Максим Мозговой

САМОУЧИТЕЛЬ

Занимательное программирование

Прочитав эту книгу, вы:

- познакомитесь с основами компьютерного моделирования;
- сумеете написать свой архиватор и простую компьютерную игру;
- узнаете, как работает машинная графика, и многое, многое другое!

ПИТЕР

Максим Мозговой

САМОУЧИТЕЛЬ

Занимательное программирование

ПИТЕР®

Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Ростов-на-Дону • Екатеринбург • Самара
Киев • Харьков • Минск
2005

Мозговой Максим Владимирович

Занимательное программирование: Самоучитель

Главный редактор	*Е. Строганова*
Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*В. Рычков*
Литературный редактор	*В. Рычков*
Художник	*Н. Биржаков*
Корректоры	*С. Беляева, И. Смирнова*
Верстка	*Ю. Сергиенко*

ББК 32.973-018я7
УДК 681.3.06(075)

Мозговой М. В.

М74 Занимательное программирование: Самоучитель. — СПб.: Питер, 2005. — 208 с.: ил.

ISBN 5-94723-853-5

Эта книга — попытка предоставить качественную пищу для ума начинающим программистам, желающим достичь высот мастерства.

Вы не найдете в ней описаний конкретных языков программирования вроде Pascal или C++, не встретите руководств по созданию библиотек DLL и системных служб. Даже такой интригующий вопрос, как структура заголовка HTTP-пакета, незаслуженно обойден вниманием. Зато к вашим услугам семь глав, охватывающих самые разнообразные логические направления программирования, масса интересных теоретических материалов и готовых к использованию листингов, а также «проекты для самосовершенствования» — идеи для досуга с компьютером

Приятного чтения!

ISBN 5-94723-853-5

ООО «Питер Принт». 194044, Санкт-Петербург, пр. Б. Сампсониевский, д. 29а.
Лицензия ИД № 05784 от 07.09.01.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 953005 — литература учебная.
Подписано в печать 28.10.04. Формат 70×100/16. Усл. п. л. 16,77. Доп. тираж 3000 экз. Заказ № 916.
Отпечатано с фотоформ в ФГУП «Печатный двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Содержание

Предисловие

Книгу, которую вы держите в руках, можно было бы озаглавить «Программирование: вторая ступень мастерства». Мне кажется, что человек, неплохо разбирающийся хотя бы в одном языке программирования и визуальной среде разработки (такой, как Visual Basic или Delphi) и знающий простые алгоритмы вроде сортировки массива или обхода бинарного дерева, может смело считать себя одолевшим первую ступень нашего искусства. Термины «объект», «рекурсия» или «динамические структуры данных» уже не вызывают сомнений; на вопрос: «умеете ли вы программировать?» он смело отвечает: «да»; ему не составит слишком большого труда написать программу для учета библиотечных карточек или поиска файлов на винчестере. А что же дальше? Еще свежи в памяти воспоминания, когда я сам задавался таким вопросом. В мире информатики, программирования столько интересного! А скольких применений своим талантам мы вовсе и не замечаем... но, к сожалению, кругозор любого человека ограничен.

Здесь я постарался рассказать о темах, занимавших меня в последние годы. Знакомство с ними по-настоящему расширило сферу моих интересов, и мне очень хочется поделиться впечатлениями с вами. Эта книга — не учебник, посвященный какому-либо языку программирования, среде разработки или новомодной технологии; не сборник готовых программ и алгоритмов на каждый день и уж точно не пособие в стиле «освой что-нибудь за 21 день». Я искренне старался написать «книгу для развития личности» (в некотором смысле). Получилось или нет — судить вам.

О выборе языка

Все листинги программ в книге приведены на языке Object Pascal (среда программирования Delphi). Я использовал седьмую версию Delphi [1], но программы старался писать как можно проще, не «завязывая» их ни на какие особенности последних версий компилятора и библиотек. Поэтому, в принципе, все должно работать без особых изменений и на третьей (в крайнем случае, на пятой) версии Delphi.

Почему Delphi? Начнем с того, что мой любимый инструмент разработки — Borland C++ Builder; я отнюдь не отношу себя к пламенным поклонникам Объектного Паскаля. Тем не менее выбираю более демократичную среду Delphi, потому что не хочу отпугнуть некоторых читателей языком C++, ибо бытует мнение, что язык C++ очень сложен, да и не все его знают. Другие языки, пожалуй, известны еще меньше или не нравятся лично мне. В принципе, все листинги можно было бы сочинить даже на QuickBasic, но DOS — это все-таки уже день вчерашний (если не позавчерашний). Язык как таковой в этой книге не так важен; поняв суть алгоритма, вы легко сможете переписать его на своем любимом языке. Тексты же программ на Паскале могут читать, наверное, все (к тому же неочевидные моменты я постараюсь пояснять везде, где смогу). С другой сторо-

---

[1] *Елманова Н., Трепалин С., Тенцер А.* Delphi и технология COM (+CD), СПб.: Питер, 2003.

ны, как уже говорилось, эта книга не учебник Delphi: я предполагаю, что вы знакомы с идеологией программирования в подобных средах.

О структуре книги

Интересное вовсе не обязательно оказывается сложным. Первая глава является по существу сборником простых, но в то же время интересных программ, вполне посильных даже для начинающих. Уверен, многие из приведенных примеров могут быть с успехом использованы в качестве упражнений при обучении программированию вместо стандартных учебниковских задачек (порою невыносимо скучных). Большинство программ первой главы – модели, иллюстрирующие ход какого-нибудь реального процесса. Последние модели («черепашья графика», «конечный автомат») посвящены абстрактным, но очень полезным понятиям, в чем вы в скором времени убедитесь.

Вторая глава посвящена принципам создания компьютерной анимации и графических эффектов. Занимаясь компьютерной анимацией, я сталкивался со множеством подводных камней; судя же по кое-каким вполне известным играм (разумеется, анимация чаще всего используется в компьютерных играх), многие другие тоже. Понимаю, история нас ничему не учит, но все же прочитайте вторую главу, прежде чем набивать шишки самостоятельно. Возможно, вы сэкономите свое время и силы. О графических эффектах разговор отдельный. Вы можете рассматривать их как готовые куски кода, которые не возбраняется просто скопировать в свою программу, но моей главной целью было описание идей, что могут быть использованы и в будущем, использованы новыми разработчиками для создания эффектов, не известных доселе. Навряд ли можно придумать что-то новое, не познакомившись с тем, что уже есть и работает.

Третья глава связана с числом «три» не только потому, что она третья, но и потому, что посвящена трехмерной графике. Основные идеи этой главы – матричные вычисления, а также ортогональное и перспективное проецирование. Надеюсь, эти приемы останутся у вас в памяти, даже если вы забудете все остальное содержание главы. К сожалению, без использования серьезной геометрии (одно упоминание о которой у многих вызывает невыразимую тоску) в трехмерной графике далеко не уйдешь. Поэтому глава и получилась небольшой.

В четвертой главе речь идет о лабиринтах. В ней вы найдете ответы на вопросы о том, как создать лабиринт автоматически, как найти в нем путь из пункта A в пункт B и в чем преимущества одних методов создания лабиринтов и поиска пути над другими. Вы увидите, что алгоритмы поиска пути – это нечто большее, чем кажется с первого взгляда. Возможно, сначала ваши подозрения будут смутными и туманными, но после прочтения шестой главы они окончательно прояснятся.

Пятая глава (пожалуй, наиболее теоретизированная) посвящена сжатию данных. Красота алгоритма сжатия сродни красоте шахматной партии – лишь немногие способны оценить ее по достоинству. Читая отдельные статьи об алгоритмах вроде LZW, вы никогда не получите полноценного представления о том, что есть искусство сжатия данных. Кроме того, уважаемые авторы подобных статей часто увлекаются, безусловно, полезными для разработчика техническими деталями,

мешающими, однако, разглядеть лес за деревьями. Пятая глава – это взгляд с птичьего полета на архивацию. В ней нет листингов, зато есть пища для ума. Надеюсь, ценителям глава придется по вкусу.

Шестую главу, описывающую алгоритмы на графах, меня уговорили написать уже тогда, когда все остальные главы были готовы. Я сопротивлялся, ибо в каждом втором учебнике что-нибудь про графы есть обязательно. С другой стороны, ситуация схожа со сжатием данных: чтобы получить общее представление, надо собирать сведения по крупицам из разных учебников (или прочитать одну толстенную монографию, язык которой лишь математики поймут). Если темы вроде задачи Прима–Краскала или алгоритма Дейкстры затрагиваются часто (я тоже решил не нарушать традицию), то, к примеру, алгоритм эвристического поиска A* так просто не найдешь. На обзор «с птичьего полета» я не решился, а вот хорошее описание некоторых весьма полезных алгоритмов из теории графов обещаю.

Седьмая глава – это компьютерные игры (или Как Это Делается). Здесь вы найдете описание четырех простых игр – Сапера, Сокобана, Удава и Тетриса. Конечно, ничего выдающегося в книгу скромного объема и не очень-то связанного с играми направления не запихнешь, однако даже такие простые игры (вкупе с пасьянсами, Color Lines и Кликоманией) в некоторых довольно обширных кругах в десять раз популярнее самой современной стратегии реального времени или 3D-стрелялки. Мне любая компьютерная игра интересна как настоящая система, как подлинная синергетика алгоритмов, графики, интерфейса пользователя и звука (а уж с «занимательностью» никто не поспорит).

В конце каждой главы есть раздел под названием «Проекты для самосовершенствования». Если после прочтения той или иной главы вас не покидает чувство неудовлетворенности, обратите на предлагаемые проекты самое пристальное внимание – надеюсь, вы не будете разочарованы.

Благодарности

Почти все материалы первой главы – заслуга людей, с которыми мне довелось общаться в 1997–2000 годах; в первую очередь это преподаватели Северо-Осетинского государственного университета Валерий Абрамович Оргун и Марья Дмитриевна Макаренко. Огромное спасибо им за то, что они делали тогда и продолжают делать сейчас; кроме того, спасибо за разрешение использовать их наработки в книге.

Многие программы мне пришлось написать, когда я преподавал на компьютерных курсах для старшеклассников в 1998–2000 годах. Если бы не эти курсы, если бы не ребята, которые заставляли постоянно придумывать что-нибудь новое и непременно «занимательное», книги бы вообще не было. Спасибо им!

Работа с издательством «Питер» оказалась удовольствием. Главная заслуга в этом принадлежит руководителю проектов Владимиру Николаевичу Рычкову, постоянно поддерживавшему меня в работе. Его благосклонность к идее книги придала мне уверенности и сил, без которых заниматься писательством просто невозможно.

От издательства

Заголовок этой книги вовсе не предусматривает какую-либо несерьезность. Но писать о сложном занимательно способны лишь настоящие ученые и хорошие писатели. Будем надеяться, автор этой книги заслуживает такой оценки. Прочтите эту книгу – и убедитесь сами. Если вы начинаете свой путь в программировании – эта книга для вас. Если вы серьезный программист, но не вполне знакомы с проблемами, которым посвящена эта книга, – вам тоже стоит ее прочесть. Если вы преподаете – эта книга тем более для вас!

Максим Владимирович Мозговой родился в 1981 году – в прошлом веке, остаток которого он упорно трудился, изучая курс начальной, средней и высшей школы и преподавая математику и программирование (C/C++, олимпиадные задачи, системное программирование, компьютерная анимация). Результаты учения отмечены золотой медалью гимназии во Владикавказе, дипломом с отличием факультета прикладной математики и процессов управления Петербургского государственного университета и званием «Чемпиона республики по информатике среди школьников». Трудно только первые сто лет, и начало нового века автор этой книги встретил в качестве (одновременно!) магистра, аспиранта и исследователя двух университетов – в России и в Финляндии. Предмет исследований – синтез речи, многопоточное программирование. Автор действительно не только настоящий ученый, но и хороший преподаватель. Плохого система просто не пустит в свои ряды. Ну, а писатель... Две книги в соавторстве – пора издать третью, соло. Что мы и сделали, к вашему, надеемся, удовольствию!

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу http://www.piter.com/download.

На веб-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Глава 1

Компьютерное моделирование

Если факты противоречат моей теории,
тем хуже для фактов.

Георг Вильгельм Фридрих Гегель

По личному мнению автора, компьютерное моделирование – одно из самых интересных применений информационных технологий в мире, начиная с самого начала компьютерной эры и до наших дней. Под моделированием здесь понимается имитация некоторых вполне реальных явлений при помощи компьютера.

Зачем и кому это может понадобиться? Финансовый аналитик прикидывает, сколько денег заработает (или потеряет) его фирма, если вложит миллион долларов в акции такой-то компании; инженер-ракетостроитель рассчитывает характеристики нового двигателя; специалисты-схемотехники разрабатывают архитектуру компьютерного процессора следующего поколения. Каждому из них в своей работе приходится ставить эксперименты: ни один космонавт не рискнет лететь в ракете, не прошедшей испытания, а производитель процессоров не будет выпускать новую модель на рынок, не испытав ее как следует (а то потом себе дороже обойдется).

К сожалению, не всегда эксперимент можно поставить. Это может быть очень дорого или опасно для жизни (если речь идет, к примеру, о ядерном реакторе), а то и вовсе невозможно. Тот же финансовый аналитик не может позволить себе «просто вложить деньги, а там посмотреть, что получится». В таких случаях обычно пытаются сымитировать реальную ситуацию на компьютере и тем самым с минимальными затратами провести «почти настоящий» опыт. Чтобы не быть голословным, приведу пару цитат:

> *«В современных условиях, когда подписан Договор о полном запрещении ядерных испытаний, центр тяжести исследований по оборонно-ядерному направлению переносится в расчетно-теоретические структуры и экспериментальные лаборатории института. То же самое происходит во всех ядерных центрах мира».*
>
> (по материалам сайта Российского федерального ядерного центра)

«На фоне неослабевающей остроты сердечно-сосудистых заболеваний актуально всемерное развитие количественных методов исследования механизмов кровообращения. Становится очевидным, что успешное решение данной проблемы возможно только при применении методов математического и компьютерного моделирования механизмов кровообращения с учетом закономерностей возбуждения сердечной ткани».

(Труды конференции, посвященной 90-летию А. А. Ляпунова, г. Новосибирск)

Разумеется, компьютерная модель — это лишь модель, и никто не гарантирует, что на практике все произойдет именно так, как доложил компьютер. Но если модель хороша (то есть достаточно близка к действительности), то она может быть очень и очень полезной. Создание такой модели требует тщательного изучения ситуации, отделения важных параметров от второстепенных, консультаций со специалистами в изучаемой области.

Модели, описываемые в этой книге, предельно упрощены, а поэтому весьма далеки от реальности. Впрочем, сейчас нас интересуют общие принципы построения моделей, а не создание хорошей модели для применения в какой-то узкой области. «Настоящие» модели, как правило, связаны с обработкой огромного количества чисел, а результатом их работы является не менее солидный набор других чисел. Иными словами, разрабатывать модели гораздо интереснее, чем наблюдать за работой программы, которая эти модели просчитывает. Наши модели, напротив, будут настолько визуальными, насколько это возможно.

Модель 1. «Молекула газа в закрытом сосуде»

Эта модель важна сразу по двум причинам: во-первых, она открывает целую серию подобных ей моделей, в которых используются сходные идеи и технологии; во-вторых, она демонстрирует конкретные воплощения этих идей на Delphi.

Итак, представьте себе аквариум, в котором нет ничего, кроме одной-единственной молекулы газа (кислорода, водорода — неважно), одиноко бороздящей просторы этого самого аквариума. На практике такой ситуации, конечно, не достичь, но по теории поведение молекулы предсказать несложно. Она будет двигаться равномерно и прямолинейно, пока не достигнет стенки аквариума, затем отразится от нее (угол падения равен углу отражения) и полетит теперь уже в другую сторону. Иными словами, молекула ведет себя примерно так же, как бильярдный шар на доске без луз и других шаров.

Уже к этому моменту я сделал несколько неявных предположений:

1) молекула ни с чем не взаимодействует, окружающая среда не оказывает на нее никакого влияния (именно поэтому она двигается равномерно и прямолинейно);
2) столкновение со стенкой аквариума упругое — не меняет энергию молекулы и модуль скорости (изменяется только направление);

3) наш аквариум плоский, как бильярдный стол, а не трехмерный[1] (рис. 1.1).

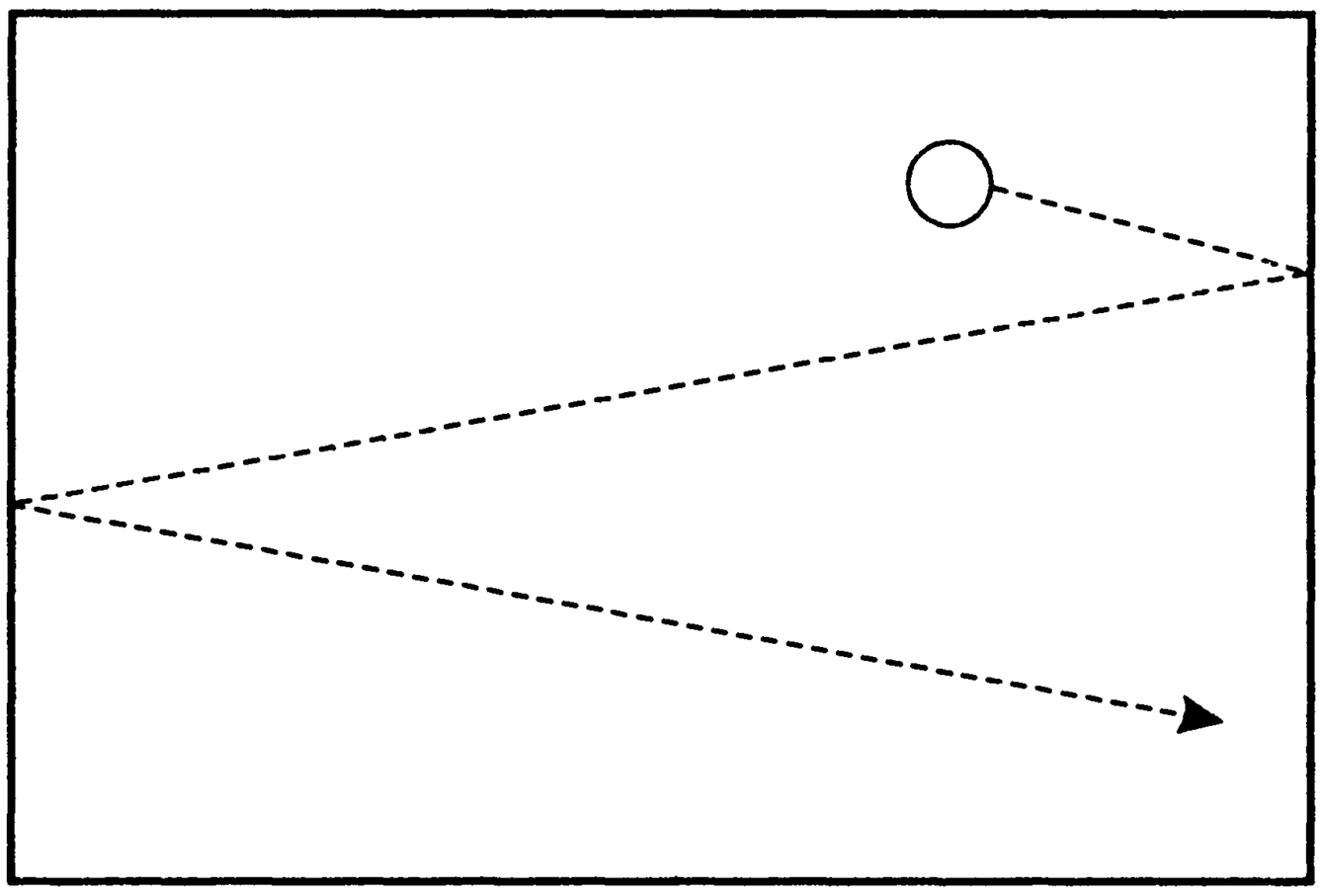


Рис. 1.1. Поведение молекулы газа в закрытом сосуде

На псевдокоде алгоритм движения молекулы выглядит так:

```
разместить молекулу в случайной точке (X, Y) аквариума
выбрать для молекулы случайное направление движения
определить, чему равны горизонтальная (Vx) и вертикальная (Vy) составляющие скорости
молекулы
ЦИКЛ
    стереть молекулу с экрана[2]
    вычислить новую позицию молекулы: X := X + Vx; Y := Y + Vy
    ЕСЛИ молекула вышла за левую или правую границу аквариума[3]
        вернуть ее к ближайшей допустимой точке
        изменить горизонтальную скорость на противоположную (Vx := -Vx)
    ЕСЛИ молекула вышла за верхнюю или нижнюю границу аквариума
        вернуть ее к ближайшей допустимой точке
        изменить вертикальную скорость на противоположную (Vy := -Vy)
    нарисовать молекулу на новой позиции
КОНЕЦ ЦИКЛА
```

Даже в таком несложном алгоритме есть свои тонкости.

Во-первых, что подразумевается под определением горизонтальной и вертикальной составляющей скорости молекулы? Для начала договоримся, что скорость молекулы не зависит от того, в какую сторону она летит (что логично). На уровне программы конкретное значение скорости должно быть определено соответствующей константой. Изначальное направление молекулы можно определить как угол отклонения направления ее движения от оси абсцисс (значение от нуля до

---

[1] Это, кстати, мало что меняет. Третье измерение моделируется совершенно аналогично, но вывести трехмерную картинку на экран уже не так просто. Этому посвящена третья глава книги.

[2] По правде говоря, в этот момент времени молекула еще не нарисована на экране. Впрочем, нет большой беды в том, что мы сотрем несуществующую молекулу.

[3] Обратите внимание: время в программе является дискретной величиной. Мы как бы делаем фотоснимки молекулы через определенный временной интервал на каждой итерации цикла. Поэтому есть вероятность упустить момент удара молекулы о стену аквариума, тем самым позволив молекуле вылететь наружу.

360 градусов). Предположим, что скорость молекулы равна V[1], а изначальное направление (то есть угол отклонения) равен α. Тогда для того, чтобы получить интересующие нас значения составляющих (рис. 1.2), можно воспользоваться формулами:

```
Vx := V * Cos(α)
Vy := V * Sin(α)
```

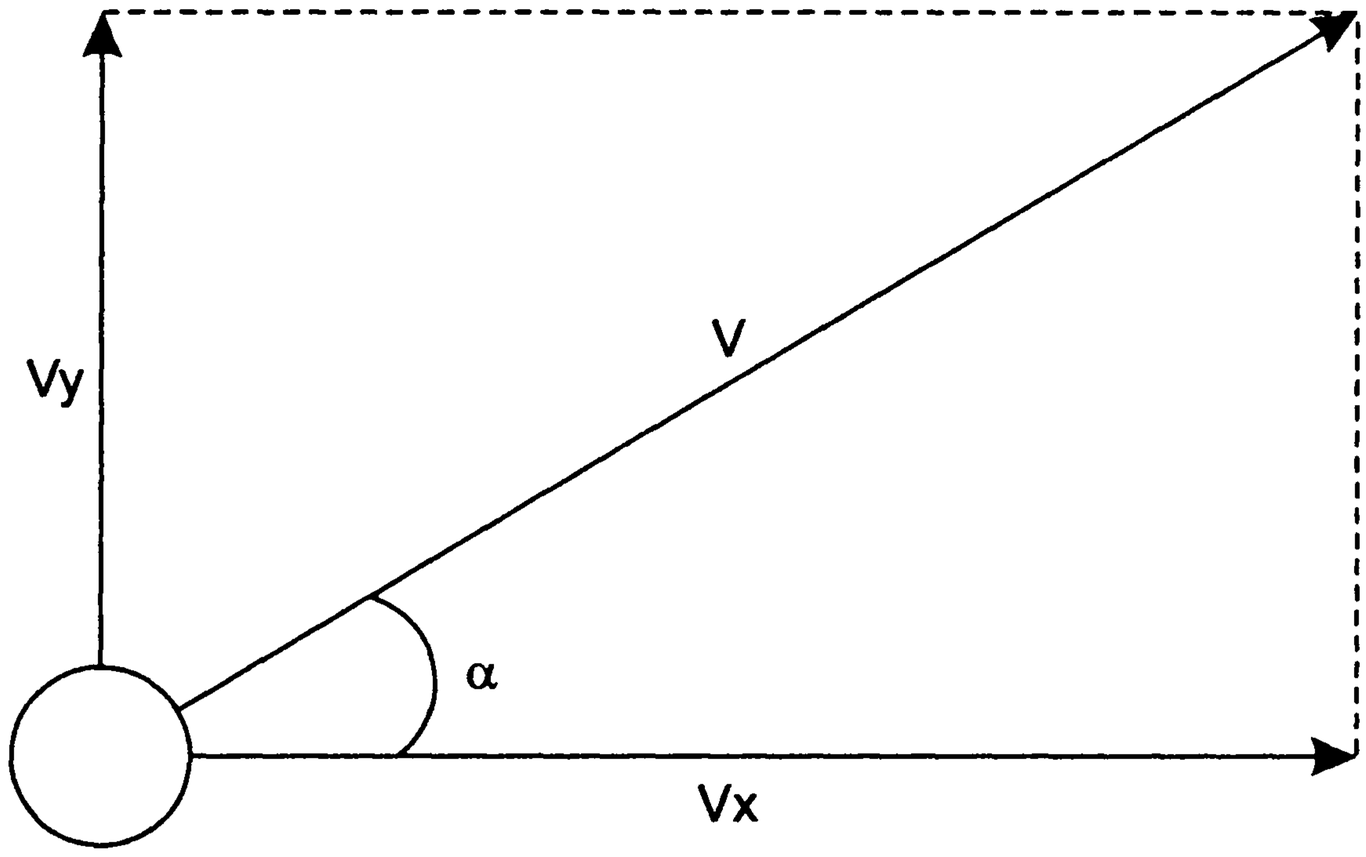


Рис. 1.2. Разложение скорости молекулы на составляющие

Во-вторых, не совсем ясно, как стирать молекулу с экрана и как ее рисовать. Во второй части книги мы займемся «настоящей» спрайтовой анимацией, а пока что ограничимся самым простым вариантом: нарисовать молекулу — значит вывести на экран окружность какого-нибудь цвета. Чтобы стереть молекулу, мы просто нарисуем ту же самую окружность на том же самом месте, но фоновым цветом.

В-третьих, что значит «вернуть молекулу к ближайшей допустимой точке»? Предположим, летит молекула влево. Предположим также, что левая стенка аквариума расположена у самого края экрана, то есть ее X-координата равна нулю, а радиус нашей молекулы равен R. Тогда X-координата молекулы никак не может быть меньше нуля плюс радиус молекулы; тем не менее в процессе работы программы такое может случиться. В этом случае нам ничего не остается делать, кроме как выполнить присваивание

```
X := R
```

и поменять направление движения молекулы на противоположное. Аналогичную процедуру придется повторить и для других случаев «вылета» молекулы — за правую, верхнюю и нижнюю стенки аквариума.

Займемся теперь практическим воплощением нашего проекта.

1. Создайте новый проект.
2. Разместите на главной форме приложения элемент типа TPaintBox — он будет служить экраном для рисования. Измените его название со значения по умолчанию PaintBox1 на более звучное Screen.

---

[1] Скорость, как известно, обычно измеряют в километрах в час или в метрах в секунду. В нашем случае это «V пикселов за цикл».

3. Добавьте на форму кнопку, нажатием которой вся система будет запускаться. В качестве ее названия введите StartStopBtn, а в качестве надписи на ней – Пуск.
4. Двойным щелчком на кнопке StartStopBtn перейдите к редактированию обработчика TForm1.StartStopBtnClick(). Отредактируйте ее в соответствии с листингом 1.1.

Листинг 1.1. Молекула газа в закрытом сосуде (первая версия)[1]

```
procedure TForm1.StartStopBtnClick(Sender: TObject),
    var X, Y   : Integer; { координаты молекулы }
        Vx, Vy : Integer; { составляющие скорости молекулы }
        angle  : Real;    { угол, задающий начальное направление полета }

    const R : Integer = 10; { радиус молекулы }
          V : Integer = 7;  { скорость молекулы }
begin
    Randomize;
    Screen.Refresh;                                  { очистка экрана }
    X := RandomRange(R, Screen.Width - R);           { выбор начального (1)[2] }
    Y := RandomRange(R, Screen.Height - R);          { положения молекулы }
    angle := Random(360)* Pi / 180;                  { и ее направления (2) }

    Vx := Round(V * Sin(angle));                     { получение составляющих }
    Vy := Round(V * Cos(angle));                     { скорости молекулы }

    while true do                                    { основной цикл }
    begin
        Screen.Canvas.Pen.Color := clBtnFace;     { стираем молекулу (3) }
        Screen.Canvas.Ellipse(X - R, Y - R, X + R, Y + R);

        X := X + Vx;                                { сдвигаем на новую позицию }
        Y := Y + Vy;
                                                    { определяем, не вышла ли }
        if X > Screen.Width - R then                { молекула за границы аквариума }
            begin  X := Screen.Width - R; Vx := -Vx; end;   { правая граница }
        if X < R then
            begin  X := R; Vx := -Vx; end;                  { левая граница }
        if Y > Screen.Height - R then
            begin  Y := Screen.Height - R; Vy := -Vy; end;  { нижняя граница }
        if Y < R then
            begin  Y := R; Vy := -Vy; end;                  { верхняя граница }

        Screen.Canvas.Pen.Color := clBlue;              { рисуем молекулу на }
        Screen.Canvas.Ellipse(X - R, Y - R, X + R, Y + R);   { новой позиции }

        Sleep(10);                                     { пауза 10 миллисекунд }
    end;
end;
```

[1] Исходные тексты всех программ книги можно найти на сайте издательства «Питер» www.piter.com.

[2] Цифрами в листинге отмечены строки, поясняемые ниже.

Пояснения к листингу:

1. Функция RandomRange(Left, Right) возвращает случайное целое число в промежутке от Left до Right включительно. Чтобы получить доступ к этой функции, необходимо добавить модуль Math к списку модулей директивы uses.
2. Напомню, что функции Sin() и Cos() требуют передавать им аргументы в радианах; мы же генерируем случайное число в промежутке от нуля до 360 градусов. Чтобы перевести его в радианы, пользуемся формулой

   ```
   радианы = градусы * Pi / 180
   ```

3. По умолчанию цвет фона формы — clBtnFace.

Уже сейчас программу можно запустить и убедиться, что молекула исправно летает по экрану. Единственная загвоздка заключается в том, что из приложения можно выйти, только аварийно завершив ее выполнение (команда Run ▸ Program Reset в среде Delphi).

Можно завести на форме вторую кнопку, которая будет завершать цикл, но я предпочту в соответствии с принципом Оккама[1] возложить вторую функцию на уже созданную кнопку (не зря же мы ее назвали StartStopBtn). Для этого заведем булеву переменную-индикатор IsRunning, которая отражает текущее состояние программы (запущена/остановлена):

```
IsRunning : Boolean = false;   { изначально молекула не двигается }
```

Переменную надо описать в глобальной области (например, сразу после строки Form1: TForm1;). Теперь алгоритм работы процедуры TForm1.StartStopBtnClick() будет выглядеть так:

```
ЕСЛИ IsRunning
    IsRunning := false;
    StartStopBtn.Caption := 'Пуск';
    выход из процедуры
IsRunning := true;
StartStopBtn.Caption := 'Стоп';
двигать молекулу в цикле пока IsRunning = true
```

Когда пользователь нажимает кнопку Пуск первый раз, значение переменной IsRunning равно false, поэтому программа начнет выполнять цикл движения молекулы. Если же он нажмет кнопку еще раз, то переменная IsRunning станет равной false, и цикл закончится.

Есть еще одна тонкость. Дело в том, что Windows не будет прерывать выполнение цикла, чтобы обработать нажатие кнопки. Иными словами, программа «зависнет» в цикле, и кнопка StartStopBtn просто не будет нажиматься. Чтобы такого не происходило, необходимо где-нибудь в цикле принудительно вызвать обработку пришедших событий (таким событием, в частности, является щелчок на кнопке). Для этого вставьте вызов процедуры Application.ProcessMessages, например, после строки Sleep(10);.

Окончательная версия процедуры TForm1.StartStopBtnClick() приведена в листинге 1.2, внешний вид программы — на рис. 1.3.

---

[1] Одна из формулировок этого принципа звучит как «не умножайте без достаточного на то основания число необходимых сущностей».

Листинг 1.2. «Молекула газа в закрытом сосуде» (вторая версия)

```
procedure TForm1.StartStopBtnClick(Sender: TObject);
    var X, Y   : Integer; { координаты молекулы }
        Vx, Vy : Integer; { составляющие скорости молекулы }
        angle  : Real;    { угол, задающий начальное направление полета }

    const R : Integer = 10; { радиус молекулы }
          V : Integer = 7;  { скорость молекулы }
begin
    if IsRunning then
begin
    IsRunning := false;
    StartStopBtn.Caption := 'Пуск';
    Exit;
end;

StartStopBtn.Caption := 'Стоп';
IsRunning := true;

Randomize;
Screen.Refresh;                              { очистка экрана }
X := RandomRange(R, Screen.Width - R);       { выбор начального }
Y := RandomRange(R, Screen.Height - R);      { положения молекулы }
angle := Random(360)* Pi /180;               { и ее направления }

Vx := Round(V * Sin(angle));                 { получение составляющих }
Vy := Round(V * Cos(angle));                 { скорости молекулы }

while isRunning do                                { основной цикл }
begin
    Screen.Canvas.Pen.Color := clBtnFace;     { стираем молекулу }
    Screen.Canvas.Ellipse(X - R, Y - R, X + R, Y + R);

    if IsRunning = false then Break;     { конец цикла }

    X := X + Vx;                         { сдвигаем на новую позицию }
    Y := Y + Vy;
                                         { определяем, не вышла ли }
    if X > Screen.Width - R then         { молекула за границы аквариума }
        begin  X := Screen.Width - R; Vx := -Vx; end;   { правая граница }
    if X < R then
        begin  X := R; Vx := -Vx; end;                  { левая граница }
    if Y > Screen.Height - R then
        begin  Y := Screen.Height - R; Vy := -Vy; end;  { нижняя граница }
    if Y < R then
        begin  Y := R; Vy := -Vy; end;                 { верхняя граница }

    Screen.Canvas.Pen.Color := clBlue;                 { рисуем молекулу на }
    Screen.Canvas.Ellipse(X - R, Y - R, X + R, Y + R);   { новой позиции }

    Sleep(10);                                      { пауза 10 миллисекунд }
    Application.ProcessMessages;      { принудительная обработка событий }
  end;
end;
```

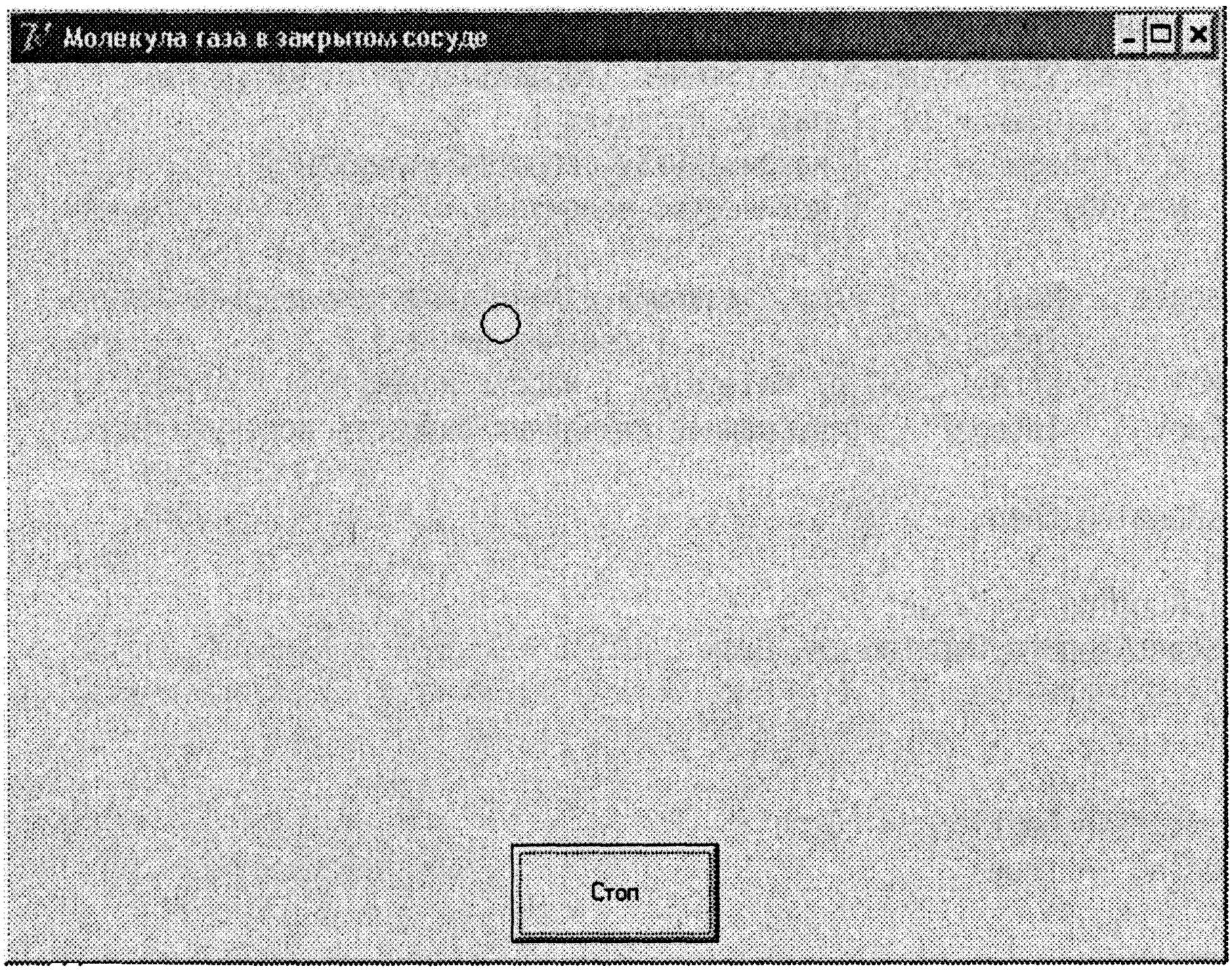


Рис. 1.3. Готовое приложение в работе

Модель 2. «Идеальный газ»

В физике идеальным называется газ, в котором силы взаимодействия между частицами пренебрежимо малы. Иными словами, молекулы газа не сталкиваются и никак не влияют на поведение друг друга. Конечно, идеальных газов не бывает, но если реально существующий газ достаточно разрежен и его температура далека от температуры конденсации, то такой газ можно считать близким к идеальному.

Имея готовую модель одной молекулы, сделать газ уже не так трудно. Для этого надо доработать программу таким образом, чтобы она работала не с одной молекулой, а с массивом из `N` молекул. Кроме того, поскольку в газе обычно встречаются как «медленные», так и «быстрые» молекулы, скорость каждой отдельной молекулы будет выбираться случайным образом.

Во-первых, раз уж мы будем работать с массивом *молекул*, не помешает описать новый тип данных «молекула»:

```
type
    Molecule = record
        X, Y   : Integer;
        Vx, Vy : Integer;
    end;
```

Кроме того, определим, сколько молекул содержит аквариум:

```
const N = 30;
```

Дальнейшие изменения программы совершенно прямолинейны. Полный текст процедуры `TForm1.StartStopBtnClick()` приведен в листинге 1.3, а внешний вид готового приложения — на рис. 1.4.

Листинг 1.3. «Идеальный газ»

```
procedure TForm1.StartStopBtnClick(Sender: TObject);
    const R : Integer = 10; { радиус молекулы }
          V : Integer = 7;  { максимальная скорость молекулы }
          N = 30;           { количество молекул }

    var angle  : Real;    { угол, задающий изначальное направление полета }
        i      : Integer;              { счетчик цикла }
        Mol    : array[1..N] of Molecule; { массив молекул }
        CurV   : Integer;   { выбранная случайная скорость молекулы }
begin
    if IsRunning then
    begin
        IsRunning := false;
        StartStopBtn.Caption := 'Пуск';
        Exit;
    end;

    StartStopBtn.Caption := 'Стоп';
    IsRunning := true;

    Randomize;
    Screen.Refresh;

    for i := 1 to N do                          { цикл по всем молекулам }
    begin
    Mol[i].X := RandomRange(R, Screen.Width - R);   { выбор начального }
    Mol[i].Y := RandomRange(R, Screen.Height - R);  { положения молекулы }
    angle := Random(360)* Pi /180;                  { и ее направления }

    CurV := RandomRange(1, V);              { выбор скорости молекулы (1-V) }
    Mol[i].Vx := Round(CurV * Sin(angle));  { получение составляющих }
    Mol[i].Vy := Round(CurV * Cos(angle));  { скорости молекулы }
end;

while isRunning do                                  { основной цикл }
begin
    for i := 1 to N do                      { цикл по всем молекулам }
    begin
        Screen.Canvas.Pen.Color := clBtnFace;    { стираем молекулу }
        Screen.Canvas.Ellipse(Mol[i].X - R, Mol[i].Y - R,
                              Mol[i].X + R, Mol[i].Y + R);

Mol[i].X := Mol[i].X + Mol[i].Vx;   { сдвигаем на новую позицию }
Mol[i].Y := Mol[i].Y + Mol[i].Vy;

{ определяем, не вышла ли молекула за границы аквариума }
if Mol[i].X > Screen.Width - R then
begin
    Mol[i].X := Screen.Width - R;
    Mol[i].Vx := -Mol[i].Vx;
end;
if Mol[i].X < R then
begin
```

```
            Mol[i].X := R;
            Mol[i].Vx := -Mol[i].Vx;
        end;
        if Mol[i].Y > Screen.Height - R then
        begin
            Mol[i].Y := Screen.Height - R;
            Mol[i].Vy := -Mol[i].Vy;
        end;
        if Mol[i].Y < R then
        begin
            Mol[i].Y := R;
            Mol[i].Vy := -Mol[i].Vy;
        end;

        Screen.Canvas.Pen.Color := clBlue;    { рисуем молекулу на новой }
        Screen.Canvas.Ellipse(Mol[i].X - R, Mol[i].Y - R, { позиции }
                              Mol[i].X + R, Mol[i].Y + R);
      end;
      Sleep(10);                                    { пауза 10 миллисекунд }
      Application.ProcessMessages;
    end;
end.
```

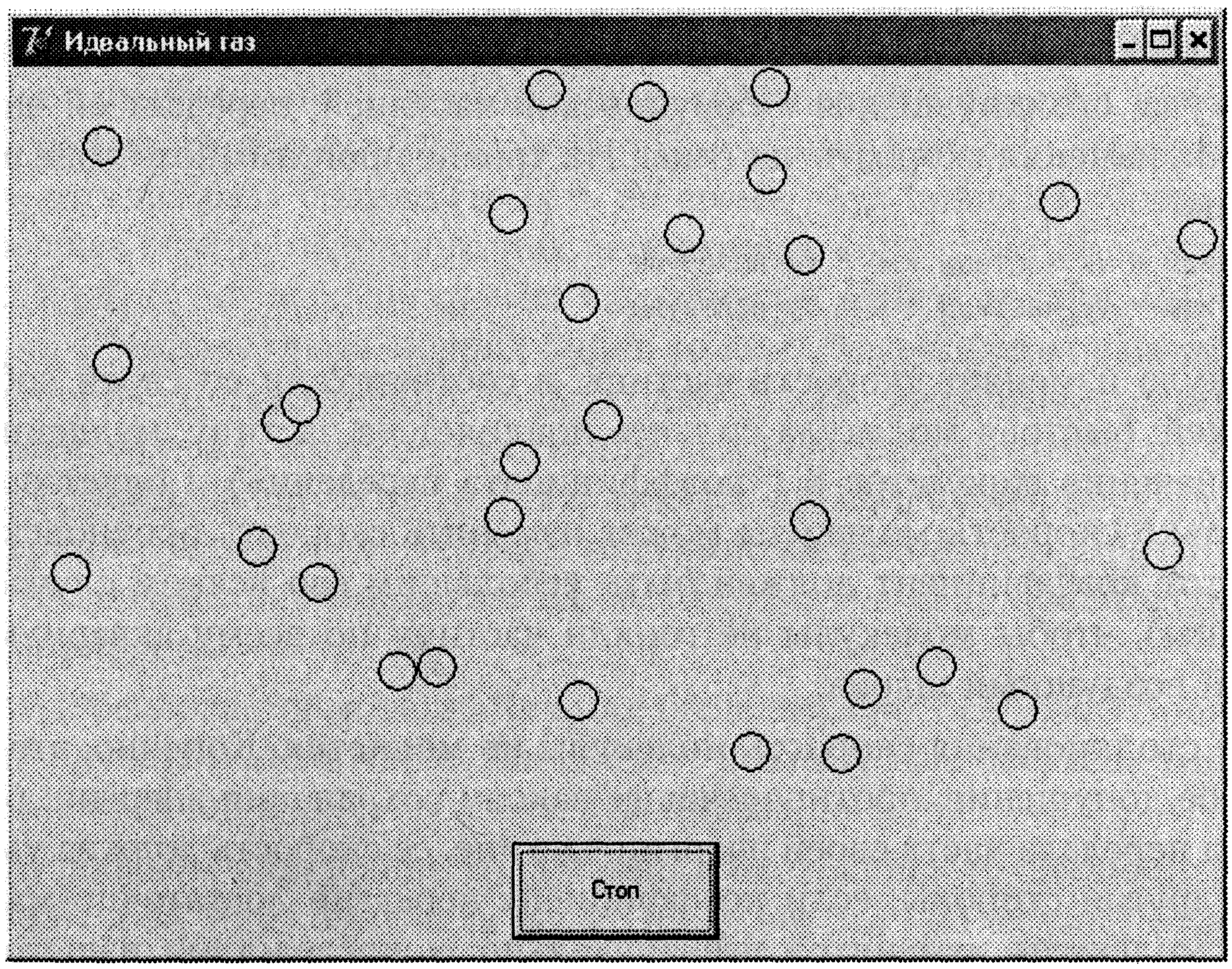


Рис. 1.4. Модель «Идеальный газ»

Я выбрал довольно большой радиус молекулы (10 пикселов) лишь для наглядности; на самом деле газ, показанный на рис. 1.4, отнюдь не идеальный – молекулы то и дело сталкиваются друг с другом. Чтобы повысить точность модели, можно уменьшить радиус (вплоть до единицы).

Модель 3. «Броуновское движение»

«Броуновское движение» — пожалуй, первая по-настоящему интересная модель, к тому же имеющая практическую пользу. Как опытным путем изучить поведение молекул жидкости или газа? В обычный микроскоп их не рассмотришь — слишком маленькие они, да и движутся молекулы слишком быстро (сотни метров в секунду). Гениальная и в то же время простая идея того, как это сделать, принадлежит английскому ботанику Роберту Броуну[1]. Если поместить в изучаемую субстанцию какую-нибудь маленькую частицу (но в то же время достаточно большую, чтобы ее можно было увидеть в микроскоп), то она будет двигаться, испытывая постоянные толчки со стороны молекул. Наблюдая за движением частицы, можно получить примерное представление о поведении молекул субстанции и даже с хорошей точностью вычислить среднее значение скорости их движения.

Чтобы получить модель «Броуновское движение», нам понадобится добавить в модель идеального газа код, отвечающий за броуновскую частицу. Броуновская частица ведет себя точно так же, как и обычная молекула — движется равномерно и прямолинейно, отражается от стенок аквариума. Но если какая-то молекула столкнулась с ней, частица должна отреагировать на это.

Просто так из «Идеального газа» «Броуновское движение» не получить. Придется сделать несколько новых допущений (к сожалению, еще сильнее отдаляющих модель от реальности).

1. Молекула, столкнувшаяся с броуновской частицей, передает ей часть своей энергии, изменяя скорость частицы. На уровне программного кода это выглядит так:

```
Vx_частицы = Vx_частицы + K * Vx_молекулы
Vy_частицы = Vy_частицы + K * Vy_молекулы
```

2. Молекула изменяет скорость частицы в соответствии со своей собственной энергией: быстрая молекула сильнее повлияет на частицу, медленная — слабее. Значение параметра K («коэффициента передачи») и будет определять, насколько скорость молекулы может изменить скорость броуновской частицы. Этот коэффициент должен быть небольшим (во всяком случае, меньше единицы), чтобы молекулы не гоняли частицу по всему аквариуму словно легкий воздушный шарик.
3. После столкновения скорость молекулы **не меняется**. Конечно, это довольно сильное допущение, противоречащее закону сохранения энергии: раз уж молекула передала часть своей энергии частице, ее скорость должна уменьшиться. С другой стороны, если коэффициент передачи достаточно мал, то скорость молекулы изменится незначительно, и последствия нашего решения скажутся не скоро.
4. Самое сильное допущение: после столкновения с частицей молекула **не меняет** направления своего движения. Надеюсь, это предположение не переполнит чашу вашего терпения: действительно, уж такое поведение в корне расходит-

[1] Как и большинство других гениальных и в то же время простых идей, она была найдена случайно — во время опытов со спорами растения плауна.

ся с реальностью, и какими бы то ни было «приближениями» оправдать его трудно. Тем не менее постараюсь это сделать. Если аквариум большой, а молекул в нем много, вероятность того, что броуновская частица скоро встретится с той же самой молекулой, с которой только что столкнулась, очень мала. Иначе говоря, какая разница, в какую сторону полетит молекула, если она в обозримом будущем не встретится с броуновской частицей? Конечно, разница есть, но она не так велика, как может показаться на первый взгляд. Второй аргумент более прост и весом: чтобы реализовать отражение молекул от частицы, надо очень хорошо потрудиться. В момент столкновения придется построить касательную к броуновской частице в точке касания, потом выяснить угол падения молекулы, потом определить новое направление движения молекулы... Все это не так просто, да и требует довольно громоздких математических выкладок[1]. Нас же больше интересуют вопросы программирования, а не формулы физики и математики, поэтому и не стоит уделять им слишком много внимания в этой модели.

Теперь, когда общая картина происходящего ясна, осталось решить, а как же мы, собственно, будем определять столкновение броуновской частицы с молекулой? Самый простой путь (которым мы и воспользуемся) состоит в том, чтобы вычислить расстояние между центром броуновской частицы и центром молекулы. Если это расстояние окажется меньше радиуса частицы плюс радиус молекулы, столкновение налицо[2]. Проделав эту процедуру для всех молекул из аквариума, мы определим все произошедшие столкновения. Обращу ваше внимание на еще один неочевидный момент. С учетом наших допущений, молекула, встретившись с броуновской частицей, как бы пролетает сквозь нее. А отсюда вывод: может так случиться, что два раза подряд (то есть во время двух подряд идущих итераций моделирования внутри основного цикла программы) расстояние между молекулой и частицей будет меньше суммы их радиусов. Иначе говоря, молекула два (а то и три, и четыре) раза подряд будет толкать частицу, хотя на самом деле столкновение было, естественно, только одно. Чтобы избавиться от этой пренеприятнейшей «фичи», можно завести для каждой молекулы отдельную булеву переменную, указывающую на то, сталкивалась ли молекула с броуновской частицей во время предыдущей итерации моделирования или нет. Если ответом будет «да», то воздействие молекулы на частицу во время текущей итерации попросту игнорируется.

Приступим к реализации модели. Как уже было сказано, основой «Броуновского движения» служит модель «Идеальный газ», поэтому нам осталось только вставить несколько строк кода в соответствующие места.

Для начала добавим в описание типа данных «молекула» ту самую булеву переменную, о которой только что шла речь. Новая версия «молекулы» будет выглядеть так:

---

[1] Естественно, это лишь мое личное мнение. Может, кто-то легко проделает все эти выкладки в уме.

[2] На самом деле, такая же ситуация произойдет, если *частица* толкнет молекулу, а не наоборот. Разумеется, этого мы не учитываем. (Еще одно допущение! Что ж, тем больше простора для фантазии тех, кто захочет улучшить модель...). С другой стороны, есть надежда, что большая, инертная частица вряд ли часто будет «догонять» ту или иную молекулу.

```
type
    Molecule = record
        X, Y          : Integer;
        Vx, Vy        : Integer;
        WasCollision : Boolean; { указывает на столкновение с броуновской }
    end;                        { частицей во время предыдущей итерации }
```

Теперь к описанию констант необходимо добавить радиус броуновской частицы[1] и «коэффициент передачи»:

```
Rb : Integer = 60; { радиус броуновской частицы }
K  : Real = 0.01;  { "коэффициент передачи" }
```

Аналогично, потребуются переменные, описывающие текущее состояние частицы:

```
Xb, Yb   : Real;      { координаты броуновской частицы }
Vxb, Vyb : Real;      { составляющие скорости броуновской частицы }
```

С их значениями дело обстоит просто: пусть вначале частица находится в центре аквариума, а скорость ее равна нулю. Где-нибудь в начале программы (например, после вызова процедуры Screen.Refresh) добавьте строки:

```
Xb := Screen.Width div 2;
Yb := Screen.Height div 2;
Vxb := 0;
Vyb := 0;
```

В тело цикла инициализации молекул (то есть там, где мы определяем начальное положение и скорость каждой молекулы) необходимо добавить указание того, что молекула еще ни разу не сталкивалась с броуновской частицей:

```
Mol[i].WasCollision := false;
```

Следующий шаг — создание кода, обрабатывающего столкновения объектов. В принципе, поместить его можно в любом месте цикла, в котором происходит движение молекул. Я сделал это сразу после строк, рисующих молекулу на новой позиции:

```
{ это еще старый код }
Screen.Canvas.Pen.Color := clBlue;                { рисуем молекулу на }
Screen.Canvas.Ellipse(Mol[i].X - R, Mol[i].Y - R,    { новой позиции }
                      Mol[i].X + R, Mol[i].Y + R);
{ а это уже новые строки }
if Sqrt((Mol[i].X - Xb)*(Mol[i].X - Xb) +        { если произошло столкновение }
        (Mol[i].Y - Yb)*(Mol[i].Y - Yb)) < Rb + R then
begin
    if Mol[i].WasCollision = false then   { если на предыдущей итерации }
    begin                                 { эта молекула не сталкивалась с }
        Vxb := Vxb + K * Mol[i].Vx;       { частицей, обрабатываем }
        Vyb := Vyb + K * Mol[i].Vy;       { столкновение }
    end;
    Mol[i].WasCollision := true;      { на данной итерации i-я молекула }
end                                   { столкнулась с броуновской частицей }
```

---

[1] Заодно можно уменьшить радиусы молекул: не забывайте, молекулы гораздо меньше частицы. Я задал значение радиуса молекулы равным двум пикселам.

```
    else
        Mol[i].WasCollision := false;     { столкновения не было }
```

Последнее, что осталось сделать, — это обновить положение броуновской частицы на экране. Делается это сразу после цикла движения молекул:

```
    Screen.Canvas.Pen.Color := clBtnFace;        { стереть частицу с экрана }
    Screen.Canvas.Ellipse(Round(Xb - Rb), Round(Yb - Rb),
                          Round(Xb + Rb), Round(Yb + Rb));

    Xb := Xb + Vxb;                              { сдвинуть на новую позицию }
    Yb := Yb + Vyb;

    if Xb > Screen.Width - Rb then    { обработать отражения от стенок аквариума }
        begin  Xb := Screen.Width - Rb; Vxb := -Vxb; end;
    if Xb < Rb then
        begin  Xb := Rb; Vxb := -Vxb; end;
    if Yb > Screen.Height - Rb then
        begin  Yb := Screen.Height - Rb; Vyb := -Vyb; end;
    if Yb < Rb then
        begin  Yb := Rb; Vyb := -Vyb; end;

    Screen.Canvas.Pen.Color := clRed;            { нарисовать частицу на новом месте }
    Screen.Canvas.Ellipse(Round(Xb - Rb), Round(Yb - Rb),
                          Round(Xb + Rb), Round(Yb + Rb));
```

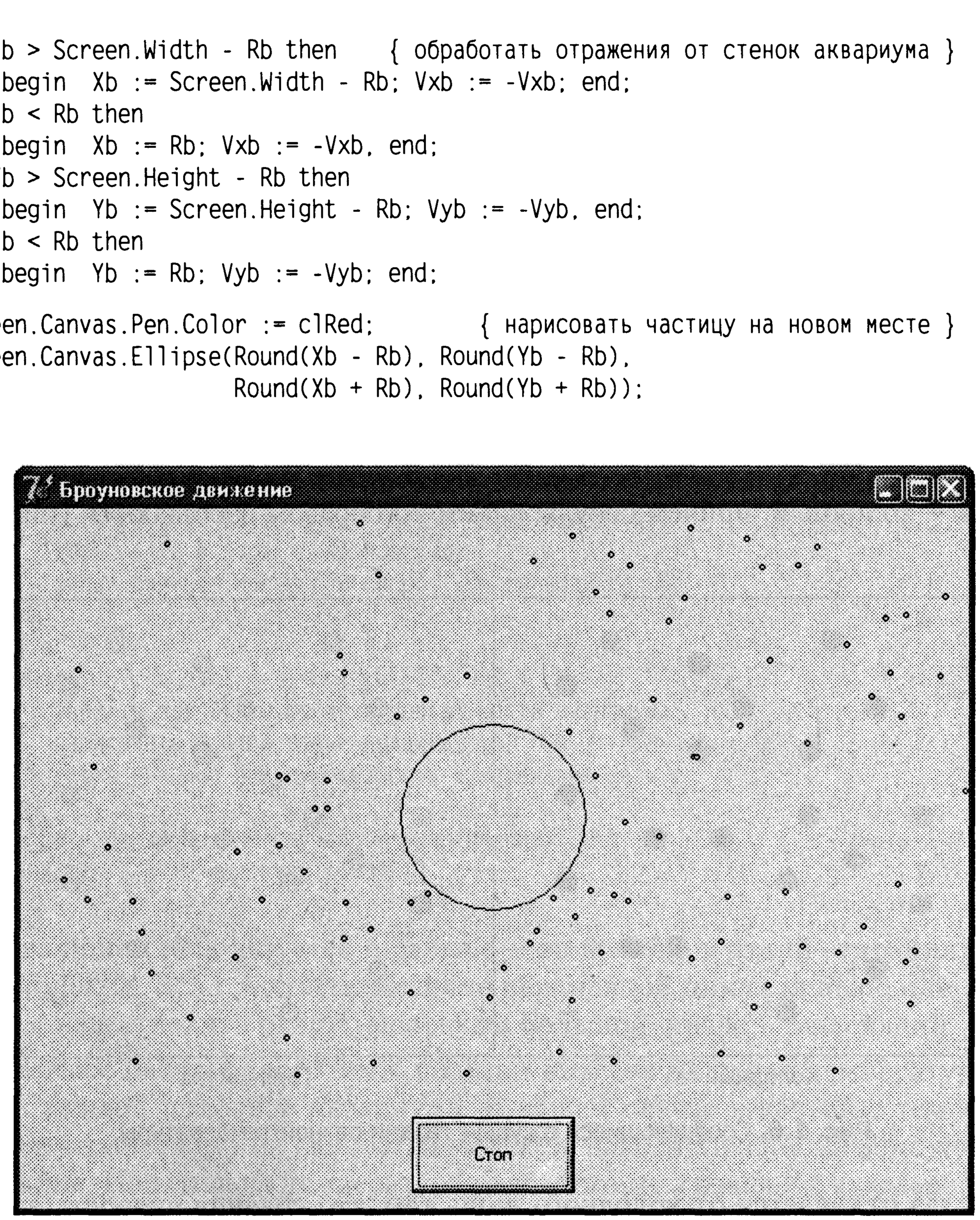


Рис. 1.5. Модель «Броуновское движение»

Полный текст готовой процедуры TForm1.StartStopBtnClick() приводить не буду, а вот результатом работы программы полюбоваться можно (рис. 1.5).

Модель 4. «Равновесие» (второе начало термодинамики)

Второе начало термодинамики в формулировке Рудольфа Клаузиуса (1850 г.) гласит: «Теплота сама по себе не может перейти от более холодного тела к более теплому». Из этого простого закона, проявления которого мы видим каждый день, многие ученые делают весьма неутешительные выводы о будущем нашей Вселенной. Они полагают, что рано или поздно тепло от всех более теплых тел перейдет к более холодным, звезды погаснут и прекратятся вообще все процессы в мире. Во Вселенной наступит динамическое равновесие — так называемая «тепловая смерть». Разумеется, это лишь гипотеза: другие ученые пытаются ее опровергать, убеждая в том, что для достижения тепловой смерти потребуется бесконечно большой отрезок времени (иными словами, Вселенная будет умирать бесконечно долго, поэтому так и не умрет) или что условия, в которых существует Вселенная, постоянно меняются, следовательно, равновесие никогда не будет достигнуто — будет лишь вечное изменение и развитие.

Не берусь судить о перспективах всего мира; поживем — увидим (впрочем, в обозримом будущем Армагеддонов не ожидается[1]), но на простейшем примере продемонстрировать справедливость второго начала термодинамики можно.

Предположим, что у нас есть два сосуда. В одном из них находится горячий газ, в другом — холодный, причем сосуды каким-то образом сообщаются (рис. 1.6).

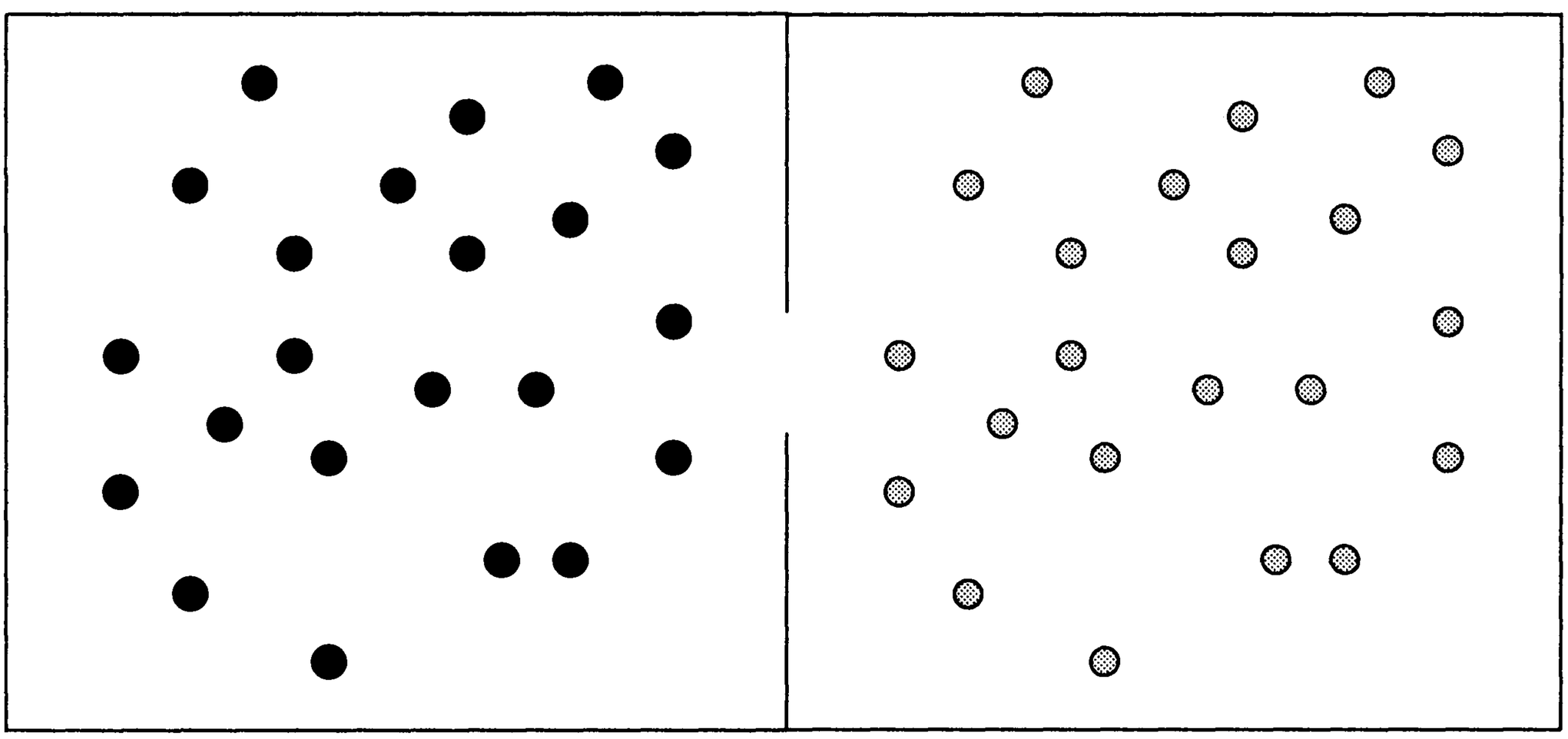


Рис. 1.6. Сообщающиеся сосуды с газами разной температуры

Горячий газ отличается от холодного лишь тем, что средняя скорость его молекул выше (поэтому, если сразу думать в терминах наших моделей, дополнительных проблем не возникает — скорость любой молекулы можно контролировать без ограничений).

[1] Дай Бог! — *Примеч. ред.*

Поскольку сосуды сообщаются, молекулы горячего газа будут проникать в левый сосуд, а молекулы холодного газа – в правый. Согласно второму началу термодинамики, система рано или поздно должна прийти к состоянию динамического равновесия, в котором температура газа в правом сосуде все время примерно равна температуре газа в левом, то есть газы перемешаются так хорошо, что уже никто не сможет сказать точно, где изначально был горячий, а где холодный газ.

Проще всего реализовать эту модель на основе предыдущей («Идеальный газ»). Я не буду приводить полный текст программы (которая получается достаточно простой по структуре, но объемной [1]), вместо этого обращу ваше внимание на несколько важных моментов.

1. В отличие от «Идеального газа», здесь нам приходится иметь дело с двумя видами молекул: холодными и горячими, причем первые изначально находятся в левой половине аквариума, а вторые – в правой. Помимо разных скоростей, неплохо бы задать для горячих и холодных молекул разные радиусы, чтобы их можно было легко различать. Самый простой и ненакладный способ изменить идеологию программы заключается в том, чтобы считать первую половину молекул в массиве Mol холодными, а вторую – горячими. Тогда цикл инициализации молекул на псевдокоде будет выглядеть так:

```
for i := 1 to 2 * N do  { будем считать, что у нас всего 2 * N молекул }
begin
    if i <= N then
    begin
        выбрать случайную точку в левой части аквариума
        нарисовать в этой точке молекулу,
        задав для нее маленькую скорость и большой радиус
    end
    else
    begin
        выбрать случайную точку в правой части аквариума
        нарисовать в этой точке молекулу,
        задав для нее большую скорость и маленький радиус
    end;
    выбрать для молекулы случайное направление
    записать данные молекулы в Mol[i]
end;
```

2. Тот же прием используется и дальше – во время операций стирания/рисования молекул. Если текущее значение переменной цикла меньше или равно N, то нам известно, что радиус молекулы большой, иначе – маленький.
3. В процессе работы программы полезно вывести на экран температуру молекул в каждой половине аквариума. В нашем случае в качестве температуры вполне сгодится сумма скоростей молекул, находящихся в интересующей нас половине. Изначально температура холодного газа равна N * V_холодной_молекулы, а горячего – N * V_горячей_молекулы (что логично). Если какая-нибудь молекула перелетает в другую половину аквариума, то температура первой половины уменьшается на скорость молекулы, температура же второй увеличивается на то же число.

---

[1] Кроме того, листинги всех программ лежат на сайте издательства «Питер» www.piter.com.

4. Вывести значения температур проще всего в двух элементах TLabel. Напомню, что для перевода числа в строку используется функция IntToStr(число).
5. Самое важное нововведение, которое встречается в этой модели, — наличие неполной перегородки между левой и правой половинами аквариума. Когда молекула подлетает к перегородке, необходимо как-то определить, что с ней произойдет дальше: перелетит ли она в другую половину аквариума или отразится от перегородки назад. Напомню, в процессе работы с молекулой мы сначала стираем ее, потом передвигаем и, наконец, рисуем на новом месте. Допустим, молекула уже стерта и передвинута. Теперь, прежде чем рисовать ее, надо убедиться, что столкновения с перегородкой не произошло. Если же столкновение произошло, то его надо обработать:

```
IF молекула находится в правой части аквариума, но при этом на предыдущей
   итерации моделирования молекула была в его левой части
    IF молекула находится на уровне отверстия в перегородке
        молекула перелетает из левой части аквариума в правую
        T1 := T1 - Vмолекулы; T2 := T2 + Vмолекулы
    ELSE
        отразить молекулу от перегородки (назад в левую часть)
ELSE IF молекула находится в левой части аквариума, но при этом на предыдущей
        итерации моделирования молекула была в его правой части
    IF молекула находится на уровне отверстия в перегородке
        молекула перелетает из правой части аквариума в левую
        T1 := T1 + Vмолекулы; T2 := T2 - Vмолекулы
    ELSE
        отразить молекулу от перегородки (назад в правую часть)
```

6. На уровне реальной программы этот фрагмент выглядит ненамного сложнее:

```
if (Mol[i].X > Screen.Width div 2 - RMolecule) and
   (Mol[i].X - Mol[i].Vx <= Screen.Width div 2 - RMolecule) then
begin
    if (Mol[i].Y >= Screen Height div 2 - RHole) and { RHole - радиус отвер- }
       (Mol[i].Y <= Screen.Height div 2 + RHole) then   { стия в перегородке }
        begin T1 := T1 - V; T2 := T2 + V  end          { изменяем температуру }
    else
    begin
        Mol[i].X := Screen Width div 2 - RMolecule;    { отражаем молекулу от }
        Mol[i].Vx := -Mol[i].Vx;                                { перегородки }
    end,
end
else if (Mol[i].X < Screen.Width div 2 + RMolecule) and          { аналогично }
        (Mol[i].X - Mol[i].Vx >= Screen Width div 2 + RMolecule) then
begin
    if (Mol[i].Y >= Screen.Height div 2 - RHole) and
       (Mol[i].Y <= Screen.Height div 2 + RHole) then
        begin N2 := N2 - V; N1 := N1 + V; end
    else
    begin
        Mol[i].X := Screen.Width div 2 + RMolecule;
        Mol[i].Vx := -Mol[i].Vx;
    end,
end;
```

Поскольку на каждой итерации к текущим координатам молекулы прибавляются составляющие ее скорости, то узнать положение молекулы на предыдущей итерации несложно: для этого надо лишь вычесть значения составляющих скорости из новых координат молекулы. Именно на этом принципе работает первая конструкция if() приведенного фрагмента.

Внешний вид работающей программы приведен на рис. 1.7.

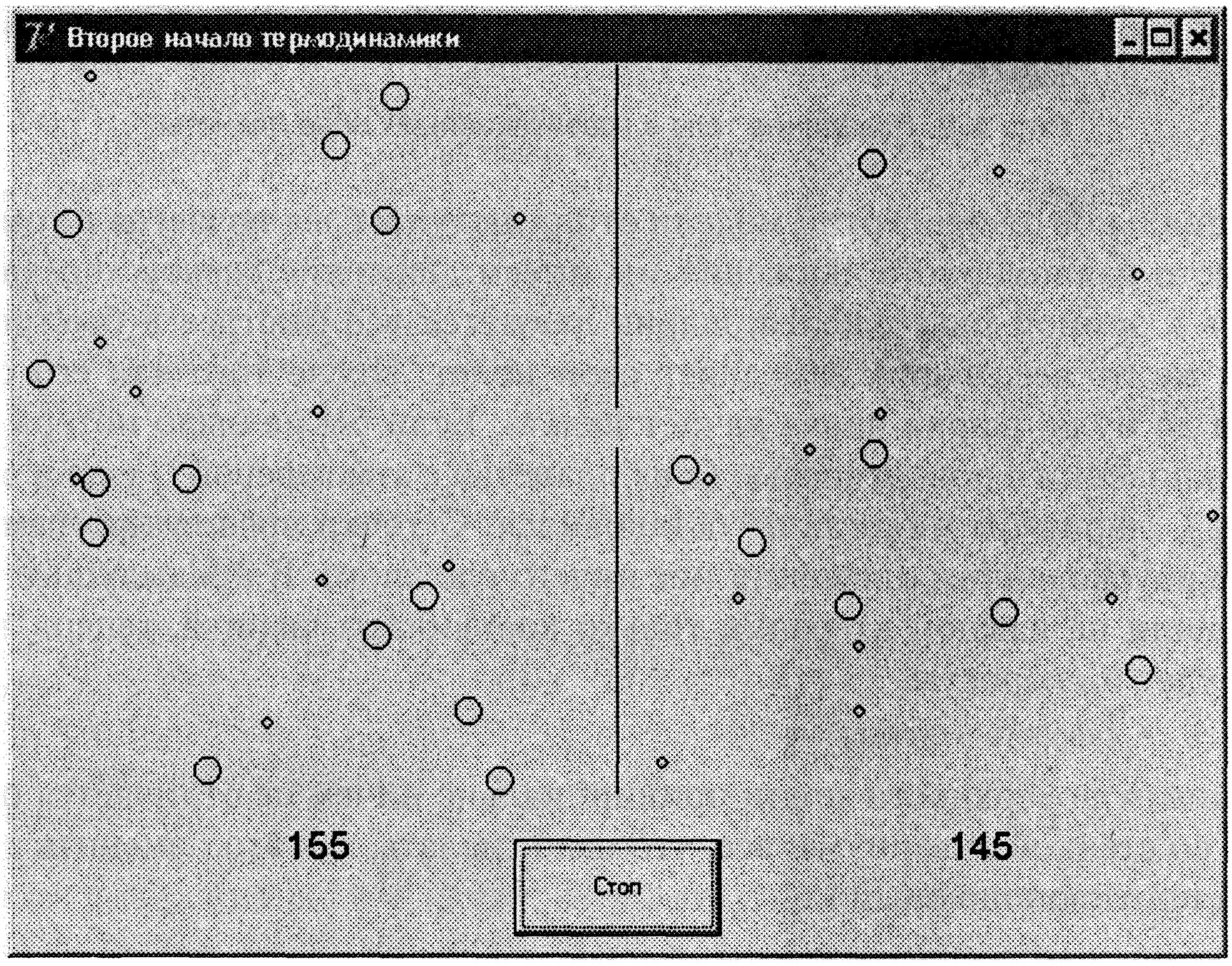


Рис. 1.7. Модель «Равновесие» в действии

Модель 5. «Падающий шар»

Эта простая и эффектная модель переводит наше внимание с микромира на мир осязаемых вещей. Представьте себе круглое тело (металлический шарик, футбольный мяч и т. п. в плоской проекции), которое тихо покоится на возвышении. Что будет, если придать ему скорость в горизонтальном направлении (проще говоря, дать пинка)? Испытывая притяжение Земли (или другой планеты, среду которой мы моделируем), тело достигнет поверхности, затем отразится от нее, потеряв при этом часть своей энергии, снова взлетит на определенную высоту (конечно, начальной высоты оно уже не достигнет) и будет прыгать дальше, пока не остановится на поверхности планеты. Думаю, особых объяснений тут не требуется: в конце концов, все видели, как скачет по земле футбольный мяч (да и рис. 1.8 пояснит положение вещей).

Как вы уже, наверное, догадались, сейчас мы займемся моделированием подобной ситуации.

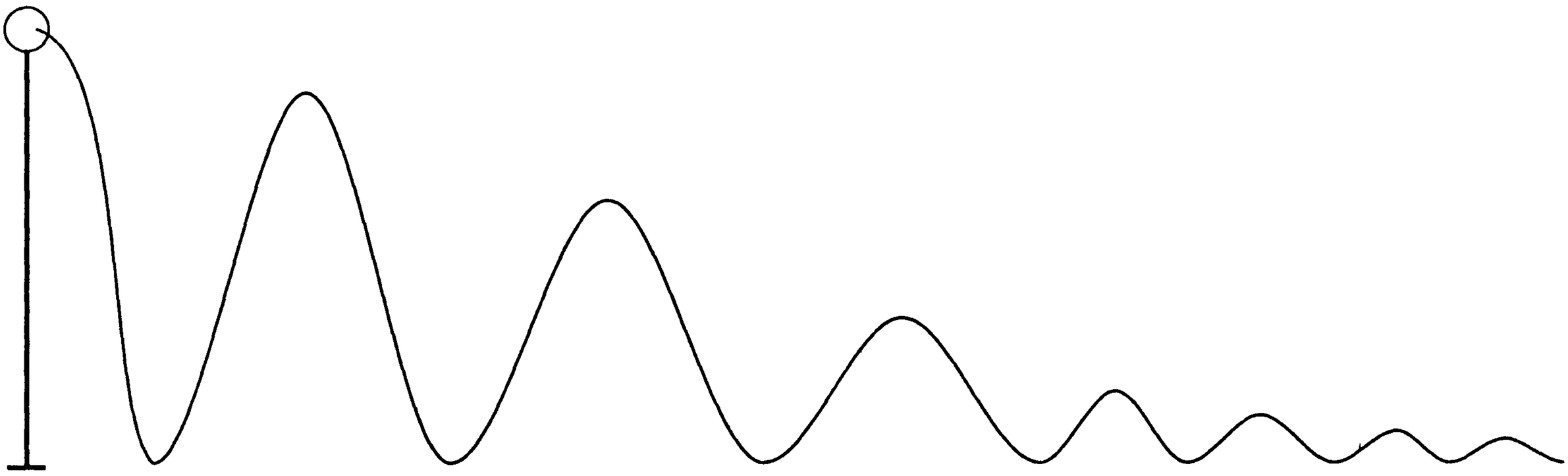


Рис. 1.8. Поведение тела в гравитационном поле планеты

Принципиальное отличие этой модели от предыдущих заключается в том, что теперь вместо равномерного движения мы имеем движение *равноускоренное*: приближаясь к земной поверхности, шарик движется все быстрее и быстрее, отразившись же от нее, шарик замедляет свой полет. После того как его скорость снизится до нуля, шарик снова начнет падать. Если бы не было потерь энергии при столкновении с поверхностью (а на планетах с атмосферой – еще и ее сопротивления), шарик бы никогда не остановился. С точки зрения программирования в равноускоренном движении нет ничего сложного. В предыдущих моделях мы изменяли на каждой итерации только координаты шарика, теперь же изменяться будет и скорость:

```
скорость := скорость + ускорение
```

Отражение от поверхности земли принципиально ничем не отличается от столкновения со стенкой аквариума: все, что надо сделать, так это изменить знак скорости на противоположный.

Как и прежде, нам придется сделать несколько упрощающих предположений (от этого, к сожалению, никуда не деться).

1. Во-первых, будем считать, что в горизонтальном направлении шарик движется равномерно. По сравнению с постоянными скачками вертикальной составляющей скорости изменения горизонтальной составляющей действительно малы.
2. Потери энергии шарика при столкновении с поверхностью сводятся к тому, что вертикальная составляющая его скорости умножается на некоторый «коэффициент потери», меньший единицы. Горизонтальная составляющая скорости не меняется. Опять-таки, в реальности уменьшение горизонтальной составляющей скорости мало по сравнению с уменьшением вертикальной (вспомните, как долго мяч катится по земле уже после того, как он перестал скакать).
3. Шарик теряет энергию только при столкновении с поверхностью. Иными словами, сопротивление атмосферы не учитывается.
4. Будем считать, что моделирование закончено, если шарик вылетел за пределы экрана либо значение вертикальной составляющей скорости стало очень маленьким (например, меньше единицы).

Теперь можно перейти к написанию программы. Я воспользовался моделью «Молекула газа в закрытом сосуде» в качестве каркаса. Как и прежде, вся работа выполняется в процедуре TForm1.StartStopBtnClick(). Ее текст приведен в листинге 1.4.

Листинг 1.4. «Падающий шар»

```
procedure TForm1.StartStopBtnClick(Sender: TObject);
    var X, Y   : Integer; { координаты шара }
        Vx, Vy · Real;    { составляющие скорости }

    const R   : Integer = 10; { радиус шара }
          Vx0 : Integer = 3;  { горизонтальная скорость шара }
          a   : Real = 1;     { ускорение }
          K   . Real = 0.8;   { коэффициент потери }
begin
    if IsRunning then
    begin
        IsRunning := false;
        StartStopBtn.Caption := 'Пуск';
        Exit;
    end;

    StartStopBtn.Caption := 'Стоп';
    IsRunning := true;

    Vy := 0;       { вертикальная составляющая скорости изначально равна нулю }
    Vx := Vx0;
    X := R + 5;                                          { начальное }
    Y := Screen.Height div 2;                            { положение шара }
    Screen Refresh;                                      { очистка экрана }
    Screen.Canvas.Pen.Color := clBlack;                  { рисуем "подставку" }
    Screen Canvas.MoveTo(R + 5, Screen.Height div 2 + R); { на которой }
    Screen.Canvas.LineTo(R + 5, Screen.Height);          { лежит шар }
    Screen.Canvas.Pen.Color := clBlue;
    Screen.Canvas.Ellipse(X - R, Y - R, X + R, Y + R);   { рисуем шар }

    Sleep(500);          { пауза, чтобы увидеть начальное положение на экране }

    while IsRunning do                                   { основной цикл }
    begin
        Screen.Canvas.Pen.Color := clBtnFace;     { стираем шар }
        Screen.Canvas.Ellipse(X - R, Y - R, X + R, Y + R);

        X := X + Round(Vx);                           { сдвигаем на новую позицию }
        Y := Y + Round(Vy);

        if X > Screen.Width - R then   { если шар вышел за экран, "нажимаем" }
            Form1.StartStopBtnClick(nil); { кнопку StartStopBtn (1) }
        if Y > Screen.Height - R then    { если шар столкнулся с землей }
        begin
            Y := Screen.Height - R;  { "отражаем" - как в предыдущих моделях }
            Vy := -Vy * K;           { но скорость уменьшается в K раз }
                                     { если скорость стала по модулю (2) }
                                     { меньше единицы, "нажимаем" StartStopBtn }
```

```
        if abs(Vy) < 1 then Form1.StartStopBtnClick(nil);
    end;

    Vy := Vy + a;          { изменяем скорость в соответствии с ускорением }
    Screen.Canvas.Pen.Color := clBlue;                { рисуем шар на }
    Screen.Canvas.Ellipse(X - R, Y - R, X + R, Y + R);   { новой позиции }

    Sleep(30);
    Application.ProcessMessages;
end;
```

Примечания:

1. Думаю, запись Form1.StartStopBtnClick(nil) кристально ясна не для всех, поэтому ее стоит пояснить. Суть достаточно проста: требуется каким-то образом сымитировать нажатие кнопки StartStopBtn (она же — кнопка Пуск/Стоп). Мы знаем, что при ее нажатии вызывается процедура StartStopBtnClick() объекта Form1. Так почему же просто не вызвать ее явно? В качестве аргумента процедура требует «объект-отправитель», но поскольку мы нигде его не используем, можно смело указать nil.
2. Поскольку скорость постоянно меняет знак, следует, естественно, оценивать ее абсолютное значение.

Изменяя значения констант Vx0, K и a, можно получить самые разнообразные траектории движения шарика. К сожалению, скриншот (снимок экрана) программы практически неотличим от «молекулы в закрытом сосуде», поэтому приводить его не буду.

Модель 6. «Солнечная система»

Теперь мы занимаемся не молекулами, не шариками и даже не футбольными мячами. Объекты этой модели — весьма немаленькие небесные тела.

На самом деле, модель правильнее было бы назвать «Солнце, Земля и Луна». Но «Солнечная система», на мой взгляд, гораздо звучнее (да и короче). Если вы еще не разочарованы столь жестоким обманом с моей стороны, приступим к ее анализу.

Итак, Земля вращается вокруг Солнца, а Луна вращается вокруг Земли. Солнце не двигается вообще (если в качестве центра системы отсчета взять Солнце, то это так и есть). Траектории движения планет — круговые (на самом деле они, конечно, эллиптические, но не будем слишком усложнять себе жизнь). Наша задача — смоделировать движение Земли и Луны.

Мы уже занимались прямолинейным движением объектов по экрану. Теперь подумаем, как осуществить движение по окружности. Предположим, шарик[1] вращается вокруг точки (x0, y0) (как Земля вращается вокруг Солнца), при этом

[1] Я уже не поясняю, что мы рисуем на плоском экране, что все объекты (планеты, молекулы и т. п.) — просто «шарики» (точнее, окружности). В этом плане все остается без изменений по сравнению с предыдущими моделями.

расстояние от него до (x0, y0) всегда равно R (радиусу окружности, по которой движется шарик). Тогда текущее положение шарика всегда можно однозначно задать углом его отклонения, например, от оси абсцисс (рис. 1.9).

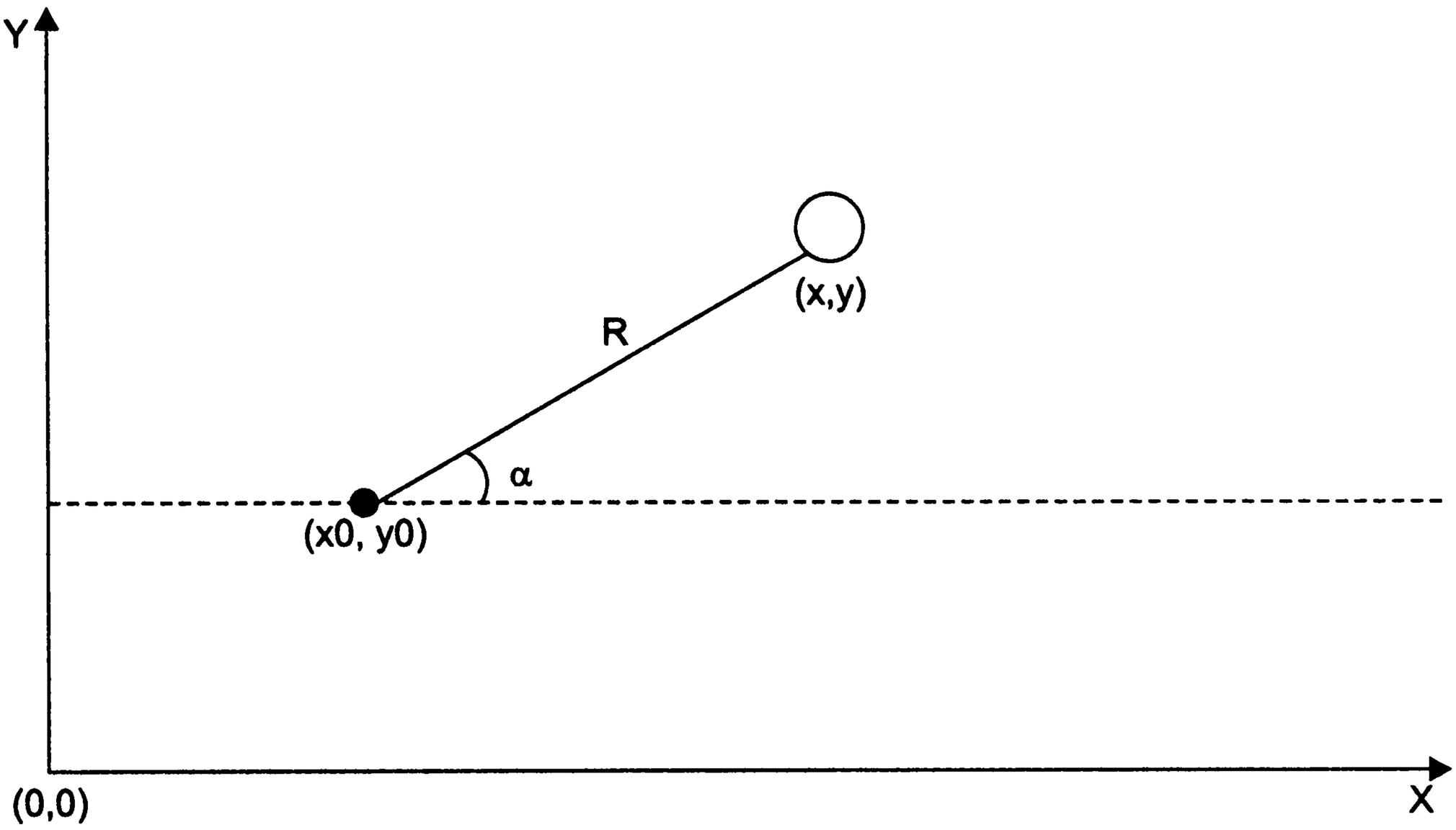


Рис. 1.9. Определение положения объекта по углу отклонения

Зная угол отклонения (угол α), можно определить координаты центра шарика по формулам:

```
x = x0 + R * Cos(α)
y = y0 + R * Sin(α)
```

Этого уже достаточно, чтобы написать процедуру движения шарика по окружности (для начала на псевдокоде):

```
a := 0
ЦИКЛ
    стереть шарик с экрана
    вычислить новую позицию шарика:
        x := x0 + R * Cos(a)
        y := y0 + R * Sin(a)
    нарисовать шарик на новой позиции
    a := a + угловая_скорость
    ЕСЛИ a > 2 * PI
        a := a - 2 * PI
КОНЕЦ ЦИКЛА
```

Здесь мы встречаемся с новой величиной — угловой скоростью, которая определяет, на сколько радианов за одну итерацию сдвинется шарик. Еще одна строчка, которая не имела аналогов в предыдущих моделях, — `a := a - 2 * PI`. Поскольку угол `a` и угол `a - 2 * PI` — это один и тот же угол, стоит периодически уменьшать значение `a` на `2 * PI` (в противном случае оно рано или поздно[1] станет слишком большим, и произойдет переполнение, что не есть хорошо).

[1] На самом деле произойдет это, скорее всего, весьма не скоро. Но оставлять подобные «часовые бомбы» без внимания не стоит.

Переводя этот псевдокод в настоящую программу, можно получить Землю, вращающуюся вокруг Солнца. Осталось запрограммировать движение Луны.

Заметьте, нигде не было оговорено, что x0 и y0 – величины постоянные. Действительно, никто не запрещает нам изменять их значения во время работы программы. Представьте теперь, что (x0, y0) – не фиксированная точка, а точка, движущаяся по окружности (например, центр Земли). Тогда получаемые координаты (x, y) будут координатами центра Луны, вращающейся вокруг Земли!

Приступим к реализации модели. Я снова использовал «Молекулу в закрытом сосуде» в качестве основы, но процедуру TForm1.StartStopBtnClick() написал заново (листинг 1.5).

Листинг 1.5. Модель «Солнечная система»

```
procedure TForm1.StartStopBtnClick(Sender: TObject);
    var EarthX, EarthY   : Integer;  { координаты Земли }
        EarthA           : Real;     { текущий угол отклонения Земли }
        MoonX, MoonY     : Integer;  { координаты Луны }
        MoonA            : Real;     { текущий угол отклонения Луны }

    const SunR   : Integer = 60;     { радиус Солнца }
          EarthR : Integer = 20;     { радиус Земли }
          MoonR  : Integer = 4;      { радиус Луны }
          EarthD : Integer = 140;    { расстояние от Солнца до Земли }
          MoonD  : Integer = 40;     { радиус от Земли до Луны }
          EarthV : Real = 0.02;      { угловая скорость Земли }
          MoonV  : Real = 0.1;       { угловая скорость Луны }

begin
    if IsRunning then
    begin
        IsRunning := false;
        StartStopBtn.Caption := 'Пуск';
        Exit;
    end;

    StartStopBtn.Caption := 'Стоп';
    IsRunning := true;

    EarthA := 0; MoonA := 0;          { инициализация переменных }
    EarthX := 0; EarthY := 0;
    MoonX := 0; MoonY := 0;

    Screen.Refresh;                          { очистка экрана }
    Screen.Canvas.Pen.Width := 5;            { толщина линий - 5 пикселов }
    Screen.Canvas.Pen.Color := clRed;        { рисуем Солнце в центре экрана }
    Screen.Canvas.Ellipse(Screen.Width div 2 - SunR,
                          Screen.Height div 2 - SunR,
                          Screen.Width div 2 + SunR,
                          Screen.Height div 2 + SunR);

    while IsRunning do                                { основной цикл }
    begin
```

```
Screen.Canvas.Pen.Color := clBtnFace;          { стираем Землю и Луну }
Screen.Canvas.Ellipse(EarthX - EarthR, EarthY - EarthR,
                      EarthX + EarthR, EarthY + EarthR);
Screen.Canvas.Ellipse(MoonX - MoonR, MoonY - MoonR,
                      MoonX + MoonR, MoonY + MoonR);
{ пересчитываем координаты Земли и Луны }

EarthX := Round(Screen.Width div 2 + EarthD * Cos(EarthA));
EarthY := Round(Screen.Height div 2 + EarthD * Sin(EarthA));
MoonX := Round(EarthX + MoonD * Cos(MoonA));
MoonY := Round(EarthY + MoonD * Sin(MoonA));

EarthA := EarthA + EarthV;  { изменяем текущие углы отклонения }
MoonA := MoonA + MoonV;     { Земли и Луны }

if EarthA > 2*PI then EarthA := EarthA - 2*PI; { корректируем углы }
if MoonA > 2*PI then MoonA := MoonA - 2*PI;    { если требуется }

Screen.Canvas.Pen.Color := clGreen;                    { рисуем Землю }
Screen.Canvas.Ellipse(EarthX - EarthR, EarthY - EarthR,
                      EarthX + EarthR, EarthY + EarthR);
Screen.Canvas.Pen.Color := clBlue;                     { и Луну }
Screen.Canvas.Ellipse(MoonX - MoonR, MoonY - MoonR,
                      MoonX + MoonR, MoonY + MoonR);
Sleep(30);                                             { пауза }
Application.ProcessMessages;
    end;
end;
```

Центром Солнца в программе считается центр экрана (Screen.Width div 2, Screen.Height div 2). Внешний вид работающего приложения показан на рис. 1.10.

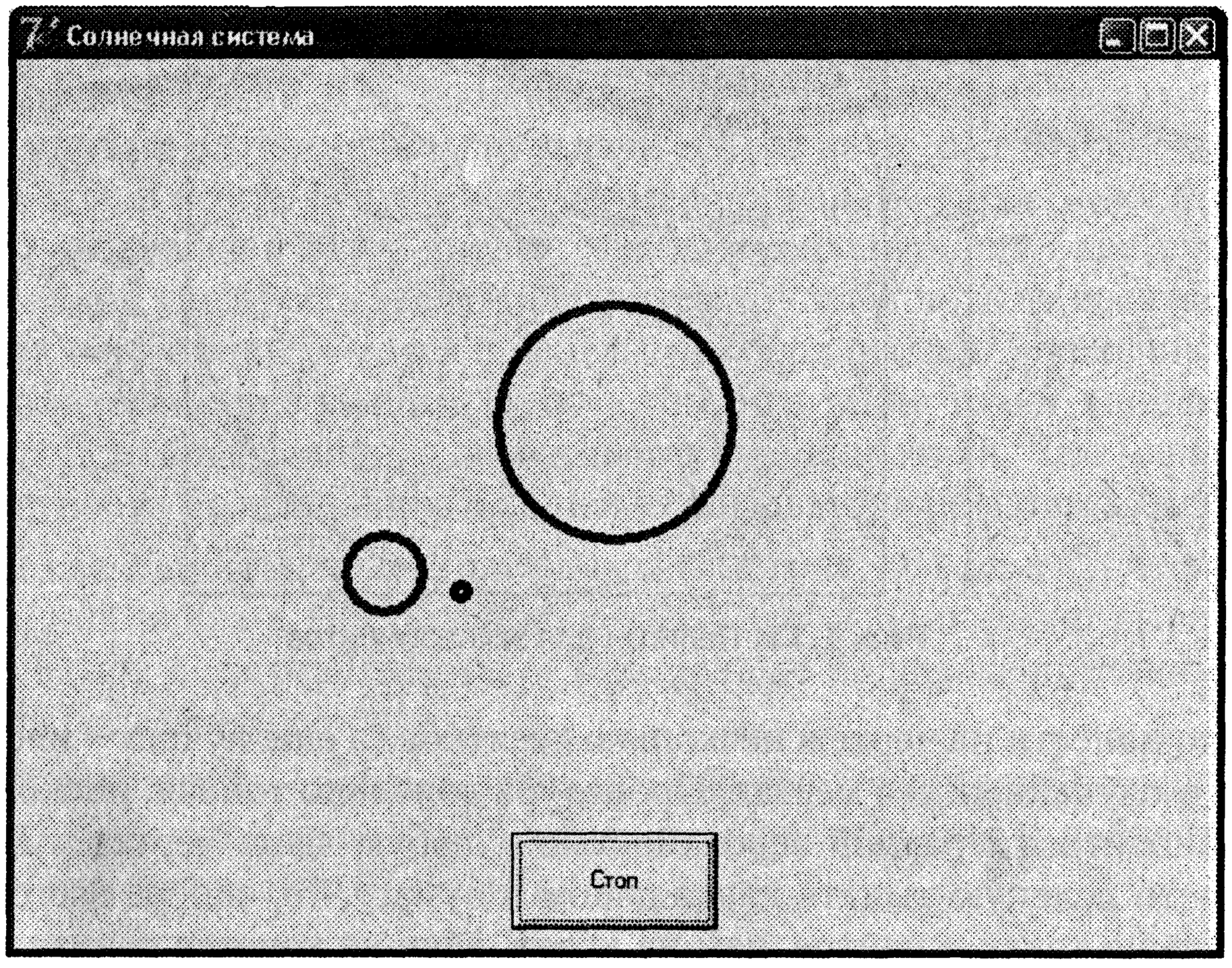


Рис. 1.10. Модель «Солнечная система»

Модель 7. «Экспериментальное определение числа π»

Предыдущая модель была последней, в которой мы имели дело с околофизическими явлениями. Сейчас мы рассмотрим одну задачу, взятую из математики, а потом перейдем к биологии.

Итак, модель номер семь интересна в первую очередь тем, что в ней очень мало натяжек. Если «Броуновское движение» было довольно далеко от реального броуновского движения, а «Солнечная система» становилась похожей на настоящую Солнечную систему только после очень сильной работы воображения, то сейчас дело обстоит иначе. Мы создадим действительно правдоподобную (хотя и не слишком наглядную) модель, которая с успехом может использоваться вместо «настоящего» эксперимента. А ведь это ситуация, к которой и надо стремиться в реальности (вспомните, о чем я говорил в начале главы – модели должны быть полезными!).

Итак, задача. Исходя из каких-то соображений (умозрительных, опытных, логических), мы полагаем, что площадь круга известного радиуса равна квадрату радиуса, умноженному на некоторую константу. Понятно, что эта «некоторая константа» есть не что иное, как число π (равное 3,14159...). Так вот, интересен факт, что примерное значение числа π можно определить экспериментально. Делается это так: на листе бумаги чертится круг радиусом r, а около него описывается квадрат со стороной 2r (рис. 1.11). Чем больше будет чертеж, тем лучше.

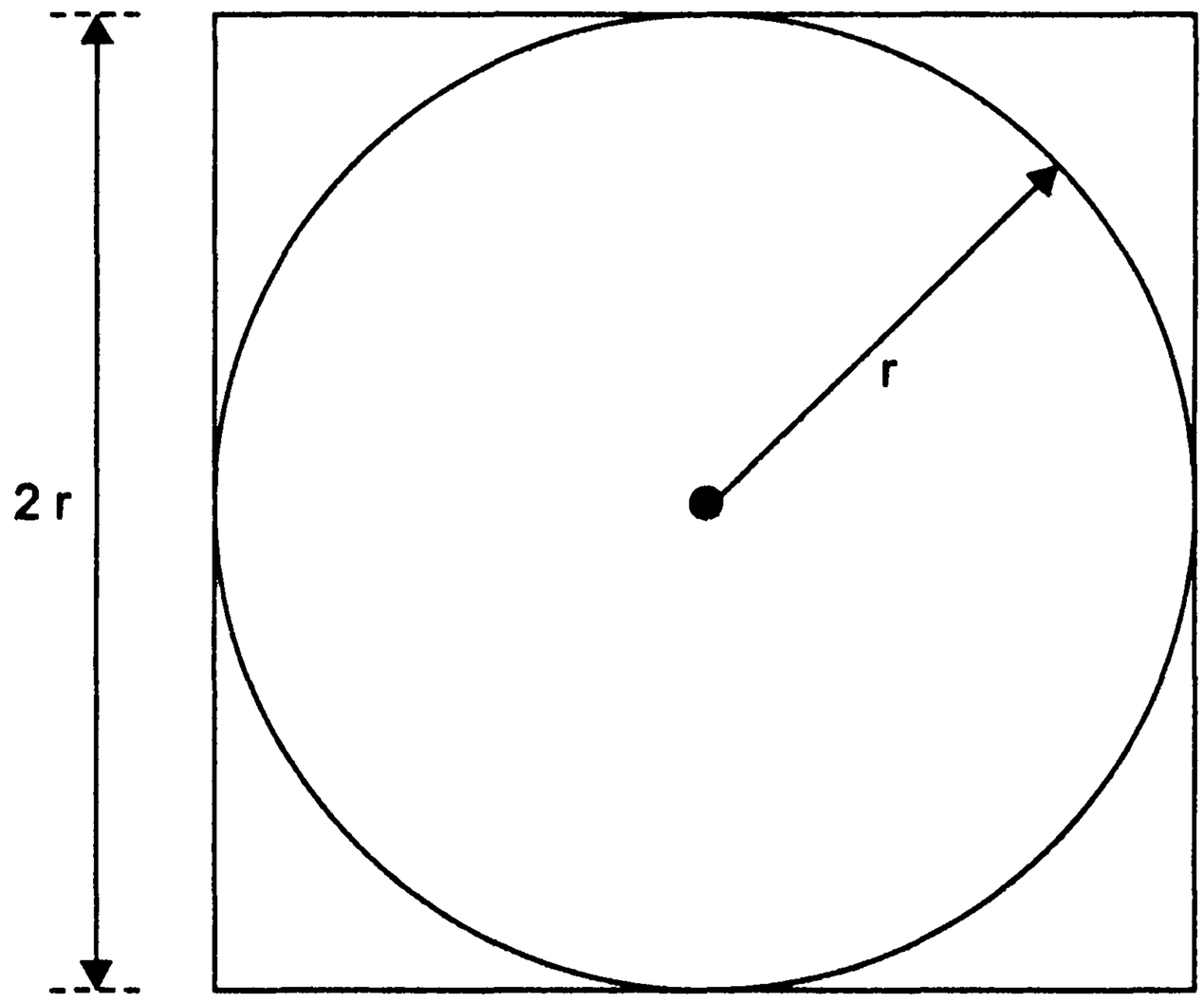


Рис. 1.11. Полигон для экспериментов

Теперь надо нанести на чертеж несколько десятков (а еще лучше – несколько сотен) точек в случайных его местах. Процедура довольно тонкая, поскольку точки должны наноситься с равной вероятностью в любую часть чертежа. Если же вы будете просто вслепую тыкать карандашом в бумагу, то, скорее всего, получите гораздо больше точек в центре, чем на краях. Мне не хочется сейчас изобретать изощренные способы того, как сделать это лучше, потому что, в конце концов,

процесс будет моделироваться на компьютере, а для него выбрать случайную точку в любой части чертежа с равной вероятностью не составит труда.

Следующий шаг — подсчет общего количества точек (N) и количества точек, попавших в круг (NO). Утверждается[1], что при достаточно большом числе экспериментов отношение N к NO примерно равно отношению площадей соответствующих фигур, то есть N / NO ≈ площадь_квадрата / площадь_круга. Причем, чем больше экспериментов было проделано, тем точнее формула. Из соотношений

площадь_квадрата = $(2r)^2$
площадь_круга = $\pi\ r^2$

получаем:

$N / NO \approx (2r)^2 / (\pi\ r^2)$.

откуда

$\pi \approx 4 * NO / N$

На этом математический анализ проблемы закончен. Теперь попробуем приспособить компьютер к ее решению, чтобы не тыкать бездумно карандашом в бумагу. В принципе, есть лишь два вопроса, на которые необходимо ответить:

1. Как выбрать случайную точку на чертеже?
2. Как определить, что точка лежит внутри круга?

Во-первых, будем считать, что центр круга совпадает с началом координат, а его радиус равен R. Тогда, выбирая пару значений из диапазона [-R, R], мы получаем абсциссу и ординату случайной точки, лежащей в пределах чертежа. Во-вторых, поскольку центр круга совпадает с началом координат, определить, попадает ли точка (x, y) в круг, очень просто: если расстояние D := Sqrt(x*x + y*y) меньше или равно радиусу круга, то попадает, иначе — нет.

Хотя на первый взгляд компьютерная модель недостатков лишена, это не совсем так. Первое: мы не можем до конца положиться на качество генератора случайных чисел Delphi. Вполне возможно, что он не дает достаточно хорошего равномерного распределения значений. Тем не менее мы им воспользуемся; писать собственный генератор случайных чисел в наши задачи сейчас не входит. Второе: выбираемые случайные точки всегда будут иметь целые координаты (поскольку функция RandomRange() возвращает целое число). Чтобы уменьшить пагубное влияние этого фактора, можно увеличить значение радиуса круга (вплоть до десятков тысяч) — тогда вклад «неучтенных» точек с нецелыми координатами в конечный результат будет сравнительно небольшим. Дело в том, что такие точки влияют на общую картину происходящего лишь в окрестностях границы круга. Чем меньше площадь этой окрестности по сравнению с общей площадью фигуры, тем меньше влияние.

Перед тем как привести текст программы (листинг 1.6), замечу, что кроме текущего значения числа π (вычисляемого как 4 * NO / N) полезно вывести среднее арифметическое всех результатов — оно меньше подвержено резким скачкам и может дать более точную оценку.

«Скелет» программы — вновь модель «Молекула в закрытом сосуде», но на сей раз я удалил объект Screen с главной формы приложения и добавил несколько надписей (элементов типа TLabel) (табл. 1.1).

---

[1] Мне это кажется интуитивно очевидным, но строгое математическое доказательство утверждения довольно громоздко.

Таблица 1.1. Надписи элементов

Название	Назначение
ExpNumber	количество экспериментов
CurrentPI	текущее экспериментальное значение π
MeanPI	среднее значение π

Листинг 1.6. Экспериментальное определение числа π

```
procedure TForm1.StartStopBtnClick(Sender: TObject);
    var N, N0 : Integer;
        x, y  : Integer;
        Sum   : Real;

    const R   : Integer = 20000;   { радиус круга }

begin
    if IsRunning then
    begin
        IsRunning := false;
        StartStopBtn.Caption := 'Пуск';
        Exit;
    end;

    StartStopBtn.Caption := 'Стоп';
    IsRunning := true;

    N := 0;
    N0 := 0;
    Sum := 0;
    Randomize;

    while IsRunning do                              { основной цикл }
    begin
        N := N + 1;                          { количество экспериментов }
        x := RandomRange(-R, R);             { определяем случайную точку }
        y := RandomRange(-R, R);
        if Sqrt(x*x + y*y) <= R then N0 := N0 + 1;  { если она лежит в круге }

        ExpNumber.Caption := IntToStr(N);
        CurrentPI.Caption := FloatToStr(4 * N0 / N);
        Sum := Sum + 4 * N0 / N;
        MeanPI.Caption := FloatToStr(Sum / N);

        Application.ProcessMessages;

    end;
end;
```

Внешний вид готового приложения показан на рис. 1.12.

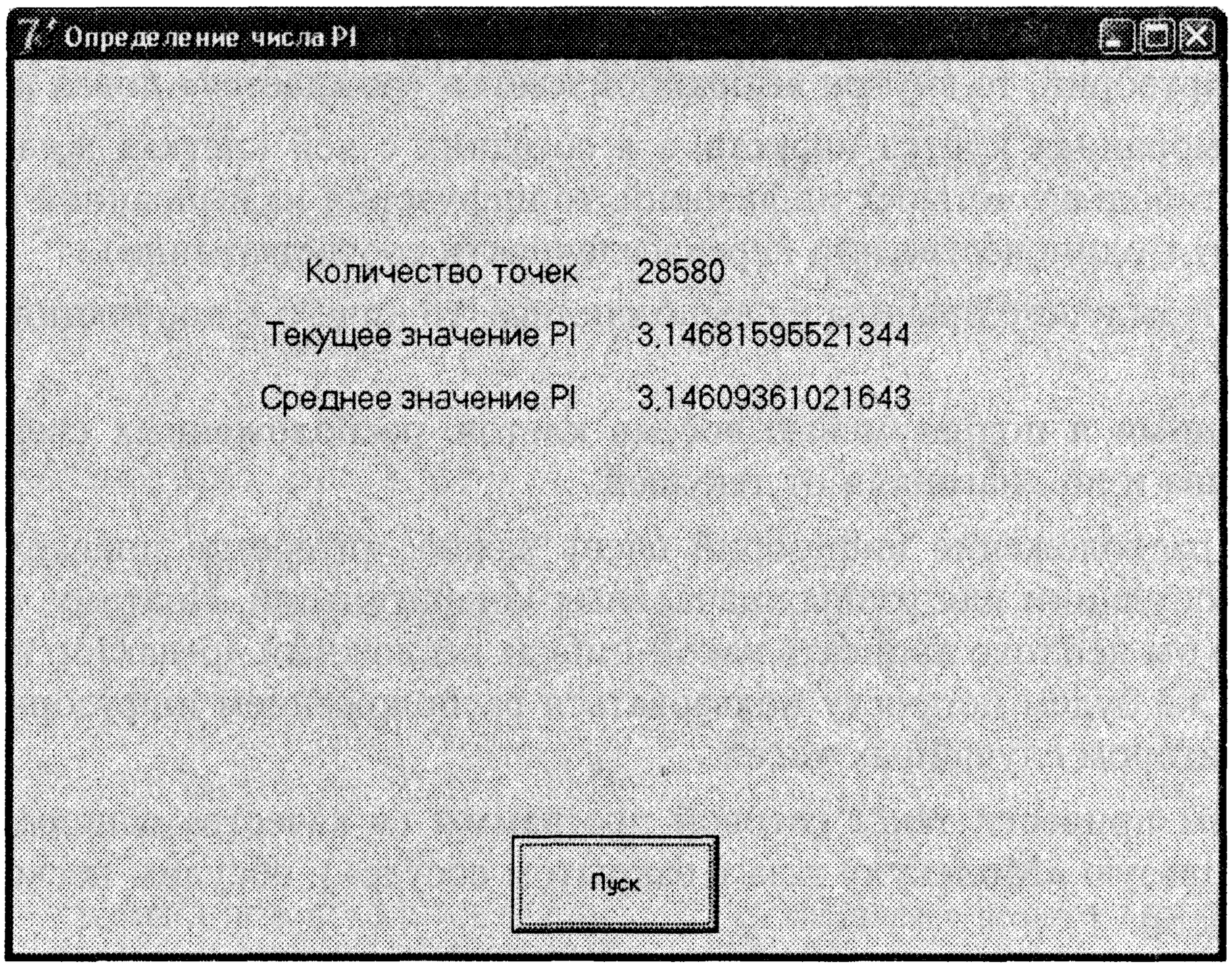


Рис. 1.12. Экспериментальное определение числа π

Модель 8. «Жизнь»

Насколько мне известно, моделирование действительно используется в биологии очень интенсивно. Многие процессы, интересующие биологов (например, динамика возрастного состава некоторой популяции или количества особей в ней), могут быть более или менее точно описаны математически и смоделированы на компьютере. Конечно, предсказывать трудно, особенно будущее (по выражению Нильса Бора), но порою это необходимо. Хорошо известна печальная история с кроликами в Австралии: завезенные в середине XIX века, эти, казалось бы, безобидные зверьки нанесли огромный урон местной фауне, истребляя растительность (и тем самым лишая корма коренных обитателей материка). Опять-таки, компьютер и компьютерные модели — не панацея, но во многих случаях они могут помочь просчитать возможные сценарии событий и тем самым избежать непродуманных решений.

Мы начнем с простой модели, которая называется «Жизнь». Представьте себе прямоугольное клетчатое поле, в каждой клетке которого может «жить» некоторое существо (пусть это будет «инфузория»). Клетка также может быть пустой, то есть в ней в данный момент никто не живет. Модель задается двумя параметрами: начальной конфигурацией инфузорий (их количеством и расположением на поле) и некоторыми «биологическими законами», регулирующими жизнь популяции. На каждой итерации моделирования выбирается случайная клетка поля (а как же без случайного фактора в жизни?!) и к ней применяются биологические законы. Сами же законы могут быть, к примеру, такими:

1. Если выбранная клетка пуста, а в соседних с ней клетках находится более двух инфузорий, то внутри выбранной клетки «рождается» новая инфузория.
2. Если выбранная клетка непуста, а в соседних с ней клетках живет меньше трех или больше четырех инфузорий, то инфузория из выбранной клетки погибает (от одиночества или перенаселенности соответственно).
3. Если предыдущие правила не выполнились, то ничего в популяции не происходит.

Под соседними я подразумеваю восемь клеток, расположенных сверху, снизу, слева, справа и по диагонали от текущей.

Исходное расположение инфузорий очень важно; интересно пронаблюдать изменения популяции для разных начальных конфигураций. Поэтому, вообще говоря, было бы неплохо написать какой-нибудь несложный «редактор конфигураций». Мы же будем попросту пользоваться расположением инфузорий, созданным генератором случайных чисел.

Прежде чем привести текст готовой программы (в качестве основы, кстати, я опять использую «Молекулу газа в закрытом сосуде»), обсудим несколько важных моментов.

1. Состояние поля удобно хранить в двумерном массиве:

   ```
   Field[0..WIDTH + 1][0..HEIGHT + 1] of Boolean
   ```

   Значение true будет соответствовать клетке с инфузорией, false — пустой клетке. Обратите внимание, что массив содержит две дополнительные строки и два дополнительных столбца.

2. Проще всего посчитать количество инфузорий, соседствующих с клеткой [x, y], с помощью кода:

   ```
   s := 0;
   for i := -1 to 1 do
       for j := -1 to 1 do
           s := s + Ord(Field[x + i][y + j]);  { Ord(true) = 1, Ord(false) = 0 }
   s := s - Ord(Field[x][y]);          { если в [x, y] есть инфузория, вычитаем ее }
   ```

   Именно для корректной работы этой процедуры массив приходится делать больше, чем требуют реальные размеры поля. В противном случае произойдет выход за границы массива, если клетка [x, y] находится на краю или в углу поля.

3. Предположим, что в клетке [x, y] находится инфузория. В каком месте экрана ее рисовать? Иными словами, как по данным координатам в поле получить координаты на экране? Пока что я приведу готовые формулы, но мы еще вернемся к этой задаче. Итак, если размеры поля равны WIDTH × HEIGHT, а размеры экрана — Screen.Width × Screen.Height, то каждой клетке поля будет соответствовать прямоугольник Screen.Width / WIDTH × Screen.Height / HEIGHT на экране. Обозначим через Rx половину его ширины, а через Ry — половину его высоты. Тогда центр прямоугольника находится по формулам:

   ```
   Xцентра := (2*x - 1)*Rx
   Yцентра := (2*y - 1)*Ry
   ```

Сама программа совершенно бесхитростна и прямолинейна (листинг 1.7).

Листинг 1.7. Модель «Жизнь»

```
procedure TForm1.StartStopBtnClick(Sender: TObject);
    const WIDTH = 30;                                    { ширина и }
          HEIGHT = 25;                                   { высота поля }
    var   x, y    : Integer;
          Rx, Ry  : Integer;                           { ширина, высота клетки }
          Field   : array [0..WIDTH + 1, 0..HEIGHT + 1] of Boolean;
          i, j, s : Integer;

begin
    if IsRunning then
begin
    IsRunning := false;
    StartStopBtn.Caption := 'Пуск';
    Exit;
end;

StartStopBtn.Caption := 'Стоп';
IsRunning := true;

Rx := (Screen.Width div WIDTH) div 2;          { определяем размеры клетки }
Ry := (Screen.Height div HEIGHT) div 2;

Randomize;
Screen.Refresh;                                { очистка экрана }

Screen.Canvas.Pen.Color := clBlue;
for i := 0 to WIDTH + 1 do                        { очистка поля }
    for j := 0 to HEIGHT + 1 do
        Field[i, j] := false;

for i := 1 to WIDTH do                         { создаем начальную конфигурацию }
    for j := 1 to HEIGHT do
        if Random(4) = 0 then                  { в среднем будет одна инфузория }
        begin                                  { на четыре клетки }
            Field[i, j] := true;
            Screen.Canvas.Ellipse((2*i - 1)*Rx - Rx, (2*j - 1)*Ry - Ry,
                                  (2*i - 1)*Rx + Rx, (2*j - 1)*Ry + Ry);
        end;

while IsRunning do                                       { основной цикл }
begin
    x := RandomRange(1, WIDTH);                  { выбираем случайную клетку }
    y := RandomRange(1, HEIGHT);

    s := 0;                                      { подсчитываем соседей }
    for i := -1 to 1 do
        for j := -1 to 1 do
            s := s + Ord(Field[x + i][y + j]);
    s := s - Ord(Field[x][y]);

    if (Field[x, y] = false) and (s > 2) then  { создаем новую инфузорию }
    begin
        Screen.Canvas.Pen.Color := clBlue;
```

```
        Screen.Canvas.Ellipse((2*x - 1)*Rx - Rx, (2*y - 1)*Ry - Ry,
                              (2*x - 1)*Rx + Rx, (2*y - 1)*Ry + Ry);
        Field[x, y] := true;
    end
    else if (Field[x, y] = true) and ((s < 3) or (s > 4)) then { удаляем }
    begin                                                      { инфузорию }
        Screen.Canvas.Pen.Color := clBtnFace;
        Screen.Canvas.Ellipse((2*x - 1)*Rx - Rx, (2*y - 1)*Ry - Ry,
                              (2*x - 1)*Rx + Rx, (2*y - 1)*Ry + Ry);
        Field[x, y] := false;
    end;

            Sleep(5);                                          { пауза 5 миллисекунд }
            Application.ProcessMessages;
        end;
    end;
```

Обратите внимание, что в качестве «инфузории» рисуется эллипс, занимающий собой все пространство клетки на экране. Готовое приложение изображено на рис. 1.13.

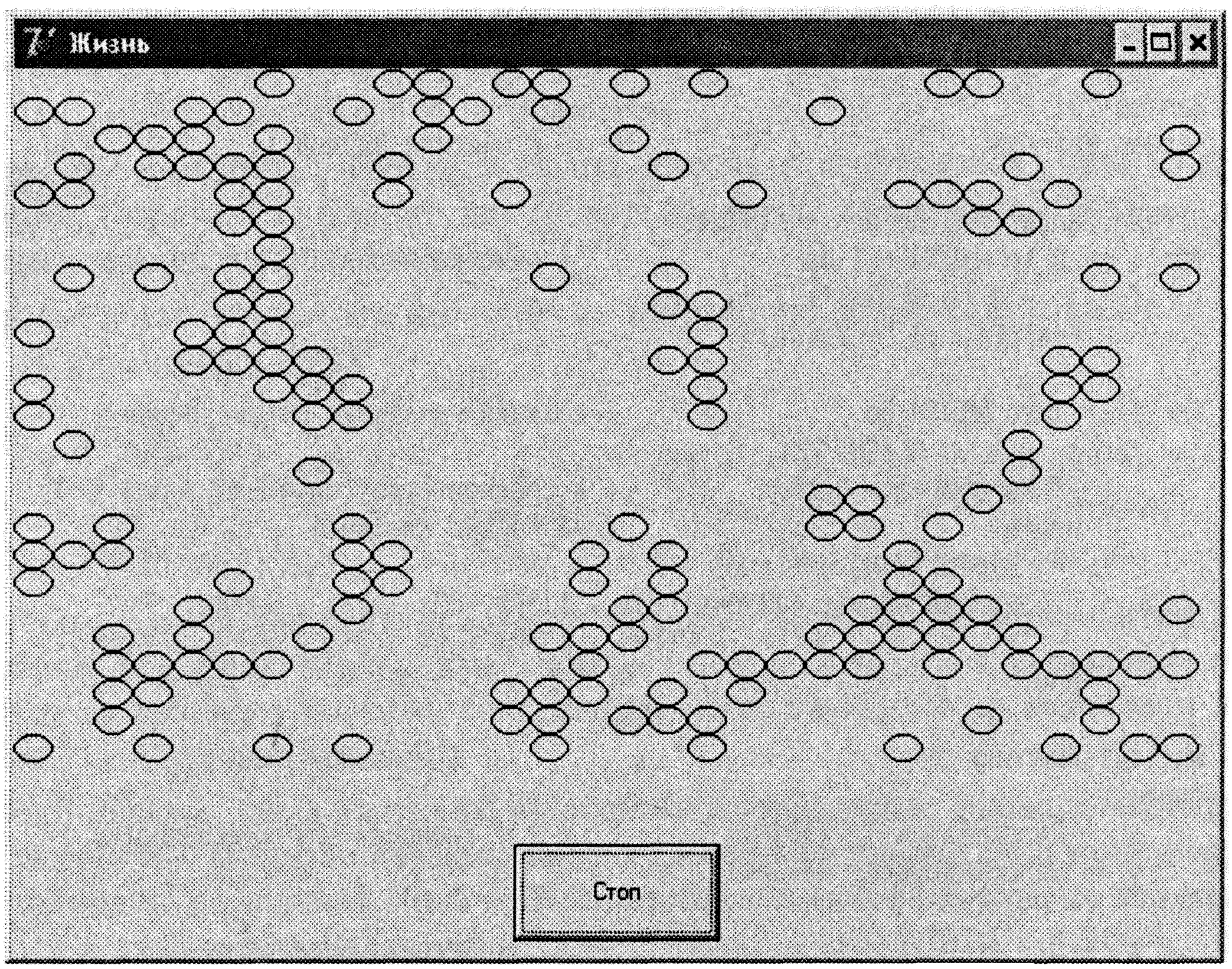


Рис. 1.13. Модель «Жизнь» в работе

Теперь обсудим задачу о том, как по данным координатам инфузории в поле найти координаты на экране. Я намеренно останавливаю на ней ваше внимание, поскольку она очень и очень часто встречается на практике в самых разных областях; решается же эта задача просто, поэтому лучше один раз решить ее в общем виде и уже не возвращаться к ней.

Задачу можно сформулировать в общем виде так: даны два интервала – [a, b] и [A, B]. Требуется каким-то образом сопоставить каждой точке из [a, b] некоторую точку из [A, B], причем точке a должна соответствовать точка A, а точке b – точка B. В нашем случае, например, чтобы определить горизонтальную координату центра инфузории, положим

```
[a, b] = [1, WIDTH]
[A, B] = [Rx, Screen.Width - Rx]
```

а чтобы найти вертикальную:

```
[a, b] = [1, HEIGHT]
[A, B] = [Ry, Screen.Height - Ry]
```

Как правило, требуется найти *линейное* сопоставление (или, как говорят математики, *отображение*). Это значит, что точки из первого интервала будут равномерно рассеиваться по второму: к примеру, точки из половины первого интервала заполнят ровно половину второго. Формула линейного отображения, как известно, выглядит так:

```
y = k * x + c
```

В нашем случае x – это координата в первом интервале, а y – во втором (та, которую требуется определить). Поскольку левая граница интервала отображается в левую границу второго интервала, получаем уравнение:

```
A = k * a + c
```

Аналогично для правой границы:

```
B = k * b + c
```

Теперь у нас есть два уравнения и две неизвестные – k и c. Решая систему, получаем:

```
k = (A - B) / (a - b)
c = A - a * (A - B) / (a - b)
```

Поскольку все неизвестные определены, можно использовать формулу линейного отображения для нахождения требуемой координаты. К примеру, в нашем случае горизонтальная координата центра инфузории находится так:

```
k = (Rx - (Screen.Width - Rx)) / (1 - WIDTH) =
    (2*Rx - Screen.Width) / (1 - WIDTH)
c = Rx - 1 * (Rx - (Screen.Width - Rx)) / (1 - WIDTH) =
    Rx - (2*Rx - Screen.Width) / (1 - WIDTH)
коорд_на_экране = k * коорд_в_поле + c
```

Формула, которую я привел в реальной программе, выглядит гораздо короче, но на самом деле ее можно получить из этой, если вспомнить, что Rx = Screen.Width / (2 * WIDTH), раскрыть скобки и выполнить действия.

Модель 9. «Жизнь» Джона Конуэя

Особенностью предыдущей модели была сильная зависимость ее поведения от генератора случайных чисел. Модель, которую мы рассмотрим сейчас, не зависит от него вообще: зная начальную конфигурацию колонии существ, мы сможем предсказать состояние модели через одну, две или хоть тысячу итераций. Автор-

ство этой замечательной модели принадлежит математику Джону Конуэю (John Conway). Во многом благодаря статье Мартина Гарднера в журнале Scientific American, написанной в теперь уже далеком 1970 году, «Жизнь» стала широко известна во всем мире. Хорошее описание модели можно найти в книге Гарднера «Математические досуги» (второе русское издание вышло в 2000 году).

«Жизнь» Джона Конуэя отличается прекрасной сбалансированностью «правил игры», поэтому в ней можно получить уйму красивых конфигураций, изменяющихся со временем самым причудливым образом.

Итак, правила:

1. Если у инфузории есть два или три соседа, она выживает.
2. Если у инфузории больше трех или меньше двух соседей, она погибает.
3. Если у пустой клетки есть ровно три соседа-инфузории, на ней рождается новая инфузория.

В отличие от предыдущей модели, здесь правила применяются ко всем клеткам поля на каждой итерации моделирования. Например, одинокая инфузория заведомо погибнет на ближайшем ходу, в то время как в предыдущей модели она погибла бы только после того, как на нее укажет генератор случайных чисел. Важно понять, что гибель и рождение происходят на всем поле одновременно: к примеру, если мы выяснили, что на данной клетке произойдет рождение, то создавать новую инфузорию следует не раньше, чем будут рассмотрены все остальные клетки поля (то есть мы знаем, что инфузория родится *в будущем*, но пока еще клетка пуста).

Поскольку в «Жизни» Конуэя интересны конкретные конфигурации инфузорий, без «редактора конфигураций» нет смысла писать основную программу. Тем не менее я на этом останавливаться не буду: во-первых, простейший редактор написать несложно; во-вторых, можно попросту загружать готовое поле из текстового файла нулей и единиц, а текстовый файл редактировать Блокнотом; в-третьих, можно вообще явно присвоить требуемые значения тем или иным клеткам поля внутри самой программы, тем самым обойдясь на первых порах без редактора.

Самое большое изменение в этой модели по сравнению с предыдущей связано именно с одновременностью рождения и смерти. Программа теперь должна работать в два этапа: на первом выясняется, на каких клетках должны произойти изменения, а на втором эти изменения реально производятся. Код программы с комментариями приведен в листинге 1.8.

Листинг 1.8. «Жизнь» Джона Конуэя

```
procedure TForm1.StartStopBtnClick(Sender: TObject);
    const WIDTH = 30;
          HEIGHT = 25;
    var   x, y    : Integer;
          Rx, Ry  : Integer;
          Field   : array [0..WIDTH + 1, 0..HEIGHT + 1] of Boolean;
          Changes : array [0..WIDTH + 1, 0..HEIGHT + 1] of Boolean;
          s, i, j : Integer;
```

```
begin
    if IsRunning then
    begin
        IsRunning := false;
        StartStopBtn.Caption := 'Пуск';
        Exit;
    end;

    StartStopBtn.Caption := 'Стоп';
    IsRunning := true;

    Rx := (Screen.Width div WIDTH) div 2;
    Ry := (Screen.Height div HEIGHT) div 2;

    Screen.Refresh;                              { очистка экрана }

    for i := 0 to WIDTH + 1 do
        for j := 0 to HEIGHT + 1 do
        begin
            Field[i, j] := false;                { очистка поля }
            Changes[i, j] := false;
        end;

    { здесь внести строки, задающие конфигурацию инфузорий }

    while IsRunning do                           { основной цикл }
    begin
        for i := 1 to WIDTH do                   { отображение текущего состояния }
            for j := 1 to HEIGHT do
            begin
                if Field[i, j] then
                    Screen.Canvas.Pen.Color := clBlue
                else
                    Screen.Canvas.Pen.Color := clBtnFace;
                Screen.Canvas.Ellipse((2*i - 1)*Rx - Rx, (2*j - 1)*Ry - Ry,
                                      (2*i - 1)*Rx + Rx, (2*j - 1)*Ry + Ry);
            end;

        for x := 1 to WIDTH do
            for y := 1 to HEIGHT do
            begin                                { для каждой клетки поля }
                s := 0;                          { подсчет соседей }
                for i := -1 to 1 do
                    for j := -1 to 1 do
                        s := s + Ord(Field[x + i][y + j]);
                s := s - Ord(Field[x][y]);

                { если произошло рождение или смерть }
                if ((Field[x, y] = false) and (s = 3)) or
                   ((Field[x, y] = true)  and ((s < 2) or (s > 3))) then
                        Changes[x, y] := true;
    end;
```

продолжение

Листинг 1.8 (продолжение)

```
        for x := 1 to WIDTH do                              { внесение изменений }
            for y := 1 to HEIGHT do
                if Changes[x, y] then
                begin
                    Field[x, y] := not Field[x, y];    { меняем состояние на }
                    Changes[x, y] := false;                { противоположное }
                end;
        Sleep(100);
        Application ProcessMessages;
    end;
end;
```

Массив Changes показывает, произойдет ли на данной клетке «изменение» (то есть рождение для пустой или смерть для непустой клетки). Вообще говоря, достаточно задать в качестве его размерностей 1..Width и 1..Height; я не сделал этого только затем, чтобы не писать второй цикл инициализации, а воспользоваться уже готовым [1].

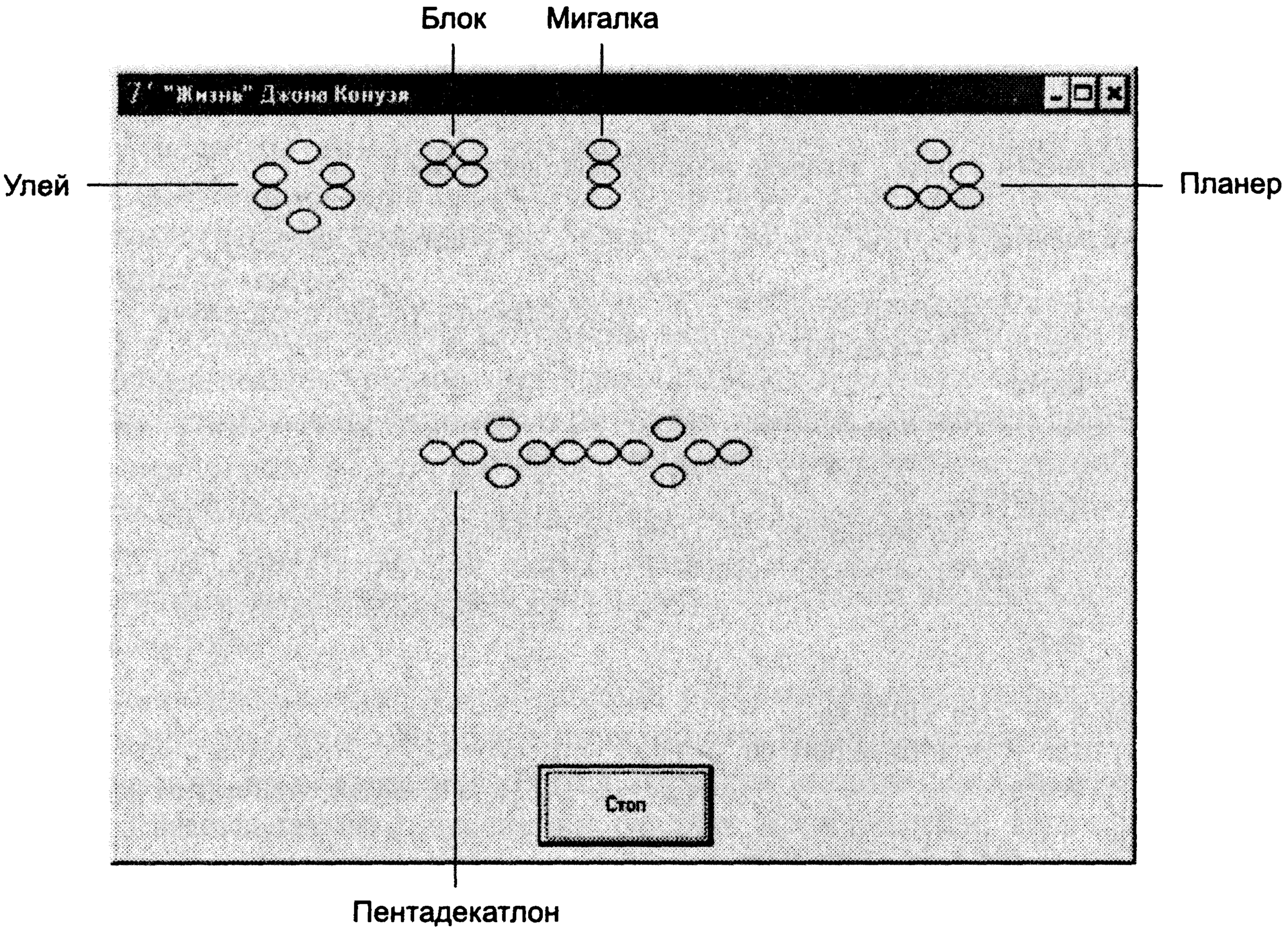


Рис. 1.14. Конфигурации в модели «Жизнь» Джона Конуэя

В уже упомянутой статье Гарднера описывается несколько интересных ситуаций, которые возникают в «Жизни» Конуэя. Во-первых, из всех возможных конфигураций можно выделить подклассы «стабильных» и «периодических» [2].

[1] Только не надо сокрушаться, что, мол, такие вот Windows написали!

[2] Знатоки «Жизни» выделяют пять разных типов конфигураций, да еще и с подтипами!

Например, «блок» и «улей» (рис. 1.14) – стабильные конфигурации. Они находятся в устойчивом состоянии и никогда не меняются. «Мигалка» – периодическая конфигурация, с периодом, равным двум ходам (то есть через каждые два хода мигалка возвращается к первоначальному состоянию). Есть и куда более изощренные периодические конфигурации, например, «пентадекатлон», возвращающийся к первоначальному состоянию через 15 ходов! Очень интересны конфигурации вроде «планера», которые тоже, в принципе, являются периодическими, но при этом они не просто возвращаются к первоначальному состоянию, но и сдвигаются относительно него на какое-то расстояние (то есть «летят» по игровому полю).

Долгое время оставался открытым вопрос о том, существуют ли популяции, способные расти бесконечно? За решение этой задачи предлагалась даже денежная премия. В конце концов, ученым удалось найти конфигурацию, впоследствии названную «планерным ружьем», которая через каждые несколько ходов «выпускает» новый планер, тем самым, естественно, увеличивая популяцию.

«Жизнь» Джона Конуэя тесно связана с так называемой теорией клеточных автоматов. В настоящее время с «Жизнью» связано много исследований, этой модели посвящено большое количество ресурсов Интернета.

Модель 10. «Черепашья графика»

Эта модель прямо связана с компьютерной графикой, но, кроме того, поучительна в общем смысле, поскольку демонстрирует, как иногда легко решается задача, если просто взглянуть на нее с другой стороны. Концепцию «черепашьей графики» напрямую поддерживали некогда популярный обучающий язык Лого, а также многие старые версии Бейсика. Теперь же она, как мне кажется, постепенно (и совершенно незаслуженно!) забывается.

Какого рода операции используются для рисования в современных средах программирования? Обычно это что-то вроде установки точки, рисования отрезка, эллипса, дуги и т. д. Многие среды (например, Delphi), облегчают рисование, предоставляя готовые функции для вывода на экран прямоугольника или даже многоугольника. Конечно, разные библиотеки предоставляют разные возможности, но суть не в этом; суть в самой идеологии. Существуют какие-то базовые графические примитивы, и мы их можем выводить в ту или иную часть экрана.

Но такой подход – не единственно возможный; есть и другие идеи. Представьте себе, что где-то на экране находится «черепашка» с карандашом. «Черепашка» эта не простая, а довольно умная, поскольку может выполнять несколько команд:

`PEN_UP` – поднять карандаш (не рисовать);

`PEN_DOWN` – опустить карандаш (рисовать);

`GO(N)` – пройти вперед на расстояние в N пикселов;

`TURN(A)` – повернуться на угол в A градусов (если угол положительный, то влево, если отрицательный – вправо).

«Черепашка» может двигаться только в ту сторону, в которую смотрит. Чтобы осуществить движение в другую сторону, надо сначала попросить ее повернуться при помощи команды TURN. Предположим также, что «черепашка» изначально находится в точке (0, 0) – левом нижнем углу экрана[1], карандаш поднят, а смотрит она строго вверх.

В принципе, этого набора команд вполне достаточно для рисования. Часто его расширяют необязательными, но полезными командами вроде «сдвинуться в точку (X, Y) экрана» или «повернуться строго на юго-запад».

Теперь можно попробовать что-нибудь нарисовать при помощи «черепашки». Например, ломаная, изображенная на рис. 1.15, получается при выполнении программы:

```
TURN(-45)
GO(100)
PEN_DOWN
GO(200)
TURN(-45)
GO(300)
```

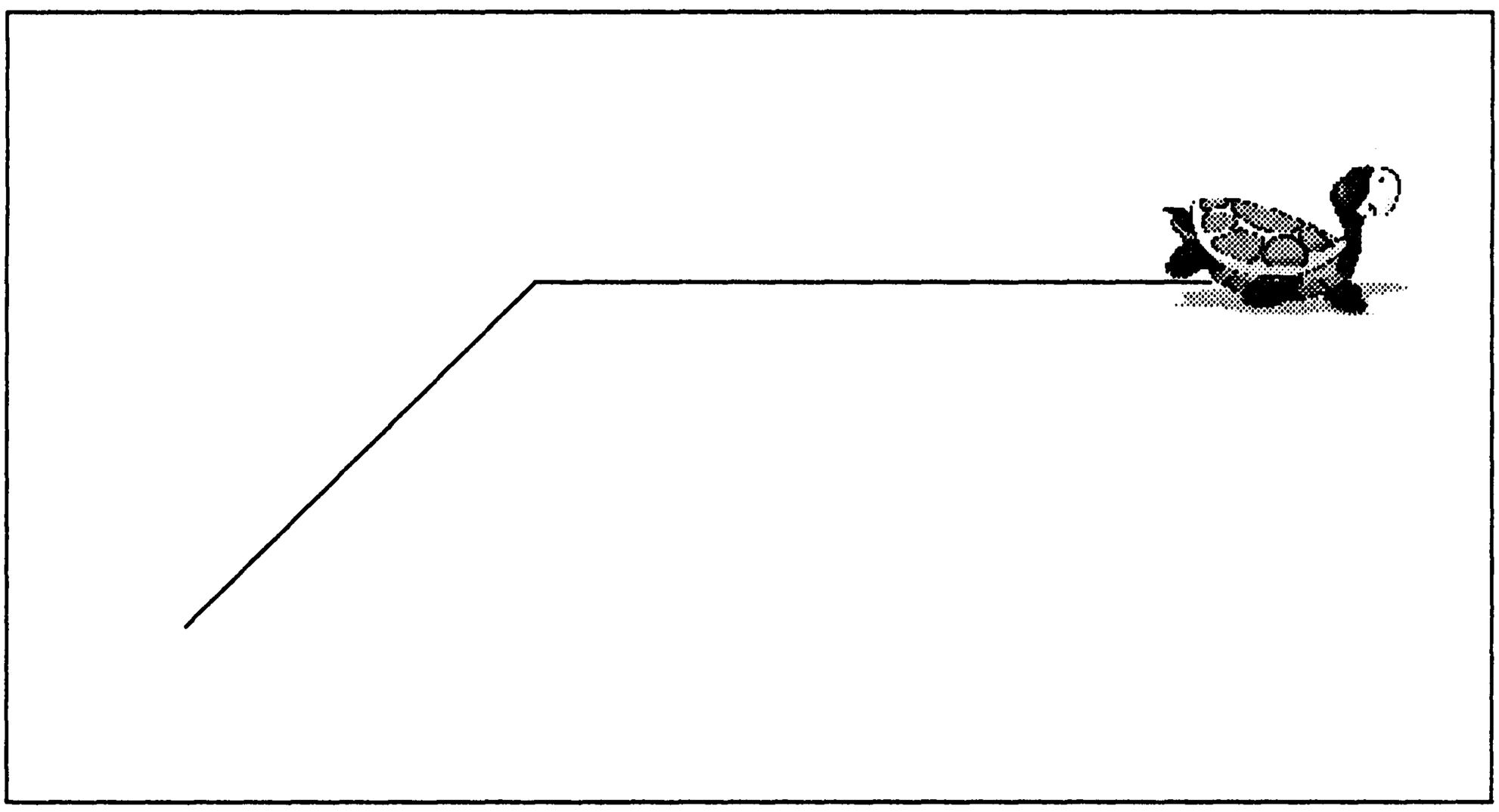


Рис. 1.15. «Черепашка» в действии

Совершенно ясно, что с помощью «черепашки» *можно* рисовать. Вопрос лишь в том, *целесообразно* это или нет. Так вот, порою концепция «черепашьей графики» ничего не дает, но иногда позволяет очень легко вычерчивать замысловатые фигуры, например спирали (рис. 1.16).

Левой спирали соответствует программа

```
TURN(-50);
GO(300);
PEN_DOWN;
len := 1;
for i := 1 to 300 do
```

[1] Обычно в компьютерной графике точкой (0, 0) считается левый верхний угол, но при работе с «черепашкой» мне кажется более удобным полагать, что ось ординат направлена вверх.

```
begin
    GO(Round(len));
    TURN(10);
    len := len + 0.1;
end;
```

правой:

```
TURN(-50);
GO(300);
PEN_DOWN;
len := 1;
for i := 1 to 40 do
begin
    GO(Round(len));
    TURN(90);
    len := len + 5;
end;
```

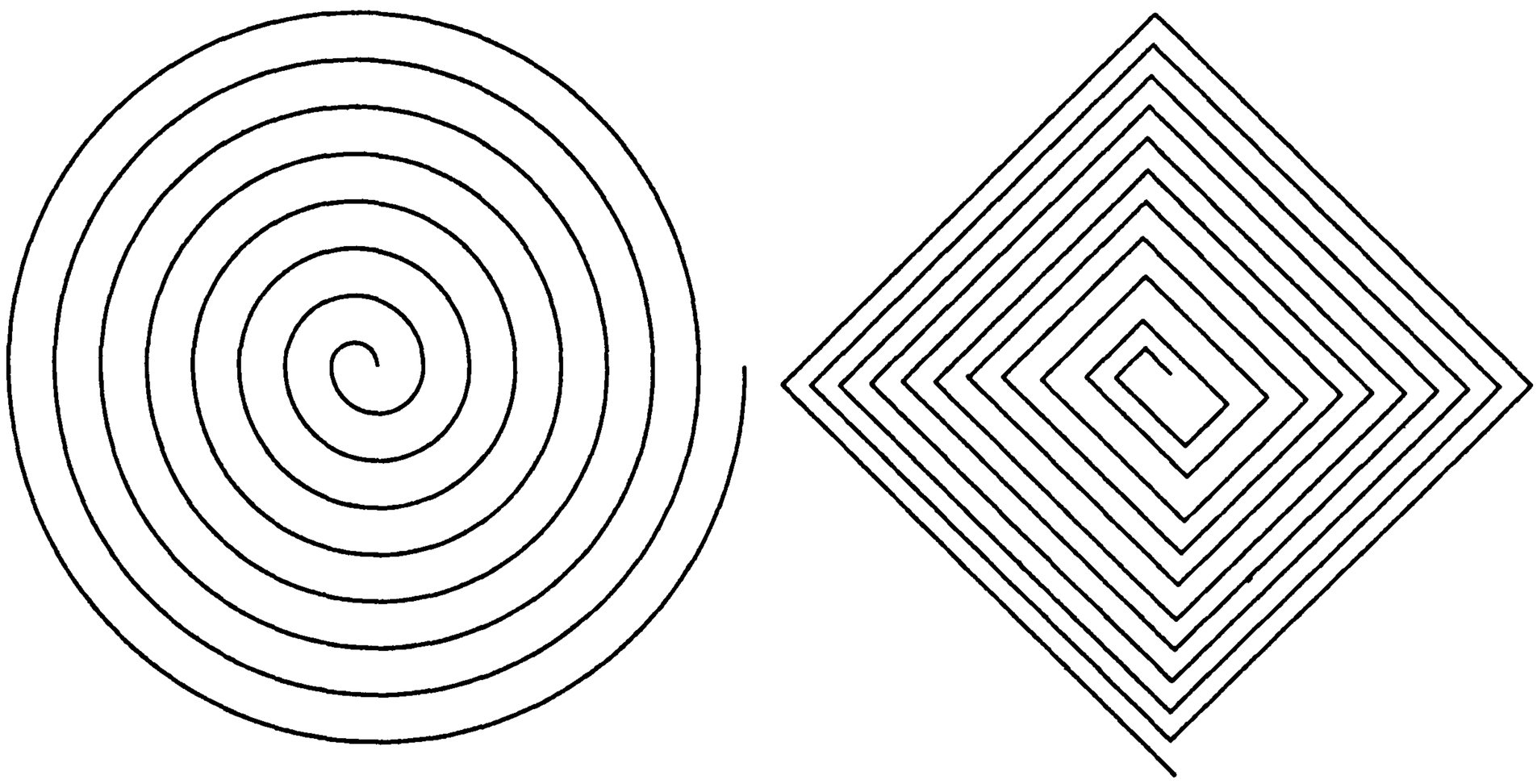


Рис. 1.16. Спирали, вычерченные «черепашкой»

Удобно с помощью «черепашки» рисовать ломаные, правильные многоугольники, да и вообще много чего.

Выяснив, что «черепашка» – это хорошая и полезная вещь, приступим к ее реализации. Нам потребуется написать лишь пять простых процедур – TURTLE_INIT (инициализация «черепашки»), GO, TURN, PEN_UP и PEN_DOWN (листинг 1.9).

Листинг 1.9. «Черепашка»

```
var x, y  : Integer;          { положение "черепашки" }
    angle : Real;             { и ее направление }
    pen   : Boolean;          { режим работы (рисовать/не рисовать) }

procedure TURTLE_INIT;        { инициализация }
begin
    x := 0;                           { начальное положение - левый }
    y := Form1.Screen.Height;         { нижний угол }
    angle := PI / 2;                  { начальное направление - "вверх" }
end;
```

продолжение

Листинг 1.9 (продолжение)

```
procedure GO(distance : Integer);    { пройти distance пикселов по прямой }
    var newx, newy : Integer;
begin
    newx := x + Round(distance * Cos(angle));    { вычисляем новое положение }
    newy := y - Round(distance * Sin(angle));
    if pen = true then
        Form1.Screen.Canvas.LineTo(newx, newy)        { рисуем }
    else
        Form1.Screen.Canvas.MoveTo(newx, newy);       { или просто сдвигаемся }
    x := newx;
    y := newy;
end;

procedure TURN(a : Real);                          { повернуться на угол a }
begin
    angle := angle + a * PI / 180;          { используем формулу }
end;                                        { радианы = градусы * PI / 180 }

procedure PEN_UP;                           { поднять карандаш (не рисовать) }
begin
    pen := false;
end;

procedure PEN_DOWN;                         { опустить карандаш (рисовать) }
begin
    pen := true;
end;
```

Модель 11. «Конечный автомат»

Вот мы и добрались до последней модели в главе. Она тоже интересна в первую очередь своим психологическим аспектом, уже затронутым в «черепашьей графике», – взглядом на проблему под другим углом зрения. Рассмотрим, к примеру, такую простую задачу. В двухэтажном здании есть лифт, «умеющий» реагировать на команды: «открыть двери» (o), «закрыть двери» (c), «спуститься на первый этаж» (D), «подняться на второй этаж» (U). Изначально лифт находится на первом этаже, его двери открыты. Лифт может выполнить и «программу», например, такую: cUo, что означает «закрыть двери, подняться на второй этаж и открыть двери». Некоторые программы для лифта будут некорректными. К примеру, нельзя двигаться с открытыми дверями или подниматься на второй этаж, когда лифт и так находится на втором этаже. Задача заключается в том, чтобы смоделировать лифт на компьютере, то есть написать программу, которая будет уметь выполнять команды лифта (либо говорить, что команда некорректна), а также возвращать его текущее состояние (этаж и положение дверей).

В принципе, можно сразу попытаться сесть и написать соответствующие процедуры, но давайте попробуем сначала взглянуть на задачу по-другому. Лифт – это устройство, которое может иметь четыре различных состояния: «первый этаж, двери закрыты (cD)»; «первый этаж, двери открыты (oD)»; «второй этаж, двери закрыты (cU)» и «второй этаж, двери открыты (oU)». Любая команда переводит

лифт из одного состояния в другое: к примеру, если лифт был в состоянии cU, а на вход пришла команда D, то лифт перейдет в состояние cD. Все эти факты можно изобразить в виде графа (рис. 1.17).

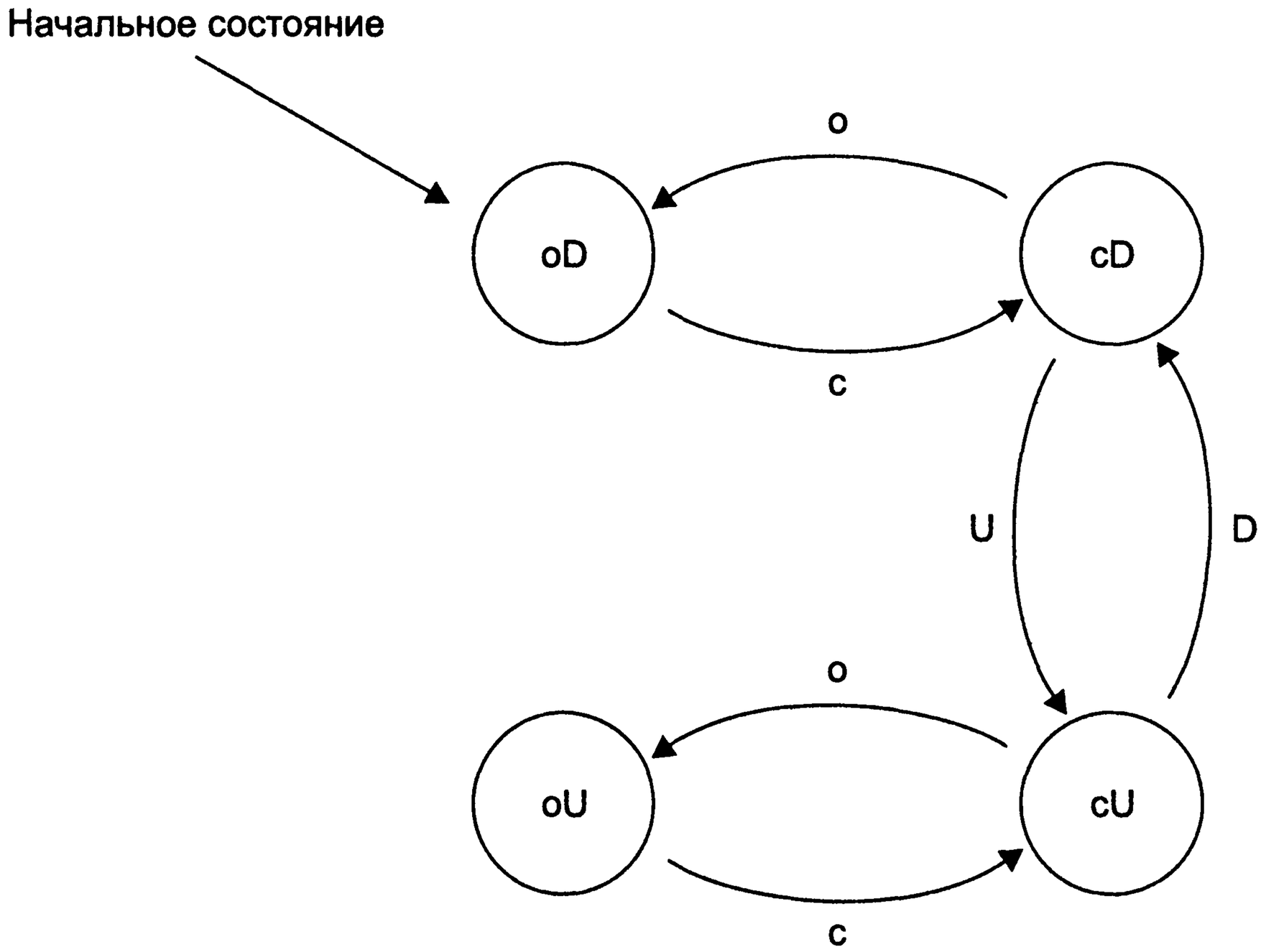


Рис. 1.17. Алгоритм управления лифтом — конечный автомат

Граф подобного вида и называется конечным автоматом[1]. Каждая вершина графа обозначает какое-либо состояние, а ребро – переход между двумя состояниями. Ребро обычно метят командой, по которой происходит переход. Глядя на граф, можно узнать и некорректные команды: например, из состояния oD нет ни одного перехода по команде U, что означает некорректность команды U в этом состоянии.

Прелесть такого автомата заключается в том, что его можно совершенно механически преобразовать в готовую программу, даже не задумываясь:

```
type StateType = (oD, cD, cU, oU);          { перечисляем возможные состояния }
type CommandType = (D, U, c, o, PRGEND);      { и команды }
var State   : StateType;
    Command : CommandType;
...
while true do
begin
    Command := GetNextCommand;          { получить следующую команду }
    if Command = PRGEND then Break;     { выход из цикла }
    case State of                       { для каждого состояния }
        oD: case Command of             { просматриваем список команд }
                c: State := cD;
            else                        { некорректная команда }
                Application.MessageBox('Ошибка!!!', '');
                Break;
```

[1] Конечно, определение более чем неформальное, но для наших целей сгодится.

```
        end;
    cD: case Command of          { аналогично для других состояний }
            o: State := oD;
            U: State := cU;
        else
            Application.MessageBox('Ошибка!!!', '');
            Break;
        end;
    cU: case Command of
            D: State := cD;
            o: State := oU;
        else
            Application.MessageBox('Ошибка!!!', '');
            Break;
        end;
    oU: case Command of
            c: State := cU;
        else
            Application.MessageBox('Ошибка!!!', '');
            Break;
        end;
    end;
end;
```

Приведенная программа действует прямолинейно: выбирает текущее состояние автомата, затем смотрит, можно ли выполнить очередную команду, и если да, переводит автомат в новое состояние. Таким образом, задача управления лифтом решена, да еще и без особых усилий. Психология автоматного программирования нередко оказывается очень полезной, экономя время и силы. Конечно, не стоит пытаться видеть в каждой задаче автоматы (так же, как и в каждой картинке — «черепашью графику»), но порою автомат сам собою напрашивается.

Вообще говоря, теория автоматов — большой и сложный раздел информатики, по которому пишут отдельные книги. Тем не менее, как сказал О. Хэвисайд, «Должен ли я отказываться от хорошего обеда лишь потому, что не понимаю процесса пищеварения?» Иными словами, то немногое, что вы почерпнули из модели «Конечный автомат», может быть полезным, так почему же не применять это на практике?

Проекты для самосовершенствования

1. Перепишите модель «Броуновское движение» так, чтобы на экране отображалась траектория движения частицы (проще всего нарисовать второй «аквариум» рядом с первым и рисовать в нем только траекторию).
2. Измените модель «Падающий шар» так, чтобы вместо движущегося шарика на экране вычерчивалась его траектория.
3. Дополните модель «Солнечная система» новыми планетами и их спутниками.
4. Модель «Борьба». Дополните модель «Жизнь» второй колонией инфузорий (пусть они имеют другой цвет). Для каждой колонии другая является невидимой; иными словами, клетка с инфузорией другого цвета всегда эквивалентна

пустой. Цикл моделирования будет состоять из двух фаз: сначала проходит один этап «Жизни» так, как будто бы существует только первая колония, затем выполняются те же действия для другой колонии. Пронаблюдайте за «взаимоотношениями» двух колоний.

5. Модель «Эволюция». Расширьте модель «Жизнь» новыми правилами и организмами. В «Эволюции» существуют организмы трех видов:
   - Протоплазма. Не совсем организм, а скорее «строительный материал». Если выбранная клетка пуста, на ней возникает протоплазма. Если на клетке уже есть протоплазма или другое существо, ничего не меняется. Если выбранная клетка является центром квадрата 3×3, восемь из девяти клеток которого заполнены протоплазмой, то в текущей клетке рождается инфузория, а вся протоплазма в квадрате исчезает.
   - Инфузории. «Живут» по правилам модели «Жизнь» (единственная тонкость: новая инфузория может родиться либо на пустой клетке, либо на клетке, содержащей протоплазму). Если три инфузории в какой-то момент времени образовали вертикальный или горизонтальный ряд, то они исчезают, а в центральной клетке ряда появляется животное.
   - Животные. Новорожденное животное имеет изначальный запас сил (5 единиц энергии). Если датчик случайных чисел указывает на клетку с животным, оно делает шаг на одну клетку в случайную сторону. Если клетка, в которую ступает животное, пуста, энергетический запас животного уменьшается на единицу. Если там находится протоплазма, животное съедает протоплазму, восстанавливая запасы энергии (то есть энергия не меняется). Если клетка содержит инфузорию, животное ее поедает, увеличивая при этом запас энергии на единицу. Если же клетка содержит другое животное, они «объединяются», образуя лишь одно животное, энергетический запас которого равен сумме запасов обоих столкнувшихся животных. Если энергетический запас животного истощается, оно погибает.

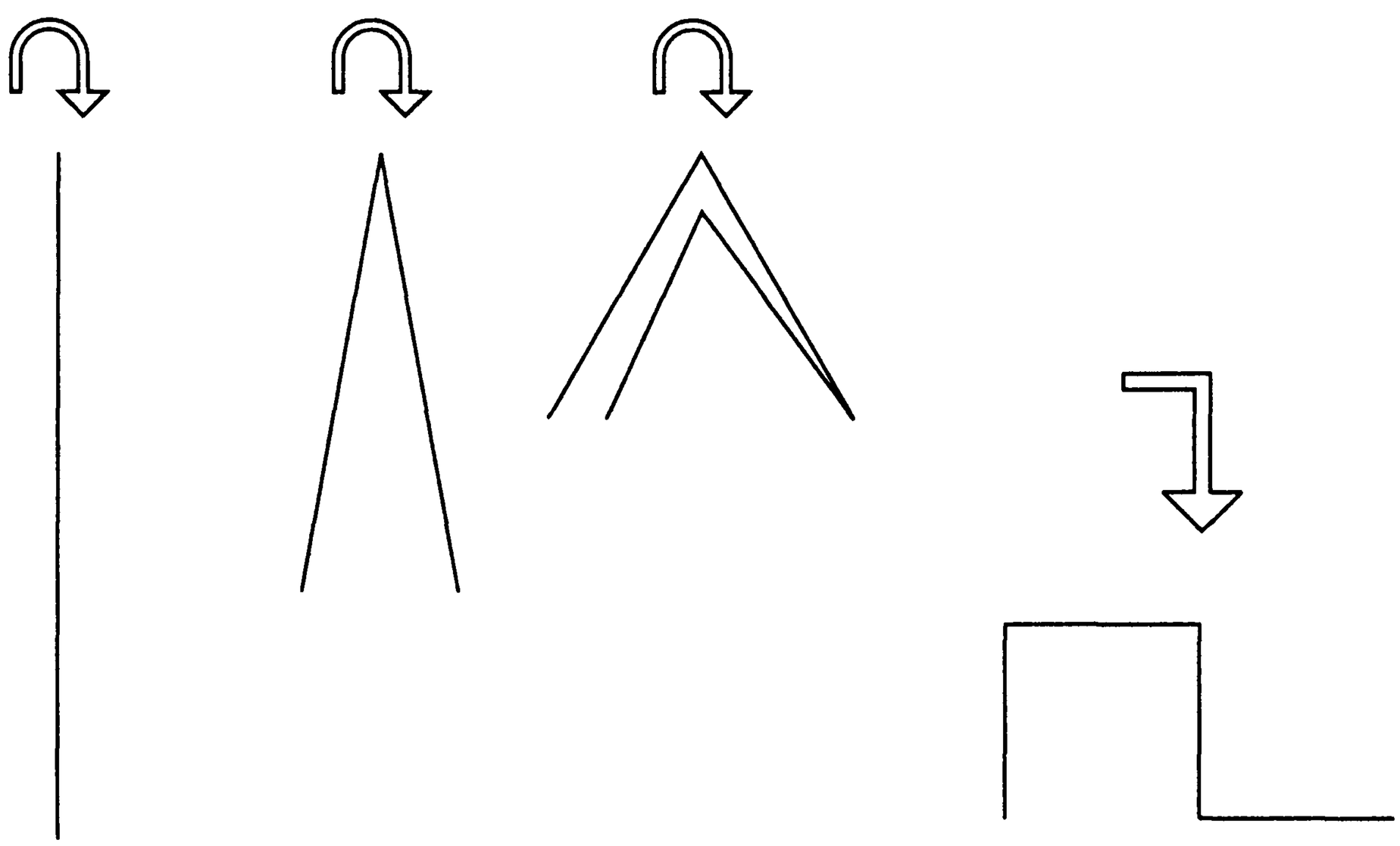


Рис. 1.18. Драконова ломаная третьего порядка

6. В теории поле для модели «Жизнь» Джона Конуэя имеет бесконечные размеры. Измените реализацию модели так, чтобы поле было замкнуто[1]: если «планер» вылетает за правую границу экрана, он появляется слева, если за верхнюю — снизу, и наоборот.
7. Если согнуть полоску бумаги пополам K раз (надо только следить, чтобы сгиб всегда производился в одну и ту же сторону), а потом разогнуть так, чтобы каждый угол сгиба был равен 90 градусов, то получится так называемая драконова ломаная K-го порядка (рис. 1.18). Напишите программу, которая строит драконову ломаную произвольного порядка. Используйте «черепашью графику» для вывода результата на экран.
8. В Science Park'е города Joensuu (это в Финляндии) установлены автоматы, которые продают кофе. Автомат устроен следующим образом. В прорезь можно вставить любое количество монет любого достоинства (5, 10, 20, 50 центов; 1 или 2 евро). После этого следует выбрать тип кофе (с сахаром, без сахара, капучино или вообще какао, а не кофе) и нажать на кнопку «налить со стаканом» или «налить без стакана». «Со стаканом» означает, что автомат сам подставит картонный стаканчик; «без стакана» — просто выльет порцию напитка, наивно рассчитывая на то, что вы сами подставите свой стакан. Любой вариант кофе стоит 70 центов, поэтому автомат отсчитает сдачу, если в монетоприемник была опущена сумма, превышающая цену[2]. Опишите алгоритм работы этого агрегата при помощи концепции конечного автомата и реализуйте его на каком-нибудь языке программирования.

---

[1] Реально такое поле будет иметь форму тора (то есть бублика).

[2] Честно говоря, это еще как повезет. Но не будем о грустном.

Глава 2

Анимация и графические эффекты

Кроме того, создание видеоигр и нас самих делает счастливыми.
Я не думаю, что написание компилятора может сделать
кого-нибудь счастливым человеком.

Андрэ Ла Мот

Эпиграф к этой главе я взял из прекрасной книги Андрэ Ла Мота и Стива Ратклиффа «Секреты программирования игр». Я бы посоветовал ее прочесть, но, к сожалению, на русском языке издана эта книга была еще в 1995 году, и найти ее теперь очень трудно. Хотя книга, которую вы держите в руках, имеет весьма далекое отношение к созданию игр, в ней все-таки есть некоторые главы, близкие по духу этому занятию. «Анимация и графические эффекты» – как раз пример такой главы. Конечно, можно написать хитроумную программу, которая считывает с клавиатуры строчку текста, а затем выводит на печать в качестве результата своей работы другую строчку (полученную путем сложных и, скажем, новаторских манипуляций) и получить от этого несказанное удовольствие; тем не менее я думаю, что таких людей немного. Кроме того, мало кто по достоинству оценит ваш талант как изобретателя нового алгоритма сортировки; стоит же только написать красивый и завораживающий скринсейвер, так тут же любому станет ясно, что вы – Очень Одаренный Программист. Я не говорю, что изобретать новые алгоритмы сортировки неинтересно (это не так), но порою всем нам хочется написать что-то такое, что бы радовало глаз (или у вас нет любви к прекрасному?), да еще позволило бы с гордостью продемонстрировать «это» не только другу-соавтору (...нового алгоритма сортировки), а кому-нибудь еще. Опять-таки, есть, должно быть, люди, никогда не восхищавшиеся при взгляде на экран компьютера, да еще и равнодушные к мнениям других... по крайней мере, я к ним не отношусь. А за то недолгое время, когда я на досуге преподавал программирование, выяснилось, что люди гораздо охотнее изучают графику и анимацию, чем, скажем, алгоритмы архивации (удивительно, правда?).

В современном «индустриальном» программировании обычно уже не ставится вопрос о том, как осуществить движение графических объектов или запрограммировать поворот трехмерной фигуры. Есть готовые стандартные библиотеки (DirectX и OpenGL, к примеру), есть библиотеки для удобной работы с этими

библиотеками (пожму руку тому несчастному, кто взялся писать программу на «чистом» DirectX – в добрый путь), есть даже готовые специализированные среды разработки (в частности, для разработки игр предназначены DIV Games Studio и Dark Basic) и т. д. Скорее всего, вам придется иметь дело хотя бы с некоторыми из этих вещей; тем не менее интересно и полезно «заглянуть за кулисы», узнать, как это происходит «на самом деле», а не просто вызвать готовую функцию. К тому же (и это более важно) существуют концепции и приемы, которые не зависят ни от конкретной библиотеки, ни даже от операционной системы. В этой главе я постараюсь рассказать хотя бы о некоторых из них. Не забывайте, что в «настоящих» программах все графические операции сильно оптимизируются (или используются хорошие графические библиотеки), поэтому наши процедуры сгодятся лишь для учебных целей – их производительность, скорее всего, будет слишком низка для применения на практике; однако главное – идеи – останется неизменным.

Движение объектов

Вернемся к самой первой программе в этой книге, в которой использовалось движение по экрану, – к модели «Молекула газа в закрытом сосуде». Увеличьте радиус молекулы (допустим, до 40 пикселов) и уменьшите ее скорость (5 пикселов за итерацию). Кроме этого, измените код так, чтобы «молекула» рисовалась закрашенной каким-нибудь цветом (для этого надо лишь вставить строчку `Screen.Canvas.Brush.Color := ЦветЗакраски` в соответствующих местах).

Теперь запустите программу. Мерцает? Скорее всего, да (честно говоря, зависит во многом от реализации компонента `TPaintBox`; по крайней мере, на Delphi 7 мерцает). Прежде чем идти дальше, необходимо научиться бороться с этим неприятным эффектом. А для того, чтобы с ним бороться, познакомимся с двумя важными понятиями – вертикальной разверткой и двойной буферизацией.

Вертикальная развертка и двойная буферизация

Традиция предписывает начинать разговор о *вертикальной развертке* с небольшой справки об устройстве монитора, а точнее о том, каким образом происходит вывод изображения на экран.

Итак, изнутри экран монитора покрыт специальным веществом, люминофором, которое освещается электронным лучом (потоком частиц, исходящих из электронной пушки). Свойства люминофора, каждый элемент которого состоит из трех точек разного цвета – красного, зеленого и синего, таковы, что он изменяет свою окраску и яркость при бомбардировке электронами, да еще и сохраняет этот цвет в течение некоторого времени, даже если луч на него уже не направлен. На этом и основана идея прорисовки экрана: луч последовательно пробегает по всем точкам дисплея, «зажигая» каждую из них по отдельности. Происходит это так быстро, что люминофор в первой зажженной точке не успевает «забыть» свой цвет даже к тому моменту, когда луч находится уже в самом дальнем от нее углу монитора. Физически луч начинает свое путешествие с левого верхнего угла

экрана, прорисовывает самую верхнюю строку трехточечных пикселов, затем переходит ко второй строке и т. д. Заканчивается маршрут в правом нижнем углу, а затем луч начинает свой путь сначала. Длительность прорисовки всего экрана определяется настройками видеоадаптера, а конкретнее, частотой обновления экрана. Типичные значения этого параметра — 75, 85, 100 или 120 герц[1] (при этом длительность кадра от начала до начала прорисовки равна соответственно 1/75, 1/85, 1/100, 1/120 секунды).

Представьте теперь, что на экране нарисован большой закрашенный круг (размером в целый экран), и вы даете компьютеру команду его стереть. Компьютер бы-стренько обновляет видеопамять, но экран моментально обновиться не может (помните про луч?). Если луч в момент выполнения команды «стереть» находился где-нибудь посередине экрана, он затрет лишь нижнюю часть круга. Верхняя же часть будет стерта лишь на следующем цикле «путешествия» луча, поскольку реально круг не может исчезнуть, пока луч не доберется до каждой из его точек. Таким образом, если частота обновления экрана, к примеру, равна 85 герцам, в течение 1/85 секунды вы будете наблюдать на экране половинку круга, скорее всего, совсем не запланированную. Аналогичные «эффекты» можно заметить и во время рисования объектов.

Избавиться от мерцания не так уж и сложно: для этого просто надо все манипуляции с экраном производить в то мгновение, когда луч находится в правом нижнем углу экрана и ничего не перерисовывает. Когда же начнется следующий цикл перерисовки, на экране появится следующий (полностью готовый) кадр анимации.

Задача «отлова» требуемой позиции луча уже решена практически во всех анимационно-ориентированных графических библиотеках: от программиста обычно требуется лишь установить какой-нибудь «флажок синхронизации», и все операции ввода-вывода будут происходить в соответствии с идеей, о которой я говорил выше. Мы не будем подробно останавливаться на этой задаче, поскольку ее решение требует использование низкоуровневых операций чтения из порта, а Windows сделать этого так просто не позволит[2]. Если вы желаете поэкспериментировать под DOS (в DOS-окне Windows этот код будет прекрасно работать), прочтите следующий абзац, если же нет — смело его пропускайте.

Для определения позиции луча служит третий бит (самый правый бит — нулевой, самый левый — седьмой) порта 3DAh. Если этот бит установлен, значит, луч сейчас ничего не рисует, а если сброшен — идет формирование изображения. Таким образом, если мы хотим отловить момент перехода из состояния рисования в пассивное состояние, достаточно вызвать процедуру, содержащую команды:

```
ПОКА (бит 3 установлен, пассивное состояние)
    ничего не делать
ПОКА (бит 3 сброшен, активное состояние)
    ничего не делать
```

---

[1] Чем больше частота, тем безопаснее для глаз. К сожалению, пока мониторы с высокой частотой развертки все еще довольно дороги. К примеру, частота развертки на моем мониторе равна сейчас 85 герцам. Принцип действия жидкокристаллических мониторов иной, но суть проблемы остается.

[2] Придется писать, например, так называемый VxD-драйвер и вообще заниматься не самыми творческими вещами.

Как только выполнение фрагмента завершится, у вас будет гарантированный отрезок времени (правда, совсем небольшой), в течение которого можно смело производить манипуляции с графикой. Напомню, что прочитать данные из порта можно при помощи специального массива Port (инструкция x := Port[$3DA]) в Паскале или функции inportb() (x = inportb(0x3DA)) в C/C++. Чтобы узнать значение третьего бита, на полученную величину придется еще наложить по AND двоичное число 00000100 (то есть восемь). Если результат операции не равен нулю, бит установлен, иначе – сброшен.

К сожалению, вертикальной разверткой проблемы не ограничиваются. Обычно той самой одной восемьдесят пятой (или тем более одной сотой) секунды не хватает для того, чтобы перерисовать весь экран. Решается эта проблема тоже достаточно просто: создадим невидимый «виртуальный экран» и будем рисовать только на нем. В период же, отведенный для рисования на основном экране (то есть в момент нахождения луча в правом нижнем углу), просто скопируем содержимое экрана виртуального на экран основной. Операция блокового копирования обычно производится очень быстро, даже сотой секунды должно хватить для ее выполнения. Скажу больше: любая приличная библиотека (DirectX, в частности) позволяет переназначить указатель видеопамяти на любой виртуальный экран, то есть попросту указать, что отныне такой-то виртуальный экран является основным. При этом никакого копирования вообще не происходит, и работает подобный механизм очень быстро. С другой стороны, мы избавляемся от необходимости что-то стирать и рисовать на новой позиции: надо лишь нарисовать «с нуля» очередной кадр и скопировать поверх предыдущего на основной экран.

Описанная техника называется *двойной буферизацией* (double buffering). Поскольку при использовании двойной буферизации каждый кадр накладывается на предыдущий (обычно мало чем от нового отличающийся), нередко удается получить плавную анимацию, не заботясь о вертикальной развертке. Во многих играх есть даже возможность включить или отключить синхронизацию с вертикальной разверткой – порою отличить эти режимы «на глаз» очень трудно[1]. Впрочем, синхронизация, как правило, все равно благотворно влияет на качество анимации.

Давайте перепишем «Молекулу газа в закрытом сосуде» с использованием техники двойной буферизации. На роль буфера вполне подойдет объект типа TImage. Его надо поместить на форму, растянуть до размера основного экрана (объекта Screen) и сделать невидимым (установить свойство Visible в False). Назовите его BackScreen. Теперь измените программу так, чтобы вместо основного экрана молекула рисовалась на буфере BackScreen. Вместо кода, стирающего молекулу с экрана, можно вставить строки, очищающие буфер целиком:

```
BackScreen.Canvas.Pen.Color := clBtnFace;
BackScreen.Canvas.Brush.Color := clBtnFace;
BackScreen.Canvas.FillRect(Rect(0, 0, BackScreen.Width, BackScreen.Height));
```

Перед строкой же Sleep(10) необходимо вставить инструкцию, копирующую содержимое буфера на основной экран:

---

[1] Именно поэтому я решил, что не стоит лезть в детали процедуры, синхронизирующей анимацию с вертикальной развёрткой под Windows, — и без нее качество наших программ будет приемлемым, хотя и не идеальным.

```
Screen.Canvas.CopyRect(Rect(0, 0, Screen.Width, Screen.Height),
                       BackScreen.Canvas, Rect(0, 0, Screen.Width, Screen.Height));
```

Итак, проблема мерцания решена.

Синхронизация с таймером

Простите, наболело. Сколько раз приходилось встречать программы (в основном это касается игр), которые прекрасно работают на «типичных» машинах того времени, когда они были разработаны, но сходят с ума, начиная передвигать объекты по экрану с фантастической скоростью на современных, более мощных компьютерах! И это касается не только чего-то жутко древнего, но и достаточно новых приложений. Не буду называть конкретных имен и стыдить разработчиков, которые к 1999–2000 годам так и не поняли одной простой и очевидной вещи: величина любых задержек, задаваемых в программе, **не должна** зависеть от аппаратной конфигурации компьютера. Если вы пишете, что для работы вашей программы достаточно процессора Pentium II, то это не должно означать, что лишь на втором пентиуме она и будет работать хорошо. Программа с такими требованиями обязана идти не хуже и на третьем, и на четвертом пентиумах, и даже на тех процессорах, которые еще не созданы. Разве так трудно сделать паузу не при помощи пустого цикла от единицы до миллиона, а вызовом задержки в N миллисекунд? Увы, подобные глупости происходили раньше и происходят до сих пор. Даже уважаемые фирмы порою упускают их из виду. Вы, наверное, знаете о знаменитой проблеме в модуле CRT компилятора Borland Pascal. Для инициализации модуля (точнее, процедуры Delay) любая программа, написанная на Паскале, пытается определить производительность вашего процессора. Делает она это просто: в течение 55 миллисекунд выполняет пустой цикл, а потом делит количество полученных итераций на 55 (определяя тем самым, сколько итераций компьютер способен выполнить за миллисекунду). Результат записывается в простую целую (16 бит) переменную. На мощных компьютерах (начиная примерно с Pentium II 266 MHz) количество итераций так велико, что попытка записать его в целую переменную приводит к переполнению, и вы получаете Runtime error 200. Сейчас эту проблему обычно решают, вырезая дающий сбой код прямо из готового EXE-файла при помощи специальной программы; но суть ведь в том, что проблемы вообще не должно было быть!

Надеюсь, я убедил вас в том, что есть над чем поработать. Вернемся к нашим задачам.

Скорость работы всех программ, которые мы рассматривали в первой главе, вообще говоря, зависит от аппаратуры. Лишь простейшие из них (например, «Молекула в закрытом сосуде») зависят от нее не так сильно, поскольку время, затраченное на рисование, значительно меньше паузы между итерациями. А на длительность паузы (то есть на время выполнения инструкции Sleep(N)) скорость процессора никак не влияет.

Давайте попробуем избавиться от этих вредных зависимостей. Тот способ, который я здесь опишу, наверное, не является самым простым, зато он универсален и достаточно надежен.

Возьмем опять в качестве примера «Молекулу в закрытом сосуде» (теперь уже модифицированную версию с двойной буферизацией). Для начала надо опреде-

лить *реальную* скорость молекулы, которую мы хотим получить. Эта скорость не должна зависеть от конкретной аппаратуры, поэтому «пикселы за итерацию» уже не годятся; более удачный выбор — «пикселы в секунду». Общая идея проста: допустим, скорость объекта равна V пикселов в секунду; тогда если одна итерация цикла выполняется K секунд, на каждой итерации объект надо сдвигать на V * K пикселов. На псевдокоде это выглядит так:

```
oldtime := текущее_время
ЦИКЛ
    dx := Vx * (текущее_время - oldtime)    { разница времен измеряется }
    dy := Vy * (текущее_время - oldtime)    { в секундах }
    oldtime := текущее_время
    X := X + dx
    Y := Y + dy
    обновить содержимое экрана
КОНЕЦ ЦИКЛА
```

На первый взгляд кажется, что и в этом алгоритме есть «часовая бомба», аналогичная той, что обнаружилась в компиляторе Borland Pascal. Если компьютер очень быстр, то время выполнения одной итерации может быть равно машинному нулю, следовательно, объекты вообще не будут двигаться. На самом деле это может быть справедливо лишь для программ, не использующих синхронизацию с вертикальной разверткой. Если же синхронизация производится, время выполнения каждой итерации будет не меньше периода обновления экрана (1/75... 1/120 секунды). С другой стороны, можно просто вставить в цикл небольшую явную задержку — Sleep(1), к примеру, — и забыть об этой проблеме. Основной недостаток алгоритма, на мой взгляд, связан с тем, что теперь нам приходится иметь дело с вещественными числами взамен целых. И координаты, и смещения должны быть вещественными, а операции с ними, как известно, выполняются гораздо медленнее. В принципе, даже расходы на них вряд ли будут существенными по сравнению с остальными (если, конечно, ваша программа делает хоть что-то осмысленное, кроме прямолинейного движения объектов по экрану). Писать дополнительный код, который вычисляет смещения, конечно, тоже не самое радостное занятие. Зато есть у приведенного алгоритма и очень приятные свойства. Например, на слишком медленных компьютерах программа не будет «тормозить» — скорость ее работы останется неизменной; просто движения станут менее плавными. Кроме того, вы можете применять этот подход в любых программах, независимо от их специфических особенностей. Конечно, универсальный алгоритм не так прост, как некоторые частные случаи. К примеру, если вы заведомо знаете, что частота развертки монитора во время работы программы будет равна 85 герц, а сама программа точно может генерировать больше 85 кадров в секунду, то синхронизация становится тривиальной: просто генерируем очередной кадр, ждем сигнала о том, что луч находится в правом нижнем углу монитора, и выводим кадр на экран.

Изменения в реальной программе достаточно прямолинейны. Во-первых, придется поменять тип всех целых переменных на Real и добавить операции округления в строку с вызовом Ellipse() (иначе программа не откомпилируется):

```
BackScreen.Canvas.Ellipse(Round(X) - R, Round(Y) - R,
                          Round(X) + R, Round(Y) + R);
```

Во-вторых, нам потребуется новая переменная oldtime типа TDateTime и две переменные dx и dy типа Real. В соответствии с алгоритмом перед началом цикла следует инициализировать переменную oldtime:

```
oldtime := Now;   { текущее время }
```

Первыми же строками в главном цикле (опять-таки, в соответствии с алгоритмом) будут инструкции

```
dx := Vx * (Now - oldtime) * SecsPerDay;
dy := Vy * (Now - oldtime) * SecsPerDay;
oldtime := Now;
```

Поскольку Delphi вычисляет разницу времен в сутках, а не в секундах, ее необходимо умножить на предопределенную константу SecsPerDay (количество секунд в сутках). Вычисление новых координат молекулы теперь будет производиться при помощи строк

```
X := X + dx;
Y := Y + dy;
```

Задержку же в конце главного цикла вообще нужно убрать — или заменить, как уже сказано, на Sleep(1). Вот и все.

Простые спрайты

Спрайтами обычно называют движущиеся объекты, не затирающие фоновый рисунок во время своего движения. Надеюсь, понятно, о чем идет речь: герои компьютерных игр, помощники в Microsoft Office и даже обычный указатель мыши в Windows являются спрайтами. До этого мы имели дело с простыми графическими объектами, которые можно нарисовать прямо в программе («молекулы», к примеру). Естественно, это не показательный случай; обычно выводимые на экран изображения не вычерчиваются в программе, а берутся из готовых файлов. Ни для кого не секрет, что в файле можно хранить только прямоугольные рисунки[1]; реально же выводимые на экран объекты — это именно «объекты» произвольной формы, и фон должен затираться лишь в тех местах, где выводятся точки собственно изображения, а не прямоугольника, в который это изображение вписано.

Чтобы правильно вывести спрайт на экран, компьютер должен каким-то образом догадаться, какие точки картинки являются «существенными» (то есть составляющими картинки), а какие нет. Обычно для этого используют понятие *прозрачного цвета*. Выберем какой-нибудь цвет, который не встречается в нашей картинке (обычно берут какой-нибудь ядовито-розовый, грязно-зеленый и тому подобные малоприятные цвета), и будем рисовать спрайт на прямоугольнике, полностью закрашенном этим цветом. После этого надо указать, что выбранный цвет считается прозрачным — и дело сделано! Любая уважающая себя графическая библиотека (VCL в том числе) поддерживает концепцию прозрачного цвета. Реализовано на практике это очень просто: когда функция вывода попиксельно прорисовывает картинку, она все время проверяет цвет текущего пиксела, и если он оказывается прозрачным, то функция не рисует его вообще. Таким образом, в областях, помеченных прозрачным цветом, остается фоновый рисунок.

[1] Разумеется, речь идет о файлах с *растровыми* изображениями (BMP, PNG, PCX, ...).

Давайте заменим «молекулу» в нашей программе настоящим спрайтом, чтобы познакомиться с этой техникой на практике. Для начала нам потребуется фоновый рисунок (чтобы убедиться, что спрайт может передвигаться, не затирая его) и собственно изображение спрайта. В архиве с исходными текстами программ к этой главе[1] есть готовые картинки, но вы можете взять любые другие. Главное, чтобы размер фонового изображения совпадал с размером рабочей области, а спрайт был нарисован на однотонной поверхности (цвет которой мы сможем указать в качестве прозрачного). Картинки должны быть сохранены в формате BMP (к сожалению, в библиотеке VCL использование других форматов ограничено; не забывайте, однако, что на практике следует обязательно хранить изображения в сжатом виде — винчестеры все-таки не резиновые).

Добавьте на главную форму приложения элемент TImage, который будет хранить фоновое изображение. Назовите элемент Background и загрузите в него заранее подготовленную фоновую картинку (вам поможет свойство Picture, отображаемое в Инспекторе объектов). Добавьте еще один элемент TImage под именем Sprite и загрузите в него изображение спрайта. Размеры этого элемента должны быть в точности равны размерам спрайта. Чтобы добиться этого, проще всего установить значение свойства AutoSize равным True. Теперь задайте True в качестве значения свойства Transparent, чтобы включить режим прозрачного цвета. По умолчанию Delphi будет считать прозрачным цвет, который имеет левый нижний пиксел картинки. Для большинства спрайтов это вполне подходит (для того, который я выбрал в качестве примера, — тоже подходит), поэтому не будем ничего менять. Не забудьте сделать обе картинки невидимыми — они должны появляться на экране лишь тогда, когда мы сами попросим их об этом.

Работа со спрайтами требует лишь двух новых операций: копирования фонового рисунка на экран и копирования самого спрайта. Первая из этих операций производится при помощи вызова CopyRect():

```
BackScreen.Canvas.CopyRect(Rect(0, 0, Screen.Width, Screen.Height),
                           Background.Canvas, Rect(0, 0, Screen.Width, Screen.Height));
```

Вторая — вызовом Draw():

```
BackScreen.Canvas.Draw(X, Y, Sprite.Picture.Bitmap);
```

С методом CopyRect() мы уже встречались, когда копировали буфер на основной экран (кстати, не забывайте, что мы по-прежнему используем двойную буферизацию). Метод Draw() рисует переданную в качестве параметра картинку (объект типа TBitmap, который можно «достать» из TImage через свойство Picture, как указано в примере) так, чтобы ее левый угол находился в точке (X, Y), не забывая при этом о прозрачном цвете.

Основной цикл программы изменится несильно:

```
ЦИКЛ
    сдвинуть спрайт
    скопировать на виртуальный экран фоновую картинку
    скопировать на виртуальный экран спрайт
    скопировать виртуальный экран на основной
КОНЕЦ ЦИКЛА
```

[1] Напомню, его можно взять на сайте издательства «Питер» www.piter.com.

Вероятно, многие сочтут расточительным копировать фоновое изображение на виртуальный экран целиком на каждой итерации — ведь реально требуется восстановить лишь тот его кусочек, который был затерт спрайтом на предыдущем шаге. В данном случае это действительно справедливо; однако, как правило, каждый следующий кадр из предыдущего так просто не получить, и приходится перерисовывать весь фон заново.

Думаю, на сей раз стоит привести листинг программы целиком (я убрал старые комментарии, чтобы не мешали замечать изменения) (листинг 2.1).

Листинг 2.1. Спрайт в закрытом сосуде

```
procedure TForm1.StartStopBtnClick(Sender: TObject);
    var X, Y   : Real;
        Vx, Vy : Real;
        angle  : Real;
        oldtime : TDateTime;
        dx, dy : Real;
    const
        V = 200;
begin
    if IsRunning then
    begin
        IsRunning := false;
        StartStopBtn.Caption := 'Пуск';
        Exit;
end;

StartStopBtn.Caption := 'Стоп';
IsRunning := true;

Randomize;
{ на сей раз X и Y - координаты верхнего левого
  угла спрайта, поэтому диапазон изменился }
X := RandomRange(0, BackScreen.Width - Sprite.Width);
Y := RandomRange(0, BackScreen.Height - Sprite.Height);
angle := Random(360)* Pi /180;

Vx := Round(V * Sin(angle));
Vy := Round(V * Cos(angle));

oldtime := Now;          { текущее время }
while IsRunning do
begin
    dx := Vx * (Now - oldtime) * SecsPerDay;          { вычисляем смещения }
    dy := Vy * (Now - oldtime) * SecsPerDay;          { спрайта }
    oldtime := Now;

    X := X + dx;
    Y := Y + dy;

    if X > BackScreen.Width - Sprite.Width then
        begin  X := BackScreen.Width - Sprite.Width; Vx := - Vx; end;
    if X < 0 then
```

```
    begin  X := 0; Vx := - Vx; end;
if Y > BackScreen.Height - Sprite.Height then
    begin  Y := BackScreen.Height - Sprite.Height; Vy := - Vy; end;
if Y < 0 then
    begin  Y := 0; Vy := - Vy; end;

{ генерация и вывод очередного кадра }
BackScreen.Canvas.CopyRect(Rect(0, 0, Screen.Width, Screen.Height),
          Background.Canvas, Rect(0, 0, Screen.Width, Screen.Height));
BackScreen.Canvas.Draw(Round(X), Round(Y), Sprite.Picture.Bitmap);
Screen.Canvas.CopyRect(Rect(0, 0, Screen.Width, Screen.Height),
                    BackScreen.Canvas, Rect(0, 0, Screen.Width, Screen.Height));

        Application.ProcessMessages;
        Sleep(1);
    end;
end;
```

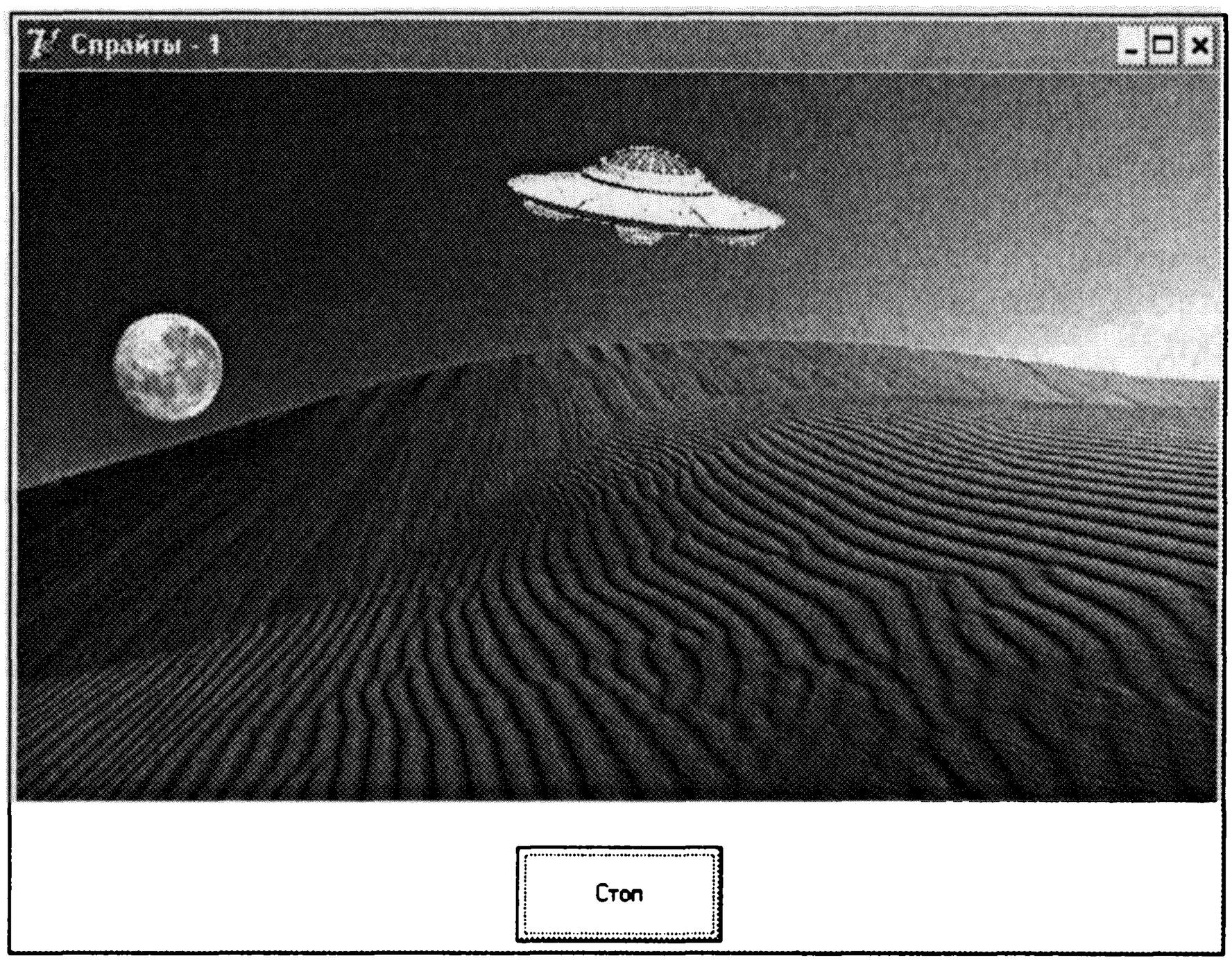


Рис. 2.1. Внешний вид приложения

Многокадровые спрайты

Разобравшись с базовыми приемами, можно заняться оживлением спрайтов. Мы рассмотрим простую задачу: допустим, спрайт состоит из нескольких кадров, и во время работы программы эти кадры должны последовательно сменять друг друга (самый последний кадр меняется на первый — то есть смена происходит циклически). Это довольно обычная ситуация — именно таким образом осуществляют движение фигурок людей и животных, велосипедов (у которых изменяется положение спиц) и даже летающих тарелок. Перед тем как читать дальше, подберите подходящий многокадровый спрайт (его можно взять в архиве, соот-

ветствующем этой главе, – изображение летающей тарелки, 24 кадра). Под «многокадровым» я подразумеваю просто какое-то количество отдельных рисунков (одинакового размера, естественно), которые служат разными кадрами одного и того же спрайта.

Алгоритм движения анимированного спрайта на псевдокоде выглядит примерно так:

```
спрайт := первый кадр
ЦИКЛ
    сдвинуть спрайт
    скопировать на виртуальный экран фоновую картинку
    спрайт := очередной кадр
    скопировать на виртуальный экран спрайт
    скопировать виртуальный экран на основной
КОНЕЦ ЦИКЛА
```

На первый взгляд кажется, что изменений почти нет: надо лишь вовремя изменять текущий кадр, и все. На самом же деле по сравнению с предыдущей программой появились две проблемы, которые надо решить. Первая (простая): как хранить кадры в памяти? Вторая (посложнее): как в произвольный момент времени определить, какой именно кадр использовать сейчас?

Для решения первой проблемы в Delphi существует очень хороший компонент, который называется TImageList. Он позволяет хранить в памяти коллекцию из любого числа картинок одинакового размера и легко доставать из коллекции любую картинку по ее порядковому номеру. К вопросу о том, как это конкретно сделать, я еще вернусь, а пока займемся второй проблемой.

Первое, что приходит в голову, – менять кадр на следующий во время каждой итерации цикла (ну или во время каждой десятой, двадцатой итерации). Несомненно, это самое простое решение, которое только можно придумать. Вот только делать так нельзя ни в коем случае – в этом таится зло! Помните о синхронизации с таймером? Если мы допустим зависимость скорости смены кадров спрайта от быстроты выполнения одной итерации, то программа сразу же начнет вести себя по-разному на компьютерах, работающих с разной скоростью[1]. Правильное решение заключается в том, чтобы «прикрутить» к спрайтам процедуру синхронизации, аналогичную той, что мы уже использовали. Предположим, что спрайт содержит N кадров, частота смены – Vc кадров в секунду, а показания таймера в начале работы программы находятся в переменной starttime. Тогда в любой момент времени номер текущего кадра можно вычислить по формуле:

```
FrameNo = Round((Now - starttime) * SecsPerDay * Vc) mod N
```

Часть формулы слева от mod эквивалентна той, что мы уже использовали. Разница в том, что номер кадра должен изменяться от нуля до N - 1, а потом снова становиться равным нулю. Проще всего этого добиться, взяв остаток от деления на N, который и будет принимать требуемые значения.

---

[1] Как просто создавать программы для игровых консолей (вроде старой доброй Dendy или Sony Playstation) или некогда популярных компьютеров ZX Spectum или MSX Yamaha! Никогда не приходится думать о том, что у пользователя будет другое разрешение экрана, другая видеокарточка или другая скорость процессора...

Итак, чтобы оживить спрайт, прежде всего добавьте на главную форму элемент типа TImageList. Назовите его FrameList. Обязательно укажите в качестве размеров элемента реальные размеры спрайта — иначе при создании коллекции картинок возникнут проблемы. Щелкните на элементе правой кнопкой мыши и выберите в контекстном меню пункт ImageList Editor. В открывшемся окне нажмите кнопку Add и добавьте в коллекцию все кадры. Сама программа изменится несильно. Потребуется новая константа Vc (сколько кадров сменять в секунду) и переменная starttime, которой в начале программы надо присвоить значение Now. Блок же, осуществляющий рисование на экране, теперь будет выглядеть так:

```
BackScreen.Canvas CopyRect(Rect(0, 0, Screen.Width, Screen.Height),
                  Background.Canvas, Rect(0, 0, Screen.Width, Screen.Height));
Sprite Canvas FillRect(Rect(0, 0, Sprite.Width, Sprite.Height));
FrameList GetBitmap(Round((Now - starttime) * SecsPerDay * Vc) mod
                                        FrameList.Count, Sprite Picture.Bitmap);
BackScreen.Canvas.Draw(Round(X), Round(Y), Sprite.Picture.Bitmap);
Screen.Canvas.CopyRect(Rect(0, 0, Screen.Width, Screen.Height),
                  BackScreen.Canvas, Rect(0, 0, Screen.Width, Screen.Height));
```

Как и прежде, спрайт выводится на экран командой BackScreen.Canvas.Draw(), но перед этим в объект Sprite копируется соответствующий кадр. Вызов FrameList.GetBitmap(index, bmp) копирует картинку с номером index из коллекции на переданный объект типа TBitmap. Перед копированием спрайт надо очистить при помощи FillRect(), иначе по каким-то странным причинам происходит наложение кадров друг на друга.

Скроллинг

Под под скроллингом обычно понимается прокрутка фонового изображения в процессе работы программы. Редкая аркадная игра, выполненная в двух измерениях, обходится без этого процесса (рис. 2.2).

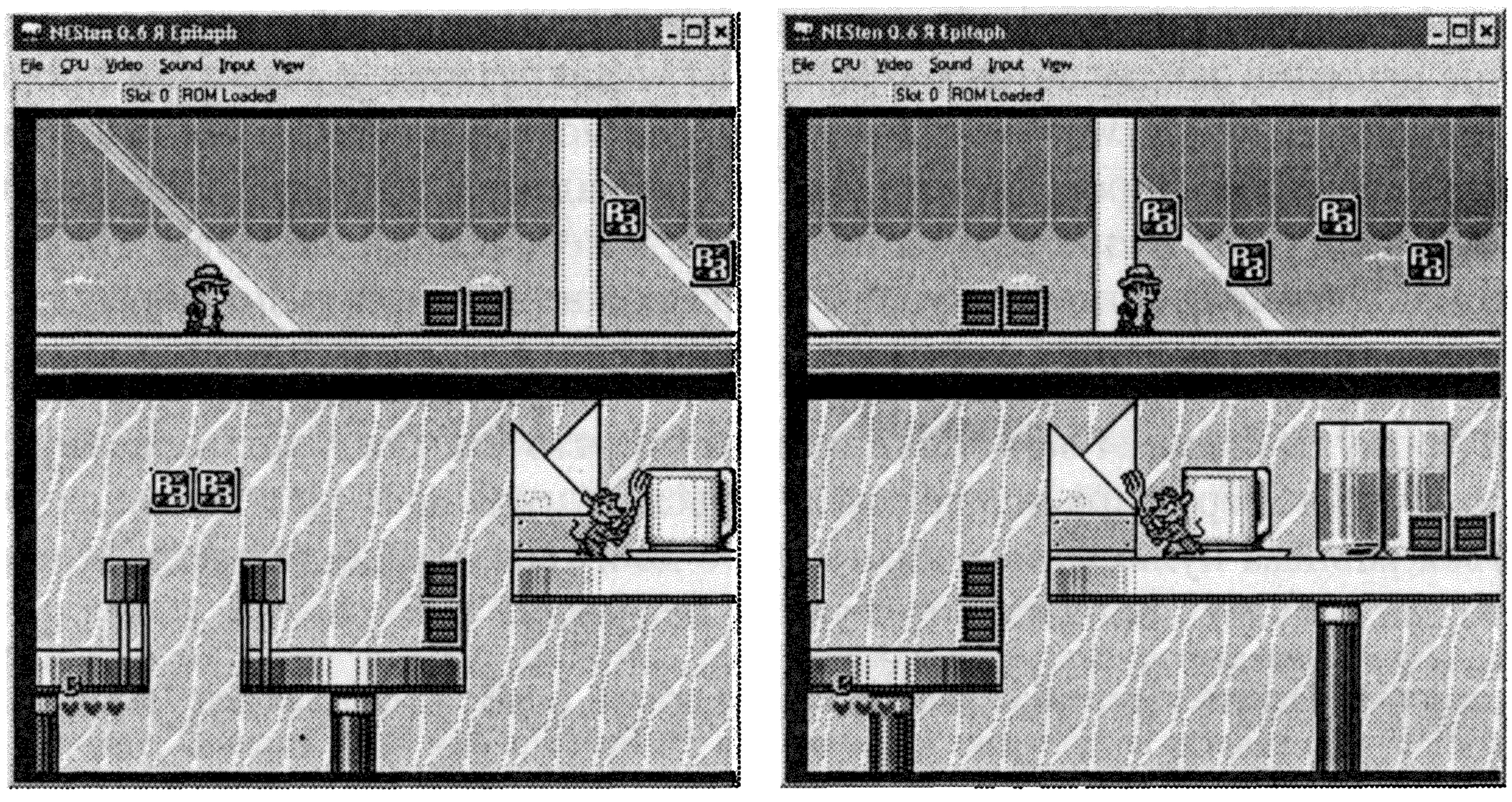


Рис. 2.2. Скроллинг (справа налево) в игре Chip 'N Dale 2

В скроллинге, в принципе, нет ничего особенно хитрого, но я решил все-таки уделить ему немного времени, учитывая то, как часто он встречается на практике.

В играх, подобных Chip 'N Dale 2, фоновая картинка обычно генерируется из «кусочков» прямо во время работы программы, но нередко фон просто загружается из готового графического файла. Часто встречается на практике и так называемый «циклический» скроллинг — когда изображение прокручивается по кругу; та его часть, которая скрылась за границей экрана, появляется с другой его стороны.

Предположим, что готовый фон хранится в виде графического файла. Во-первых, не так уж и важно, загружается ли он с диска или генерируется в памяти «на лету» из кусочков; во-вторых, это удобно для нас, поскольку в предыдущем примере мы уже использовали фоновое изображение и можем снова использовать уже написанный код сейчас[1]. Вся процедура скроллинга сводится к тому, чтобы на каждой итерации главного цикла выводить некоторый участок фоновой картинки на экран. Этот «некоторый участок» имеет размеры экрана; единственное, что потребуется определить — это смещение начала участка внутри фоновой картинки, значение `Xbg` (рис. 2.3).

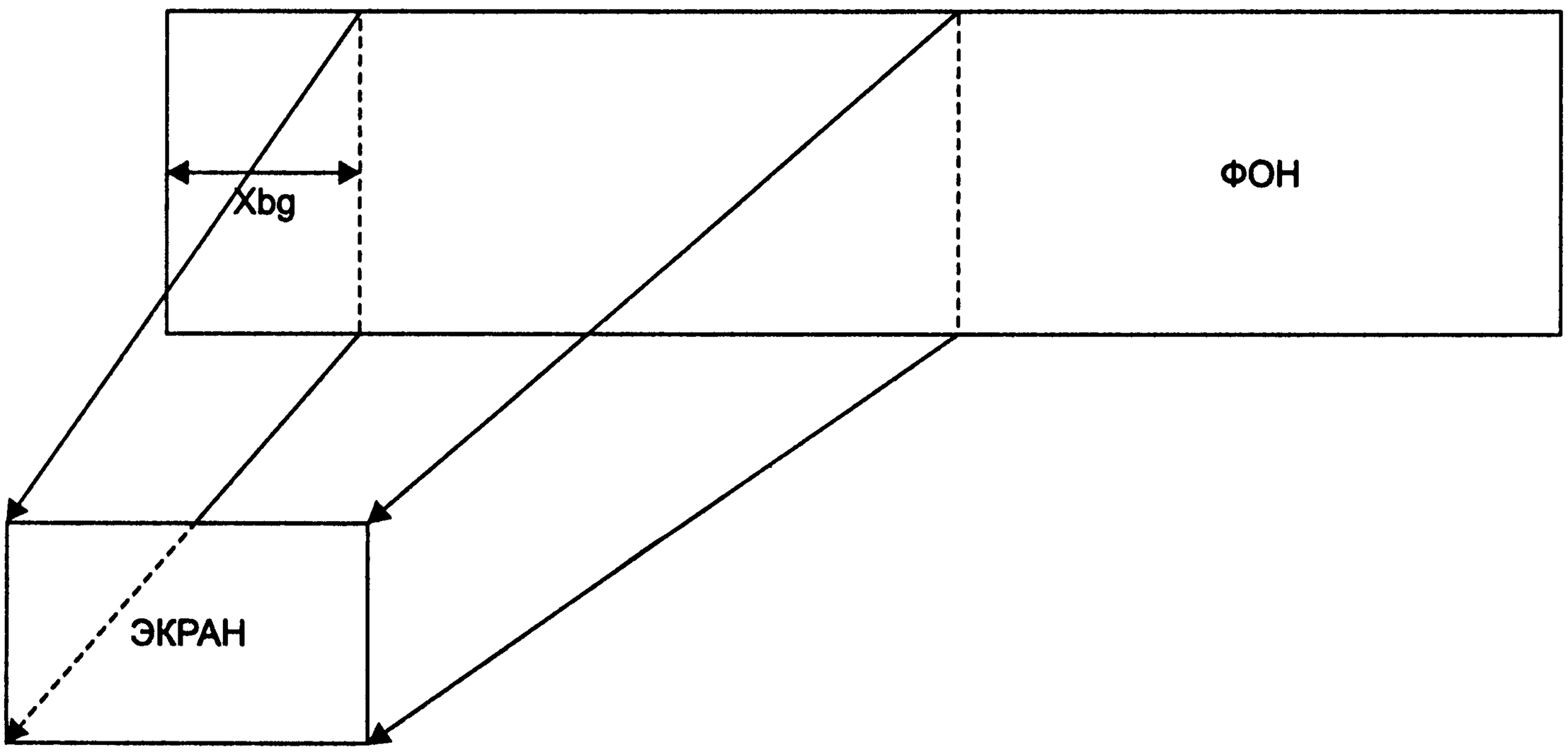


Рис. 2.3. Схема скроллинга

Если ограничиться равномерным движением фона (что мы и сделаем), то значение `Xbg` просто будет постоянно расти. Опять-таки, очень заманчиво увеличивать `Xbg` на некоторую константу на каждой итерации, но я, как и раньше, буду настаивать на точной синхронизации с часами системы (пусть даже мое упорство кому-то покажется фанатичным). Если значение `Xbg` уже определено, остается лишь скопировать соответствующий участок фоновой картинки на экран (естественно, для начала на виртуальный экран) при помощи вызова[2]

```
BackScreen.Canvas.CopyRect(Rect(0, 0, Screen.Width, Screen.Height),
          Background.Canvas, Rect(Xbg, 0, Screen.Width + Xbg, Screen.Height));
```

---

[1] В реальной программе я сделал его немного шире — если фон имеет те же размеры, что и экран, это изображение просто некуда прокручивать!

[2] На самом деле значение Xbg будет храниться в переменной типа Real, поэтому его придется еще округлить.

Вычисление смещения Xbg происходит совершенно аналогично вычислению позиции спрайта в предыдущем примере:

```
{ вычисляем смещение на данной итерации }
ds := direction * Скорость_скроллинга * (Now - oldtime) * SecsPerDay;
{ вычисляем новое "глобальное" смещение }
ЕСЛИ фон прокручивается справа налево
    Xbg := Xbg + ds;
ИНАЧЕ (слева направо)
    Xbg := Xbg - ds;
```

В качестве конкретного примера дополним предыдущую программу прокруткой фона. Сначала фоновый рисунок прокручивается справа налево до конца, потом слева направо и так до бесконечности.

Для начала потребуются новые переменные: ds, Xbg (вещественные) и direction (целая). Переменная direction будет использоваться для хранения текущего направления скроллинга. Добавим также описание константы Vs — скорости скроллинга.

Перед началом главного цикла переменные следует инициализировать так:

```
Xbg := 0;                { нулевое смещение }
direction := 1;          { направление - справа налево }
```

Внутри цикла вычисляем новое значение Xbg:

```
ds := direction * Vs * (Now - oldtime) * SecsPerDay;
Xbg := Xbg + ds;
```

Если фоновое изображение прокрутилось целиком, направление меняется на противоположное (делается это так же, как и «отскок» молекулы от стенки сосуда):

```
if Xbg > Background.Picture.Width - Screen.Width then
begin
    Xbg := Background.Picture.Width - Screen.Width;
    direction := -direction;
end;
if Xbg < 0 then
begin
    Xbg := 0;
    direction := -direction;
end;
```

Добавляем еще одну строку — вывод участка фоновой картинки на экран:

```
BackScreen.Canvas.CopyRect(Rect(0, 0, Screen.Width, Screen.Height),
                           Background.Canvas, Rect(Round(Xbg), 0,
                           Screen.Width + Round(Xbg), Screen.Height));
```

На этом изменения заканчиваются.

Графические эффекты

В этом разделе описываются три простых и очень популярных графических эффекта: «затухание» (fading), красивая смена фонового изображения и составле-

ние картинки из отдельных точек. Эффекты программировать проще с точки зрения синхронизации с таймером. Чем больше кадров за секунду может сгенерировать программа, тем лучше, при условии, что вы управляете происходящим на экране. Движения становятся более плавными, а объекты – более управляемыми. Другое дело – графический эффект, выполнение которого контролируется только компьютером. Здесь, как правило, достаточно добиться постоянной скорости работы (скажем, 30 кадров в секунду) и остановиться на этом.

«Затухание»

«Затухание» – один из самых давно известных и популярных эффектов, который вы, скорее всего, не раз видели: за пару секунд фоновое изображение становится все менее и менее контрастным, пока целиком не превращается в однотонный прямоугольник. Его реализацией мы сейчас и займемся.

Все, что нам нужно для начала, – это фоновое изображение (которое будет затухать) и «целевой цвет» (фоновое изображение в конце работы программы превратится в прямоугольник, целиком закрашенный этим цветом). Идея эффекта очень проста: на каждой итерации главного цикла надо просто «приближать» каждый пиксел фона к целевому цвету. Поскольку цвет в компьютере хранится в виде трех чисел – интенсивности красной, зеленой и синей составляющих, «приблизить» один цвет к другому – значит уменьшить разность между соответствующими составляющими. К примеру, если цвет текущего пиксела равен (15, 130, 43), а целевым цветом является черный (0, 0, 0), для «приближения» цвета на очередной итерации следует просто уменьшить значение каждой его составляющей; при этом получится тройка (14, 129, 42). Если бы целевым цветом был белый (255, 255, 255), то нам, наоборот, потребовалось бы увеличивать значения интенсивностей.

Поскольку для хранения каждой составляющей отводится один байт, ее значение лежит в пределах от нуля до 255. Отсюда вывод: алгоритм «затухания» требует не более 255 итераций, после чего результат будет заведомо достигнут.

Итак, весь эффект сводится к тому, чтобы уменьшать, увеличивать или оставлять без изменения значение каждой составляющей исходного цвета на очередной итерации главного цикла, пока исходный цвет не превратится в целевой.

В программе нам впервые придется работать с отдельными пикселами картинки. Для доступа к ним служит свойство Pixels[X, Y] класса TCanvas. При считывании свойства возвращается цвет (объект типа TColor) пиксела, находящегося на позиции (X, Y). При записи, соответственно, цвет пиксела меняется. Из полученного цвета нам потребуется определить значения его RGB-составляющих[1]. Для этого служат функции GetRValue(цвет), GetGValue(цвет) и GetBValue(цвет).

Теперь пару слов о синхронизации с таймером. Как я уже говорил, ограничимся простой фиксацией количества генерируемых в секунду кадров. Для этого прежде всего надо задать где-нибудь константу, отвечающую за эту величину:

```
MSecsPerFrame = 25;        { количество миллисекунд на один кадр }
```

[1] То есть Red-Green-Blue — интенсивности красного, зеленого и синего каналов.

Тело главного цикла будет выглядеть так:

```
oldtime := Now;
{ генерация кадра }
pause := Round(MSecsPerFrame - (Now - oldtime) * MSecsPerDay);
if pause > 0 then Sleep(pause);
```

Идея проста: мы определяем, сколько времени съела генерация очередного кадра, вычитаем это количество из общего отведенного на это действие времени и тем самым определяем величину задержки до начала генерации следующего кадра. Если компьютер не успел сгенерировать кадр за выделенный временной интервал, задержка окажется отрицательной (тогда, естественно, процедуру Sleep() вызывать не надо). Подобный подход гарантирует, что скорость работы программы не будет выше заданной (вызов Sleep() затормозит любой самый быстрый компьютер), однако на медленных компьютерах эффект будет выполняться медленнее. Мне кажется, что это не проблема: во-первых, эффект – это лишь эффект, и скорость его выполнения не так уж важна (в разумных пределах, естественно); во-вторых, всегда можно установить нижний предел аппаратных требований, для которых все будет работать замечательно (а если кто-то запускает программу на более слабых машинах, мы ничего не гарантируем); в-третьих, все эффекты, которые мы тут рассматриваем, не так уж «тяжелы» для аппаратуры – они будут прекрасно работать и на достаточно медленных компьютерах.

Текст программы приведен в листинге 2.2. Предполагается, что фоновое изображение загружено в объект Background типа TImage. О том, как выглядит эффект, можно судить по рис. 2.4.

Листинг 2.2. «Затухание»

```
procedure TForm1.StartStopBtnClick(Sender: TObject);
    var X, Y    : Integer;                    { координаты текущей точки }
        R, G, B : Integer;                    { RGB-компоненты ее цвета }
        dR, dG, dB : Integer;                 { RGB-компоненты целевого цвета }
        TheEnd : Boolean;                     { признак завершения работы }
        c : TColor;                           { временная переменная }
        oldtime : TDateTime;
        pause   : Integer;
    const
        DestColor : TColor = clWhite;         { целевой цвет }
        MSecsPerFrame = 25;                   { скорость работы }

begin
    if IsRunning then
    begin
        IsRunning := false;
        StartStopBtn.Caption := 'Пуск';
        Exit;
    end;

    StartStopBtn.Caption := 'Стоп';
    IsRunning := true;

    dR := GetRValue(DestColor);               { определяем RGB-компоненты }
    dG := GetGValue(DestColor);               { целевого цвета }
```

```
dB := GetBValue(DestColor);

while IsRunning do
begin
   oldtime := Now;
   TheEnd := true;
    for X := 0 to Background.Width - 1 do          { цикл по всем   }
        for Y := 0 to Background.Height - 1 do     { пикселам рисунка }
        begin
            c := Background.Canvas.Pixels[X, Y]; { текущий пиксел }
            if c <> DestColor then                { если остались пикселы, }
              TheEnd := false;                    { не равные DestColor }

        R := GetRValue(c);
        G := GetGValue(c);
        B := GetBValue(c);

        if R > dR then R := R - 1;      { "приближаем" цвет текущего }
        if R < dR then R := R + 1;      { пиксела к DestColor }
        if G > dG then G := G - 1;
        if G < dG then G := G + 1;
        if B > dB then B := B - 1;
        if B < dB then B := B + 1;
                                        { изменяем цвет пиксела }
        Background.Canvas.Pixels[X, Y] := TColor(RGB(R, G, B));
end;

        if TheEnd then Form1.StartStopBtnClick(nil);   { выход из цикла }

        { копируем рисунок на основной экран }
        Screen.Canvas.CopyRect(Rect(0, 0, Background.Width,
                               Background.Height), Background.Canvas,
                               Rect(0, 0, Background.Width, Background.Height));
        Application.ProcessMessages;
        pause := Round(MSecsPerFrame - (Now - oldtime) * MSecsPerDay);
        if pause > 0 then Sleep(pause);
    end;
end;
```

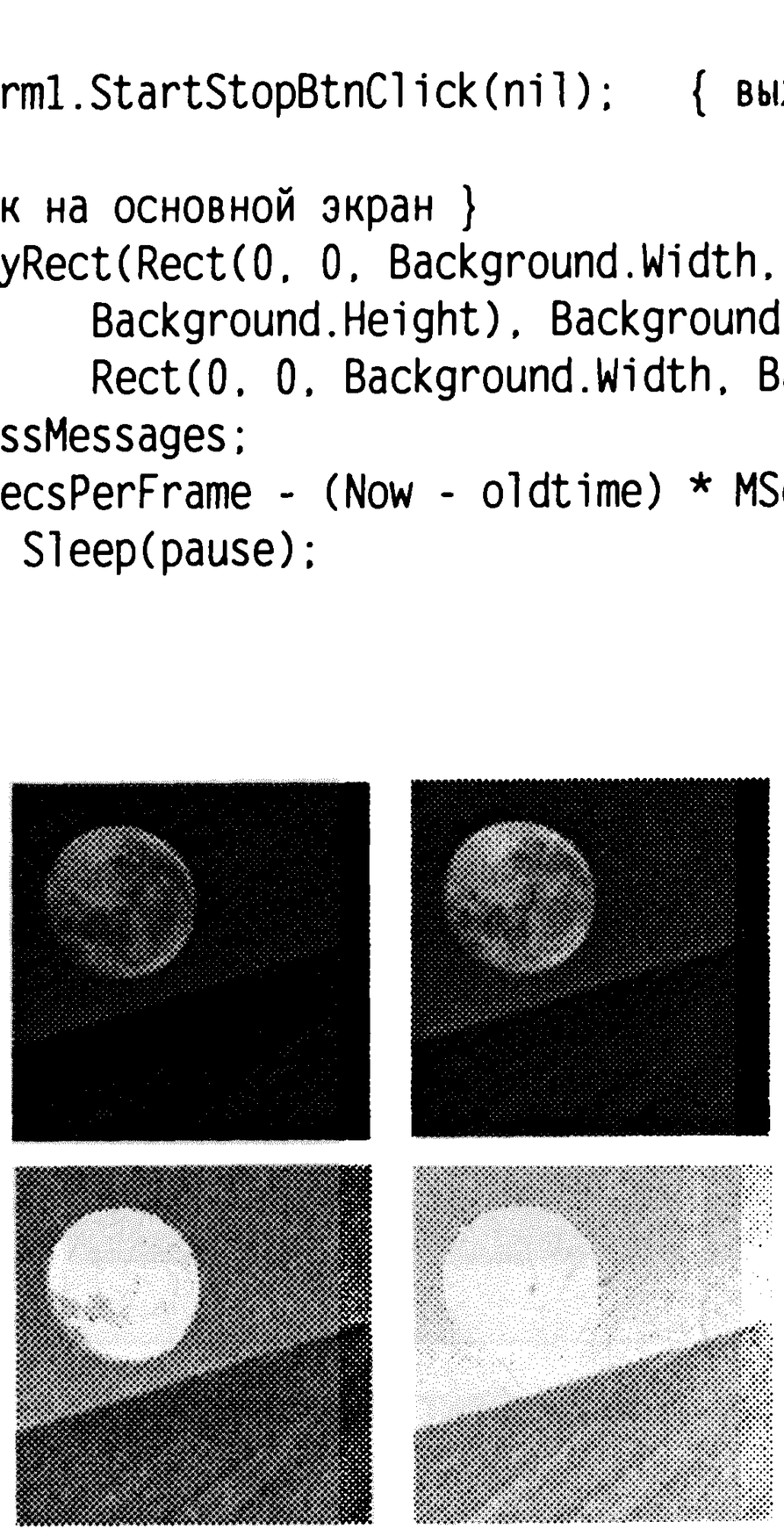


Рис. 2.4. Эффект «затухания»

Перед тем как перейти к следующему эффекту, я должен вас предостеречь: используйте для экспериментов с «затуханием» небольшие картинки (максимум 100 × 100 точек). К сожалению, и без того медленная в отношении графики библиотека VCL показывает совершенно «впечатляющие» по скорости результаты при работе со свойством Pixels.

Если вы перепишете все программы главы под DirectX (что сделать не так сложно), производительность, словно по мановению волшебной палочки, увеличится в десятки раз. Я все-таки решил использовать именно VCL, потому что не хочу вдаваться в технические подробности DirectX (поскольку библиотека низкоуровневая, в ней этих подробностей предостаточно). Использовать же удобную «прослойку» (wrapper) тоже не хочется – кто-то предпочитает CDX, кто-то NDX, кто-то DelphiX, а кто-то «самый лучший в мире wrapper», название которого мне вообще ни о чем не скажет. VCL же знают все; кроме того, она действительно удобна в использовании и позволяет не «зацикливаться» на деталях. Иными словами, для учебных целей просто находка. Кстати, даже переписав полностью графический ввод-вывод на DirectX, все равно можно с успехом хранить изображения в объектах типа TBitmap, а спрайты – в TImageList. Свои задачи эти компоненты выполняют прекрасно и при этом достаточно быстро (впрочем, никаких сверхскоростей от них и не требуется).

Красивая смена фона

Суть этого эффекта лучше всего понятна из рис. 2.5. Фоновое изображение «расползается» по сторонам (нечетные полоски ползут влево, четные – вправо), открывая тем самым новый фон.

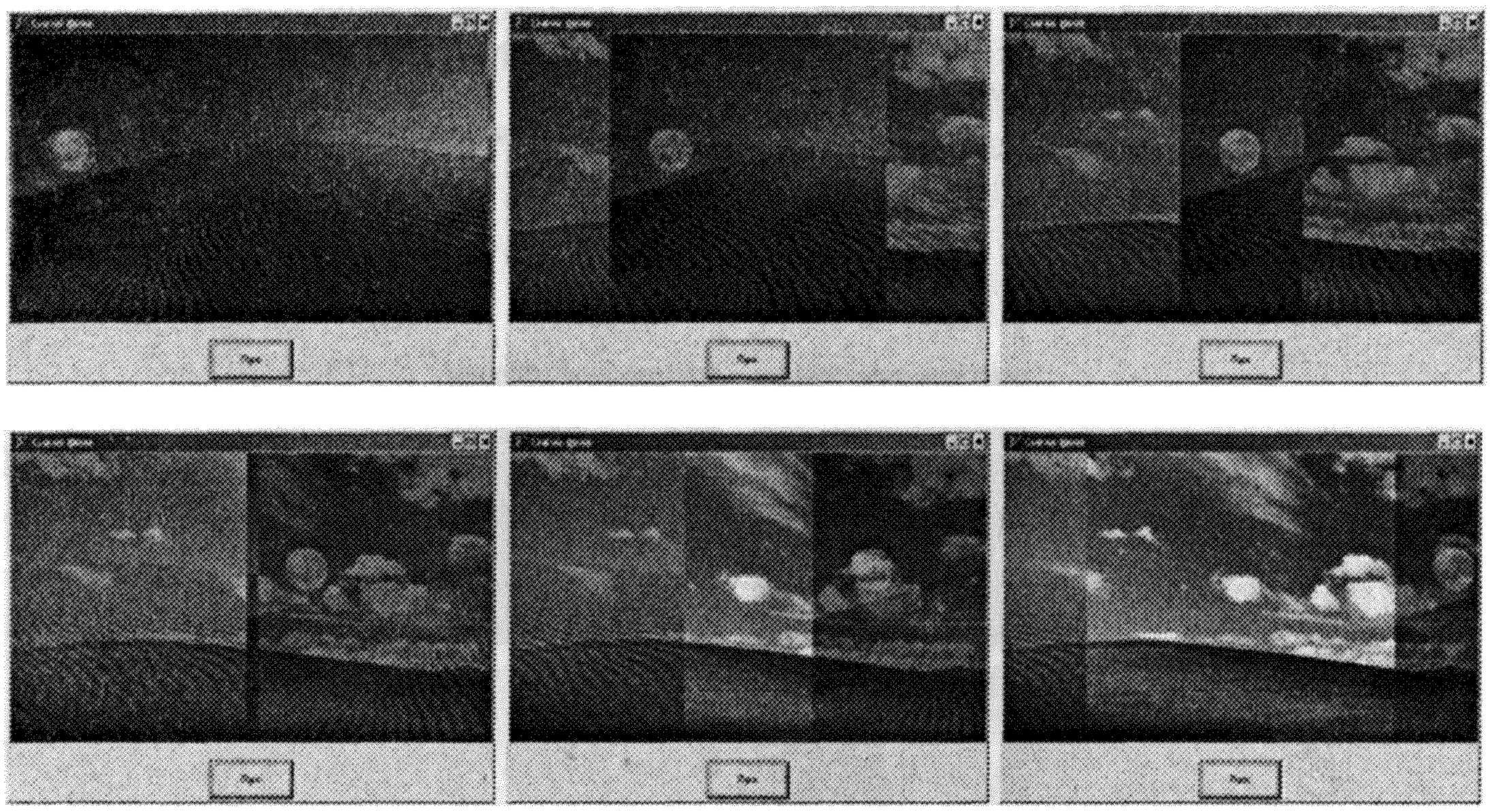


Рис. 2.5. Эффект красивой смены фона

Программа, реализующая этот эффект, достаточно прямолинейна. Есть две фоновые картинки: текущая и та, которая должна появиться в процессе работы

приложения. Есть также некоторое «смещение» d, которое изначально равно нулю, а во время каждой итерации цикла увеличивается на единицу. В процессе формирования кадра на экран сначала накладывается «новая» фоновая картинка, а затем — особым образом фрагменты строк изначального фона. Надеюсь, дело немного прояснит рис. 2.6.

Участок первой строки фона [d...Width] накладывается на участок [0...Width - d] первой строки экрана; для второй строки эти диапазоны равны [0...Width - d] и [d...Width] соответственно, для третьей они такие же, как и для первой и т. д.

Если значение смещения d увеличилось до размера ширины экрана, это значит, что старая фоновая картинка полностью исчезла; эффект закончен.

На самом деле размер реальной программы (листинг 2.3) даже короче моих объяснений принципа ее работы. Предполагается, что оба фоновых изображения имеют те же размеры, что и экран; первое из них (старое) находится в объекте Background, второе — в Background2 (оба объекта имеют тип TImage, как и раньше). Синхронизация с таймером делается аналогично предыдущей программе.

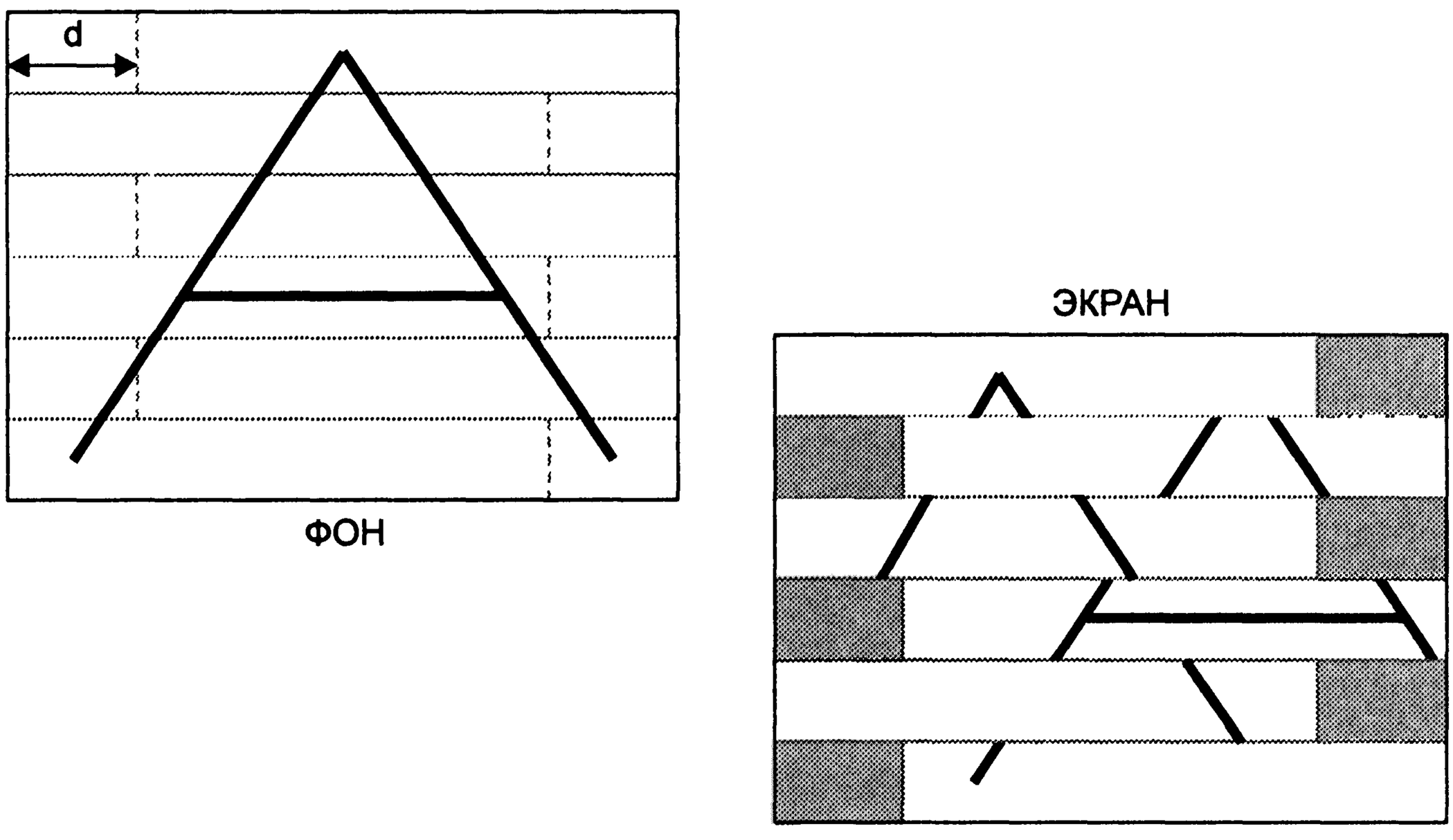


Рис. 2.6. Наложение участков фонового рисунка на экран

Листинг 2.3. Красивая смена фона

```
procedure TForm1.StartStopBtnClick(Sender: TObject);
    var i       : Integer;
        d       : Integer;
        oldtime : TDateTime;
        pause   : Integer;
    const
        MSecsPerFrame = 15;
begin
    if IsRunning then
    begin
        IsRunning := false;
        StartStopBtn.Caption := 'Пуск';
```

продолжение

Листинг 2.3 (продолжение)

```
        Exit;
    end;

    StartStopBtn.Caption := 'Стоп';
    IsRunning := true;
    d := 0;

    while IsRunning do
    begin
        oldtime := Now;
    { копируем новый фон }
    BackScreen.Canvas.CopyRect(Rect(0, 0, Screen.Width, Screen.Height),
            Background2.Canvas, Rect(0, 0, Screen.Width, Screen.Height));
    for i := 0 to Screen.Height - 1 do
    begin
        if i mod 2 = 0 then  { четная строка }
            BackScreen.Canvas.CopyRect(Rect(0, i, Screen.Width - d,
              i + 1), Background.Canvas, Rect(d, i, Screen.Width, i + 1))
        else                    { нечетная строка }
            BackScreen.Canvas.CopyRect(Rect(d, i, Screen.Width, i + 1),
              Background.Canvas, Rect(0, i, Screen.Width - d, i + 1));
    end;

    { копируем виртуальный экран на основной }
    Screen.Canvas.CopyRect(Rect(0, 0, Screen.Width, Screen.Height),
                    BackScreen.Canvas, Rect(0, 0, Screen.Width, Screen.Height));
            d := d + 1;
            if d > Screen.Width then Form1.StartStopBtnClick(nil);     { конец }
            Application.ProcessMessages;
            pause := Round(MSecsPerFrame - (Now - oldtime) * MSecsPerDay);
            if pause > 0 then Sleep(pause);          { синхронизация с таймером }
        end;
    end;
```

Составление картинки из точек

Вот мы и дошли до последнего (и, на мой взгляд, самого красивого) из описываемых здесь эффектов. Представьте себе, что по всему экрану разбросаны разноцветные точки. Они приходят в движение, причем каждая точка стремится к своей цели, к своему месту на экране. В итоге, когда все они заняли свои позиции, мы видим, что из этого хаоса точек возникло вполне осмысленное изображение (рис. 2.7).

Как ни странно, запрограммировать этот эффект гораздо проще, чем может показаться на первый взгляд.

Предположим, картинка, которую мы хотим составить из точек, находится в объекте `Background` типа `TImage`. Допустим также, что в результате эта картинка должна появиться точно в центре экрана. Это означает, что координаты ее левого верхнего угла (`x0`, `y0`) можно определить по формулам:

```
x0 := (Screen.Width - Background.Width) div 2
y0 := (Screen.Height - Background.Height) div 2
```

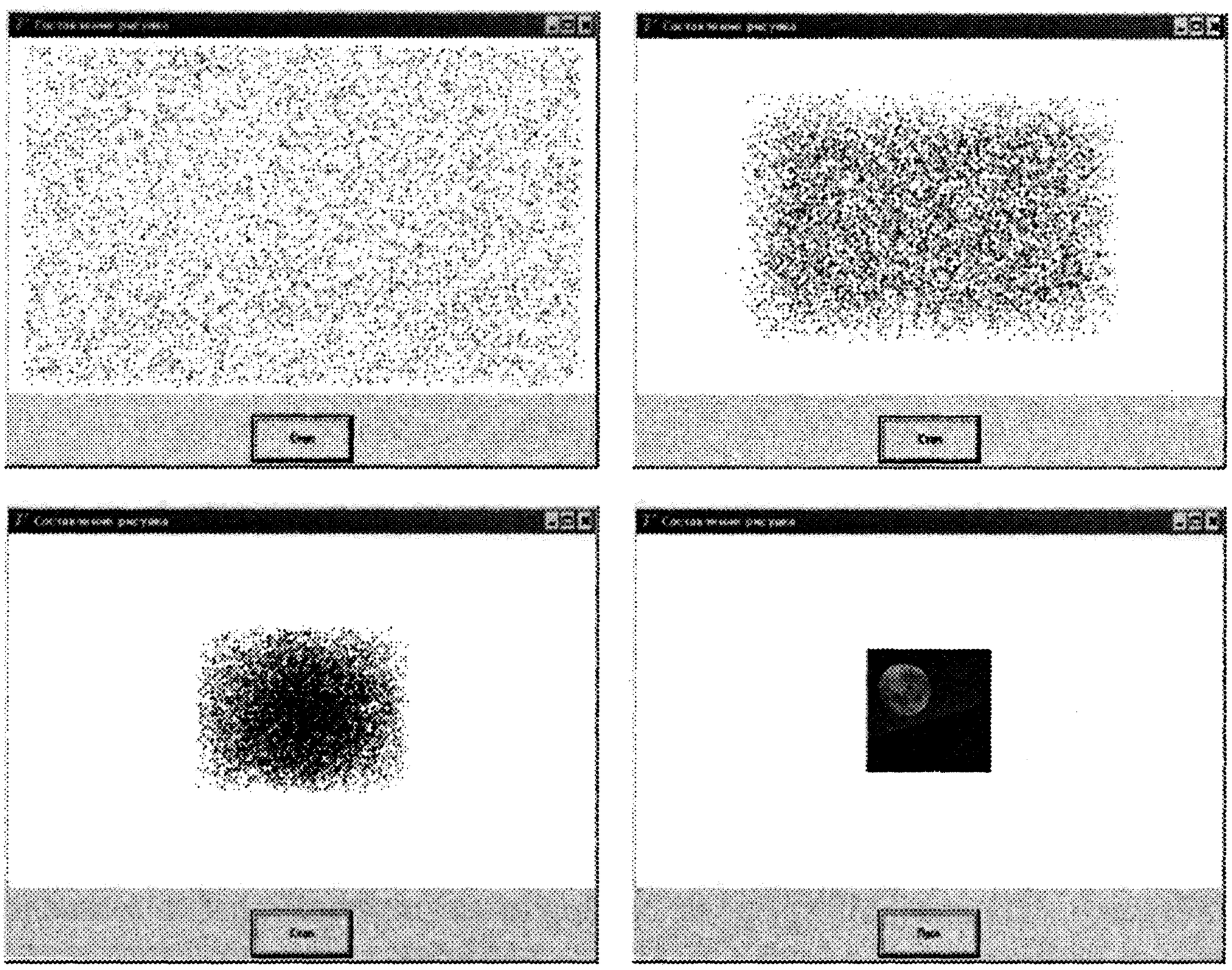


Рис. 2.7. Эффект в действии

Теперь представьте, что все пикселы рисунка разбросаны по случайным (но известным) позициям на экране. Пусть пиксел, который находился на позиции (i, j) в исходной картинке (то есть Background.Canvas.Pixels[i, j]), теперь оказался в точке (x, y) экрана. Договоримся также, что весь эффект должен выполниться за некоторое заранее определенное количество итераций главного цикла. Итак, мы знаем, что пиксел, который находится на экране в точке (x, y), должен за сколько-то шагов «дойти» до точки (x0 + i, y0 + j). Чтобы это сделать, надо просто составить два линейных отображения (мы уже сталкивались с этой задачей во время реализации модели «Жизнь» в предыдущей главе). Первое отображение будет переводить значение из отрезка [0...MaxStep] в значение из отрезка [x...x0 + i]. Здесь MaxStep – это максимальный номер шага (если первый шаг имеет номер 0, то последний шаг будет иметь номер, равный общему количеству шагов минус один). Второе отображение будет аналогичным образом определять значение из диапазона [y...y0 + j]. Чтобы сэкономить время, я сразу приведу готовые формулы, по которым можно определить позицию пиксела на экране во время любого шага step:

```
Xstep := x - step * (x - (x0 + i)) / MaxStep
Ystep := y - step * (y - (y0 + j)) / MaxStep
```

Вот, в принципе, и все. Остальное – дело техники (тем не менее доведу дело до конца).

Прежде всего нам придется где-то хранить информацию о каждом пикселе картинки (откуда и куда летит, какого цвета). Для этого опишем типы Point и PointArray:

```
type
  Point = record
      color : TColor;        { цвет пиксела }
      x, y  : Integer;       { откуда летит }
  end;
type PointArray = array of array of Point;     { тип массива пикселов }
```

Далее, нам потребуются переменные и константы:

```
var i, j          : Integer;              { счетчики циклов }
    oldtime       : TDateTime;            { время в начале цикла }
    pause         : Integer;              { величина задержки }
    pa            : PointArray;           { массив пикселов }
    x0, y0        : Integer;              { левый верхний угол картинки }
    step          : Integer;              { номер текущего шага }
    Xstep, Ystep : Integer;               { позиция пиксела на данном шаге }

const
    MaxStep = 100;                        { номер последнего шага }
    MSecsPerFrame = 25;                   { количество миллисекунд на кадр }
```

Вы уже, наверное, обратили внимание, что информация о целевой точке каждого пиксела («куда летит») нигде не хранится. Этого и не нужно, если договориться, что в элементе массива pa[i, j] будет содержаться информация о пикселе, который летит в позицию (x0 + i, y0 + j). Перед началом главного цикла программы надо «раскидать» пикселы картинки по экрану:

```
Randomize;
SetLength(pa, Background.Width, Background.Height); { выделение памяти для }
                                                    { массива пикселов }
for i := 0 to Background.Width - 1 do
    for j := 0 to Background.Height - 1 do
    begin
        pa[i, j].color := Background.Canvas.Pixels[i, j]; { запомнить цвет }
        pa[i, j].x := Random(Screen.Width);               { пиксела и позицию, }
        pa[i, j].y := Random(Screen.Height);              { откуда он летит }
    end;
```

Интересующая нас часть главного цикла тоже довольно очевидна:

```
{ очистить экран }
BackScreen.Canvas.FillRect(Rect(0, 0, BackScreen.Width, BackScreen.Height));
for i := 0 to Background.Width - 1 do
    for j := 0 to Background.Height - 1 do
    begin                                  { определить текущую позицию пиксела }
        Xstep := Round(pa[i, j].x - step * (pa[i, j].x - (x0 + i)) / MaxStep);
        Ystep := Round(pa[i, j].y - step * (pa[i, j].y - (y0 + j)) / MaxStep);
        { и нарисовать его }
        BackScreen.Canvas.Pixels[Xstep, Ystep] := pa[i, j].color;
    end;
{ скопировать буфер на экран }
Screen.Canvas.CopyRect(Rect(0, 0, Screen.Width, Screen.Height),
                       BackScreen.Canvas, Rect(0, 0, Screen.Width, Screen.Height));
step := step + 1;
if step > MaxStep then Form1.StartStopBtnClick(nil); { конец алгоритма }
```

Проекты для самосовершенствования

1. Подумайте, как можно реализовать циклический скроллинг. Замените скроллинг из примера (помните, у нас фон сначала прокручивался до конца влево, потом — до конца вправо и так до бесконечности) на циклический.
2. Осуществите вертикальный и диагональный скроллинг.
3. Напишите программу, в которой бы использовался многослойный фон (к примеру, на самом дальнем плане были бы облака, чуть поближе — деревья, а совсем близко — шоссе). Те фоновые картинки, которые расположены дальше, должны прокручиваться медленнее, а те, которые ближе — быстрее. Не забывайте использовать прозрачность, чтобы разные фоновые картинки не загораживали друг друга.
4. Реализуйте «затухание наоборот» (fading in в отличие от разобранного fading out): изначально на экране виден лишь однотонный прямоугольник, затем на нем проступает все больше деталей, пока они не превратятся в полноценную картинку.
5. Используйте эффект затухания для плавного перетекания одной картинки в другую. Если приближать каждый пиксел картинки не к какому-то одному цвету, а к цвету соответствующего пиксела другой картинки, первая картинка постепенно превратится во вторую.
6. Если развернуть в обратную сторону процесс составления картинки из отдельных пикселов, можно получить не менее интересный эффект: целая картинка по пикселам разлетается во все стороны. Напишите программу, которая этот эффект демонстрирует.
7. В эффекте «красивая смена фона» четные линии двигаются в одну сторону, нечетные — в другую. Перепишите программу так, чтобы вся левая половина фонового изображения двигалась влево, а правая — вправо, открывая вторую картинку (получатся «шторки»).
8. Создайте программу, демонстрирующую возможности «черепашьей графики». Пользователь должен видеть саму «черепашку» и давать ей команды (через текстовое поле, расположенное на форме). Используйте спрайтовую анимацию для рисования «черепашки» и отображения ее движений.

Глава 3

Трехмерная графика

An unspeakable horror seized me. There was a darkness; then a dizzy, sickening
sensation of sight that was not like seeing; I saw a Line that was no Line;
Space that was not Space: I was myself, and not myself.
When I could find voice, I shrieked loud in agony,
"Either this is madness or it is Hell."
"It is neither, calmly replied the voice of the Sphere,
"it is Knowledge; it is Three Dimensions:
open your eye once again and try to look steadily."

Edvin A. Abbot, «Flatland»[1]

В этой части книги мы разберемся, как же реализуется в компьютере трехмерная графика. Хотя наше знакомство с нею будет довольно поверхностным, я думаю, этого хватит, чтобы понять, интересно это вам или нет. Если интересно, почитаете другие книги (к счастью, тема сейчас популярная, хорошая литература есть), а если нет — тогда и жалеть не о чем.

Представление трехмерных объектов в памяти

Сразу же скажу, что мы ограничимся так называемыми каркасными (буквально, проволочными — wireframe) моделями. Этим словом называют фигуры, которые выглядят так, как будто бы они сделаны из проволоки (рис. 3.1).

Именно так выглядели объекты в первых трехмерных играх, и так выглядят разрабатываемые модели даже в самых современных редакторах 3D-графики (по крайней мере, этот режим очень широко в них используется). Более «продвинутые» темы (в первую очередь, удаление невидимых поверхностей и наложение текстур) остаются за пределами книги.

---

[1] Меня охватил невыразимый ужас. Сначала была тьма, потом головокружение, отвратительное чувство, словно я продолжаю видеть, но это не было похоже на зрение. Я видел Прямую, которая не была Прямой; Пространство, которое не было Пространством: я был самим собою и не был самим собою. Когда ко мне вернулся дар речи, я выкрикнул в агонии: «Это или сумасшествие, или ад». «Это ни то ни другое, — невозмутимо ответил голос Сферы, — это Знание; это Три Измерения: открой снова свой глаз и старайся смотреть внимательно». Эдвин Эббот, «Флатландия» (*перевод мой*).

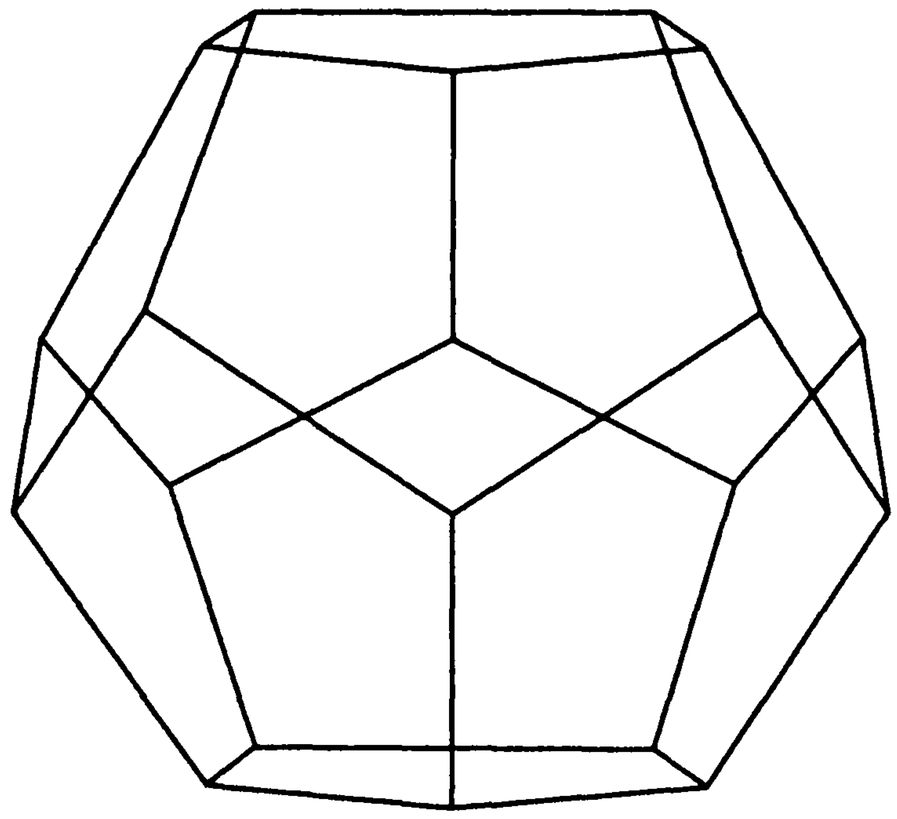


Рис. 3.1. Пример «проволочной» модели

Каркасные модели состоят из точек (вершин), соединенных отрезками прямых (ребрами), поэтому для представления моделей нам потребуются типы данных «вершина» и «ребро»:

```
type
  Vertex = record
      x, y, z : Real;  { координаты вершины }
  end;

  Edge = record
      src, dest : Integer;   { индексы соединяемых вершин }
  end;
```

Вершины модели будут храниться в массиве, поэтому для задания ребра достаточно указать порядковые номера вершин, которые это ребро соединяет. Сам трехмерный объект будет храниться в записи Object3D:

```
Object3D = record
    vertices    : array of Vertex;   { вершины }
    edges       : array of Edge;     { ребра }
    xc, yc, zc : Real;               { координаты центра объекта }
end;
```

Центральная точка модели выполняет две важные функции: во-первых, она задает положение объекта в пространстве; во-вторых, она служит началом координат во время всех операций с ним. Даже если ваша модель имеет сложную форму (космический корабль), все равно какой-то центр у нее должен быть. Иначе как реализовать, к примеру, разворот модели в противоположную сторону? Должна существовать точка, которая при этом развороте останется неподвижной; иначе говоря, точка, через которую пройдет ось вращения модели.

Удобно считать, что координаты всех вершин объекта заданы относительно центральной его точки, а не относительно начала координат: такой подход избавляет от большого количества работы во время операций над моделью (скоро вы сами в этом убедитесь).

Саму модель проще всего хранить в текстовом файле, например, такого формата:

```
N                { количество вершин }
x1 y1 z1         { координаты первой вершины (относительно цетра) }
x2 y2 z2         { координаты второй вершины }
```

```
...
xN yN zN          { координаты последней вершины }
M                 { количество ребер }
src1 dest1        { первое ребро (откуда - куда) }
src2 dest2        { второе ребро }
...
srcM destM        { последнее ребро }
```

Для дальнейших экспериментов вы, конечно, можете создать любой объект, какой вам только нравится; я же воспользуюсь описанием:

```
6
-100 -100 0
-100 100 0
100 100 0
100 -100 0
0 0 -141.42
0 0 141.42
12
0 1
1 2
2 3
3 0
4 0
4 1
4 2
4 3
5 0
5 1
5 2
5 3
```

Эти числа определяют фигуру, называемую октаэдром (рис. 3.2).

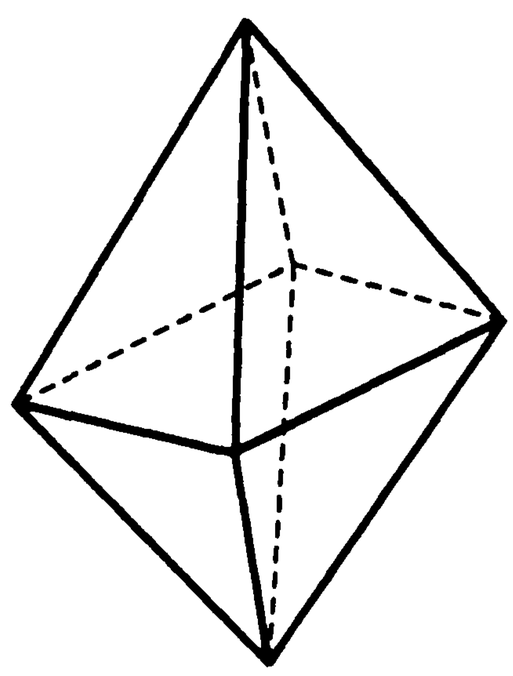


Рис. 3.2. Октаэдр

Теперь нам потребуется несложная функция, которая загружает модель из текстового файла:

```
function LoadObject3D(filename : String) : Object3D;
var F       : TextFile;          { исходный файл }
    NV, NE  : Integer;           { количество вершин, количество ребер }
    Shape   : Object3D;          { модель }
    i       : Integer;           { счетчик цикла }
begin
```

```
    AssignFile(F, filename);          { открываем исходный файл }
    FileMode := 0;                    { в режиме "только чтение" }
    Reset(F);

    ReadLn(F, NV);                      { считываем количество вершин }
    SetLength(Shape.vertices, NV);      { и корректируем размер массива }
    for i := 0 to NV - 1 do             { считываем каждую вершину }
        ReadLn(F, Shape.vertices[i].x, Shape.vertices[i].y,
                                       Shape.vertices[i].z);

    ReadLn(F, NE);                      { то же для ребер }
    SetLength(Shape.edges, NE);
    for i := 0 to NE - 1 do
        ReadLn(F, Shape.edges[i].src, Shape.edges[i].dest);

    CloseFile(F);
    LoadObject3D := Shape;
end;
```

Не забывайте, что первый элемент динамического массива всегда имеет нулевой индекс.

Итак, первая задача – хранение объекта в памяти – выполнена. Теперь посмотрим, что можно делать с нашими творениями.

Операции над трехмерными объектами

Под операцией над объектом я подразумеваю любое действие, которое вы только можете себе вообразить. Конечно, есть операции простые, которые легко реализовать (например, поворот или сдвиг объекта), есть и сложные (деформация), однако в любом случае объект состоит из вершин (ребра лишь соединяют вершины), поэтому операция над объектом – это какое-то действие над каждой его вершиной. Проще всего запрограммировать операцию, при которой над всеми вершинами объекта производится одно и то же действие.

Итак, любая операция над объектом сводится к операциям над его вершинами, то есть простыми точками в трехмерном пространстве. В свою очередь, любое действие над отдельной точкой – это какая-то последовательность атомарных (элементарных) действий, коих всего три [1]:

1. Перенос на вектор (Tx, Ty, Tz): точка сдвигается на величину Tx вдоль оси Ox, на величину Ty вдоль оси Oy и на величину Tz вдоль оси Oz (рис. 3.3).
2. Поворот на угол α вокруг любой оси (Ox, Oy или Oz): координата, соответствующая оси, не изменяется, зато меняются две оставшиеся (рис. 3.4).
3. Растяжение/сжатие/отражение: значение каждой координаты точки умножается на соответствующий коэффициент (Sx, Sy, Sz). Если значение коэффициента по модулю больше единицы, расстояние от точки до начала координат увеличится (растяжение, рис. 3.5), если меньше – уменьшится (сжатие). Если

[1] Точнее, любое действие можно запрограммировать как последовательность атомарных действий.

при этом коэффициент имеет отрицательное значение, точка «перепрыгнет» в позицию, лежащую по другую сторону от начала координат. Таким образом можно получать зеркальные отражения объектов.

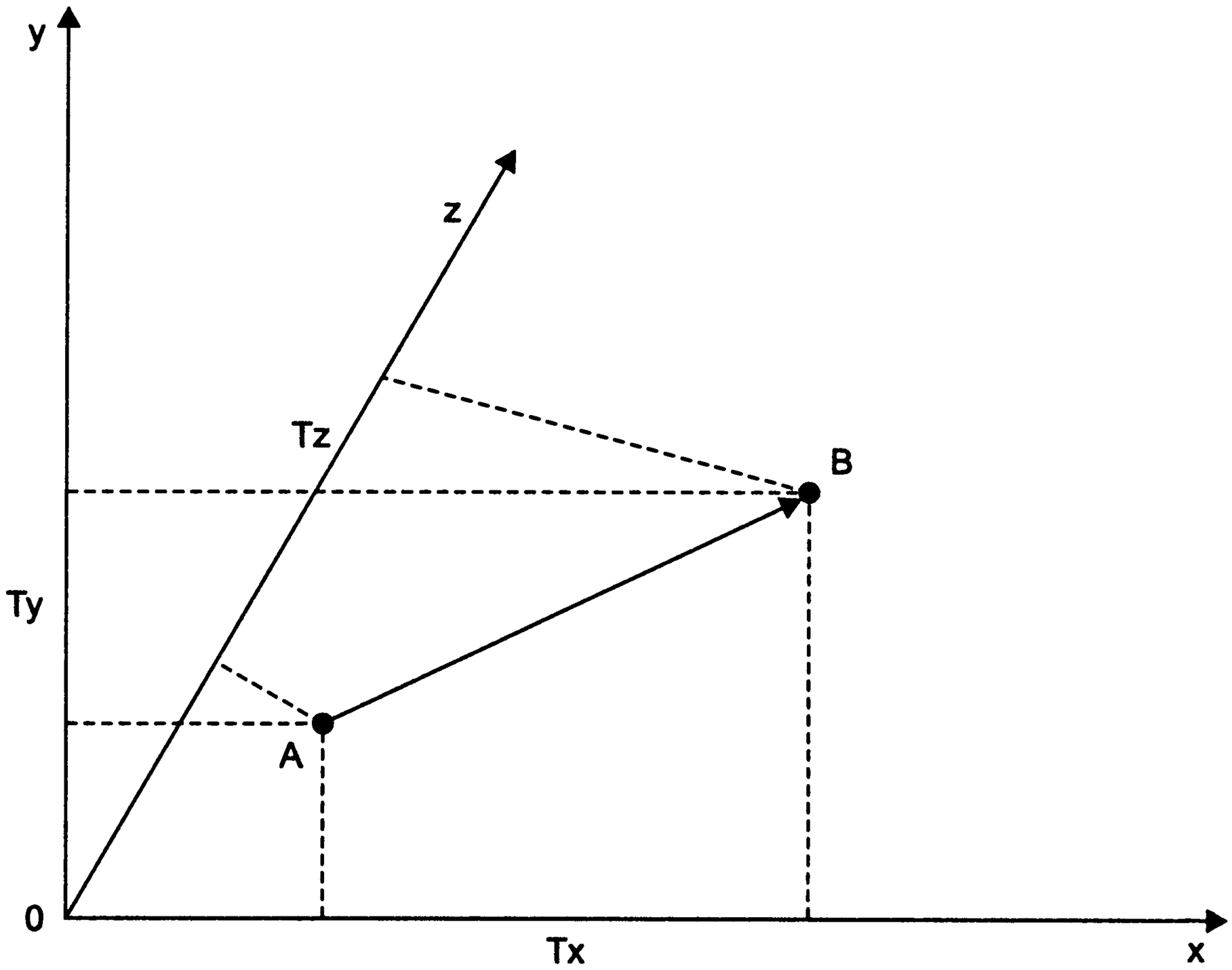


Рис. 3.3. Перенос точки из пункта A в пункт B

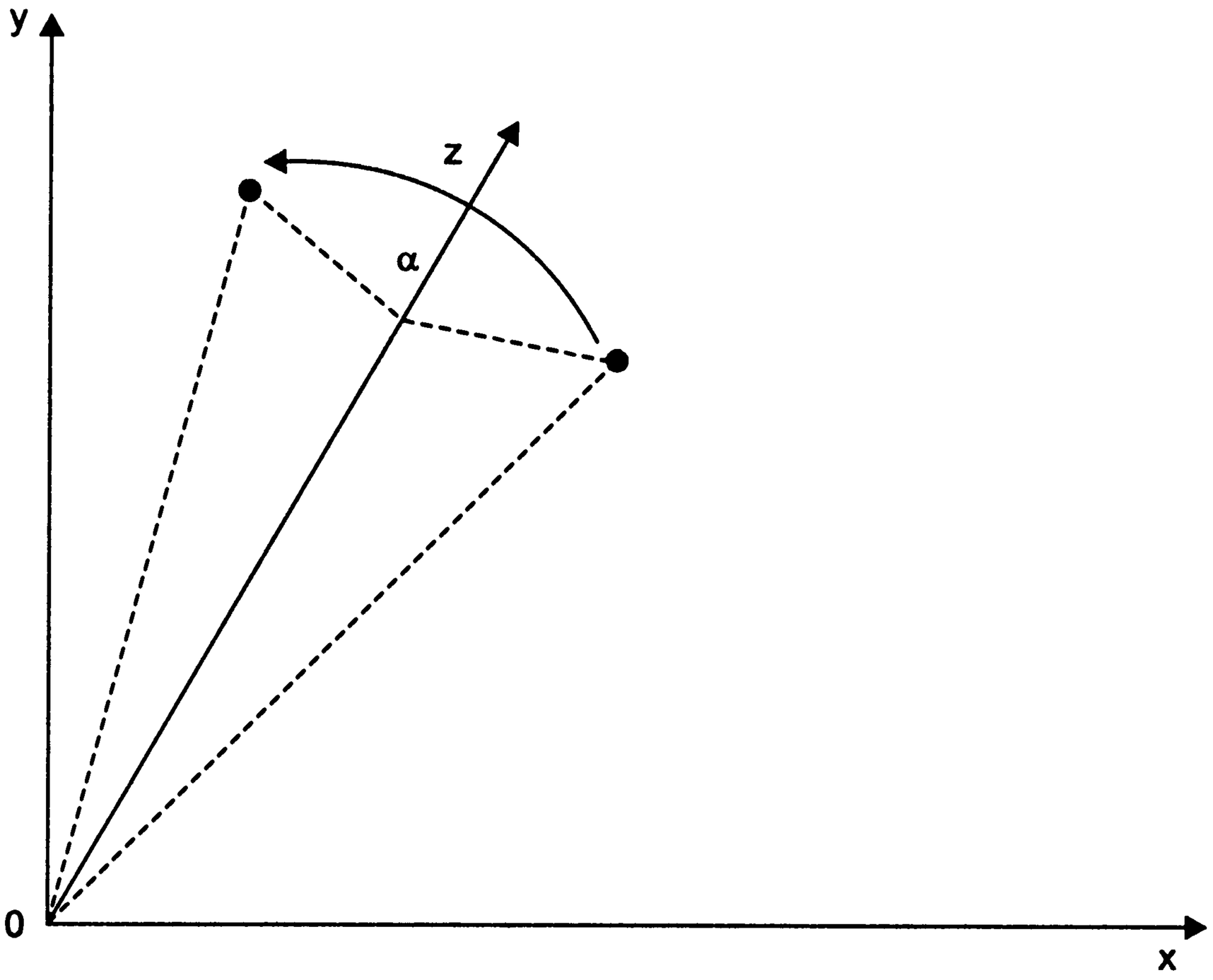


Рис. 3.4. Поворот точки вокруг оси Oz

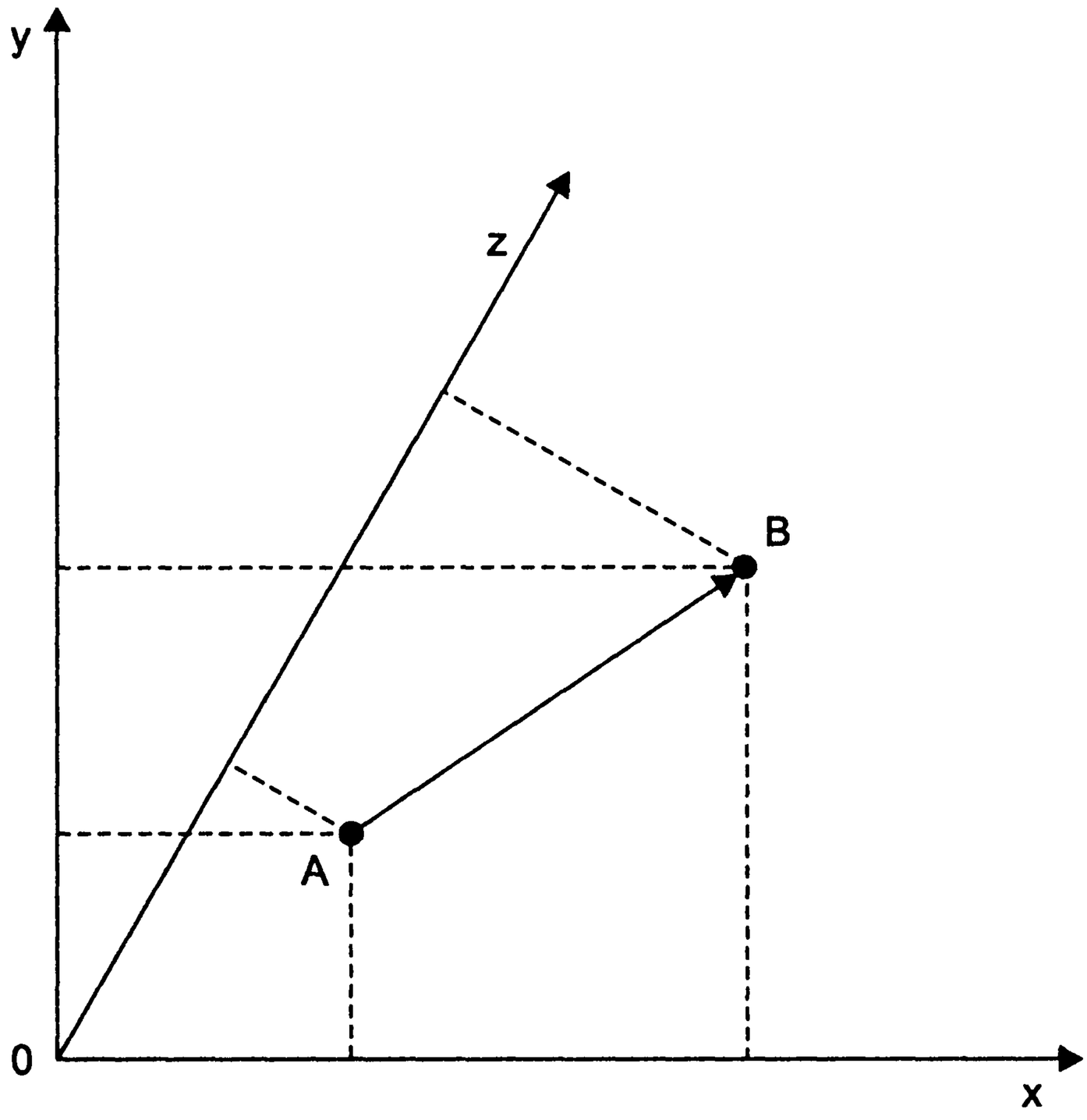


Рис. 3.5. Растяжение с коэффициентами (2, 2, 2): каждая координата увеличивается вдвое

Я полагаю, что ось Oz направлена вглубь экрана: если z-координата объекта увеличивается, он оказывается все дальше и дальше от зрителя и наоборот. Нулевое значение z-координаты — это уже поверхность монитора, недостижимая для виртуальных объектов; отрицательное значение — наш реальный мир (к сожалению или к счастью, объект не материализуется, если просто задать в качестве значения его z-координаты число меньше нуля).

Три операции, о которых только что шла речь, действительно очень просты, «атомарны». Даже для такого несложного действия, как, скажем, поворот одной точки (A) относительно другой (B), требуется проделать три шага:

1) совместить начало координат с точкой B (то есть передвинуть A таким образом, чтобы начало координат оказалось относительно нее там же, где была точка B ранее — до процедуры сдвига);
2) осуществить поворот;
3) вернуть начало координат на прежнее место.

По этой причине, в частности, удобно задавать положения вершин именно относительно центра объекта: тогда для каждой вершины центр и будет началом координат. Чтобы повернуть модель целиком вокруг своей оси, никаких сдвигов не потребуется.

Я понимаю, эти вещи не так просты, как хотелось бы, и ничего с этим не поделаешь. Если вы не совсем поняли, что к чему, прочитайте раздел еще раз — он действительно важен. Если же все понятно (или хотя бы кажется, что понятно), идем дальше. Самое трудное уже позади.

Итак, чтобы управлять трехмерным объектом, необходимо уметь производить с ним три действия: перенос, поворот и растяжение/сжатие/отражение. Сейчас мы попробуем создать процедуры, которые этим, собственно, и будут заниматься, но сначала – немножко математики.

Вы уже, наверное, догадываетесь, что все преобразования, о которых шла речь, давно известны в геометрии. Существуют, конечно, готовые «волшебные формулы»: подставляем, к примеру, в такую формулу координаты точки, значения величин Tx, Ty и Tz и получаем новые координаты, соответствующие точке после переноса на вектор (Tx, Ty, Tz). Я могу, конечно, просто переписать их из справочника по математике сюда, обвести в рамочку, а потом создать на их основе все необходимые процедуры. Тем не менее поверьте пока на слово, что это не самый лучший путь. Мы потратим на десять минут больше, и затраченное время окупится с лихвой теми возможностями, которые открывают для нас *матрицы*.

Матрица в математике – это то же самое (ну, или почти то же самое), что и двумерный массив в программировании. Интерес для нас сейчас представляет тот факт, что матрицы можно перемножать! Для этого есть несколько правил.

Первое. Матрицу A можно умножить на матрицу B (получая при этом произведение A * B) только тогда, когда количество столбцов в матрице A равно количеству строк в матрице B. В результирующей матрице будет столько строк, сколько в матрице A, и столько столбцов, сколько в матрице B. На более «математическом» языке это правило звучит так: «матрицу размера $(N \times M)$[1] можно умножить только на матрицу размера $(M \times K)$, при этом размерность результирующей матрицы будет равна $(N \times K)$».

Второе (вытекает из первого). Если матрицу A можно умножить на матрицу B, это еще не означает, что B можно умножить на A. Если повезет, и это удастся (например, обе матрицы квадратные и равны между собой по размеру), от перестановки мест сомножителей произведение изменится: $A * B \neq B * A$!

Третье. Элемент, который в результирующей матрице находится на позиции [i, j], определяется по формуле:

```
R[i, j] = A[i, 1] * B[1, j] + A[i, 2] * B[2, j] + ... + A[i, M] * B[M, j]
```

Иными словами, чтобы получить элемент R[i, j], надо умножить каждый компонент i-й строки первой матрицы на соответствующий компонент j-го столбца второй матрицы («строка на столбец») и сложить результаты. Поначалу кажется странным, но со временем привыкаешь.

Обратите внимание на два следствия из этих правил:

1) если умножить квадратную матрицу на другую квадратную матрицу того же размера, снова получится квадратная матрица того же размера;
2) если умножить квадратную матрицу на столбец (то есть на матрицу, у которой столько же строк, сколько и у первой, но всего один столбец), снова получится столбец.

Теперь самое главное: зачем нам все это нужно. Представьте себе, что координаты некоторой точки записаны в столбец из четырех элементов. Первые три эле-

[1] N строк, M столбцов.

мента столбца содержат три координаты точки, а четвертый – единицу. Если умножить на этот столбец квадратную матрицу (4 × 4), то результатом снова будет столбец. Первые три элемента полученного столбца можно интерпретировать как координаты некоторой другой точки (рис. 3.6).

$$\begin{pmatrix} A11 & A12 & A13 & A14 \\ A21 & A22 & A23 & A24 \\ A31 & A32 & A33 & A34 \\ A41 & A42 & A43 & A44 \end{pmatrix} \times \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix}$$

Рис. 3.6. Преобразование координат при помощи матрицы

Разумеется, если взять какую-нибудь случайную матрицу и умножить ее на столбец, ничего осмысленного в результате не получится; однако вся суть в том, что для каждого из рассмотренных нами преобразований существует соответствующая матрица. Иначе говоря, берем соответствующую, к примеру, повороту матрицу, умножаем ее на столбец с координатами исходной точки и получаем в результате столбец с координатами новой, искомой точки. Вот они, «волшебные» матрицы преобразований.

1. Перенос на вектор (Tx, Ty, Tz).

```
1 0 0 Tx
0 1 0 Ty
0 0 1 Tz
0 0 0 1
```

2. Поворот на угол A вокруг оси Ox.

```
1   0       0    0
0 cos(A) -sin(A) 0
0 sin(A)  cos(A) 0
0   0       0    1
```

3. Поворот на угол A вокруг оси Oy.

```
cos(A)  0 sin(A)  0
  0     1    0    0
-sin(A) 0 cos(A)  0
  0     0    0    1
```

4. Поворот на угол A вокруг оси Oz.

```
cos(A) -sin(A)  0 0
sin(A)  cos(A)  0 0
  0       0     1 0
  0       0     0 1
```

5. Растяжение/сжатие/отражение с коэффициентами (Sx, Sy, Sz).

```
Sx 0  0  0
0  Sy 0  0
0  0  Sz 0
0  0  0  1
```

Итак, чтобы выполнить операцию над точкой, надо лишь выбрать соответствующую матрицу и умножить ее на столбец, содержащий координаты точки. Пусть,

к примеру, точку с координатами (1, 2, 3) надо перенести на вектор (3, 2, 1). Для этого берем матрицу из пункта 1, подставляем вместо Tx, Ty и Tz соответствующие числа и умножаем на столбец, содержащий значения (1, 2, 3, 1) (не забывайте про единицу в конце). В результате получим требуемые координаты новой точки (рис. 3.7).

$$\begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 4 \\ 4 \\ 1 \end{pmatrix}$$

Рис. 3.7. Перенос точки на вектор при помощи матричного преобразования

Как видите, матрицы позволяют легко и элегантно записать любое атомарное преобразование. Но основное их преимущество состоит в другом: из матриц, определяющих операции над объектами, можно составлять цепочки. Представьте, что матрица A1 задает одну операцию, матрица A2 – вторую, а матрица A3 – третью. Если перемножить эти матрицы между собой, то результирующая матрица R = A1 × A2 × A3 будет определять итоговое преобразование объекта. Применить это преобразование к модели – то же самое, что и применить по очереди преобразования, задаваемые матрицами A1, A2 и A3. Если вы заметили, что постоянно используете какую-то последовательность атомарных преобразований, имеет смысл записать ее в виде одной итоговой матрицы – это, естественно, сильно снизит накладные расходы на вычисления.

Перейдем теперь от теории к практике. Сразу хочу сказать, что в итоге у нас получится программа, отображающая на экране трехмерную модель, которая вращается вокруг своего центра и перемещается в пространстве. Поэтому здесь я приведу процедуры, которые занимаются только вращением[1]. Остальные создаются совершенно аналогичным образом.

Итак, начнем, как всегда, с описания новых типов данных. На сей раз нам потребуются «матрица» (размерности 4 × 4) и «столбец» (матрица 4 × 1):

```
Matrix = array[1..4, 1..4] of Real;   { матрица 4x4 }
Column = array[1..4] of Real;         { столбец }
```

Напишем также функции умножения матрицы на матрицу и матрицы на столбец:

```
function MMMult(lhs, rhs : Matrix) : Matrix; { умножение матрицы на матрицу }
var i, j, k : Integer;
    r       : Matrix;
    s       : Real;
begin
    for i := 1 to 4 do                        { простая интерпретация }
        for j := 1 to 4 do                    { соответствующей формулы }
        begin
            s :=.0;
```

[1] В данном конкретном случае для перемещения модели нам не потребуется производить матричные преобразования.

```
            for k := 1 to 4 do
                s := s + lhs[i, k] * rhs[k, j];
            r[i, j] := s;
        end;
    MMMult := r;
end;

function MCMult(lhs : Matrix; rhs : Column) : Column;  { умножение матрицы }
var k, i : Integer;                                    { на столбец }
    s    : Real;
    r    : Column;
begin
    for i := 1 to 4 do                                 { аналогично MMMult }
    begin
        s := 0;

        for k := 1 to 4 do
           s := s + lhs[i, k] * rhs[k];
        r[i] := s;
    end;

    MCMult := r;
end;
```

Атомарные операции позволяют повернуть объект только вокруг какой-нибудь одной оси координат. Поскольку же мы собираемся вращать модель произвольным образом, необходимо создать матрицу, соответствующую этой неатомарной операции. Матрица, задающая произвольное вращение (на любой заданный угол вокруг любой заданной оси), — это произведение трех матриц, задающих атомарные операции вращения (об этой идее мы уже говорили). Следующая функция генерирует матрицу преобразования по заданным углам поворота относительно каждой из осей координат:

```
function RotateMatrix(xa, ya, za : Real) : Matrix; { матрица поворота модели }
var xr, yr, zr : Matrix;
begin
    { матрица поворота вокруг оси Ox }
    xr[1, 1] := 1; xr[1, 2] := 0;       xr[1, 3] := 0;         xr[1, 4] := 0;
    xr[2, 1] := 0; xr[2, 2] := Cos(xa); xr[2, 3] := -Sin(xa); xr[2, 4] := 0;
    xr[3, 1] := 0; xr[3, 2] := Sin(xa); xr[3, 3] := Cos(xa);  xr[3, 4] := 0,
    xr[4, 1] := 0; xr[4, 2] := 0;       xr[4, 3] := 0;         xr[4, 4] := 1;

    { матрица поворота вокруг оси Oy }
    yr[1, 1] := Cos(ya);  yr[1, 2] := 0; yr[1, 3] := Sin(ya); yr[1, 4] := 0,
    yr[2, 1] := 0;        yr[2, 2] := 1; yr[2, 3] := 0;       yr[2, 4] := 0;
    yr[3, 1] := -Sin(ya); yr[3, 2] := 0; yr[3, 3] := Cos(ya); yr[3, 4] := 0,
    yr[4, 1] := 0;        yr[4, 2] := 0; yr[4, 3] := 0;       yr[4, 4] := 1;

    { матрица поворота вокруг оси Oz }
    zr[1, 1] := Cos(za); zr[1, 2] := -Sin(za); zr[1, 3] := 0; zr[1, 4] := 0;
    zr[2, 1] := Sin(za); zr[2, 2] := Cos(za),  zr[2, 3] := 0; zr[2, 4] := 0,
    zr[3, 1] := 0;       zr[3, 2] := 0;        zr[3, 3] := 1; zr[3, 4] := 0;
    zr[4, 1] := 0;       zr[4, 2] := 0;        zr[4, 3] := 0; zr[4, 4] := 1,
```

```
    RotateMatrix := MMMult(MMMult(xr, yr), zr);
end;
```

Как вы можете убедиться, работает она совершенно прямолинейно: генерирует три уже знакомых нам матрицы атомарных поворотов и возвращает результат их умножения друг на друга.

Имея матрицу итогового преобразования, совсем не трудно написать процедуру, которая вращает модель, используя эту матрицу:

```
procedure RotateShape(var Shape : Object3D; xa, ya, za : Real);
var rm : Matrix;
    i  : Integer;
    c  : Column;
begin
    rm := RotateMatrix(xa, ya, za);          { сгенерировать матрицу вращения }
    c[4] := 1;                  { последний элемент столбца всегда равен единице}

    for i := 0 to High(Shape.vertices) do  { цикл по всем вершинам }
    begin                              { High(a) возвращает верхний индекс массива a }
        c[1] := Shape.vertices[i].x;       { инициализация столбца }
        c[2] := Shape.vertices[i].y;
        c[3] := Shape.vertices[i].z;

        c := MCMult(rm, c);                { вызов преобразования }

        Shape.vertices[i].x := c[1];       { внесение изменений в модель }
        Shape.vertices[i].y := c[2];       { в соответствии с полученным }
        Shape.vertices[i].z := c[3];       { результатом преобразования }
    end;
end;
```

На этом тему операций над трехмерными объектами можно считать исчерпанной.

Отображение трехмерных объектов на экране

Экран плоский. Модели трехмерные. Конечно, было бы замечательно иметь трехмерное отображающее устройство, рисование трехмерных фигур на котором было бы столь же простым делом, что и отображение плоских спрайтов на мониторе компьютера. К сожалению, таких устройств у нас нет, поэтому приходится искать какой-то выход из ситуации. И этот выход есть: проекции, то есть методики, которые позволяют получить на экране примерно такое же изображение трехмерного объекта, как и на фотографии или, скажем, на рисунке в книге. При работе с трехмерной графикой чаще всего используются *ортогональные* и *перспективные* проекции (рис. 3.8).

При ортогональном проецировании не учитывается тот факт, что на расстоянии предметы нам кажутся меньше. Дальняя грань куба на рисунке слева имеет тот же размер, что и передняя. На самом же деле наблюдателю она должна казаться чуть меньше (это отражено в рисунке справа). Если модель приближается

к наблюдателю, она должна увеличиваться в размерах, если отдаляется — уменьшаться.

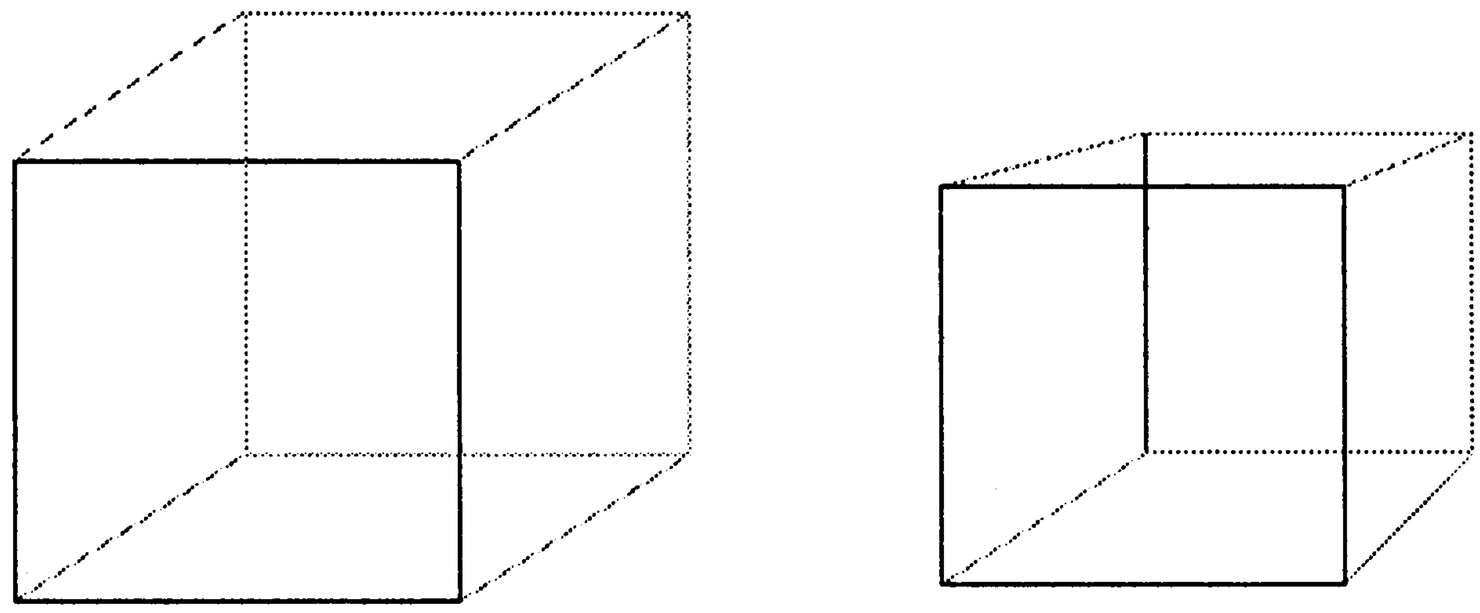


Рис. 3.8. Ортогональная (слева) и перспективная (справа) проекции

Если ваша программа имеет дело с небольшими объектами (для которых изменение размеров в пределах самой модели невелико), да еще и не двигающимися вдоль оси Oz, можно вполне обойтись ортогональным проецированием. В противном случае лучше воспользоваться перспективными проекциями. Мы рассмотрим оба варианта.

Нет ничего проще, чем реализовать ортогональную проекцию: для этого надо всего лишь игнорировать z-координаты точек. Предположим, координаты некоторой точки равны (x, y, z). При выводе на экран этой точке будет соответствовать пиксел (x, y). Вот и все.

Перспективная проекция немного сложнее. Для начала заметьте, что, удаляясь от наблюдателя, все объекты визуально стремятся в одну и ту же точку на горизонте. В подавляющем большинстве случаев разумно выбрать в качестве этой точки центр экрана (то есть бесконечно движущиеся вдаль фигуры будут все ближе и ближе к центру экрана). При таком выборе нас ожидает небольшое «последствие»: отныне начало координат тоже будет находиться в центре экрана. Это мало что меняет, просто запомните, что если x- и y-координаты центра объекта равны нулю, то сам объект находится в центре экрана. Теперь — формулы. Допустим, координаты некоторой точки равны (x, y, z). Тогда ей будет соответствовать пиксел с координатами x' и y', которые определяются по формулам:

```
x' = 0.5 * ширина_экрана + N * x / z
y' = 0.5 * высота_экрана + N * y / z
```

Обратите внимание, что при увеличении z-координаты обе формулы возвращают все более близкие к нулю (то есть к центру экрана) значения — и это правильно. Заметьте также, что при z = 0 происходит деление на нуль. Поскольку нулевое значение z-координаты означает «поверхность» монитора, такая ситуация не имеет смысла. Тем не менее, советую подстраховаться и вставить в программу соответствующую проверку (мы это сделаем позднее). В формуле осталось еще одно «магическое число» — константа N. Ее назначение — регулировка «глубины» экрана. Высота и ширина экрана всегда известны (640 × 480, 800 × 600, 1024 × 768 и т. д.), а вот его глубина (то есть максимальное значение z-координаты) — величина потенциально бесконечная. Тем не менее мы можем влиять на то, насколько быстро уменьшается в размерах объект при движении вглубь. Чем

меньше N, тем короче расстояние до горизонта; тем быстрее модель уменьшается, удаляясь от наблюдателя (рис. 3.9).

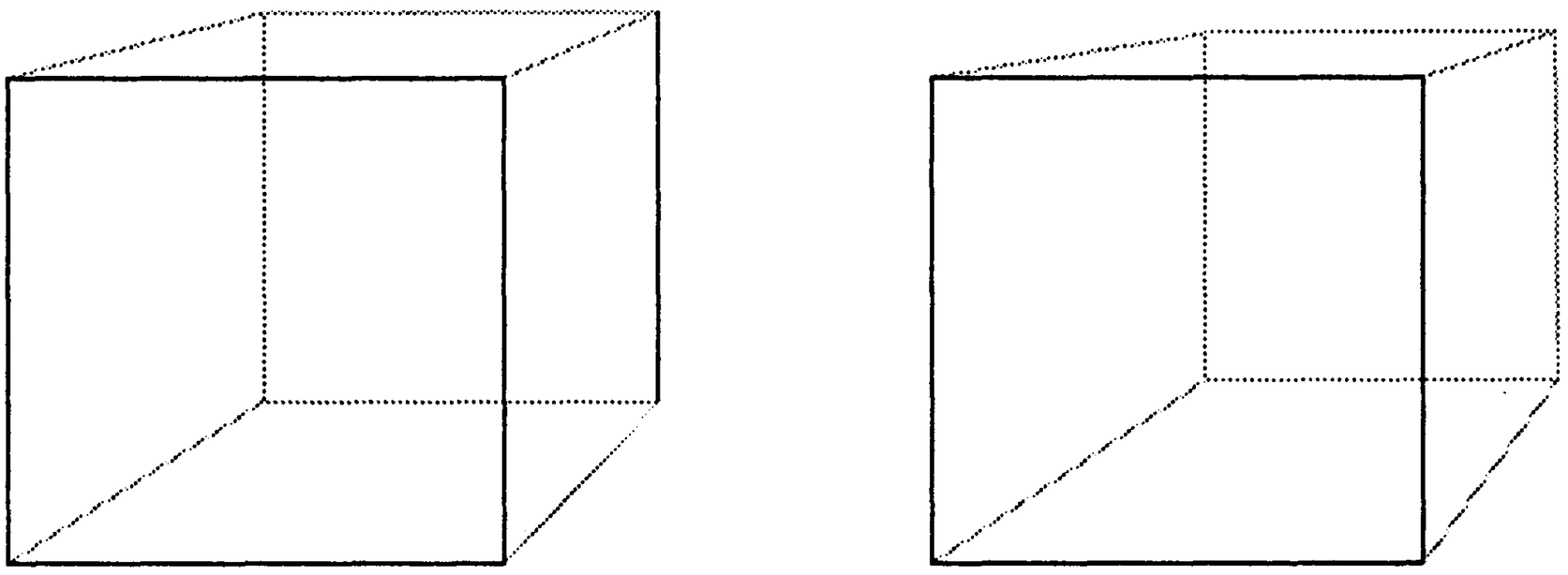


Рис. 3.9. Перспективная проекция одной и той же модели при разных значениях константы N

Напишем процедуру, выполняющую перспективное проецирование (проще говоря, процедуру, которая рисует модель на экране, используя перспективную проекцию):

```
procedure ShowShape(Shape : Object3D);                    { нарисовать объект }
var i, Xs, Ys : Integer;
    x, y, z   : Real;
const N = 300;                { "глубина" экрана }
begin
    for i := 0 to High(Shape.edges) do                    { цикл по всем ребрам }
begin                                                     { координаты первой точки ребра }
    x := Shape.vertices[Shape.edges[i].src].x + Shape.xc;
    y := Shape.vertices[Shape.edges[i].src].y + Shape.yc;
    z := Shape.vertices[Shape.edges[i].src].z + Shape.zc;
    if Abs(z) < 0.01 then z := 0.01;   { избегаем деления на нуль }
                                                          { вычисляем экранные координаты }
    Xs := Round(Form1.Screen.Width div 2 + N * x / z);
    Ys := Round(Form1.Screen.Height div 2 + N * y / z);

    Form1.BackScreen.Canvas.MoveTo(Xs, Ys);
                                                          { координаты второй точки ребра }
    x := Shape.vertices[Shape.edges[i].dest].x + Shape xc;
    y := Shape.vertices[Shape.edges[i].dest].y + Shape.yc;
    z := Shape.vertices[Shape.edges[i].dest].z + Shape.zc;
    if Abs(z) < 0.01 then z := 0.01;   { избегаем деления на нуль }
                                                          { и ее экранные координаты }
        Xs := Round(Form1.Screen.Width div 2 + N * x / z);
        Ys := Round(Form1.Screen.Height div 2 + N * y / z);

        Form1.BackScreen Canvas.LineTo(Xs, Ys).  { рисуем ребро }
    end.
end.
```

Обратите внимание, что в процедуре графический вывод реально происходит на буфер — элемент **BackScreen**. Здесь используется идея двойной буферизации из предыдущей главы; там же описан способ синхронизации с таймером (сейчас мы применим более простой подход из раздела о графических эффектах). Осталось описать главную процедуру, которая подытожит наши усилия (надеюсь, коммен-

тариев в тексте хватит для ее понимания; кроме того, я старался сделать ее максимально похожей по структуре на программы из предыдущей главы):

```
procedure TForm1.StartStopBtnClick(Sender: TObject);  { главная процедура }
    var oldtime : TDateTime;
        Model : Object3D;                { трехмерный объект }
        Vx, Vy, Vz : Real;               { значения составляющих его скорости }
        pause : Integer;
    const
        MSecsPerFrame = 25;              { скорость работы (кадров в секунду) }
        xa = 0.01;                       { скорость вращения вокруг оси Ox }
        ya = 0.05;                       { скорость вращения вокруг оси Oy }
        za = 0.08;                       { скорость вращения вокруг оси Oz }

begin
    if IsRunning then
    begin
        IsRunning := false;
        StartStopBtn.Caption := 'Пуск';
        Exit;
    end;

    StartStopBtn.Caption := 'Стоп';
    IsRunning := true;

    Model  = LoadObject3D('octahedron.txt');          { загрузить объект }

Model.xc := 0;                                        { начальное положение }
Model yc := 0;                                        { объекта }
Model.zc := 200;
Vx := 10;                                             { и составляющие }
Vy := 15;                                             { скорости }
Vz := 20;

while IsRunning do
begin
    oldtime := Now;

    Model.xc  := Model.xc + Vx;                  { переместить объект }
    Model.yc  := Model.yc + Vy;
    Model.zc  := Model.zc + Vz;

{ отразить от стены (3D-аналог "молекулы в закрытом сосуде") }
if Model.xc > BackScreen.Width div 2 then
    begin Model.xc := BackScreen.Width div 2; Vx := -Vx; end;
if Model.xc < -BackScreen.Width div 2 then
    begin Model.xc := -BackScreen.Width div 2; Vx := -Vx; end;

if Model.yc > BackScreen.Height div 2 then
    begin Model.yc := BackScreen.Height div 2; Vy := -Vy; end;
if Model.yc < -BackScreen.Height div 2 then
    begin Model.yc := -BackScreen.Height div 2; Vy := -Vy; end;

if Model.zc > 1000 then
```

```
    begin Model.zc := 1000; Vz := -Vz; end;
if Model.zc < 200 then
    begin Model.zc := 200. Vz := -Vz; end;

RotateShape(Model, xa, ya, za);                    { поворот модели }
BackScreen.Canvas.FillRect(Rect(0,0,Screen Width, Screen.Height));
ShowShape(Model);                  { очистить экран и нарисовать объект }
                                 { отобразить объект на основном экране }
Screen.Canvas.CopyRect(Rect(0, 0, Screen.Width, Screen.Height),
          BackScreen.Canvas, Rect(0, 0, Screen.Width, Screen.Height));
          Application.ProcessMessages;
          pause := Round(MSecsPerFrame - (Now - oldtime) * MSecsPerDay);
          if pause > 0 then Sleep(pause);                    { задержка }
    end;
end;
```

Результат работы приложения показан на рис. 3.10.

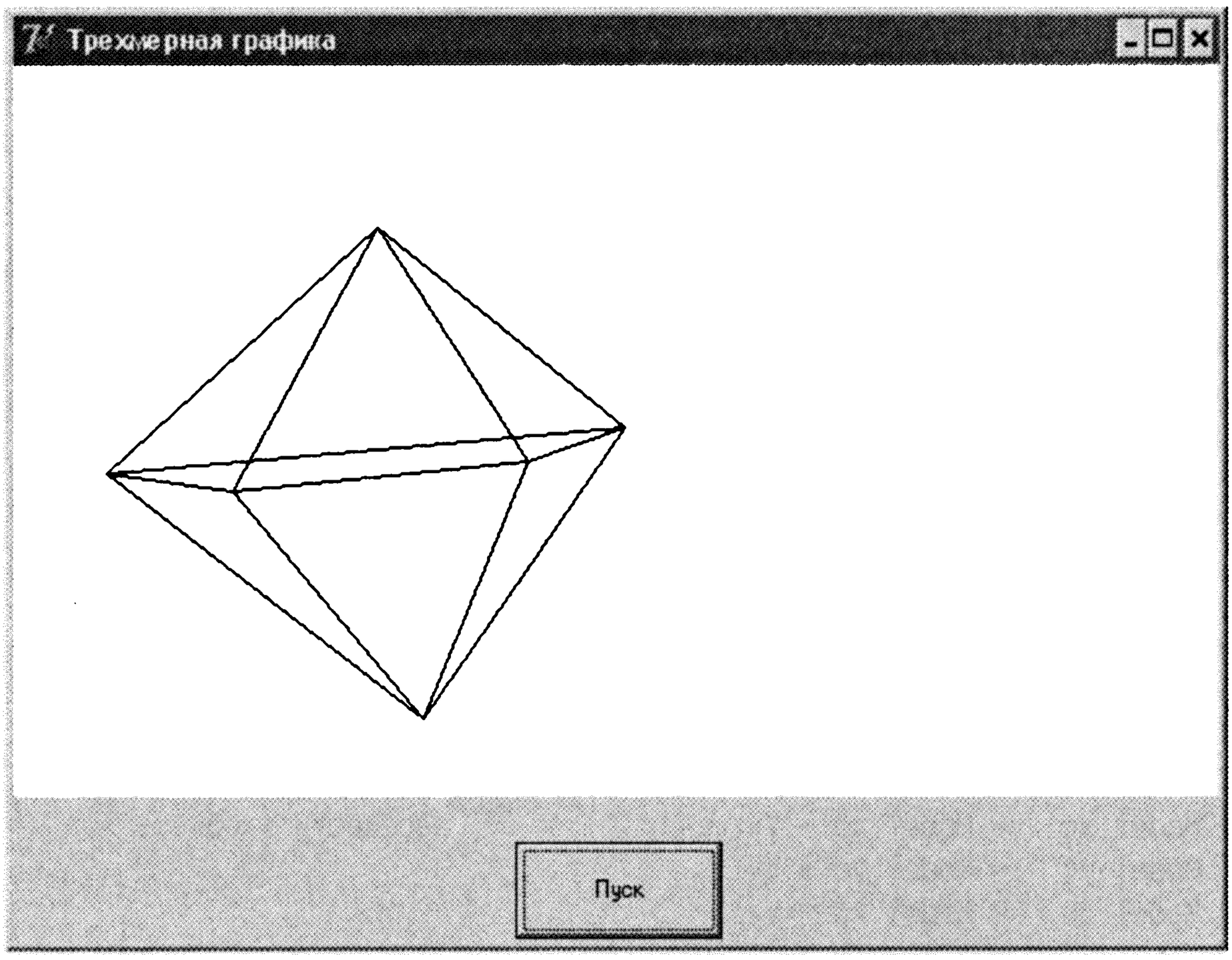


Рис. 3.10. Приложение «Трехмерная графика»

Проекты для самосовершенствования

1. Реализуйте процедуры, которые выполняют оставшиеся два преобразования – перенос и растяжение/сжатие/отражение. Напишите программу, демонстрирующую работу этих процедур.
2. Замените в нашей программе перспективное проецирование на ортогональное и посмотрите, что изменится.

3. Для простоты я убрал из демонстрационной программы все случайные величины. Измените программу так, чтобы модель появлялась в случайной точке пространства, двигалась в случайном направлении и со случайной скоростью. Скорости вращения вокруг каждой оси тоже должны определяться датчиком случайных чисел.
4. Замените в программе октаэдр на куб.
5. Попробуйте написать трехмерный аналог «Идеального газа». «Молекулами» могут служить любые трехмерные объекты.

Глава 4

Лабиринты

В лабиринте у вас, по крайней мере, есть цель.

Евгений Кащеев

Мне нравится эта тема. С одной стороны, она занимательна, с другой – полезна (я надеюсь, что вы найдете не одно и не два применения алгоритмам, описанным в этой части книги), а с третьей – даже научна. Лабиринты, выражаясь математическим языком – это *графы*; все алгоритмы, которые мы рассмотрим, соответственно, являются алгоритмами на графах. Графам же посвящена целая теория в математике (она так и называется: *теория графов*). Недаром два алгоритма, с которыми вы познакомитесь в этой главе, носят имена Прима (Prim) и Краскала (Kruskal) – очень известных в теории графов людей.

Итак, сейчас мы займемся *лабиринтами*. Если говорить более конкретно, в область наших интересов входят следующие задачи:

1) определить, что такое лабиринт и как представить его в памяти компьютера;

2) договориться, что значит «решить» лабиринт и как это сделать;

3) выяснить, как лабиринт может быть создан компьютером автоматически.

Представление лабиринтов в памяти

Толковый словарь русского языка определяет лабиринт как «запутанную сеть дорожек, ходов, сообщающихся друг с другом помещений». Пожалуй, это определение слишком неконкретное и чересчур общее для наших целей. Сразу скажу, что наши лабиринты – плоские (двумерные, если угодно), поэтому их можно легко нарисовать на бумаге. Кроме того, все помещения (их обычно называют *локациями*) будут квадратными, а сам лабиринт (естественно) – прямоугольным. У каждого такого помещения есть не более четырех соседних, и в каждое соседнее помещение может вести проход (а может и не вести – в этом случае там будет «стена»). Думаю, проще всего посмотреть на пример (рис. 4.1).

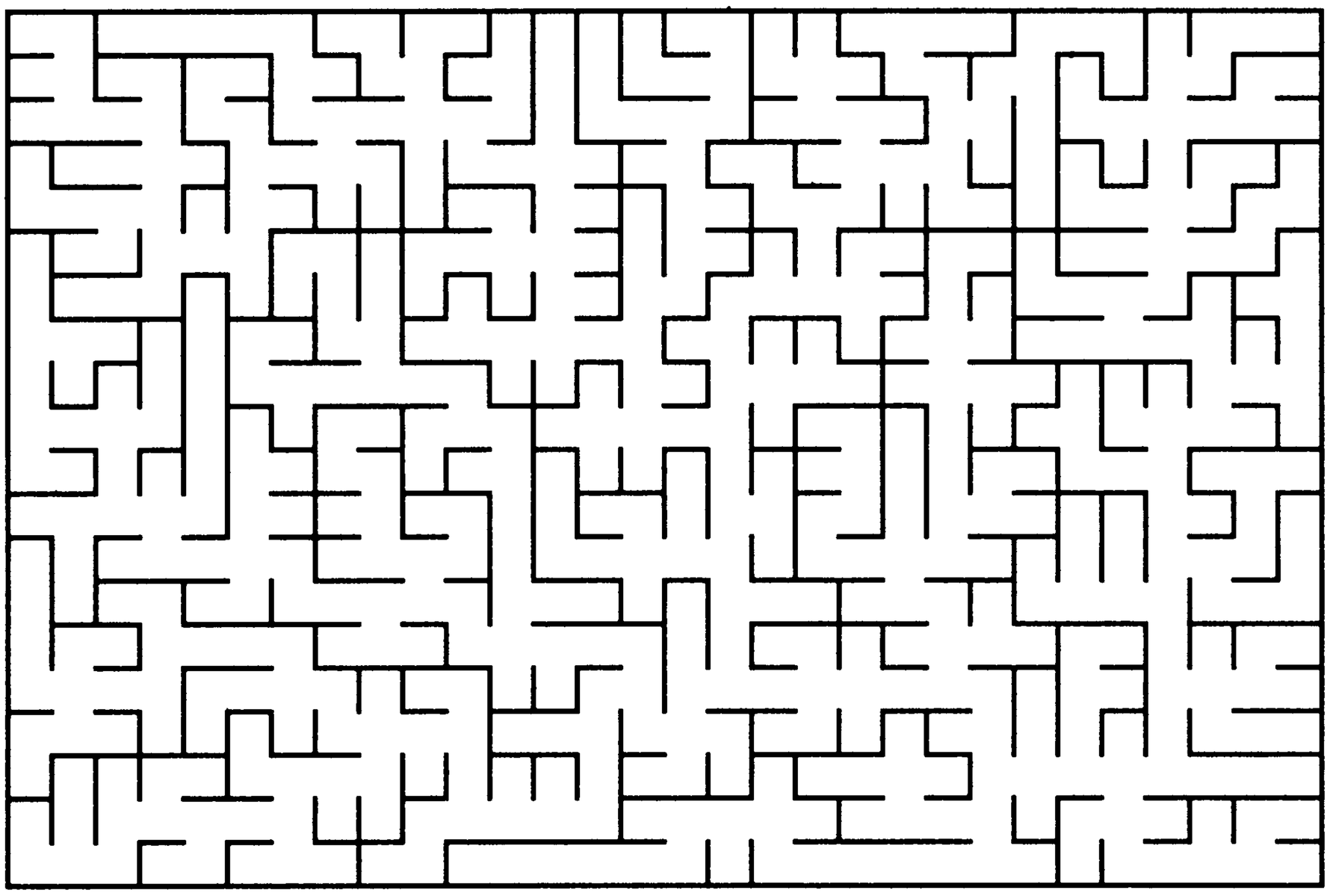


Рис. 4.1. Пример лабиринта

Именно с лабиринтами такого типа мы и будем работать. Следующий вопрос – как представить лабиринт в памяти компьютера. Заманчиво просто создать двумерный массив из нулей и единиц: нуль будет обозначать стену, единица – проход (рис. 4.2).

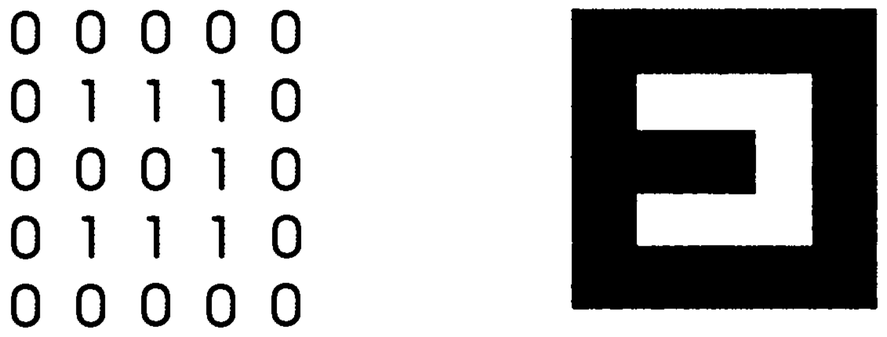


Рис. 4.2. Простое представление лабиринта в памяти

Надо только при выводе на экран рисовать стенки тонкими линиями (я не стал этого делать для наглядности) – и все. Этот способ, в принципе, позволяет описать любой лабиринт «нашего» типа. К сожалению, есть у него и существенный недостаток: если разрушить любую стену (то есть поменять нуль на единицу), в лабиринте образуется новая локация на месте разрушенной стены. Сейчас это, вроде бы, и не имеет значения, однако для алгоритмов генерации лабиринтов, которые мы изучим, подобное поведение недопустимо. При разрушении стены должен открываться проход, но никак не образовываться новая локация. Поэтому нам придется воспользоваться не столь простым и удобным, зато лишенным этого недостатка методом.

В каждой локации лабиринта нас интересует информация о стенах/проходах. В локации может существовать от одной до четырех стен (сверху, снизу, слева и справа). Поэтому разумно задавать локацию записью:

```
type Location = record
    left_wall, right_wall, up_wall, down_wall : Boolean;
end;
```

Если значение поля равно true, значит, соответствующая стена существует, иначе – нет. Обратите внимание, что правая стена любой локации и левая стена локации, находящейся справа от нее, – это одна и та же стена. То же самое справедливо для всех остальных стен – любая из них принадлежит одновременно двум разным локациям. Отсюда вывод: нет нужды хранить в каждой записи информацию обо всех четырех стенах. Достаточно выбрать две: горизонтальную (к примеру, верхнюю) и вертикальную (к примеру, левую). Тогда запись Location станет еще более простой:

```
type Location = record
    left_wall, up_wall : Boolean;
end;
```

Сам лабиринт будет представлять собой двумерный массив таких записей:

```
type Maze = array of array of Location;
```

При этом нельзя забывать о правиле: лабиринт везде по краям должен быть ограничен стеной. Ничего не стоит «построить» стены сверху и слева – для этого надо лишь следить, чтобы значение поля up_wall у всех локаций, расположенных в самом верху лабиринта, и значение поля left_wall у всех локаций, расположенных по левому его краю, были равны true. Справа и снизу лабиринт не ограничить так просто. Помните, мы исключили поля right_wall и down_wall из типа Location? Самое простое решение проблемы заключается в том, чтобы добавить еще ряд локаций справа и внизу лабиринта. Если установить равными true значения полей up_wall и left_wall у этих, дополнительных, локаций, мы получим как раз те стены, которые нам нужны. Думаю, излишние объяснения только усложняют простую идею. Желаете иметь стену справа? Установите значение поля left_wall равным true у локации, расположенной справа от текущей. Требуется стена снизу? Установите значение поля up_wall равным true у локации, расположенной снизу от текущей.

В дальнейшем нам потребуются процедуры загрузки/сохранения лабиринта. Давайте напишем их сейчас.

Поскольку пока еще у нас нет возможности сгенерировать лабиринт автоматически, я решил, что разумно хранить его в текстовом файле – так, по крайней мере, вы сможете вручную создать какой-нибудь тестовый пример. Структура файла будет простой. В первой строке запишем два числа – ширину и высоту лабиринта. Затем в файле последует еще столько же строк, сколько локаций в лабиринте. В любой из этих строк будет храниться пара чисел, каждое из которых равно либо нулю, либо единице. Первое число показывает наличие стены сверху, а второе – стены слева в текущей локации. Локации хранятся последовательно, строка за строкой.

Остается выяснить, что же делать с дополнительными, «служебными» локациями, которые ограничивают лабиринт справа и снизу. Давайте договоримся, что они все-таки не являются частью лабиринта. Если размеры поля равны 10 × 20, то и в файле будут записаны эти числа, а общее количество сохраненных локаций окажется равным 10 × 20 = 200. Реально же в памяти будет храниться матрица 11 × 21, но в файле служебные локации хранить незачем – мы и так прекрасно

знаем, какие стены в них должны существовать (на практике проще всего указать существование обеих стен, чтобы не думать, где именно находится та или иная служебная локация).

Для примера можете воспользоваться определением маленького лабиринта (5 × 4), который я только что набросал на бумаге:

```
5 4
1 1
1 1
1 0
1 1
1 0
0 1
1 0
0 1
0 0
1 1
0 1
1 0
1 0
0 0
0 0
1 1
0 0
0 1
0 1
```

Процедура чтения лабиринта приведена целиком ниже:

```
{ загрузить лабиринт }
procedure LoadMaze(var TheMaze : Maze; FileName : string);
var f              : TextFile;              { файл с описанием лабиринта }
    Height, Width : Integer;               { высота и ширина лабиринта }
    x, y           : Integer;              { текущая локация }
    lw, uw         : Integer;              { временные переменные }
begin
    AssignFile(f, FileName);               { открыть файл }
    Reset(f);

    ReadLn(f, Width, Height);                      { прочитать высоту и ширину }
    SetLength(TheMaze, Width + 1, Height + 1);     { изменить размер лабиринта }

for y := 0 to Height do                            { цикл по всем локациям }
    for x := 0 to Width do
    if (y = Height) or (x = Width) then            { если локация - служебная }
    begin
        TheMaze[x, y].left_wall := true;           { обе стены существуют }
        TheMaze[x, y].up_wall := true;
    end
    else
    begin                                                { иначе считываем }
        ReadLn(f, uw, lw);                               { из файла }
        TheMaze[x, y].left_wall := Boolean(lw);          { прочитанное целое }
        TheMaze[x, y].up_wall := Boolean(uw);            { число надо привести }
```

```
        end;                                          { к типу Boolean }
    CloseFile(f);
end.                                                  { закрыть файл }
```

Обратите внимание на некоторые тонкости. Во-первых, в качестве размеров сторон при вызове SetLength() указываются значения Width + 1 и Height + 1: как мы и договорились, лабиринт будет на одну локацию выше и на одну локацию шире. Во-вторых, реальные индексы в динамических массивах начинаются с нуля (мы уже сталкивались с этим), поэтому первоначальное значение счетчика каждого цикла тоже равно нулю. В-третьих, не забывайте о преобразовании считанного целого к типу Boolean. Вообще говоря, подобные операции не слишком приветствуются, однако сейчас мы точно уверены, что чисел, кроме нуля и единицы, в файле нет.

Процедура записи выглядит немного проще:

```
{ сохранить лабиринт }
procedure SaveMaze(TheMaze : Maze, FileName : string);
var f              : TextFile;                { файл с описанием лабиринта }
    Height, Width : Integer;                 { высота и ширина }
    x, y           : Integer;                { координаты текущей локации }
begin
    AssignFile(f, FileName);                 { открыть файл }
    Rewrite(f);                              { для записи }

    Height := High(TheMaze[0]);              { определяем высоту }
    Width := High(TheMaze);                  { и ширину лабиринта }

    WriteLn(f, Width, ' ', Height);          { запись в файл высоты и ширины }

    for y := 0 to Height - 1 do              { запись данных локаций }
        for x := 0 to Width - 1 do
            WriteLn(f, Integer(TheMaze[x, y].up_wall), ' ',
                       Integer(TheMaze[x, y].left_wall));

    CloseFile(f);                            { закрыть файл }
end;
```

Главная «хитрость» здесь — это определение размеров лабиринта. Не забывайте, что двумерный массив по сути дела является массивом, каждый элемент которого — одномерный массив. Размер этого «супермассива» в данном случае равен ширине лабиринта (потому что мы именно так его описали — как массив, каждый элемент которого есть столбец из локаций). Размер же каждого его элемента (то есть одномерного массива) равен, соответственно, высоте лабиринта. Поэтому, чтобы определить ширину, я использую вызов

```
Width := High(TheMaze);
```

Функция High(), как вы, надеюсь, помните, возвращает наибольший возможный индекс в массиве. Если размер массива равен N, то его индексы лежат в пределах [0...N-1]. Однако мы выделили память для одного лишнего элемента (столбца служебных локаций), поэтому в нашем случае этот диапазон равен [0...N]; High(TheMaze) вернет N, что нам, собственно, и требуется.

Чтобы определить высоту, надо взять любой столбец (то есть любой элемент массива TheMaze) и вызвать для него High(). Я взял нулевой.

При записи в файл служебные локации игнорируются, как было оговорено выше.

Давайте еще напишем процедуру, которая рисует лабиринт на главной форме приложения, чтобы в дальнейшем не отвлекаться на этот процесс. Я предполагаю, что главная форма называется Form1, а на ней расположены объекты Screen и BackBuffer (оба имеют тип TImage). Как и раньше, объект BackBuffer служит для двойной буферизации (не забывайте, он должен быть невидимым). Я использую этот механизм здесь потому, что кое-какая анимация у нас в дальнейшем будет.

Вы, наверное, уже обратили внимание: раньше я использовал элемент TPaintBox для вывода на экран, теперь же решил ограничиться типом TImage. Разница между этими в общем-то похожими компонентами заключается в том, что TImage «запоминает» свой рисунок. Если вы поместите на форму элемент TPaintBox, нарисуете на нем что-нибудь, а потом свернете и снова развернете окно приложения, то рисунок пропадет. Если же использовать TImage, все данные сохранятся. В главе про анимацию я выбрал TPaintBox, потому что подобное «автоматическое сохранение» нам не было нужно, а даром оно не дается, естественно — это все дополнительные затраты времени (а скорость была критическим вопросом). Если же скорость работы не так важна, лучше использовать TImage (хотя, конечно же, готовых рецептов «правильного» подхода не существует).

```
procedure ShowMaze(TheMaze : Maze);     { нарисовать лабиринт }
var x, y           : Integer;
    Height, Width . Integer;            { высота и ширина лабиринта }
begin
    Width := High(TheMaze);             { определить высоту и ширину }
    Height := High(TheMaze[0]);

    with Form1.BackBuffer.Canvas do
    begin                                       { очистка буфера }
        FillRect(Rect(0, 0, Form1.BackBuffer.Width, Form1.BackBuffer.Height));
        for x := 0 to Width - 1 do
            for y := 0 to Height - 1 do
            begin
                { если в локации есть верхняя стена }
                if TheMaze[x, y].up_wall then
                begin
                    MoveTo(x * CellSize, y * CellSize);         { рисуем ее }
                    LineTo((x + 1) * CellSize, y * CellSize);
                end;

                { если в локации есть левая стена }
                if TheMaze[x, y].left_wall then
                begin
                    MoveTo(x * CellSize, y * CellSize);         { рисуем и ее }
                    LineTo(x * CellSize, (y + 1) * CellSize);
                end;
        end;
    MoveTo(0, Height * CellSize);                   { рисуем стену снизу и }
```

```
        LineTo(Width * CellSize, Height * CellSize);  { справа от лабиринта }
        LineTo(Width * CellSize, 0);
    end;
    { отобразить результат на основном экране }
    Form1.Screen.Canvas.CopyRect(Rect(0, 0, Form1.Screen.Width,
        Form1.Screen.Height), Form1.BackBuffer.Canvas,
        Rect(0, 0, Form1.Screen.Width, Form1.Screen.Height));
end;
```

CellSize — это константа, которая определяет размер стороны локации в пикселах. Внешний вид маленького лабиринта, текстовое представление которого приведено выше, показан на рис. 4.3 (разумеется, он получен с помощью процедуры ShowMaze(); значение CellSize равно 50).

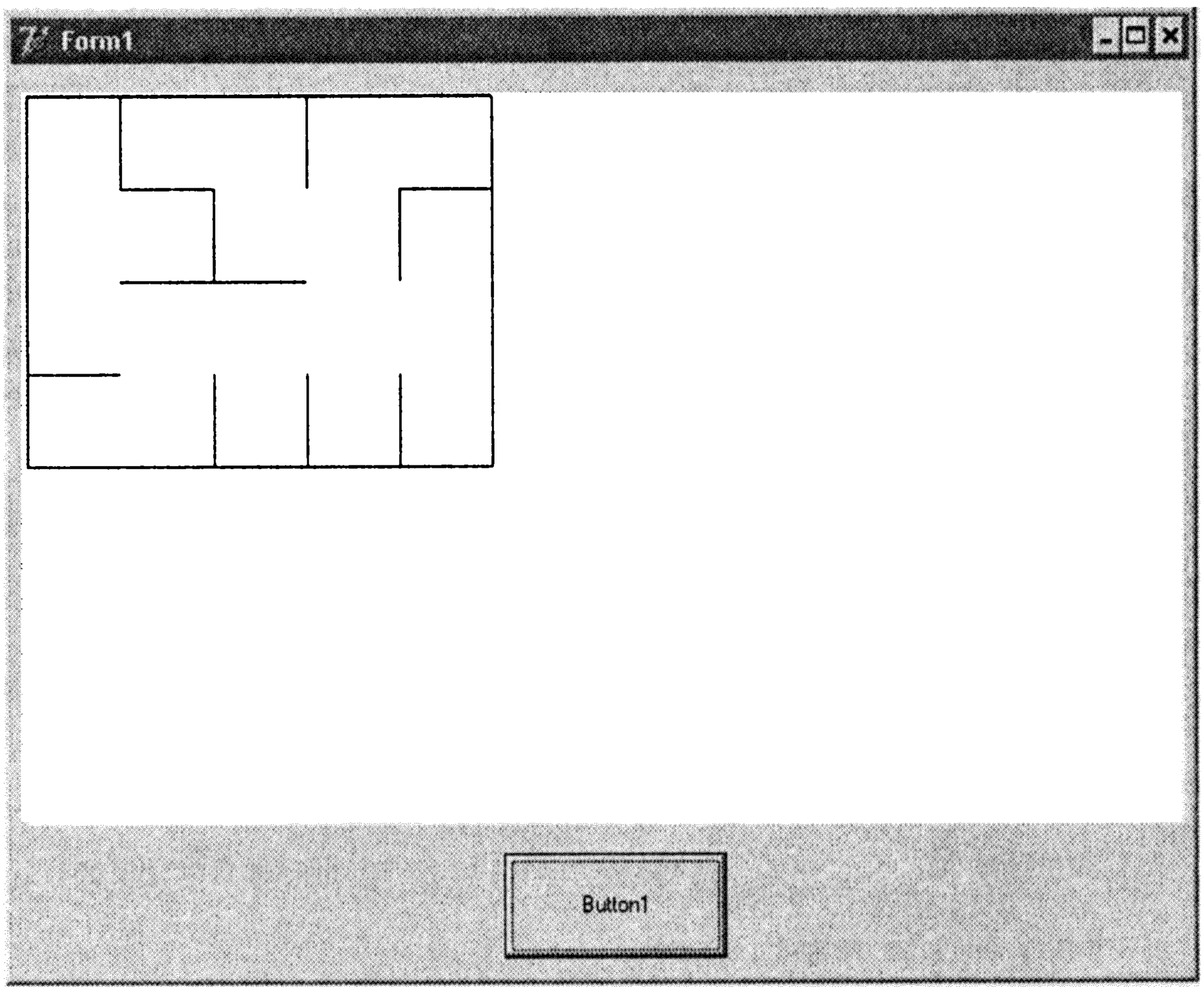


Рис. 4.3. Результат работы процедуры ShowMaze()

Решение лабиринта

Итак, лабиринт создан и загружен в память компьютера. Теперь надо научиться его решать. Под этим термином подразумевается поиск пути из некоторой стартовой локации в некоторую другую (финишную). Стартовая и финишная локации в лабиринте не фиксированы: мы можем попытаться решить лабиринт для любой пары локаций, какой только захотим. Полезно рассмотреть также задачу не просто поиска какого-то пути, а пути самого короткого.

Мы остановимся на двух популярных методах решения лабиринтов: *рекурсивном обходе* и алгоритме *волновой трассировки*.

Рекурсивный обход

По правде говоря, у этого метода есть только одно достоинство – он простой. А еще он очень популярный (как и далеко не лучшая «пузырьковая» сортировка).

Давайте подумаем, как бы стал решать лабиринт человек. Поскольку у нас нет идей, какой путь может вести к финишной локации, ничего не остается делать, кроме как последовательно изучать лабиринт, пока не наткнемся на нее. Если наш путь проходит по коридору, от которого нет ответвлений, надо идти вперед (а больше просто некуда); «интересности» же начинаются, когда мы оказываемся на перекрестке. Имеет смысл поступить следующим образом. Отметим какой-нибудь из возможных путей (московское время такое-то, выбрали такой-то путь) и будем двигаться по нему. Если нам придется вернуться (зачем – я скажу позже), мы можем выбрать уже другой путь.

Если на пути встретилась финишная локация – замечательно, конец алгоритма. При этом надо не забывать отмечать свой путь (в «бортовом журнале»), чтобы знать, каким именно образом мы сюда попали.

К сожалению, гораздо чаще, чем хотелось бы, мы будем встречаться не с финишной локацией, а с тупиком. Тупик бывает двух видов: либо просто некуда идти (идем по коридору, а в конце ждет глухая стена), либо там, куда можно идти, мы уже были. Второе означает, что в маршруте образовалась петля, а никакого смысла в том, чтобы идти по тем локациям, где уже все изучено, нет. В этом случае проще всего сделать шаг назад, забыв о том, что мы посетили текущую локацию (вычеркнуть ее из «бортового журнала»), и выбрать другой путь. Если шаг назад ничего не дает, надо сделать еще один шаг назад, а если понадобится – и еще несколько, до тех пор, пока не появятся альтернативы. Если все варианты исчерпаны, а решение так и не найдено, это означает, что его попросту не существует. Например, финишная локация полностью окружена сплошной стеной.

Вот и весь алгоритм. Может, кто-нибудь вспомнит о простом правиле: если хочешь выйти из лабиринта, всегда, что бы ни случилось, держись рукой правой стены (можно и левой, конечно; главное – не менять решение в середине пути). К сожалению, это правило гарантирует только то, что мы вернемся туда, откуда пришли; при этом существенная часть лабиринта может остаться неисследованной.

Давайте сначала рассмотрим процедуру рекурсивного обхода (листинг 4.1), а потом я буду ее критиковать.

Листинг 4.1. Рекурсивный обход лабиринта

```
procedure RecursiveSolve(TheMaze : Maze; xs, ys, xf, yf : Integer);
var Visited       : array of array of Boolean;  { карта посещенных локаций }
    x, y, xc, yc  : Integer;
    i             : Integer;
    Path          : array of TPoint;            { результирующий маршрут }
    Height, Width : Integer;
const dx : array[1..4] of Integer = (1, 0, -1, 0);  { смещения }
      dy : array[1..4] of Integer = (0, -1, 0, 1);
{ служебная функция: определяет, можно ли пройти из локации
```

продолжение

Листинг 4.1 (продолжение)

```
    (x, y) в локацию (x + dx, y + dy), то есть нет ли между ними стены }
function CanGo(x, y, dx, dy : Integer) : Boolean;
begin
    if dx = -1 then CanGo := not TheMaze[x, y].left_wall
    else if dx = 1 then CanGo := not TheMaze[x + 1, y].left_wall
    else if dy = -1 then CanGo := not TheMaze[x, y].up_wall
    else CanGo := not TheMaze[x, y + 1].up_wall;
end;

{ поиск финишной локации из точки (x, y) }
function Solve(x, y, depth : Integer) : Boolean;
var i : Integer;
begin
    Visited[x, y] := true;                  { пометить локацию как посещенную }
    Path[depth] := Point(x, y);             { добавить ее в описание маршрута }
    Path[depth + 1] := Point(-1, -1);       { добавить признак конца маршрута }

    if (x = xf) and (y = yf) then           { если финишная локация найдена }
    begin
        Solve := true;                      { конец алгоритма }
        Exit;
    end;

    for i := 1 to 4 do
        { если дорожка свободна, идем по ней }
        if CanGo(x, y, dx[i], dy[i]) and
                                        not Visited[x + dx[i], y + dy[i]] then
            if Solve(x + dx[i], y + dy[i], depth + 1) then
            begin
                Solve := true;                  { если решение найдено }
                Exit;                           { конец алгоритма }
            end;

    Visited[x, y] := false;                 { пометить локацию как непосещенную }
    Solve := false;                         { решение не найдено }
end;

begin                                               { главная процедура }
    Width := High(TheMaze);
    Height := High(TheMaze[0]);
    SetLength(Path, Height * Width + 1);    { выделяем память для маршрута }
    SetLength(Visited, Width, Height);      { и для списка посещенных локаций }

    for x := 0 to Width - 1 do
        for y := 0 to Height - 1 do
            Visited[x, y] := false;         { изначально ни одна не посещена }

    if Solve(xs, ys, 0) then                { если найдено решение, рисуем его }
    begin
        i := 0;
        while not ((Path[i].X = -1) and (Path[i].Y = -1)) do
        begin
            xc := CellSize * (2 * Path[i].X + 1) div 2;
```

```
            yc := CellSize * (2 * Path[i].Y + 1) div 2;
            Form1.Screen.Canvas.Ellipse(xc - 5, yc - 5, xc + 5, yc + 5);
            i := i + 1;
        end;
    end;
end;
```

На первый взгляд процедура кажется довольно громоздкой, но если присмотреться, то становится ясно, что ядро у нее не такое уж большое.

Для реализации процедуры мне потребовались две дополнительные функции: CanGo() и Solve(). Функция CanGo() определяет, есть или нет стены между локациями (x, y) и (x + dx, y + dy) (то есть можно ли пройти из одной в другую). При этом предполагается, что локации находятся по соседству – первая расположена сверху, слева, снизу или справа от второй. Функция Solve() более интересна и сложна. Ей передается три параметра – координаты текущей локации и номер шага. Номер нужен для того, чтобы знать, в какую строчку «бортового журнала» записывать текущее движение. Дальше все происходит по алгоритму, о котором мы говорили выше:

```
пометить текущую локацию как посещенную
добавить ее в «бортовой журнал»...
```

Обратите внимание на «признак конца». Нам как-то надо отмечать конец журнала. Поскольку локации с координатами (-1, -1) не существует, эту «псевдолокацию» вполне можно использовать.

```
если локация найдена – конец алгоритма
исследуем каждую свободную дорожку
```

Вот здесь я применил небольшую хитрость: всего из каждой локации может вести максимум четыре разные дорожки. Первая начинается от соседней сверху локации, вторая – от левой, третья – от нижней, а четвертая – от правой. Чтобы не писать четыре раза

```
исследовать_дорожку(x, y - 1)
исследовать_дорожку(x - 1, y)
исследовать_дорожку(x, y + 1)
исследовать_дорожку(x + 1, y)
```

я воспользовался циклом (не забывайте, что исследовать_дорожку() – это не вызов процедуры, а фрагмент из нескольких строк кода, поэтому экономия заметная):

```
for i := 1 to 4 do
    исследовать_дорожку(x + dx[i], y + dy[i])
```

Если решение найдено, то работа алгоритма заканчивается; в противном случае текущая локация помечается как непосещенная и происходит выход из процедуры.

Обратите внимание, что каждый шаг в лабиринте – это рекурсивный вызов. Такое решение позволяет легко вернуться на шаг назад (помните, это важная часть алгоритма) – для этого надо всего лишь выйти из функции, сообщив при этом вызывающей функции, что решение не найдено.

Главная процедура делает не так уж и много работы: она инициализирует массивы, затем вызывает Solve() и рисует на экране полученное решение (рис. 4.4).

В примере я выбрал верхнюю левую локацию лабиринта как стартовую, а верхнюю правую — как финишную.

Обратите также внимание на строку:

```
if CanGo(x, y, dx[i], dy[i]) and
                                not Visited[x + dx[i], y + dy[i]] then ...
```

Если текущая локация находится на краю или в углу лабиринта, числа x + dx[i] и y + dy[i] могут оказаться некорректными индексами массива Visited (то есть попросту указывать на локацию, лежащую за границами лабиринта). Ошибки не возникает потому, что Delphi по умолчанию использует так называемую сокращенную схему вычисления логических выражений: если значение CanGo() оказалось равно false, нет нужды вычислять правую часть выражения (после and) — и так ясно, что значение всего выражения будет равно false. Если функция CanGo() определит, что мы пытаемся выйти за пределы лабиринта, она, естественно, вернет false, поскольку весь лабиринт ограничен стеной; следовательно, правая часть выражения в таких случаях никогда не будет вычисляться. Мы будем пользоваться этой особенностью Delphi и в дальнейшем.

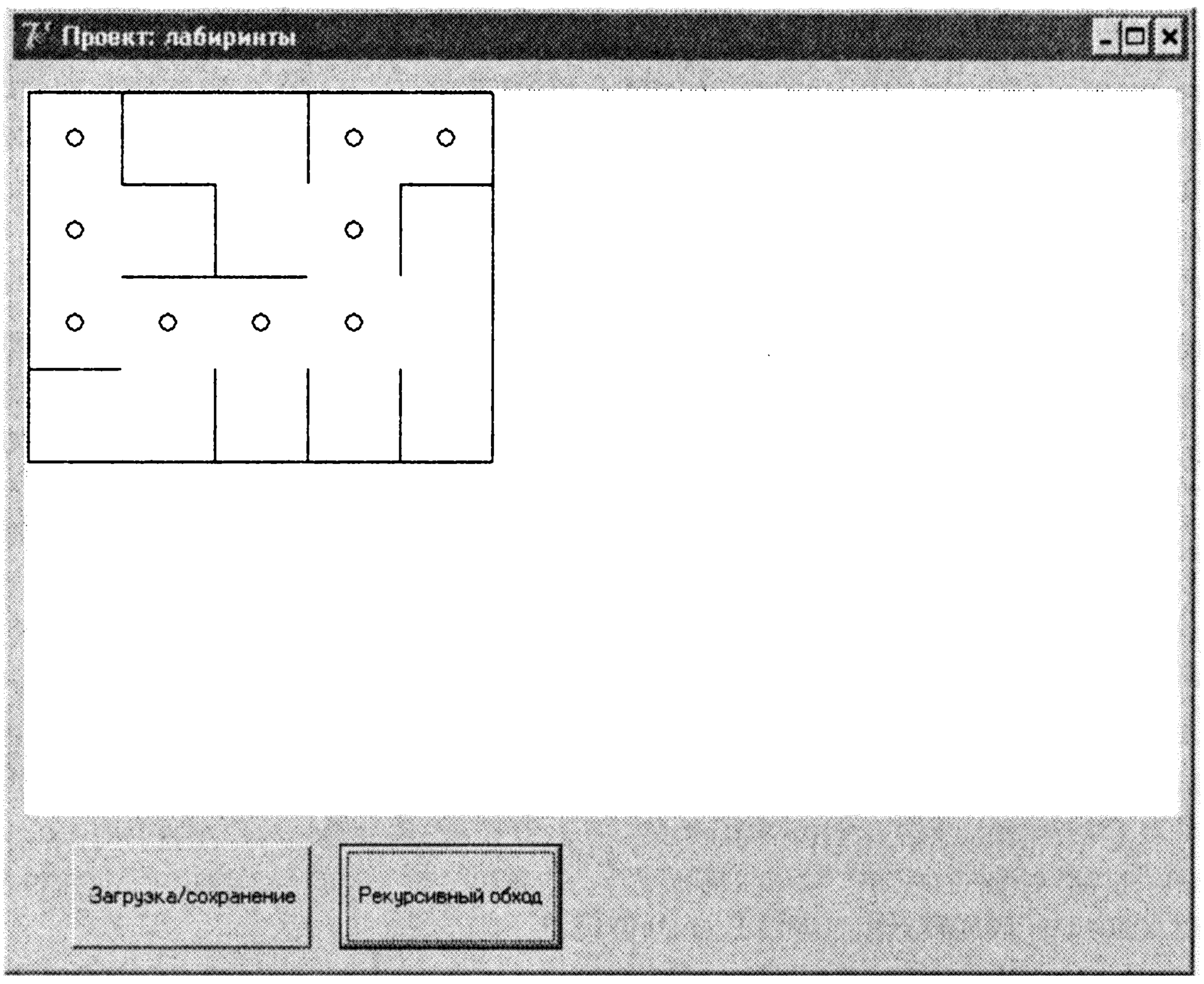


Рис. 4.4. Решение, найденное с помощью рекурсивного обхода

Надеюсь, принцип работы рекурсивного обхода более или менее ясен. Возможно, стоило подробнее на нем остановиться и постараться объяснить все тонкости более доходчиво (если алгоритм рекурсивен, он уже не слишком прост — по крайней мере, для меня), однако мне просто жаль на него времени. Сейчас я подробно распишу недостатки рекурсивного обхода, а потом мы рассмотрим куда более передовой алгоритм волновой трассировки.

Итак, два недостатка алгоритма лежат на поверхности:

1) он обходит лабиринт нерационально: может пройти достаточное количество времени, прежде чем решение будет найдено;
2) полученное решение может не быть оптимальным (в примере алгоритм находит оптимальное решение, но в приведенном лабиринте другого решения просто нет).

Третий недостаток гораздо более существенный, хотя и проявляется он не всегда. Посмотрите на лабиринт, изображенный на рис. 4.5.

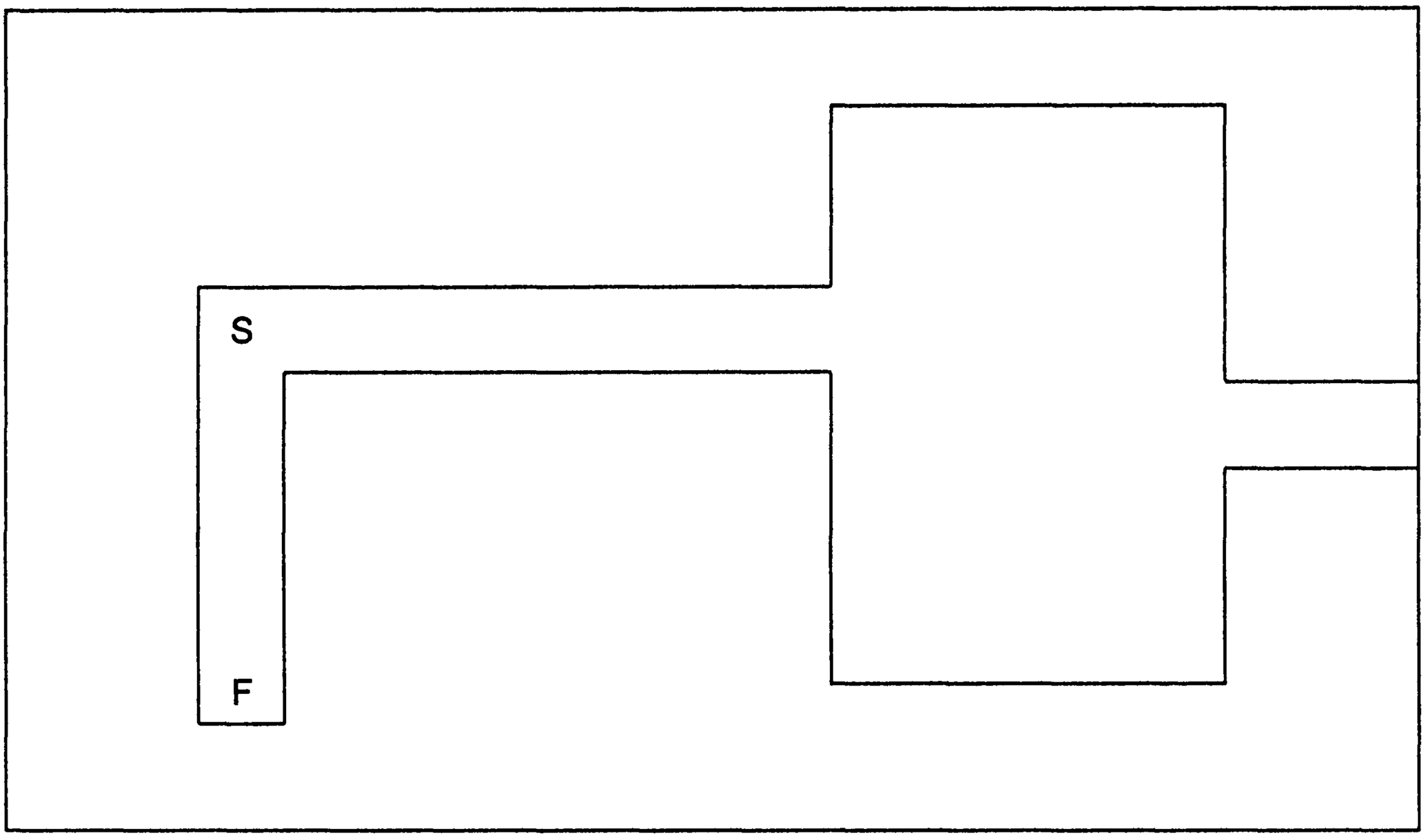


Рис. 4.5. Лабиринт, проблематичный для процедуры рекурсивного обхода

Проход, ведущий вниз от стартовой локации, заканчивается финишной локацией. Если же пойти направо, мы сначала попадем в «зал» — группу локаций безо всяких стен между ними, а затем дойдем до тупика.

Если рекурсивный алгоритм запрограммирован так, что дорожка, ведущая вниз, будет рассмотрена первой — замечательно, решение будет найдено, причем очень быстро. Если же алгоритм сначала пойдет направо, у нас будут большие проблемы. Мы знаем, что движение вправо бесперспективно: в конце ждет тупик. Тем не менее алгоритм устроен так, что отступление идет до ближайшего шага, на котором возможны альтернативы (помните?). Так вот: каждая клетка «зала» — это перекресток из четырех дорожек! На каждом перекрестке алгоритм последовательно постарается перебрать все варианты движений, наивно полагая, что на сей раз повезет, и тупика не будет. Таким образом, в итоге будут перебраны *все* возможные варианты пересечения «зала»: прямо, по змейке, вдоль стены... только представьте себе масштабы подобного перебора! На перекрестке четырех дорожек у вас есть три варианта, куда пойти (поскольку назад идти нельзя). На каждой из трех новых локаций у вас есть опять же по три варианта на выбор (итого 3*3 = 9); на следующем этапе это число уже будет равно 3*3*3 = 27 и т. д. Количество вариантов растет в геометрической прогрессии, что делает рекурсивный

обход совершенно непригодным для лабиринтов с подобными «залами»: программа попросту «зависнет», просчитывая мириады бесперспективных маршрутов.

Алгоритм волновой трассировки

Описание рекурсивного обхода я начал с фразы: «Давайте подумаем, как бы стал решать лабиринт человек». Разумеется, копирование человеческих действий — не единственный способ достижения решения; вполне можно применить и другие модели поведения. К примеру, такую: представьте, что в стартовой локации мы опрокинули бочку воды (а еще лучше, густого киселя). Жидкость начинает растекаться по сторонам, постепенно добираясь даже до самых отдаленных локаций лабиринта. Рано или поздно она достигнет и финишной локации: в этом случае надо проследить, каким путем жидкость туда попала — а это и будет маршрут от старта до финиша (причем, заметьте, кратчайший!). Если киселю уже некуда течь, а финишная локация так и не достигнута, это означает, что решения не существует. Моделированием такого поведения занимается алгоритм волновой трассировки [1].

Пометим сначала все локации лабиринта нулями (что означает «локация не содержит киселя»). Стартовую локацию пометим единицей (вылили кисель). Теперь выполняем действия:

```
найти в лабиринте локации, помеченные единицами[2]
для каждой из четырех соседних с ней локаций проверить два условия:
    1) помечена ли она нулем
    2) есть ли стена между двумя локациями (выбранной и соседней)
если оба условия выполнены, помечаем соседнюю локацию двойкой
```

Рисунок 4.6 иллюстрирует сказанное.

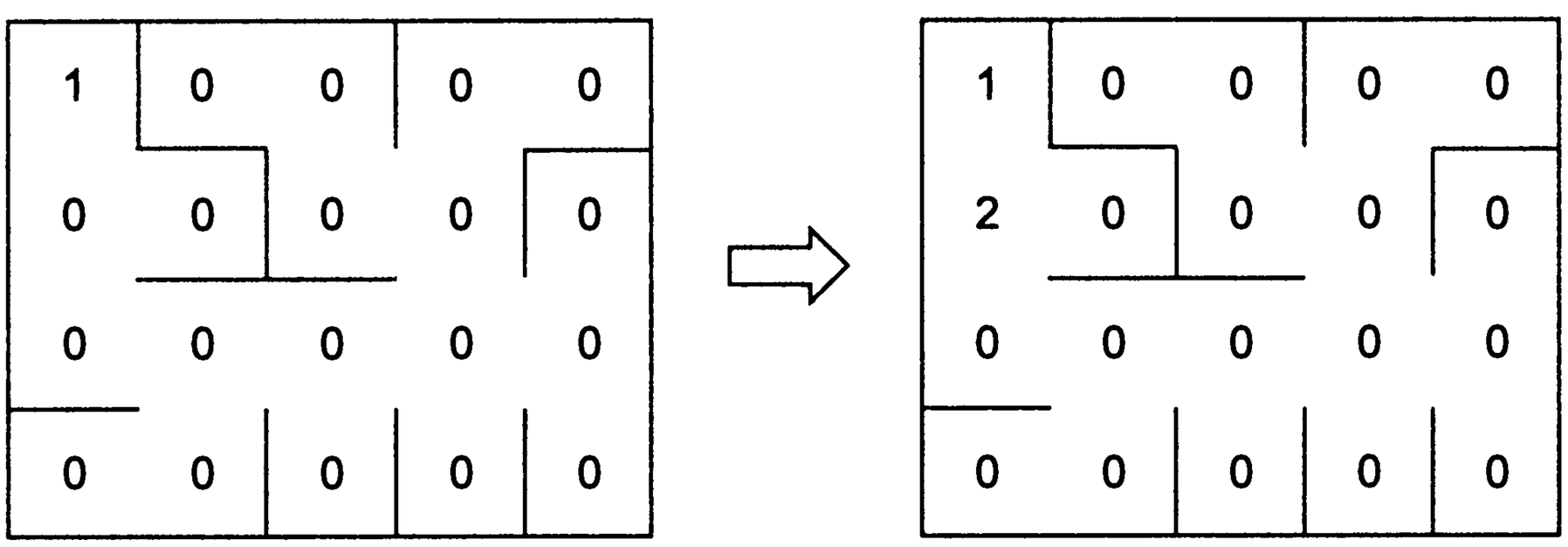


Рис. 4.6. Первая итерация алгоритма волновой трассировки

Из стартовой позиции можно попасть лишь в локацию, расположенную снизу от нее. Поскольку она помечена нулем, записываем двойку.

Вторая итерация алгоритма выглядит так:

---

[1] Название намекает на похожую идею: представьте, что весь лабиринт заполнен водой. Вы бросаете в стартовой локации камешек в воду, а от него расходятся волны во все стороны.

[2] Вообще говоря, единицей помечена только стартовая локация, но для других чисел (не-единиц) это уже будет неверно — см. далее.

```
найти в лабиринте локации, помеченные двойками
для каждой из четырех соседних с ней локаций проверить те же условия
если оба условия выполнены, помечаем соседнюю локацию тройкой
```

Надеюсь, общая идея алгоритма ясна. На N-й итерации нам придется выполнить действия:

```
найти в лабиринте локации, помеченные числом N
для каждой из четырех соседних с ней локаций проверить те же условия
если оба условия выполнены, помечаем соседнюю локацию числом N + 1
```

Результат работы алгоритма после восьмой итерации показан на рис. 4.7.

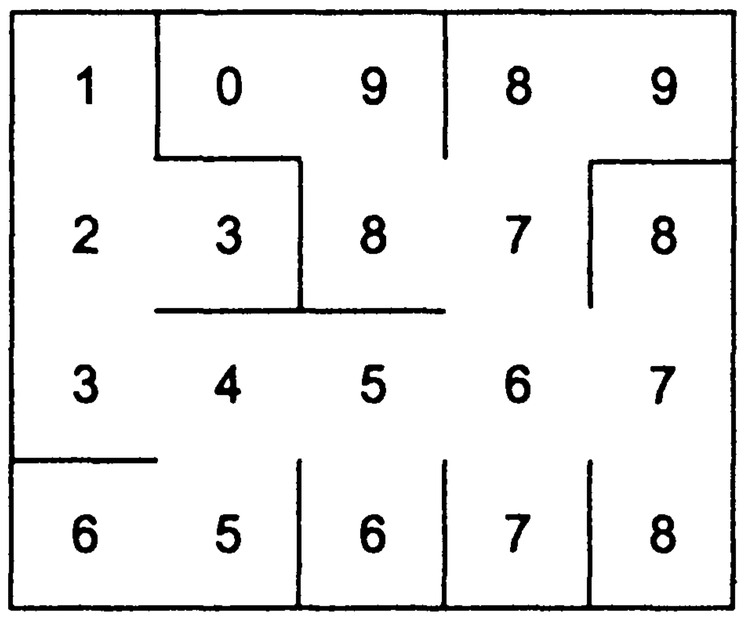


Рис. 4.7. Результат работы алгоритма волновой трассировки

Если на какой-то итерации мы достигли финишной локации (я считаю финишной правую верхнюю локацию лабиринта), работа алгоритма заканчивается. Если в течение итерации мы не сумели занять ни одной новой клетки, решения не существует.

Если решение найдено на N-й итерации (в нашем случае — на восьмой), финишная локация помечена числом N + 1. Теперь осталось лишь определить собственно путь. Сделать это несложно: в финишную локацию (номер N + 1) мы попали из той соседней с ней локации, которая имеет номер N; в свою очередь, в нее можно попасть из локации с номером N – 1 и т. д. Если в процессе определения пути мы нашли две локации, откуда можно было попасть в текущую, можно выбрать любую из них — оба маршрута будут оптимальными. Разумеется, надо следить, чтобы между соседними локациями маршрута не было стены.

Давайте сначала реализуем этот алгоритм, а потом обсудим его сильные и слабые стороны.

Листинг 4.2. Алгоритм волновой трассировки

```
procedure WaveTracingSolve(TheMaze : Maze; xs, ys, xf, yf : Integer);
var Mark           : array of array of Integer;  { метки локаций }
    x, y, xc, yc   : Integer;
    N, i           : Integer;
    Height, Width  : Integer;
const dx : array[1..4] of Integer = (1, 0, -1, 0);  { смещения }
      dy : array[1..4] of Integer = (0, -1, 0, 1);

{ служебная функция: определяет, можно ли пройти из локации
 (x, y) в локацию (x + dx, y + dy), то есть нет ли между ними стены }

function CanGo(x, y, dx, dy : Integer) : Boolean;
```

продолжение ⇩

Листинг 4.2 (продолжение)

```
    begin
        if dx = -1 then CanGo := not TheMaze[x, y].left_wall
        else if dx = 1 then CanGo := not TheMaze[x + 1, y].left_wall
        else if dy = -1 then CanGo := not TheMaze[x, y].up_wall
        else CanGo := not TheMaze[x, y + 1].up_wall;
    end;

    function Solve : Boolean;                          { поиск решения }
        var i, N, x, y : Integer;
            NoSolution : Boolean;
    begin
        N := 1;                                        { начинаем с итерации номер 1 }

        repeat
            NoSolution := true;          { пессимистично полагаем, что решения нет }
            for x := 0 to Width - 1 do
                for y := 0 to Height - 1 do
                    if Mark[x, y] = N then  { найти локации, помеченные числом N }
                        for i := 1 to 4 do   { просмотр соседних локаций }
                            if CanGo(x, y, dx[i], dy[i]) and
                               (Mark[x + dx[i], y + dy[i]] = 0) then
                            begin              { локация доступна и помечена нулем }
                                NoSolution := false;  { есть шанс найти решение }
                                { помечаем соседнюю локацию числом N + 1 }
                                Mark[x + dx[i], y + dy[i]] := N + 1;
                                if (x + dx[i] = xf) and (y + dy[i] = yf) then
                                begin
                                    Solve := true;   { дошли до финишной локации }
                                    Exit;            { конец алгоритма }
                                end;
                            end;
            N := N + 1;                              { переход к следующей итерации }
        until NoSolution;     { повторять, если есть надежда найти решение }

        Solve := false;      { нет, решение не найдено }
    end;

    begin
        Width := High(TheMaze);
        Height := High(TheMaze[0]);
        SetLength(Mark, Width, Height);    { выделение памяти для пометок }

        for x := 0 to Width - 1 do         { изначально все заполняется нулями }
            for y := 0 to Height - 1 do
                Mark[x, y] := 0;

        Mark[xs, ys] := 1;                 { стартовой локации соответствует единица }
        if Solve then                      { если найдено решение, рисуем его }
        begin
            x := xf; y := yf;
            for N := Mark[xf, yf] downto 1 do
            begin
                { рисуем окружность на очередной локации маршрута }
```

```
xc := CellSize * (2 * x + 1) div 2;
yc := CellSize * (2 * y + 1) div 2;
Form1.Screen.Canvas.Ellipse(xc - 5, yc - 5, xc + 5, yc + 5);

for i := 1 to 4 do
    if CanGo(x, y, dx[i], dy[i]) and
       (Mark[x + dx[i], y + dy[i]] = N - 1) then
    begin
        x := x + dx[i];   { ищем следующую локацию маршрута }
        y := y + dy[i];
        Break;
                end;
            end;
        end;
    end;
```

Главная часть процедуры очень похожа на аналогичный фрагмент из алгоритма рекурсивного обхода: инициализация переменных, вызов функции поиска решения, вывод найденного решения на экран. Функция `CanGo()` идентична предыдущей реализации, а функция `Solve()` прямолинейно реализует алгоритм, который я набросал выше.

Итак, поговорим теперь о качествах алгоритма волновой трассировки. Его плюсы налицо: он простой (наверное, даже проще рекурсивного обхода), отлично справляется с залами и находит оптимальное решение (причем очень быстро). Есть у него и существенный минус: на каждой итерации приходится исследовать весь лабиринт целиком, определяя, каким числом помечена та или иная локация. Если мы имеем дело с большим лабиринтом, этот недостаток может быстро стать критическим. Другой недостаток – большой расход памяти: приходится содержать двумерный массив с метками локаций. В принципе, при рекурсивном обходе мы тоже использовали подобный массив, чтобы определить, была ли посещена локация раньше или нет; однако здесь ситуация другая. В случае рекурсивного обхода можно просто хранить координаты посещенных локаций в списке, экономя тем самым память, сейчас же такая уловка не поможет: количество меток очень быстро разрастается, покрывая существенную часть всего лабиринта.

О том, как улучшить алгоритм волновой трассировки, я расскажу в конце главы, а пока займемся генерацией лабиринтов.

Генерация лабиринтов

Эта часть главы посвящена тому, как научить компьютер создавать лабиринты автоматически, без участия человека. Мы рассмотрим два алгоритма: Прима и Краскала. Оба они работают вполне удовлетворительно, тем не менее алгоритм Краскала считается более совершенным (в том плане, что создает более запутанные лабиринты), но работает он медленнее.

Лабиринты, сгенерированные каждым из этих алгоритмов, обладают двумя свойствами: во-первых, они не содержат залов, а во-вторых, из любой локации лаби-

ринта можно попасть в любую другую (то есть не существует замкнутых областей, отделенных от остальных частей лабиринта).

Алгоритм Прима

Создадим «заготовку» — лабиринт, в котором все локации полностью окружены стенами (разумеется, далеко от стартовой точки в таком лабиринте не уйдешь). Сопоставим каждой локации переменную-атрибут (соответственно, у нас получится двумерный массив атрибутов), которая может принимать значения Inside (внутри), Outside (снаружи) и Border (на границе). Изначально атрибут каждой локации должен быть равен Outside. Выберем случайную локацию в лабиринте и присвоим ее атрибуту значение Inside. Присвоим также атрибутам соседних с ней локаций значение Border. Теперь надо действовать по алгоритму:

```
ПОКА атрибут хотя бы одной локации равен Border
    выберем случайную локацию, атрибут которой равен Border, и присвоим ей
    атрибут Inside
    изменим на Border атрибут всех соседних с текущей локаций, атрибут которых
    равен Outside
    из всех соседей текущей локации, атрибут которых равен Inside, выберем
    случайную и разрушим стену между ней и текущей локацией
```

В последнем действии предполагается, что такие соседи (имеющие атрибут, равный Inside) у текущей локации имеются. Почему? А потому, что атрибут текущей локации изначально был равен Border — это гарантирует выполнение условия. По той же причине между такими локациями (текущей, атрибут которой был равен Border и соседней — с атрибутом Inside) всегда есть стена.

Я понимаю, эти утверждения не так очевидны; на самом деле они вытекают из принципа работы алгоритма — попробуйте сделать несколько шагов вручную, на бумаге, и вы убедитесь в их справедливости. Если какая-то клетка имеет атрибут Inside, это означает, что она принадлежит к «внутренней», уже построенной области лабиринта. Любые две Inside-локации косвенно связаны между собой (то есть существует маршрут между ними). Outside-локации никак не связаны с Inside-локациями — их еще только предстоит обработать. Border-локации тоже находятся извне лабиринта, но каждая из них граничит хотя бы с одной Inside-локацией.

Листинг 4.3 содержит реализацию алгоритма Прима (далеко не лучшую в плане быстродействия, но прямолинейную и понятную).

Листинг 4.3. Генерация лабиринта по алгоритму Прима

```
function PrimGenerateMaze(Width, Height : Integer) : Maze;
type  AttrType = (Inside, Outside, Border);      { тип "атрибут локации" }
var   TheMaze    : Maze;                         { сам лабиринт }
      x, y, i    : Integer;
      xc, yc     : Integer;
      xloc, yloc : Integer;
      Attribute  : array of array of AttrType;  { карта атрибутов }
      IsEnd      : Boolean;
      counter    : Integer;
const dx : array[1..4] of Integer = (1, 0, -1, 0);  { смещения }
```

```
      dy : array[1..4] of Integer = (0, -1, 0, 1);

label ExitFor1, ExitFor2, ExitFor3;          { используемые метки }

procedure BreakWall(x, y, dx, dy : Integer);              { разрушить стену }
begin                                                     { между локациями }
    if dx = -1 then TheMaze[x, y].left_wall := false
    else if dx = 1 then TheMaze[x + 1, y].left_wall := false
    else if dy = -1 then TheMaze[x, y].up_wall := false
    else TheMaze[x, y + 1].up_wall := false;
end;
```

```
begin
    SetLength(Attribute, Width, Height);    { выделение памяти для атрибутов }
    SetLength(TheMaze, Width + 1, Height + 1);   { изменить размер лабиринта }

    for x := 0 to Width - 1 do              { изначально все атрибуты }
        for y := 0 to Height - 1 do         { равны Outside }
            Attribute[x, y] := Outside;

    for y := 0 to Height do                       { все стены изначально }
        for x := 0 to Width do                    { существуют }
        begin
            TheMaze[x, y].left_wall := true;
            TheMaze[x, y].up_wall := true;
        end;
```

```
    Randomize;
    x := Random(Width);                           { выбираем начальную локацию }
    y := Random(Height);
    Attribute[x, y] := Inside;                    { и присваиваем ей атрибут Inside }

    for i := 1 to 4 do                            { всем ее соседям присваиваем }
    begin                                         { атрибут Border }
        xc := x + dx[i];
        yc := y + dy[i];
        if (xc >= 0) and (yc >= 0) and (xc < Width) and (yc < Height) then
            Attribute[xc, yc] := Border;
    end;

    repeat                                                { главный цикл }
        IsEnd := true;
        counter := 0;
        for x := 0 to Width - 1 do                  { подсчитываем количество }
            for y := 0 to Height - 1 do       { локаций с атрибутом Border }
                if Attribute[x, y] = Border then counter := counter + 1;

        counter := Random(counter) + 1;                 { выбираем из них }
        for x := 0 to Width - 1 do                      { одну случайную }
            for y := 0 to Height - 1 do
                if Attribute[x, y] = Border then
                begin
                    counter := counter - 1;
                    if counter = 0 then
```

продолжение ⇩

Листинг 4.3 (продолжение)

```
                    begin
                        xloc := x;              { xloc, yloc - ее координаты }
                        yloc := y;
                        goto ExitFor1;          { выход из цикла }
                    end;
                end;
ExitFor1:
        Attribute[xloc, yloc] := Inside;      { присвоить ей атрибут Inside }

        counter := 0;
        for i := 1 to 4 do
        begin
            xc := xloc + dx[i];
            yc := yloc + dy[i];
            if (xc >= 0) and (yc >= 0) and (xc < Width) and (yc < Height) then
            begin        { подсчитать количество локаций с атрибутом Inside }
                if Attribute[xc, yc] = Inside then counter := counter + 1;
                if Attribute[xc, yc] = Outside then  { заменить атрибуты с }
                    Attribute[xc, yc] := Border;     { Outside на Border }
            end;
        end;

        counter := Random(counter) + 1;   { выбрать случайную Inside-локацию }
        for i := 1 to 4 do
        begin
            xc := xloc + dx[i];
            yc := yloc + dy[i];
            if (xc >= 0) and (yc >= 0) and (xc < Width) and (yc < Height)
               and (Attribute[xc, yc] = Inside) then
            begin
                counter := counter - 1;
                if counter = 0 then          { разрушить стену между ней и }
                begin                        { текущей локацией }
                    BreakWall(xloc, yloc, dx[i], dy[i]);
                    goto ExitFor2;
                end;
            end;
        end;

ExitFor2:
        for x := 0 to Width - 1 do                { определить, есть ли }
            for y := 0 to Height - 1 do           { хоть одна локация с }
                if Attribute[x, y] = Border then  { атрибутом Border }
                begin
                    IsEnd := false;               { если да, продолжаем }
                    goto ExitFor3;                { выполнять алгоритм }
                end;
ExitFor3:
        ShowMaze(TheMaze);                    { отобразить процесс генерации }
        Application.ProcessMessages;
    until IsEnd;
    PrimGenerateMaze := TheMaze;
end;
```

Я добавил в конец процедуры вызов ShowMaze(), чтобы отображать в динамике процесс генерации лабиринта — очень интересное зрелище на самом деле (рис. 4.8).

В алгоритме постоянно используется идея того, как можно выбрать случайную локацию, помеченную некоторым атрибутом:

```
N := количество локаций, помеченных заданным атрибутом
n := случайное число от 1 до N
выбрать n-ю по счету локацию, помеченную заданным атрибутом
```

Этот метод прост (этим и хорош), зато требует для работы двух циклов, каждый из которых пробегает по всему лабиринту.

В трех местах я применил оператор goto — надеюсь, никто не считает дурным тоном его использование для выхода из вложенного цикла?

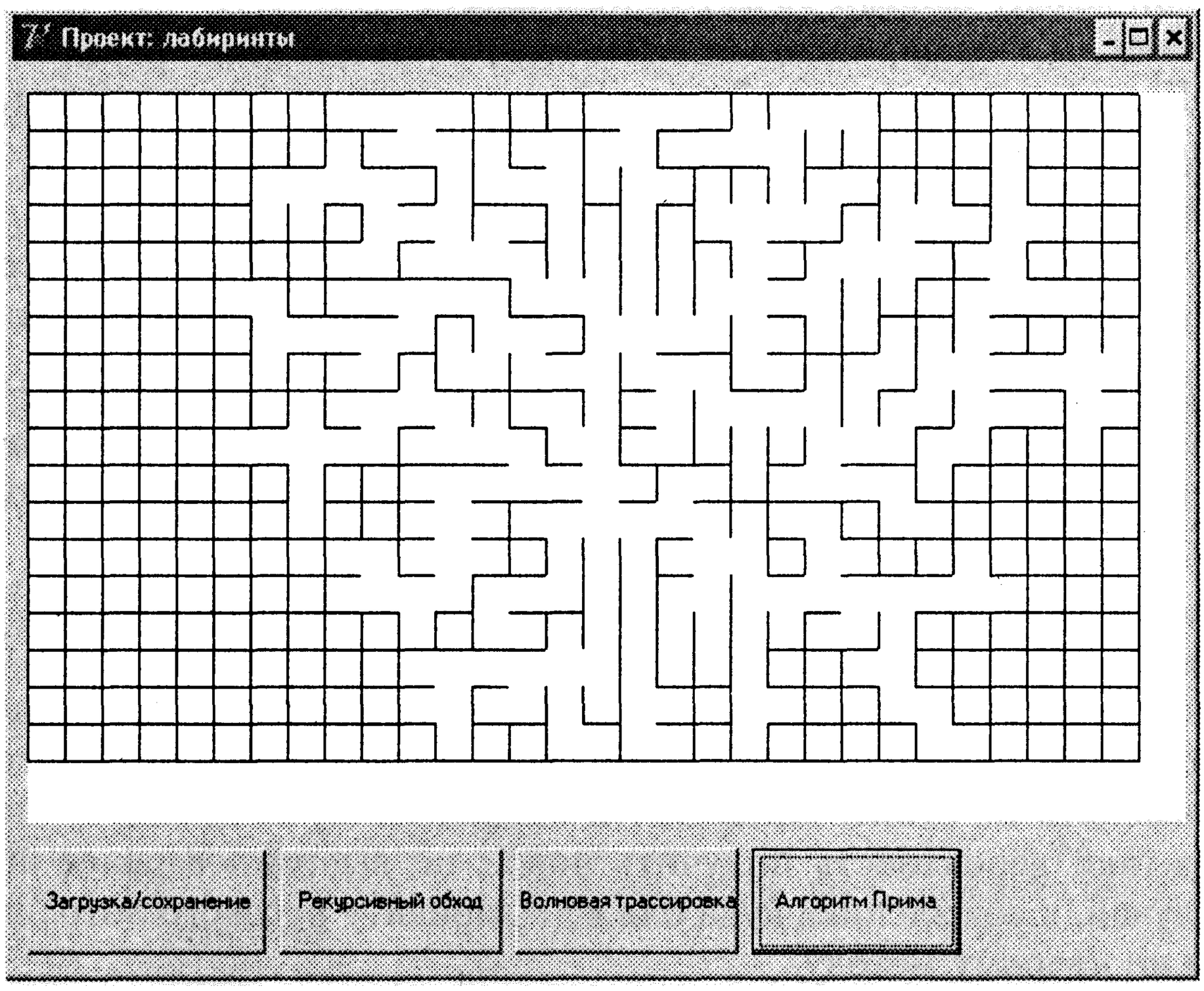


Рис. 4.8. Алгоритм Прима в процессе работы

Алгоритм Краскала

Прежде всего, создадим заготовку, аналогичную той, что мы использовали в алгоритме Прима — лабиринт со всеми возможными стенами. Алгоритм Краскала описывается всего семью строками на псевдокоде:

```
locations := количество локаций в лабиринте
ПОКА locations > 1
    выбираем случайную стену в лабиринте
    ЕСЛИ не существует пути между локациями, разделенными этой стеной,
        разбиваем стену
        locations := locations - 1
КОНЕЦ ЦИКЛА
```

Для того чтобы реализовать его на практике, потребуется, конечно же, гораздо больше усилий. Во-первых, нам понадобится функция, которая определяет, существует ли путь между двумя заданными локациями или нет. Для этого можно (и нужно) воспользоваться рекурсивным обходом или алгоритмом волновой трассировки, которые описаны выше в этой главе. Думаю, вам не составит труда чуть доработать любую из приведенных процедур, чтобы получилась функция:

```
function IsConnected(x1, y1, x2, y2) : Boolean
```

Предполагается, что она будет локальной и поэтому сможет получить доступ к лабиринту через переменную TheMaze — мы уже применяли такой подход. Во-вторых (и это более интересно), нам придется решить задачу *случайного выбора без повторов*.

В процессе построения лабиринта мы только разрушаем существующие стены, поэтому если между какими-то двумя локациями существует путь, он уже никуда не исчезнет. Таким образом, нет никакого резона выбирать два раза одну и ту же стену. Если некоторая стена была выбрана генератором случайных чисел однажды и осталась не разрушенной после текущей итерации алгоритма, путь между соответствующими локациями есть, и он останется в будущем.

Похожая ситуация возникает при моделировании игры в русское лото: если некоторый бочонок уже вынут из мешка, его номер больше не должен выпадать.

Для решения этой задачи существуют, по крайней мере, два известных метода.

1. Запоминать все ранее выбранные генератором случайных чисел значения. Если на очередной итерации выпало какое-то «старое» значение, просто запустить генератор еще раз (а если понадобится, то еще и еще раз).

   Этот метод прекрасно работает, если вам надо, к примеру, выбрать десять неповторяющихся чисел из диапазона [1...10 000]. Вероятность того, что одно и то же число выпадет два раза, очень невелика, и нет беды, если пару раз генератор сработает два-три раза вхолостую. А вот в случае вроде русского лото его применять не стоит: когда почти все значения из диапазона исчерпаны, как раз-таки вероятность получить новое, не выбранное ранее число мала. Наш случай, к сожалению, куда ближе к варианту русского лото, поэтому перейдем ко второму методу.
2. Предположим, что в массиве A[1..N] находятся все значения, которые надо выбрать в случайном порядке. Создадим массив B[1..N] и заполним его произвольными случайными числами. Практически любой универсальный алгоритм сортировки массива основан на операции обмена элементов. Я имею в виду, что основой алгоритма всегда является операция

   ```
   поменять местами B[i] и B[j]
   ```
3. Теперь надо отсортировать массив B по возрастанию, меняя местами каждый раз элементы массива A при обмене элементов из B. Иными словами, вместо кода

   ```
   поменять местами B[i] и B[j]
   ```

   везде будет использоваться

   ```
   поменять местами B[i] и B[j]
   ```

```
поменять местами A[i] и A[j]
```

4. После работы алгоритма сортировки массив A окажется полностью перемешанным в случайном порядке. В качестве первого случайного элемента можно взять первый элемент массива A, в качестве второго — второй и т. д. Вот и все.

Давайте теперь немного уточним алгоритм Краскала с учетом сказанного:

```
locations := количество локаций в лабиринте { Width * Height }
записываем все стены лабиринта в массив Walls
перемешиваем массив Walls в случайном порядке
i := 0
ПОКА locations > 1
    текущая стена := i-й элемент массива Walls
    i := i + 1
    ЕСЛИ не существует пути между локациями, разделенными этой стеной
        разбиваем стену
        locations := locations - 1
КОНЕЦ ЦИКЛА
```

Любую стену можно задать четырьмя числами: (x, y, dx, dy), то есть с помощью координат локации и смещений (я особо не останавливаюсь на этом, потому что мы не раз уже пользовались таким способом, например в функции CanGo() и процедуре BreakWall()). Таким образом, «массив стен» — это массив таких четверок.

Теперь можно заняться реализацией алгоритма. Мой вариант приведен в листинге 4.4, скриншот работающей программы — на рис. 4.9.

Листинг 4.4. Генерация лабиринта по алгоритму Краскала

```
function KruskalGenerateMaze(Width, Height : Integer) : Maze;
type Wall = record                    { тип "стена" }
    x, y, dx, dy : Integer;
end;
var   TheMaze   : Maze;               { сам лабиринт }
      Walls     : array of Wall;      { массив стен }
      Temp      : array of Real;      { временный массив для сортировки стен }
      i, j      : Integer;
      tempw     : Wall;
      tempr     : Real;
      CurWall   : Wall;
      locations : Integer;
      counter   : Integer;

procedure BreakWall(x, y, dx, dy : Integer);              { разрушить стену }
begin                                                     { между локациями }
    if dx = -1 then TheMaze[x, y].left_wall := false
    else if dx = 1 then TheMaze[x + 1, y].left_wall := false
    else if dy = -1 then TheMaze[x, y].up_wall := false
    else TheMaze[x, y + 1].up_wall := false;
end;

function IsConnected(xs, ys, xf, yf : Integer) : Boolean;
... { используется алгоритм волновой трассировки }
```

продолжение ⇩

Листинг 4.4 (продолжение)

```
    begin
        { выделение памяти для массива стен }
        { в лабиринте Width * Height изначально
        { (Width - 1) * Height + (Height - 1) * Width стен }
        SetLength(Walls, (Width - 1) * Height + (Height - 1) * Width);
        SetLength(Temp, (Width - 1) * Height + (Height - 1) * Width);
        SetLength(TheMaze, Width + 1, Height + 1);   { указать размер лабиринта }

        for i := 0 to Width do                          { все стены изначально }
        for j := 0 to Height do                         { существуют }
        begin
            TheMaze[i, j].left_wall := true;
            TheMaze[i, j].up_wall := true;
        end;

    Randomize;
    for i := 0 to (Width - 1) * Height + (Height - 1) * Width - 1 do
        Temp[i] := Random;      { заполнение массива Temp случайными числами }

    counter := 0;                           { заполнение массива стен }
    for i := 1 to Width - 1 do
        for j := 0 to Height - 1 do
        begin                                       { сначала все горизонтальные }
            Walls[counter].x := i; Walls[counter].y := j;
            Walls[counter].dx := -1; Walls[counter].dy := 0;
            counter := counter + 1;
        end;
    for i := 0 to Width - 1 do
        for j := 1 to Height - 1 do
        begin                                   { затем все вертикальные }
            Walls[counter].x := i; Walls[counter].y := j;
            Walls[counter].dx := 0; Walls[counter].dy := -1;
            counter := counter + 1;
        end;

    for i := 0 to (Width - 1) * Height + (Height - 1) * Width - 1 do
        for j := i to (Width - 1) * Height + (Height - 1) * Width - 1 do
            if Temp[i] <@062> Temp[j] then          { перемешиваем массив стен }
            begin
                tempr := Temp[i]; Temp[i] := Temp[j]; Temp[j] := tempr;
                tempw := Walls[i]; Walls[i] := Walls[j]; Walls[j] := tempw;
            end;

    locations := Width * Height;
    i := 0;
    while locations > 1 do                  { прямолинейная реализация }
    begin                                   { алгоритма Краскала }
        CurWall := Walls[i];
        i := i + 1;
        if not IsConnected(CurWall.x, CurWall.y,
                           CurWall.x + CurWall.dx, CurWall.y + CurWall.dy) then
        begin
            BreakWall(CurWall.x, CurWall.y, CurWall.dx, CurWall.dy);
```

```
                locations := locations - 1;
                ShowMaze(TheMaze);
                Application.ProcessMessages;
            end;
        end;

        KruskalGenerateMaze := TheMaze;
    end;
```

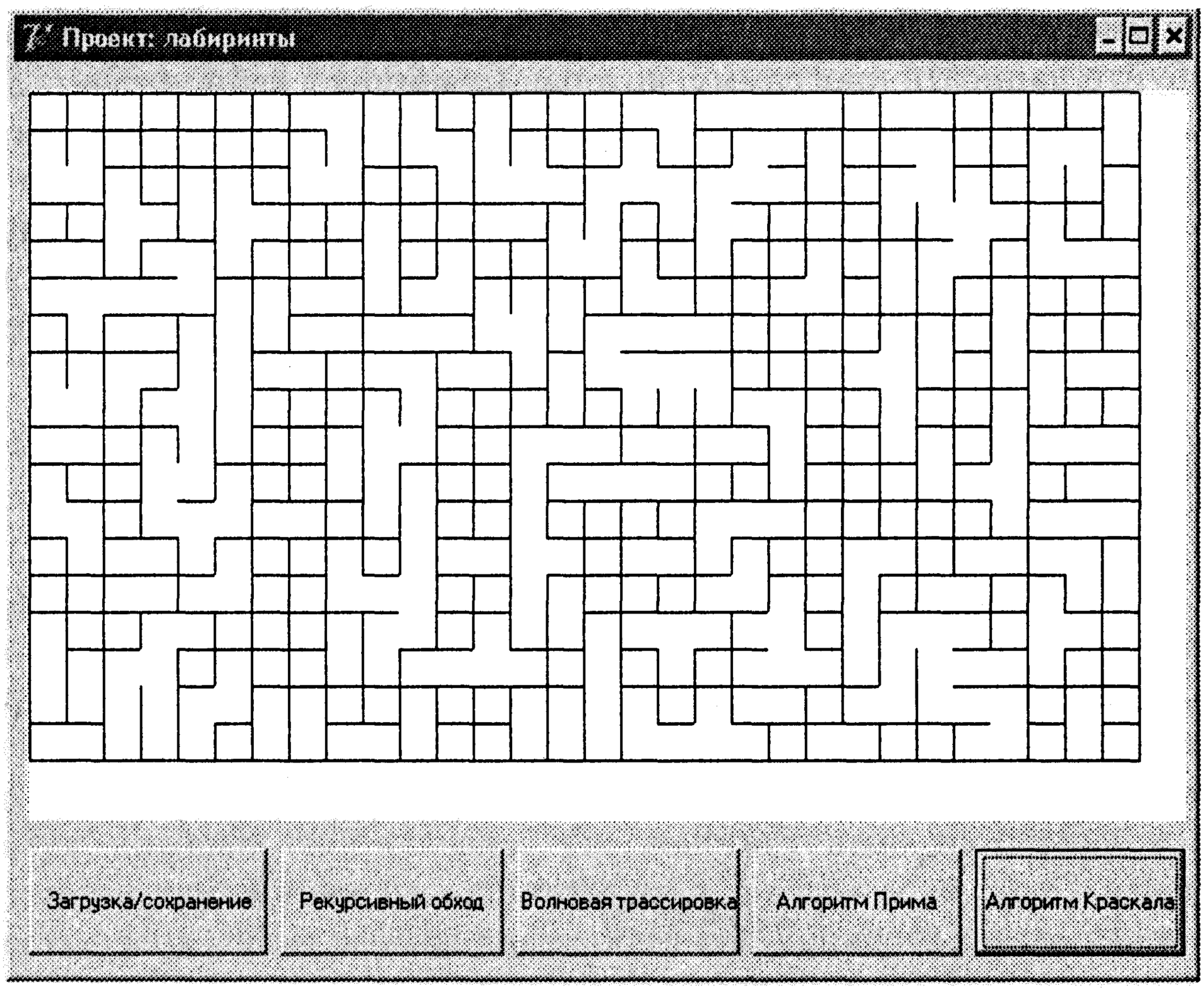


Рис. 4.9. Алгоритм Краскала в процессе работы

Проекты для самосовершенствования

1. Помните, что алгоритм волновой трассировки фактически моделирует поведение разлитой в лабиринте воды (киселя). Подумайте, как можно использовать этот алгоритм в компьютерной графике для закрашивания замкнутых областей.
2. Есть два простых способа улучшить алгоритм волновой трассировки:
   - На каждой итерации работы алгоритма происходит поиск локаций, помеченных числом N. Чтобы не пробегать по всему лабиринту, можно просто хранить их в отдельном списке (разумеется, его придется постоянно обновлять).
   - Можно «разлить кисель» не только в стартовой локации, но и в финишной. Как только оба потока жидкости пересекутся в некоторой локации, маршрут найден.

   Реализуйте оба варианта на практике.

3. Добавьте в процедуры обхода лабиринта код, который позволит наблюдать ход решения. К примеру, в алгоритме рекурсивного обхода можно рисовать на экране текущий маршрут, а в алгоритме волновой трассировки выделять локации, помеченные значением N.
4. Подумайте, как можно реализовать алгоритм Прима более эффективно. К примеру, храните все Border-локации в отдельном списке, чтобы не просматривать в их поиске весь лабиринт на каждой итерации.
5. Вы уже умеете генерировать лабиринты и знакомы с трехмерной графикой. Попробуйте написать программу, которая генерирует лабиринт и выводит на экран в трехмерном виде его стартовую локацию (как будто путешественник находится внутри нее и видит все своими глазами). Пользователю должны быть доступны хотя бы три действия: повернуться налево на 90 градусов, повернуться направо на 90 градусов и пройти вперед к следующей локации. Я думаю, в начале 80-х такая нехитрая игра могла бы иметь коммерческий успех.

Глава 5

Сжатие данных

— Ты и представить себе не можешь, до чего фрекен Бок
хочет попасть в телевизор, — начал он.
Но Карлсон прервал его новым взрывом хохота:
— Домомучительница хочет залезть в такую
маленькую коробочку?! Такая громадина!
Да ее пришлось бы сложить вчетверо!

Астрид Линдгрен, «Малыш и Карлсон»

Я долго думал, стоит ли включать главу о сжатии данных в книгу, посвященную занимательному программированию. Действительно, что тут «занимательного»? Скормил программе двухмегабайтный текстовый файл, она немного похрустела винчестером и выдала другой, содержащий откровенную белиберду, зато занимающий всего один мегабайт. Скормил второй программе этот выходной файл — получил тот, который был вначале... Да, действительно, на первый взгляд дела обстоят именно так. Я тоже раньше так думал. На самом же деле в задаче сжатия данных есть очень много работы для творческой личности; пусть даже результат не так очевиден, как при разработке компьютерной игры или при создании анимационного ролика.

Я почти не сомневаюсь, что среди вас будет гораздо больше людей, которым глава о компьютерной графике окажется куда ближе этой, тем не менее все же попробую доказать, что и в сжатии данных есть много интересного. К тому же я убедился, что на русском языке практически нет литературы, в которой сжатие описывалось бы интересно и доступно; может быть, мне удастся сделать это лучше.

Сразу оговорюсь: речь пойдет о так называемом сжатии без потерь качества. Такой тип сжатия используется во всех архиваторах общего назначения вроде RAR или ZIP, а также, к примеру, в графическом формате PNG. Сжатие с потерями качества (используемое в форматах JPEG, MP3, MPEG4) рассматриваться не будет — это отдельная, большая и довольно специфическая тема.

Еще должен предупредить: в отличие от других глав этой книги, здесь вы не найдете готовых (доведенных до реализации) примеров. Этому есть масса причин:

1) в проблеме сжатия теория гораздо интереснее конкретных реализаций (которые, действительно, не особо «занимательны»);

2) эта часть книги стоит немного особняком, поэтому отсутствие примеров не помешает вам понять содержимое оставшихся глав;
3) программы сжатия/распаковки не слишком сложны логически, но довольно громоздки (и зачастую требуют специальных структур данных);
4) кроме того, эти программы часто связаны с конкретными форматами входных данных, а мне не хочется ударяться в частности;
5) к пятой главе мне немного надоело изобретать и отлаживать примеры — не буду лишать вас удовольствия сделать это самостоятельно.

Немного теории

Предположим, что мы собираемся сжать файл, который состоит из *элементов* A_1, ..., A_n. Что значит «состоит из элементов»? Любой файл, независимо от его природы, можно рассматривать как последовательность двух символов — нуля и единицы, ибо все в компьютере представляется в виде двоичных чисел. В этом случае количество элементов у нас равно двум: $A_1 = 0$, $A_2 = 1$. Тот же самый файл можно представить как последовательность байтов. А каждый байт — это число от нуля до 255. Значит, с другой стороны, наш файл состоит из элементов $A_1 = 0$, $A_2 = 1$, ..., $A_{256} = 255$. Если файл имеет некоторую специфическую природу, появляются новые возможности. Например, 16-цветную картинку (помните такие? уже, скорее всего, нет) можно полагать состоящей из пикселов; количество разных элементов будет равно шестнадцати. Если высота и ширина картинки — четные числа, никто не мешает рассматривать ее как совокупность квадратиков 2 × 2 пиксела. Размер картинки в элементах будет в четыре раза меньше размера картинки в пикселах, а всего разных элементов окажется 16 × 16 × 16 × 16 = 65 536. Файл, содержащий документ на русском языке, можно представить в виде последовательности слов; количество элементов при этом равно количеству слов в русском языке. MIDI-файл — последовательность нот, WMF-файл — последовательность команд графической системы Windows и т. д.

Главное — что вы *сами* определяете, из каких именно элементов состоит тот или иной файл, в зависимости от поставленной задачи. Если угодно, полагайте, что файл состоит из байтов; если более целесообразно считать, что он составлен из нот или пикселов — пожалуйста! Чем больше размер каждого элемента, тем меньше размер итогового файла (если считать в элементах), и наоборот. Как правило, чем больше размер отдельных элементов, тем больше их итоговое количество. Бит — мельчайшая единица информации, и разных элементов размером в один бит всего два. Байт в восемь раз больше бита (в том смысле, что для хранения одного байта требуется столько же места, сколько для хранения восьми битов), но разных элементов размером в байт уже будет 256.

Рассмотрим теперь понятие *вероятности появления элемента* A_i (обозначается $p(A_i)$). Если элемент вообще не встречается в файле, вероятность его появления равна нулю; если файл состоит лишь из одного элемента, который все время повторяется, вероятность его появления равна единице. Если данный элемент

встречается в среднем один раз из четырех случаев, то вероятность равна 0,25 и т. д.

Если есть возможность заранее проанализировать входной файл, вероятность можно определить точно:

$p(A_i)$ = количество элементов A_i в файле / размер файла в элементах

Предположим, что файл состоит из пятидесяти букв. Анализ выявил, что буква «б» встречается в файле десять раз. Тогда p(«б») = 10/50 = 1/5.

На практике этой формулой не всегда пользуются (почему – об этом пойдет речь ниже); часто вероятности появления элементов определяют, исходя из каких-то других (например, опытных) соображений. Допустим, вы проанализировали тысячу черно-белых изображений, которые содержат сканированный текст, и определили, что вероятность встретить черный пиксел в среднем равна такому-то числу, а вероятность встретить белый – такому-то. Логично предположить, что и в тысяча первом изображении эти вероятности будут недалеки от истины. Разумеется, подобный способ не гарантирует абсолютной точности, а при ошибках он может хорошо наказать (файл плохо сожмется); но во многих случаях его использование вполне оправданно.

Теперь – внимание! «Волшебная» формула. Для оптимального хранения входного файла на элемент, равный A_i, требуется отвести ровно

$$H(A_i) = -\log_2(p(A_i))$$

бит. Эта величина ($H(A_i)$) называется *энтропией* элемента A_i. Формула энтропии берет начало из термодинамики (где сама энтропия определяется как «мера вероятности пребывания системы в данном состоянии») и связана, в первую очередь, с работами физика Больцмана. В середине двадцатого столетия понятие энтропии использовал Клод Шеннон при разработке теории информации. Здесь энтропия используется как, скажем так, «мера информативности» элемента. Сколько информации реально содержится в том или ином элементе входного файла (пикселе, букве, ноте, слове)? Если формат файла предусматривает, что на хранение, к примеру, каждой ноты отводится два байта, это еще ни о чем не говорит: возможно, информативность одних нот выше, а других – ниже. Перераспределив занимаемое нотами место (в соответствии с их энтропиями), мы можем получить выигрыш в итоговом размере файла. Энтропия в теории информации измеряется в битах. А бит – это единица измерения информации. Таким образом, энтропия элемента равна «количеству информации», которое он несет.

Если мы знаем вероятность появления каждого элемента, можно посчитать среднюю энтропию элемента по всему файлу:

$$H_{\text{элемента}} = p(A_1) * H(A_1) + p(A_2) * H(A_2) + \ldots + p(A_n) * H(A_n)$$

Умножив среднюю энтропию элемента на размер файла (в элементах), получаем итоговый размер файла в битах при условии его оптимального хранения:

размер файла (в битах) = размер файла (в элементах) * $H_{\text{элемента}}$

Если формулы наводят на вас тоску, спешу обрадовать: больше их не будет (или почти не будет). Но те, что мы уже рассмотрели, пожалуйста, запомните – это наш главный инструментарий.

Чтобы прояснить ситуацию, сейчас я приведу пару примеров, но перед этим давайте рассмотрим некоторые следствия из приведенных формул.

1. Энтропия элемента непосредственно связана с вероятностью его появления. Зависимость между этими величинами показана на рис. 5.1.

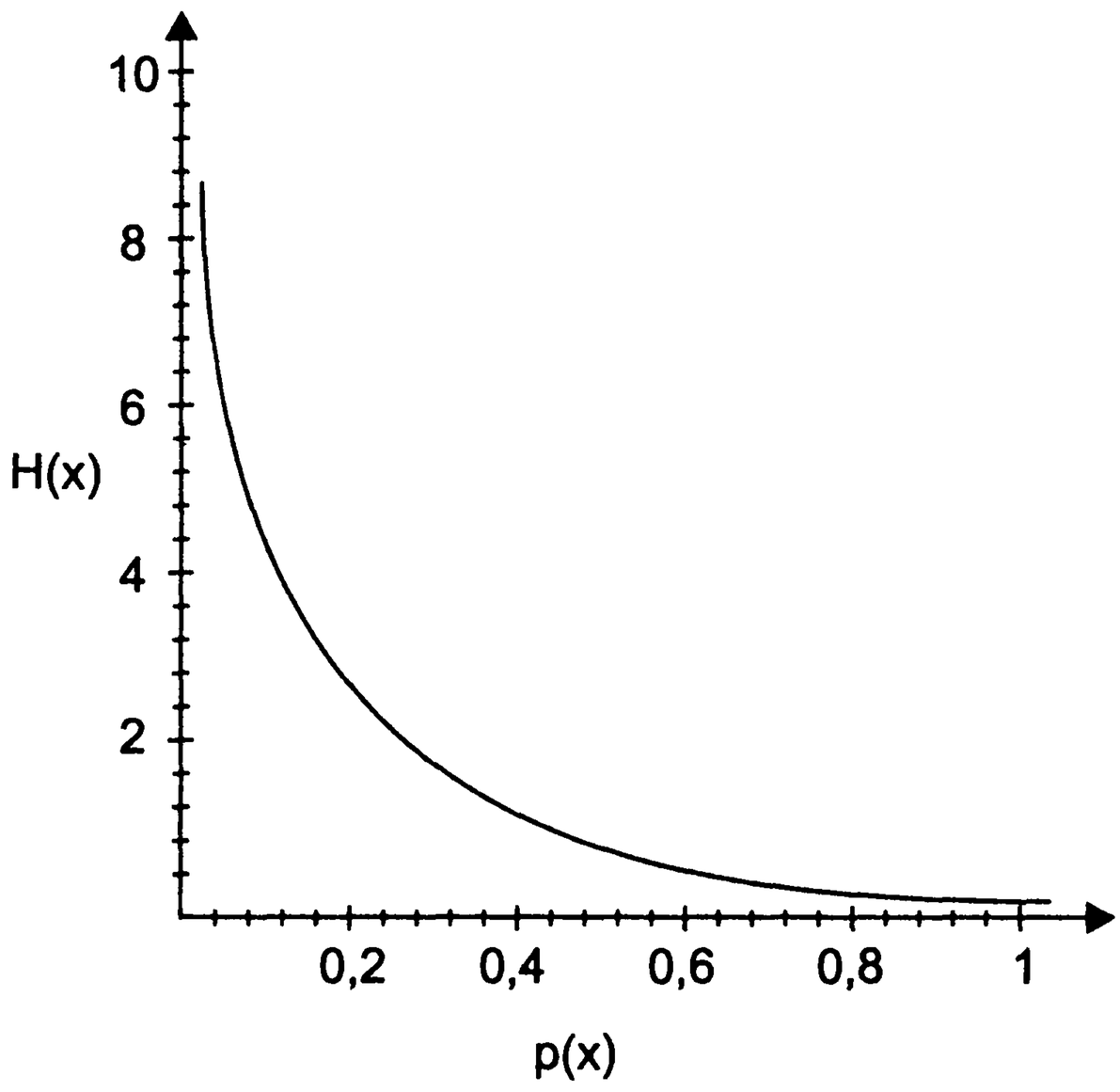


Рис. 5.1. Связь энтропии с вероятностью появления элемента в файле

2. Как видно из графика, чем больше вероятность появления элемента, тем меньше его энтропия. Если элемент встречается в файле очень часто, его энтропия (то есть количество информации, которое несет в себе элемент) приближается к нулю; если же элемент встречается крайне редко, энтропия может быть очень большой.
3. Энтропия всегда неотрицательна (что логично, поскольку количество информации не может быть меньше нуля).
4. Формулы дают нам понять, насколько можно сжать файл (иными словами, сколько места будет занимать файл при условии оптимального его хранения), но не объясняют, как это сделать.
5. В результате вычисления энтропии может получиться дробное число. Как мы знаем, «половина бита» не существует. Тем не менее вычисленное значение имеет смысл. Если, к примеру, энтропия некоторого элемента оказалась равна 1/4 бита, то четыре таких элемента действительно можно сохранить в одном бите данных, как ни парадоксально это звучит. Как именно это сделать — вопрос второй.

Теперь — обещанные примеры. Рассмотрим текстовый файл:

```
ааббббаааав
```

Пример 1. Будем считать, что этот файл состоит из отдельных букв. Тогда длина файла (в элементах) равна десяти (он состоит из десяти букв). Посчитаем вероятности появления каждой буквы (для этого, как уже говорилось, надо разделить количество появлений буквы на общую длину файла):

p("а") = 5/10
p("б") = 4/10
p("в") = 1/10

По формуле энтропии считаем значение, что логично, энтропии каждого элемента-буквы:

$H("а") = -\log_2(5/10) = 1$
$H("б") = -\log_2(4/10) \approx 1.3219$
$H("в") = -\log_2(1/10) \approx 3.3219$

Теперь можно подсчитать общий объем файла при условии его оптимального хранения:

$H_{элемента} = (5/10) * 1 + (4/10) * 1.3219 + (1/10) * 3.3219 = 1.3610$ (бит)
размер файла (бит) = 10 * 1.3610 = 13.610 (бит)

Итак, теоретически для хранения данного файла достаточно 13,610 бит.

Пример 2. Рассматриваем тот же файл, но будем на сей раз считать, что он состоит не из отдельных букв, а из двухбуквенных элементов. В нашем случае количество таких элементов невелико: **«аа»**, **«бб»** и **«ав»**. Размер файла в элементах теперь будет равен пяти (пять элементов по две буквы в каждом). Вычислим вероятности появления каждого элемента и соответствующие значения энтропии:

p("аа") = 2/5
p("бб") = 2/5
p("ав") = 1/5
$H("аа") = -\log_2(2/5) \approx 1.3219$
$H("бб") = -\log_2(2/5) \approx 1.3219$
$H("ав") = -\log_2(1/5) \approx 2.3219$

Тогда общий объем файла при условии оптимального хранения равен:

$H_{элемента} = (2/5) * 1:3219 + (2/5) * 1.3219 + (1/5) * 2.3219 = 1.5219$ (бит)
размер файла (бит) = 5 * 1.5219 = 7.6095 (бит)

Пример 3 (последний). Теперь полагаем, что файл (все тот же) состоит из двух элементов: **«ааббб»** и **«бааав»**. Длина файла в элементах равна двум.

p("ааббб") = 1/2
p("бааав") = 1/2
H("ааббб") = 1
H("бааав") = 1
$H_{элемента} = (1/2) * 1 + (1/2) * 1 = 1$
размер файла (бит) = 2 * 1 = 2 (бита)

Из этих примеров можно сделать выводы:

1. Для любого входного файла можно составить *модель*, то есть представление файла в виде тех или иных элементов. Способ представления (то есть реальный набор элементов) мы выбираем сами.
2. Теоретические пределы сжатия очень сильно зависят от выбранной модели. Если модель оказалась неудачной, файл не сожмется (а то и вырастет в размерах); если же модель хороша, степень сжатия может быть высокой.
3. Для того чтобы оценить качество модели, нет нужды садиться и писать архиватор. Достаточно воспользоваться приведенными формулами (кстати, при подобных расчетах очень удобным инструментом оказывается Microsoft Excel).

Моделирование и кодирование

Думаю, вы уже заметили, что формулы выдают совершенно фантастические результаты. В третьем примере получилось, что для хранения файла из десяти букв требуется всего два бита! Разумеется, не все так просто. О бесплатном сыре, о мышеловках и о том, что же делать на практике, мы сейчас и поговорим.

Действия, которые мы осуществляем с файлом в процессе его сжатия, обычно разделяют на *моделирование* и *кодирование*. На практике они нередко пересекаются (например, модель пересматривается в процессе кодирования), тем не менее такое разделение, пусть даже порою условное, существует.

Под моделированием в первую очередь понимают выбор *алфавита*, то есть набора элементов, составляющих исходный файл. Именно моделированием мы занимались в трех примерах, приведенных выше. Перед тем как выбрать алфавит, исходный файл нередко преобразовывают некоторым хитрым способом (о таких способах мы еще поговорим).

Если модель уже выбрана, можно подсчитать теоретический объем сжатого файла (этим мы тоже занимались). Кодирование – это собственно процесс компрессии файла в соответствии с моделью. В принципе, кодирование – вопрос в большей или меньшей степени решенный. Кодировщик – это всего лишь алгоритм, инструмент, если угодно. Уже изобретены очень приличные способы кодирования, которыми мы можем воспользоваться. Не хочу брать на себя слишком большую ответственность, но, на мой взгляд, изобретать новый алгоритм кодирования – это почти то же самое, что изобретать новый алгоритм сортировки; я сомневаюсь, что здесь можно достичь хоть сколько-нибудь заметного прогресса. Вот моделирование – дело другое. Здесь есть хороший простор для фантазии.

Давайте теперь подумаем, какого рода информация нужна кодировщику для работы. Во-первых, это сам входной файл, который кодировщик будет сжимать. Во-вторых, поскольку мы сами решаем, из каких элементов состоит входной файл, необходимо предоставить кодировщику эту информацию (сам он, конечно, не догадается). В-третьих, для определения размера каждого элемента при оптимальном хранении мы использовали вероятности появления. Чтобы кодировщик мог решить, что на такой-то элемент надо выделить пять битов, а вот на этот хватит и трех, ему тоже потребуются эти вероятности. Плохие новости состоят в том, что программе, которая осуществляет распаковку сжатого файла, *тоже* понадобятся данные об элементах и вероятностях их появления; в противном случае получится как в анекдоте.

Разговаривают два программиста:

– Слушай, вчера написал новый архиватор. Любой файл сжимает в 5 байт!

– Ну, просто здорово!

– Ага. Сейчас работаю над разархиватором...

В этом и кроется объяснение «бесплатного сыра»: выбрав сложную модель с большим количеством элементов, мы, конечно, сожмем исходный файл в несколько байт. Однако, если при этом придется к файлу подсоединить «прицеп»

размером в пару мегабайт (содержащий список элементов и соответствующих вероятностей), никто не оценит вашей гениальности. Поэтому разработчику архиватора подчас приходится искать баланс между размером модели и размером сжатого файла: больше модель – меньше сжатый файл; меньше модель – больше сжатый файл.

Хорошие новости состоят в том, что модель не всегда приходится передавать вместе со сжатым файлом; нередко удается «договориться» с распаковщиком менее болезненным способом.

Статическая, полуадаптивная и адаптивная схемы сжатия

Начнем с того, что исходный алфавит (множество элементов, из которых состоит сжимаемый файл), как правило, выбирается один раз и навсегда. Если вы пишете архиватор, который рассчитан на MIDI-файлы, скорее всего, вы будете работать с нотами независимо от того, какая музыкальная композиция передана на вход. Если архиватор пытается сжимать изображения, он опять-таки будет полагать, что любое изображение состоит из элементов одного и того же типа (пикселов, к примеру)[1]. Отсюда вывод: передавать распаковщику исходный алфавит не придется; достаточно лишь запрограммировать его на какой-то конкретный тип алфавита («просто байты», ноты, буквы, пикселы). Проблема обычно заключается не в алфавите, а в значениях вероятностей появления того или иного элемента[2]. Эти значения могут очень сильно варьироваться от файла к файлу: к примеру, в одном изображении присутствует очень много красных пикселов, а в другом превалируют зеленые. На вопрос о том, что делать с вероятностями, есть три ответа:

1) использовать статическую схему сжатия;
2) использовать полуадаптивную схему сжатия;
3) использовать адаптивную схему сжатия.

Смысл статической схемы прост: не будем со сжатым файлом передавать вообще никаких вероятностей; договоримся об их конкретных значениях заранее. К примеру, создаю я архиватор, который сжимает файлы, состоящие из букв «а», «б», «в» и «г». Предположу, что вероятности появления букв таковы:

```
p("а") = 1/2
p("б") = 1/10
p("в") = 1/6
p("г") = 7/30
```

---

[1] Я говорю об «архиваторах, рассчитанных на MIDI-композиции», «архиваторах, рассчитанных на изображения»... Наверное, большинство из вас привыкло больше пользоваться универсальными архиваторами, работающими с любыми типами файлов. Стоит ли говорить, однако, что архиватор, приспособленный под определенный тип файла, может более эффективно использовать его особенности?

[2] Конечно, я не утверждаю, что исходный алфавит *никогда* не приходится передавать. Все, в конечном счете, зависит от специфики задачи.

Запрограммирую упаковщик и распаковщик на использование только этих чисел — и дело сделано: передавать вместе со сжатым файлом ничего не надо. Напрашивается вопрос: а что же будет, если реальные вероятности появления букв во входном файле будут отличаться от тех, что я задал? Ответ неутешительный: в этом случае файл сожмется плохо (а то и вырастет в размере). Это легко проверить. Попробуем сжать, например, такой файл:

абббгвбб

Вычислим энтропии элементов:

$H("а") = -\log_2(1/2) = 1$
$H("б") = -\log_2(1/10) \approx 3.3129$
$H("в") = -\log_2(1/6) \approx 2.5850$
$H("г") = -\log_2(7/30) \approx 2.0995$

Энтропия элемента, как вы помните, — это его размер в битах при оптимальном хранении. Таким образом, суммарный размер файла (в битах) будет равен:

```
   1 + 3 * 3.3129 + 2.0995 + 2.5850 + 2 * 3.3129 = 22.2490
("а" +   "ббб"    +   "г"  +  "в"   +    "бб")
```

Теперь посмотрим, как изменится итоговый размер сжатого файла, если модель соответствует действительности, то есть если вероятности определены корректно:

$p("а") = 1/8$
$p("б") = 5/8$
$p("в") = 1/8$
$p("г") = 1/8$

$H("а") = -\log_2(1/8) = 3$
$H("б") = -\log_2(5/8) \approx 0.6781$
$H("в") = -\log_2(1/8) = 3$
$H("г") = -\log_2(1/8) = 3$

итоговый размер = 3 + 3 * 0.6781 + 3 + 3 + 2 * 0.6781 = 12.3905 (бит)

Как говорится, комментарии излишни.

Существуют, тем не менее, случаи, когда статическая схема сжатия применима. Вероятность появления той или иной буквы русского языка вряд ли сильно варьируется от текста к тексту — можно один раз определить эти вероятности и пользоваться ими для любого входного текстового файла. Та же ситуация с распределением черных и белых пикселов в изображениях, содержащих отсканированный текст (о чем уже говорилось). Конечно, букв в алфавите не так много, поэтому выгода от использования статической схемы не так велика (в конце концов, несколько десятков значений вероятностей можно себе позволить сохранить в результирующем файле), но порою количество элементов исчисляется не десятками, а тысячами. В этом случае статическая схема может оказаться очень выигрышной (если, конечно, ее удастся применить).

Полуадаптивная схема сжатия предполагает предварительный анализ входного файла и определение реальных вероятностей появления элементов. Вычисленные вероятности сохраняются вместе со сжатым файлом, чтобы распаковщик был способен выполнить свою работу. Достоинства и недостатки метода видны невооруженным взглядом: с одной стороны, вероятности будут определены точ-

но, а с другой стороны, на их хранение требуется место (порою значительное). Еще один недостаток метода – предварительный анализ файла. Во-первых, это требует времени, а во-вторых, такой анализ не всегда возможен: к примеру, если вы слушаете радио в Интернете[1] и в реальном времени сохраняете передачу на винчестер в формате MP3. Тогда вплоть до окончания трансляции полное содержимое входного файла кодировщику неизвестно.

Адаптивная схема сжатия – хороший, часто используемый компромисс. Изначально кодировщик полагает, что вероятности появления каждого элемента равны между собой[2]: $p(A_1) = p(A_2) = ... = p(A_n) = 1/n$. По мере сжатия входного файла производится его анализ, и значения вероятностей обновляются. Таким образом, в самом начале работы кодировщика вероятности могут быть очень далеки от реальности, но постепенно модель «адаптируется» к входному файлу. Поскольку распаковщик может выполнить аналогичную процедуру во время своей работы, значения вероятностей хранить в сжатом файле не нужно. В этом, естественно, заключается очень большой плюс адаптивной схемы сжатия. Другой плюс – отсутствие фазы анализа входного файла, поэтому адаптивную схему сжатия можно применять в случаях, когда на момент сжатия его содержимое целиком неизвестно (как в примере с интернет-радио). Трудно дать однозначную оценку методу. С одной стороны, очень часто он быстро адаптируется к входным данным (если начало файла более или менее объективно отражает реальную ситуацию); с другой стороны, ничего не гарантируется: возможно, адаптация будет происходить медленно, и вся процедура в итоге окажется не слишком эффективной.

Рассмотрим теперь два популярных алгоритма кодирования, а затем поговорим о различных инструментах моделирования.

Кодирование методом Хаффмана

Этот алгоритм был изобретен еще в 1952 году Дэвидом Хаффманом (David Huffman). Объяснить его суть проще всего на примере. Допустим, входной файл состоит из шести различных элементов, и нам известны вероятности их появления (а вероятности так или иначе известны всегда, пусть даже не соответствующие действительности):

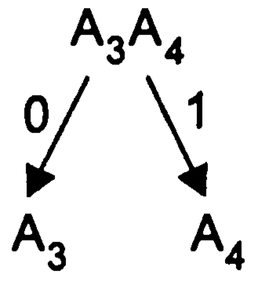
$p(A_1) = 10/34$
$p(A_2) = 7/34$
$p(A_3) = 3/34$
$p(A_4) = 2/34$
$p(A_5) = 8/34$
$p(A_6) = 4/34$

---

[1] Прошу не скрипеть зубами тех, для кого подобное использование Интернета — все еще непозволительная роскошь.

[2] Я нигде не упоминал, что сумма всех вероятностей появления элементов должна равняться единице, что представляется довольно очевидным. Если, к примеру, количество элементов равно трем, а на долю первых двух элементов приходится половина файла ($p(A_1) + p(A_2) = 1/2$), то на долю третьего элемента придется оставшаяся половина файла, поскольку других вариантов просто нет: $p(A_3) = 1/2$.

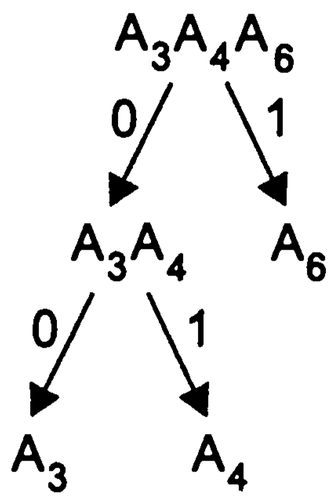
Выберем два наиболее редко встречающихся элемента (в нашем случае это A_3 и A_4) и объединим их в дерево:



Элементы A_3 и A_4 исключаются из исходной таблицы вероятностей, а их место занимает новый «псевдоэлемент» A_3A_4, вероятность появления которого равна сумме вероятностей появления элементов A_3 и A_4: $p(A_3A_4) = p(A_3) + p(A_4)$. Ветви, идущие от нового элемента к исходным, помечаются нулем и единицей (при этом неважно, какую ветвь каким числом пометить). Теперь таблица вероятностей выглядит так:

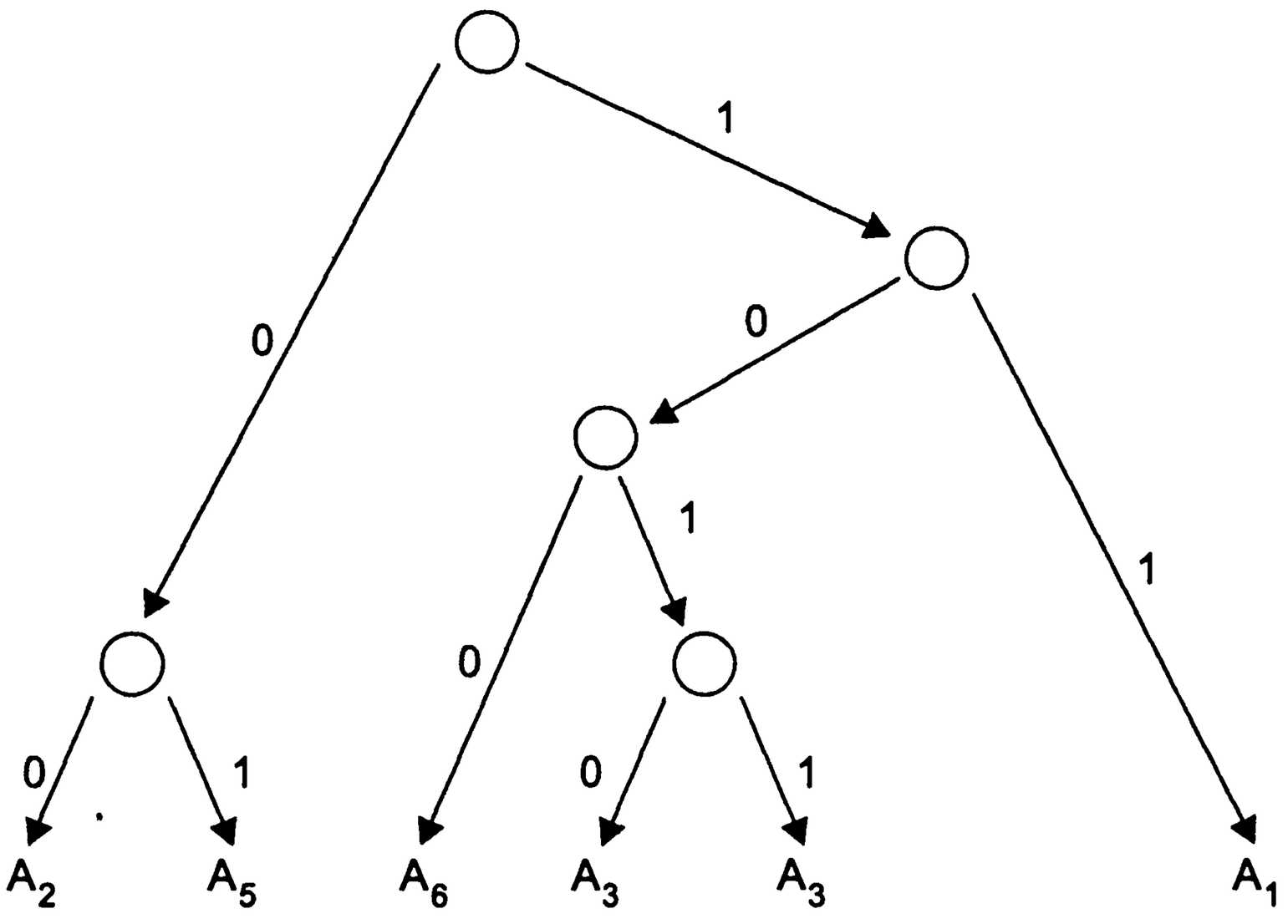
$p(A_1) = 10/34$
$p(A_2) = 7/34$
$p(A_3A_4) = 5/34$
$p(A_5) = 8/34$
$p(A_6) = 4/34$

Выполним процедуру объединения самых редких элементов еще раз. На этом этапе самые редкие элементы — A_3A_4 и A_6:



Вероятность появления нового элемента, как и раньше, определяется как сумма вероятностей появления дочерних элементов: $p(A_3A_4A_6) = p(A_3A_4) + p(A_6) = 9/34$.

Продолжим процесс, пока в таблице вероятностей не окажется один-единственный элемент. В результате мы получим *дерево Хаффмана*:



Любой элемент (A_1...A_6) может быть однозначно задан маршрутом от корня дерева до него. К примеру, код элемента A_3 равен 1011. Таким образом, дерево Хаффмана определяет *таблицу кодов* элементов:

A_1 – 11
A_2 – 00
A_3 – 1011
A_4 – 1010
A_5 – 01
A_6 – 100

Коды состоят только из нулей и единиц, поэтому для хранения каждой цифры кода потребуется всего один бит. Получается, что для кодирования, к примеру, элемента A_1 придется потратить всего два бита, а для кодирования элемента A_3 – четыре. Таким образом, к примеру, последовательность $A_1A_1A_2A_6$ будет закодирована битовой строкой 111100100.

Пожалуй, самый необычный элемент алгоритма — ввод-вывод одиночных битов. Разумеется, один бит нельзя записать в файл; зато можно подождать, пока их накопится 8 штук, и вывести байт целиком. Думаю, стоит рассмотреть реализацию такой процедуры подробнее:

```
{ глобальные переменные }
var This   : byte = 0;          { текущий байт }
    count  : Integer = 0;       { счетчик битов }
    MyFile : File of byte;      { выходной файл }

procedure PutBit(bit : byte);          { вывести один бит в файл }
begin                                  { (на входе ожидается число 0 или 1) }
    This := This or (bit shl count); { записать бит в соответствующий разряд }
    count := count + 1;                { перейти к следующему разряду }

    if count = 8 then                        { если выведено 8 бит }
    begin
        Write(MyFile, This);                 { записать 1 байт в выходной файл }
        count := 0;                          { обнулить счетчик битов }
        This := 0;                           { и текущий байт }
    end;
end;
```

Квинтэссенция всей процедуры — строка

```
This := This or (bit shl count);
```

Она записывает значение bit (нуль или единица) в разряд count в байте This[1]. Рассмотрим на примере, как это происходит.

Допустим, count = 3, bit = 1. Операция bit shl count сдвигает все биты числа bit на count позиций влево. Изначально значение bit было равно единице (двоичное 00000001); теперь же оно окажется равным двоичному 00001000. Далее это значение накладывается на This с использованием битовой операции OR. Напомню, что

```
0 or 0 = 0
0 or 1 = 1
```

[1] Самый правый разряд имеет номер 0, самый левый — 7.

```
1 or 0 = 1
1 or 1 = 1
```

Таким образом, все разряды числа This, кроме третьего, останутся без изменения. Если там был нуль, выполнится операция 0 or 0 = 0, если же там была единица, получаем 1 or 0 = 1. В третьем разряде в любом случае появится единица: 0 or 1 = 1; 1 or 1 = 1. В итоге мы получили то, что, собственно, и требовалось.

Если значение bit равно 0, никаких изменений число This не претерпит; но поскольку изначально значение This равно нулю (двоичное 00000000), в требуемом разряде будет находиться нуль.

Функция GetBit() реализуется сходным образом:

```
{ глобальные переменные }
var This  : byte;
    count : Integer = 8;
    MyFile : File of byte;

function GetBit : byte;
var bit : byte;
begin
    if count = 8 then              { если считано 8 бит }
    begin
        Read(MyFile, This);        { берем из файла очередную порцию }
        count := 0;
    end;

    bit := (This shr count) and 1;   { определяем очередной бит }
    count := count + 1;

    GetBit := bit;                 { возвращаем его в качестве результата }
end;
```

Поскольку изначально в переменной This ничего не записано, count инициализируется числом 8 — чтобы при первом же вызове считать байт из файла. Предположим, нам надо получить бит из третьего разряда. После сдвига вправо на count разрядов при помощи операции shr он переместится в нулевую позицию. Теперь мы применяем наложение по AND с числом 1 (двоичное 00000001):

```
0 and 0 = 0
0 and 1 = 0
1 and 0 = 0
1 and 1 = 1
```

Если в нулевом разряде оказалась единица, ее мы и получим (1 and 1 = 1); если же нуль — операция and опять выдаст правильный ответ (0 and 1 = 0).

При использовании описанной методики нельзя забывать о двух тонкостях:

1. Поскольку PutBit() реально записывает данные в файл лишь на каждом восьмом вызове, надо убедиться, что последняя порция битов действительно оказалась на диске. Проще всего это сделать семикратным вызовом PutBit().
2. По той же самой причине функция GetBit() не может определить, сколько битов из входного файла действительно значимы, а сколько используются лишь

для заполнения пустого места в последнем выводимом байте. Простейшее решение этой проблемы – хранить вместе со сжатыми данными общее количество элементов в исходном файле. Тогда после раскодирования этого числа элементов распаковщик закончит работу.

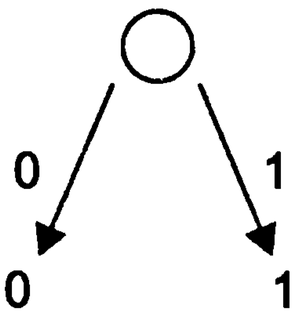
У алгоритма Хаффмана, разумеется, есть свои плюсы и минусы:

+ он простой – легко запоминается и сравнительно легко программируется;
+ он подходит для всех схем сжатия (статической, полуадаптивной и адаптивной);
+ он быстро работает;
+ он генерирует таблицу кодов элементов (порою это очень удобно);
– при использовании адаптивной схемы приходится постоянно обновлять дерево (не самое знатное развлечение);
– он генерирует таблицу кодов элементов и уже поэтому *не оптимален.*

Самая главная проблема алгоритма – его неоптимальность. Вы, конечно, заметили, что каждому элементу входного файла алгоритм Хаффмана сопоставляет некоторую последовательность из *целого* количества битов. А что будет, если энтропия символа не равна целому числу? Ответ прост: размер сжатого файла окажется больше, чем он мог бы быть по теории. На практике алгоритм Хаффмана работает лучше, если количество различных элементов во входном файле достаточно велико, а вероятности их появления не слишком сильно разнятся. В «экстремальных» случаях сжатия вообще не получится. Рассмотрим, например, файл, состоящий из битов:

```
10010011100011100111...
```

В несжатом виде он будет занимать столько места (в битах), сколько в нем элементов. Если же мы попытаемся составить для него дерево Хаффмана, то получится безрадостная картинка вроде этой:



Длина кода для каждого элемента окажется равной одному биту; таким образом, добиться сжатия не удастся.

С другой стороны, дело обстоит не так уж и плохо: можно попытаться посмотреть на файл под иным углом зрения. Например, если рассматривать его не как набор битов, а как последовательность байтов, то количество разных элементов уже будет равно 256. Вполне возможно, что в этом случае кодирование Хаффмана покажет себя с лучшей стороны.

Как бы то ни было, алгоритм Хаффмана и в наше время остается очень популярным и часто используемым на практике, даже с учетом появившегося в семиде-

сятых годах *арифметического кодирования*, идею которого мы сейчас кратко обсудим.

Арифметическое кодирование

Хотя принцип арифметического кодирования высказывался еще Клодом Шенноном в его классической работе 1948 года по теории информации, первые серьезные практические реализации относятся к 1976 году – времени работ Паско (Pasco) и Риссанена (Rissanen).

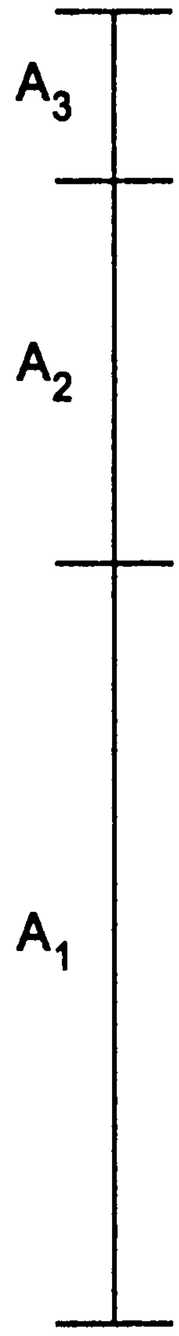
Как и в алгоритме Хаффмана, все начинается с таблицы элементов и соответствующих вероятностей. Допустим, входной алфавит состоит всего из трех элементов[1]: A_1, A_2 и A_3, и при этом

$$p(A_1) = 1/2$$
$$p(A_2) = 1/3$$
$$p(A_3) = 1/6$$

Предположим также, что нам надо сжать последовательность

$$A_1A_1A_2A_3$$

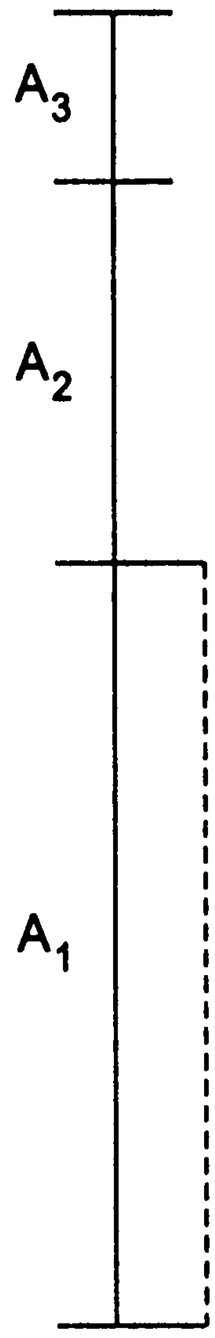
Разобьем полуинтервал [0, 1) на три части в соответствии с вероятностями элементов:



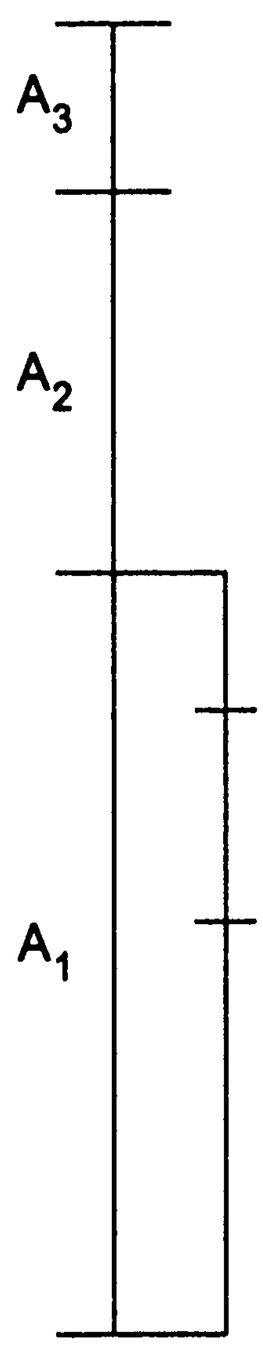
Элементу A_1 будет соответствовать диапазон [0, 1/2); элементу A_2 – диапазон [1/2, 1/2 + 1/3), а элементу A_3 – диапазон [1/2 + 1/3, 1).

[1] Разумеется, как и в предыдущем случае, это лишь пример; количество элементов алфавита может быть любым.

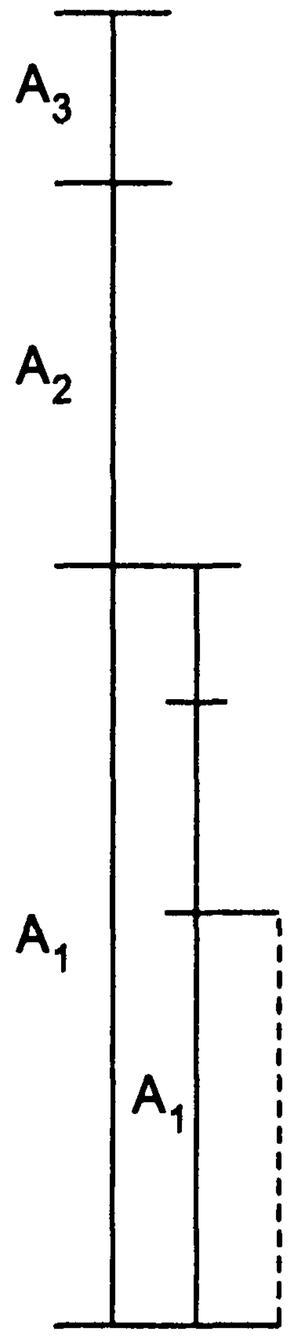
Первый элемент сжимаемого файла равен A_1. Выберем полуинтервал, соответствующий ему ([0, 1/2)):



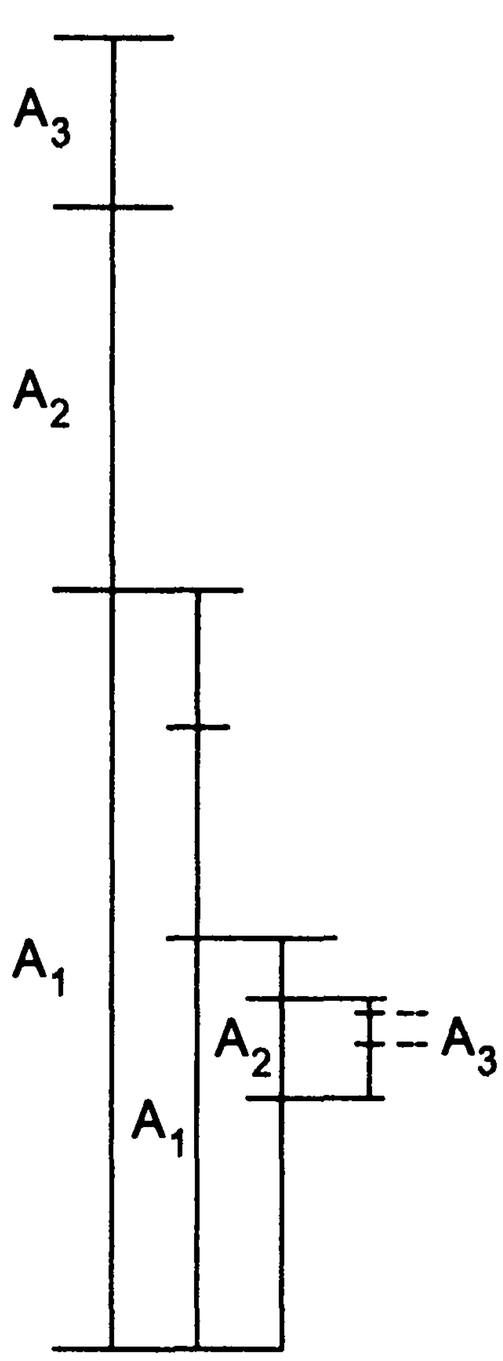
Разобьем его таким же образом на три части:



Поскольку второй элемент сжимаемой последовательности тоже равен A_1, снова выбираем самый нижний диапазон (точнее, теперь уже поддиапазон [0, 1/4)):



Аналогичным образом, продолжая процесс для оставшихся элементов входного файла (A_2 и A_3), получаем такой рисунок:



Последним полученным диапазоном будет [5/36, 5/24) (7/36 — это примерно 0,1944, а 5/24 — 0,2083). Теперь — внимание! *Любое* число из этого диапазона (например, 0,2) однозначно описывает *весь* входной файл. Зная вероятности появления элементов, декодер может правильно разбить диапазон [0, 1) на три части и определить, что число 0,2 находится в поддиапазоне [0, 1/2). В свою очередь, это означает, что первый элемент декодированного файла равен A_1. Затем декодер определит, что 0,2 также принадлежит интервалу [0, 1/4), следовательно, следую-

щий элемент выходного файла — снова A_1, и т. д. Таким образом, описываемая процедура на каждом шаге сужает рабочий диапазон, разбивает его в соответствии с вероятностями появления элементов и выбирает очередной элемент-поддиапазон в соответствии с числом, полученным от кодировщика. Поскольку декодер не может точно определить, когда следует закончить работу (процесс при желании можно продолжать бесконечно), придется либо явно прописать во входном файле количество элементов, подлежащих декодированию, либо ввести специальный символ «конец файла», встретив который, распаковщик остановится.

Хотя для описания сжимаемого файла годится любое число конечного диапазона (0,195, 0,19777, 0,199999999, ...), выбирать надо, разумеется, то, которое содержит меньше цифр, поскольку каждая лишняя цифра увеличивает размер выходного файла. В работах, посвященных арифметическому кодированию, доказывается утверждение о том, что количество битов, требуемое для хранения самого короткого «подходящего» числа из конечного диапазона, равно размеру сжатого файла при оптимальном кодировании в соответствии с формулами энтропии, округленному до большего целого. Таким образом, арифметическое кодирование является *оптимальным*: с его помощью можно достичь теоретических пределов сжатия, определяемых по формулам.

Кодирование алгоритмом Хаффмана сопоставляет каждому элементу входного файла некоторую битовую последовательность. Поскольку количество битов — всегда целое число, длина такой последовательности не может быть в точности равна энтропии элемента (если только сама энтропия случайно не оказалась целым числом). Любое отклонение от энтропии уже означает неоптимальное кодирование. Арифметическое же кодирование создает одну-единственную кодовую последовательность для всего файла. Длина этой последовательности также является целым числом; однако согласитесь, что один лишний бит во всем сжатом файле (полученный в результате округления) погоды не сделает, а лишний бит в *каждом* элементе — очень даже может. Впрочем, не стоит преувеличивать: многие исследователи отмечают, что алгоритм Хаффмана работает в большинстве случаев очень хорошо (получается *почти* оптимальное кодирование).

Теперь поговорим о плюсах и минусах алгоритма арифметического кодирования. Основных плюсов два:

+ оптимальное хранение сжатого файла в соответствии с формулами энтропии;
+ хорошая сочетаемость с адаптивной схемой сжатия: в отличие от процедуры Хаффмана, к примеру, никаких деревьев обновлять в процессе работы не надо; достаточно пересчитывать значения в таблице вероятностей.

Минусы, к сожалению, тоже есть:

– реализации арифметического кодирования работают, вообще говоря, медленно (в первую очередь, потому, что приходится использовать не самые скоростные операции умножения и деления);
– арифметическое кодирование нельзя распараллелить: если в компьютере установлено два физических процессора, вам не удастся разделить входной файл пополам, закодировать каждую половину отдельно, а затем объединить результаты[1];
– реализация кодировщика не так проста, как хотелось бы.

[1] С другой стороны я уверен, что эта проблема безразлична подавляющему большинству читателей.

Проблемы с реализацией возникают потому, что нельзя напрямую запрограммировать описанный алгоритм. Мы не можем, просто анализируя элемент за элементом входную последовательность, определить конечный интервал и записать число из него в результирующий файл, поскольку количество разрядов найденного числа может быть очень и очень велико[1]. Во-первых, это непозволительная трата памяти (сжимать приходится и гигабайты данных), а во-вторых, операции умножения и деления, постоянно выполняемые с числами такого размера, займут целую вечность.

На практике используют (в общих чертах) такую методику: как только в процессе анализа входного файла становится известна первая двоичная цифра результирующего числа, ее тут же записывают в выходной файл, а текущий интервал масштабируют до размера [0, 1), чтобы не скапливалось слишком много знаков после запятой. К сожалению, и тут не все просто: бывает, что очередную двоичную цифру слишком долго не удается определить. Для таких случаев существуют свои ухищрения.

Я решил не вдаваться в технические подробности реального арифметического кодировщика[2]. Если кого-то они очень интересуют, могу посоветовать прекрасный текст «Practical Implementations of Arithmetic Coding» (авторы – Paul G. Howard и Jeffrey S. Vitter). Есть несколько хороших работ на русском языке (как принадлежащих перу отечественных исследователей, так и переводных); думаю, что большинство из них (если не все) можно найти на сервере «Все о сжатии» (www.compression.ru).

Принципы моделирования

Полагаю, мы добрались до самого интересного. Подведем итоги того, что уже было сказано.

1. Наилучших результатов можно добиться, если создавать программу сжатия для какого-то конкретного типа данных. При этом можно улучшить сжатие, используя знания о специфике входных файлов.
2. Сжатие состоит из двух этапов: моделирования и кодирования.
3. На этапе моделирования мы определяем тип элементов, из которых состоит входной файл, а также схему вычисления вероятностей их появления (варианты: статическая, полуадаптивная или адаптивная).
4. Построив модель, можно определить размер наилучшим образом сжатого файла в соответствии с моделью, используя формулы энтропии. Программировать реальный кодировщик для этой цели нет необходимости.
5. Построив для одного и того же исходного файла две разные модели, можно получить два разных размера результирующего файла. Таким образом, степень сжатия зависит от сконструированной модели.

---

[1] Не забывайте, что число, о котором идет речь, *и есть* сжатый файл, поэтому для его хранения требуется объем памяти, обычно сопоставимый с объемом исходного файла.

[2] Не хочу отпугнуть вас от этой темы — не так уж она и сложна. Просто в главе о сжатии мне хочется акцентировать внимание совсем на других вещах.

6. Обладая сведениями об используемой модели, алгоритм Хаффмана может сохранить исходный файл близким к оптимальному образом в большинстве случаев. Арифметический кодировщик всегда может сохранить исходный файл оптимальным образом.

Если полагать все файлы состоящими из битов, байтов или машинных слов, хорошего результата не добьешься. К процессу моделирования надо подходить творчески, используя все знания об особенностях типа сжимаемых данных. О том, какие инструменты есть в нашем распоряжении, сейчас и пойдет речь.

Основные идеи

Почему одни файлы сжимаются хорошо, другие — плохо, а третьи не сжимаются совсем? От каких параметров зависит степень сжатия? Что мы можем сделать с файлом для того, чтобы он сжимался лучше? Ответы оказываются на удивление простыми. При помощи формул энтропии можно посчитать размер файла при оптимальном хранении (то есть сжатого файла). Средняя энтропия элемента зависит лишь от значений вероятностей появления каждого элемента модели. Иными словами, определив эти вероятности, мы уже можем точно предсказать, насколько хорошо сожмется данный файл. К примеру, если распределение вероятностей появления элементов в некотором файле похоже на то, что показано на рис. 5.2, ничего хорошего ожидать не приходится: данные окажутся практически несжимаемыми.

Рис. 5.2. Пример неблагоприятного распределения вероятностей

Дело в том, что все вероятности из примера близки друг к другу, то есть распределены более или менее равномерно. В экстремальном случае, когда все вероятности равны между собой, сжатия не будет вообще. Представьте себе, к примеру, что мы сжимаем поток байтов, причем с одной и той же вероятностью мы можем встретить любой элемент. Поскольку всего разных элементов 256, вероятность появления любого из них равна 1/256. Подсчитаем энтропию:

$$H(\text{элемента}) = -\log_2(1/256) = 8 \text{ (бит)}$$

Таким образом, модель «радует» нас перспективой сжать любой элемент (то есть любой байт) «всего» в 8 бит!

С другой стороны, неравномерное распределение вероятностей (рис. 5.3) – очень обнадеживающий знак.

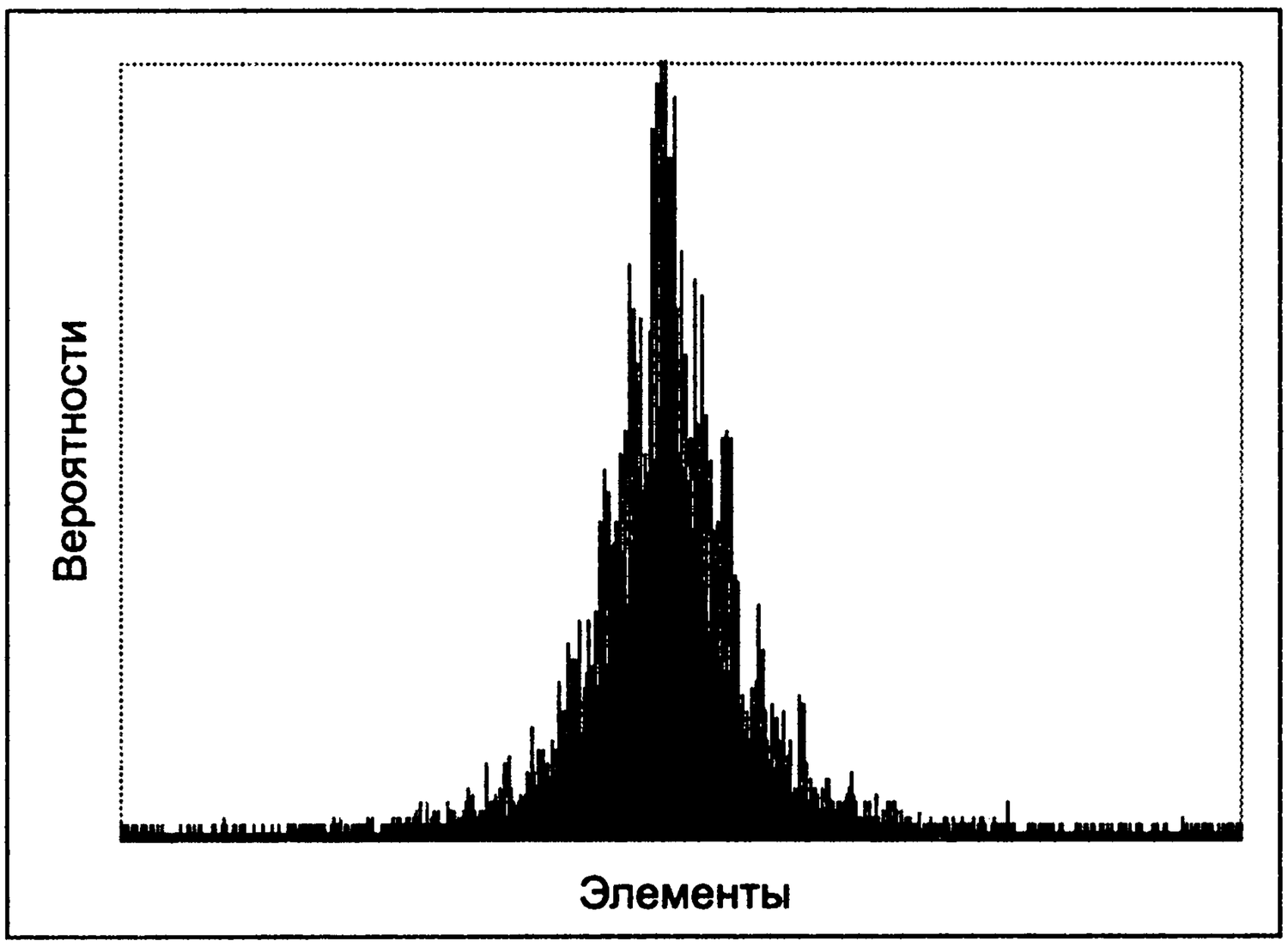


Рис. 5.3. Пример благоприятного распределения вероятностей

Отсюда вывод: чтобы хорошо сжать файл, надо научиться видеть в нем совокупность таких элементов, которые распределены неравномерно. Не исключено при этом, что файл придется специальным образом преобразовать. В этом и состоит искусство, творческий подход, о котором я говорил. Большую помощь на пути совершенствования в искусстве сжатия вам окажут знания о таких понятиях, как *замена алфавита*, *контекстное моделирование* и *предиктивное моделирование*.

Замена алфавита

Суть замены алфавита состоит в изменении угла зрения, под которым мы рассматриваем входной файл. Допустим, требуется сжать текстовый документ. Из чего он состоит? Из отдельных символов? Возможно. Из двухсимвольных сочетаний? Вполне может быть. Из слов, пробелов, цифр и знаков препинания? Спорно, но допустим. Посмотрите на файл по-другому; не позволяйте стереотипам завладеть вашим мышлением. Может быть, вам удастся взглянуть на обыкновенный текстовый файл абсолютно новым, оригинальным образом и получить совершенно фантастическое распределение вероятностей.

Наверное, самый лучший пример замены алфавита – методика RLE (Run-Length Encoding, групповое кодирование, или кодирование длин серий), широко применяемая в компьютерной графике. Предположим, требуется сжать некоторое черно-белое изображение (рис. 5.4).

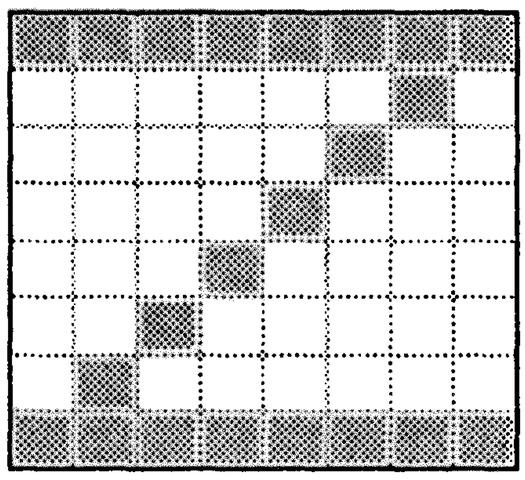


Рис. 5.4. Сжимаемое изображение

Самый простой подход состоит в том, чтобы полагать картинку состоящей из черных и белых пикселов. Допустим. Давайте оценим, насколько хорошо удастся сжать данные при использовании такой модели:

p(черного пиксела) = 22/64 = 11/32
p(белого пиксела) = 42/64 = 21/32

H(черного пиксела) = $-\log_2(11/32) \approx 1.540$
H(белого пиксела) = $-\log_2(21/32) \approx 0.608$

$H_{элемента} \approx (11/32)*1.540 + (21/32)*0.608 = 0.928$
размер файла = 0.928 * 64 = 59.416 (бит)

Надеюсь, вы еще помните, как вычисляется размер итогового файла. Сначала определяются энтропии отдельных элементов, затем находится средняя энтропия, а потом средняя энтропия умножается на размер файла в элементах (в данном случае размер равен 64 пикселам).

Как видите, степень сжатия оказалась не слишком впечатляющей. Черно-белое несжатое изображение займет 64 бита, а сжатое — «всего» 60[1].

Теперь посмотрим на рисунок под другим углом зрения. Будем полагать, что он состоит не из отдельных пикселов, а из *последовательностей* черных и белых точек, каждая из которых определяется только своей длиной. Картинка всегда начинается с последовательности какого-то количества белых пикселов, затем идет последовательность черных, затем снова белых и т. д. Данные во входном файле хранятся строка за строкой; информация о том, что очередная строка закончена, нигде не содержится (если декодер знает ширину картинки, она ему и не нужна).

Таким образом, первым элементом входного файла будет число 0 (изображение начинается с нуля белых пикселов), вторым — 8 (затем идет восемь черных пикселов), третьим — 6 (шесть белых пикселов), четвертым — 1 (один черный пиксел) и т. д:

0 8 6 1 6 1 6 1 6 1 6 1 6 1 6 8 (длина файла – 16 элементов)

Итак, теперь элементами входного файла являются не пикселы, а длины последовательностей: 0, 1, 6 и 8. Определим размер сжатого файла:

p(0) = 1/16
p(1) = 6/16
p(6) = 7/16
p(8) = 2/16

---

[1] Хочу заметить, что я полагаю размеры картинки известными и кодировщику и декодеру.

$H(0) = -\log_2(1/16) = 4$
$H(1) = -\log_2(6/16) \approx 1.415$
$H(6) = -\log_2(7/16) \approx 1.193$
$H(8) = -\log_2(2/16) = 3$

$H_{элемента} \approx (1/16)*4 + (6/16)*1.415 + (7/16)*1.193 + (2/16)*3 = 1.678$
размер файла = 1.678 * 16 = 26.841 (бит)

Что ж, на сей раз файл оказался сжатым более чем в два раза; прогресс налицо. Осталось лишь применить к новой модели и входному файлу алгоритм Хаффмана или арифметическое кодирование и получить результат.

На практике эта оценка оказывается слишком оптимистичной. Вам удастся сжать файл в 27 бит лишь при использовании статической схемы сжатия (когда таблица вероятностей жестко запрограммирована в архиваторе). Если же применяется полуадаптивная схема, придется прибавить размер таблицы вероятностей; адаптивная схема окажется эффективной, если повезет (если по начальным данным сжимаемого файла удастся составить правильное представление об оставшихся элементах).

Заметьте, что элементом файла при использовании RLE может быть любое целое неотрицательное число, поэтому размер алфавита потенциально неограничен. Само по себе это не страшно, однако при использовании полуадаптивной схемы сжатия таблица вероятностей может слишком сильно вырасти в объеме. В этом случае можно легко ограничить ее размер, запретив использование слишком больших длин последовательностей. Предположим, что нам встретилось N, скажем, черных пикселов, и следующий пиксел тоже оказался черным. Тогда можно представить эту ситуацию последовательностью

```
N 0 ...
```

(N черных пикселов, затем 0 белых, затем снова идут черные). Теперь в таблице вероятностей будут находиться только числа [0...N].

Обязательно запомните этот мощный метод — замену алфавита. В принципе, оставшиеся методы, которые мы рассмотрим, являются в какой-то мере вариациями замены алфавита, хотя и не такими прямолинейными.

Контекстное моделирование

До сих пор мы не говорили об одной важной особенности многих типов данных — связи вероятности появления каждого элемента с его *контекстом* (то есть соседними элементами). В файле, заполненном случайными числами, вы, скорее всего, не найдете никакой корреляции (то есть взаимозависимости) между его элементами. Однако это весьма нетипичный случай — скорее всего, какая-то зависимость есть. Например, в текстовом документе после запятой или точки, скорее всего, будет следовать пробел; с другой стороны, после буквы «ж» вы навряд ли найдете «ы», а после «й» — «г». То же самое верно для многих других данных, например для изображений. Представьте себе, что текущая точка черно-белой картинки слева и сверху окружена лишь белыми точками (рис. 5.5, слева).

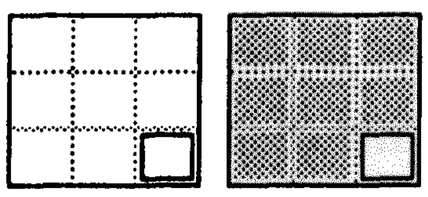


Рис. 5.5. Зависимость цвета пиксела от его контекста

С большой долей вероятности можно утверждать, что и рассматриваемый пиксел будет белым; аналогичное утверждение справедливо и для черных соседних точек (рис. 5.5, справа). Изображения, кстати, во многом интересны своей двумерной структурой: если в тексте под контекстом обычно понимают сколько-то предыдущих символов, то в картинке мы можем использовать пикселы сверху и слева от текущего. Кстати, использовать элементы, *следующие за* текущим, в качестве контекста нельзя. Дело в том, что декодер должен обладать полной информацией о контексте. При декодировании некоторого элемента распаковщик уже знает о значениях предыдущих элементов, но не знает о величинах следующих.

Подобными зависимостями вполне можно воспользоваться при сжатии, но для этого нам сначала придется познакомиться с некоторыми новыми понятиями.

Во-первых, мы будем оперировать уже упомянутой концепцией *контекста*. Под контекстом понимается определенное количество элементов, расположенных до текущего. Основная характеристика контекста — его размер, то есть число составляющих его элементов. Например, контекст может состоять из одного элемента, непосредственно предшествующего текущему, или, скажем, из двух элементов, обработанных на двух предыдущих шагах.

Поскольку количество разных контекстов в пределах файла ограничено, а один и тот же контекст может встретиться несколько раз, можно ввести понятие *вероятности появления контекста*, аналогичное концепции вероятности появления отдельного элемента: $p(C_i)$. Предположим, в исходном файле всего встречается десять различных контекстов, при этом контекст C_1 появился лишь один раз. В этом случае $p(C_1) = 1/10$.

Во-вторых, на смену понятию вероятности появления элемента в контекстном моделировании приходит вероятность появления элемента *в данном контексте*: $p(A_i \mid C_j)$. Смысл этой записи прост: нас больше не интересует вероятность появления элемента A_i во всем файле, а лишь в данном контексте C_j. Если элемент A_i встретился в другом контексте (C_k) — это уже будет иная величина: $p(A_i \mid C_k)$. Предположим, в текстовом файле контекстом считается одна буква, предшествующая текущей. Допустим также, что в трех случаях из пяти после буквы «б» встречается буква «а». Тогда p(«а» | «б») = 3/5.

При использовании контекстного моделирования создается столько различных моделей[1], сколько есть разных контекстов. В процессе работы кодировщик определяет текущий контекст и пользуется соответствующей моделью для сжатия (например, методом Хаффмана) очередного элемента. Мы уже выяснили, что алгоритмы кодирования (арифметического и по Хаффману) требуют на входе только сжимаемые данные и таблицу вероятностей элементов; при этом для них

[1] То есть наборов элементов и их вероятностей.

неважно, чем именно являются эти вероятности — какими-то глобальными величинами или значениями, имеющими смысл только внутри некоторого контекста.

Для любой разработанной контекстной модели тоже можно определить расчетный размер сжатого файла, хотя это и несколько сложнее. Размер файла в битах, как и раньше, вычисляется как размер файла в элементах, умноженный на среднюю энтропию одного элемента. Но сама энтропия элемента определяется другим способом:

$$H_{элемента} = p(C_1) * H(C_1) + p(C_2) * H(C_2) + \ldots + p(C_n) * H(C_n)$$

Здесь $p(C_i)$ — это вероятность появления контекста C_i во входном файле. $H(C_i)$ — это новая величина, *энтропия контекста*. Если в некотором контексте встречается лишь один возможный элемент (например, после буквы «а» в тексте всегда идет буква «б»), энтропия такого контекста равна нулю. В противном случае величина энтропии определяется по формуле:

$$H(C_i) = -(p(A_1|C_i) * \log_2(p(A_1|C_i)) + p(A_2|C_i) * \log_2(p(A_2|C_i)) + \ldots + p(A_k|C_i) * \log_2(p(A_k|C_i)))$$

Здесь $A_1 \ldots A_k$ — элементы, которые встречаются в данном контексте. Если некоторый элемент встречается только в других контекстах, в формулу энтропии C_i он не входит.

Вот, пожалуй, и все. В качестве примера попробуем применить контекстное моделирование к картинке, изображенной на рис. 5.4. Возьмем в качестве контекста три соседних с текущим пиксела: верхний, левый и расположенный влево-вверх по диагонали. Таким образом, возможны восемь различных контекстов (рис. 5.6).

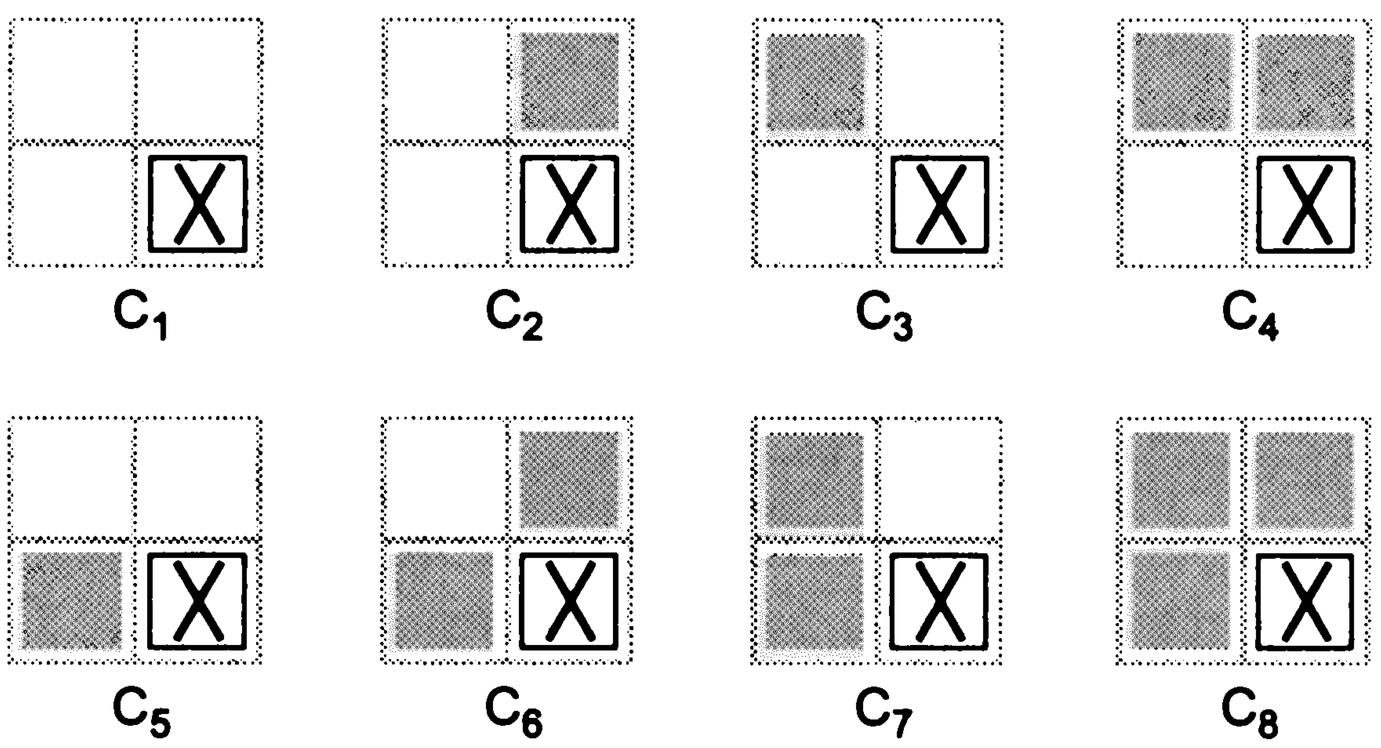


Рис. 5.6. Контексты в сжимаемом изображении

Крестиком отмечена позиция текущего пиксела. Примем одно обычное в таких случаях допущение: везде за пределами картинки находятся белые точки. Поэтому, к примеру, пиксел, находящийся в верхнем левом углу сжимаемого изображения, относится к контексту C_1. Анализируя входные данные, определяем вероятность появления того или иного контекста:

$p(C_1) = 32/64$
$p(C_2) = 1/64$
$p(C_3) = 5/64$
$p(C_4) = 6/64$

$p(C_5) = 5/64$
$p(C_6) = 6/64$
$p(C_7) = 1/64$
$p(C_8) = 1/64$

Теперь нам потребуются вероятности цвета текущего пиксела в каждом контексте:

C_1 : $p(\text{белый} \mid C_1) = 25/32$, $p(\text{черный} \mid C_1) = 7/32$
C_2 : встречается только белый цвет
C_3 : встречается только белый цвет
C_4 : $p(\text{белый} \mid C_4) = 5/6$, $p(\text{черный} \mid C_4) = 1/6$
C_5 : встречается только черный цвет
C_6 : $p(\text{белый} \mid C_6) = 5/6$, $p(\text{черный} \mid C_6) = 1/6$
C_7 : встречается только черный цвет
C_8 : встречается только белый цвет

Осталось только аккуратно воспользоваться приведенными выше формулами:

$H(C_1) = -((25/32) * \log_2(25/32) + (7/32) * \log_2(7/32)) \approx 0.758$
$H(C_2) = 0$
$H(C_3) = 0$
$H(C_4) = -((5/6) * \log_2(5/6) + (1/6) * \log_2(1/6)) \approx 0.650$
$H(C_5) = 0$
$H(C_6) = -((5/6) * \log_2(5/6) + (1/6) * \log_2(1/6)) \approx 0.650$
$H(C_7) = 0$
$H(C_8) = 0$

$H_{\text{элемента}} = (32/64)*0.758 + (6/64)*0.650 + (6/64)*0.650 = 0.500$
размер файла (в битах) = $64*0.5 = 32$

Получается, что исходный файл сожмется вдвое.

Контекстное моделирование — очень мощный инструмент (хотя в данном конкретном случае методика RLE обеспечила более сильное сжатие), пригодный в самых разнообразных ситуациях. Я думаю, что в подавляющем большинстве файлов, содержащих осмысленные данные, существует какая-то корреляция между элементами. Искусство контекстного моделирования состоит, прежде всего, в правильном выборе контекста. Например, вероятно, что между двумя соседними символами текстового файла существует какая-то взаимосвязь; а вот между буквами, разделенными десятью другими, скорее всего, заметной зависимости обнаружить не удастся.

Наверное, вы уже заметили самую большую проблему контекстного моделирования: с увеличением количества элементов контекста размер таблицы вероятностей растет экспоненциально. Если для формирования контекста оказалось достаточно одного-двух элементов — считайте, что вам крупно повезло: обычно с такими маленькими контекстами хорошего результата не достичь. С другой стороны, чем больше контекст, тем сильнее разрастается таблица вероятностей, поэтому при использовании полуадаптивной схемы сжатия о серьезном применении контекстного моделирования говорить не приходится. Зато, если удастся свести задачу к статической схеме, возможности для фантазии значительно расширяются.

Предиктивное моделирование

Закончить обзор наших инструментов мне хочется предиктивным моделированием – замечательным эвристическим (то есть воплощающим разумные идеи, но ничего не гарантирующим) средством, очень часто использующимся в реальных программах сжатия. Предиктивное моделирование – это интересная попытка совместить мощь контекстного моделирования со скромным размером таблицы вероятностей.

Мы уже говорили о том, что вероятность появления того или иного элемента входного файла может сильно зависеть от текущего контекста. А если так, можно попытаться *предсказать* значение следующего элемента, зная его контекст. Если в английском тексте встретилась буква «q», можно с большой долей вероятности утверждать, что следующей будет «u» (это почти всегда так); если пять пикселов подряд в изображении оказались белыми, опять-таки можно рассчитывать (хотя и нельзя утверждать), что и шестым будет белый.

Представьте теперь, что мы написали функцию-предиктор f(C), которая принимает на входе некоторый контекст C и возвращает элемент, который, по ее мнению, наиболее вероятен как следующий. Иными словами, она пытается предсказать значения следующих элементов, обладая информацией о предыдущих.

Поскольку все в компьютере, в конечном итоге, представляется в виде чисел, всегда можно каким-нибудь образом определить разность между двумя элементами. Например, разность между элементами-буквами может задаваться разностью между соответствующими ASCII-кодами; разность между элементами-пикселами – тройкой чисел, состоящей из разностей между RGB-компонентами, и т. д.

Разность между реальным значением очередного элемента и значением предсказанным (которое возвращает функция f(C)) называется *ошибкой предсказания*:

```
ошибка = очередной элемент – f(текущий контекст)
```

Зная ошибку предсказания и текущий контекст, можно определить очередной элемент:

```
очередной элемент = ошибка + f(текущий контекст)
```

Суть предиктивного моделирования состоит в том, чтобы кодировать не сами элементы входного файла, а ошибки предсказания. Обладая предсказывающей функцией и ошибками предсказания, декодер сможет восстановить исходный файл.

Хорошая предсказывающая функция делает много корректных предположений и небольших ошибок, но мало серьезных промахов. Таким образом, получается очень приятное глазу распределение вероятностей (аналогичное показанному на рис. 5.3):

```
p(ошибка = 0) – высокая
p(ошибка = -1 или 1) – ниже
p(ошибка = -2 или 2) – еще ниже
...
```

Рассмотрим работу предиктивного моделирования на примере сжатия все того же изображения, показанного на рис. 5.4. В качестве предиктора выберем значение пиксела, находящегося слева от текущего:

```
f(C)  = пиксел слева от текущего
```

Помня допущение о том, что область вне рисунка полностью состоит из белых точек, и полагая, что белому пикселу соответствует нуль, а черному – единица [1], создадим файл ошибок предсказания (рис. 5.7).

1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	–1
0	0	0	0	0	1	–1	0
0	0	0	0	1	–1	0	0
0	0	0	1	–1	0	0	0
0	0	1	–1	0	0	0	0
0	1	–1	0	0	0	0	0
1	0	0	0	0	0	0	0

Рис. 5.7. Файл ошибок предсказания

Осталось теперь уже привычным способом вычислить расчетный размер сжатого файла (напоминаю, мы кодируем файл ошибок):

```
p(0) = 50/64
p(-1) = 6/64
p(1) = 8/64

H(0) = -log2(50/64) ≈ 0.356
H(-1) = -log2(6/64) ≈ 3.415
H(1) = -log2(8/64) = 3

Hэлемента = (50/64)*0.356 + (6/64)*3 415 + (8/64)*3 = 0 973
размер файла (в битах) = 64 * 0.973 = 62 297
```

Как видите, в данном конкретном случае особого сжатия не получилось – что, впрочем, и неудивительно, поскольку исходный файл состоял всего из двух различных элементов. По-настоящему вся сила метода проявляется, когда входные данные состоят из сотен (а то и тысяч) почти равномерно распределенных величин, а предиктор позволяет почти бесплатно преобразовать это распределение к очень «хорошему» виду.

Кстати, насчет «почти бесплатности»: предиктивное моделирование в общем случае увеличивает вдвое таблицу вероятностей, поскольку разностей между элементами вдвое больше [2], чем самих элементов (это видно из примера). Мне кажется, что это не такая уж и большая плата; по крайней мере, двукратное увеличение таблицы вероятностей – все-таки не экспоненциальный рост в случае контекстного моделирования.

Составление хорошего предиктора – настоящее искусство. Иногда выбор функции предсказания очевиден, но, как правило, приходится хорошо изучить специфику входных данных и научиться видеть в них скрытые зависимости.

---

[1] Тогда разность черного и белого пикселов равна единице, а белого и черного — минус единице

[2] Точнее, 2*N – 1 разностей для N элементов.

На этой оптимистической ноте мы заканчиваем знакомство с технологиями сжатия информации. Конечно, я лишь затронул самые «верхушки» идей и алгоритмов, а многие идеи, основанные совсем на других принципах (например, основанные на словарях алгоритмы сжатия семейства LZ), остались за пределами главы. Тем не менее надеюсь, что общее впечатление о сжатии данных окажется у вас верным.

Проекты для самосовершенствования

1. Напишите простейший архиватор, который рассматривает любой входной файл как последовательность байтов и сжимает его, используя полуадаптивную схему сжатия и алгоритм Хаффмана. Само собой, реализуйте и распаковщик.
2. Система Delphi делает очень удобной работу с изображениями: вам ничего не стоит загрузить картинку в элемент типа TImage, а затем получить доступ к отдельным пикселам при помощи свойства Canvas.Pixels[x, y] (мы уже использовали эту технику во второй главе). Возможен и обратный процесс: программным способом на объект типа TImage выводится некоторый рисунок, а затем сохраняется на диске вызовом Объект.Picture.SaveToFile(filename). Используя возможности Delphi, напишите архиватор для черно-белых изображений. Реализуйте в архиваторе RLE-кодирование, полуадаптивную схему сжатия и алгоритм Хаффмана.
3. Постройте контекстную модель для документа на русском языке. Контекстом считайте два предшествующих текущему символа. Оцените, насколько контекстное моделирование оправдывает себя в таком случае (сравните с результатом прямолинейного кодирования исходного файла как потока байтов). Определите, можно ли применить статическую схему сжатия (иными словами, проверьте, сильно ли отличаются контекстные модели, построенные для разных документов).
4. Звуковой файл формата WAV (8 бит, без сжатия, моно) имеет очень простую структуру. Сначала хранится 44-байтный заголовок, а затем – поток 8-битовых чисел (то есть, попросту, байтов), представляющих собой значения амплитуды в данный момент времени. В большинстве звукозаписей прослеживается такая тенденция: если амплитуда начала уменьшаться, то, скорее всего, она будет уменьшаться и дальше в течение некоторого времени; то же самое верно и для роста амплитуды. Используя этот факт, примените предиктивное моделирование к WAV-файлам. В качестве функции-предиктора попробуйте для начала

$$f(C_1, C_2) = C_1 + (C_1 - C_2)$$

где C_1 – это предыдущий элемент, а C_2 – элемент, предшествующий предыдущему. Заголовок WAV сохраните в сжатом файле без изменений. Оцените эффективность архиватора.

Глава 6

Алгоритмы на графах

Но это не тот граф, который в замке живет,
а совсем другое...

Валерий Абрамович Оргун

Недавно мне на глаза попалась книга под забавным названием «Конкретная математика». Во времена создания курса (70-е годы XX века), на основе которого позже была написана эта книга, математические дисциплины в лучших университетах мира подвергались жесткой критике. Под влиянием стремительно развивающейся компьютерной индустрии пошатнулись многолетние устои в преподавании математики. Стало очевидным, что акценты сильно сместились: то, что раньше считалось важным, оказалось никому не нужным, а то, на что математики практически не обращали внимания, вдруг обрело невиданный спрос. Велись даже разговоры на темы вроде «может ли математика быть спасена?». Так или иначе, математика выжила, но взгляды на нее сильно изменились. Остается лишь сожалеть, что произошедшая революция (ладно, пусть лучше будет «переосмысление идеалов» — не буду столь категоричен) как-то прошла мимо нашей страны. До сих пор студенты учатся по книгам шестидесятых, а то и тридцатых годов (а ведь в книгах помимо теорем отражаются идеологические взгляды того времени на освещаемые проблемы). Авторы книги «Конкретная математика» выбрали из математики то, что они реально использовали в работе, и создали свой труд на основе лишь этих, избранных разделов. Надо сказать, что подборка тем озадачит специалиста, работающего в области «чистой» математики, но будет очень по душе практику.

К чему это я? О графах пишут много и в самых разных книгах. Теория графов существует уже давно и объем имеет весьма приличный. Здесь же я расскажу лишь о тех алгоритмах, которые мне реально пригодились на практике. Могу обещать: материал этой главы будет полезен многим.

Конечно, подборка не претендует на полноту — о самых что ни на есть полезных алгоритмах из копилки теории графов можно рассказать в десять раз больше, но будьте снисходительны: я не Грэхем, не Кнут и не Паташник, а книга, которую вы держите в руках, — далеко не «Конкретная математика».

Понятие графа

Под графом понимается некий объект, который можно при желании представить в виде точек (называемых *вершинами*), соединенных линиями (*ребрами*) (рис. 6.1).

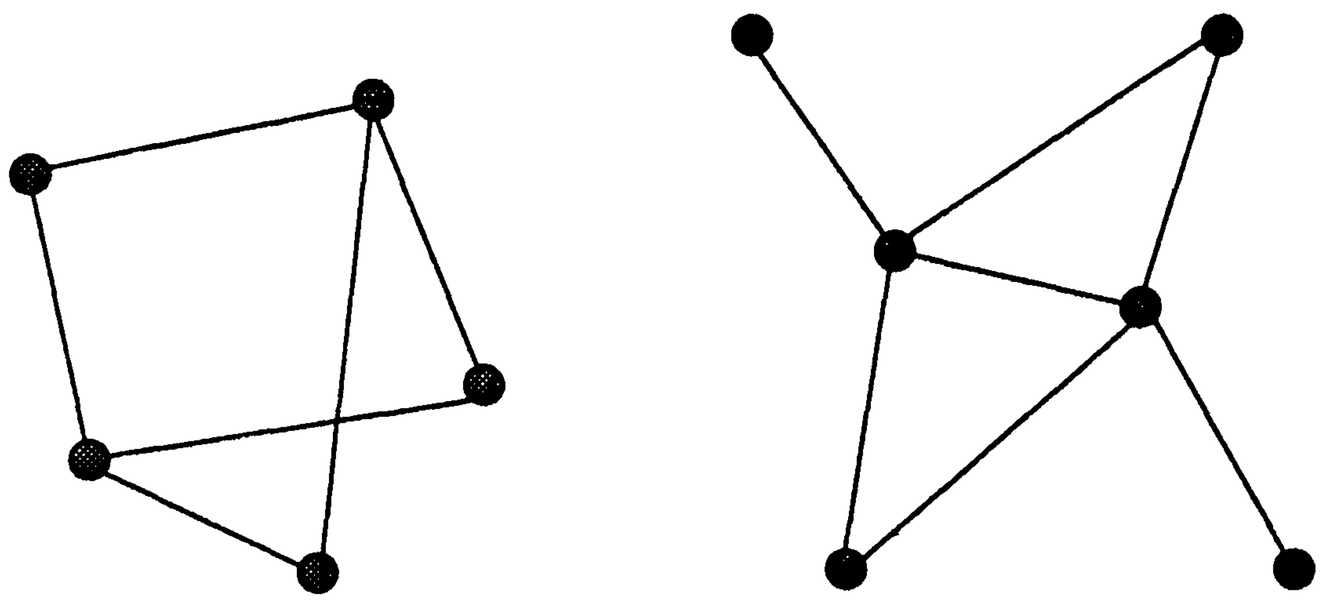


Рис. 6.1. Примеры графов

Я написал «при желании», потому что граф — это нечто большее, чем простой рисунок из точек и линий. Рисунок играет роль, скажем так, схематического чертежа представляемого объекта, точно так же, как черточки на карте города символизируют собой асфальтовые дороги или железнодорожные пути. Вершины и ребра можно помечать любыми числами и надписями, символизирующими собой какие-либо параметры объекта (например, на карте отмечаются названия улиц и важных учреждений).

Графы бывают разные — ориентированные, взвешенные, связные... нас эта классификация до поры до времени интересовать не будет. Если в каком-либо алгоритме вид графа имеет существенное значение, я оговорю это явно.

Интересуют же нас графы потому, что многие реальные объекты мира могут быть представлены с их помощью; соответственно, целый ряд насущных задач хорошо формулируется на языке теории графов (а затем небезуспешно решается).

Сейчас мы рассмотрим некоторые из них.

Задача Прима—Краскала (о телефонной сети)

В некоторой стране есть *N* городов. Требуется соединить их телефонной сетью, минимизируя при этом расходы на провода. В каждом городе расположена всего одна телефонная станция; от нее может отходить любое количество проводов, ведущих в другие города. Телефонное соединение между городами не обязательно должно быть прямым: если город A связан с городом B, а город B – с городом C, можно смело полагать, что A связан с C. Расстояние между любыми двумя городами (соответственно и требуемая длина провода) считается известным.

Невооруженным взглядом видно, что задача такого рода хорошо описывается в виде графа. Допустим, в стране есть города A, B, C и D. Расстояние между A и B

равно 100 км, между A и C – 150 км, между B и C – 100 км, между B и D – 200 км, между C и D – тоже 200 км, а между городами A и D находится непроходимый лес. Считая города вершинами графа, получаем фигуру, изображенную на рис. 6.2.

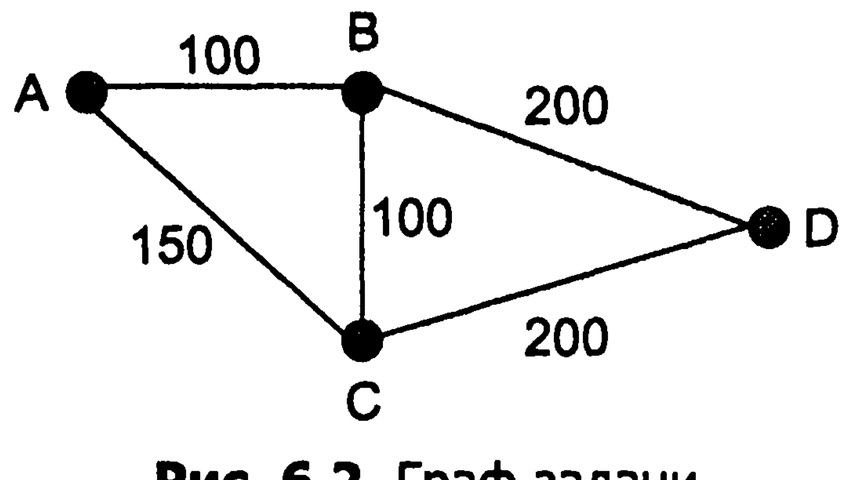


Рис. 6.2. Граф задачи

Наличие ребра между вершинами символизирует возможность установить прямую связь между соответствующими городами; число, используемое в качестве метки, означает расстояние. Обратите внимание, что ребра A-D не существует, поскольку по условию эти два города запрещено связывать напрямую.

Разумеется, приведенный граф не стоит считать географической картой страны. Возможно, город A на самом деле находится на западе, а D – на востоке. В нашем случае интересны лишь расстояния между городами, а не их расположение. Кроме того, расстояния между городами «по карте» могут не совпадать с длинами проводов: если провод придется тянуть через гору, его длина будет явно больше.

Прежде чем рассмотреть способ решения задачи, сделаем два допущения.

1. Предполагается, что решение существует. Иными словами, нет такого города (или группы городов), который был бы отделен от остальных непроходимым лесом или зоной Вселенского Коллапса: мы считаем, что каким-либо способом сеть провести можно. На языке математики это означает требование *связности* графа.
2. Предполагается, что расстояние от города A до города B равно расстоянию от B до A (требование *неориентированности* графа). В задаче о телефонной сети такое допущение вполне естественно [1], однако не забывайте, что в других случаях оно может и не выполняться. Например, в какой-нибудь модели, связанной с путешествиями, нас может интересовать не расстояние между городами, а время, проведенное в пути. Если город A находится на горе, а город B – в низине, то путешествие из A в B, очевидно, займет меньше времени, чем путешествие из B в A.

Еще нам потребуется (неформальное) определение *цикла* в графе. Если, двигаясь из какой-либо вершины графа по ребрам (при этом дважды проходить по одному и тому же ребру запрещается), вы можете снова попасть в нее, то рассматриваемый граф называется *циклическим*, а ваш путь – *циклом*. Граф, изображенный на рис. 6.2, является циклическим (в нем даже не один, а несколько циклов). К примеру, выходя из вершины A, вы можете легко в нее вернуться: A ▸ B ▸ C ▸ A.

[1] Более того, мы уже неявно им воспользовались, когда сопоставляли задаче граф, изображенный на рис. 6.2.

Поскольку в задаче Прима–Краскала идет поиск минимальной телефонной сети, полученный в результате граф (то есть собственно сеть) не должен содержать циклов. В противном случае сеть заведомо не будет минимальной, так как из цикла всегда можно убрать любое ребро, не нарушая телефонной связи между городами. Допустим, вам предлагают решение, в котором проводами прямо соединяются города A и B, B и C, C и A. Тогда вы можете легко улучшить его, убрав любое из этих трех соединений; все равно все три города останутся «на связи».

Теперь обсудим решение задачи (алгоритм Прима–Краскала):

```
ЦИКЛ N - 1 раз
    добавить в сеть самое короткое ребро, удовлетворяющее условиям:
        1) оно не было добавлено ранее
        2) с добавлением ребра в сети не появится цикл
КОНЕЦ ЦИКЛА
```

Если количество городов невелико (как в примере), можно проделать описанные действия вручную (рис. 6.3).

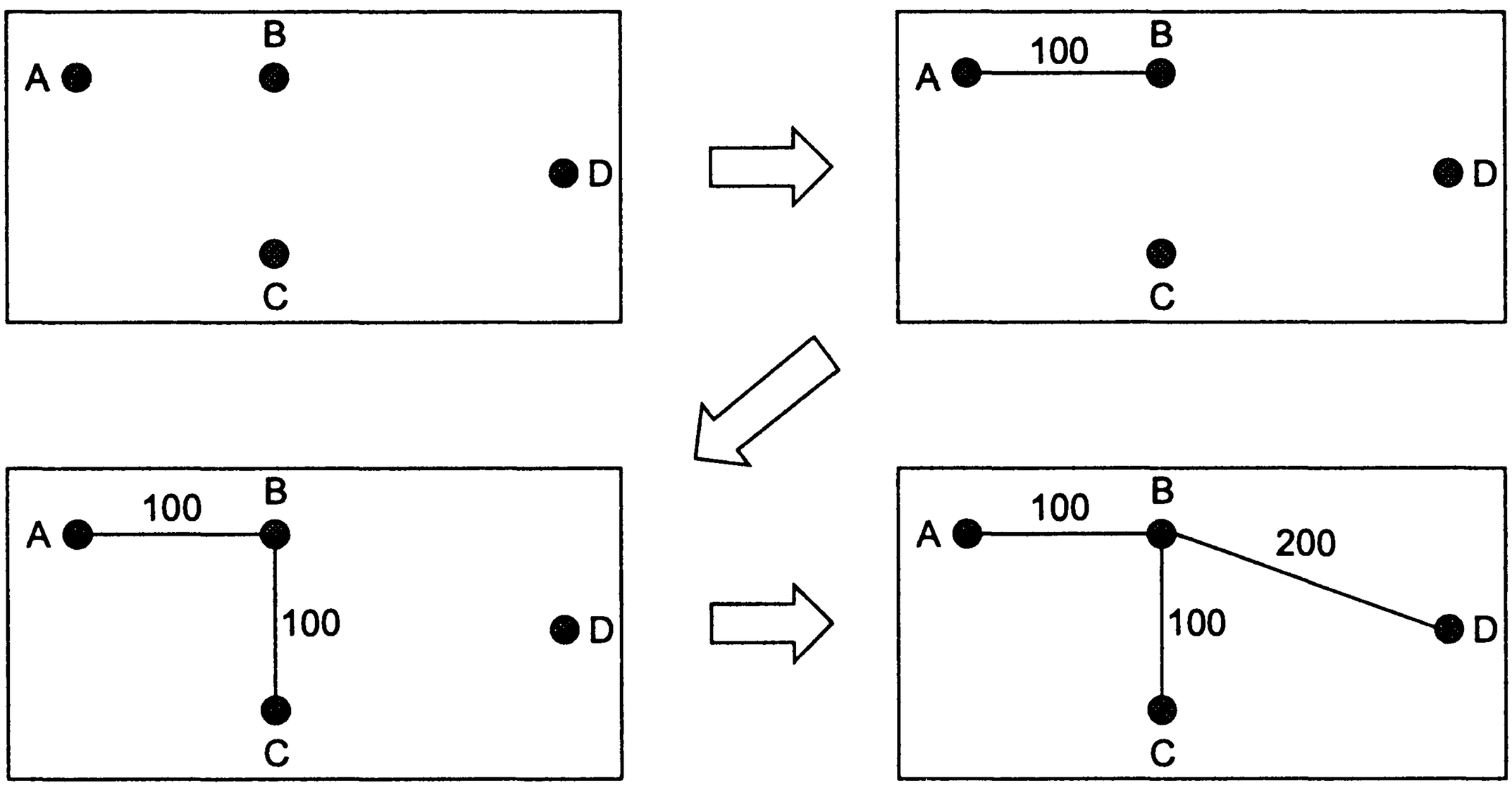


Рис. 6.3. Нахождение минимальной телефонной сети алгоритмом Прима—Краскала

Если на каком-либо шаге требуемым условиям удовлетворяют несколько ребер, можно выбрать любое из них.

Конечно, в «настоящих» задачах карандашом и бумагой уже не обойтись. Поэтому следующая цель – реализация алгоритма на компьютере.

Прежде всего, подумаем, как можно представить граф в памяти. С одной стороны, никто не ограничивает вашей творческой свободы; с другой же – существуют некоторые стандартные способы, пригодные во многих случаях. Один из них – так называемая *матрица смежности*. Представьте себе, что все вершины графа перенумерованы целыми числами. Тогда граф можно представить в виде квадратной матрицы M, элемент (i, j) которой содержит вес ребра[1], соединяющего

[1] То есть число, соответствующее ребру. В нашем случае — расстояние между городами.

вершины i и j. Если некоторого ребра не существует, придется вместо веса внести в матрицу какое-то особое значение. Выбор, в принципе, сильно зависит от задачи: в нашем случае можно просто использовать очень большое число («машинную бесконечность»). Это даст уверенность в том, что алгоритм Прима–Краскала никогда не выберет такое ребро.

Рассмотрим, к примеру, матрицу смежности, соответствующую графу с рис. 6.2 (на который я постоянно ссылаюсь):

```
0       100     150     10^7
100     0       100     200
150     100     0       200
10^7    200     200     0
```

Города нумеруются естественным образом: A – 1, B – 2, C – 3, D – 4. Поэтому расстояние, скажем, между городами A и C, равно M[1, 3]. Главную диагональ матрицы обычно заполняют нулями (расстояние от любого города до него же равно нулю). В качестве «машинной бесконечности» я выбрал десять в седьмой степени – число, явно превосходящее любые величины в нашей задаче. Обратите внимание, что матрица смежности для неориентированных графов симметрична (поскольку расстояние от города i до города j равно расстоянию от j до i).

Для любителей спортивной статистики не могу не провести аналогии между матрицей смежности и сводной таблицей результатов какого-нибудь футбольного турнира[1] (рис. 6.4).

	ЦСКА	Спартак	Факел	Алания
ЦСКА		2:0	3:3	0:1
Спартак	1:4		2:1	1:2
Факел	0:0	1:1		3:2
Алания	1:2	0:0	0:1	

Рис. 6.4. Таблица результатов футбольного турнира

Основной плюс матрицы смежности – возможность «за один присест» выяснить вес любого ребра графа. Минус – в излишнем расходе памяти: если граф состоит, к примеру, из ста вершин и двухсот ребер, матрица все равно будет занимать память, достаточную для хранения $100 \times 100 = 10\,000$ ребер. В подобных случаях имеет смысл хранить список ребер (троек вида (i, j, w), состоящих из пары номеров соединяемых вершин и веса ребра). При этом поиск веса того или иного ребра будет занимать некоторое время, хотя процесс можно ускорить, используя специальные структуры данных вроде бинарных деревьев.

Вернемся к задаче Прима–Краскала. Сейчас мы напишем программу, которая считывает граф задачи из файла на диске и рисует на экране построенную телефонную сеть. Входной файл организован следующим образом:

[1] Команды и результаты матчей выбраны случайно: они не отражают ни реального положения в нашем футболе, ни предпочтений автора книги.

```
N                                { количество вершин-городов }
x1 y1                            { координаты города на карте }
x2 y2
...
xN yN
<матрица смежности>
```

Помимо матрицы смежности нам понадобятся координаты городов на карте. В задаче эти данные никак не участвуют, а вот при выводе готовой сети на экран без них не обойтись. Для нашего случая (см. рис. 6.2) вполне подойдут такие данные:

```
4
50 50
250 50
250 250
550 150
0          100    150    10000000
100        0      100    200
150        100    0      200
10000000 200    200    0
```

Будем предполагать, что они хранятся в файле graph.txt.

Чтобы не потерять нить рассуждений, давайте проясним общую картину происходящего. Мы начали с того, что алгоритм Прима–Краскала строит оптимальную телефонную сеть (в теории графов этот объект называется *минимальным покрывающим деревом*), если расстояния между городами известны (то есть дан граф задачи). Мы уже разобрались, что граф можно хранить в матрице смежности. Теперь посмотрим, как записываются шаги алгоритма Прима–Краскала на языке программирования.

```
добавить в сеть самое короткое ребро, удовлетворяющее условиям...
```

Чтобы выбрать хоть какое-то ребро, надо организовать просмотр всех ребер графа. Поскольку для любых двух вершин i и j элемент матрицы смежности M[i, j] равен весу ребра, ведущего из i в j, сделать это можно так:

```
for i := 0 to N - 1 do
    for j := 0 to N - 1 do
        { анализируем ребро (i -> j) }
```

Не забывайте, однако, что граф нашей задачи является неориентированным, поэтому вес ребра, ведущего из вершины i в j, равен весу ребра, ведущего из j в i. Поэтому нет смысла просматривать всю матрицу: достаточно организовать перебор лишь одной ее половины. Кроме того, незачем исследовать ребра, ведущие из вершины в нее же саму. Таким образом, цикл просмотра принимает вид:

```
for i := 1 to N - 1 do
    for j := 0 to i - 1 do
        { анализируем ребро (i -> j) }
```

Теперь подумаем, как проверить выполнение условий алгоритма:

```
1) ребро не было добавлено ранее
2) с добавлением ребра в сети не появится цикл
```

Сделать это проще, чем кажется на первый взгляд. Присвоим каждой вершине v уникальный числовой идентификатор id (записывается v.id). Далее, выбирая

всякий раз в процессе работы алгоритма ребро, соединяющее некоторые вершины v1 и v2, будем выполнять действия:

```
id := v2.id
ЦИКЛ по всем вершинам графа
    ЕСЛИ идентификатор текущей вершины равен id
        идентификатор текущей вершины := v1.id
КОНЕЦ ЦИКЛА
```

Тогда оба условия алгоритма превращаются в одно (причем очень простое):

```
идентификаторы соединяемых вершин различны
```

Итак, окончательная версия алгоритма Прима–Краскала выглядит следующим образом:

```
ЦИКЛ N - 1 раз
    найти самое короткое ребро в графе, соединяющее пару вершин (v1 и v2)
    с разными идентификаторами

    id := v2.id
    ЦИКЛ по всем вершинам графа
        ЕСЛИ идентификатор текущей вершины равен id
            идентификатор текущей вершины := v1.id
    КОНЕЦ ЦИКЛА
КОНЕЦ ЦИКЛА
```

Приступим к программированию. На главную форму приложения поместите одну кнопку (назовите ее StartBtn), при нажатии которой будет запускаться алгоритм, и элемент Screen типа TImage, служащий экраном для вывода решения.

Далее нам потребуется тип данных «вершина» и несколько глобальных описаний:

```
type Vertex = record                    { вершина графа }
  id   : Integer;                       { идентификатор вершины }
  x, y : Integer;                       { координаты на карте }
end;

var
  M : array of array of Integer;        { матрица смежности }
  N : Integer;                          { количество вершин }
  V : array of Vertex;                  { массив вершин графа }
```

Мы уже обсуждали способ хранения графа в файле. Опишем теперь процедуру, считывающую входной файл.

```
procedure LoadGraph;                    { загружаем граф из файла }
var F    : TextFile;
    i, j : Integer;
begin
    AssignFile(F, 'graph.txt');         { открываем исходный файл }
    FileMode := 0;                      { в режиме "только чтение" }
    Reset(F);

    ReadLn(F, N);                   { считываем количество вершин }
    SetLength(M, N, N);             { корректируем размер матрицы смежности }
    SetLength(V, N);                { корректируем размер массива вершин }
```

```
    for i := 0 to N - 1 do              { считываем координаты вершин }
        ReadLn(F, V[i].x, V[i].y);

    for i := 0 to N - 1 do              { считываем матрицу смежности }
        for j := 0 to N - 1 do
            Read(F, M[i, j]);

    CloseFile(F);
end;
```

Последний шаг — создание обработчика, вызываемого при нажатии кнопки **StartBtn**:

```
procedure TForm1.StartBtnClick(Sender: TObject);
var i, j, k    : Integer;           { счетчики циклов }
    v1, v2     : Integer;           { индексы вершин }
    id         : Integer;           { переменная для хранения идентификатора }
    MinLength : Integer;            { текущий минимальный вес ребра }
begin
    LoadGraph;                                          { загрузить граф }
    Form1.Screen.Canvas.Brush.Color := clWhite;         { очистить экран }
    Form1.Screen.Canvas.FillRect(Rect(0, 0, 600, 400));

    for i := 0 to N - 1 do                              { инициализация }
        V[i].id := i;                                   { идентификаторов }

    for k := 1 to N - 1 do                              { цикл N - 1 раз }
    begin
        MinLength := 10000000;
        for i := 1 to N - 1 do                          { пробегаем по ребрам графа }
            for j := 0 to i - 1 do
                if (V[i].id <> V[j].id) and (M[i, j] < MinLength) then
                begin
                    v1 := i;     { нахождение следующих соединяемых вершин }
                    v2 := j;
                    MinLength := M[i, j];
                end;

        id := V[v2].id;                 { обновление идентификаторов }
        for i := 0 to N - 1 do
            if V[i].id = id then V[i].id := V[v1].id;

            Form1.Screen.Canvas.MoveTo(V[v1].x, V[v1].y); { вывод очередного }
            Form1.Screen.Canvas.LineTo(V[v2].x, V[v2].y); { найденного ребра }
        end;

        Form1.Screen.Canvas.Brush.Color := clRed;       { рисование вершин }
        for i := 0 to N - 1 do                          { графа (городов)  }
            Form1.Screen.Canvas.FillRect(Rect(V[i].x - 5, V[i].y - 5,
                                              V[i].x + 5, V[i].y + 5));
end;
```

Результат работы программы для тестового задания (все тот же граф, изображенный на рис. 6.2) показан на рис. 6.5.

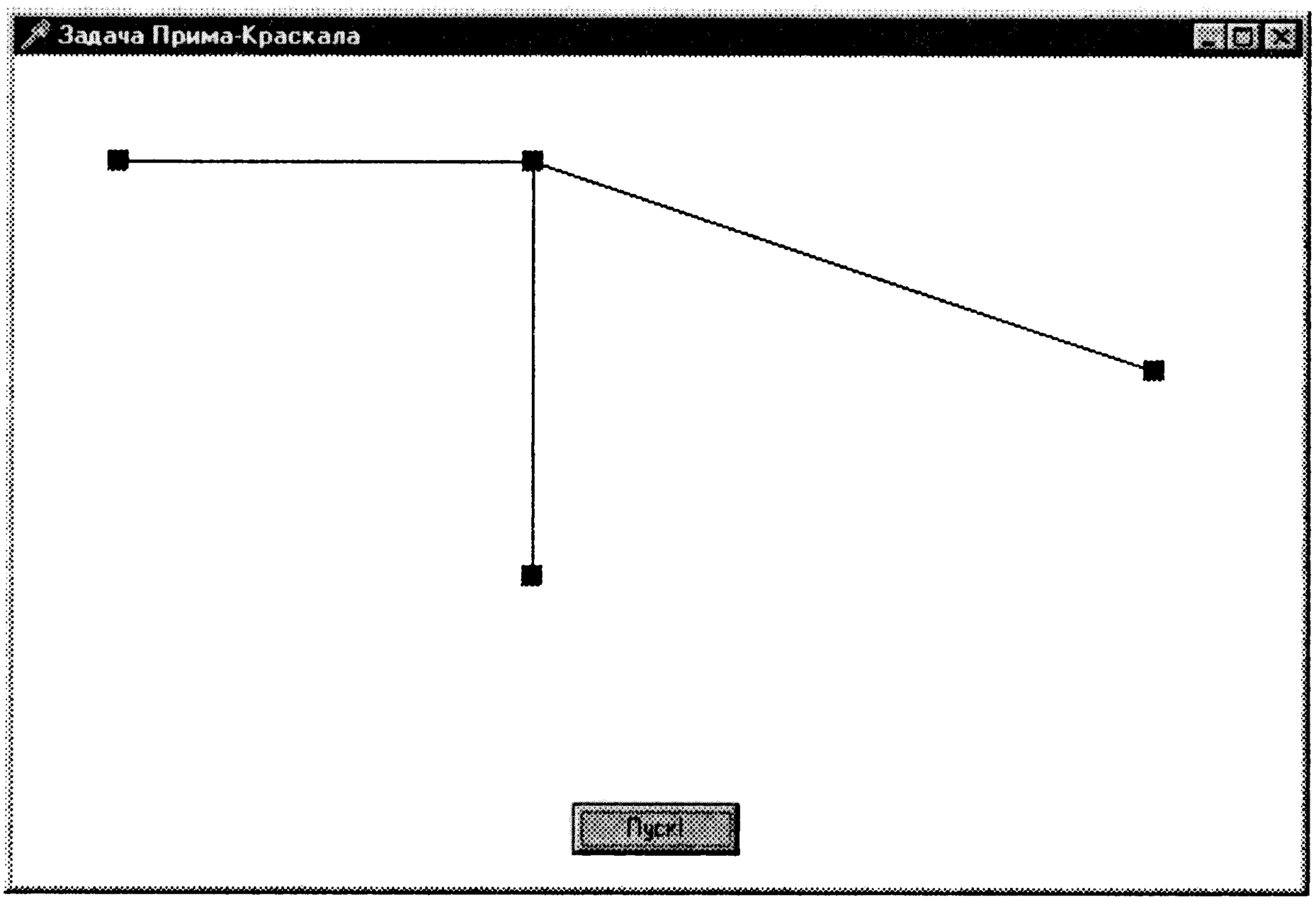


Рис. 6.5. Автоматически построенная оптимальная телефонная сеть

Алгоритм Дейкстры

Рассмотрим другую классическую задачу теории графов — поиск оптимального маршрута между вершинами. Дается карта автодорог некоторой страны. Требуется найти кратчайший путь из одного города в другой. На сей раз вес пути из города A в B может быть не равен весу пути из B в A. Мы уже обсуждали подобную ситуацию (в гору ехать дольше, чем с горы). На языке теории графов задача формулируется следующим образом. Дан *ориентированный* граф. Найти маршрут из вершины, помеченной как стартовая, в вершину, помеченную как финишная; при этом сумма весов ребер, входящих в маршрут, должна быть минимальной.

Давайте опять выберем какой-нибудь простой граф для иллюстрации задачи (предлагаю изображенный на рис. 6.6[1]).

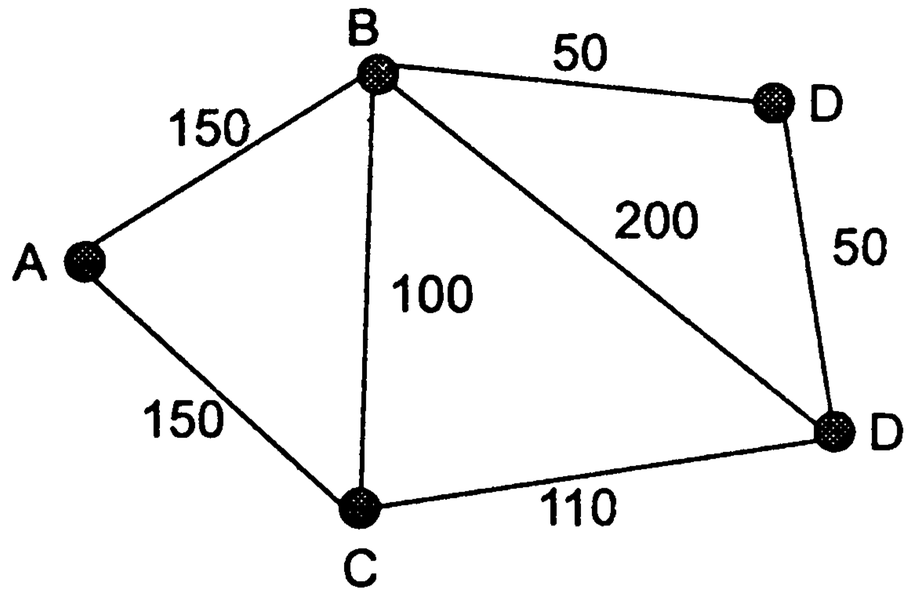


Рис. 6.6. Граф задачи о поиске кратчайшего пути

[1] Правда, он не является ориентированным, но в качестве примера вполне подойдет.

Чтобы добраться из вершины A в вершину E, лучше всего воспользоваться маршрутом A ▸ B ▸ D ▸ E. При этом пройденное расстояние окажется равным 250 км.

Теперь наша задача — научить компьютер находить оптимальный маршрут. Для этого существует известный алгоритм Дейкстры, который мы сейчас и рассмотрим.

Пусть, как и раньше, M — матрица смежности графа (то есть M[i,j] есть вес ребра, ведущего из вершины i в вершину j). Кроме того, сопоставим каждой вершине три величины.

1. Булев индикатор Marked, указывающий, отмечена вершина или нет. В процессе работы алгоритма изначально не отмеченные вершины постепенно становятся отмеченными. Для определения текущего состояния каждой вершины нам и понадобится этот индикатор.
2. Значение DistFromStart, равное длине текущего оптимального пути от стартовой вершины до данной. Алгоритм Дейкстры на каждом шаге пытается уменьшить эту величину.
3. Номер PrevVertex вершины, предшествующей данной на оптимальном пути от стартовой вершины до нее. Эти номера потребуются для хранения результатов работы алгоритма.

Вообще говоря, алгоритм Дейкстры находит оптимальный маршрут из стартовой вершины во все остальные. Просто нас будет интересовать лишь результат, полученный для одной из них — финишной.

Описание алгоритма на псевдокоде выглядит так:

```
изначально ни одна вершина не отмечена
отметить стартовую вершину s
s.PrevVertex := -1 (у стартовой вершины нет предыдущей)
для любой вершины v значение v.DistFromStart := M[s, v]
ЦИКЛ пока есть неотмеченные вершины
    найти неотмеченную вершину vm с минимальным значением поля DistFromStart
    отметить вершину vm
    ЦИКЛ по всем неотмеченным вершинам
        v := текущая просматриваемая вершина
        ЕСЛИ v.DistFromStart > vm.DistFromStart + M[vm, v]
            v.DistFromStart := v.DistFromStart + M[vm, v]
            v.PrevVertex := vm
    КОНЕЦ ЦИКЛА
КОНЕЦ ЦИКЛА
```

После выполнения этих действий длина кратчайшего пути окажется записанной в поле DistFromStart финишной вершины, а список вершин, через которые проходит маршрут (от финиша к старту) можно узнать с помощью фрагмента кода:

```
занести финишную вершину в итоговый список
pv := значение поля PrevVertex финишной вершины
ЦИКЛ
    занести вершину pv в итоговый список
    pv := значение поля PrevVertex вершины pv
ПОКА pv <> -1
```

Теперь займемся реализацией программы на Delphi. В качестве основы возьмем предыдущее приложение, решающее задачу Прима–Краскала. Изменятся только описание типа Vertex и текст процедуры TForm1.StartBtnClick():

```
type Vertex = record                              { вершина графа }
  Marked       : Boolean;                         { индикатор состояния }
  DistFromStart : Integer;                        { расстояние от стартовой вершины }
  PrevVertex    : Integer;                        { предыдущая вершина }
end;

const StartVertex = 0;                            { стартовая вершина }
      FinishVertex = 4;                           { финишная вершина }
...

procedure TForm1.StartBtnClick(Sender: TObject);
var i          : Integer;                          { счетчик цикла }
    NotMarked : Integer;                          { количество неотмеченных вершин }
    vm, pv    : Integer;                          { индексы вершин }
    MinDist   : Integer;                          { текущее минимальное расстояние }
begin
    LoadGraph;                                                { загрузить граф }
    Form1.Screen.Canvas.Brush.Color := clWhite;               { очистить экран }
    Form1.Screen.Canvas.FillRect(Rect(0, 0, 600, 400));

    for i := 0 to N - 1 do
    begin
    V[i].Marked := false;                          { ни одна вершина не отмечена }
    V[i].DistFromStart := M[StartVertex, i];       { начальные расстояния }
end;

V[StartVertex].Marked := true;     { отметить стартовую вершину }
V[StartVertex].PrevVertex := -1;  { у стартовой вершины нет предыдущей }
NotMarked := N - 1;               { начальное количество неотмеченных вершин }

while NotMarked <> 0 do                       { пока есть неотмеченные вершины }
begin
    MinDist := 10000000;
    for i := 0 to N - 1 do
        if not V[i].Marked and (V[i].DistFromStart < MinDist) then
        begin                                          { найти неотмеченную вершину }
            vm := i;                  { с минимальным значением DistFromStart }
            MinDist := V[i].DistFromStart;
        end;

    V[vm].Marked := true;                     { отметить ее }
    NotMarked := NotMarked - 1;

    for i := 0 to N - 1 do
        if not V[i].Marked then     { цикл по всем неотмеченным вершинам }
            if V[i].DistFromStart > V[vm].DistFromStart + M[vm, i] then
            begin
                V[i].DistFromStart := V[vm].DistFromStart + M[vm, i];
                V[i].PrevVertex := vm;
            end;
end;
```

```
Form1.Screen.Canvas.MoveTo(V[FinishVertex].x, V[FinishVertex].y);

pv := V[FinishVertex].PrevVertex;          { вывод полученного маршрута }
repeat
    Form1.Screen.Canvas.LineTo(V[pv].x, V[pv].y);
    pv := V[pv].PrevVertex;
until pv = -1;

Form1.Screen.Canvas.Brush.Color := clRed;    { вывод вершин графа }
for i := 0 to N - 1 do
    Form1.Screen.Canvas.FillRect(Rect(V[i].x - 5, V[i].y - 5,
                                      V[i].x + 5, V[i].y + 5));
end;
```

Результат работы программы для входных данных

```
5
50 150
250 50
190 250
400 70
440 250
0          150    150        10000000 10000000
100        0      100        50       200
150        100    0          10000000 110
10000000 50     10000000 0          50
10000000 200    110        50       0
```

показан на рис. 6.7.

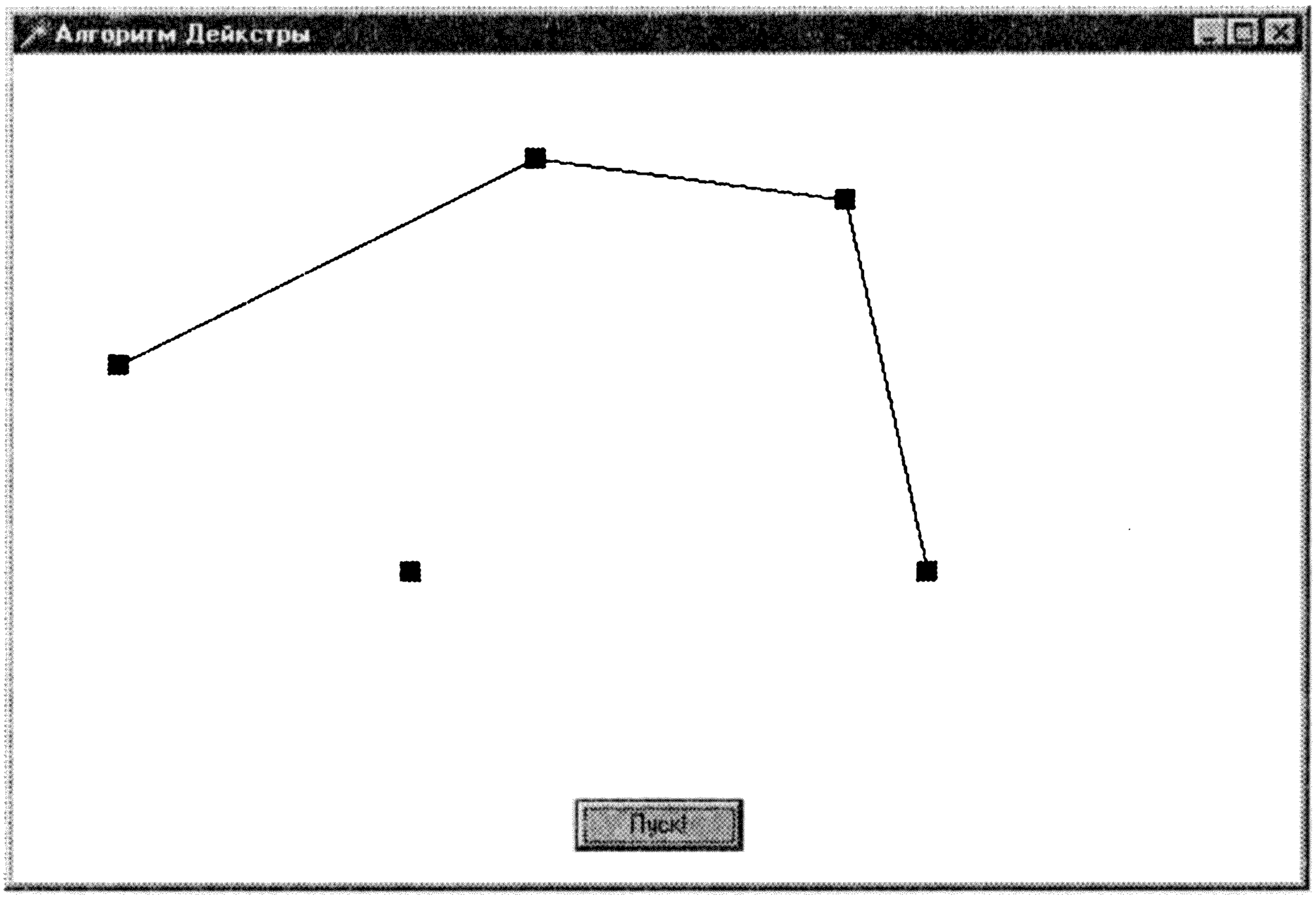


Рис. 6.7. Приложение «Алгоритм Дейкстры» в работе

Методы поиска на графах

В задачах о телефонной сети и поиске оптимального маршрута рассматриваемый граф был виден как на ладони: мы могли легко «бегать» по его вершинам (и не один раз), исследовать ребра, сравнивать веса... В жизни дело нередко обстоит совсем иначе. Порою граф, соответствующий задаче, огромен (или даже потенциально бесконечен), и мы не можем себе позволить исследовать его целиком. Более того, очень часто мы располагаем даже не графом, а лишь инструментами для его построения. Уверен, сейчас вы убедитесь в том, что такие ситуации возникают сплошь и рядом.

Рассмотрим теперь *задачу поиска*. Известна одна-единственная вершина графа (назовем ее *стартовой*). Дана также процедура, при помощи которой мы можем получить список вершин, связанных ребрами с любой известной нам вершиной [1]. Требуется найти вершину, которая обладает некоторыми свойствами (назовем ее *целевой*), и путь, соединяющий стартовую вершину с целевой.

Существует много вариаций задачи поиска. Например, требуется определить все целевые вершины в графе (если их несколько) или найти не просто «некоторый» путь, соединяющий стартовую вершину с целевой, а непременно оптимальный.

Если граф известен целиком, да и размер у него приемлемый (как в задачах Прима–Краскала и Дейкстры), вопрос о поиске целевой вершины вообще не стоит. «Пробежимся» по всем вершинам, найдем целевую, а затем с помощью алгоритма Дейкстры определим требуемый путь (который, к тому же, будет оптимальным). Но наша ситуация, к сожалению, не столь проста.

Игра в 8 и поиск маршрута на карте

Сейчас мы рассмотрим две задачи, очень часто используемые в качестве примеров задач, сводящихся к поиску на графе. Зачем же нарушать традицию?

Надеюсь, все знают, что такое игра в 15. В квадратной коробке лежат 15 пронумерованных кубиков, перемешанных некоторым образом. Требуется, двигая по одному кубику за раз, добиться расположения, показанного на рис. 6.8.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Рис. 6.8. Игра в 15

Игра в 8 – уменьшенный вариант игры в 15, в котором используется коробка размером 3×3 и всего 8 кубиков.

[1] Обратите внимание: имея эту процедуру, нетрудно построить граф задачи целиком, но изначально мы не знаем о графе ничего, кроме стартовой вершины.

И первый, и второй варианты игры являются не чем иным, как задачами поиска на графе (рис. 6.9).

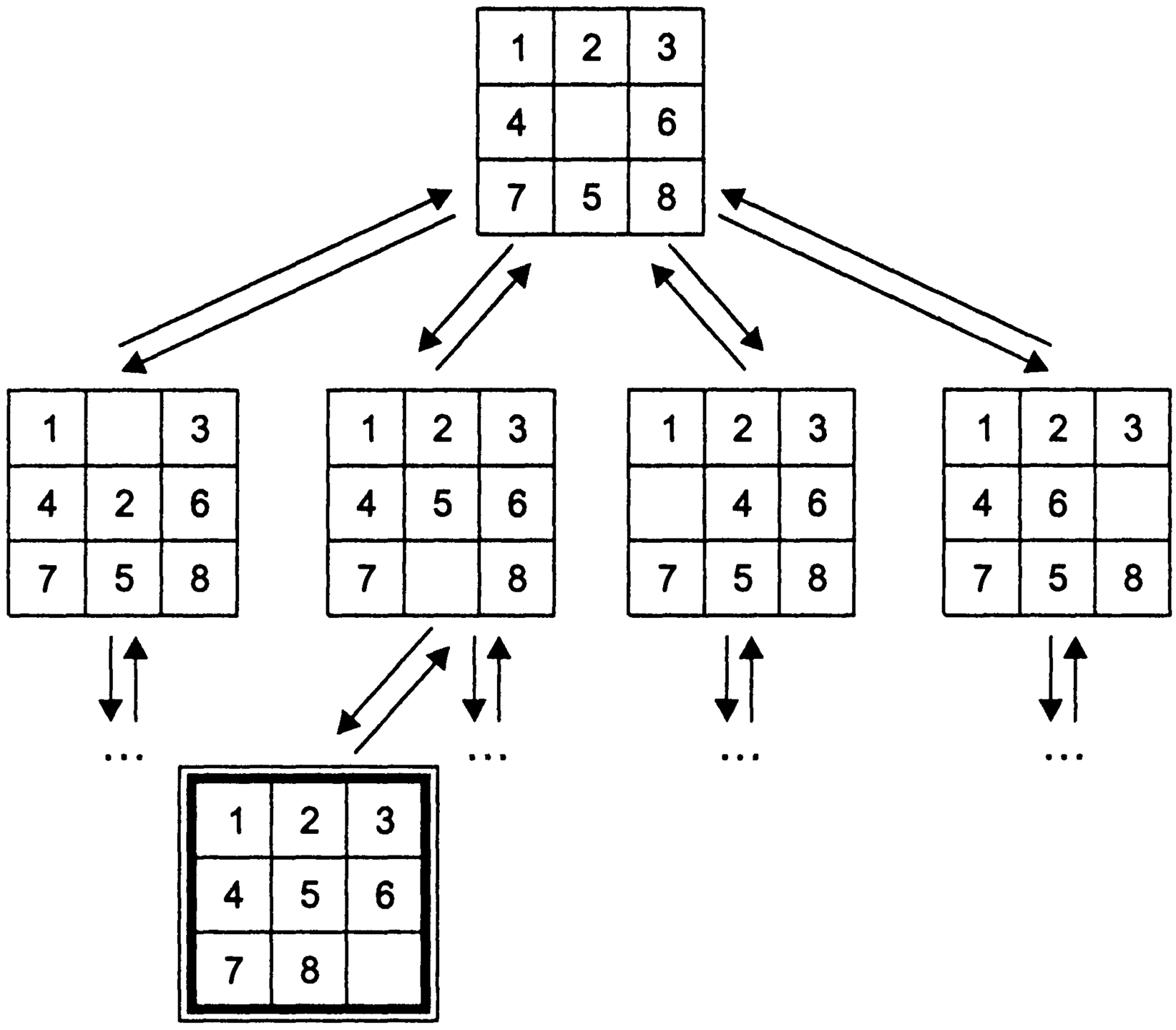


Рис. 6.9. Граф игры в 8

Любое расположение кубиков на доске соответствует вершине графа, а перемещение (то есть ход) – ребру. В начале игры нам дается расположение кубиков, являющееся стартовой вершиной графа. Мы также знаем правила игры, при помощи которых можно получить все соседние (связанные ребрами с текущей) расположения-вершины. Целевой вершиной, разумеется, будет победное расположение кубиков в коробке (на рисунке оно выделено жирной рамкой).

Как видите, игра в 8 прекрасно вписывается в общую схему задачи поиска. Стартовая вершина дана, свойства целевой известны. Граф задачи, вообще говоря, не известен, но теоретически может быть построен при необходимости. Даже такой простой, «игрушечной» головоломке, как игра в 8, соответствует довольно разветвленный граф. Говорить же о построении графа для игры в 15, в 24 (на доске 5×5) или, тем более, для головоломок посложнее не приходится.

В виде (как правило, огромных размеров) графа можно представить любую игру или головоломку на доске – шашки, шахматы, Сокобан (речь об этой игре пойдет в следующей главе), Солитер... О том, как научить компьютер «играть в 8», мы поговорим позже, а пока перейдем к поиску маршрута на карте.

В принципе, речь идет об уже известной вам задаче Дейкстры (поиск маршрута между городами по карте автодорог), но на сей раз предположим, что карта имеет очень большие размеры (допустим, карта мира), и применить к ней алгоритм Дейкстры нет никакой возможности. Менее надуманный случай – карта мегаполиса, в котором требуется добраться от одного дома до другого. Учитывая, что в

крупных городах количество улиц измеряется сотнями, мы будем в своих опасениях не так уж далеки от реальности. Ко всему прочему, время выполнения алгоритма Дейкстры растет пропорционально квадрату количества вершин в графе, что явно не прибавляет оптимизма.

В новой трактовке задачи будем считать, что мы имеем доступ ко всему графу, но исследовать его целиком – слишком большая роскошь. Вопрос об оптимальности найденного маршрута не ставится (лишь бы добраться до пункта назначения), но, по возможности, построенный путь должен быть хотя бы не самым плохим. На мой взгляд, условия очень даже реалистичные. Если вы пишете программу, решающую подобную задачу, то, скорее всего, заказчик не будет особо рад, если найденный «всего за два часа» при помощи алгоритма Дейкстры маршрут всегда будет оптимальным. Моментально построенный, но никуда не годный путь тоже навряд ли удовлетворит клиента, а вот если маршрут будет сгенерирован в разумное время, да и отличаться от оптимального он будет не слишком сильно, вами будут довольны.

Методы поиска на графе разделяют, в первую очередь, на *информированные* и *неинформированные*. Если вы не в состоянии даже предположить, где в графе задачи может скрываться целевая вершина (например, речь идет о поиске слитка золота, случайно размещенного в лабиринте), придется использовать методы неинформированного поиска. Если же идеи есть, можно попытаться воплотить их в параметрах методов второй группы.

Неинформированные методы поиска

Суть любого неинформированного метода проста: раз мы не знаем, где может находиться целевая вершина, будем перебирать все по очереди, пока не наткнемся на то, что искали. Тем не менее даже в такой, казалось бы, безвариантной ситуации у вас есть кое-какой выбор.

Поиск в глубину

При *поиске в глубину* (depth-first search) анализируется первый по списку сосед текущей вершины, затем – его первый сосед и т. д. Если у некоторой вершины нет соседей, аналогичным образом анализируется второй сосед вершины, рассмотренной до нее. Таким образом, процедура поиска сразу же устремляется к вершинам, далеким от стартовой (то есть «в глубину»), и, лишь достигнув вершины, не имеющей соседей, возвращается назад.

Вот текст программы на псевдокоде, реализующей поиск в глубину:

```
function DepthFirstSearch(v : Vertex; depth : Integer) : Boolean;
begin
    Path[depth] := v;          { записываем очередную вершину на пути к цели }
    ЕСЛИ v - целевая вершина
        выход из процедуры с результатом true (цель найдена)

    ЦИКЛ по всем соседним вершинам
        vn := очередная соседняя вершина
        ЕСЛИ DepthFirstSearch(vn, depth + 1) = true
```

```
                выход из процедуры с результатом true
        КОНЕЦ ЦИКЛА
        выход из процедуры с результатом false (цель не найдена)
    end;
```

При вызове функции DepthFirstSearch() ей необходимо передать стартовую вершину и число 0:

```
res := DepthFirstSearch(StartVertex, 0);
```

Если в качестве результата функция вернула значение false, целевая вершина не найдена. В противном случае поиск закончился успешно; путь от стартовой вершины до целевой хранится в массиве Path.

Главный плюс процедуры поиска в глубину состоит в малом расходе памяти. На каждом шаге приходится хранить только вершины, находящиеся на пути от стартовой вершины до текущей, да номера их просматриваемых в данный момент соседей; еще можно вспомнить о некоторых затратах на рекурсию[1].

Есть у приведенного метода и существенный минус. Дело в том, что при поиске в графе, содержащем цикл, процедура может «застрять». Рассмотрим граф, изображенный на рис. 6.10.

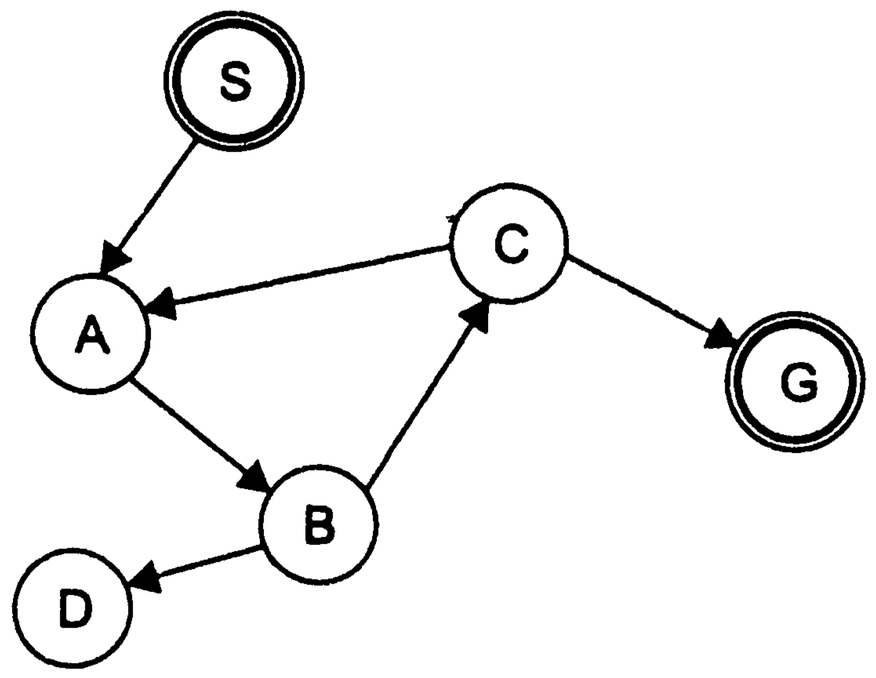


Рис. 6.10. Пример «неудобного» графа для процедуры поиска в глубину

Допустим, процедура поиска начинает работу с вершины S и пытается достичь вершины G. У S есть только один сосед – A, поэтому процедура направляется туда. Единственный сосед A – вершина B. У B два соседа. Допустим, первой в списке оказалась вершина C. У C тоже два соседа. Если на сей раз первой в списке соседей будет A, а не G, мы снова попадаем в A, и процедура поиска зацикливается. Компьютер даже не знает, что текущая вершина уже когда-то была посещена, и продолжает процесс «до победного конца» (то есть до тех пор, пока вам не надоест ждать и вы не закроете окно приложения).

Поиск в ширину

При *поиске в ширину* (breadth-first search) сначала анализируются все соседи текущей вершины. Затем – соседи соседей, соседи соседей соседей и т. д. Пока процедура не рассмотрит все вершины, находящиеся на расстоянии n ребер от стартовой, перехода к более далеким вершинам, длина пути до которых (в ребрах)

[1] Если подобные расходы вам не кажутся маленькими, вспомните об этом, когда мы будем говорить о поиске в ширину.

равна n + 1, не произойдет. Реализация поиска в ширину на псевдокоде выглядит очень просто:

```
procedure BreadthFirstSearch(v : Vertex) : Boolean;
begin
    внести в список L (изначально он пуст) вершину v
    ЦИКЛ
        v := первый элемент списка L
        удалить первый элемент списка L

        ЕСЛИ v - целевая вершина
            выход из процедуры с результатом true (цель найдена)

        добавить в хвост списка L всех соседей вершины v
    ПОКА список L непуст
    выход из процедуры с результатом false (цель не найдена)
end;
```

Эта процедура застрахована от «застревания». Даже если какая-то вершина попадет в список просматриваемых соседей вторично, все равно ее приоритет окажется более низким по сравнению с вершинами, уже находящимися там. К примеру, граф, изображенный на рис. 6.10, для процедуры поиска в ширину не представляет проблемы. На первом шаге будет рассмотрена вершина S, затем – ее единственный сосед A, затем – B. После этого в списке окажутся вершины C и D. После просмотра вершины C список будет иметь вид (D, A, G). Поскольку вершина D не имеет соседей, после ее анализа в списке останутся лишь A и G. Затем в список будет внесен сосед A – вершина B (теперь список выглядит так: (G, B)), и уже на следующем шаге цель окажется достигнутой. Ключевой момент алгоритма – добавление новых элементов именно в хвост, а не в начало и не в середину списка L. Не забывайте об этом.

Проблема поиска в ширину (часто делающая его непригодным на практике) заключается в солидных затратах памяти. На каждом шаге процедура сохраняет в списке все вершины, находящиеся на некоторой «глубине» от стартовой. Если граф, к примеру, имеет вид бинарного дерева (очень частая ситуация на практике), количество хранимых вершин растет экспоненциально. На первом шаге нам понадобится держать в памяти лишь стартовую вершину, затем – ее двух потомков, затем – четыре вершины и т. д. Уже на десятом шаге размер списка составит 1024 элемента, а на двадцатом – 1 048 576 элементов. А ведь двадцать вершин «вглубь» – это, в принципе, не так много для процедуры просмотра (особенно если речь идет всего лишь о бинарном дереве, а не о более ветвистой структуре данных). Кроме того, я привел упрощенную реализацию, которая только ищет целевую вершину, но не строит маршрута от стартовой вершины до нее. Если маршрут вас тоже интересует, придется потратить дополнительную память. Можно поступить так же, как в реализации алгоритма Дейкстры: хранить в каждой просматриваемой вершине ссылку на предыдущую (то есть на ту, откуда мы пришли в текущую вершину). При этом придется сохранять в списке L все вершины, не удаляя их «до победного конца» – иначе путь восстановить не удастся.

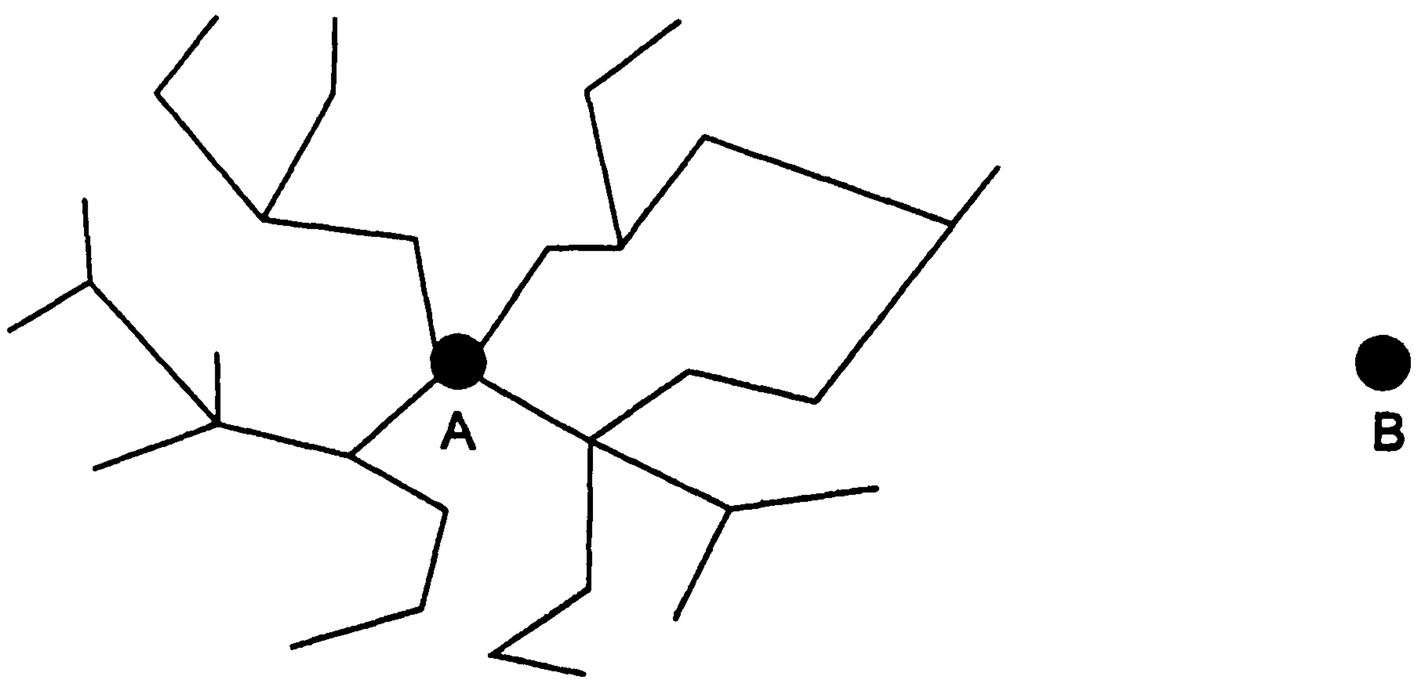
Двунаправленный поиск

Предположим, после некоторого анализа задачи, вы пришли к выводу, что при всем богатстве выбора, к сожалению, придется остановиться на поиске в ширину. При этом, однако, есть одно важное «но»: вы знаете, где находится целевая вершина, и кроме построения пути от стартовой вершины до нее вас ничего не интересует. Такой, в частности, является задача о поиске маршрута на карте города: вы точно знаете, откуда и куда ехать; вопрос – как?

В подобной ситуации вместо обычного поиска в ширину лучше использовать *двунаправленный поиск* (bidirectional search). Его идея проста. Запускаем сразу две процедуры поиска в ширину. Одна начинает работу со стартовой вершины, а другая – с целевой. Как только некоторая третья вершина v оказывается в поле зрения каждой из этих процедур, можно строить маршрут: как добраться от стартовой вершины до v мы знаем благодаря первой процедуре, а как от v до целевой – благодаря второй. Под фразой «запускаем сразу две процедуры» я имею в виду вовсе не многопоточное программирование на многопроцессорной машине (хотя было бы неплохо, конечно), а простую очередность действий: сделали один шаг в первой процедуре, передали управление второй и наоборот.

На первый взгляд особых (если не сказать никаких) преимуществ у двунаправленного поиска не замечается, но на практике дело обстоит иначе. Используя этот метод, можно получить решение быстрее и с меньшими затратами памяти по сравнению с простой процедурой поиска в ширину.

Предположим, вершинами графа являются перекрестки города, а ребрами – улицы. Требуется найти маршрут от перекрестка A до перекрестка B. Допустим, расстояние от A до B по прямой равно 10 км.



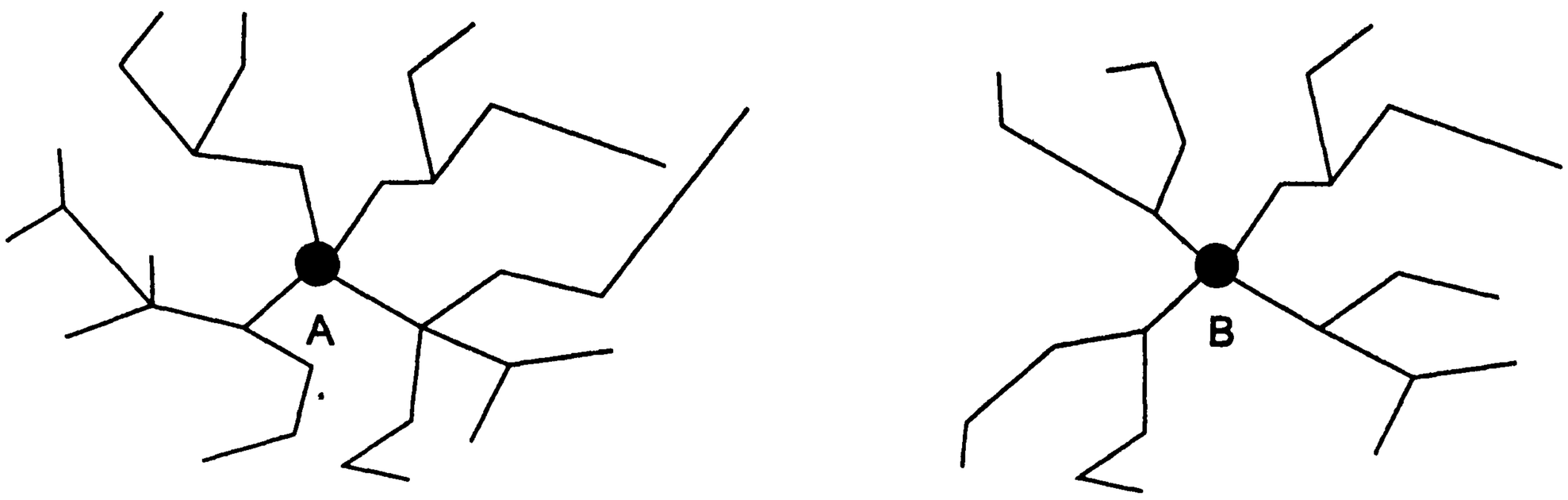


Рис. 6.11. Процедуры поиска: breadth-first vs. bidirectional

Процедура поиска в ширину (рис. 6.11, сверху) начнет просматривать карту вокруг стартовой точки. За время работы она обойдет все перекрестки в среднем в радиусе десяти километров, то есть на площади PI * 100 км^2 (предполагая, что перекрестки распределены по городу более-менее равномерно).

В то же время каждая из процедур двунаправленного поиска (рис. 6.11, снизу) за время работы исследует круг радиусом в пять километров, то есть общая площадь просматриваемого пространства составит PI * 50 км^2 – вдвое меньше, чем в случае поиска в ширину.

Поиск в ширину в действии: решение задачи «игра в 8»

Думаю, практический пример использования поиска в ширину будет достойным завершением обзора неинформированных методов поиска на графе.

Об игре в 8 вы уже знаете, о поиске в ширину – тоже. Остается лишь реализовать его на Delphi. Из трех описанных методов поиска для игры в 8 лучше всего, наверное, подходит двунаправленный поиск, поскольку мы знаем как стартовую, так и целевую вершины. Поиск в глубину не годится из-за цикличности графа задачи. Я же остановился на поиске в ширину лишь для того, чтобы не усложнять программу. Ничего принципиально нового двунаправленный поиск нам не даст, а листинг разрастется вполне ощутимо.

Разработку, как всегда, начнем с пользовательского интерфейса. Поместите на главную форму приложения три элемента (табл. 6.1).

Таблица 6.1. Элементы главной формы

Тип	Имя	Комментарий
TButton	BFSearchBtn	кнопка запуска процедуры поиска
TStringGrid	InputDataGrid	элемент для ввода исходных данных
TMemo	OutputMemo	текстовое поле для вывода результатов

Установите строку Поиск в ширину в качестве свойства Caption кнопки. Теперь придется настроить элемент InputDataGrid в соответствии с табл. 6.2.

Таблица 6.2. Элемент InputDataGrid

Свойство	Значение
ColCount	3
DefaultColWidth	24
DefaultRowHeight	24
FixedCols	0
FixedRows	0
Options4goEditing	True
RowCount	3
ScrollBars	ssNone

В результате описанных действий у вас должна получиться форма наподобие изображенной на рис. 6.12.

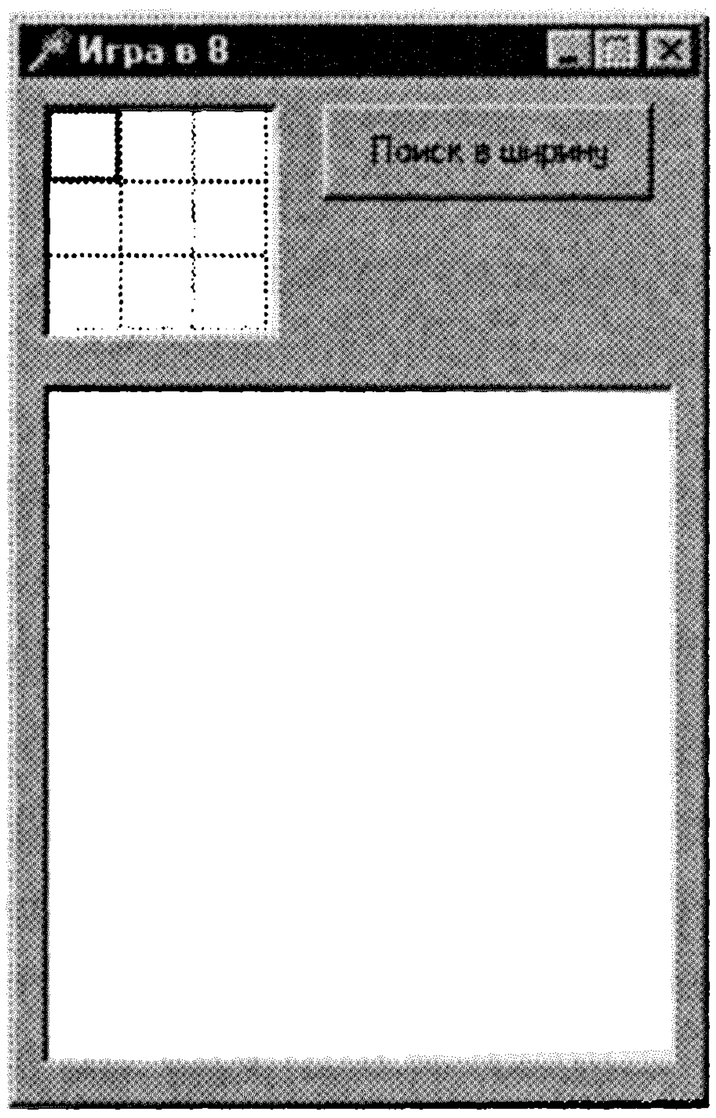


Рис. 6.12. Главная форма приложения «Игра в 8»

Работать приложение будет так: вы вводите начальное состояние игрового поля в элемент InputDataGrid, нажимаете кнопку Поиск в ширину, а компьютер печатает в текстовом поле шаги, ведущие к целевому состоянию. Элементы игрового поля – целые числа; единственное исключение – пустая клетка – будет представляться нулем.

Перейдем к тексту программы. Опишем используемые типы и глобальные переменные (кстати, перечитайте описание поиска в ширину на псевдокоде – разобраться в программе будет легче).

```
type GameState = array[0..2, 0..2] of Integer;    { состояние игрового поля }

type Vertex = record                                { вершина графа }
    State      : GameState;                         { соответствующее состояние }
    PrevVertex : Integer;        { номер предыдущей вершины на пути к текущей }
end;
```

```
var
  StartingState : GameState;                  { начальное состояние }
  Neighbours : array[0..3] of GameState;      { массив "соседних" состояний }
  L : array[0..9999] of Vertex;               { список просматриваемых вершин }
  TailIdx : Integer;                          { указатель на хвост списка }
```

Стремясь предельно упростить программу, я описал список L как массив из 10 000 элементов. Конечно, на практике это решение никуда не годится: для любого более-менее сложного начального расположения кубиков произойдет выход за границы массива. В реальных приложениях лучше использовать динамические массивы или связные списки.

Следующая процедура служит для инициализации программы.

```
procedure Initialize;                              { инициализация }
var i, j : Integer;
begin
    for i := 0 to 2 do                             { загрузка стартового состояния }
        for j := 0 to 2 do
            StartingState[i, j] := StrToInt(Form1.InputDataGrid.Cells[j, i]);
    Form1.OutputMemo.Text := '';                   { очистка поля вывода }
end;
```

В процессе работы метода нам постоянно требуется определять вершины, соседние с текущей. Этим занимается функция **GetNeighbours()**. Она заполняет глобальный массив **Neighbours** и возвращает количество соседних вершин (их не может быть больше четырех).

```
function GetNeighbours(s : GameState) : Integer;      { получить список }
var i, j, k : Integer;                                { соседних вершин }
    zi, zj  : Integer;
    idx     : Integer;
const di : array[0..3] of Integer = (-1, 0, 1, 0);
      dj : array[0..3] of Integer = (0, -1, 0, 1);
begin
    for i := 0 to 2 do                          { находим пустую (нулевую) клетку }
        for j := 0 to 2 do                      { zi, zj - ее координаты }
            if s[i, j] = 0 then begin zi := i; zj := j; end;

    idx := 0;                       { порядковый номер текущей соседней вершины }
    for k := 0 to 3 do
    begin
        i := zi + di[k];        { i, j - координаты клетки, соседней с пустой }
        j := zj + dj[k];        { (на каждой итерации рассматривается новая) }

        { если соседняя клетка находится в пределах поля }
        if (i >= 0) and (j >= 0) and (i <= 2) and (j <= 2) then
        begin                                   { записываем очередной элемент }
            Neighbours[idx] := s;               { в массив Neighbours }
            Neighbours[idx][i, j] := 0;         { пустая клетка меняется }
            Neighbours[idx][zi, zj] := s[i, j];   { местами с соседней }
            idx := idx + 1;
        end;
    end;
    GetNeighbours := idx;       { возвращаем количество найденных вершин }
end;
```

В любой игровой ситуации можно сделать не меньше двух и не больше четырех различных ходов. Любой ход есть не что иное, как сдвиг пустой ячейки в одну из четырех сторон — влево, вправо, вверх или вниз. Понятно, что пустая ячейка, расположенная в центре поля, дает больше свободы действий, чем находящаяся в углу. Функция **GetNeighbours()** просматривает по очереди все четыре соседние с пустой клетки и, если ход оказывается допустимым, меняет местами пустую клетку с текущей соседней, тем самым генерируя новую вершину графа.

Чтобы найти целевое состояние, необходимо уметь его определять. Этим занимается функция **IsGoal()**:

```
function IsGoal(s : GameState) : Boolean;      { проверка: является ли }
var i, j : Integer;                            { состояние s целевым? }
const goal : GameState = ((1,2,3), (4,5,6), (7,8,0));  { целевое состояние }
begin
     for i := 0 to 2 do
         for j := 0 to 2 do
             if s[i, j] <> goal[i, j] then  { если найдено несоответствие }
             begin
                 IsGoal := false;           { возвращаем false }
                 Exit;
             end;
     IsGoal := true;                        { s - целевое состояние  }
end;
```

Последняя служебная функция, которая нам понадобится, представляет игровое состояние в виде строки, пригодной для вывода в текстовое поле **OutputMemo**:

```
function StateToString(s : GameState) : String;  { преобразование состояния }
var i, j : Integer;                              { в строковую форму }
    r    : String;
begin
    r := '';
    for i := 0 to 2 do
    begin
        for j := 0 to 2 do
            r := r + IntToStr(s[i, j]) + ' ';       { вывод очередного ряда }
        r := r + Chr(13) + Chr(10);      { возврат каретки / перевод строки }
    end;

    StateToString := r;
end;
```

Основные действия происходят при нажатии кнопки **BFSearchBtn**, то есть в процедуре **TForm1.BFSearchBtnClick()**, являющейся довольно прямолинейной реализацией алгоритма поиска в ширину:

```
procedure TForm1.BFSearchBtnClick(Sender: TObject);     { поиск в ширину }
var HeadIdx : Integer;                  { указатель на начало списка L }
    N : Integer;                        { количество соседних состояний }
    i : Integer;                        { счетчик цикла }
    v : Vertex;                         { текущая исследуемая вершина }
    c : Integer;                        { количество уже исследованных вершин }
begin
Initialize;

L[0].State := StartingState;      { вносим в список L стартовую вершину }
L[0].PrevVertex := -1;            { предыдущей вершины у стартовой нет }
HeadIdx := 0;
TailIdx := 1;
c := 0;

repeat
    v := L[HeadIdx];              { v := первый элемент списка }
    c := c + 1;

    if IsGoal(v.State) then       { если текущая вершина является целевой }
```

```
begin
    { выводим ее в текстовое поле }
    Form1.OutputMemo.Text := StateToString(v.State) + Chr(13) + Chr(10);

    { шаг за шагом определяем путь от целевой вершины до стартовой, }
    { на каждой итерации выводя соответствующее состояние }
    repeat
        v := L[v.PrevVertex];
        Form1.OutputMemo.Text := StateToString(v.State) + Chr(13)
                                 + Chr(10) + Form1.OutputMemo.Text;
    until v.PrevVertex = -1;   { добрались до стартовой вершины }

    Form1.OutputMemo.Text := Form1.OutputMemo.Text + Chr(13)
                + Chr(10) + 'Исследовано состояний: ' + IntToStr(c);
    Exit;
end;

N := GetNeighbours(v.State);     { определяем соседние вершины }
for i := 0 to N - 1 do           { и вносим их в список L }
begin
    L[TailIdx].State := Neighbours[i];
    L[TailIdx].PrevVertex := HeadIdx;
    TailIdx := TailIdx + 1;
end;

HeadIdx := HeadIdx + 1;          { сдвиг головы списка }
until HeadIdx = TailIdx;         { цикл пока список непуст }

    Form1.OutputMemo.Text := 'Решение не найдено';
end;
```

Как уже отмечалось, рассмотренные вершины физически не удаляются из списка L, чтобы можно было построить интересующий нас маршрут.

Для проверки работы программы попробуйте дать ей какую-нибудь простую задачку (то есть начальное расположение кубиков). Например, такую:

```
4 1 3
2 0 6
7 5 8
```

В ответ вы должны получить последовательность из шести шагов:

```
4 1 3
2 0 6
7 5 8

4 1 3
0 2 6
7 5 8

0 1 3
4 2 6
7 5 8

1 0 3
4 2 6
```

```
7 5 8

1 2 3
4 0 6
7 5 8

1 2 3
4 5 6
7 0 8

1 2 3
4 5 6
7 8 0

Исследовано состояний: 632
```

Внешний вид приложения показан на рис. 6.13.

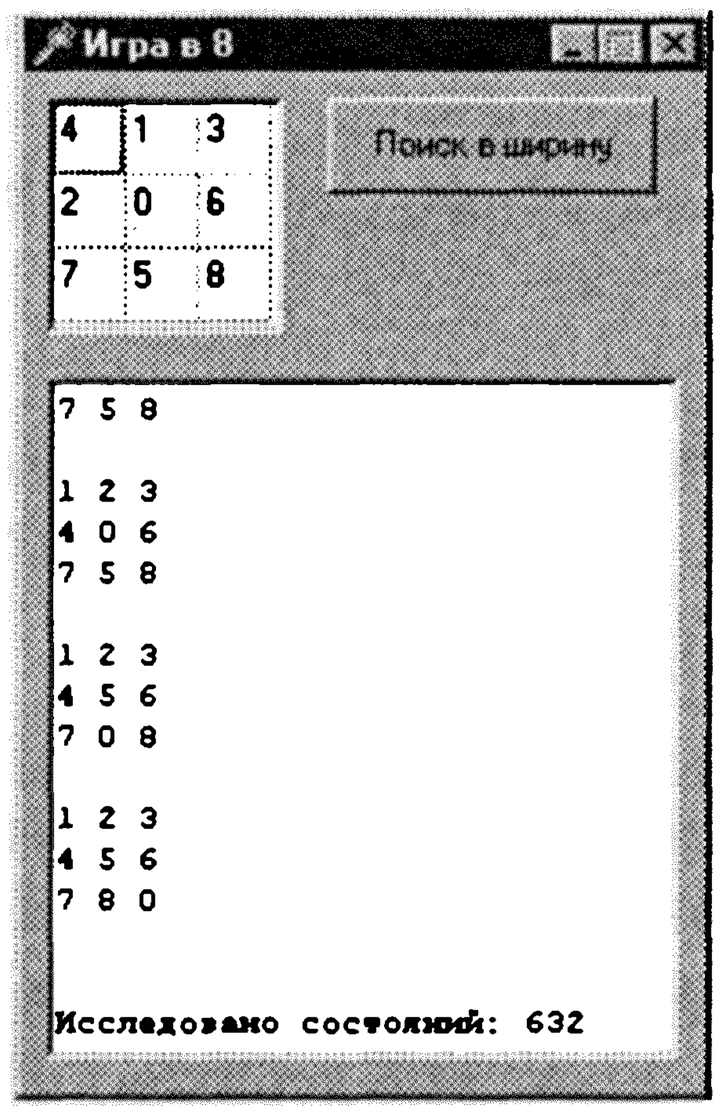


Рис. 6.13. Приложение «Игра в 8» в работе

Информированные методы поиска

Помните, с чего начинался разговор о неинформированных методах поиска? «Если вы не в состоянии даже предположить, где в графе задачи может скрываться целевая вершина...» Так вот, теперь мы считаем, что кое-какие предположения о нахождении целевой вершины сделать можно, и наша задача – использовать их.

В настоящее время разработано несколько известных информированных методов поиска. Мы рассмотрим, возможно, самый популярный из них – процедуру A* (читается «A со звездой») и ее модификацию IDA* (Iterative Deepening A*).

Процедура A*

Общая идея этого метода сходна с уже рассмотренными нами алгоритмами поиска на графе. Сначала для стартовой вершины формируется список вершин, со-

седних с ней. Затем очередной элемент некоторым образом выбирается в списке и становится текущим. В список добавляются соседи текущей вершины, и цикл повторяется до тех пор, пока цель не будет найдена или же все вершины графа не окажутся рассмотренными. В алгоритме поиска в ширину в качестве следующей анализируемой вершины мы просто брали очередной элемент списка L, а в случае поиска в глубину — первую нерассмотренную вершину, соседнюю с текущей. Отличие процедуры A* состоит в том, что мы теперь можем на каждом шаге контролировать процесс выбора следующей рассматриваемой вершины, делая его более разумным[1]. Замечательно, что сделать это совсем не сложно.

Процедура A* применяется к *взвешенным* графам (то есть к графам, ребра которых помечены некоторыми числами — весами). В задаче о поиске маршрута на карте веса ребер, очевидно, равны расстояниям между объектами, которым соответствуют вершины. В графе, соответствующем игре в 8, можно приписать любому ребру вес, равный единице. Тогда суммарная стоимость маршрута от стартовой вершины до целевой окажется равной количеству сделанных ходов. Если вас вообще не интересуют веса ребер (а применить процедуру A* хочется), просто считайте, что вес каждого ребра равен единице.

Допустим, в процессе работы алгоритм поиска анализирует некоторую вершину v. Сопоставим ей два числа — *цену* (cost) и *эвристику* (heuristics). Цена — это суммарный вес ребер на пути от стартовой вершины до текущей (то есть до v). Поскольку процедура добралась до вершины v, анализируя граф со стартовой вершины, цена известна нам абсолютно точно; вернее, мы можем ее определить, если пожелаем. Обратите внимание: под ценой понимается стоимость именно того маршрута, которым процедура следовала от стартовой вершины до текущей; никто не требует вычислять цену оптимального пути (которую мы, скорее всего, не знаем). Эвристика — это расчетная стоимость минимального пути от v до целевой вершины. Точное значение этой величины мы, конечно, знать не можем, но сделать более-менее разумное предположение обычно в состоянии.

Функция, сопоставляющая любой вершине графа некоторую эвристику, называется *эвристической функцией* (heuristic function). Эвристическая функция, прогнозы которой всегда точны или оптимистичны (то есть предсказываемое значение стоимости оптимального пути никогда не превосходит его реальной стоимости), называется *допустимой* (admissible). Использование только допустимых эвристических функций гарантирует: целевая вершина будет найдена, а используемый маршрут от стартовой вершины до нее — оптимальным. Недопустимость эвристической функции, несмотря на название, еще не означает полную ее непригодность, но гарантий в этом случае у вас никаких не будет; старайтесь, чтобы ваши функции всегда были допустимыми.

Написать простейшую допустимую эвристическую функцию проще простого: достаточно сопоставлять любой вершине графа нуль в качестве значения эвристики. Действительно, стоимость какого бы то ни было маршрута не может оказаться меньше нуля. Но пользы от такой функции не будет: в действительности

---

[1] Использовать голову все равно придется. Можно ухитриться запрограммировать процедуру поиска так, чтобы она всегда выбирала наихудший вариант из всех возможных.

вы получите обыкновенный поиск в ширину. Вообще из двух эвристических функций более эффективна та, которая дает наиболее близкие к реальности прогнозы. Таким образом, выбирая эвристическую функцию для своей программы, вы зажаты с двух сторон: с одной стороны, получаемые значения должны быть оптимистичными, чтобы сохранить допустимость функции, а с другой — настолько пессимистичными, насколько это только возможно для повышения ее эффективности. В идеале функция должна точно угадывать стоимость оптимального маршрута, но это, конечно, мечта.

Выбор хорошей эвристической функции — настоящее искусство, сравнимое с выбором используемого в алгоритмах сжатия данных предиктора. Я абсолютно уверен в том, что оно может принести вам немало удовольствия. В этом искусстве, как и в любом другом, нет правил. Используйте свой талант и опыт. Думайте! Пробуйте! Единственное, что я могу сделать, так это дать несколько простых советов, которым стоит следовать:

- следите, чтобы ваша эвристическая функция ВСЕГДА была допустимой;
- из двух ДОПУСТИМЫХ эвристических функций выбирайте ту, прогнозы которой пессимистичнее;
- следите за балансом между эффективностью эвристической функции и скоростью ее работы: порою имеет смысл выбрать не самую умную, зато быструю функцию.

Вернемся теперь к процедуре A*. Ее суть заключается в том, чтобы на каждом шаге выбирать из списка L вершину, сумма значений эвристики и цены которой минимальна. Вот и все.

Процедура IDA*

Если бы для любой задачи можно было создать хорошую эвристическую функцию, процедура A* удовлетворила бы любого разработчика. К сожалению, не так уж редки случаи, когда не очень хорошая (хотя и допустимая) эвристическая функция уводит процедуру поиска куда-то в дебри графа, хотя решение находится где-то под самым носом. К примеру, прервав после десяти минут работы программу анализа шахматного этюда, вы с удивлением обнаруживаете, что она просматривает какую-то позицию на глубине в семь ходов от начальной: вы-то ожидали что-то вроде мата в три хода.

В подобных случаях нередко применяют процедуру IDA*. Ее суть состоит в последовательном применении алгоритма A* к графу задачи с постепенным снятием ограничений на глубину просмотра. Сначала процедуре A* разрешается искать лишь на единичной глубине (то есть среди непосредственных соседей стартовой вершины), затем, в случае неудачи, – на глубине, не превосходящей 2 (среди соседей стартовой вершины и их соседей) и т. д. Таким образом, каждый раз в случае неудачи процедура A* запускается заново, но при этом ей позволяется заглянуть глубже.

На первый взгляд может показаться, что подобная методика на редкость неэффективна. Запуская алгоритм A* на n-м шаге процедуры IDA*, мы повторяем уже осуществленный ранее просмотр графа на глубину $n - 1$. Если некоторый

граф содержит 10 000 вершин, находящихся на глубине в $n - 1$ ребер от стартовой, то при n-м запуске процедуры A* они все будут проанализированы заново, хотя нам доподлинно известно, что целевой вершины среди них нет.

На практике эти расходы оказываются не так уж велики. Дело в том, что графы реальных задач, как правило, сильно ветвятся. Поэтому количество вершин, расположенных на n-м уровне, может легко превзойти совокупный объем вершин, находящихся на всех уровнях выше; следовательно, время, затраченное на просмотр предыдущих уровней, окажется не таким большим по сравнению со временем, которое придется уделить n-му уровню. По некоторым оценкам, процедура IDA*, просматривающая некоторый «типичный» граф на глубину в n ребер, работает лишь на 10–20 процентов медленнее соответствующей процедуры A*.

Решение задачи «Игра в 8» при помощи процедуры A*

Дополним предыдущую программу реализацией алгоритма A*. В первую очередь, необходимо определить, каким образом будут вычисляться цена и эвристика каждой вершины графа задачи. С ценой дело обстоит просто: количество ходов, требуемое для получения текущей позиции из стартовой, и есть цена. В качестве эвристики предлагаю пока что выбрать количество кубиков, находящихся не на своих местах. Конечно, такой «прогноз» имеет мало общего с истинной ценой маршрута (то есть количеством ходов, требуемым для получения целевой позиции из текущей), но, тем не менее, наша эвристическая функция будет допустимой. Если, к примеру, четыре кубика расположены неверно, то меньше чем за четыре хода получить желаемое расположение, очевидно, не удастся.

Итак, приступим. Добавьте на главную форму еще одну кнопку. В качестве ее имени укажите AStarSearchBtn, а в качестве надписи — A*.

Далее, перепишите определение типа Vertex:

```
type Vertex = record                                  { вершина графа }
    State      : GameState;                           { соответствующее состояние }
    PrevVertex : Integer;       { номер предыдущей вершины на пути к текущей }

    Cost       : Integer;         { "цена" вершины }
    Heuristics : Integer;         { эвристика }
    Marked     : Boolean;         { индикатор "просмотрена / не просмотрена" }
end;
```

Теперь запрограммируем эвристическую функцию:

```
{ эвристическая функция }
function CalculateHeuristics(s : GameState) : Integer;
      { строка и столбец соответствующего кубика (в целевой позиции) }
const RowOf : array[1..8] of Integer = (0, 0, 0, 1, 1, 1, 2, 2);
      ColOf : array[1..8] of Integer = (0, 1, 2, 0, 1, 2, 0, 1);
var i, j, r : Integer;
begin
    r := 0;
    for i := 0 to 2 do
        for j := 0 to 2 do          { подсчитываем количество кубиков, }
            if s[i, j] <> 0 then    { находящихся не на своих местах }
                if (i <> RowOf[s[i, j]]) and (j <> ColOf[s[i, j]]) then
```

```
                r := r + 1;

    CalculateHeuristics := r;
end;
```

Массивы RowOf и ColOf содержат координаты кубиков в целевой позиции. Например, кубик 4 должен находиться в строке RowOf[4] = 1, столбце ColOf[4] = 0. Нулевой кубик — и не кубик вовсе, поэтому при подсчете его позиция не учитывается.

Следующая функция возвращает индекс следующего извлекаемого элемента в списке L:

```
function GetIndexOfNextElement : Integer;   { определить следующую вершину, }
var Min, idx, i : Integer;                  { извлекаемую из списка L }
begin
    Min := 1000000;

    for i := 0 to TailIdx - 1 do
        if (L[i].Marked = false) and (L[i].Cost + L[i].Heuristics < Min) then
        begin              { определяем неотмеченный элемент списка с }
            idx := i;      { минимальным значением величины "цена + эвристика" }
            Min := L[i].Cost + L[i].Heuristics;
        end;
    GetIndexOfNextElement := idx;
end;
```

Работает она весьма бесхитростно: просматривает список L и находит элемент, удовлетворяющий условиям:

1) он не был извлечен ранее (Marked = false);

2) значение Cost + Heuristics минимально.

Последовательный просмотр всего списка на каждом шаге алгоритма — решение далеко не лучшее; хорошо оно лишь своей простотой. К сожалению, более скоростные способы организовать работу алгоритма требуют использования и более сложных структур данных, а не простых массивов.

Главная процедура TForm1.AStarSearchBtnClick() очень похожа на поиск в ширину:

```
procedure TForm1.AStarSearchBtnClick(Sender: TObject);    { процедура A* }
var N   : Integer;                          { количество соседних состояний }
    i   : Integer;                          { счетчик цикла }
    v   : Vertex;                           { текущая исследуемая вершина }
    c   : Integer;                          { количество уже исследованных вершин }
    idx : Integer;                          { индекс анализируемого элемента }
begin
    Initialize;

    L[0].State := StartingState;       { вносим в список L стартовую вершину }
    L[0].PrevVertex := -1;             { предыдущей вершины у стартовой нет }
    L[0].Cost := 0;                    { цена стартовой позиции равна нулю }
    { вычисляем эвристику }
    L[0].Heuristics := CalculateHeuristics(StartingState);
    L[0].Marked := false;

    TailIdx := 1;
```

```
c := 0;

repeat
    { определяем следующий анализируемый элемент списка }
    idx := GetIndexOfNextElement;
    v := L[idx];                          { извлекаем элемент из списка }
    c := c + 1;
    L[idx].Marked := true;                { помечаем его как рассмотренный }

    if IsGoal(v.State) then          { если он - целевой }
    begin                            { печатаем результат }
        Form1.OutputMemo.Text := StateToString(v.State) + Chr(13) + Chr(10);
        repeat
            v := L[v PrevVertex];
            Form1.OutputMemo.Text := StateToString(v.State) + Chr(13) +
                                     Chr(10) + Form1.OutputMemo.Text;
        until v.PrevVertex = -1;

        Form1.OutputMemo.Text := Form1.OutputMemo.Text + Chr(13) +
                    Chr(10) + 'Исследовано состояний: ' + IntToStr(c);
        Exit;
    end;

    N := GetNeighbours(v.State);          { определяем соседние вершины }
    for i := 0 to N - 1 do                { и вносим их в список L }
    begin
        L[TailIdx].State := Neighbours[i];
        { цена := цена предыдущей вершины + вес ребра (1) }
        L[TailIdx].Cost := v.Cost + 1;
        L[TailIdx].Heuristics := CalculateHeuristics(Neighbours[i]);
        L[TailIdx].PrevVertex := idx;

        TailIdx := TailIdx + 1;
    end;
  until TailIdx = 0;                              { цикл, пока список непуст }

  Form1.OutputMemo.Text := 'Решение не найдено';
end;
```

Тестируя программу на тех же входных данных, что и поиск в ширину, можно убедиться: решение получается тем же самым, но количество исследованных вершин уменьшается до 289! А было, если помните, 632. Таким образом, использование эвристической функции сократило перебор более чем вдвое. Но, как говорится в известной рекламе, и это еще не все! Давайте попробуем написать более умную эвристическую функцию, которая сопоставляет каждой вершине сумму расстояний от каждого кубика до его целевой позиции. Мы будем использовать так называемое «манхэттенское расстояние»:

```
расстояние(кубик, цель) := Abs(Xкубика - Xцели) + Abs(Yкубика - Yцели)
```

Новая версия эвристической функции выглядит так:

```
function CalculateHeuristics(s : GameState) : Integer;
      { строка и столбец соответствующего кубика (в целевой позиции) }
const RowOf : array[1..8] of Integer = (0, 0, 0, 1, 1, 1, 2, 2);
```

```
      ColOf : array[1..8] of Integer = (0, 1, 2, 0, 1, 2, 0, 1);
var i, j, r : Integer;
begin
    r := 0;
    for i := 0 to 2 do
        for j := 0 to 2 do
            if s[i, j] <> 0 then
                r := r + Abs(RowOf[s[i, j]] - i) + Abs(ColOf[s[i, j]] - j);

    CalculateHeuristics := r;
end;
```

Теперь для того, чтобы добраться до целевой позиции, алгоритму потребуется просмотреть всего девять (!) вершин графа (рис. 6.14).

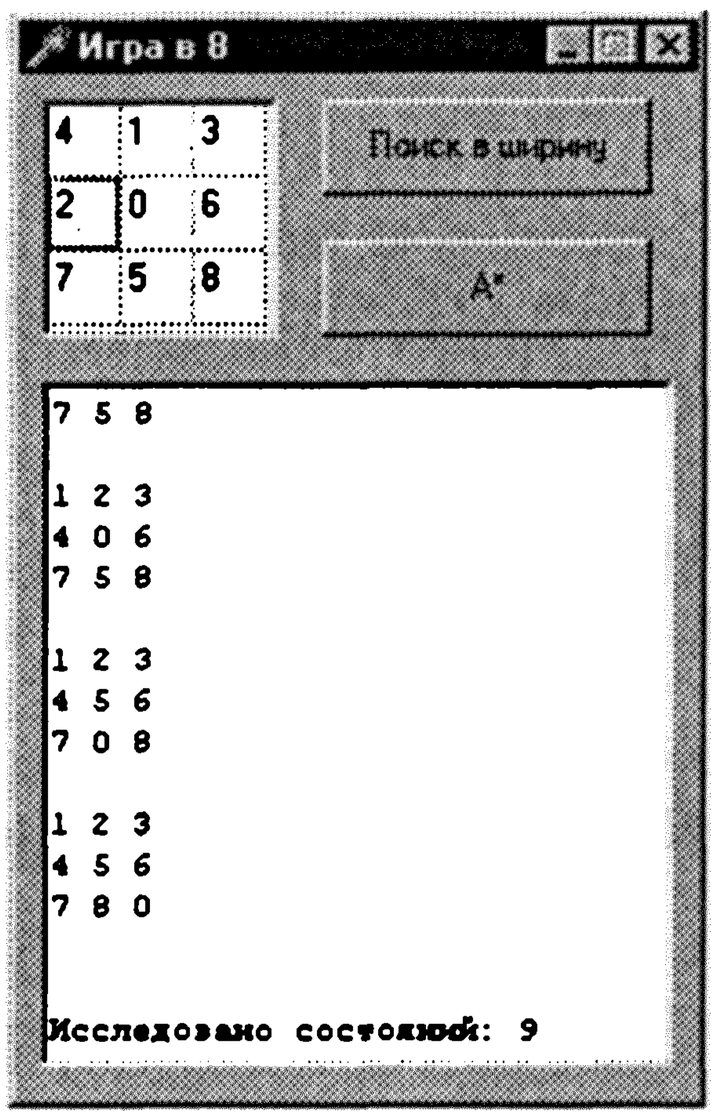


Рис. 6.14. Использование процедуры A* для решения головоломки «Игра в 8»

Думаю, какие-либо комментарии излишни. На этой оптимистической ноте позвольте закончить главу.

Проекты для самосовершенствования

1. Сравните поиск в глубину с рекурсивным обходом лабиринта. Лабиринт в общем случае является циклическим графом. Какой прием используется в рекурсивном обходе для предотвращения зацикливания?
2. Сравните поиск в ширину с алгоритмом волновой трассировки.
3. Подсчитайте (хотя бы примерно), какой объем памяти потребуется для анализа графа игры в 8 на глубину в десять ходов при использовании поиска в ширину.
4. Напишите эвристическую функцию для задачи о поиске маршрута на карте.

Глава 7

Простые компьютерные игры

Если вы хотите доставить другим радость, то можете не сомневаться, что видеоигры — это то, что надо! Видеоигры — это способ выражения самых фантастических идей и образов.

Андрэ Ла Мот

Вот уже второй эпиграф я беру из «Секретов программирования игр» Андрэ Ла Мота и Стива Ратклиффа. Мне кажется, лучше о компьютерных играх и сказать нельзя. С одной стороны, создание игр — это самое что ни на есть «занимательное» программирование; с другой стороны, игры — прекрасный пример систем, в которых органично сочетаются графика, звук и действие. С точки зрения программирования, игра — это довольно серьезный проект, не уступающий (а часто и значительно превосходящий) по сложности реализации многие приложения «для дела».

По моим наблюдениям, многие (если не все) люди, изучающие программирование, пытались написать хотя бы одну компьютерную игру. В этой главе мне хотелось на конкретных примерах показать, как пишутся игры. Наверное, это слишком громко сказано: без мощных графических библиотек хорошую игру не напишешь (разве что логическую, где анимация минимальна), да и объем книги не позволяет особо размахнуться. С другой стороны, даже те простые игры, которые мы здесь рассмотрим, популярны и сейчас (а уж лет двадцать назад [1] были просто хитами!).

Понимаю, приводя готовые листинги игр, я открываю широкое поле для критики: мол, здесь и здесь можно было написать гораздо лучше. Безусловно, любую программу можно улучшить; тем не менее наши игры будут прекрасно работать и, я надеюсь, послужат поучительными примерами.

Всего мы обсудим четыре игры, переходя от простого к сложному.

Сапер (Minesweeper)

Кто не знает этой замечательной маленькой игрушки, входящей в состав ОС Windows (рис. 7.1)?

---

[1] Разумеется, в то время подобную игру было написать куда сложнее.

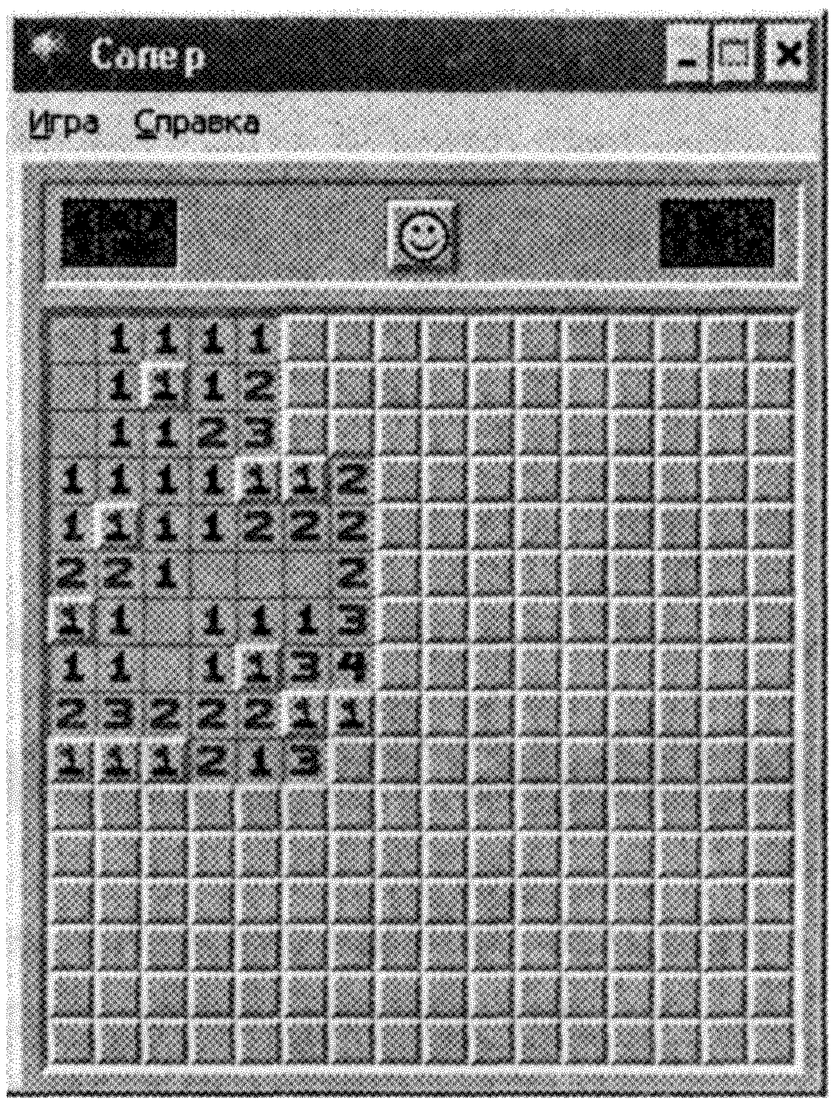


Рис. 7.1. Сапер (реализация из ОС Windows)

Напомню вкратце ее правила. На экране появляется прямоугольное клетчатое поле. Изначально каждая клетка закрыта, поэтому вы не знаете, что находится в той или иной части поля. Некоторые клетки безопасны, в то время как в других спрятаны мины. При щелчке левой кнопкой мыши на любой клетке «прикрытие» последней исчезает, и вы можете видеть ее содержимое. Если вам попалась мина — что ж, не повезло: конец игры. В противном случае отображается общее количество мин в клетках, соседних с текущей. Максимальное число «опасных» соседей — восемь (у каждой клетки всего восемь соседей; диагонально расположенные клетки соседними считаются), минимальное — естественно, нуль (в стандартной реализации Сапера вместо нуля отображается пустое пространство). Если открыта пустая (то есть нулевая) клетка, можно также открыть все соседние с ней — в них заведомо не будет мин. Поэтому хорошая реализация Сапера делает это автоматически (а если какая-нибудь из этих только что обработанных клеток тоже окажется пустой, для нее проделывается аналогичная операция). Располагая информацией о соседних минах, можно делать определенные выводы об опасности той или иной клетки. Цель игры — открыть все не заминированные участки игрового поля.

Для удобства во многих версиях Сапера можно устанавливать «флажки», помечающие отдельные клетки. Если вы уже точно вычислили, что в данной клетке находится мина, можно пометить ее флажком и двигаться дальше, уже не возвращаясь к ней.

В качестве хорошего начала давайте реализуем упрощенный вариант Сапера — без «флажков» и автоматического открывания клеток.

В первую очередь обсудим архитектуру программы. Обратите внимание, что клетки игрового поля очень похожи на обычные кнопки с изображениями (объекты типа TSpeedButton) — хотя бы потому, что их можно нажимать. И действительно, зачем как-то отслеживать положение мыши на экране и программировать реакцию на щелчки, если в кнопках эти функции уже реализованы? Представляя каждую клетку обычной кнопкой (таким образом, игровое поле бу-

дет сплошь заполнено кнопками), мы избавляемся от большого количества ненужной работы.

Предположим, пользователь нажимает на какую-либо кнопку на экране (что означает «открыть клетку»). В этом случае нам надо сначала проверить, нет ли на этом месте мины. Если есть, тогда не повезло: конец игры; если же нет, подсчитываем количество соседних мин и выводим его на месте кнопки. При последующих нажатиях на кнопку уже ничего не происходит.

Получается, что почти все самые важные действия производятся в обработчике события «кнопка нажата». Обычно при работе с кнопками вопросов не возникает: поместили кнопку на форму, дважды щелкнули и отредактировали процедуру-обработчик события Click. Однако в нашем случае все немного сложнее: во-первых, кнопка у нас не одна, не две и не три, а целый массив; во-вторых, обработчик события для всех кнопок один и тот же. Поэтому «стандартной» кнопкой типа TSpeedButton не воспользуешься: надо немного ее доработать (не забудьте добавить модуль Buttons в uses-список):

```
type MySpeedButton = class(TSpeedButton)    { "доработанная" кнопка }
  public
    x, y : Integer;                         { ее положение на игровом поле }
    clicked : Boolean;                      { нажата или нет }
    is_mine : Boolean;                      { находится ли в ней мина }
    procedure Click; override;              { обработчик события Click }
  end;
```

При работе в Delphi не так уж и часто приходится создавать новые классы, «вручную» наследуя их от существующих; здесь мы имеем дело с хорошим примером такого действия.

Для тех, кто не очень хорошо знаком с принципами ООП, разъясню ситуацию. Мы создали свою кнопку на основе библиотечного типа TSpeedButton. Помимо обычных полей стандартной кнопки (Left, Top, Width, Height, Caption, ...) наша реализация также содержит новые: x, y, clicked и is_mine. Кроме того, при щелчке на любой кнопке типа MySpeedButton будет автоматически вызвана процедура MySpeedButton.Click. Мы еще не запрограммировали эту процедуру, но уже явно указали ее как обработчик события «нажата кнопка».

Объявим теперь несколько констант и глобальных переменных:

```
const
  FieldWidth = 20;          { ширина поля }
  FieldHeight = 25,         { высота поля }
  CellsPerMine = 6;         { среднее значение количества клеток на одну мину }

var
  { игровое поле (двумерный массив кнопок) }
  Field : array[0..FieldWidth - 1, 0..FieldHeight - 1] of MySpeedButton;
  GameOver : Boolean = false;      { индикатор конца игры }
  Mines : Integer;                 { количество мин }
  Opened : Integer;                { количество открытых клеток }
```

Константа CellsPerMine регулирует среднее количество устанавливаемых мин на игровом поле.

Следующий шаг — создание процедуры инициализации игрового поля. Как вы помните, игровое поле — это массив кнопок Field. Пока что он содержит «ведущие в никуда» ссылки, и чтобы на их месте появились реальные кнопки, придется проделать некоторые действия.

```
procedure MakeButtons;                          { создание игрового поля }
var i, j : Integer;
begin
    for i := 0 to FieldWidth - 1 do
        for j := 0 to FieldHeight - 1 do
        begin
            Field[i, j] := MySpeedButton.Create(nil);  { создать кнопку }
            with Field[i, j] do
            begin
                Left := i * 16;      { указываем координаты кнопки }
                Top := j * 16;
                Width := 16;         { ее размеры }
                Height := 16;
                Parent := Form1;     { а также родительскую форму }
                x := i;              { x, y - позиция кнопки на игровом поле }
                y := j;
            end;
        end;
end;
```

Обычно программисту не приходится писать код, создающий кнопку и добавляющий ее на некоторую форму (среда Delphi автоматически генерирует требуемые инструкции, когда вы перетаскиваете мышью кнопку с палитры компонентов на форму), но на сей раз нам придется это сделать. Заметьте, мы не только создаем кнопку, указываем ее размеры и положение на главной форме, но и запоминаем координаты клетки (x, y), ей соответствующей.

Раз уж мы взяли на себя ответственность создавать объекты типа «кнопка» вручную, освобождение выделенной памяти — тоже наша забота. Сделать это можно при выходе из приложения (событие OnDestroy главной формы):

```
procedure TForm1.FormDestroy(Sender: TObject);      { уничтожение кнопок }
var i, j : Integer;
begin
    for i := 0 to FieldWidth - 1 do
        for j := 0 to FieldHeight - 1 do
            Field[i, j].Free;
end;
```

На самой главной форме приложения следует поместить одну кнопку (тип: TButton, имя: Button1), которая будет запускать игру и одну надпись (тип: TLabel, имя: Status), использующуюся для вывода текущего состояния (идет игра/победа/поражение). Картинки, встречающиеся в программе (мина, цифры 0–8) следует загрузить в объект ImageList типа TImageList. Размер каждой картинки — 16 × 16 пикселов (равен размеру кнопки), поэтому значения Width и Height объекта ImageList должны быть равны 16. Поскольку прозрачность не используется, в раскрывающемся меню Transparent Color выбираем clNone (рис. 7.2).

Главную форму, естественно, надо растянуть так, чтобы все игровое поле на ней поместилось.

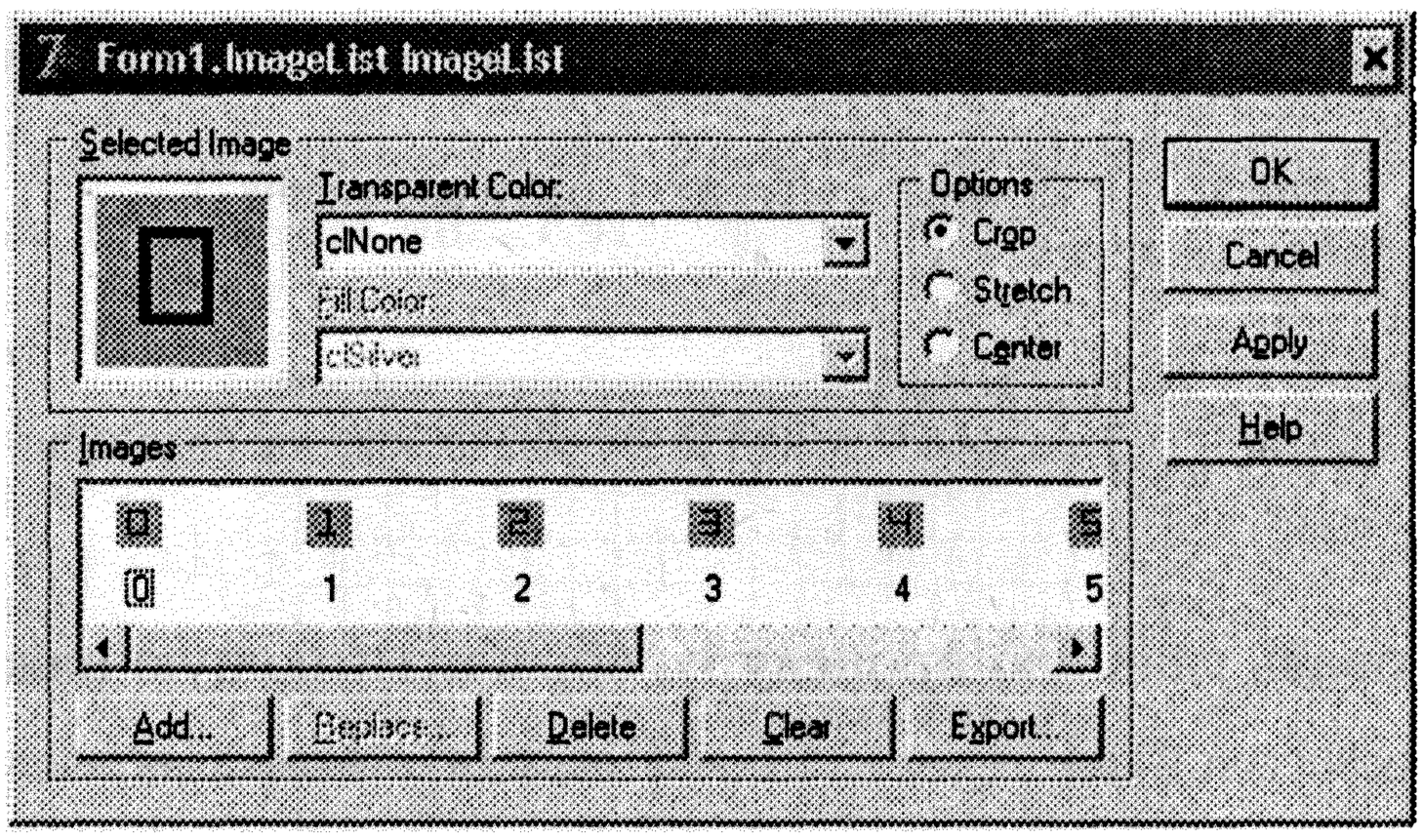


Рис. 7.2. Загрузка изображений в объект ImageList

Вовсе не обязательно загружать картинки из десяти разных файлов. Можно создать одно-единственное изображение, содержащее все картинки (рис. 7.3), а при загрузке ее в элемент ImageList нажать кнопку Yes в ответ на вопрос «Separate into 10 separate bitmaps?»

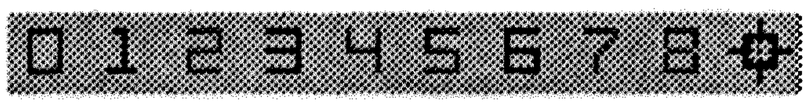


Рис. 7.3. Полная коллекция картинок для Сапера в одном файле

Старт игры происходит в двух случаях: сразу при запуске программы и если пользователь нажал кнопку «Новая игра» (Button1):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    StartGame;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Randomize;
    MakeButtons;   { при запуске программы создаем кнопки }
    StartGame;
end;
```

Сама процедура StartGame выглядит так:

```
procedure StartGame;
var i, j : Integer;
begin
    Form1.Status.Caption := '';     { строка статуса изначально пуста }
    Mines := 0;                     { счетчик мин }
    Opened := 0;                    { счетчик открытых клеток }
    GameOver := false;              { еще не конец игры }

    for i := 0 to FieldWidth - 1 do
        for j := 0 to FieldHeight - 1 do
```

```
begin
    Field[i, j].Glyph := nil;          { пока на кнопках нет картинок }
    Field[i, j].clicked := false;      { и ни одна из них не нажата }

    if Random(CellsPerMine) = 0 then { с вероятностью 1/CellsPerMine }
    begin
        Field[i, j].is_mine := true; { размещаем мину }
        Mines := Mines + 1;
    end
    else
        Field[i, j].is_mine := false; { иначе клетка безопасна }
        end;
end;
```

Осталось реализовать главную процедуру, обрабатывающую нажатия кнопок:

```
procedure MySpeedButton.Click;                  { обработка щелчка на кнопке }
var c : Integer;
    i, j    : Integer;
    dx, dy  : Integer;
begin
    if GameOver or clicked then Exit;      { если кнопка уже нажата или }
                                           { игра закончена, не обрабатываем }
    clicked := true;                       { теперь кнопка точно нажата }
    if is_mine then                        { если в ней оказалась мина }
    begin
        GameOver := true;                              { конец игры }
        Form1.Status.Caption := 'Поражение!';
        Form1.ImageList.GetBitmap(9, Glyph);  { выводим изображение мины (9) }
end
else
begin                                      { иначе подсчитываем }
    c := 0;                                { количество "опасных" соседей }
    for dx := -1 to 1 do
        for dy := -1 to 1 do
            if not ((dx = 0) and (dy = 0)) then { клетка сама себе }
            begin                                { не сосед }
                i := x + dx;               { координаты текущего соседа }
                j := y + dy;
                { если сосед не за пределами поля, учитываем его }
                { Integer(Field[i, j].is_mine) = 1, если есть мина, }
                { Integer(Field[i, j].is_mine) = 0 в противном случае }
                if (i >= 0) and (j >= 0) and (i < FieldWidth) and
                   (j < FieldHeight) then
                    c := c + Integer(Field[i, j].is_mine);
            end;
        Form1.ImageList.GetBitmap(c, Glyph); { рисуем требуемую картинку }

        Opened := Opened + 1;                { мы открыли еще одну клетку }
        if Opened + Mines = FieldWidth * FieldHeight then { открыты все }
        begin                                             { безопасные клетки }
            GameOver := true;                        { конец игры }
            Form1.Status.Caption := 'Победа!';
        end;
    end;
end;
```

Обратите внимание, что поля объекта типа MySpeedButton, используемые в процедуре Click (например, clicked или Glyph), относятся именно к нажатой кнопке, поэтому нам и удалось обойтись одним общим обработчиком события OnClick для всех кнопок на экране.

На этом создание Сапера можно считать делом законченным; внешний вид готового приложения показан на рис. 7.4.

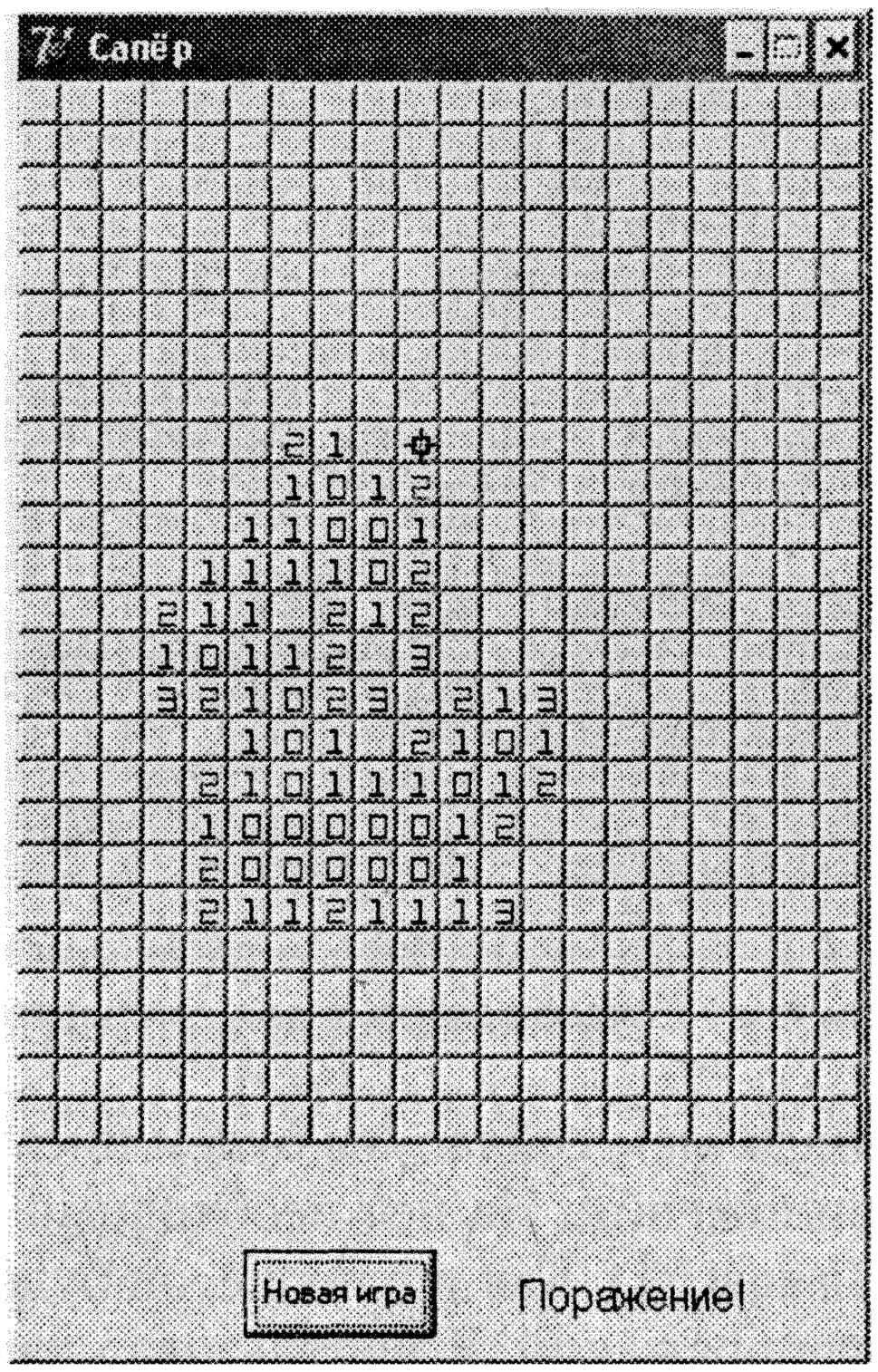


Рис. 7.4. Super Minesweeper 2003

Сокобан (Sokoban)

Сокобан – великолепная, знаменитая логическая игра, написанная в 1982 году японским программистом Хироюки Имабаяси (Hiroyuki Imabayashi). Довольно быстро она стала популярной во всем мире. Даже сейчас, набрав «Sokoban» в качестве запроса в любой поисковой системе, вы получите десятки релевантных ссылок на современные реализации этой игры.

Классический Сокобан (рис. 7.5) знаменит своими тщательно разработанными уровнями. Большинство современных версий могут пощеголять красивой графикой и стереозвуком, но подборка уровней редко оказывается столь же удачной.

Идея игры очень проста. Вы – Сокобан (что по-японски означает «кладовщик»), и вам требуется растолкать все ящики на уровне по местам. Ящики и места никак не пронумерованы, поэтому вы можете попытаться задвинуть любой ящик на любое место. Проблема в том, что двигать ящики разрешается только вперед, то есть по направлению движения кладовщика (одним словом, толкать можно,

тянуть нельзя); таким образом, если ящик оказался придвинутым к стене, его никак не удастся вернуть назад.

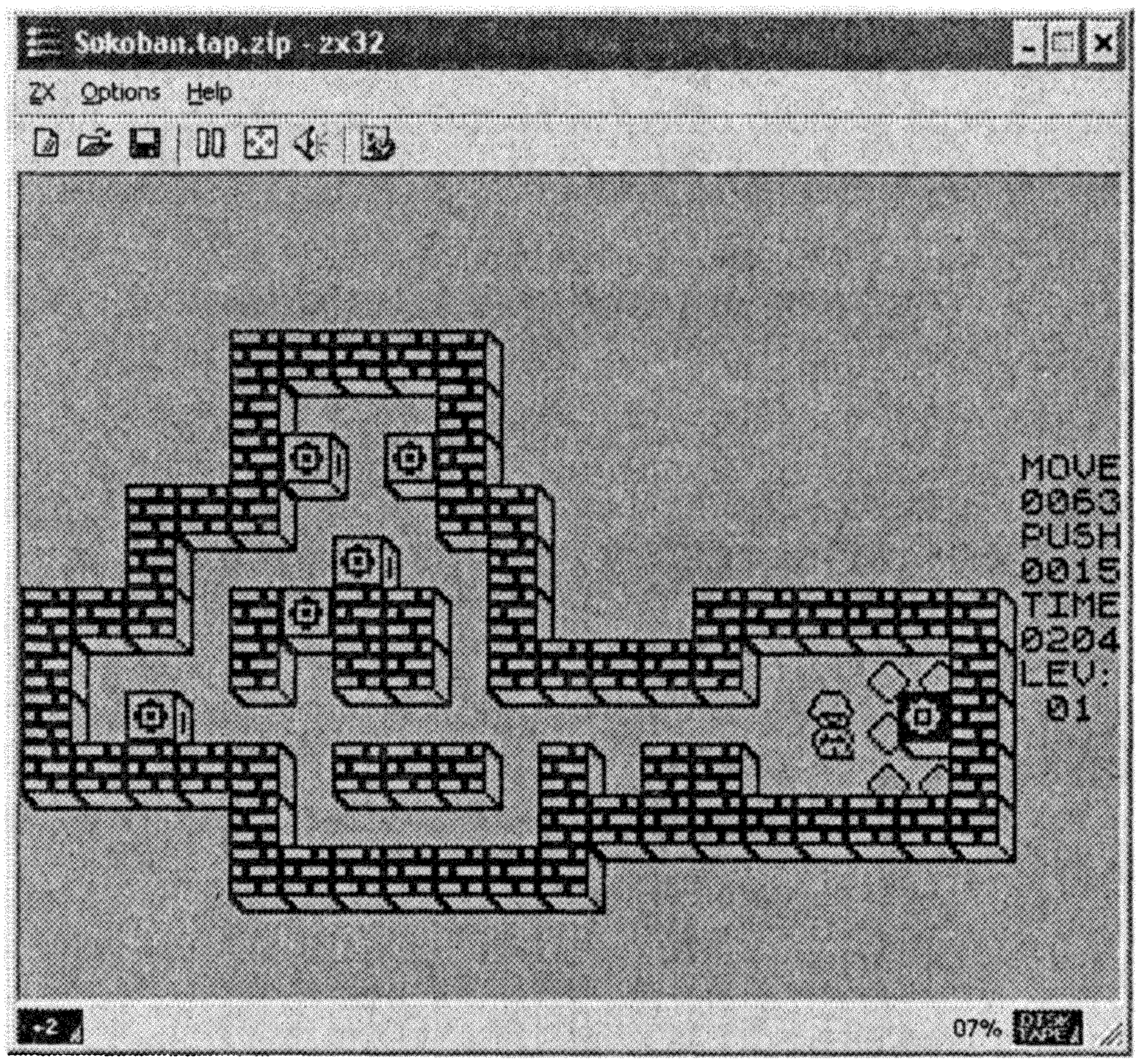


Рис. 7.5. Классический Сокобан (версия для компьютера ZX Spectrum)

Как вы уже, наверное, догадались, сейчас мы будем писать «свой» Сокобан. Начнем с главной формы приложения. На нее необходимо поместить элемент типа TImageList (назовите его ImageList). Как и раньше, он будет служить для хранения использующихся в игре изображений. Я использовал картинки размером 32 × 32 пиксела, взятые из игры Bulldozer – одной из реализаций Сокобана (рис. 7.6).

Рис. 7.6. Элементы игрового поля Сокобана

Первое изображение – это пустое поле, второе – целевая ячейка (то есть «место»), третье – камень (в нашем случае используется вместо ящика), четвертое – камень, находящийся на месте. Далее идут всевозможные положения главного героя и «стена».

Поскольку в нашем Сокобане будет возможность загрузки уровней из файла, нам потребуется элемент TOpenDialog (введите OpenDialog в качестве его имени). Для того чтобы применить механизм двойной буферизации, разместите на форме два элемента типа TImage (аналогично проектам из второй главы один из элементов будет называться Screen, а другой – BackBuffer; кроме того, элемент BackBuffer должен быть невидимым). Последний элемент дизайна формы – главное меню, содержащее всего два пункта: Открыть (файл с уровнем) и Выход. При выборе пункта Выход должна выполняться строка:

```
Application.Terminate;
```

Обработчик пункта Открыть мы напишем чуть позже.

Теперь можно постепенно добавлять новые участки кода в программу. Для начала, чтобы не иметь дела с «магическими числами», опишем несколько констант:

```
const WALL = 8; SPACE = 0; BOULDER = 2; PLACE = 1; PLACEDBOULDER = 3;
      SL = 6; SR = 7; SU = 4; SD = 5;     { индексы спрайтов }
```

Каждая из них обозначает номер какой-либо картинки, изображенной на рис. 7.6.

Нам также понадобятся переменные:

```
Field:  array[1..20, 1..12] of Integer;    { игровое поле }
Places: array[1..20, 1..12] of Boolean;    { положение мест }
N_of_places : Integer;                     { общее число мест на уровне }
CurX, CurY : Integer;                      { текущее положение Сокобана }
Positioned : Integer;                      { количество размещенных камней }
Busy : Boolean = false;                    { статус обработчика клавиш }
```

В качестве размера игрового поля я выбрал 20 × 12, хотя это, конечно, непринципиально. В массиве Field будут храниться числа, соответствующие тем или иным элементам игрового поля, а в массиве Places — значения True на позициях, соответствующих местам, и значения False во всех остальных случаях.

Следующая задача — загрузка уровней. Чтобы не усложнять программу, давайте полагать, что уровень хранится в обыкновенном текстовом файле (который можно создать и отредактировать Блокнотом), состоящем из 12 строк по 20 символов в каждой. Любой символ представляет собой какой-либо объект игрового поля: пробел — пустое пространство, буква «x» — стену, «b» — камень, «p» — место, «s» — стартовую позицию. Типичный уровень может выглядеть, к примеру, так:

```
xxxxxxxxxxxxxxxxxxxx
x                  x
x                  x
x  b         x     x
x            x     x
x        s   x     x
x            x     x
x     p      xx xxxx
x           bx     x
x            x p   x
x            x     x
xxxxxxxxxxxxxxxxxxxx
```

Обратите внимание на то, что уровень всюду окружен стенами: если вы позаботитесь, чтобы это условие всегда выполнялось, то не придется программировать проверку на выход персонажа за пределы экрана. Еще один важный момент: последняя строка файла (как и все до нее) должна заканчиваться парой символов «возврат каретки»/«перевод строки» — в этом случае процедура загрузки уровня реализуется проще.

Так как элементы файла — символы, а элементы реального игрового поля — картинки (точнее, их порядковые номера), нам понадобится функция, переводящая символ из файла в порядковый номер (код) картинки:

```
function SymbolToCode(c : Char) : Integer;  { перевод символа в порядковый }
begin                                       { номер картинки }
    case c of
        'x': SymbolToCode := WALL;          { 'x' - стена }
        ' ': SymbolToCode := SPACE;         { ' ' - пустое пространство }
        'b': SymbolToCode := BOULDER;       { 'b' - камень }
        's': SymbolToCode := SL;            { 's' - стартовая локация }
    else
        SymbolToCode := -1;                 { неизвестный символ }
    end;
end;
```

Теперь можно написать функцию загрузки уровня:

```
procedure LoadLevel(FileName : String);     { загрузить уровень из файла }
var f : File of Char;
    i, j : Integer;
    c : Char;
begin
    AssignFile(f, FileName);                { открыть файл }
    Reset(f);

    N_of_places := 0;                       { подсчет количества мест }
    for j := 1 to 12 do                     { цикл по строкам }
    begin
        for i := 1 to 20 do                 { цикл по элементам строки }
begin
    Read(f, c);                             { считываем текущий элемент }
    if c = 'p' then                         { если это "место" }
    begin
        Field[i, j] := 0;       { в поле равнозначно пустому элементу }
        Places[i, j] := true;               { True в массиве Places }
        N_of_places := N_of_places + 1;
    end
    else
    begin
        Places[i, j] := false;              { иначе НЕ "место" }
        Field[i, j] := SymbolToCode(c);     { определяем код элемента }

        if c = 's' then                     { если текущая локация - }
        begin                               { стартовая, запоминаем }
            CurX := i;                      { ее координаты }
                CurY := j;
            end;
        end;
    end;
    Read(f, c);  { считываем и пропускаем возврат каретки }
    Read(f, c);  { считываем и пропускаем перевод строки }
end;

    CloseFile(f);       { закрыть файл }
end;
```

Заметьте, что элемент «место» в массиве Field эквивалентен пустому пространству, поскольку он, как и пустая клетка, беспрепятственно проходится кладовщиком.

Надеюсь, теперь понятно, зачем нужны символы возврата каретки и перевода строки в конце файла: в противном случае на последней итерации цикла произойдет попытка считывания несуществующих элементов.

Чтобы еще упростить программу, я решил, что мы можем позволить себе перерисовывать экран полностью при каждом движении персонажа (в 1982 году подобная политика была недопустимой роскошью). А если так, то следующая необходимая нам процедура — обновление экрана (иными словами, перерисовка уровня):

```
procedure RedrawField;                  { перерисовка уровня }
var i, j   : Integer;
    code   : Integer;
    bitmap : TBitmap;
begin
    bitmap := TBitmap.Create;      { объект для временного хранения рисунка }
    Positioned := 0;               { считаем, что размещено 0 камней }

    for j := 1 to 12 do
        for i := 1 to 20 do
        begin
            code := Field[i, j];                          { код текущего элемента }

            { пара "пустое пространство" / "место" означает "место" }
            if (Field[i, j] = SPACE) and Places[i, j] then
                code := PLACE
            { а пара "камень" / "место" - "камень на месте" }
            else if (Field[i, j] = BOULDER) and Places[i, j] then
            begin
                code := PLACEDBOULDER;
                Positioned := Positioned + 1;  { при этом увеличиваем }
            end;                               { счетчик размещенных камней }

            Form1.ImageList.GetBitmap(code, bitmap);  { достаем картинку }
            { и рисуем ее на соответствующем месте виртуального экрана }
            Form1.BackBuffer.Canvas.Draw((i - 1)*32, (j - 1)*32, bitmap);
        end;

    bitmap.Free;

    { копируем содержимое виртуального экрана на основной }
    Form1.Screen.Canvas.CopyRect(Rect(0, 0, 640, 384),
                                 Form1.BackBuffer.Canvas, Rect(0, 0, 640, 384));
end;
```

Теперь становится очевидным, что происходит при выборе пункта меню Открыть:

```
if OpenDialog.Execute then
begin
    LoadLevel(OpenDialog.FileName);
    RedrawField;
end;
```

При запуске программы (событие OnShow главной формы) тоже следует сразу же запросить у пользователя загружаемый уровень. Более того, пока уровень не будет загружен, двигаться дальше просто невозможно:

```
while not OpenDialog.Execute do
    ;
LoadLevel(OpenDialog.FileName);
RedrawField;
```

Вот, практически, и все. Осталось лишь отреагировать на команды кладовщику, поступающие от курсорных клавиш. Сделать это можно в процедуре-обработчике события KeyDown главной формы:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
var dx, dy : Integer;           { смещения Сокобана (куда идем) }
    SprIdx : Integer;           { текущий номер его спрайта }
begin
    if Busy then Exit;          { если обработчик занят, выходим }

    Busy := true;               { статус = занято }
    case Key of
        VK_LEFT: begin dx := -1; dy := 0; SprIdx := SL; end;   { идем влево }
        VK_RIGHT: begin dx := 1; dy := 0; SprIdx := SR; end;   { вправо }
        VK_UP: begin dx := 0; dy := -1; SprIdx := SU; end;     { вверх }
        VK_DOWN: begin dx := 0; dy := 1; SprIdx := SD; end;    { вниз }
    else
        begin dx := 0; dy := 0; SprIdx := Field[CurX, CurY]; end; { никуда }
    end;                                                       { не идем }

    if Field[CurX + dx, CurY + dy] = SPACE then  { если целевая клетка пуста }
    begin
        Field[CurX + dx, CurY + dy] := SprIdx;   { переходим в нее }
        Field[CurX, CurY] := SPACE;    { на старой позиции теперь ничего нет }
        CurX := CurX + dx;
        CurY := CurY + dy;
    end
    { если целевая клетка содержит камень, а клетка, следующая за ней, пуста }
    else if (Field[CurX + dx, CurY + dy] = BOULDER) and
            (Field[CurX + 2*dx, CurY + 2*dy] = SPACE) then
    begin                                              { двигаем камень: }
        Field[CurX + dx, CurY + dy] := SprIdx;         { новая позиция Сокобана }
        Field[CurX + 2*dx, CurY + 2*dy] := BOULDER;    { новая позиция камня }
        Field[CurX, CurY] := SPACE;                    { на старой позиции - пусто }
        CurX := CurX + dx;
        CurY := CurY + dy;
    end;

    RedrawField;                                       { перерисовываем уровень }
    { если все камни размещены }
    if Positioned = N_of_places then
                  Application.MessageBox('Уровень пройден!',
                  'Sokoban', MB_ICONEXCLAMATION);

    Application.ProcessMessages;

    Busy := false;              { статус = свободно }
end;
```

Медленный компьютер может просто не справиться с перерисовыванием всего уровня за короткий интервал времени между поступающими командами от клавиатуры. Избежать подобной ситуации можно, используя переменную Busy, как показано в листинге.

Внешний вид готовой программы показан на рис. 7.7.

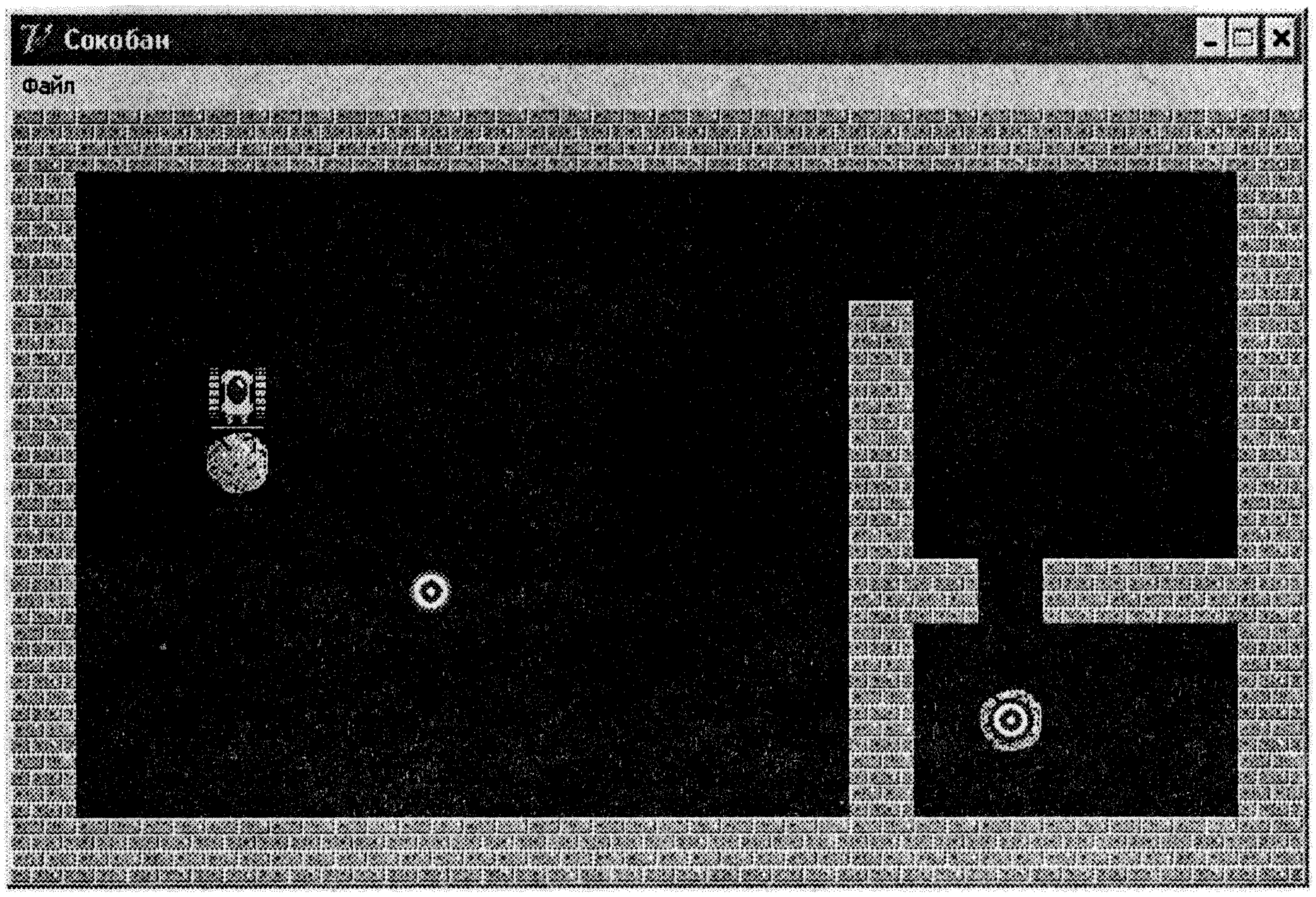


Рис. 7.7. Mega Sokoban Millennium

Удав (Snake)

Думаю, эта игра может претендовать на одно из первых мест в номинации «за распространенность». Действительно, ее вариации (порою очень изощренные) можно встретить практически на любом устройстве, имеющем микропроцессор, начиная с карманных «тетрисов» и мобильных телефонов и заканчивая современными игровыми приставками и компьютерами (рис. 7.8).

Идея любого варианта Удава проста: съедать какие-либо предметы, разбросанные по игровому полю (ягоды, цифры или даже лягушек), при этом, по-возможности, избегая других (ядовитых) предметов. Столкновение головы удава с краем уровня (во многих реализациях) или собственным хвостом (во всех) означает потерю жизни. Проблема в том, что при поедании предметов удав растет, и управлять им становится все сложнее. В простых вариантах игры этим все и ограничивается (цель состоит в том, чтобы вырастить как можно более длинного удава); в «продвинутых» же версиях вас время от времени переводят на новый уровень со своими сложностями.

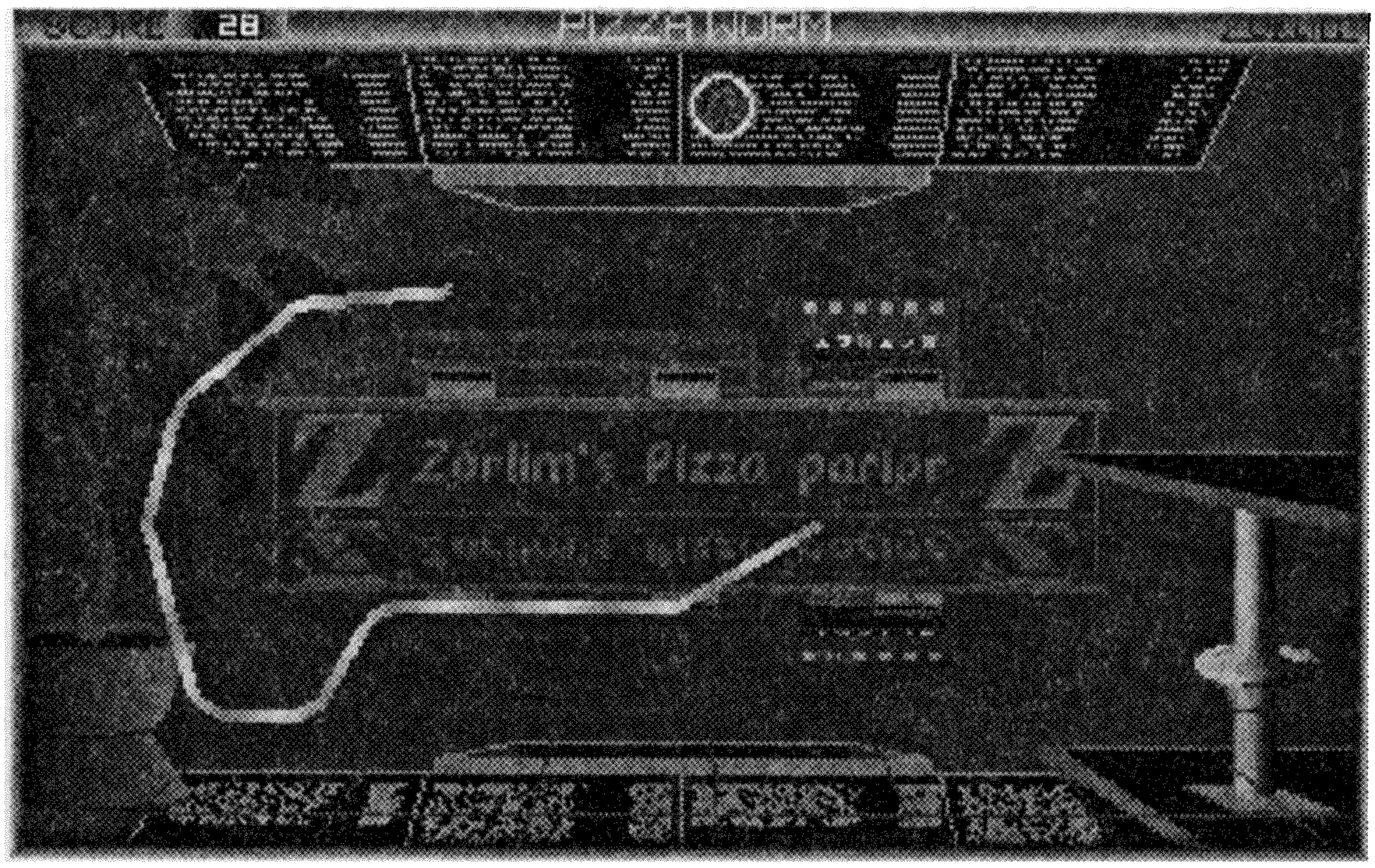


Рис. 7.8. Игра Pizza Worm (PC)

Наша разновидность игры будет аналогична версии под названием Удав-Цифроед, в свое время написанной для компьютера ZX Spectrum. Суть ее состоит в том, чтобы вырастить как можно более длинного удава, а специфика – в подборке «съедобных» объектов, которыми являются цифры от единицы до девяти. Съев цифру, удав удлиняется на соответствующее количество сегментов.

Хотя Удав Цифроед – игра весьма незатейливая, в ее реализации есть тонкие моменты. Рассмотрим сначала алгоритм, записанный на псевдокоде:

```
очистить экран
нарисовать змея из трех сегментов (голова, тело и хвост)
поместить случайную цифру на игровое поле
ОСНОВНОЙ ЦИКЛ
    определить новые координаты головы (куда идем)
    ЕСЛИ ход запрещен, конец игры
    ЕСЛИ целевая клетка содержит цифру
        ToGrow := цифра      { на сколько сегментов еще расти }
        поместить случайную цифру в другом месте поля
    ЕСЛИ ToGrow > 0
        ToGrow := ToGrow - 1          { удав растет }
        длина удава := длина удава + 1
    обновить координаты удава (т.е. сдвинуть его)
    обработать сообщения от клавиатуры
КОНЕЦ ЦИКЛА
```

На таком высоком уровне абстракции все должно быть более или менее понятно. Вопросы возникают при переходе к деталям.

1. «Поместить случайную цифру на игровое поле». Во время этой операции надо убедиться, что новая цифра появляется на свободном участке поля, а не на теле самого удава. Проще всего генерировать случайные координаты цифры до тех пор, пока они не окажутся допустимыми (иными словами, запускать генератор случайных чисел до победного конца). Подобная идея не всегда срабатывает удовлетворительно, однако в данном случае доля «холостых» запусков должна быть невелика – количество свободных клеток поля обычно

гораздо больше (или, по крайней мере, ненамного меньше) количества клеток, занятых удавом.

2. «Определить новые координаты головы». В отличие от персонажа игры Сокобан, удав всегда находится в движении: нажатия клавиш только корректируют направление.
3. «ЕСЛИ ход запрещен, конец игры». Несложно определить, что координаты головы оказались за пределами поля; но нам требуется также уметь вычислять ситуацию, когда змей врезался в собственный хвост. Решение, которое мы будем использовать (оно пригодится нам и в дальнейшем), состоит в том, чтобы хранить координаты каждого элемента удава в специальном массиве:

```
Snake[1] = координаты хвоста
Snake[2] = координаты последнего элемента тела
...
Snake[длина удава - 1] = координаты первого элемента тела
Snake[длина удава] = координаты головы
```

Чтобы определить допустимость хода, придется пробежаться по массиву и убедиться, что целевая клетка (куда идем) не совпадает ни с одной клеткой, занятой удавом. К сожалению, с ростом удава размер массива увеличивается, поэтому расходы на проверку допустимости каждого хода тоже изменяются «не в нашу пользу». Тем не менее мне кажется, что поскольку длина удава — величина сравнительно небольшая (десятки, сотни сегментов), то для компьютера не составит особого труда «пробежаться» по соответствующему массиву на каждой итерации главного цикла (тем более что другой работы будет не так уж и много).

4. «Обновить координаты удава». Это действие в свою очередь разбивается на составляющие:

```
записать координаты целевой клетки в Snake[длина удава + 1]
стереть с экрана хвост (координаты хвоста – в Snake[1])
нарисовать хвост на месте последнего элемента тела (Snake[2])
сдвинуть данные в массиве Snake.
    Snake[1] := Snake[2]
    Snake[2] := Snake[3]
    ...
    Snake[длина удава] := Snake[длина удава + 1]
нарисовать сегмент тела в клетке Snake[длина удава - 1]
нарисовать голову в клетке Snake[длина удава]
```

Пожалуй, единственный способ убедиться в том, что все здесь правильно, — это, вооружившись карандашом и клетчатой бумагой, вручную проделать описанные действия. Не забывайте, что удав может расти (на один сегмент за итерацию).

Создание приложения начнем, как всегда, с дизайна главной формы. На сей раз на ней будет совсем немного элементов: Screen (поверхность для рисования; тип — TImage), ImageList (набор картинок, использующихся в игре; тип — TImageList) и SnakeLenLabel (надпись, отображающая текущую длину удава; тип — TLabel).

В элемент ImageList следует загрузить коллекцию картинок (точно таким же образом, как в Сокобане и Сапере). Используемые в игре изображения показаны на рис. 7.9.

Рис. 7.9. Элементы игры Удав

Первые девять изображений – цифры (как нетрудно заметить), десятое – хвост удава, одиннадцатое – сегмент тела, двенадцатое – голова, а последнее – пустое пространство. Размер каждой такой картинки – 16 × 16 пикселов, поэтому при использовании поля в 40 × 30 клеток размер экрана оказывается равным 640 × 480 пикселов.

Разобравшись с оформлением, перейдем к написанию кода. Опишем сначала константы и переменные:

```
type Coords = record                                    { координаты клетки }
    x, y : Integer;
end;

const FieldHeight = 30; FieldWidth = 40;                { размеры поля }
      MSecsPerFrame = 150;                              { скорость игры }
var
  Form1: TForm1;
  Snake: array[1.. FieldHeight * FieldWidth] of Coords;     { удав }
  SnakeLength: Integer;                                    { его длина }
  dx, dy: Integer;                                      { текущее направление }
  Background, Body, Head, Tail: TBitmap;                { элементы игрового поля }
  FirstRun: Boolean = true;                             { индикатор первого запуска }
  Number: Coords;                                       { положение текущей цифры }
  CurNumber: Integer;                                   { и ее значение }
```

Понятно, что длина удава не может быть больше общего количества клеток на игровом поле, поэтому массив Snake состоит из FieldHeight * FieldWidth элементов.

Поскольку мы будем постоянно иметь дело с картинками, соответствующими голове, телу и хвосту удава, а также свободному пространству, имеет смысл загрузить их в соответствующие объекты типа TBitmap. Сделать это можно в специальной процедуре LoadBitmaps:

```
procedure LoadBitmaps;                                  { загрузка картинок }
begin                                                   { (элементов игры) }
    Background := TBitmap.Create;
    Body := TBitmap.Create;
    Head := TBitmap.Create;
    Tail := TBitmap.Create;

    Form1.ImageList.GetBitmap(9, Tail);
    Form1.ImageList.GetBitmap(10, Body);
    Form1.ImageList.GetBitmap(11, Head);
    Form1.ImageList.GetBitmap(12, Background);
end;
```

Если где-то в программе выделяется память, где-то в другом месте следует ее освободить:

```
procedure FreeBitmaps;                                  { освобождение памяти }
begin
    Background.Free;
    Body.Free;
```

```
    Head.Free;
    Tail.Free;
end;
```

Перед запуском игры должна происходить очистка экрана. Это действие будет выполняться в процедуре ClearField:

```
procedure ClearField;                          { очистка экрана }
var i, j : Integer;
begin
    for i := 0 to FieldWidth do
        for j := 0 to FieldHeight do
            Form1.Screen.Canvas.Draw(i*16, j*16, Background);
end;
```

Следующая процедура инициализирует массив Snake и рисует «новорожденного» удава на экране:

```
procedure InitSnake;                              { инициализация удава }
begin
    SnakeLength := 3;  { изначально длина равна трем (голова, тело и хвост) }
    Snake[1].x := 0; Snake[2].x := 1; Snake[3].x := 2;  { координаты }
    Snake[1].y := 0; Snake[2].y := 0; Snake[3].y := 0;  { сегментов }
    dx := 1; dy := 0;                          { изначально удав ползет вправо }
    Form1.Screen.Canvas.Draw(0, 0, Tail);      { рисуем удава }
    Form1.Screen.Canvas.Draw(16, 0, Body);
    Form1.Screen.Canvas.Draw(32, 0, Head);
end;
```

Мы уже обсуждали, как можно определить допустимость следующего хода: сначала необходимо убедиться, что целевая клетка находится в пределах игрового поля, а затем «пробежаться» по массиву Snake и выявить, столкнулся удав со своим хвостом или нет. Функция IsSnake(x, y) возвращает true, если клетка (x, y) занята телом удава; в противном случае результатом будет false:

```
function IsSnake(X, Y: Integer) : Boolean;  { определение статуса клетки }
var i : Integer;                            { (удав/свободно) }
begin
    IsSnake := false;
    for i := 1 to SnakeLength do
        if (Snake[i].x = X) and (Snake[i].y = Y) then IsSnake := true;
end;
```

Последняя служебная процедура размещает на экране случайную цифру (о ней мы тоже уже говорили):

```
procedure PlaceNumber;                          { разместить случайную цифру }
var Temp : TBitmap;                             { на игровом поле }
begin
    Temp := TBitmap.Create;          { временный объект для хранения картинки }
    repeat
        Number.x := Random(FieldWidth);      { выбираем случайные координаты }
        Number.y := Random(FieldHeight);
    until not isSnake(Number.x, Number.y);  { пока они не окажутся }
                                            { допустимыми }
    CurNumber := 1 + Random(9);           { выбираем случайное значение цифры }
    Form1.ImageList.GetBitmap(CurNumber - 1, Temp);           { рисуем цифру }
    Form1.Screen.Canvas.Draw(16*Number.x, 16*Number.y, Temp); { на экране }
```

```
    Temp.Free;
end;
```

Реакция на нажатия клавиш программируется точно так же, как и в Сокобане – внутри процедуры-обработчика события **KeyDown**:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key  Word;
  Shift: TShiftState);
begin                                                { обработка клавиш }
    case Key of
        VK_LEFT: begin dx := -1; dy := 0; end;       { влево }
        VK_RIGHT: begin dx := 1; dy := 0; end;       { вправо }
        VK_UP: begin dx := 0; dy := -1; end;         { вверх }
        VK_DOWN: begin dx := 0; dy := 1; end;        { вниз }
    end;
end;
```

Основная программа размещается в процедуре-обработчике события **Activate** главной формы приложения:

```
procedure TForm1.FormActivate(Sender: TObject);    { главная процедура }
var oldtime : TDateTime;
    NewX, NewY : Integer;                    { координаты целевой клетки }
    ToGrow : Integer;
begin
    if not FirstRun then Exit; { процедура работает лишь при первом запуске }

    FirstRun := false;
    Randomize;
    LoadBitmaps;                  { загрузка изображений }

    while ID_CANCEL <> Application.MessageBox('OK - запуск, Cancel - выход',
                                              'Snake', MB_OKCANCEL) do
    begin
        ClearField;                { очистка игрового поля }
        InitSnake;                 { создание удава }
        PlaceNumber;               { генерация случайной цифры }
        ToGrow := 0;               { количество сегментов, на которое }
                                   { должен вырасти удав }
        while true do              { главный цикл }
        begin
            oldtime := Now;

            NewX := Snake[SnakeLength].x + dx;       { определение }
            NewY := Snake[SnakeLength].y + dy;       { целевой клетки }

            { если сделан недопустимый ход, конец игры }
            if (NewX < 0) or (NewX >= FieldWidth) or (NewY < 0) or
               (NewY >= FieldHeight) or IsSnake(NewX, NewY) then
                Break;   { конец игры }

            if (NewX = Number.x) and (NewY = Number.y) then  { если текущая }
            begin                                            { клетка содержит цифру }
                ToGrow := ToGrow + CurNumber;     { программируем рост удава }
                PlaceNumber;                      { генерируем очередную цифру }
            end;
```

```
if ToGrow > 0 then              { если удав растет }
begin
    ToGrow := ToGrow - 1;
    SnakeLength := SnakeLength + 1;    { увеличиваем его длину }
end;

Snake[SnakeLength + 1].x := NewX;  { обновление координат удава }
Snake[SnakeLength + 1].y := NewY;

Form1.Screen.Canvas.Draw(16*Snake[1].x, 16*Snake[1].y,
                                                    Background);
Form1.Screen.Canvas.Draw(16*Snake[2].x, 16*Snake[2].y, Tail);

{ сдвиг массива }
Move(Snake[2], Snake[1], SizeOf(Coords) * SnakeLength);

Form1.Screen.Canvas.Draw(16*Snake[SnakeLength - 1].x,
                         16*Snake[SnakeLength - 1].y, Body);
Form1.Screen.Canvas.Draw(16*Snake[SnakeLength].x,
                         16*Snake[SnakeLength].y, Head);

{ вывод на экран текущей длины удава }
Form1.SnakeLenLabel.Caption := IntToStr(SnakeLength);

{ задержка перед следующим циклом }
Application.ProcessMessages;
          while Round(MSecsPerFrame - (Now - oldtime) * MSecsPerDay) > 0 do
              Application.ProcessMessages;
        end;
    end;
    FreeBitmaps;                { освобождение выделенной памяти }
    Application.Terminate;      { выход из программы }
end;
```

В принципе, алгоритмическую основу процедуры мы уже обсудили — ничего принципиально нового в листинге нет. Запуск алгоритма происходит при выводе на экран главной формы. К сожалению, нельзя воспользоваться обработчиком события Show — при его вызове форма еще не отображается на экране, вывод формы происходит лишь после вызова FormShow(). Можно воспользоваться событием Activate, которое происходит при активизации формы (что мы и сделали), но тогда возникает другая проблема: стоит переключиться на какое-нибудь другое приложение (чтобы форма стала неактивной), а потом назад — и событие Activate вновь произойдет. Чтобы избежать многократного вызова основной процедуры, используется переменная FirstRun. Если процедура FormActivate() будет вызвана повторно, ее выполнение сразу же завершится:

```
if not FirstRun then Exit; { процедура работает лишь при первом запуске }
```

Обратите внимание также на строку

```
Move(Snake[2], Snake[1], SizeOf(Coords) * SnakeLength);
```

Процедура **Move(src, dest, size)** копирует **size** байтов из участка памяти, на который указывает параметр src, в участок, указываемый параметром dest. Таким образом, вызов

```
Move(Snake[2], Snake[1], SizeOf(Coords) * SnakeLength);
```

осуществляет копирование:

```
Snake[1] := Snake[2]
Snake[2] := Snake[3]
...
Snake[SnakeLength] := Snake[SnakeLength + 1]
```

Поскольку нам надо скопировать SnakeLength элементов, а размер каждого из них равен SizeOf(Coords) байтов, общий размер (в байтах) составляет SizeOf(Coords) * SnakeLength.

Синхронизация с таймером происходит точно так же, как и в примерах из главы об анимации.

Вот и все. Внешний вид работающей программы показан на рис. 7.10.

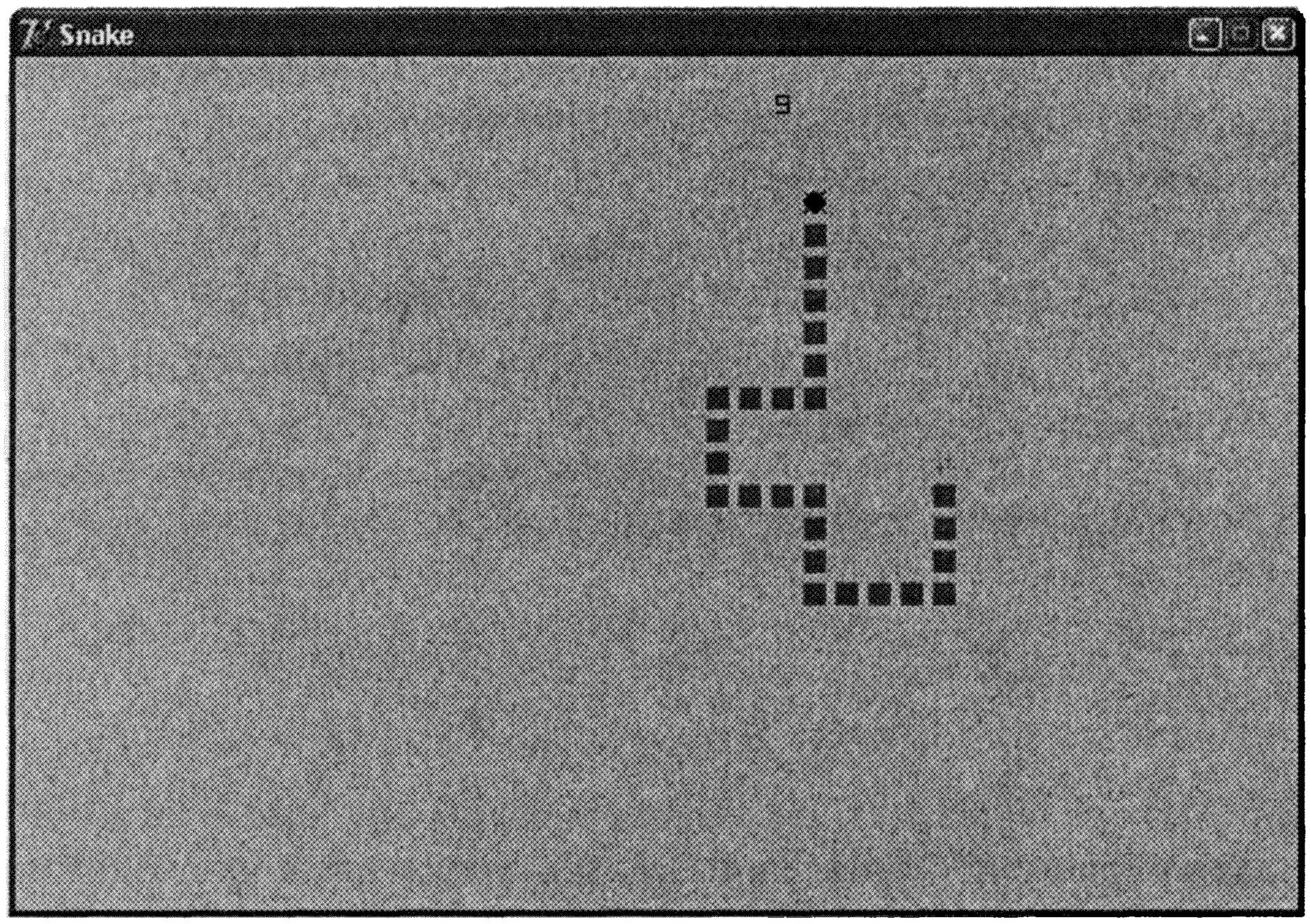


Рис. 7.10. New Age Snake Atomic Edition

Тетрис (Tetris)

Думаю, мне даже не стоит рассказывать об этой поистине великой игре (год за годом признаваемой экспертами лучшей компьютерной игрой всех времен и народов). Особенно приятен факт, что ее автор Алексей Пажитнов – наш соотечественник.

Как и Удав, Тетрис реализован под бесчисленное количество платформ. Для примера привожу снимок экрана версии, написанной для приставки NES, более известной в нашей стране как Dendy (рис. 7.11).

Написать Тетрис сложнее, чем другие игры из главы, но ненамного. Перед тем как разобраться с идеологией программы, спроектируем довольно-таки очевидный интерфейс (то есть разместим элементы управления на главной форме).

Уверен, вы без труда угадаете большинство размещаемых объектов: два элемента Screen и BackBuffer типа TImage (причем объект BackBuffer – невидимый), элемент ImageList типа TImageList и элемент LinesLabel типа TLabel. Надпись LinesLabel будет использоваться для вывода количества собранных линий, а назначение остальных элементов вам уже известно из предыдущих проектов.

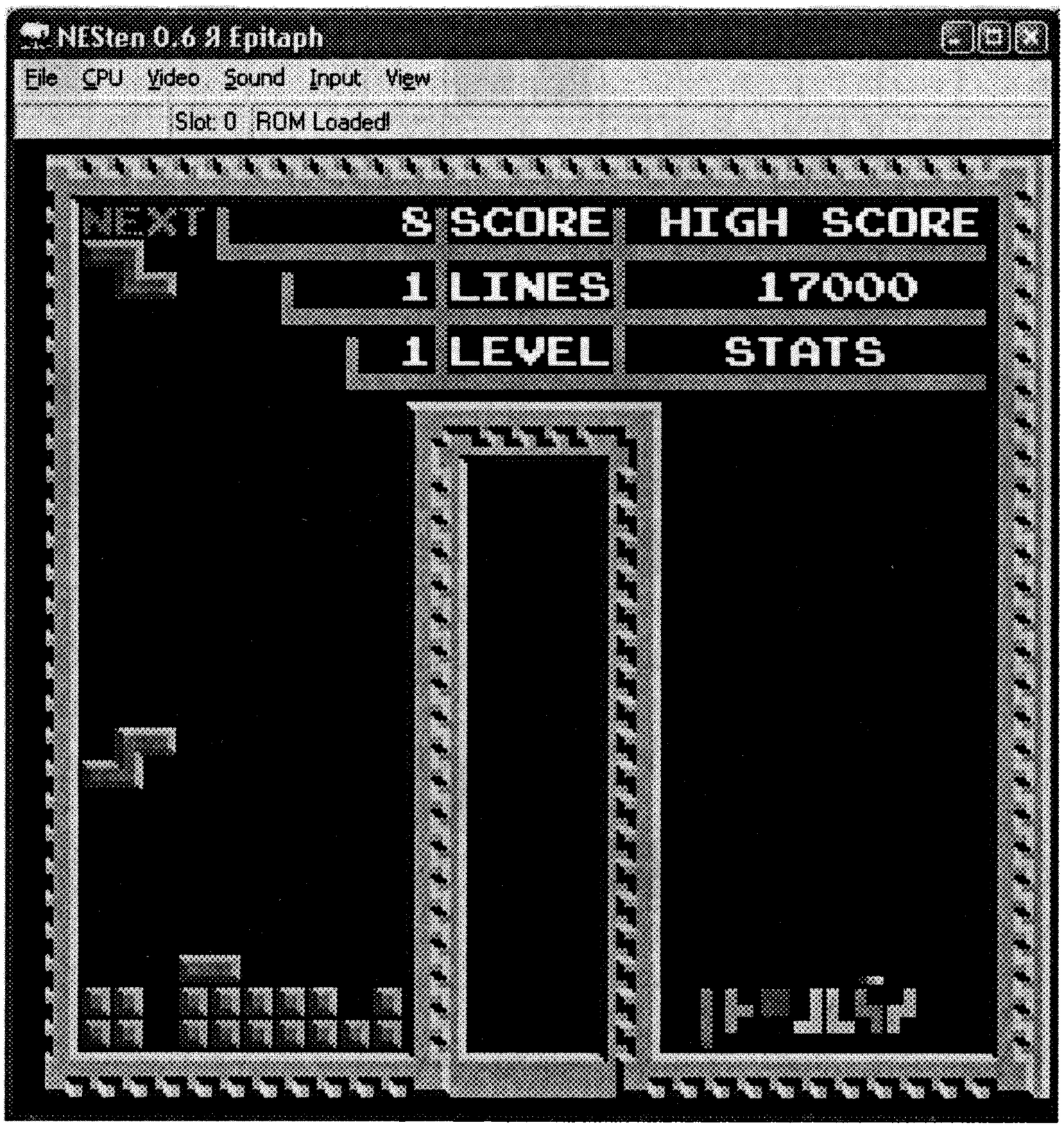


Рис. 7.11. Tetris (версия для приставки NES)

Давайте теперь прикинем размеры игрового экрана. Основной графический элемент Тетриса – это квадратный «строительный блок». Из четырех таких блоков состоит любая падающая в стакан фигура (поэтому Тетрис и назван Тетрисом). Мне кажется вполне разумным установить размер экрана равным 10 × 20 («строительных блоков»). Если при этом размер блока равен 22 × 22 пиксела (мой личный выбор, на котором я не настаиваю), то ширина экрана в итоге окажется равной 220, а высота – 440 пикселам.

Как и прежде, в элемент ImageList необходимо загрузить используемые в игре изображения. На сей раз ими будут просто «строительные блоки» разных цветов (рис. 7.12). Я остановился на четырех цветах (что, естественно, непринципиально) плюс блок фонового цвета, который идет первым в списке.

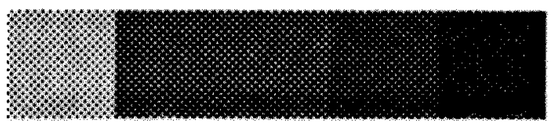


Рис. 7.12. «Строительные блоки»

Игру будем разрабатывать так: сначала напишем все процедуры, без которых обойтись ну просто нельзя, а там станет видно, чего не хватает.

Первое, что приходит на ум, – работа с фигурами, падающими в стакан. Всего их в Тетрисе семь штук (рис. 7.13).

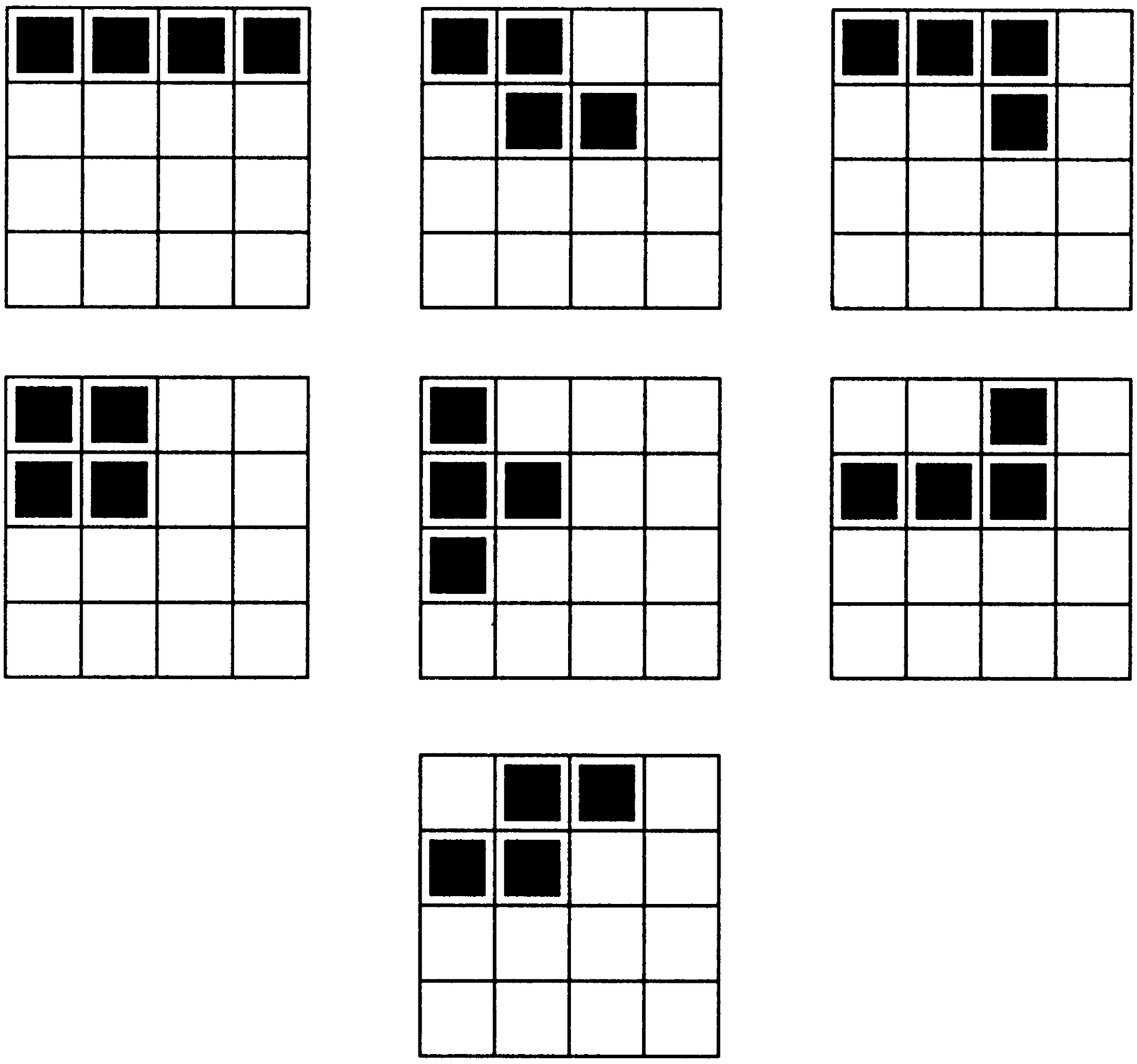


Рис. 7.13. Фигуры Тетриса

Некоторые из фигур могут быть получены друг из друга с помощью зеркального отражения, но поскольку по правилам игры фигуры можно только вращать, но не отражать, будем считать такие различными.

Самый простой и естественный способ представления фигур в памяти – трехмерный массив Pieces:

```
Pieces : array[1..7, 0..3, 0..3] of Integer = ( ((1,1,1,1),
                                                 (0,0,0,0),
                                                 (0,0,0,0),
                                                 (0,0,0,0)),

                                                ((1,1,0,0),
                                                 (0,1,1,0),
                                                 (0,0,0,0),
                                                 (0,0,0,0)),

                                                ((1,1,1,0),
                                                 (0,0,1,0),
                                                 (0,0,0,0),
                                                 (0,0,0,0)),
```

```
((1,1,0,0),
 (1,1,0,0),
 (0,0,0,0),
 (0,0,0,0)),

((1,0,0,0),
 (1,1,0,0),
 (1,0,0,0),
 (0,0,0,0)),

((0,0,1,0),
 (1,1,1,0),
 (0,0,0,0),
 (0,0,0,0)),

((0,1,1,0),
 (1,1,0,0),
 (0,0,0,0),
 (0,0,0,0)) );
```

Нарисовать ту или иную фигуру во время работы программы несложно. Если пользователь нажимает клавиши «влево» или «вправо», это влечет за собой лишь смену горизонтальной координаты текущей фигуры. «Интересности» начинаются при вращении. Тут уж не обойтись без преобразований (альтернативный вариант — хранение всех возможных состояний фигуры — мне кажется более громоздким). Таких преобразований будет два. Во-первых, что очевидно, поворот фигуры вокруг своей оси на 90 градусов (по часовой стрелке):

```
procedure Rotate90(piece : Integer);        { поворот фигуры на 90 градусов }
var i, j : Integer;
    temp : array[0..3, 0..3] of Integer;  { "временная" фигура }
begin
    for i := 0 to 3 do       { поворот на 90 градусов по часовой стрелке }
        for j := 0 to 3 do
            temp[3 - j, i] := Pieces[piece, i, j];

    for i := 0 to 3 do        { копируем измененную фигуру в исходный массив }
        for j := 0 to 3 do
            Pieces[piece, i, j] := temp[i, j];
end;
```

Второе менее очевидно. При повороте (если он реализован описанным способом) происходит сдвиг фигуры относительно ее предыдущего положения (рис. 7.14).

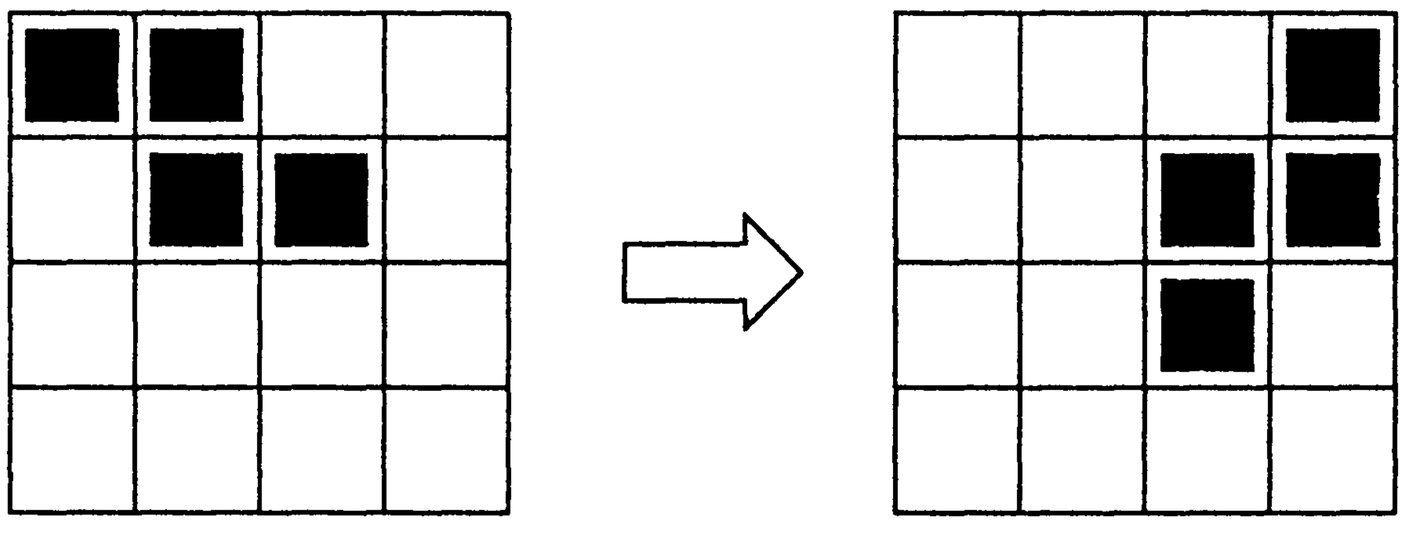


Рис. 7.14. Сдвиг при повороте

Для игрока это будет выглядеть так, словно фигура «скачет» туда-сюда при нажатии на клавишу поворота вокруг оси. Разумеется, подобных эффектов не должно наблюдаться. Для этого следует сдвинуть повернутую фигуру так, чтобы она находилась в левом верхнем углу своей матрицы размером 4 × 4 клетки, с помощью процедуры ToCorner():

```
procedure ToCorner(piece : Integer);          { сдвиг фигуры в угол }
var i, j : Integer;
label exit1, exit2;
begin
    while true do                          { цикл вертикального сдвига }
    begin
        for i := 0 to 3 do                  { если больше не надо двигать вверх }
            if Pieces[piece, i, 0] = 1 then goto exit1;   { переходим к }
                                                  { горизонтальному сдвигу }
        for i := 0 to 3 do                { иначе сдвигаем на клетку вверх }
            for j := 0 to 2 do
                Pieces[piece, i, j] := Pieces[piece, i, j + 1];
        for i := 0 to 3 do
            Pieces[piece, i, 3] := 0;   { нижний ряд заполняем нулями }
    end;

exit1:                                 { аналогично: }
    while true do                          { цикл горизонтального сдвига }
    begin
        for j := 0 to 3 do                 { если больше не надо двигать влево }
            if Pieces[piece, 0, j] = 1 then goto exit2;          { выход }

        for j := 0 to 3 do                { иначе сдвигаем на клетку влево }
            for i := 0 to 2 do
                Pieces[piece, i, j] := Pieces[piece, i + 1, j];
        for j := 0 to 3 do
            Pieces[piece, 3, j] := 0;  { правый ряд заполняем нулями }
    end;

exit2:          { конец работы }
end;
```

Надеюсь, идея процедуры достаточно ясна. Сначала мы проверяем, есть ли хотя бы одна единица (то есть «строительный блок») в верхней строке матрицы. Если оказывается, что нет, сдвигаем все на одну клетку вверх (нижний ряд заполняется нулями). Если же есть, фигура уже находится в самом верхнем положении. Очередь за сдвигом влево, который производится совершенно аналогично.

Как и в игре «Удав», проще сразу загрузить все используемые картинки («строительные блоки») в объекты типа TBitmap, чтобы не «доставать» их из элемента ImageList в процессе работы.

Опишите глобальный массив Bitmaps и пару процедур LoadBitmaps/FreeBitmaps:

```
Bitmaps : array[0..4] of TBitmap;          { "строительные блоки" }
...
procedure LoadBitmaps;                     { загрузить "строительные блоки" }
var i : Integer;
begin
```

```
    for i := 0 to 4 do
    begin
        Bitmaps[i] := TBitmap.Create;
        Form1.ImageList.GetBitmap(i, Bitmaps[i]);
    end;
end;

procedure FreeBitmaps;                    { освободить память }
var i : Integer;
begin
    for i := 0 to 4 do
        Bitmaps[i].Free;
end;
```

Теперь поговорим об игровом поле. Когда очередная фигура вступает в действие, на поле, как правило, уже что-нибудь да имеется, и это «что-нибудь» сильно влияет на игровой процесс. Отсюда вывод: игровое поле надо хранить в массиве, чтобы иметь возможность анализировать его содержимое в любой момент времени. Поскольку элементы игрового поля — те самые «строительные блоки», а любой из них однозначно задается номером соответствующего ему изображения (помните, есть и блок фонового цвета, который означает отсутствие фигуры), то элементами массива будут целые числа. Отразим этот факт в программе:

```
const FieldHeight = 20; FieldWidth = 10;   { высота и ширина игрового поля }
var Field : array[-1..FieldWidth, 0..FieldHeight] of Integer; { игровое поле }
```

Обратите внимание, что размеры массива превосходят реальные размеры игрового поля. Сделано это для того, чтобы можно было легко отловить выход фигуры за пределы экрана. Используется та же идея, что и в Сокобане: зачем как-то специально отслеживать внештатную ситуацию, если можно просто воспользоваться правилами игры? В Тетрисе мы можем двигать фигуры лишь по свободному, не занятому никакими «строительными блоками» пространству. Следовательно, достаточно пометить участки, находящиеся вне поля, как занятые — и вопрос решен. «Стакан» ограничен с трех сторон: слева, справа и снизу. Поэтому используемый массив на два столбца шире и на одну строку выше, чем настоящее поле. Разумеется, перед началом игры массив Field придется проинициализировать:

```
procedure InitField;                      { инициализация игрового поля }
var i, j : Integer;
begin
    for i := 0 to FieldWidth - 1 do       { заполняем собственно поле }
        for j := 0 to FieldHeight - 1 do  { нулями (соответствующими }
        begin                             { свободному пространству) }
            Field[i, j] := 0;
            Form1.BackBuffer.Canvas.Draw(22*i, 22*j, Bitmaps[0]);  { очистка }
        end;                                                       { экрана }

    for i := 0 to FieldWidth - 1 do     { помечаем участки, находящиеся }
        Field[i, FieldHeight] := 1;     { под "стаканом", как занятые }

    for j := 0 to FieldHeight do        { таким же образом помечаем }
    begin                               { участки слева и справа от поля }
```

```
        Field[-1, j] := 1;
        Field[FieldWidth, j] := 1;
    end;
end;
```

Следующая процедура занимается выводом фигуры на экран:

```
procedure DrawPiece(x, y, piece, colour : Integer); { рисование фигуры }
var i, j : Integer;
begin
  for i := 0 to 3 do
    for j := 0 to 3 do
      if Pieces[piece, i, j] = 1 then
      begin                          { рисуем очередной "строительный блок" }
        Form1.BackBuffer.Canvas.Draw(22*(x + i), 22*(y + j), Bitmaps[colour]);
        Field[x + i, y + j] := colour; { вносим изменения в игровое поле }
      end;
end;
```

Теперь подумаем над такой проблемой. Как уже говорилось, фигуры в игре можно перемещать лишь по свободным участкам поля. Когда играющий пытается сдвинуть очередную фигуру влево или вправо или повернуть ее, мы не всегда должны разрешить такое действие. Если фигура сама (без участия человека) падает в «стакан», ее падение тоже должно рано или поздно прекратиться, и отследить этот момент – наша задача. Поможет нам в этом функция, определяющая, можно или нельзя поместить фигуру в данную клетку игрового поля:

```
function CanPlace(x, y, piece : Integer) : Boolean;   { можно ли поместить }
var i, j : Integer;                                   { фигуру в (x, y)? }
begin
    for i := 0 to 3 do
        for j := 0 to 3 do
            if (Pieces[piece, i, j] = 1) and (Field[x + i, y + j] <> 0) then
            begin
                CanPlace := false;          { участок фигуры накладывается на }
                Exit;                  { занятую клетку поля; запрещаем действие }
            end;
    CanPlace := true;                  { пересечений не обнаружено }
end;
```

Суть Тетриса состоит в сборке линий. Если на игровом поле в какой-то момент времени образовалась хотя бы одна линия, ее надо уничтожить. При этом все содержимое поля выше уничтожаемой линии сдвигается вниз.

Напишем процедуру, которая отслеживает собранные линии и, если надо, сдвигает содержимое поля вниз:

```
Lines : Integer;          { глобальная переменная: количество собранных линий }
...
procedure ShiftField;                  { сдвиг поля (уничтожение линий) }
var i, j : Integer;
    fullrow : Boolean;
    curline : Integer;
begin
    curline := FieldHeight - 1;        { текущая линия (начинаем с нижней) }
    while curline >= 0 do              { идем до самого верха }
    begin
```

```
fullrow := true;                          { определяем, собрана линия }
for i := 0 to FieldWidth - 1 do           { целиком или нет }
    if Field[i, curline] = 0 then
    begin                                 { если поле в текущей строке содержит }
        fullrow := false;                 { хотя бы один нуль, линия не собрана }
        Break;
    end;

if fullrow then                           { если линия собрана }
begin
    Lines := Lines + 1;                   { увеличиваем кол-во собранных линий }

    for i := 0 to FieldWidth - 1 do       { сдвигаем верхнюю часть }
        for j := curline downto 1 do      { поля вниз }
            Field[i, j] := Field[i, j - 1];

    { сдвигаем вниз также изображение на экране }
    Form1.BackBuffer.Canvas.CopyRect(Rect(0, 22, 220, 22*(curline+1)),
            Form1.BackBuffer.Canvas, Rect(0, 0, 220, 22*(curline)));

    for i := 0 to FieldWidth - 1 do       { самая верхняя строка поля }
    begin                                 { теперь пуста }
        Field[i, 0] := 0;                 { заполняем ее нулями }
        { на экране нулям соответствует фоновый цвет }
        Form1.BackBuffer.Canvas.Draw(22*i, 0, Bitmaps[0]);
    end;
            end
            else
                curline := curline - 1;   { если линия не собрана, переходим }
        end;                              { к следующей }
    end;
```

Надеюсь, вы разберетесь с устройством этой процедуры. Вставляя в текст подобный листинг, каждый раз испытываю противоречивые чувства. С одной стороны, вроде бы алгоритм громоздкий, и неплохо пояснить хотя бы некоторые его аспекты. С другой же стороны, прекрасно осознаешь, что ничего хитрого в нем нет; просто программа длинная, и оттого кажется сложной. А описывая тот же алгоритм на русском языке, получаешь еще более громоздкий, но ничуть не более понятный текст. Уж лучше оставить «as is», как говорится... В таких ситуациях могу только порекомендовать, вооружившись ручкой и листком бумаги, вообразить себя компьютером и представить, как будут работать те или иные фрагменты кода.

Вот так незаметно мы и дошли до главной процедуры. Точнее, до участка кода, занимающегося управлением игрового процесса. Перед тем, как запрограммировать основной цикл, давайте подумаем немного об управлении игрой. В этом аспекте Тетрис похож на Сокобан: пока вы удерживаете нажатой ту или иную клавишу, фигура будет двигаться в соответствующем направлении; как только клавиши окажутся отпущенными, движение прекратится. Но есть и существенное отличие: в Сокобане все действия происходят при нажатии какой-либо клавиши; здесь же фигура падает в «стакан» независимо от желания игрока. Поэтому процесс идет «сам собой», человек может лишь его корректировать. На уровне

программного кода это означает, что все действие происходит в некотором цикле. Если где-то внутри цикла система определила нажатие клавиши, она соответствующим образом должна отреагировать на него. Проще всего реализовать такое поведение, программируя процедуры-обработчики событий KeyDown и KeyUp главной формы:

```
var Key_Space, Key_Left, Key_Right, Key_Down : Boolean;  { состояния клавиш }
...
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
    case Key of
        VK_SPACE : Key_Space := true;      { клавиша нажата }
        VK_LEFT  : Key_Left := true;
        VK_RIGHT : Key_Right := true;
        VK_DOWN  : Key_Down := true;
    end;
end;

procedure TForm1.FormKeyUp(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
    case Key of
        VK_SPACE : Key_Space := false;    { клавиша отпущена }
        VK_LEFT  : Key_Left := false;
        VK_RIGHT : Key_Right := false;
        VK_DOWN  : Key_Down := false;
    end;
end;
```

Теперь основная процедура сможет просто проверить значения переменных Key_Space, Key_Left, Key_Right и Key_Down, чтобы убедиться, нажата соответствующая клавиша или нет. Кстати, о назначении клавиш. Курсорные стрелки «влево» и «вправо» двигают фигуру соответственно влево и вправо. Клавиша «вниз» ускоряет падение фигуры, а пробел поворачивает ее на 90 градусов.

Все. Осталось лишь написать главную процедуру. Она не так проста, поэтому псевдокод все-таки привести стоит. Замечу: как и в Удаве, мы будем использовать обработчик события Activate. Запрет последующих вызовов при помощи переменной FirstRun тоже остается в силе.

```
ЦИКЛ пока пользователь не выбрал Cancel (как в Удаве)
инициализация уровня и переменных
ГЛАВНЫЙ ЦИКЛ ИГРЫ
    ЕСЛИ на экране нет движущейся фигуры
        генерируем новую фигуру
        ЕСЛИ ее не удается разместить, конец игры
    стираем текущую фигуру с экрана
    ЕСЛИ нажата клавиша «вниз», увеличиваем скорость игры
    ЕСЛИ нажата клавиша «влево», и можно сдвинуть фигуру влево, сдвигаем
    ЕСЛИ нажата клавиша «вправо», и можно сдвинуть фигуру вправо, сдвигаем
    ЕСЛИ нажат пробел
        поворачиваем фигуру на 90 градусов
        ЕСЛИ ее нельзя разместить на экране
            возвращаем фигуру в исходное состояние
```

```
        ЕСЛИ на данной итерации сдвиг фигуры вниз не требуется, рисуем фигуру
        ИНАЧЕ
            ЕСЛИ фигуру можно сдвинуть вниз
                сдвигаем и рисуем
            ИНАЧЕ
                рисуем на текущем месте
                указываем программе, что на экране больше нет движущихся фигур
                сдвигаем игровое поле (уничтожаем собранные линии)
        обновляем содержимое экрана
        пауза (синхронизация с таймером)
    КОНЕЦ ЦИКЛА
```

Как видите, Тетрис не так прост. Настоятельно рекомендую убедиться, что вы понимаете происходящее, прежде чем читать дальше.

Мне кажется, что единственное совсем не очевидное место в программе — это условие

```
ЕСЛИ на данной итерации сдвиг фигуры вниз не требуется
```

Суть вот в чем. Если вы будете сдвигать текущую фигуру вниз на каждой итерации главного цикла программы, игра окажется совершенно «неиграбельной». Дело даже не в том, что процесс будет слишком быстрым, нет — его как раз-таки затормозить несложно. Проблема в другом: поскольку на каждой итерации пользователь может сдвинуть фигуру лишь на одну клетку влево или вправо, вертикальная скорость (то есть скорость падения) фигуры окажется не меньше горизонтальной. Таким образом, за время движения фигуры у вас остается не так много возможностей для маневров. Анализируя же любую «нормальную» реализацию Тетриса, несложно убедиться, что за время падения можно успеть сделать очень многое (по крайней мере, на начальных уровнях игры).

Решение проблемы может быть, например, таким: будем сдвигать фигуру вниз не на каждой итерации основного цикла, а лишь на каждой *n*-й. Тогда вертикальная составляющая скорости будет в *n* раз меньше горизонтальной.

Теперь рассмотрим готовую процедуру TForm1.FormActivate(), в которой и производятся интересующие нас действия:

```
const Delay = 3;                        { задержка падения фигуры (в кадрах) }
var MSecsPerFrame : Integer;            { скорость работы (миллисекунд на кадр) }
...
procedure TForm1.FormActivate(Sender: TObject);
var oldtime      : TDateTime;
    CurX, CurY   : Integer;                 { координаты текущей фигуры }
    CurPiece     : Integer;                 { ее идентификатор }
    CurColour    : Integer;                 { и цвет }
    v            : Integer;                 { счетчик кадров }
begin
    if not FirstRun then Exit;   { запрет повторного вызова }

    FirstRun := false;
    Randomize;
    LoadBitmaps;             { загрузка "строительных блоков" }
    CurY := -1;              { текущее значение Y-координаты, равное -1 }
                             { служит индикатором отсутствия движущихся фигур }
```

```
v := 0;                  { инициализация переменных }
CurPiece := 0;
CurX := 0;
CurColour := 0;

while ID_CANCEL <> Application.MessageBox('OK - запуск, Cancel - выход',
                                          'Tetris', MB_OKCANCEL) do
begin
    Key_Space := false;      { считаем, что клавиши изначально не нажаты }
    Key_Left := false;
    Key_Right := false;
    Key_Down := false;

    InitField;               { очистка игрового поля }
    Lines := 0;              { пока не собрано ни одной линии }

    while true do            { главный цикл }
    begin
        oldtime := Now;
        if CurY = -1 then     { если на экране нет движущихся фигур }
        begin
            CurY := 0;                    { создаем новую фигуру }
            CurX := FieldWidth div 2;     { в верхней части экрана }
            CurPiece := 1 + Random(7);    { ее тип и цвет выбираются }
            CurColour := 1 + Random(4);   { случайным образом }

            { если ее нельзя разместить, конец игры }
            if not CanPlace(CurX, CurY, CurPiece) then Break;
        end;

        DrawPiece(CurX, CurY, CurPiece, 0);   { стираем фигуру с экрана }

        { если нажата клавиша "вниз", увеличиваем скорость игры }
        if Key_Down then MSecsPerFrame := 20 else MSecsPerFrame := 100;

        if Key_Left and CanPlace(CurX - 1, CurY, CurPiece) then
            CurX := CurX - 1;         { сдвиг фигуры влево }
        if Key_Right and CanPlace(CurX + 1, CurY, CurPiece) then
            CurX := CurX + 1;         { сдвиг фигуры вправо }

        if Key_Space then         { поворот фигуры }
        begin
            Rotate90(CurPiece);           { поворачиваем на 90 градусов }
            ToCorner(CurPiece);
            if not CanPlace(CurX, CurY, CurPiece) then   { если фигуру }
            begin                                        { нельзя разместить }
                Rotate90(CurPiece);       { возвращаем ее }
                Rotate90(CurPiece);       { в первоначальное положение }
                Rotate90(CurPiece);       { (повернуть еще три раза) }
                ToCorner(CurPiece);
            end;
        end;

        v := v + 1;                       { увеличиваем счетчик кадров }

        { если на текущей итерации нет вертикального сдвига }
        if v <> Delay then
          DrawPiece(CurX, CurY, CurPiece, CurColour)  { рисуем фигуру }
        else                                          { иначе }
```

```
begin
    v := 0;                                          { обнуляем счетчик }
    if CanPlace(CurX, CurY + 1, CurPiece) then
    begin
        CurY := CurY + 1,          { если фигуру можно разместить }
        DrawPiece(CurX, CurY, CurPiece, CurColour);  { размещаем }
    end
    else                                                { иначе }
    begin                                 { оставляем на прежнем месте }
        DrawPiece(CurX, CurY, CurPiece, CurColour);
        ShiftField;                       { уничтожаем собранные линии }
            CurY := -1;       { на экране больше нет движущихся фигур }
        end;
    end;

    { обновляем содержимое экрана (копируем буфер на экран) }
    Form1.Screen.Canvas.CopyRect(Rect(0, 0, 220, 440),
                Form1 BackBuffer.Canvas, Rect(0, 0, 220, 440));

    { обновляем индикатор количества собранных линий }
    Form1.LinesLabel.Caption := IntToStr(Lines);

    Application.ProcessMessages;     { обработка событий }
    while Round(MSecsPerFrame - (Now - oldtime) * MSecsPerDay) > 0 do
        Application.ProcessMessages;  { синхронизация с таймером }
  end;
end;

    FreeBitmaps;                          { освобождение памяти }
    Application.Terminate;                { конец работы }
  end;
```

Внешний вид готовой программы показан на рис. 7.15.

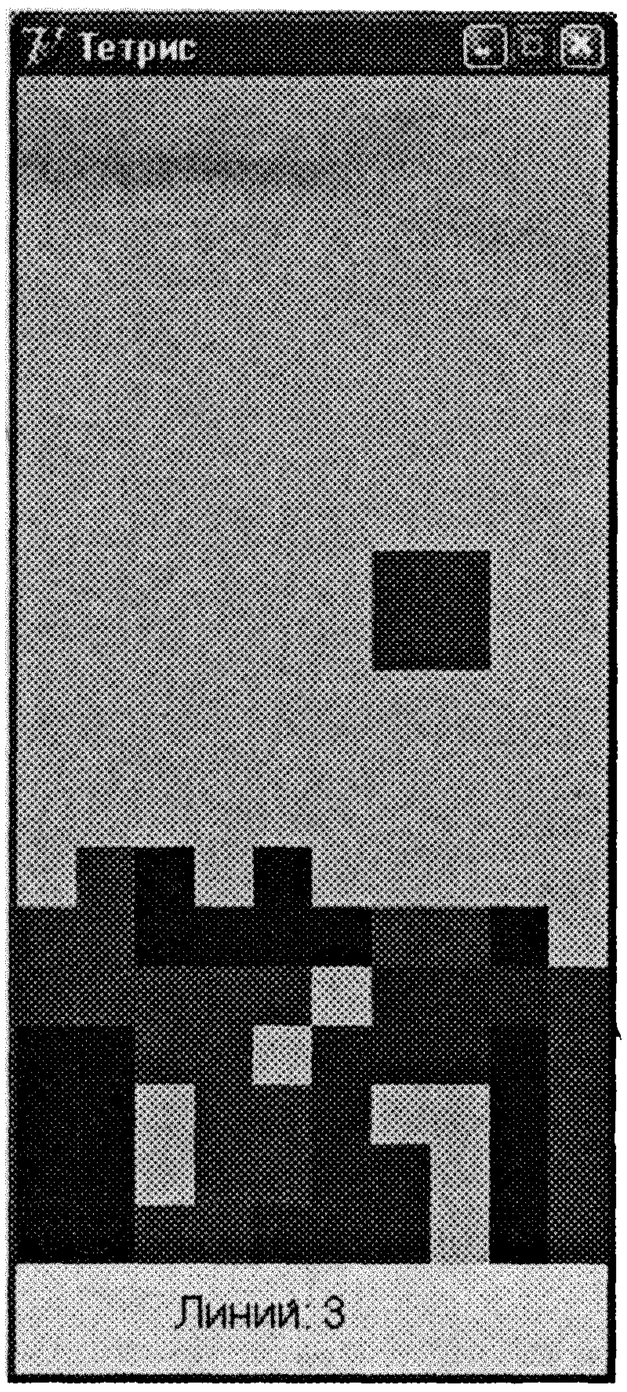


Рис. 7.15. World Classics Series: Tetris

Проекты для самосовершенствования

1. Добавьте в нашу реализацию Сапера возможность установки флажков и автоматическое раскрытие нулевых клеток.
2. Дополните Сапер другими стандартными возможностями: выбором уровня сложности (размера игрового поля), измерением прошедшего с начала игры времени и таблицей почета.
3. Напишите усложненный вариант Сокобана, в котором каждому ящику (камню) соответствует свое собственное место. Иными словами, имеет значение, какой ящик куда толкать.
4. (По-настоящему творческое задание!) Попробуйте написать программу, которая автоматически решает уровни Сокобана. Подумайте, как представить задачу в виде поиска на графе, постарайтесь минимизировать размер графа, разработайте хорошую эвристическую функцию и примените процедуру поиска A* (или IDA*).
5. В Удаве мы применили одну хитрость: все части тела змея специально были нарисованы симметричными, чтобы не думать о том, в какую сторону он ползет. Нарисуйте «нормальные», несимметричные голову и хвост. У вас получится четыре разных картинки для каждой части тела, соответствующие разным направлениям движения удава. Внесите необходимые изменения в программу.

Рис. 7.16. Columns 3 (SEGA Genesis/Mega Drive II)

6. Сделайте реализацию Тетриса более профессиональной. Во-первых, после уничтожения определенного количества линий должен произойти переход на следующий уровень (при этом скорость игры возрастает). Во-вторых, отдельно от количества собранных линий ведите учет очков. За одну собранную линию дается одно очко, за две линии, собранные «за один присест», — три очка, за три — пять очков, за четыре — семь очков.
7. Напишите игру Columns, родственную Тетрису (рис. 7.16). В «стакан» падают фигуры, состоящие из трех сегментов разных цветов. Игрок может двигать фигуру влево и вправо, а также «прокручивать» цвета по вертикали (нижний цвет оказывается сверху, средний — снизу, а верхний — в середине). Когда очередная фигура оказывается установленной, проводится проверка на существование вертикальных, горизонтальных или диагональных линий из трех или более элементов одного и того же цвета. Найденные линии уничтожаются. «Строительные блоки» не могут висеть в воздухе и поэтому падают, если под ними ничего нет (такие ситуации возникают при уничтожении собранных линий).

Послесловие

Времена меняются. Еще лет десять назад раздобыть хорошую (и даже не очень хорошую) книгу по программированию было весьма проблематично даже в крупных городах, а уж в провинции просто невбзможно. Помню, как счастлив был я, когда удалось достать давным-давно устаревшее издание книги Кернигана и Ричи «Язык программирования С» на пятидюймовой дискете (помните такие?). Сейчас, по крайней мере в Москве и Петербурге, есть очень хорошие книжные магазины с большим выбором компьютерной литературы, а если даже в вашем городе с книгами проблемы, вы всегда можете воспользоваться Интернетом, чтобы заказать литературу почтой. На полках лежат десятки книг по MS Office и ОС Windows; почти такой же широкий выбор наблюдается среди самоучителей популярных языков программирования вроде C++, Visual Basic или Java; к вашим услугам огромное количество руководств по самым различным технологиям и программным продуктам. Казалось бы, книг — море, чего же еще желать? И все-таки меня не покидает чувство, что авторы обходят стороной очень важные, интересные вопросы. Трудно так сразу сказать, какие именно; пока сам с ними не столкнешься — не поймешь. Не думаю, что пользователи в 80-е годы ждали чего-то наподобие Windows, зато когда эта операционная система появилась, все сразу же поняли: вот — именно то, чего мы подсознательно хотели. Порою попадается на глаза какая-нибудь новая книга, и тут же спрашиваешь себя: почему же раньше никто об этом не писал? Не так давно с большим удовольствием читал очень интересную американскую монографию по искусственному интеллекту. А почему на русском языке ничего подобного нет (по крайней мере, я не видел; книги 60-х годов не предлагать)? Неужели в нашей стране никому это не интересно? Впрочем, не все так плохо. Спасибо, к примеру, издательству «Питер» за серию «Классика Computer Science». Наконец-то на русском языке стали появляться книги, уже по четыре раза изданные на английском... А вывод такой: несмотря на обилие самой разнообразной литературы есть, есть еще благодатная почва для новых начинаний.

Признаюсь, мною двигало честолюбивое желание написать книгу, непохожую на другие, пусть даже и не очень серьезную. Получилось или нет — судить вам. Отзывы, критика, пожелания приветствуются (мой электронный адрес — maxim_mozgovoy@hotbox.ru). В любом случае, если вы решите, что провели время с пользой и получили удовольствие, я уже буду считать свой труд не напрасным; если же после прочтения книги вы захотите узнать побольше об архивации или, скажем, трехмерной графике, моя миссия окажется полностью выполненной. Удачи в делах!