отметить, что память, используемая под растры, принадлежит не Smalltalk'у, а Windows, и *автоматически сборщиком мусора не собирается,* поэтому ответственность за освобождение этой памяти лежит на программисте.

В следующем примере

```
| aBitmap |
aBitmap:= (Bitmap screenWidth: 100 height: 100).
aBitmap pen erase; place: 20 @ 20;
```

```
        box: 80 @ 80.
aBitmap displayAt: 0 @ 0 with: Display pen.
aBitmap release
```

создаётся растр в памяти размером 100 на 100 пикселов, пером растра поле заполняется фоновым цветом экрана (**erase**), перо помещается в точку растра **20 @ 20 (place: 20 @ 20)** и внутри поля рисуется контур прямоугольника **20 @ 20 corner: 80 @ 80** (**box: 80 @ 80**). Созданный растр пером среды **Display** отображается в верхнем левом углу экрана, при этом верхняя левая точка растра помещается в точку экрана **0 @ 0 (displayAt: 0 @ 0 with: Display pen)**. Последняя строка метода освобождает память, связанную с растром.

В отличие от метода с селектором **outputToPrinter** из класса **String**, который печатает строки в символьном виде, метод с тем же селектором из класса **Bitmap** позволяет вывести на принтер растр, рассматривая принтер системы как соответствующую графическую среду. Следующий код напечатает на бумаге эллипс (имейте в виду, что начало координат (0,0) — верхний левый угол листа):

```
| bitmap |
bitmap := Bitmap extent: 100 @ 100.
bitmap pen home;
   ellipse: 40 minor: 20.
bitmap outputToPrinter.
bitmap release
```

Растр всегда можно сохранить в файле (**outputToFile:**) с расширением '`*.bmp`', а сохранённый ранее растр можно вновь загрузить в систему и использовать его. Код следующего примера сначала создаст растр, аналогичный растру в предыдущем примере, затем запишет его в файл, после чего восстановит его из файла в другой экземпляр класса **Bitmap** и выведет на экран новый растр:

```
| bitmap bitmap1|
bitmap := Bitmap extent: 100 @ 100.
bitmap pen home;
       ellipse: 40 minor: 20.
bitmap outputToFile: 'c:\test.bmp'.
bitmap1 := Bitmap fromFile: 'c:\test.bmp'.
bitmap1 displayAt: 0 @ 0 with: Display pen.
bitmap release.
bitmap1 release.
```

Конечно, первые примеры — плохие примеры, поскольку производимый на экран вывод «портит» экран. И хотя объект **Display** — экземпляр класса **Screen**

— делает доступным физически существующий экран монитора для графических операций и проявляет все программируемые графические возможности среды GraphicsMedium, он рассматривается как “недостаточная среда” для непосредственного воспроизведения рисунка. Переменная Display нужна прежде всего для того, чтобы начать сеанс работы системы Smalltalk, предоставляя ей исходную графическую среду.

Принципы графического интерфейса пользователя предлагают выполнять графические операции внутри специальных окон (экземпляров класса Window). Поэтому нет необходимости переносить растровые изображения непосредственно на экран. Графика, использующая основной графический интерфейс, связанный с окном, позволяет рисовать в графической панели (экземпляре класса GraphPane), расположенной внутри открытого окна. А перо экрана (Display pen) чаще всего используется тогда, когда необходимо переместить рисунок из одного окна в другое. Дело в том, что перо окна не может рисовать вне своего окна, в то время как перо экрана рисует в любом месте экрана.

Покажем на примере перенос растра внутрь окна, добавляя небольшой кусочек «мерцающей» анимации. Сначала приведём полный текст примера.

```
| major minor |
Pictures := Array new: 8.
1 to: Pictures size do: [: i |
      Pictures at: i put: (Bitmap screenExtent: 100 @ 100).
      major := (i - 1 * 10) integerCos // 2.
      minor := (i - 1 * 10) integerSin // 2.
      (Pictures at: i) pen home;
                       fill: ClrRed;
                       foreColor: ClrGreen;
                       ellipseFilled: major minor: minor].
Window turtleWindow: 'Flexing Ellipses'.
10 timesRepeat: [ 1 to: Pictures size do: [:i |
                (Pictures at: i) displayAt: 10 @ 100 with: Turtle]].
1 to: Pictures size do: [: i | (Pictures at: i) release]
```

Теперь рассмотрим, что же происходит. Сначала создаётся массив Pictures, в котором будут содержаться восемь растров, располагаемых в памяти, каждый из которых, используя собственное перо, создаёт в растре рисунок. Глобальной переменной этот объект сделан для того, чтобы им можно было воспользоваться позже в примере, связанном с анимацией. Сам рисунок состоит из зелёного контура эллипса на красном поле, размеры которого зависят от индекса растра в массиве Pictures. Затем создаётся окно с именем 'Flexing Ellipses' ('Мерцающие эллипсы'), графической панелью и графическим инструментальным средст-

вом (пером панели) Turtle (все это делает сообщение turtleWindow:). В этом окне растры последовательно выводятся 10 раз подряд, создавая иллюзию “бьющегося сердца”. В окне рисунок выполняется пером Turtle, используя простое итерационное выражения с методом displayAt:, показывая один за другим каждый из растров. Последнее выражение освобождает память, связанную с растрами.

Если компьютер очень быстрый, и динамика изменения растров не воспринимается, в предпоследнем выражении перед выводом растра в окно поставьте, например, выражение 200 factorial. То есть это часть метода должна принять следующий вид:

```
10 timesRepeat: [ 1 to: pictures size do: [:i | 200 factorial.
                (pictures at: i) displayAt: 10 @ 100 with: Turtle]].
```

2.3 Класс StoredPicture

Класс StoredPicture используется для того, чтобы сохранять последовательность графических операций в метафайле Windows, дескриптор которого содержится в переменной экземпляра этого класса с именем hMetaFile. Метафайл Windows — это не файл дисковой файловой системы. Это средство фиксации «телодвижений» программы по отношению к графическому контексту. Точнее, это особый графический контекст, в котором обращения к GDI не выполняют рисование, а записываются в сжатой независимой от устройства форме. Потом можно выбрать один из действительных контекстов (окно, экран, принтер,...) и послать метафайлу сообщение play:, выполняя в заданном контексте сохранённые обращения. В принципе, именно метафайл создаёт в Windows независимую от устройства графику. (В OS/2 метафайл имеет расширение met, а в Windows — wmf.) Почти вся созданная пером графика может сохраняться в метафайле. Экземпляр класса StoredPicture может, если это необходимо, сохранить метафайл в файле и позже загрузить в систему для повторного использования, или поместить его в буфер обмена для другой задачи, которой он нужен. Следующая последовательность выражений показывает, как можно использовать экземпляр класса StoredPicture:

```
| meta |
meta := StoredPicture new.     "Создать экземпляр StoredPicture."
meta create: nil.              "Создать метафайл в объекте meta."
meta pen place: 100 @ 100;
          box: 200 @ 200.      "Рисовать в объекте meta."
meta close.                    "Закрыть объект meta."
meta save: 'test.met'.         "Сохранить на диске."
meta load: 'test.met'.         "Загрузить с диска."
```

```
Window turtleWindow: 'test'.  "Создать окно для вывода."
meta play: Turtle.            "Пользуясь пером Turtle,
                              вывести образ в окно."
meta release                  "Освободить системные ресурсы,
                              выделенные для метафайла."
```

К сожалению нет ещё полного стандарта на язык Smalltalk, поэтому одни и те же операции в разных реализациях, полностью совпадая концептуально, не совпадают текстуально. Все примеры до сих пор относились к системе SmalltalkExpress. Ниже — тот же самый пример, но только в реализации VisualSmalltalk 3.0:

```
| meta |
meta := StoredPicture new.          "Создать экземпляр StoredPicture."
meta startRecording.                "Создать метафайл."
meta pen place: 100 @ 100;
      box: 200 @ 200.               "Рисовать в нем."
meta stopRecording.                 "Закрыть объект meta."
meta outputToFile: 'e:\test.met'.   "Сохранить на диске."
meta fromFile: 'e:\test.met'.       "Загрузить с диска."
Window turtleWindow: 'test'.        "Создать окно для вывода."
meta displayWith: Turtle.           "Пользуясь пером Turtle,
                                    вывести образ в окно."
meta release                        "Освободить системные ресурсы,
                                    выделенные для метафайла."
```

Поэтому, все, что говорилось и будет говориться о графике в системе Smalltalk, при переходе от реализации к реализации должно изменяться в технических деталях, при идейном совпадении программ.

2.4 Принтер

При рассмотрении растров уже отмечалось, что растры и метафайлы можно печатать. Класс Printer позволяет печатать на бумаге графику, используя те же самые принципы. Таким образом, можно по-прежнему использовать перо, в этом случае перо принтера, которое является экземпляром класса Pen), для того, чтобы создать графический объект в этой графической среде, а затем вывести графику на принтер по правилу “что увидели бы в окне, то получим и на бумаге”. Из-за очень разных физических характеристик и принципов действия дисплеев и принтеров этот результат достигается только иногда. Для получения качественных документов следует вывод на принтер программировать отдельно, учитывая как его

достоинства (прежде всего высокое разрешение), так и существенные ограничения.

Когда на принтер выводится графика, начало координат (0,0) — верхний левый угол страницы. В Windows, печатаемый объект обычно выводится в очередь спулера принтера. Связь между спулером и самим принтером управляется Windows. Всегда можно реконфигурировать очередь спулера и соединение принтера, пользуясь панелью управления системы.

Следующий пример использует ваш принтер по умолчанию так, как это определено в панели управления печатью графики системы Windows:

```
| printer |
printer:= Printer new. "Создаёт объект для принтера по умолчанию."
printer startPrintJob. "Начинаем процесс печати."
printer pen "Использует перо принтера для рисования."
        place: 100 @ 100; "Устанавливает начальную позицию пера."
        mandala: 8 diameter: 90.
         "Рисует вписанный в шар диаметром 90 единиц правильный
          8-угольник и соединяет друг с другом все его вершины."
printer endPrintJob "Завершить процесс печати."
```

К возможностям класса Printer вернёмся позже, после того как рассмотрим классы графических инструментов и, в частности, класс RecordingPen.

3 Графические инструменты

Объекты, с помощью которых можно создавать рисунки в графических средах, сгруппированы в иерархию абстрактного класса GraphicsTool, который только содержит переменные и методы, общие для всех графических инструментальных средств, а также методы, которые реализуют операции, подобные BitBlt — операции переноса битов. (Об операциях BitBlt, реализованных на языке Smalltalk, включая прямоугольники отсечения и правила комбинирования, можно прочитать в [3].)

Класс GraphicsTool имеет несколько подклассов, каждый из которых расширяет функциональные возможности суперкласса. Экземпляр класса TextTool работает подобно пишущей машинке и отображает только литеры. Экземпляр класса Pen способен не только печатать, но и рисовать. Экземпляр класса RecordingPen имеет все возможности классов TextTool и Pen, а также способен запоминать графические операции и последовательно выполнять их позже. Экземпляр класса Commander управляет сразу несколькими перьями.

Как уже отмечалось, к графическому инструменту, созданному по умолчанию при создании любой графической среды, обращаются, посылая графической среде сообщение **pen**. А когда такому перу посылается некоторое рисующие сообщение, результат отображается в связанной с пером графической среде.

При желании можно разорвать связь графического инструмента с его текущей средой и связать его с другой. Следовательно, можно сначала связывать графический инструмент с окном, создать в интерактивном режиме нужное изображение, а потом связать его, например, с графическим принтером, чтобы создать окончательную твёрдую копию изображения.

Экземпляр класса **GraphicsTool** определяет следующие переменные экземпляра:

deviceContext — дескриптор контекста устройства, связанного с графическим инструментом.

graphicsMedium — содержит экземпляр класса **GraphicsMedium**, связанный с графическим инструментом.

width — ширина страницы среды; для окна, это ширина окна. Для принтера, это ширина бумаги.

height — высота страницы среды; для окна, это высота окна. Для принтера, это высота бумаги.

foreColor — цвет, используемый графическим инструментом для рисования.

backColor — цвет фона.

location — текущая позиция, занимаемая графическим инструментом в связанной с ним графической среде.

logicalTool — дескриптор графического инструмента.

Посылая экземпляру одного из подклассов класса **GraphicsTool** сообщения из протокола доступа, селекторы которых совпадают с именами переменных, всегда можно получить значение переменной. Можно и переопределить значение любой переменной, посылая экземпляру ключевое сообщение, представляющее собой имя переменной с конечным двоеточием (кроме переменной **deviceContext**, значение которой устанавливается посредством сообщения **handle:**).

Вся реальная работа по созданию графического образа выполняется GDI-функциями, которые скрыты в методах, написанных на языке **Smalltalk**, так что пользователь может не заботиться о технических тонкостях. Наиболее мощной и эффективной из GDI-функций наверное является функции `Bitblt`. В классе

GraphicsTool, функции `Bitblt` скрыты в методах copy (копировать) и fill (заполнить). Например, метод

```
Display pen copy: Display pen
        from: (0 @ 0 extent: 100 @ 100)
        to: (0 @ 0 extent: 200 @ 200)
```

скопирует часть экрана из прямоугольника 0 @ 0 extent: 100 @ 100 на экран в прямоугольник 0 @ 0 extent: 200 @ 200, используя операцию Bitblt. В этом примере прямоугольник-адресат большие прямоугольника-источника, поэтому прямоугольник-источник будет масштабирован до размеров адресата (увеличен в два раза).

Выражение

```
Display pen place: 100 @ 100; ellipseFilled: 100 minor: 50
```

заполнит эллипс с центром в точке экрана с координатами 100 @ 100 и указанными размерами цветом, хранящимся в переменной backColor (по умолчанию — nil, системный цвет фона, он же задаётся константой ClrBackground, зависит от выбранной цветовой схемы Windows). Конечно же цвет изображения и фоновый цвет можно установить выражениями foreColor: aColor и backColor: aColor. Параметры этих методов — одно из имен цвета (с префиксом Clr), которое является константой из словаря пула ColorConstants:

Таблица 1: Цвета из пула ColorConstants

ClrDefault	ClrBackground	ClrNeutral	ClrHighlight	ClrHighlighttext
ClrGreen	ClrDarkgreen	ClrWhite	ClrPalegray	ClrDarkgray
ClrCyan	ClrDarkcyan	ClrYellow	ClrBlue	ClrDarkblue
ClrRed	ClrDarkred	ClrBrown	ClrPink	ClrDarkpink
ClrBlack				

Вообще говоря, методы foreColor: и backColor: требуют в качестве параметра длинного целого числа, которое может быть или индексом из системной цветовой палитры (цветовые константы типа ClrBackground и ClrHighlight и есть такие индексы) или RGB(Red-Green-Blue)-значением (цветовые константы типа ClrRed и ClrGreen и есть такие значения). Если надо задать явное RGB-значение, следует использовать метод класса из класса GraphicsTool с селектором red:green:blue:, который создаст соответствующее RGB-значение из трех компонент, задаваемых параметрами сообщения, каждый из которых — целое число от 0 до 255. Подробнее об управлении цветами и использовании палитры см. [4, С. 104—112]. Вычисляя выражение

```
Display pen backColor: ClrRed;
        place: 100 @ 100; ellipseFilled: 100 minor: 50;
        backColor: nil  "Восстановить значение по умолчанию."
```

получим эллипс из предыдущего примера, но красного цвета. Тот же результат получиться и при вычислении выражения

```
Display pen backColor: (GraphicsTool red: 255 green: 0 blue: 0);
        place: 100 @ 100; ellipseFilled: 100 minor: 50;
        backColor: nil
```

Когда создаётся графика, можно определять ещё и правило комбинирования (смешивания), которое сообщает графическому интерфейсу, как объединить элементы исходного растра с элементами растра-шаблона (растра-маски) и элементами растра адресата (целевого растра). Для графики, реализуемой классом GraphicsTool, правило смешивания можно установить посредством посылки экземпляру сообщения setRop2: R2Constants, где R2Constants — одна из констант, определяющих комбинационное правило (эти константы определены в словаре пула WinConstants).

3.1 Класс TextTool

Экземпляр класса TextTool в дополнение к тому, что наследует из класса GraphicsTool методы, которые позволяют ему производить заполнение областей и перемещение блоков, ещё может по переданной ему строке рисовать в графической среде глифы (изображения символов) из шрифтов, имеющихся в Windows. Экземпляр класса TextTool обычно используется совместно с окнами, которые содержат экземпляры класса TextPane, в которых невозможна графика.

Наследуемая переменная location используется экземпляром класса TextTool как позиция первого глифа выводимой строки. Поэтому всегда можно послать сообщение, меняющее её, или сделать запрос о её текущей позиции размещения литеры.

Следующий код открывает окно с именем TextWindow, содержащее текстовую панель, и использует связанный с этой панелью экземпляр класса TextTool для того, чтобы в разных точках (позициях) окна отобразить строки, состоящие из литер:

```
| aWindow textTool |
aWindow := TextWindow windowLabeled: 'TextTool Examples'
frame: (100 @ 100 extent: 400 @ 200). "Определить и открыть окно."
aWindow nextPutAll: 'Hi!'; cr.        "Отобразить начальный текст."
textTool := aWindow pane pen.         "Получить доступ к TextTool."
```

Таблица 2: Правила комбинирования

Правило	Результат
R2Black	Пиксел всегда чёрный
R2White	Пиксел всегда белый
R2Nop	Пиксел остаётся неизмененным
R2Not	Пиксел — инверсия адресата
R2Copypen	Пиксел — цвет пера
R2Notcopypen	Пиксел — инверсия цвета пера
R2Mergepennot	Пиксел — комбинация цвета пера и инверсии адресата
R2Maskpennot	Пиксел — комбинация цветов, общих для пера и инверсированного адресата
R2Mergenotpen	Пиксел — комбинация адресата и инверсии цвета пера
R2Masknotpen	Пиксел — комбинация цветов, общих для адресат и инверсии пера
R2Mergepen	Пиксел — комбинация пера и цвета адресата
R2Notmergepen	Пиксел — инверсия R2Mergepen
R2Maskpen	Пиксел — комбинация цветов, общих для пера и адресата
R2Notmaskpen	Пиксел — инверсия R2Maskpen
R2Xorpen	Пиксел — комбинация несовпадающих цветов пера и адресата
R2Notxorpen	Пиксел — инверсия R2Xorpen

```
textTool place: 10 @ 10;                "Определить позицию инструмента."
displayText: 'Welcome'.                 "Отобразить строку в этой позиции."
textTool displayText: 'to'             "Отобразить строку 'to',"
        at: textTool location + (50 @ 30). "переместившись вниз и вправо."
textTool centerText: 'Smalltalk/V'     "Отобразить строку 'Smalltalk/V',"
        at: textTool extent // 2.      "переместившись в центр окна."
"Теперь переместиться ближе к нижнему правому
углу окна и отобразить строку 'Windows'."
textTool lineDisplay: 'Windows'
        at: (textTool width - 40) @ (textTool height - 20).
```

Перед тем, как закрыть окно с именем 'TextTool Examples', нажмите минимизирующую кнопку, а затем двойным щелчком на пиктограмме окна снова восстановите его. Обратите внимание на то, что восстановится только строка "Hi".

Эта строка единственная, записанная в окно методом nextPutAll:, который добавляет аргумент метода в переменную file используемого экземпляра класса TextWindow, чтобы поддерживать состояние текста в панели, когда в ней происходят некоторые события, типа getContents. Остальной текст был записан в панель прямо экземпляром класса TextTool, а он не поддерживает возможности восстановления состояния.

К тем переменным, которые класс TextTool наследует от суперкласса GraphicsTool, он добавляет собственную переменную экземпляра font. Переменная font хранит описание любого доступного шрифта и используется для вывода глифов в связанный с инструментом экземпляр класса GraphicsMedium. В качестве примера использования переменной font, рассмотрим следующий код:

```
| font aWindow textTool |
aWindow := TextWindow windowLabeled: 'Пример TextTool'
frame: (100 @ 100 extent: 400 @ 200).   "Определить и открыть окно."
textTool := aWindow pane pen.           "Получить доступ к TextTool."
font := Font chooseAFont: 'Выберите шрифт, пожалуйста.'.
font isNil ifTrue: [ ↑self].            "Выбрать шрифт."
textTool font: font;                    "Установить в инструменте шрифт"
    foreColor: ClrLightgray;        "и цвет выводимых литер."
    "Отобразить строку 'Hello, World!' в указанной позиции."
    displayText: 'Hello, World!' at: (60 @ (textTool height // 2));
    foreColor: ClrBlack;            "Сменить цвет выводимых литер."
    "Отобразить 'Hello, World!' в новой указанной позиции."
    displayText: 'Hello, World!' at: (60 @ (textTool height // 2 + font height)).
```

Во время выполнения этого кода, возникнет диалоговое окно с текстом 'Выберите шрифт, пожалуйста.'. Выберите шрифт из тех, что будут перечислены в списковой панели окна, и нажмите на кнопку “OK”.

3.2 Несколько слов о шрифтах

Smalltalk/V for Windows представляет литеры в строках, используя их коды ASCII. Чтобы отображать их в среде вывода, их ASCII-значения должны преобразовываться в графические образы (глифы). Класс TextTool выполняет преобразование, вызывая функции GDI Windows, а класс Font обеспечивает информацию о том, какой шрифт используется в этом преобразовании. Сами шрифты обеспечиваются системой Windows. Эти шрифты можно разделить на две категории: растровые шрифты и векторные шрифты. Растровые шрифты отображаются быстрее, но они, в отличие от векторных, не могут масштабироваться и зависят от характеристик используемого графического устройства. Таблица 3 перечисляет

глобальные переменные системы, хранящие шрифты, используемыми различными объектами.

Таблица 3: Стандартные шрифты Smalltalk/V for Windows

Глобальная переменная	Какими объектами используется
ListFont	ListPane
TextFont	TextPane
SysFont	все другие

Можно изменить значения первых двух переменных по своему вкусу, после чего все открываемые новые окна уже будут использоваться новые шрифты. Для этого можно воспользоваться опцией **Fonts...** из Системного меню. При этом появляется стандартное диалоговое окно выбора шрифта, которое позволяет выбирать гарнитуру шрифта, его стиль и его кегль (размер). Шрифт SysFont используется во всех остальных случаях. Для текста, не отображаемого системой Smalltalk (например, имена элементов меню) по умолчанию используется системный шрифт Windows.

Приведём наиболее часто используемые сообщения класса для класса Font:

allFonts — возвращает массив всех доступных шрифтов.

chooseAFont: aTitleString — открывает диалоговое окно, которое позволяет пользователю, выбрать шрифт; возвращает экземпляр класса Font, со шрифтом уже доступным для работы; параметр aTitleString отображается как заголовок диалогового окна (в некоторых версиях Smalltalk игнорируется).

new — создаёт новый объект-шрифт, хранящий атрибуты шрифта в графической оконной системе. Этот шрифт ещё не доступен для работы, он может быть сделан доступным посылкой экземпляру сообщений makeFont или fontHandle.

После создания нового экземпляра шрифта, его необходимо настроить, устанавливая многочисленные атрибуты шрифта. Некоторые атрибуты не используются в Windows, т.к. включены для совместимости с версиями Smalltalk/V для OS/2 и Macintosh — их можно игнорировать. Приведём сообщения экземпляра для установки атрибутов, используемых в Windows. Посылая сообщения с этими же именами, но без двоеточия, можно узнать их значения.

bold: aBoolean — устанавливает полужирный шрифт, когда параметр aBoolean равен true.

italic: aBoolean — устанавливает курсивный шрифт, когда параметр aBoolean равен true.

underscore: aBoolean — устанавливает подчёркнутый шрифт, когда параметр **aBoolean** равен **true**.

strikeOut: aBoolean — устанавливает перечёркнутый шрифт, когда параметр **aBoolean** равен **true**.

fixedWidth: aBoolean — указывает, что задаваемый шрифт должен быть моноширинным, когда **aBoolean** равен **true**.

charSet: aCharSet — задаёт одну из возможных кодировок в Win16:

AnsiCharset — кодировка Windows.

OemCharset — кодировка DOS.

SymbolCharset — кодировка для шрифтов, содержащих неалфавитные (математические и декоративные) глифы.

charSize: aPoint — установка размера шрифта. x-координата задаёт (необязательную) среднюю ширину символов шрифта, y-координата задаёт высоту шрифта.

faceName: aString — установка гарнитуры по её имени. Здесь требуется указать имя гарнитуры шрифта, установленного в систему, или включённого в список подстановок шрифтов (в раздел `[FontSubstitutes]` файла `win.ini`).

Экземпляр класса **Font** может использоваться при выводе текста на экран только после обращения к GDI, позволяя сделать выбранный шрифт (или наиболее схожий с ним) доступным в данной графической среде. Как обычно, при обращении к API оконной системы, выделенные ресурсы после использования надо вернуть. Эту работу выполняют следующие методы.

makeFont: aGraphicMedium — делает доступным новый шрифт для графической среды **aGraphicMedium**.

makeFont — делает доступным новый экранный шрифт.

deleteFont — освобождает дескриптор шрифта Windows, связанный с экземпляром класса **Font**.

fontHandle — возвращает дескриптор шрифта Windows, связанный с экземпляром класса **Font**. Используется для того, чтобы получить дескрипторы шрифтов системы, которые необходимы в Windows для обращений к API.

Заметим, что метод deleteFont *не удаляет* объект класса Font; он освобождает ресурсы Windows, запрошенные для использования шрифта. Сам экземпляр класса Font является обычным объектом Smalltalk и по ненадобности будет утилизирован при сборке мусора. Поэтому если какой-то шрифт будет нужен при многих операциях вывода, имеет смысл один раз создать экземпляр класса Font и описать его как глобальную переменную, а доступ к самому шрифту получать только на время операций вывода в конкретной графической среде (именно это и сделано разработчиками системы, см. приведённую выше таблицу). Идея сразу получить доступ к шрифту и отдать его по завершению работы программы неудачна, так как графические ресурсы крайне ограничены в младших версиях Windows (3.x и 95) и при малейшей неаккуратности это может привести к утечке или исчерпанию ресурсов, что равносильно краху системы.

3.3 Класс Pen

Класс Pen наследует все возможности класса TextTool и, дополнительно, обеспечивает интерфейс «черепашьей графики» для рисования. Поле, например, можно нарисовать, соединяя прямыми линиями точки, начиная с (0,0) через (0,10) и (10,10) к точке (10,0) и, наконец, вновь к (0,0). В «черепашьей» геометрии, рисунок описывается так, как будто вдоль него идёт черепаха, оставляя след своим хвостом, опущенным в чернила.

Помимо позиции, перо (экземпляр класса Pen) хранит направление движения, состояние пера (опущено оно, и потому рисует, или поднято, и потому при перемещении не оставляет следа), ширину и стиль пера.

Переменная location сообщает перу откуда начать следующее перемещение. Когда перо создаётся, оно помещается в центр нового окна (что равносильно посылке перу сообщения home). Направление движения (переменная direction) — целое число от 0 до 359 градусов, с отсчётом по часовой стрелке, при этом направление на «восток» (к правому краю экрана) соответствует 0 градусам, а на «юг» (к нижнему краю экрана) — 90. По умолчанию перо устанавливается на «север» — 270. Пользуясь переменной direction, перо вычисляет конечную точку перемещения, когда посылается сообщение go:, в котором определяется только величина перемещения (единица измерения — пиксел). Если значение переменной downState равно true (истина), то во время своего перемещения перо рисует (оставляет след); иначе оно только перемещается. Состояние пера всегда можно установить:

aPen down — установить состояние пера равным true;

aPen up — установить состояние пера равным false.

Ширина пера — ширина кончика пера (маски) или, другими словами, ширина рисуемой линии. Для того, чтобы установить ширину пера, можно использовать сообщения setLineWidth:, defaultNib:.

Следующий код рисует замкнутый контур, состоящий из прямолинейных отрезков:

```
"Создать окно с графической панелью."
Window turtleWindow: 'Pen Examples'.
"Провести прямую из центра панели в точку с координатами 30 @ 30."
Turtle goto: 30 @ 30;
      box: Turtle extent - 30; "Нарисовать прямоугольник."
      home;                    "Перо возвратить в центр панели."
      defaultNib: 4.           "Установить ширину пера в 4 пиксела."
10 timesRepeat: [               "Повторяя 10 раз,"
   4 timesRepeat:               "рисовать квадрат со стороной в 100"
   [Turtle go: 100; turn: 90].  "пикселов, после прорисовки которого"
   Turtle turn: 36].            "поворачиваться на 36 градусов."
Turtle defaultNib: 1            "Установить начальную ширину пера."
```

Можно рисовать не только прямоугольники и отрезки, но и эллипсы, круги, хорды, сектора. Код следующих выражений демонстрирует некоторые из возможностей, выполняя в отличие от предыдущего примера операции не в окне, а непосредственно на экране:

```
Display pen place: Display extent // 2; "Перо - в середину экрана."
        boxOfSize: 100 @ 100; "Рисует прямоугольник 100 на 100."
        circle: 100;          "Рисует круг радиусом 100 единиц."
        "Рисует эллипс с полуосями 100 (по Ox) и 50 (по Oy)."
        ellipse: 100 minor: 50;
        chord: 200 minor: 100 angles: 0 @ 90; "Рисует сегмент."
        "Последним рисует сектор."
        pie: 150 minor: 150 angles: 0 @ 135.
```

В предыдущих двух примерах рисовались только контуры. Но есть методы, поддерживающие и цвет заполнения и окраску линий.

```
"Создать окно с графической панелью."
Window turtleWindow: 'Pen Examples'.
Turtle fill: ClrYellow;          "Заполнить окно жёлтым цветом."
      home;                      "Перо поставить в центр панели."
      foreColor: ClrDarkgray;    "Установить темно-серый цвет линии."
      defaultNib: 4;             "Установить ширину пера в 4 пиксела."
```

```
boxOfSize: 100 @ 100;     "Нарисовать прямоугольник."
foreColor: ClrRed;        "Изменить цвет линии на красный."
defaultNib: 6;            "Установить ширину пера в 6 пикселов."
home;                     "Перо возвратить в центр панели."
circle: 50;               "Нарисовать круг радиусом 100."
displayText: 'Hello'      "Вывести в окно строку 'Hello'"
        at: 20 @ 20       "в точке с координатами 20 @ 20."
```

Обратите внимание, что класс Pen наследует из класса TextTool возможности по отображению текста. Минимизируйте и восстановите окно 'Pen Examples'. Как и ранее в случае с текстом, рисунок не восстанавливается.

Цветом заполнения области можно распорядиться и по другому. GDI-функции, реально выполняющие графические операции, используют текущую кисть для заполнения внутренних областей изображений, типа эллипсов, прямоугольников, хорд, и т.д. Есть семь предопределённых кистей, которые могут быть выбраны в контексте устройства: GrayBrush, LtgrayBrush, DkgrayBrush, BlackBrush, WhiteBrush, NullBrush, HollowBrush. Следующий пример выбирает серую кисть, чтобы заполнить внутренность прямоугольника

```
|aPen|
aPen := Display pen.
aPen selectStockObject: GrayBrush;
    place: 0 @ 0; boxFilled: 100 @ 100.
```

Можно создавать и собственные однородные или штрихующие кисти. Следующий пример создаёт шаблон штрихующей кисти в виде красного креста:

```
|aPen|
aPen := Display pen.
aPen setHatchBrush: HsCross color: ClrRed;
    place: 100 @ 100; ellipseFilled: 100 minor: 50
```

Метод setHatchBrush:color: создаёт штрихующую кисть в виде прямого креста красного цвета и помещает созданную кисть в контекст графического инструмента. Кисть остаётся активной до тех пор, пока не будет выбрана другая кисть. В дополнение к стилю штрихования HsCross, можно использовать следующие, штрихующие стили: HsHorizontal, HsBdiagonal, HsFdiagonal, HsVertical и HsDiagcross. Эти константы определены в словаре пула WinConstants.

3.4 Класс **RecordingPen**

Чтобы добиться восстанавливаемой графики, сравнимой с восстанавливаемым текстом в текстовых панелях, вместо экземпляров класса Pen надо исполь-

зовать экземпляры класса RecordingPen (ЗаписывающееПеро). Этот класс реализует все возможности своих суперклассов, и вводит дополнительные переменные и методы для регистрации графики, которые позволяют записывать рисунки в графический сегмент и позднее этот сегмент воспроизводить. Графический сегмент есть экземпляр StoredPicture — объектная оболочка над метафайлом используемой оконной системы. Есть три способа нарисовать сегмент:

```
aPenRecorder retainPicture: aBlock
aPenRecorder drawPicture: aBlock
aPenRecorder drawRetainPicture: aBlock
```

Каждое из этих выражений выполнит aBlock, содержащий графические операции. Метод retainPicture: только записывает изображение, но фактически ничего не изображает. Метод drawPicture: только создаёт изображение, но не записывает его, следовательно, повторение невозможно. Только метод drawRetainPicture: и создаёт изображение и записывает его.

С графической панелью в системе Smalltalk/V for Windows связывается именно экземпляр класса RecordingPen (объект Turtle). Поэтому приложение может создавать графический образ, и всякий раз, когда графическую панель нужно повторно отобразить на экране (когда она получает сообщение WmPaint), окно воспроизводится, отображая содержимое графической панели. Когда графическая панель открывается первый раз, её экземпляр класса RecordingPen создаёт начальный графический сегмент и выполняет метод, определяемый селектором, связанным с событием getContents. Графика, создаваемая этим методом регистрируется в начальном сегменте, позже приложение может добавить сюда любое количество сегментов. Когда графическая панель получает сообщение WmPaint, она повторно воспроизводить все сегменты, принадлежащие этому экземпляру класса RecordingPen.

Другой способ отображения содержимого графической панели состоит в том, чтобы выполнить метод, определяемый селектором, связанным с событием display. Этот метод не использует регистрируемую графику; селектор, связанный с событием display выполняется тогда, когда графическая панель вновь должна отображаться на экране. Приложение сможет отвечать на событие display, если в метод прикладного класса, открывающий окно, добавить строчку when: #display perform: #draw:. Подробности, связанные с этими вопросами, можно найти в примерах GraphicsDemo, Dashboard, Puzzle15, FreeDrawing, поставляемых с системой.

В примерах ниже демонстрируются некоторые из возможностей класса RecordingPen. В первом примере

```
Window turtleWindow: 'RecordingPen Examples'.
Turtle drawRetainPicture: [Turtle fill: ClrYellow; home;
```

```
mandala: 16 diameter: 250 ]
```

в центре жёлтого окна нарисует 16-ти угольную мандалу и сохранит рисунок в сегменте, обеспечивая возможность его восстановления. Если теперь изменить размеры этого окна, минимизировать и потом восстановить его, или перекрыть другим окном, а потом вновь сделать активным, первоначальное состояние окна восстановится.

Следующий пример рисует четыре прямоугольных поля, сохраняя их в виде сегмента id, который можно многократно воспроизводить. Затем сегмент id воспроизводится 121 раз, создавая в окне черно-красную шахматную доску.

```
| id |
Window turtleWindow: 'test'.
id := Turtle drawRetainPicture: [ "Нарисовать 4 прямоугольника:"
   Turtle setSolidBrush: ClrBlack;
          place: 0 @ 0; boxFilled: 50  @ 50;
          place: 50 @ 50; boxFilled: 100 @ 100; "два чёрных,"
          setSolidBrush: ClrRed;
          place: 0 @ 50; boxFilled: 50 @ 100;
          place: 50 @ 0; boxFilled: 100 @ 50     "два красных."].
Turtle setMapMode: MmAnisotropic;
  "Установить режим экрана, позволяющий масштабирование."
setWindowExt: 1000 @ 1000;       "Установить размер окна."
setViewportExt: Turtle extent.  "Установить размер области просмотра."
  "Повторно отображать сегмент id, меняя точку начала окна."
0 to: 10 do: [: i | 0 to: 10 do: [: j |
                        Turtle setWindowOrg: (-100 * i) @ (-100 * j).
                        Turtle drawSegment: id]]
```

Следующий сохраняющий графику пример дважды открывает графический сегмент, создаёт в сегменте графический образ и закрывает сегмент, тем самым, создавая в окне графический образ и запоминая его. Всегда можно создать цепочку таких сегментов, а затем всю её повторно воспроизвести. Более того, поскольку используется экземпляр класса RecordingPen, записывающий графику, его можно использовать для вывода графики на принтер, доступный по умолчанию, выполняя выражение вида Printer printWith: aPenRecorder. Такое выражение есть в нашем примере, поэтому, выполняя его, не забудьте включить принтер.

```
Window turtleWindow: 'test'.
Turtle openSegment;              "Начать новый сегмент."
       place: 0 @ 0;             "Нарисовать в сегменте прямоугольник."
       box: 100 @ 100;
```

```
        closeSegment.            "Закрыть первый сегмент."
Turtle openSegment;              "Начать новый сегмент."
        place: 100 @ 100;        "Нарисовать в сегменте круг."
        circle: 50;
        closeSegment.            "Закрыть второй сегмент."
Printer printWith: Turtle.       "Содержимое панели напечатать."
```

Заметим, что каждый сегмент имеет индекс, который можно использовать для того, чтобы отобразить только этот сегмент, посылая перу сообщение drawSegment: index.

Вывести графику в окно и на принтер можно ещё и так:

```
Window turtleWindow: 'test'.
Turtle retainPicture: [ "Создать 4-ре прямоугольных поля:"
   Turtle setSolidBrush: ClrBlack;
          place: 0 @ 0; boxFilled: 50  @ 50;
          place: 50 @ 50; boxFilled: 100 @ 100; "два чёрных,"
          setSolidBrush: ClrWhite;
          place: 0 @ 50; boxFilled: 50 @ 100;
          place: 50 @ 0; boxFilled: 100 @ 50     "два белых."].
Printer printWith: Turtle.        "Содержимое панели напечатать."
```

3.5 Класс **Commander**

Класс Commander — подкласс класса RecordingPen. Экземпляр класса Commander состоит из массива перьев (экземпляров класса RecordingPen), каждое из которых можно установить в некоторую точку окна с некоторым направлением. Класс Commander имеет уникальное сообщение — fanOut, которое располагает веером все находящихся под его управлением перья с направлениями, начиная с 0, отличающимися на 360/число перьев. В классе заново определяются методы, связанные с сообщениями подобными place:, turn:, down, up, go: и goto:, чтобы передавать соответствующее сообщение всем перьям экземпляра. Тем самым, когда единицы перемещения малы, создаётся иллюзия, что все перья рисуют одновременно.

В следующем примере рисуются пять расположенных веером драконовых кривых:

```
Window turtleWindow: 'Commander Dragons'.
  "Создать экземпляр класса Commander,"
(Commander pen: 5
          forDC: Turtle handle
```

```
medium: Turtle graphicsMedium)
up;          "все его перья поднять,"
home;        "поместить в центр окна,"
fanOut;      "установить перья веером"
go: 60;      "каждое перо переместить на 60 пикселов"
down;        "все перья опустить,"
dragon: 9    "нарисовать драконовы кривые."
```

4 Классы AnimatedObject и AnimationPane

Анимация — непрерывное, автоматизированное перемещение графических элементов внутри окна, является уникальной формой графики в Smalltalk/V. Анимация в Smalltalk/V реализована как результат сотрудничества двух классов. Первый класс (AnimatedObject) определяет графические объекты, которые «оживляются» внутри окна. Второй класс (AnimationPane) — это специализированный для анимации подкласс класса GraphPane.

Экземпляр класс AnimationPane может содержать один или несколько экземпляров класса AnimatedObject. В свою очередь экземпляр класса AnimatedObject содержит набор из экземпляров класса Bitmap, называемых кадрами (или фреймами). Кадры экземпляра AnimatedObject описывают упорядоченные изображения объекта в движении, например, «раздувающуюся» букву, или бегущую собаку.

Экземпляр AnimatedObject управляется непосредственно, посредством посылки ему сообщений, которые воздействуют на направление его перемещений, заставляют объект поворачиваться, определяют скорость перемещения и шаги в чередовании кадров. Самым лучшим способом понять анимационные возможности системы Smalltalk/V является экспериментирование.

Приведём два демонстрационных примера. Перед тем как выполнять их, стоит внимательно исследовать исходный текст методов в классах AnimatedObject и AnimationPane. Это поможет понять основные идеи того, как они работают.

Следующий код создаёт окно анимации, «отскакивающий» (bouncer) экземпляр класса AnimatedObject (он умеет отражаться от рамок окна), помещает последний в окно и запускает анимационный процесс.

```
Animator1 := AnimationPane openWindow: 'Animation Examples'.
Bouncer := AnimatedObject new
        bouncer;          direction: 45;
        speed: 1;         stepsPerFrame: 1;
        frames: Pictures; position: Animator pen extent // 2.
```

```
Animator1 addObject: Bouncer.
Bouncer animate
```

Можно изменять поведение экземпляра AnimatedObject, прямо «на лету» посылая ему подходящие сообщения. Например, разместите рабочее окно так, чтобы оно не перекрывало окно предыдущего примера, наберите в рабочем окне следующие строки

```
Bouncer speed: 5.            Bouncer direction: 95.
Bouncer stepsPerFrame: 16.  Bouncer stop.
```

А теперь снова выполните код предыдущего примера, и выбирая и выполняя по одной строке в рабочем окне, посмотрите на последствия их выполнения.

Другой пример анимации в системе Smalltalk/V — «ведомый» (chaser) экземпляр класса AnimatedObject. Когда экземпляру класса AnimatedObject сообщают, что он будет «ведомым» объектом, его перемещения следуют за перемещениями курсора, управляемыми мышью. Вычисление следующего кода добавит в систему кадры для анимации собаки, которые созданы заранее и поставляются вместе с системой.

```
"Загрузить образы собаки из bmp-файлов на диске."
| dogPictures |
TutorialPictures := Dictionary new.
dogPictures := Array new: 4.
  1 to: dogPictures size do:  [ :i |
        dogPictures at: i put: (Bitmap
           fromFile: 'tutorial\dog', i printString,'.bmp')].
TutorialPictures at: 'dogs' put: dogPictures
```

Следующий код создаёт окно для анимации, «ведомый» (chaser) экземпляр класса AnimatedObject, помещает последний в окно и запускает анимационный процесс.

```
| dogs |
Animator2 := AnimationPane openWindow: 'Animation Examples'.
dogs  := TutorialPictures at: 'dogs'.
Chaser := AnimatedObject new
          chaser; direction: 45; frames: dogs.
Animator2 addObject: Chaser.
Chaser animate
```

Можно в одном окне запускать объекты разных типов. Попробуйте сделать это самостоятельно.

Bouncer и Chaser — только два способа «оживления». Расширяя класс AnimatedObject, можно реализовать любое поведение, которое удаётся вообразить, например, создать «отталкивающие» (Repellers) или «привлекательные» объекты (Attractors).

Ещё раз напомним, что глобальные переменные и растры занимают предназначенные только для них области памяти. Чтобы освободить память, выделенную Windows для растров анимационных объектов, надо вычислить выражения следующего вида:

```
Bouncer frames do: [:each | each release].
Chaser frames do: [:each | each release].
```

А чтобы позволить Smalltalk уничтожить экземпляры анимационных классов, надо вычислить следующее выражение:

```
Animator1 := Animator2 := Chaser := Bouncer := nil
```

В результате ранее связанные с глобальными переменными объекты становятся доступными для сборки мусора.

5 Замечание о работе с курсорами

Чтобы визуально указать на состояние системы, Smalltalk/V использует различные формы курсора. Например, форма курсора в виде песочных часов используется для того, чтобы указать на происходящие вычисления. Формы курсора — хороший способ передачи информации о приложениях.

Курсоры системы Smalltalk/V управляется классом CursorManager. CursorManager просто обеспечивает интерфейс между системой Smalltalk/V и мышью.

Обычно, когда что-либо отображается поверх курсора, сначала нужно скрыть курсор; иначе курсор может вмешаться в происходящее. Чтобы упростить эту проблему, все примитивы системы, которые изменяют текущее содержимое экрана, сначала проверяют что курсор скрыт, а после сделанных изменений восстанавливают курсор. Выражение Cursor hide скрывает курсор, Cursor display — отображает его. Эти два сообщения работают подобно круглым скобкам: они должны быть сбалансированы. Например, предположим, что курсор в настоящее время отображается; если его дважды скрыть, а затем один раз отобразить, курсора видно не будет, надо отобразить его ещё раз. Если курсор не отображается и нет уверенности, в каком состоянии он находится в настоящее время, следует вычислить выражение: Cursor reset. После чего восстановится равновесие скрывающих и отображающих сообщений и курсор появиться на экране.

5.1 Формы курсора

Чтобы изменить форму курсора, надо просто посылать сообщение о желаемом изменении. Например выражение

```
CursorManager execute change
```

изменяет форму курсора на песочные часы (что на языке курсора означает «ждите, выполняются вычисления»). Если форму курсора надо изменить временно, например, только на время выполнения блока aBlock, можно вычислить выражение

```
CursorManager origin changeFor: aBlock.
```

Smalltalk/V включает несколько форм курсора, доступ к которым осуществляется посредством сообщений, посылаемых CursorManager:

arrow — стрелка, указывающая на северо-запад;

crossHair — курсор-перекрестие;

execute — песочные часы;

normal — стрелка, указывающая на северо-запад;

origin — двунаправленная стрелка, указывающая на юго-запад;

text — карет (курсор, похожий на букву `I`),

Чтобы форму курсора, определяемую операционной системой, сделать доступной системе Smalltalk/V, надо сначала выполнить следующий код:

```
CursorConstants at: Cross put: (CursorManager new
                handle: (CursorManager getWinCursor: IdcCross))
```

а затем создать метод класса в классе CursorManager:

```
cross
    ↑ Cross
```

Теперь можно вывести новый курсор на экран, вычисляя выражение CursorManager cross change. Курсор IdcCross определяется в словаре WinConstants. Там же есть и несколько других курсоров, идентифицированных приставкой Idc.

6 Пример: сеть, состоящая из узлов

Как заключительный пример построения изображения в окне, в котором кроме графических классов активно используются потоки и наборы, построим сеть, состоящую из узлов, и научимся определять путь в сети от одного заданного узла до другого. Многие встречающиеся на практике задачи могут быть описаны в терминах узлов сети и путей в сети, типа маршрутов и диаграмм.

6.1 Класс **Network**

Сеть (network) — набор узлов, которые соединены или не соединены друг с другом. Создадим класс Network как подкласс класса Object, и определим в нем единственную переменную экземпляра с именем connections (связи). Когда класс создан, его определение в окне просмотра Иерархии Классов должно быть таким:

```
Object subclass: #Network
   instanceVariableNames: 'connections '
   classVariableNames: ' '
   poolDictionaries: ' '
```

Переменная экземпляра connections будет содержать словарь соединений между узлами. Ключом в словаре будет узел, а значением — множество всех тех узлов, которые связаны (соединены) с ключом. Поэтому метод экземпляра, определяющий переменную экземпляра будет таким

```
initialize
 "Инициализировать переменную connections пустым словарем."
  connections := Dictionary new
```

Следовательно, для того, чтобы создать новый экземпляр класса Network с именем Net и тем самым сделать его глобальной переменной, надо вычислить выражение

```
Net := Network new initialize.
```

Кроме инициализирующего метода, создадим ещё следующие методы экземпляра:

```
connect: nodeA to: nodeB
 "Связать в сети узел nodeA с узлом nodeB."
  (connections at: nodeA
      ifAbsent: [connections at: nodeA put: Set new]) add: nodeB.
  (connections at: nodeB
      ifAbsent: [connections at: nodeB put: Set new]) add: nodeA

pathFrom: nodeA to: nodeB avoiding: nodeSet
 "Найти путь, который соединяет узел nodeA с узлом nodeB и не
  проходит через узлы из набора nodeSet. Этот путь возвращается
  как новая сеть. Возвратить nil, если такого пути нет."
  | answer |
  nodeSet add: nodeA.
  (connections at: nodeA ifAbsent: [ ↑nil]) do:
     [:node| node = nodeB
```

```
            ifTrue: [ ↑Network new initialize connect: nodeA to: node].
            (nodeSet includes: node)
                ifFalse: [answer := self pathFrom: node
                                              to: nodeB
                                        avoiding: nodeSet.
                          answer isNil
                            ifFalse: [ ↑answer connect: nodeA to: node]]].
  ↑nil

printOn: aStream
 "Напечатать описание приёмника в поток aStream."
  connections keys asSortedCollection do:
      [:node | node printOn: aStream.
               (connections at: node) asSortedCollection do:
                   [:neighbor| aStream cr; nextPutAll: '>>'.
                               neighbor printOn: aStream].
               aStream cr]
```

Обратите внимание на рекурсию в методе pathFrom:to:avoiding:. Предлагаемый алгоритм поиска прост; он не ищет оптимального (самого короткого) пути. Если надо найти оптимальный маршрут, метод надо изменить. Это одна из особенностей языка Smalltalk/V: всегда можно быстро построить некоторые работающие фрагменты модели, чтобы, опираясь на них, подробно изучить решаемую задачу. Когда задача понята, можно быстро сделать необходимые изменения.

6.2 Класс NetworkNode

Перед тем, как использовать класс Network, определим класс NetworkNode как подкласс класса Object, с двумя переменными экземпляра name (имя) и position (местоположение) и с доступными классу словарями пула WinConstants и ColorConstants. Когда класс создан, его определение в окне просмотра Иерархии Классов должно быть таким:

```
Object subclass: #NetworkNode
    instanceVariableNames: 'name position '
    classVariableNames: ' '
    poolDictionaries: 'WinConstants ColorConstants '
```

Переменные экземпляра name, position и словари WinConstants и ColorConstants потребуются и при построении графического образа сети. Определим в этом классе следующие методы экземпляра:

```
<= aNode
  "Возвращает true, если имя приёмника сообщения
   меньше или равно имени аргумента aNode."
   ↑name <= aNode name

hash
  "Возвращает хеш-значение приёмника."
  ↑name hash

name: aString position: aPoint
  "Определяет значения для переменных приёмника."
   name := aString.
   position := aPoint

name
  "Возвращает имя приёмника."
  ↑name

position
  "Возвращает местоположение приёмника."
  ↑position

printOn: aStream
  "Печатает описание приёмника в поток aStream."
  aStream nextPutAll: 'Node(', name; space;
                          nextPutAll: position printString;
                          nextPut: $)
```

Таким образом, чтобы создать новый экземпляр класса NetworkNode с именем aString и расположением в точке aPoint, надо вычислить выражение NetworkNode new name: aString position: aPoint.

6.3 Построение сети

После того, как введены основные классы создадим сеть, вычислив следующее выражение:

```
Net := Network new initialize
```

Затем вычислим выражения, которые создадут шесть узлов и соединят их в сеть, причем каждый узем сети создадим как глобальную переменную системы:

```
N1 := NetworkNode new name: 'one' position: 100 @ 100.
N2 := NetworkNode new name: 'two' position: 150 @ 150.
N3 := NetworkNode new name: 'three' position: 200 @ 120.
N4 := NetworkNode new name: 'four' position: 50 @ 50.
N5 := NetworkNode new name: 'five' position: 125 @ 220
N6 := NetworkNode new name: 'six' position: 260 @ 120.
Net connect: N1 to: N2;  connect: N2 to: N3;
    connect: N4 to: N5;  connect: N5 to: N1;
    connect: N3 to: N6;  connect: N3 to: N5;
    connect: N3 to: N1.
```

Вычисляя выражение `Net pathFrom: N1 to: N5 avoiding: Set new`, мы найдем путь от `N1` до `N5`. Чтобы найти путь между теми же узлами, но который не проходит через узел `N3`, надо вычислить выражение:

```
Net pathFrom: N1 to: N5 avoiding: (Set with: N3)
```

6.4 Построение графического образа сети Net

Пользуясь всеми графическими методами можно нарисовать диаграмму сети `Net`. Для этого добавим в классы `NetworkNode` и `Network` методы, позволяющие нарисовать, соответственно, узел и сеть

Сначала в класс `NetworkNode` добавим метод

```
drawWith: aPen
  "Рисует указанным в качестве параметра пером aPen
   узел-приемник в виде достаточно большого эллипса
   желтого цвета, содержащего в центре имя узла."
| major minor |
  "Сначала вычислить размеры эллипса."
  major := (SysFont stringWidth: name) + 16//2.
  minor := SysFont height + 16 // 2.
  aPen setTextAlign: TaTop;  "Теперь нарисовать узел."
       setSolidBrush: ClrYellow;  place: position;
       ellipseFilled: major minor: minor;
       centerText: name.
```

В этом методе есть незнакомое сообщение — `setTextAlign: aFlag`, которое определяет характер выравнивания текста (в данном случае имени) при выводе в графическую панель. Параметр `aFlag` может принимать значение одной из следующих констант, которые храняться в словаре `WinConstants`: `TaBaseline`, `TaBottom`,

TaCenter, TaLeft, TaNoupdatecp, TaRight, TaTop, TaUpdatecp. Поэкспериментируйте с этими константами и определите характер описываемого ими выравнивания.

Теперь в класс Network добавим метод метод draw, чтобы, используя метод drawWith:, нарисовать все узлы, составляющие сеть, и линии между узлами, согласно связям, определенным в экземпляре. Обратите внимание, что каждое выражение, рисующее линию или узел, фиксируется как сегмент посредством его включения в блок-параметр метода drawRetainPicture:.

```
draw
 "Нарисовать сеть. Для каждого узла нарисовать все его
  связи, а затем (сверху проведенных линий!) сам узел.
  Все посещенные узлы запоминаются во временной переменной
  visited, чтобы избежать их дублирования при рисовании."
| visited pen |
pen := (Window turtleWindow: 'Net') pen.
pen erase.
visited := Set new.
pen drawRetainPicture:
    [connections keys do:
       [:nodeA | visited add: nodeA.
                 (connections at: nodeA) do:
                       [:nodeB | (visited includes: nodeB)
                                 ifFalse: [pen place: nodeA position;
                                           goto: nodeB position]].
       nodeA drawWith: pen]]
```

Теперь, чтобы увидеть рисунок сети Net, который создается пером, сохраняющим графику, надо вычислить выражение: Net draw.

Список литературы

[1] Goldberg A., Robson D., Smalltalk-80. *The language.* — Addison-Wesley Publishing Company, 1988.

[2] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык* Smalltalk. *Ядро графики: классы* Point, Rectangle, Form. — Ростов-на-Дону: УПЛ РГУ, 1997

[3] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык* Smalltalk. *Ядро графики: класс* BitBlt. — Ростов-на-Дону: УПЛ РГУ, 1997

[4] Фролов А.В., Фролов Г.В., *Графический интерфейс GDI в MS Windows*. — М.: «Диалог-МИФИ», 1994. — 228 с. — (Библиотека системного программиста; Т.14)

Список иллюстраций

Список таблиц

Содержание