ВАЛЕРИЙ РУБАНЦЕВ

ЗАНИМАТЕЛЬНЫЕ УРОКИ С КОМПЬЮТЕРОМ

43 урока
80 проектов на новейшем языке программирования Microsoft Small Basic

Тыблоки
Тьюрмиты
Тараканьи бега
Дефектный логопеф
Полярная Черепашка
Фрактальные снежинки
Осьминожьи гадания
Рекурсивные сказки
Кролики Фибоначчи
Галилеевы ядра
Дрим тим

Валерий Рубанцев

ЗАНИМАТЕЛЬНЫЕ УРОКИ С КОМПЬЮТЕРОМ, или

Small Basic для начинающих

RVGAMES.DE 2013

Бесплатное издание

От автора

Учиться надо весело, учиться надо весело,

Учиться будем весело, чтоб хорошо учиться.

Детская песенка

Прошло немало лет с тех пор как компьютеры появились в школе и дома. В школе вы изучаете с их помощью информатику (что полезно), а дома он служит вам партнёром для игр (что приятно). А *цель* этой книги - показать, как можно сочетать приятное с полезным, то есть использовать компьютер при изучении школьной программы. В этой книге вы найдёте несколько десятков уроков по разным школьным предметам. Но уроки эти не простые, а *занимательные*! Поэтому на каждом уроке мы будем писать интересные компьютерные программы.

На этих уроках вы **узнаете**:

- как гениальный немецкий художник Альбрехт Дюрер составил знаменитый *магический квадрат* и почему он поместил его на своей гравюре *Меланхолия*;
- о черепахе, на панцире которой был нарисован первый в мире магический квадрат *Ло-шу*;
- и о другой *Черепашке* - она умеет бегать по экрану, оставляя за собой причудливый след в виде замысловатых фигур;
- о *литорее* обычной и мудрой;
- как средневековый математик *Фибоначчи* разводил кроликов, и что из этого вышло;
- что такое *тыблоко*;
- как просеивать числа через *решето Эратосфена*;
- чем занимается *высшая арифметика* и *комбинаторика*;
- о секретах *транслитерации*.

И **научитесь**:

- угадывать результаты футбольных матчей не хуже осьминога Пауля;
- *рисовать* пикселями, линиями, прямоугольниками и эллипсами красивые узоры;
- дрессировать курьёзных компьютерных зверушек - неугомонного *тьюрмита* и фрактальную *Киберчерепашку*;
- расшифровывать *тарабарскую грамоту*;
- устраивать *тараканьи бега*;
- сбрасывать тяжёлые ядра с Пизанской башни;
- писать *консольные* и *графические* приложения в среде *Смолл Бейсик* (это новый «хит» фирмы *Майкрософт*, который специально создан для изучения основных конструкций и приёмов программирования);
- *импортировать* из Интернета интересные программы и *публиковать* свои собственные проекты, которые будут доступны во всём мире;
- решать шахматную *задачу Гаусса* за пару минут;
- составлять огромные *магические квадраты*;
- говорить на *логопефическом языке*;
- играть в *Супернаборщика* и в *Города*;
- отыскивать *палиндромы*, *факториалы*, *простые* числа;
- подсчитывать предметы с первого взгляда…

В общем, всего и не перечесть! А теперь – звенит звонок, пора на первый занимательный урок!

Валерий Рубанцев

Условные обозначения, принятые в книге:

Дополнение или замечание

Ненавязчивое требование или указание

Домашнее задание для самостоятельного решения

Папка с исходным кодом **программы**

Исходные коды программ, выполняемые файлы и установочный файл Small Basic 1.0 находятся в папке ***_Small Basic Projects***.

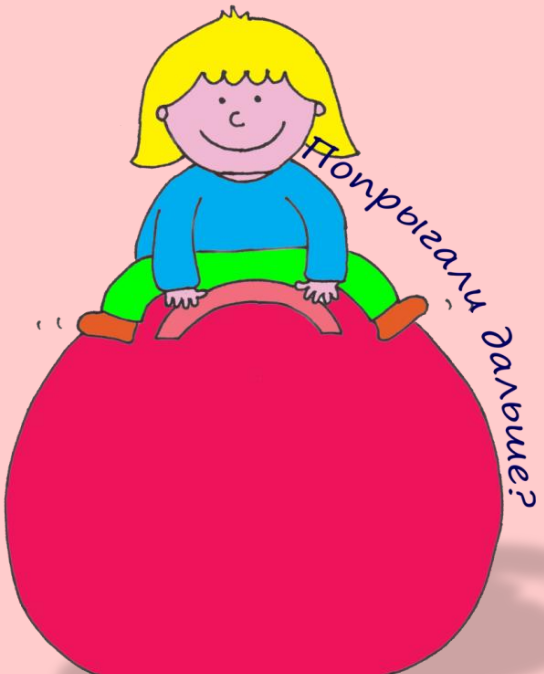
Официальный сайт **книги:** ***www.RVGames.de***

Оглавление

Урок 1. Устанавливаем *Small Basic*

Железный конь идет на смену
крестьянской лошадке!
Остап Бендер

В современной школе невозможно обойтись без компьютера. Великий комбинатор Остап Бендер сказал бы сейчас: *Компьютер – не роскошь, а средство передвижения по дорогам жизни!* А вот чтобы управлять компьютером, нужно учиться программированию. В самом деле, современный компьютер не должен быть только интерактивным телевизором, средством для общения в чатах и на форумах или игровой приставкой. Это скорее инструмент для воплощения творческих замыслов. С его помощью можно сочинять музыку, писать книги, рисовать картины, обрабатывать фотографии и редактировать фильмы. Эх, да что там говорить: компьютер может почти всё! А пользователю остаётся только освоить нужные программы и иметь творческую натуру, ищущую самовыражения. И это очень занятно, но… Это всё *чужие* программы, а ведь куда интереснее написать свою собственную - которая умеет делать то, чего никакая другая сделать не сможет! Правда, для этого придётся овладеть языком программирования, без которого объяснить компьютеру свои замыслы не удастся.



Начинаем!

Лёд тронулся,

господа присяжные заседатели!

Остап Бендер

В мире существует не одна сотня языков программирования - на все случае жизни. Большинство из них очень сложны для начинающих и могут напрочь отбить желание учиться программированию. Но есть языки, специально предназначенные для обучения. В первую очередь, это *паскаль*, созданный в 1969 году Никлаусом Виртом, и *бейсик,* разработанный несколькими годами раньше профессорами Дартмутского колледжа Томасом Куртом и Джоном Кемени.

Язык *паскаль* получил своё название в честь выдающегося французского математика и физика *Блеза Паскаля* (1623 – 1662), а название языка *бейсик* представляет собой сокращение его английского названия *Beginner's All-purpose Symbolic Instruction Code* (*универсальный код символических инструкций для начинающих*) – *BASIC*. Несколько неуклюжее название языка объясняется желанием авторов придать ему глубокий смысл: по-английски слово *basic* значит *основной*. Поскольку эти языки программирования действительно являются основными для многих любителей программирования, то их названия стали нарицательными и пишутся с маленькой буквы.

Оба языка до сих пор широко используются в программировании и имеют множество реализаций для всех операционных систем и процессоров. В качестве языка программирования для этой книги выбран **Small Basic** («*Смолл Бейсик*» - «*маленький бейсик*»), специально созданный фирмой *Майкрософт* для обучения школьников. Скоро вы и сами убедитесь, что выучить его можно за несколько уроков. Но, несмотря на свою простоту, бейсик позволяет писать как *полезные* приложения по школьной программе, так и *занимательные* игры с красивой графикой.

Дальше мы будем называть *Small Basic* просто *SB* или *СБ*.

Поскольку эта книга рассказывает об использовании компьютера на *разных* уроках, а не только на уроках программирования, то мы не успеем изучить *все* возможности языка и написать БОЛЬШИЕ программы, но скучно на уроках вам точно не будет!

Чтобы научиться программировать, нужно программировать! А для этого нам потребуются:

1. Сам **компьютер** - без него нельзя!

2. На компьютере должна быть установлена операционная система **Windows XP** (здесь возможны ошибки при выполнении программ), **Windows Vista, Windows 7** или **Windows 8**.

3. Подключение к **Интернету**.

4. **.NET Framework 3.5**, **4.0** или **4.5**.

5. Программа **Microsoft Small Basic**.

6. Знание **английского языка** не обязательно, но несколько английских слов выучить необходимо!

Будем считать, что первые три пункта этого списка уже выполнены. А если нет? - А если нет, вам придётся самоотверженным трудом и примерным поведением - с посильной помощью родителей, конечно, - эти три пункта непременно выполнить. Потому что без них программировать нельзя, а ведь программирование - это самый занимательный школьный предмет!

Начнём же мы наш первый урок с того, что установим *Small Basic* на компьютер. Но прежде его нужно *скачать*. Вот для этого нам просто необходим *Интернет*!

Установочный файл *SmallBasic.msi* можно скачать с официального сайта программы, то есть *www.smallbasic.com*. На главной странице сайта найдите в правом в верхнем углу кнопку *Download* и нажмите её (Рис. 1.1).

Рис. 1.1. Главная страница сайта *www.smallbasic.com*

Появится диалоговое окно (Рис. 1.2).

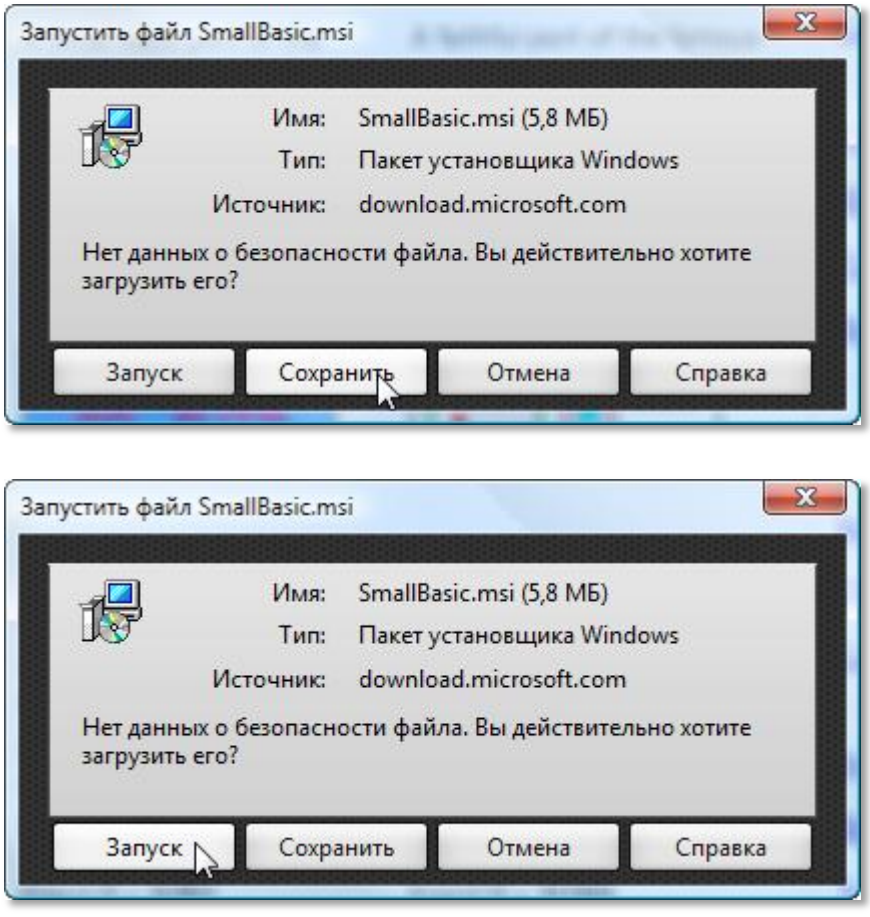


Рис. 1.2. Выбираем способ установки

Если вы нажмёте кнопку *Сохранить*, то вам придётся выбрать папку для сохранения установочного файла, а когда файл окажется на вашем компьютере (ждать придётся недолго, потому что он совсем небольшой), смело дважды кликните по нему мышкой!

Но вы можете *сразу* установить *Small Basic* на свой компьютер, если предпочтёте кнопку *Запуск*.

И в том, и в другом случае вы увидите новое диалоговое окно (Рис. 1.3).

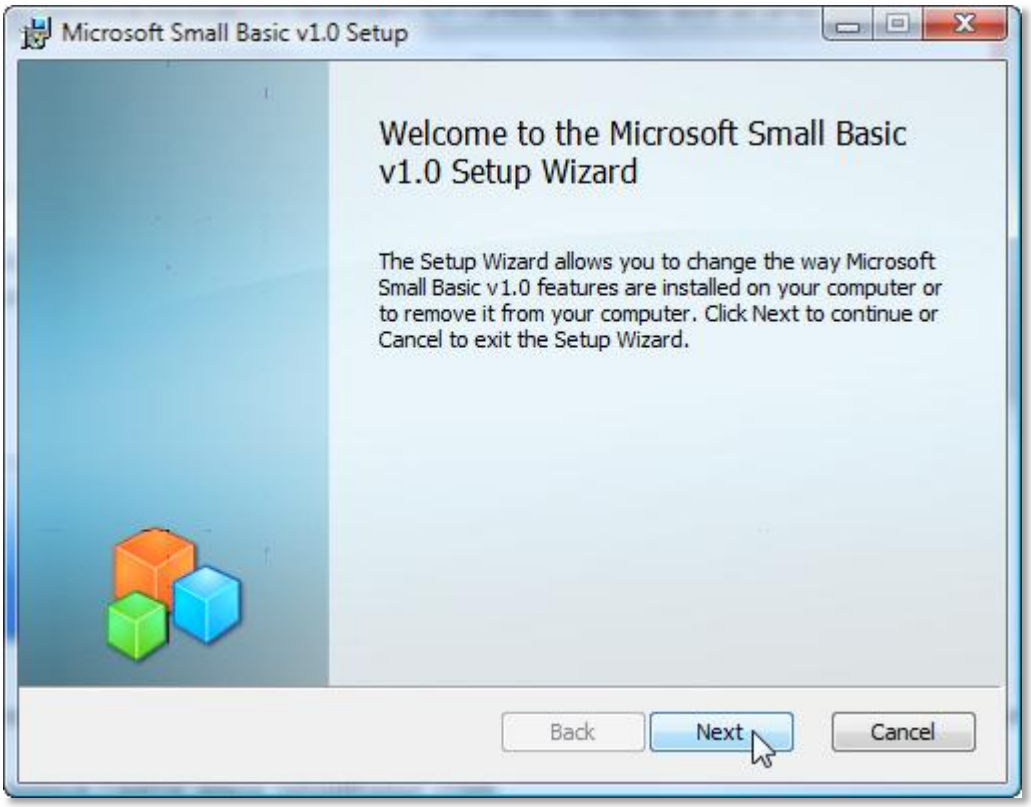


Рис. 1.3. Диалоговое окно: вас приветствует установщик *СБ*

Не мудрствуя лукаво, просто нажмите кнопку **Next** (*Дальше*). Установка продолжится, и на экране появится следующее диалоговое окно (Рис. 1.4).

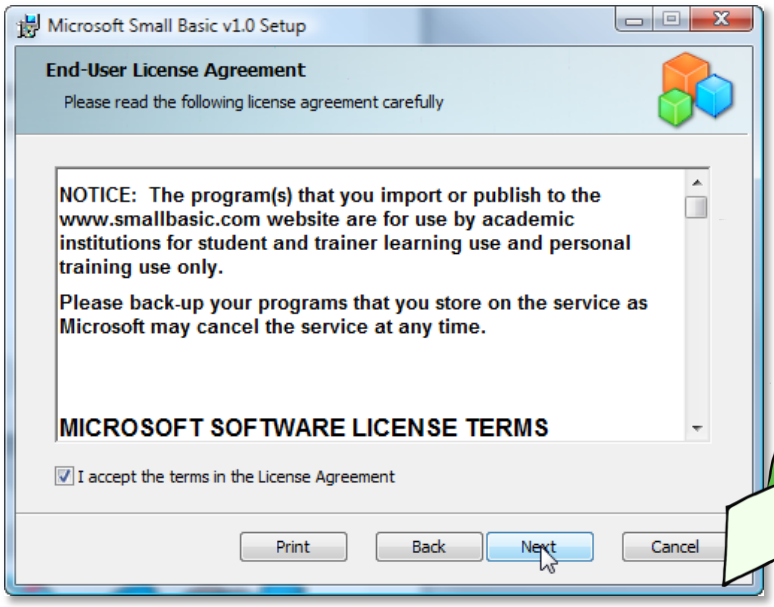


Рис. 1.4. Диалоговое окно лицензирования *СБ*

Вас не должно удивлять такое обилие разных окон - ведь и название самой операционной системы *Windows* переводится как *Окна*.

Снова кликаем по кнопке **Next** и получаем новое диалоговое окно, в котором можно выбрать *язык интерфейса* программы (Рис. 1.5).

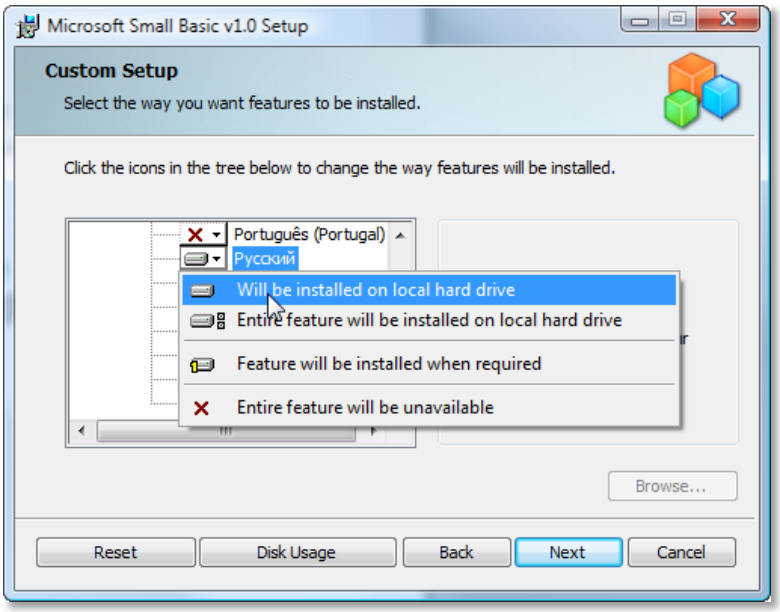


Рис. 1.5. Выбираем *русский* язык

Если на вашем компьютере стоит *русифицированная* операционная система, то ничего в этом окне вам не делать не нужно.

В следующем окне нажмите кнопку **Install** (*Установить*) (Рис. 1.6).

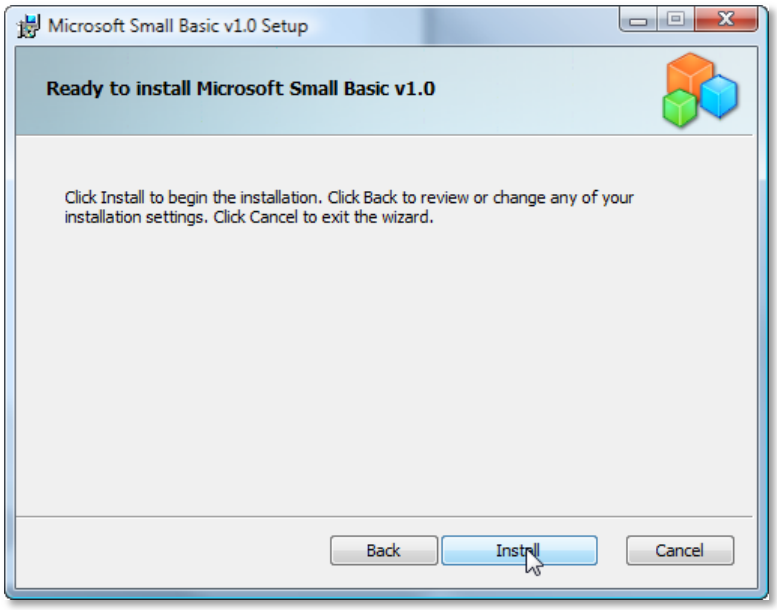


Рис. 1.6. Предварительная подготовка завершена

Процесс установки показывает **зелёная** полоска (Рис. 1.7).

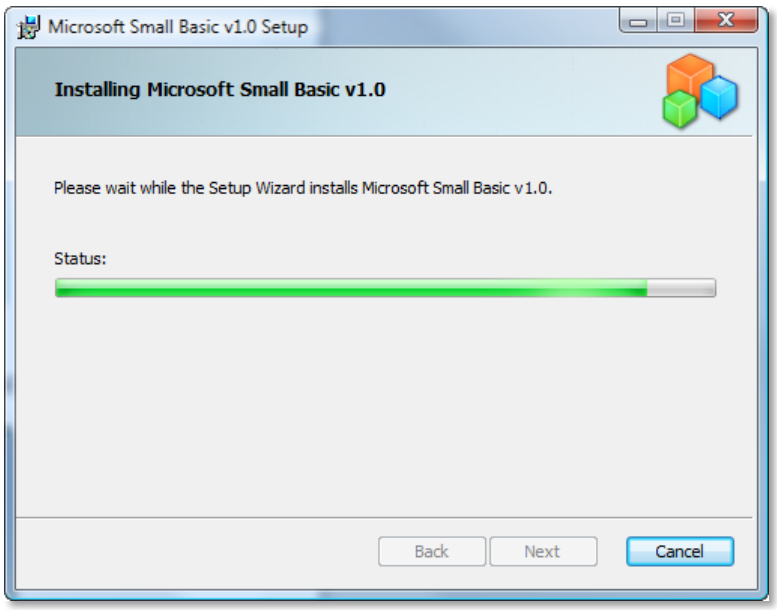


Рис. 1.7. *СБ* устанавливается на компьютер

Готово! Прошло всего несколько минут - и вы уже счастливые обладатели замечательной программы!

Осталось закрыть установщик программы, кликнув по кнопке **Finish** (*Закончить*) (Рис. 1.8).

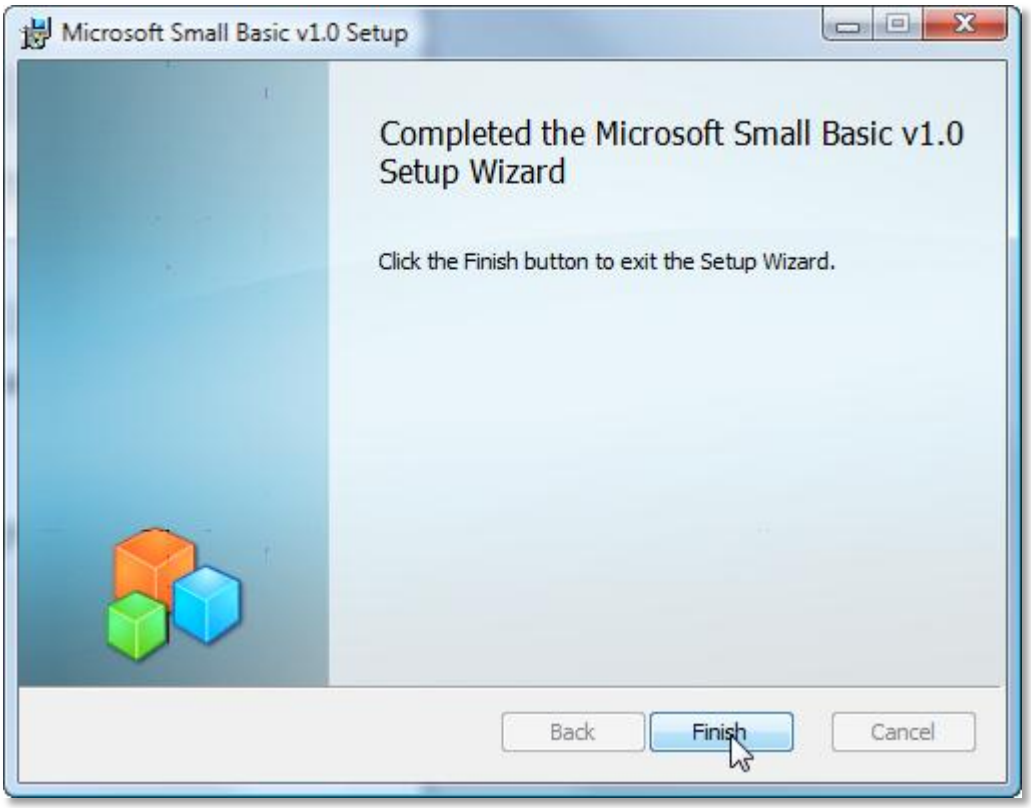


Рис. 1.8. Установка *СБ* закончена

Бейсик находится в папке *«C:\Program Files\Microsoft\Small Basic\»* (если у вас установлена английская версия *Windows)*. Запомните папку с программой, она вам ещё может пригодиться!

Вы можете открыть установочную папку и запустить файл *SB.exe*, дважды щёлкнув по его значку, но лучше один раз найти этот файл в *Проводнике Windows* и создать *ярлык* для программы (Рис. 1.9).

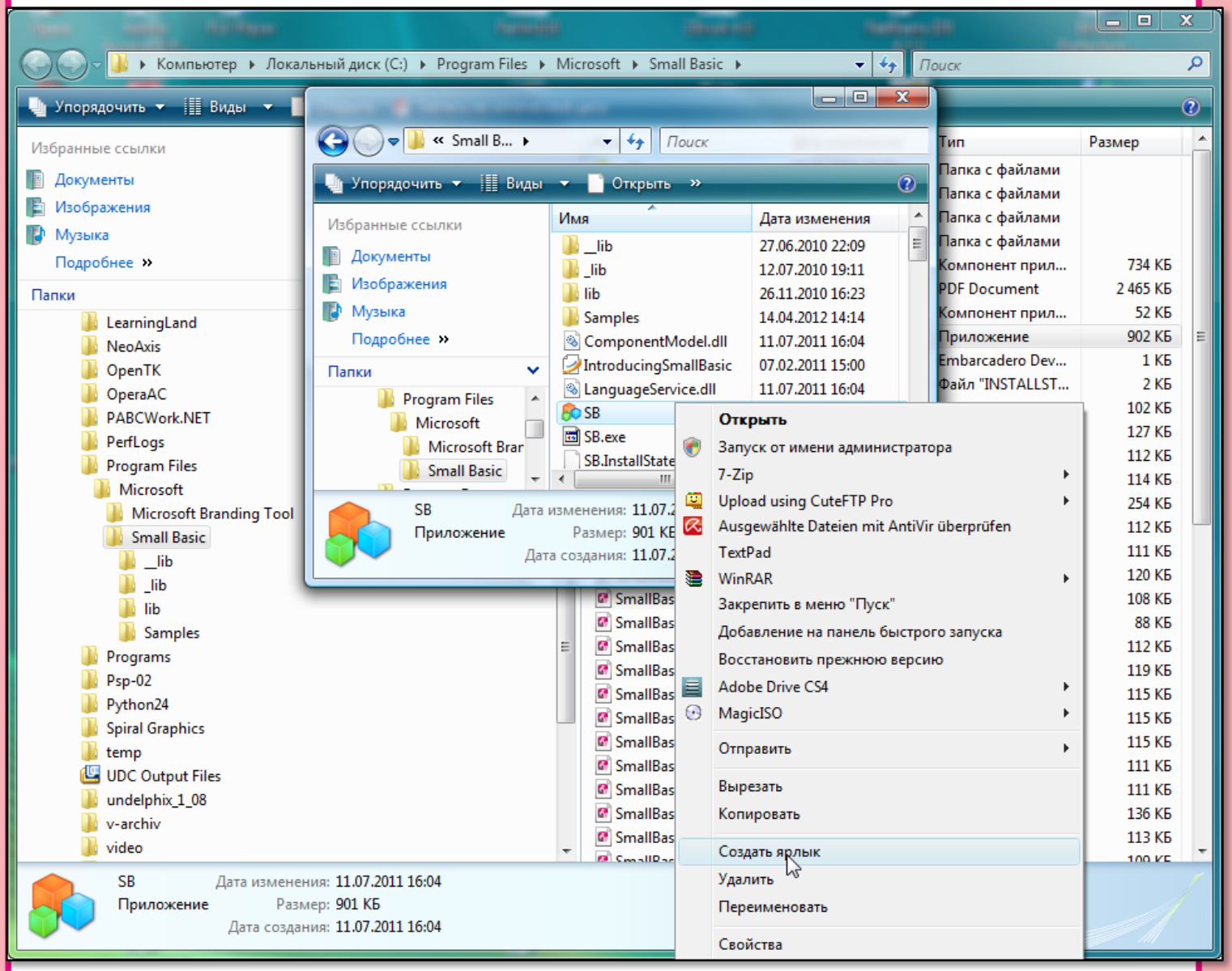


Рис. 1.9. Создаём ярлык

Он будет удобно лежать на *Рабочем столе*, и вам не придётся каждый раз выискивать его в папках (Рис. 1.10).

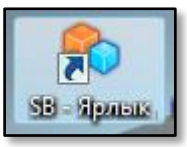


Рис. 1.10. Стандартный ярлык

Конечно, надпись на ярлыке лучше изменить (Рис. 1.11).

Рис. 1.11. Исправленный ярлык

Теперь всё понятно: программа называется *Small Basic*, а версия – *1.0*.

И вот настал торжественный момент: дважды кликаем по ярлыку и в первый раз запускаем бейсик...

Время этого урока закончилось, с бейсиком мы будем знакомиться на следующем уроке.

1. Обязательно найдите папку, в которой был установлен *Small Basic*, и ознакомьтесь с её содержимым.

2. Запустите бейсик из папки, дважды щёлкнув по названию программы.

3. Закройте бейсик, нажав кнопку с крестиком в правом верхнем углу его окна.

4. Запустите бейсик с *Рабочего стола*.

Урок 2. Запускаем *Small Basic*

Small Basic, да дорог!

Программистская пословица

Сначала мы увидим фирменную заставку программы (Рис. 2.1).

Рис. 2.1. Заставка *СБ*

Но скоро наш маленький бейсик предстанет перед нашими очами в полном великолепии (Рис. 2.2).

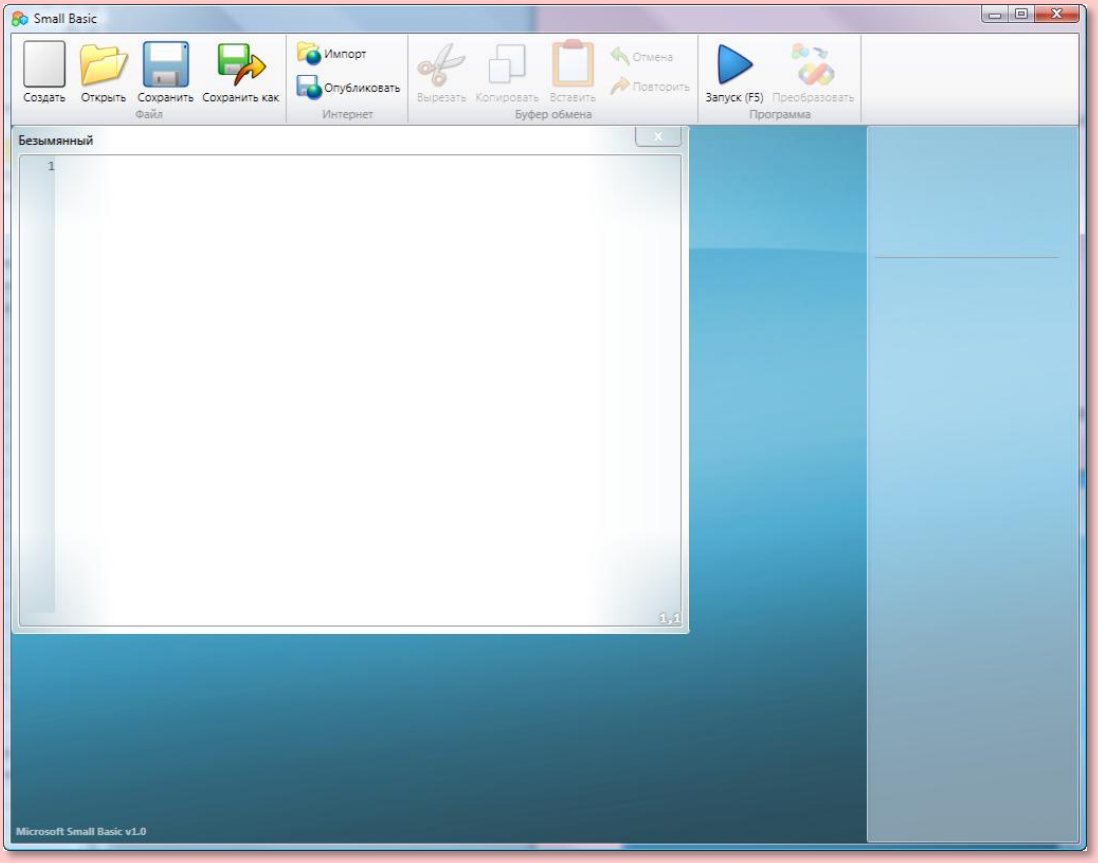


Рис. 2.2. Главное окно *СБ*

И что особенно приятно: все надписи на *русском языке*!

Освоившись среди красивых кнопок, вы увидите пустой *документ* (или *проект*) под названием *Безымянный*.

В папке с бейсиком вы найдёте папку *Samples* с небольшими программами. Вы можете сразу же загрузить их, но мы поступим более радикально и усилим нашу радость от первого знакомства, ведь так хочется сразу же, немедленно посмотреть, что умеет делать этот бейсик.

Если вы всё-таки захотите посмотреть демонстрационные программы (Рис. 2.3) и получите вот такое нелюбезное сообщение (Рис. 2.4),

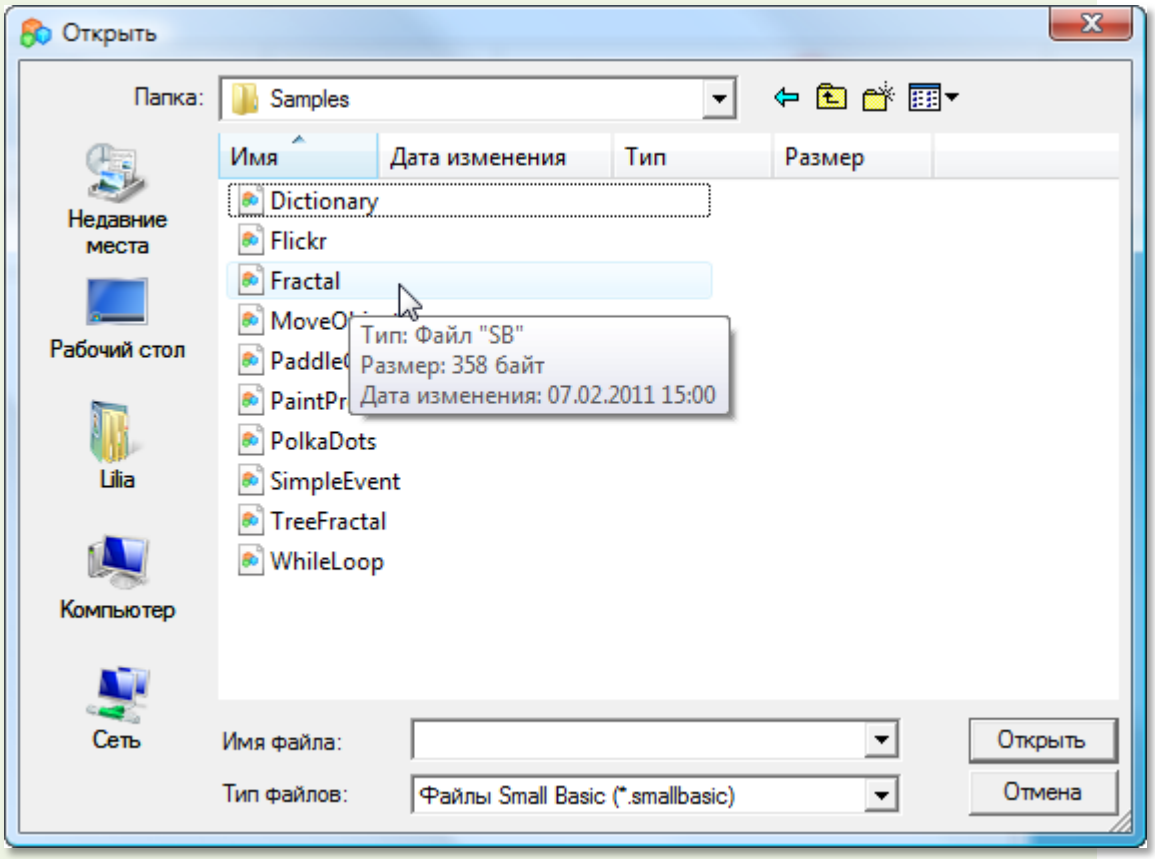


Рис. 2.3. Папка с демонстрационными программами

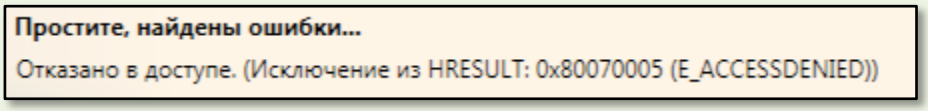


Рис. 2.4. Ошибка вышла!

запустите *СБ* от имени *администратора* и снова загрузите программу (Рис. 2.5).

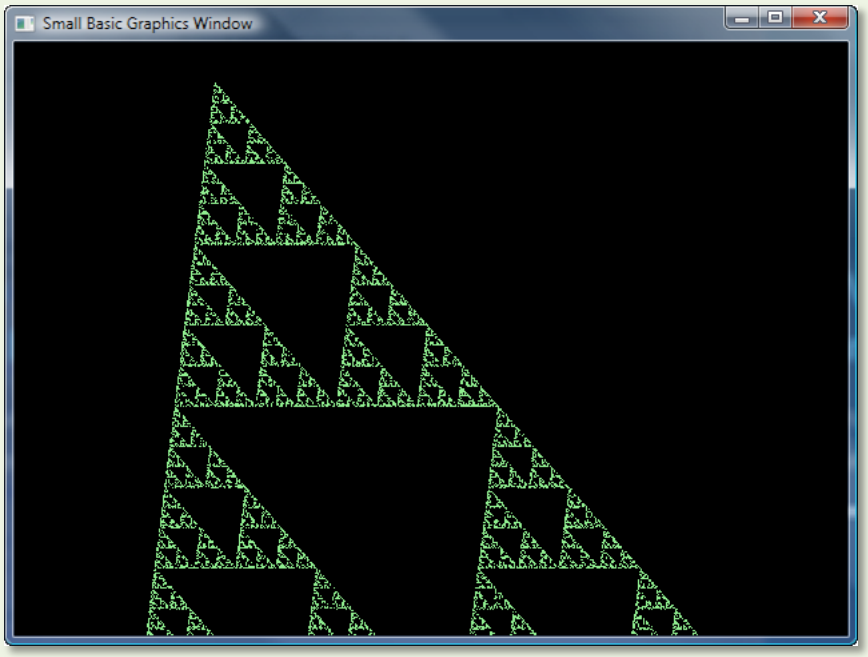


Рис. 2.5. Демонстрационный фрактал

Свою первую программу мы напишем на следующем уроке, а пока давайте посмотрим, как работают чужие программы. И тут нам пригодится кнопка с «импортным» названием (Рис. 2.6).

Рис. 2.6. Кнопка *Импорт*

Нажав её, мы получим в ответ - ещё одно диалоговое окно (Рис. 2.7).

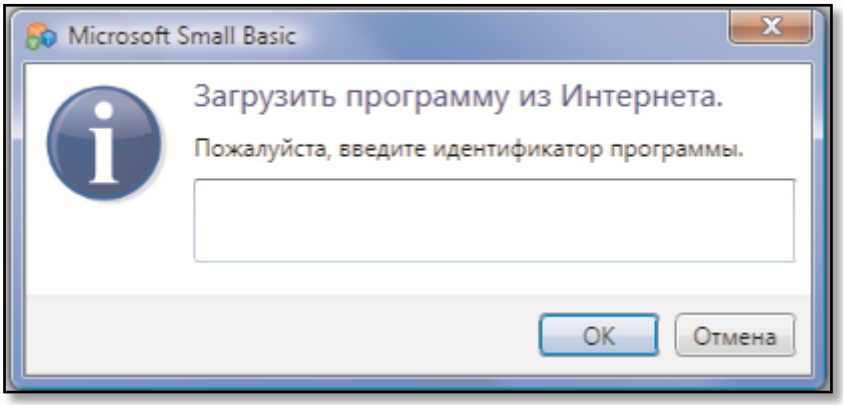


Рис. 2.7. *СБ* требуется информация

Оно настойчиво просит ввести *идентификатор* (или, попросту говоря, *код*) нужной нам программы. Вы, конечно, в полном недоумении: что нужно бейсику? - А всё очень просто: на сайте *www.smallbasic.com* лежат и ждут своего часа разные программы, которые нужно как-то отличать друг от друга. Для этого и служит идентификатор. Это как бы пароль, зная который можно загрузить в бейсик нужную программу. Особенно интересные программы хранятся под своим *именем*, а все остальные - под *кодовым названием*, состоящим их букв и цифр.

Для первого знакомства с бейсиком нам вполне сгодится замечательная игрушка *Snakebite* (или *Змейка*). Набираем её код и нажимаем кнопку *ОК* (Рис. 2.8).

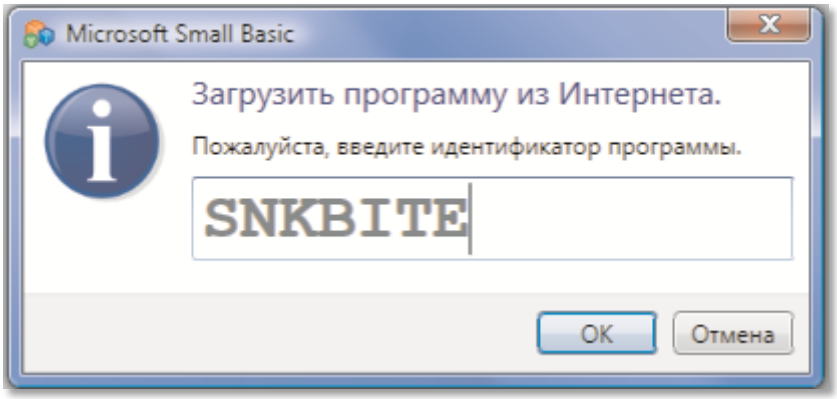


Рис. 2.8. Вводим информацию с клавиатуры

Так как программа будет скачана (*импортирована*), с интернет-сайта, то вам предварительно следует позаботиться о подключении своего компьютера к Интернету!

Через некоторое время наряду с документом *Безымянный* откроется еще один - *SNKBITE - Импортировано*» (Рис. 2.9).

Первое слово - это *название* программы, а второе говорит о том, что ее текст был скачан из Интернета.

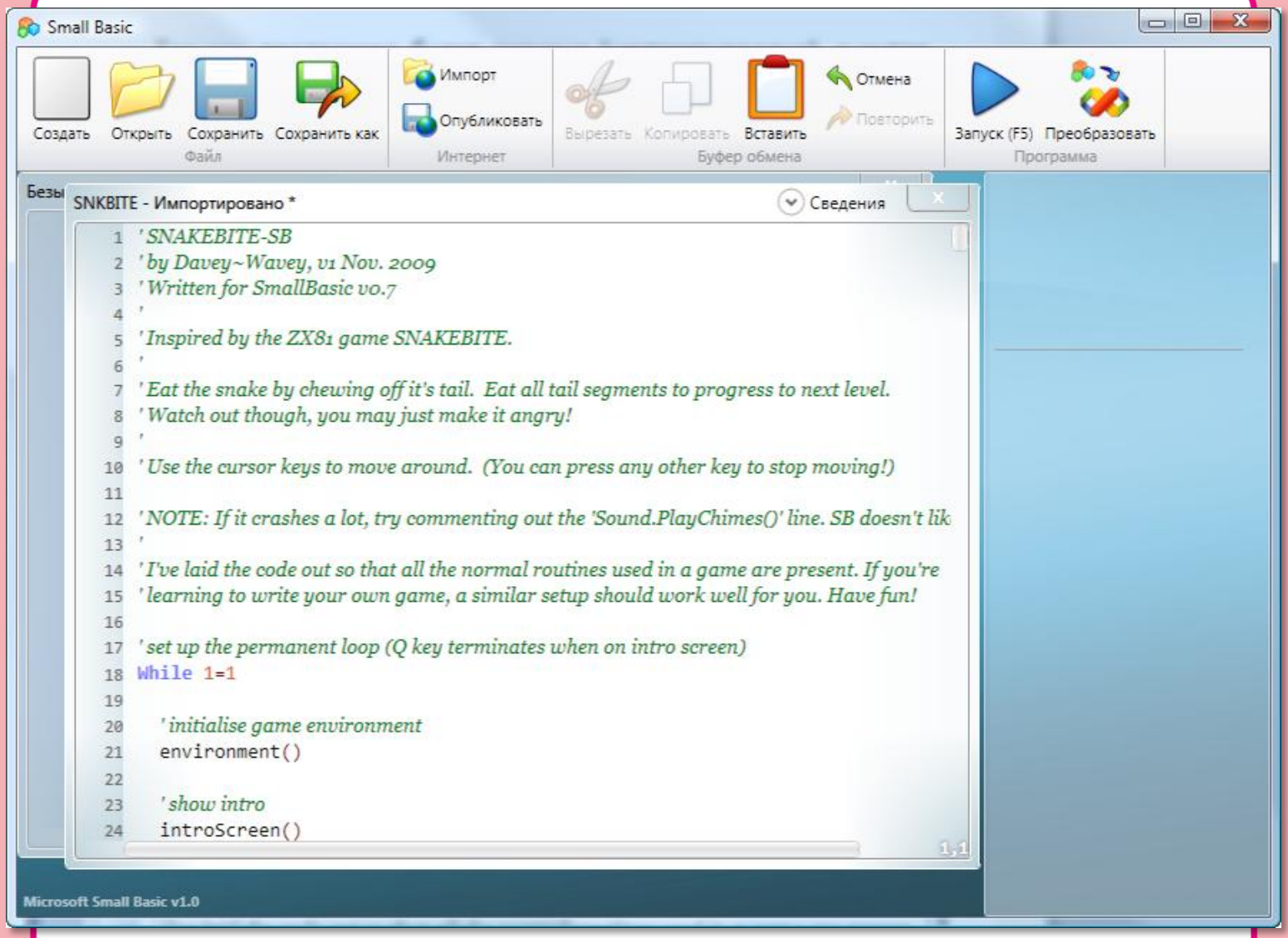


Рис. 2.9. Текст программы загружен в *СБ*

Было бы правильнее сразу же сохранить программу на диске, но мы просто *запустим* её. Для этого нажмите кнопку *Запуск* или клавишу *F5* (Рис. 2.10).

Рис. 2.10. Кнопка *Запуск*

Сначала вы услышите бодрую музыку, а затем появится и заставка игры (Рис. 2.11).

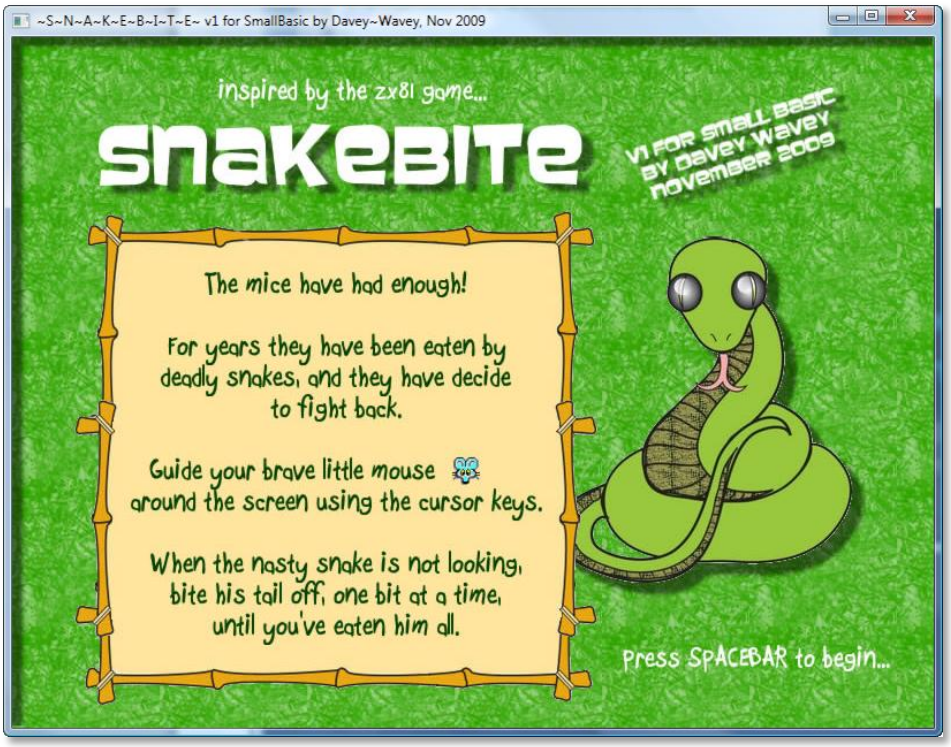


Рис. 2.11. Змейка приветствует игрока

Тут уже трудно удержаться от искушения - да и не надо лишний раз себя терзать! - и не нажать на клавишу *ПРОБЕЛ*, как и просит нас эта самая змейка.

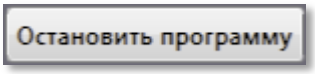
И вот тут уж начнется настоящая потеха (Рис. 2.12).

Рис. 2.12. Игра началась!

Змейка, больше похожая на червяка, поползёт вниз и съест мышку. А мышка - это вы! Признайтесь: вы ведь подумали, что будете управлять змейкой и ловить беззащитных мышек? Конечно, поедать других приятнее и безопаснее, чем самому спасаться от шустрых врагов, но в игре у вас ещё несколько жизней в запасе, так что можно набраться опыта и жить долго и счастливо.

Тем более что мышка при надлежащей изворотливости вполне может постоять за себя и изрядно потрепать змейку. Для этого нужно укусить змейку за самый хвост и откусить его. Так вы получите 10 очков. Сама мышка управляется вовсе не мышкой, как можно было бы подумать, а курсорными клавишами *ВВЕРХ-ВНИЗ-ВЛЕВО-ВПРАВО* - на них нарисованы стрелочки на все четыре стороны. Приостановить разогнавшуюся мышку можно, если нажать любую другую клавишу с буквой или цифрой. Вот и вся премудрость этой нехитрой игры.

Чтобы закончить игру (*закрыть программу*), достаточно нажать на кнопку с крестиком в правом верхнем углу окна программы, то есть точно так же, как и при работе с любым другим приложением *Windows*.

При выполнении программы вид окна *СБ* изменится: оно станет *тёмным*, в центре будет написано название (*имя*) запущенной программы и появится дополнительная кнопка 

, нажав на которую, вы получите именно то, что на ней написано (Рис. 2.13).

Поскольку программа загружает данные из Интернета, то компьютер во время работы программы должен быть подключён к нему!

Игра прекрасная и оформлена со вкусом! Вот такие игры вы сможете сами делать с помощью *СБ*, но, конечно, не сразу: посмотрите на текст программы (*листинг*) – он довольно длинный и наверняка совершенно вам непонятный.

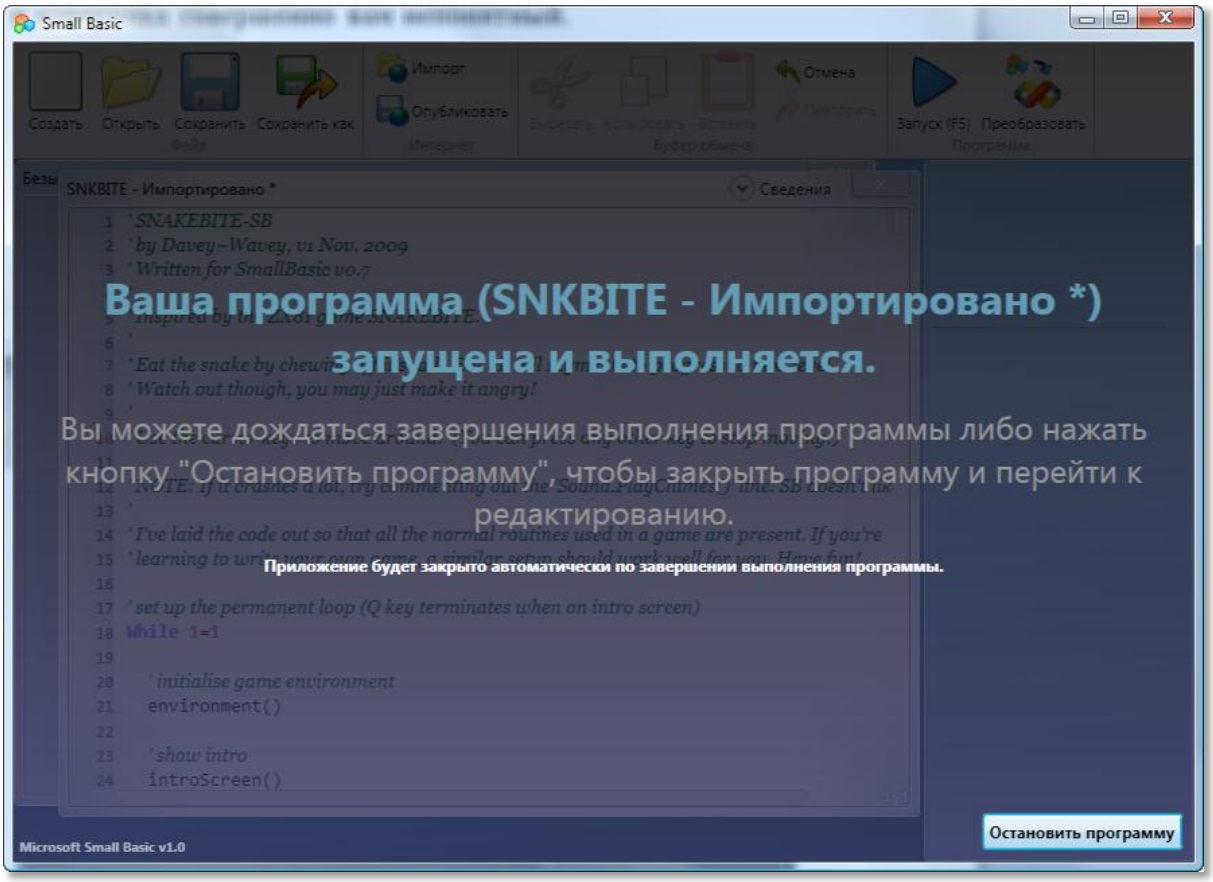


Рис. 2.13. Окно *СБ* становится **тёмным** при запуске программы

Программа написана не безукоризненно и «вылетает», когда мышка догоняет змейку у края поля. Но игра чужая, поэтому чинить мы её не будем. Да она и нужна нам только для того, чтобы посмотреть, как наш бейсик работает. Мы в этом убедились, а изъяны чужой программы нас не должны волновать.

Вы найдёте в Интернете еще много интересных программ для *СБ*, например, *Сокобан* (Рис. 2.14, 2.15).

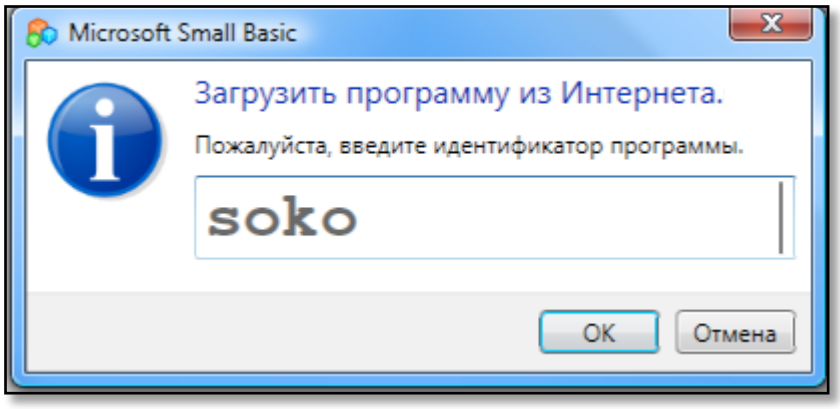


Рис. 2.14. Загружаем *Сокобан*

Рис. 2.15. Играем в *Сокобан*

Лучшую игру всех времен и народов – наш родной *тетрис* (Рис. 2.18, 2.19).

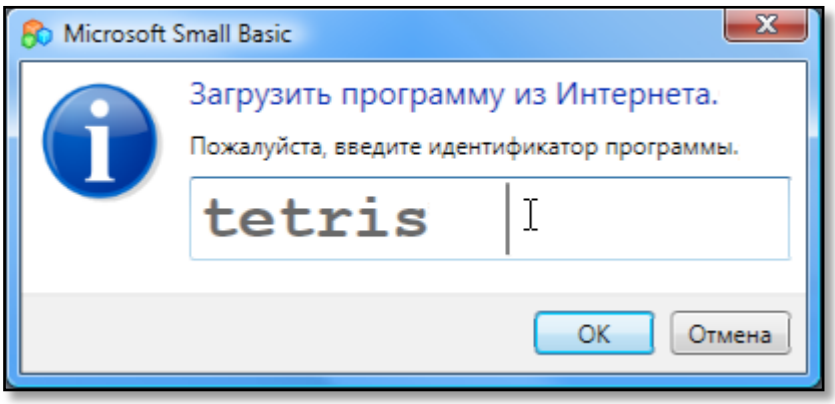


Рис. 2.18. Загружаем *тетрис*

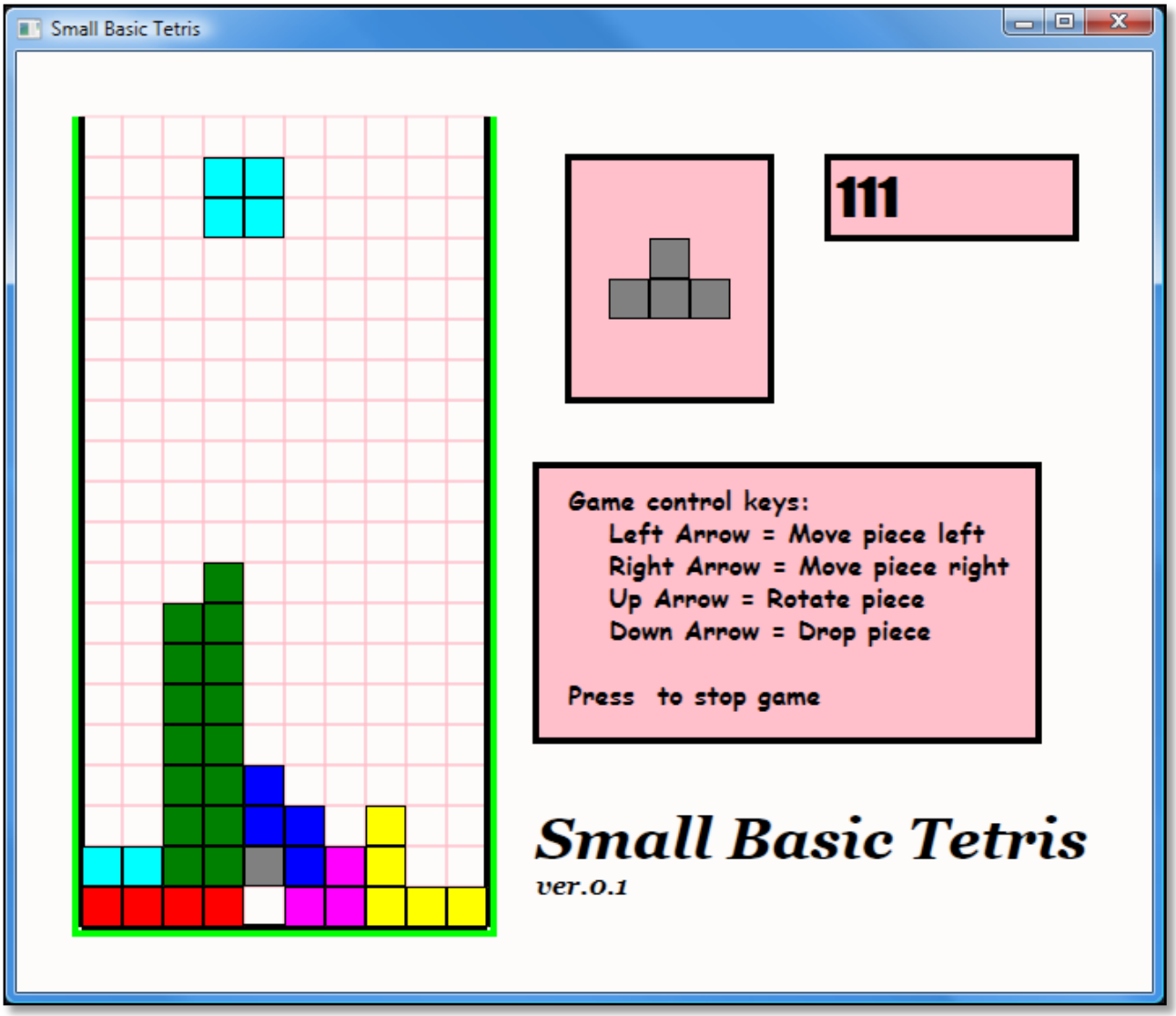


Рис. 2.19. Собираем падающие блоки

И головоломку *Ханойские башни* (Рис. 2.20, 2.21).

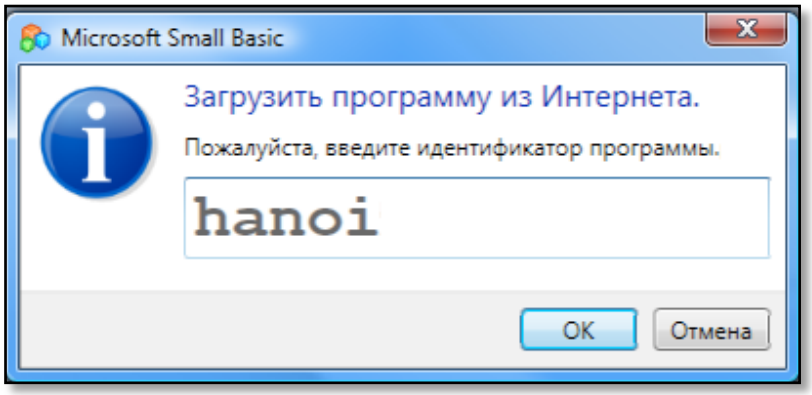


Рис. 2.18. Загружаем *Ханойские башни*

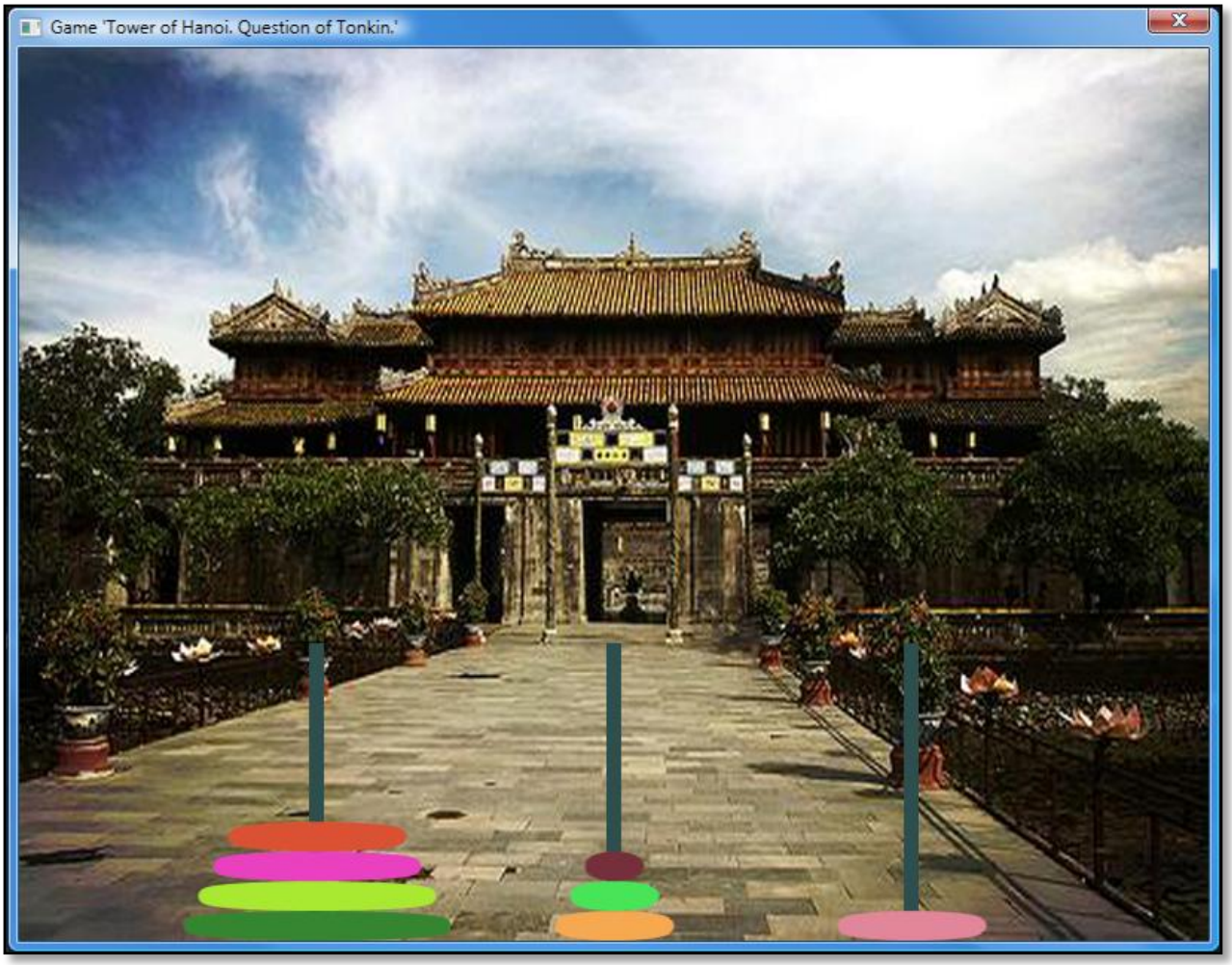


Рис. 2.19. Перекладываем диски

1. Загрузите эти игры и оцените возможности нашего бейсика.

2. Посетите сайт *www.smallbasic.com*, где вы сможете найти еще немало программ для изучения, а также много другой полезной информации.

Урок 3. Как сберечь программу

Сохраняйся!

Программистская пословица

Игра - дело хорошее, но у нас есть дела и поважнее! Обратите внимание: в *Редакторе кода* бейсика открыто *два* документа - *Безымянный* и *SNKBITE - Импортировано* * (Рис. 3.1).

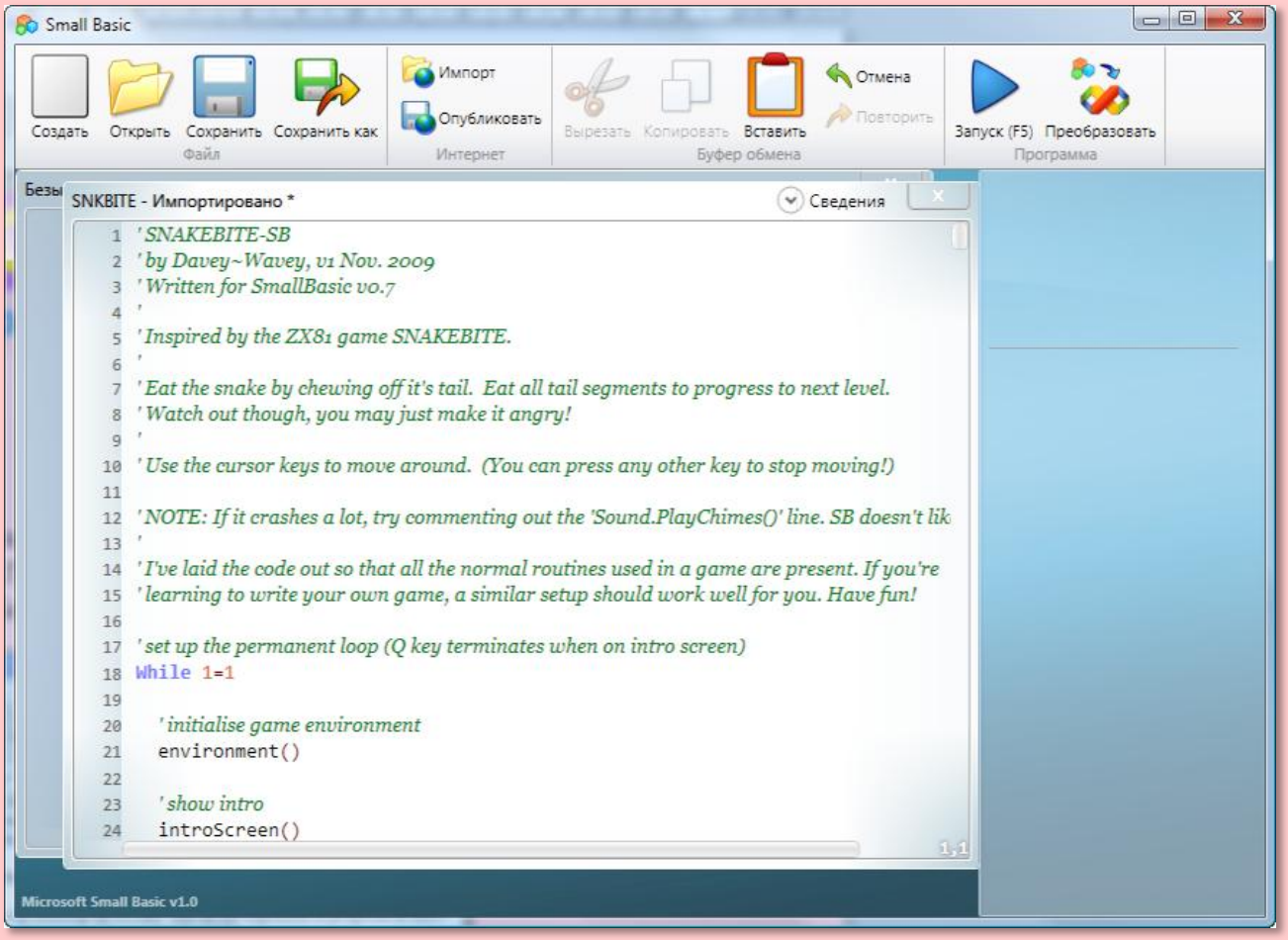


Рис. 3.1. *Текстовый редактор (Редактор кода) СБ*

Первый документ пустой, а во втором - пока непонятные разноцветные слова, которые программисты называют *исходным кодом* (или *исходным текстом*) программы. Он написан латинскими буквами, но не на английском языке (хотя некоторые слова английские), не на немецком и даже не на французском. К сожалению, компьютеры пока не знают ни одного человеческого языка, а понимают только *машинный*, который состоит из ну-

лей и единиц. Но на таком «тарабарском» языке невозможно написать даже самую простую программу, поэтому люди придумали *языки программирования*, чтобы с их помощью объяснять компьютеру, что же он должен делать. Выучить язык программирования, например, бейсик тоже непросто, но компьютерный процессор не понимает даже его. Для перевода с языка программирования на язык машинный необходима программа-переводчик, которая называется *транслятором*. Трансляторы бывают двух видов: *интерпретаторы* и *компиляторы*.

Интерпретатор последовательно, строчку за строчкой просматривает исходный текст программы и передаёт соответствующие команды компьютеру. Если оператор языка программирования в какой-нибудь строке текста выполняется сто раз (а, может быть, и миллион!), столько же раз интерпретатор будет переводить текст в команды процессора. Нетрудно догадаться, что программа будет работать медленно. Чтобы ускорить процесс трансляции, иногда сначала весь исходный текст переводят в *промежуточный код*, который затем интерпретируется значительно быстрее.

Первые версии бейсика были оснащены интерпретаторами, поэтому и скорость работы программ была невысокой. Другой недостаток интерпретатора состоит в том, что для запуска любой программы был необходим весь исходный код, а также сам бейсик-интерпретатор. То есть сначала нужно было запустить бейсик, затем загрузить в него исходный текст и только потом выполнить нужную программу. Конечно, это создавало неудобства для программиста. И поделиться с кем-то своей программой было непросто, ведь другой пользователь также должен был иметь на своем компьютере бейсик и уметь им пользоваться! Правда, у интерпретатора есть и небольшое преимущество - написанная программа сразу же, без предварительной обработки начинает исполняться, что очень важно при отладке.

Компиляторы работают по-другому: они *сразу* просматривают весь исходный текст и преобразуют его в *машинный код*, который процессор исполняет очень быстро. Но вот на компиляцию программы уже нужно некоторое время, поэтому при отладке придётся ждать, пока будет скомпилирована *вся* программа, хо-

тя была изменена, может быть, всего одна буква (современные компиляторы, конечно, «умнее», и не перекомпилируют всю программу целиком). В результате компиляции программист получает *выполняемый файл* программы, который в операционной системе *Windows* называется также *приложением*. Его легко отличить от других файлов на диске по расширению *EXE* (сокращение от английского слова *execute* - *выполнять, исполнять*). Чтобы *запустить* приложение, достаточно дважды щёлкнуть по названию файла мышкой.

А теперь вопрос: наш бейсик - интерпретатор или компилятор? - Мы это скоро узнаем, но, прежде всего, вам следует приучить себя всегда *сохранять* исходный текст программы на диске!

О том, что мы не ещё не сохранили программу, нам подсказывает *звёздочка* после названия документа.

Если вы сейчас попробуете закрыть бейсик, то получите табличку с предупреждением (Рис. 3.2).

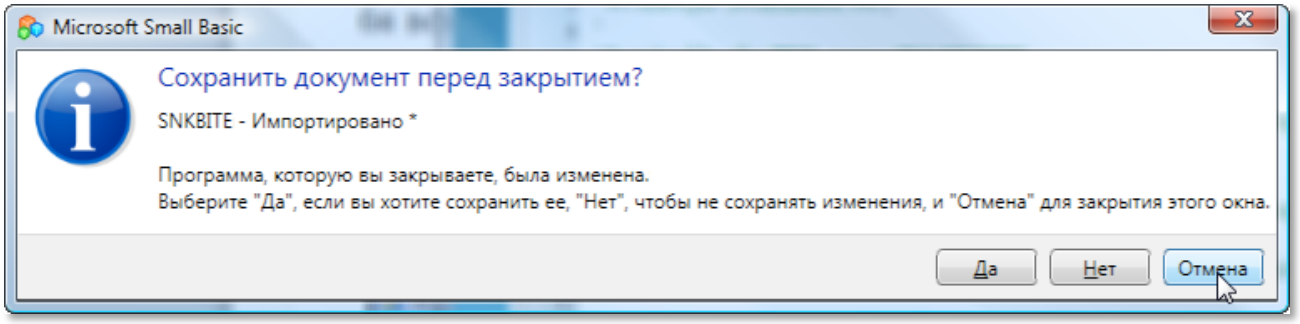


Рис. 3.2. Бейсик предупреждает!

Это хорошо, что бейсик, как и фирма Тефаль, думает о нас, иначе вся программа (правда, мы ещё ничего не написали, но ведь напишем!) была бы безвозвратно потеряна. В данном случае лучше нажать кнопку *Отмена* и сохранить программу без напоминания. Для этого нажмите кнопку *Сохранить как* (Рис. 3.3).

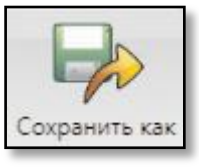


Рис. 3.3. Кнопка для сохранения исходного кода на диске

Откроется диалоговое окно (Рис. 3.4).

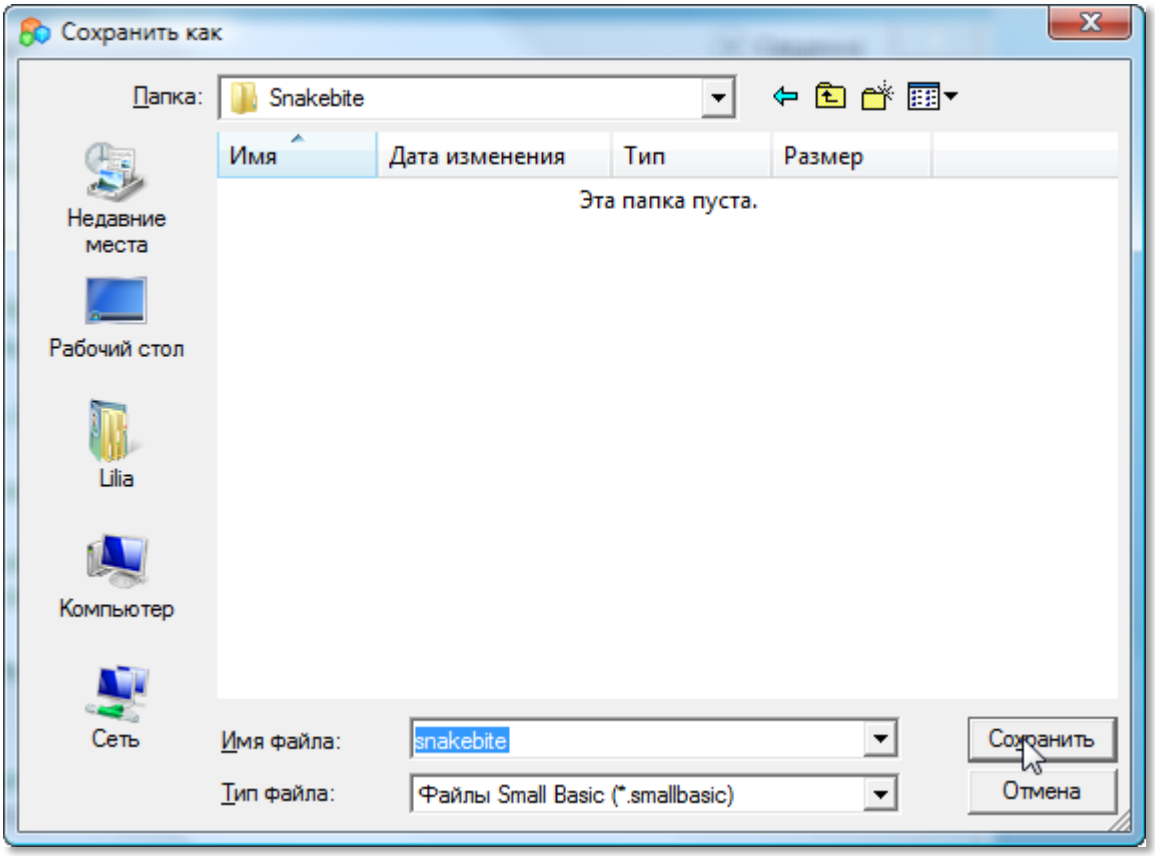


Рис. 3.4. Диалоговое окно для выбора папки и имени файла

В нём нужно выбрать папку для файла и его название.

Конечно, файл с исходным текстом можно сохранить в любом месте на диске. Но найдёте ли вы его потом среди множества других файлов? - Сомнительно! Поэтому для всех своих проектов вообще и на *СБ* в частности следует завести *отдельную папку* в корневом каталоге диска, тогда вам не составит труда эту папку найти. Например, папку со всеми проектами можно так и назвать *Мои проекты* (или *My Projects*). В ней (или отдельно) заведите папку *Мои проекты на Small Basic* (или *My Small Basic Projects*). Еще лучше предварить название папки знаками *подчёркивания* (*__Мои проекты*), тогда эти папки всегда будут выше других папок в *Проводнике Windows* (или в файловом менеджере) и найти их будет ещё проще.

Итак, будем считать, что папку для проектов вы завели. Для каждого нового проекта в ней нужно создать *отдельную папку* с

названием проекта. Например, наш проект естественно назвать *Snikebite*.

Всегда давайте проектам и файлам *вразумительные* имена, а не «ёклмнопрст»!

Ну вот, все папки готовы, осталось назвать сам файл (если программа скачана из Интернета, то можно использовать его идентификатор, чтобы в случае необходимости импортировать оригинал) и нажать кнопку *Сохранить*.

Готово! В папке проекта появился первый файл - *snkbite.sb*. Первая часть названия файла (до точки) это как раз то *имя*, которое мы выбрали, а вторая (после точки) - *расширение* файла, которое *СБ* автоматически добавляет, чтобы отличать его от других файлов. Легко убедиться, что *СБ* узнаёт свои файлы. Закройте бейсик, найдите новую папку и дважды кликните по названию файла - бейсик снова запустится и автоматически загрузит исходный текст программы, но теперь документ будет называться *snkbite.sb*. После названия открытого файла будет указан *полный путь* к нему, включая название папки с проектом. Обратите внимание: звёздочки в конце названия документа нет. Это говорит о том, что файл был сохранён на диске. Но поставьте курсор на *пустую* строчку программы (чтобы не испортить её) и нажмите *ПРОБЕЛ* - звёздочка снова появится, сигнализируя о том, что документ *изменён* и, возможно, его следует сохранить на диске. Если вы хотите записать проект в *новую* папку, то снова нажмите кнопку *Сохранить как*. Но обычно файл сохраняют в той же самой папке, поэтому достаточно нажать кнопку *Сохранить* (Рис. 3.5).

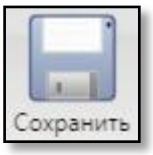


Рис. 3.5. Кнопка для повторного сохранения исходного кода

Звёздочка, конечно, исчезнет, но первоначальный файл на диске будет *заменён* текущим. Имейте это в виду!

При разработке новой программы регулярно нажимайте кнопку *Сохранить*! Как говорится, жизнь полна неожиданностей, всякое может случиться, и ваш тяжкий труд может пойти прахом. Не беда, если вам снова придётся набрать несколько строчек текста, но вот если программа отлажена, вы нашли и исправили все ошибки, а сохранить её на диске забыли, то вам придётся ещё раз отлавливать всех *жучков* (слова *bug, баг, жучок* на программистском жаргоне означают *ошибку*, которую трудно найти). А отладка программы – это самая ответственная и трудная часть работы любого программиста.

В папке с проектом обязательно заведите папку для *архивных* файлов. Назовите ее *_ARCHIV*, или *_АРХИВ*, или как угодно иначе, но она должна быть обязательно, если вы планируете достаточно долго работать над проектом. В этой папке периодически сохраняйте файлы, которые изменяются при работе, например, файлы бейсика с расширением *.sb*. Чтобы файлы занимали меньше места на диске, пользуйтесь *архиватором*. Сжатые файлы последовательно нумеруйте, чтобы всегда можно было вернуться к отлаженной версии программы, если вы наделаете ошибок при дальнейшей работе. Если проект ответственный, то сохраняйте его дополнительно на *другом* диске: вдруг вы случайно сотрёте папку или диск выйдет из строя… И результат месячной (а то и более!) работы над проектом придётся восстанавливать. Принято считать, что нет ничего хуже, чем ждать и догонять, поверьте: писать с самого начала уже готовую программу – куда хуже! Трепетно относитесь к своей работе и берегите её!

СБ запоминает последнюю папку, в которой сохранялся файл, поэтому вы сразу попадете в неё, если захотите загрузить файл с диска кнопкой *Открыть*.

Но давайте снова вернёмся к игре и запустим её, нажав клавишу *F5*. На этот раз мы не будем гоняться за змейкой, а сразу закроем программу и полюбопытствуем, что же происходит в папке с

проектом. А там много интересного! Появились *новые* файлы: *SmallBasicLibrary.dll, snkbite.pdb* и *snkbite.exe*. Назначение первых двух файлов мы обсуждать не будем (их лучше не трогать!), а вот последний даёт нам прямой ответ на наш прямой вопрос: *Small Basic* умеет *компилировать* программы и создавать выполняемый файл.

Small Basic на самом деле создаёт не двоичный файл, который непосредственно выполняется процессором, а код на *промежуточном языке CIL* (по-английски - *Common Intermediate Language*), как и все другие компиляторы для платформы *.NET*. Например, *C#, Managed C++, Visual Basic.NET, Visual J#.NET*.

Так как *Small Basic* это не только язык программирования, но и текстовый редактор, и отладчик, и компилятор, то его называют *интегрированной средой разработки* программ, или сокращенно *ИСР* (по-английски - *Integrated development environment , IDE*).

Если вы захотите поделиться программой с другом, то скопируйте на его компьютер всю папку проекта *целиком*. Также на его компьютере должна быть установлена платформа *Microsoft .NET Framework 3.5 или 4.0*. Но дальше мы рассмотрим более простой и удобный способ распространения ваших программ.

1. Поищите на сайте www.smallbasic.com программы, импортируйте их и сохраните на диске.

2. Загрузите сохранённые программы в *СБ*, воспользовавшись кнопкой *Открыть* (Рис. 3.6).

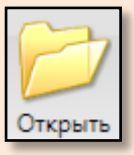


Рис. 3.6. Кнопка для загрузки исходного кода в *Текстовый редактор*

В появившемся одноимённом диалоговом окне (Рис. 3.7) перейдите в нужную папку и щёлкните по названию файла программы, чтобы выделить его, после чего нажмите кнопку *Открыть* (или сразу *дважды* щёлкните по названию).

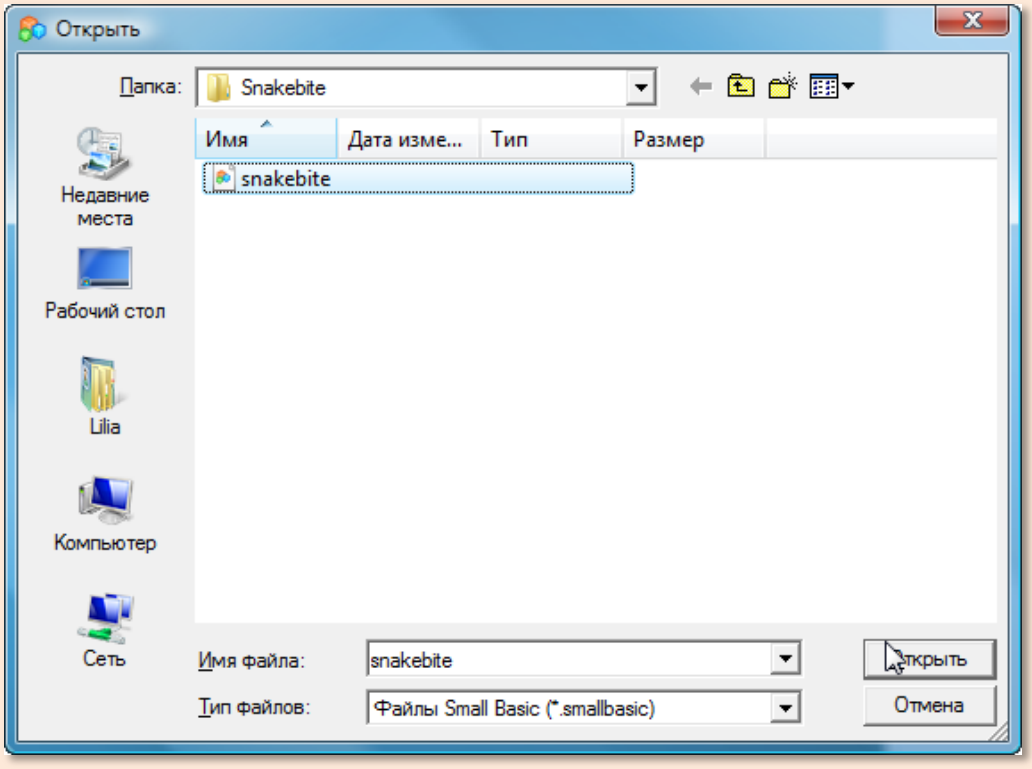


Рис. 3.7. Диалоговое окно для выбора программы на *СБ*

3. Когда в окне *Редактора кода* будет открыто *несколько* документов, вы можете любой из них сделать *активным*, просто щёлкнув по нему. Активное окно можно перемещать, как и любое другое окно, взявшись за его заголовок. Размеры окна также можно изменять, потянув его за сторону или за угол. Поупражнявшись с окнами, закройте их.

Урок 4. Наша первая программа!

Лиха беда начало!

Программистская пословица

Я поэт, зовусь я Цветик!
От меня вам всем приветик!

Мультфильм про Незнайку

В компьютерной литературе принято начинать обучение программированию с создания приложения, выводящего на экран надпись *Hello, World!*. Мы не станем нарушать эту славную традицию, поэтому запустите *Small Basic* и сразу же сохраните документ *Безымянный* в новой папке *Hello* под тем же названием. Как вы помните, для этого следует нажать кнопку *Сохранить как*, а затем выполнить традиционные процедуры для сохранения нового файла. Название документа изменится на *hello.sb*, и мы, наконец, сможем заняться самым интересным в программировании - написать хоть и крохотную программу, но зато своими руками!

Наберите в *Редакторе кода* строчку:

```
1 TextWindow.WriteLine("Hello, World!")
```

Цвет отдельных слов определяется самой *ИСР*, поэтому не ищите кнопок выбора цвета! А выделяются слова не столько для красоты, сколько для удобства ориентирования в исходном коде. Например, *комментарии* выводятся на экран ***зелёным наклонным*** шрифтом. Названия *объектов* – **синезелёным**, *методов* (*операций*) – **коричневым**, *переменных* – **чёрным**, их *значения* – **светло-коричневым**, *ключевые слова* – жирным **синим**. О смысле этих элементов любой программы мы ещё поговорим, но уже сейчас вы должны обратить внимание на то, что сходные по назначению слова выделяются одним и тем же цветом.

Вот и вся программа! Нажимаем кнопку *Запуск* и видим на экране скромные плоды нашего скромного же труда (Рис. 4.1).

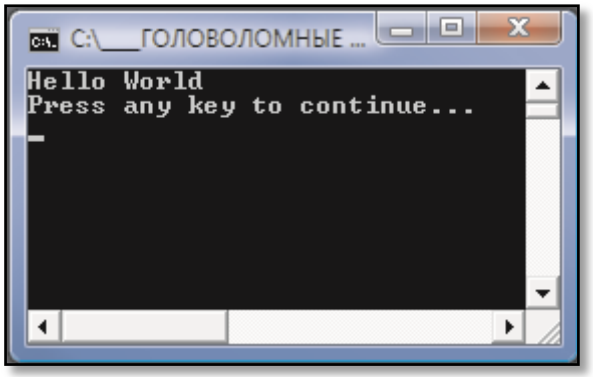


Рис. 4.1. Наша первая программа приветствует мир!

Вы, должно быть, ожидали большего? - Тогда добавим к документу ещё пару строк:

```
1 TextWindow.WriteLine("Hello World")
2
3 GraphicsWindow.DrawText(10,10,"Hello World")
```

Как только вы нажмёте клавишу *ВВОД* (или *ENTER*), к тексту добавится ещё одна пустая строка, причем строки последовательно *нумеруются*, начиная с единицы.

Строки нумеруются только для удобства перемещения по длинному исходному тексту, в самой программе они не используются. Согласитесь, гораздо проще найти нужную строку, если она имеет номер. Правда, номер строки может и измениться, если вы перед ней вставите одну или несколько строк, но тут уж ничего не попишешь!

Некоторые строки принято оставлять *пустыми*, чтобы отделять друг от друга смысловые части программы.

Наверное, вы заметили, что при наборе строки появляется *подсказка*, которая объясняет действие оператора. Например, мы начинаем набирать первую строку программы и, как только мы нажмём клавишу *T*, сразу же увидим *всплывающее окно* подсказки (Рис. 4.2).

Рис. 4.2. «Интеллектуальная» подсказка

Так как *СБ* не может знать заранее, какое слово мы хотим набрать, то она полагает, что это переменная по имени *t*. Но нам нужен класс *TextWindow*, который мы видим ниже, поэтому продолжаем набирать строку дальше (Рис. 4.3).

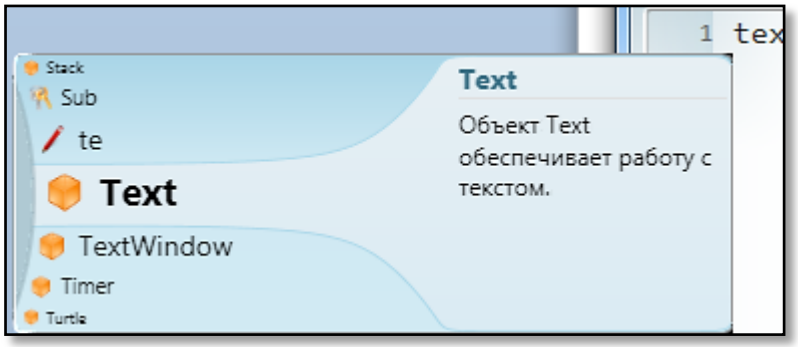


Рис. 4.3. Подсказка автоматически прокручивает список

И ещё дальше (Рис. 4.4).

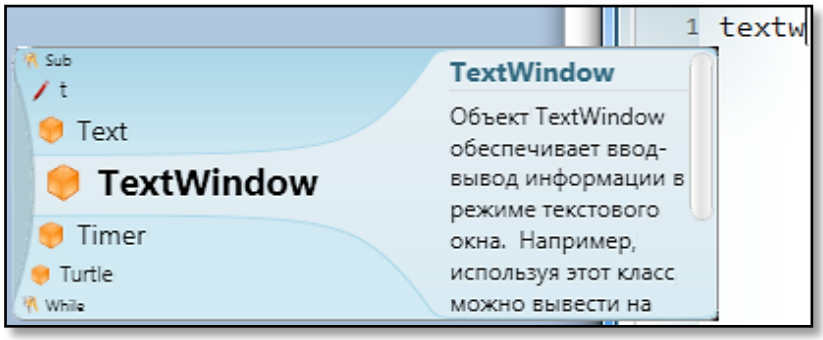


Рис. 4.4. Подсказка нашла нужное слово

Вот теперь в середине окна подсказки оказалось нужное нам слово, и мы можем не продолжать набор, а просто нажать клавишу

ВВОД. Слово целиком появится в окне редактирования (Рис. 4.5), а подсказка исчезнет с экрана.

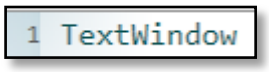


Рис. 4.5. Слово вставлено в строку программы

Заметьте: нам не пришлось до конца вводить довольно длинное слово, и напечатано оно без ошибок, которые мы могли бы сделать!

Можно ещё облегчить себе ввод слов, если просто напечатать первую букву, а когда появится подсказка, прокрутить её на нужное нам слово с помощью колёсика мышки или клавиш со стрелочками *ВВЕРХ-ВНИЗ*. Когда появится нужное нам слово, нажимаем клавишу *ВВОД* или дважды щёлкаем по нему мышкой.

Подсказку можно вызвать в любое время, нажав клавиши *CTRL+ПРОБЕЛ*.

Обратите также внимание на то, что в правой части окна подсказки (Рис. 4.4) появляется информация о текущем объекте (он выделен крупным шрифтом). Если она полностью в окне не помещается, то нужно прокрутить его, как обычно. Когда вы хорошо выучите все операторы *СБ*, эта информация вам не потребуется, но на первых порах очень даже пригодится!

Еще больше сведений о выбранном операторе вы получите на *правой панели ИСР* (Рис. 4.6). Например, вы узнаете, что *TextWindow* это объект, который служит для ввода и вывода текстовой информации. Далее перечислены все его *свойства* и *методы*, которыми вы можете пользоваться в своих программах.

Название свойства или метода отделяется от названия объекта *точкой*. Только вы поставите точку после слова *TextWindow,* как снова всплывет окно подсказки, в котором вы сможете выбрать свойство или метод (Рис. 4.7).

В правой панели информация обновится (Рис. 4.8).

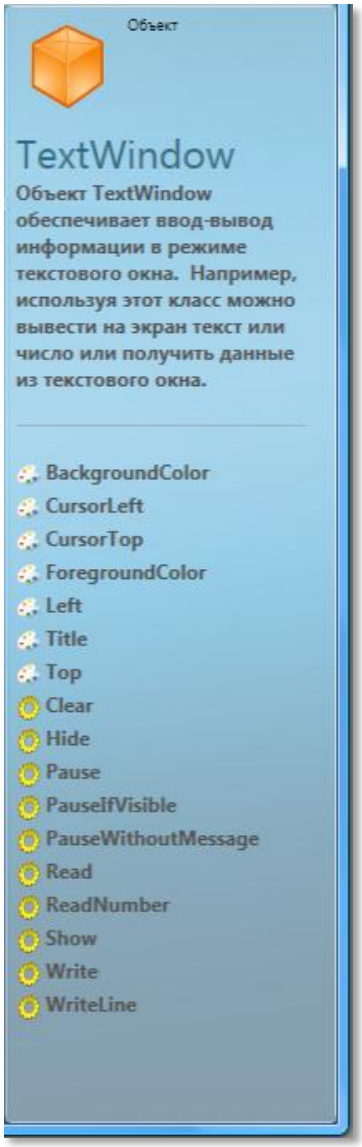


Рис. 4.6. Панель справки *СБ*

Здесь вы найдете все сведения, например, об операции (методе) *WriteLine*. Вот так, покручивая колёсико мышки, можно выучить весь бейсик. Это, конечно, шутка: по словарю немецкого языка говорить не научишься. С бейсиком ничуть не проще.

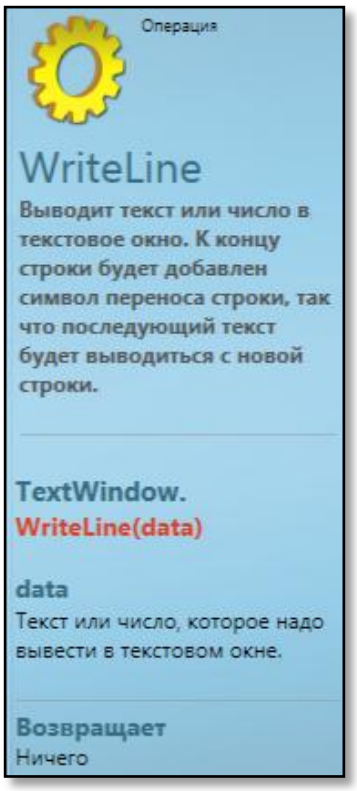


Рис. 4.8. Информация о методе *WriteLine*

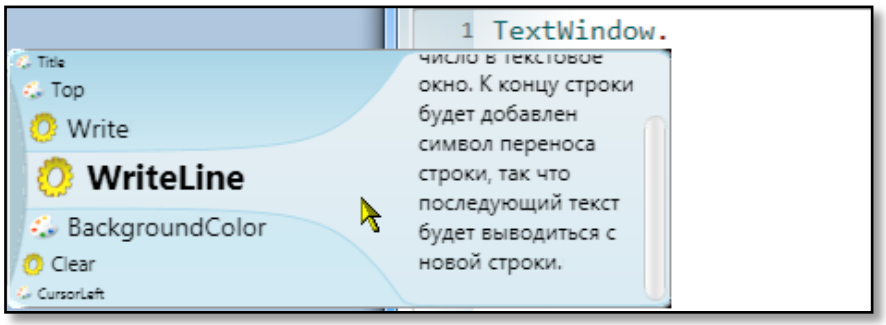


Рис. 4.7. Подсказка выводит список свойств и методов объекта *TextWindow*

Если вы захотите получить помощь по любому оператору программы, который имеется в исходном коде, щёлкните по нему и прочитайте в *правой панели* короткую справку. Так что весь справочник по *СБ* у вас всегда под рукой!

После прогулки по *СБ* снова запускаем программу на выполнение - теперь появится ещё одно окно – *графическое* (*текстовое окно* также останется) - с приветствием (Рис. 4.9).

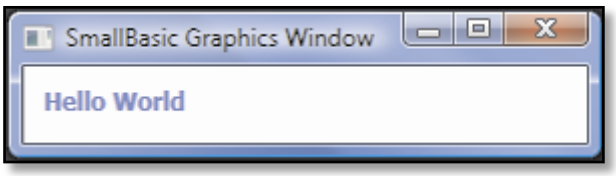


Рис. 4.9. *Графическое* окно

Этот вариант, наверное, вас полностью удовлетворит, особенно если учесть, что вы набрали всего две строки текста, а ваша программа уже успешно работает. Ни в одной другой среде разработки вы не сможете вот так запросто написать работоспособную программу! Это уже здòрово, но ведь вы легко можете добавить и другие строки к документу, которые затем появятся на экране. Например, вы можете поприветствовать мир и на родном языке, присовокупив к исходному тексту программы ещё одну строку:

```
1 TextWindow.WriteLine("Hello World")
2
3 GraphicsWindow.DrawText(10,10,"Hello World")
4
5 GraphicsWindow.DrawText(10,30,"Здравствуй, Мир!")
```

Запускаем программу и получаем результат (Рис. 4.10).

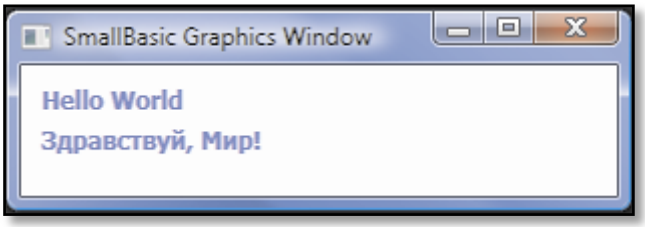


Рис. 4.10. Именно то, что мы хотели!

А заглянем-ка в папку с нашей программой - в ней опять появились новые файлы, в том числе и готовое приложение *hello.exe*. Если вы дважды кликнете по нему, то программа запустится, и на экране появятся окна с приветствиями. А всё-таки ловкие мы ребята: ничего не зная о программировании, сумели написать полноценное приложение для *Windows!*

Конечно, вы должны помнить, что бейсик многое сделал за вас, потому что даже создание пустого окна *Windows* требует немало усилий со стороны программиста, а вы можете вывести пустое окно на экран с помощью всего одной строки.

Нажмите кнопку *Создать* (Рис. 4.11), и в окне *Редактора кода* появится новый пустой документ с тем же самым названием *Безымянный*.

Рис. 4.11. Кнопка *Создать*

С помощью подсказок наберите строчку:

Запускаем программу - и перед нами настоящее окно *Windows*, со всеми кнопками (Рис. 4.12).

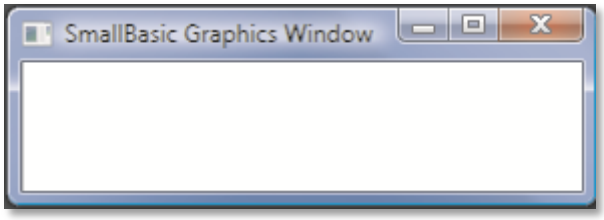


Рис. 4.12. Стандартное окно *Windows*

Вы можете перемещать его по экрану, изменять размеры, сворачивать и разворачивать и, в конце концов, закрыть, нажав на кнопку с крестиком.

Если вы не планируете использовать программу ещё раз, как, например, эту, то и сохранять её на диске не обязательно.

Как видите, создавать приложения для *Windows*, имея *СБ*, очень просто.

Обратите также внимание на то, что сейчас в *Редакторе кода одновременно* открыты *два* документа, и вы можете легко переключаться между ними, просто кликая на нужном вам документе. Вы можете открыть *сколько угодно* документов (Рис. 4.13).

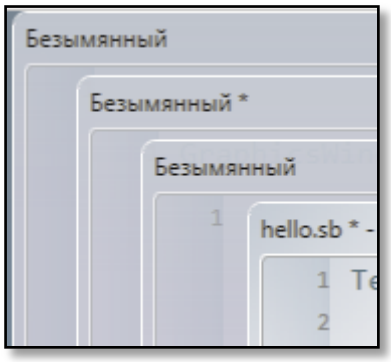


Рис. 4.13. Документы в *Редакторе кода*

Если окно с каким-либо документом полностью закрыто другим окном, вы можете просто перетащить в сторону верхнее окно, подцепив его мышкой за строку заголовка. Такой *многодокументный интерфейс СБ* очень удобен при работе над несколькими проектами одновременно, потому что вам не придётся постоянно закрывать и открывать нужные вам проекты.

Одно или несколько окон вы можете использовать для *временного хранения* части кода разрабатываемой программы. Вырежьте или скопируйте несколько строк из основной программы и вставьте их во временное окно. Измените эти строки при отладке программы. Если новый вариант программы работает неверно, то вы легко сможете вернуть изменённые строки на место из временного хранилища.

В проекте *Hello* при запуске программы мы получаем сразу два окна - одно **невзрачное** - *текстовое*, второе **красочное** - *графи-*

ческое. Вам может показаться, что текстовое окно вовсе не нужно, если есть графическое, но это не совсем так.

Раньше все программы были *консольными* и выводили информацию исключительно в *текстовом* виде. Не очень красиво, но тогда и компьютеры использовались только для серьёзных вычислений, так что результаты вполне можно было представить в виде строк, состоящих из слов и чисел. «Ну, это было давно!» - скажете вы, и опять будете неправы: и сейчас нередко результат работы программы достаточно вывести в текстовом виде. Например, если вы хотите узнать у компьютера, сколько будет дважды два, то вам совсем не нужно графическое окно. Пишете «программу»:

```
1 TextWindow.WriteLine("2 * 2 = " + 2 * 2)
```

Запускаете её и в текстовом окне получаете результат (Рис. 4.14).

Рис. 4.14. Результат вычислений в *Текстовом окне*

Всё, вы удовлетворили свое любопытство!

Конечно, когда вы программируете на *СБ*, программа,

```
1 GraphicsWindow.DrawText(10,10, "2 * 2 = " + 2 * 2)
```

которая выводит тот же самый результат в графическом окне (Рис. 4.15), ничуть не сложнее, но если бы вам пришлось программировать на *C++*, то разница была бы ощутимой.

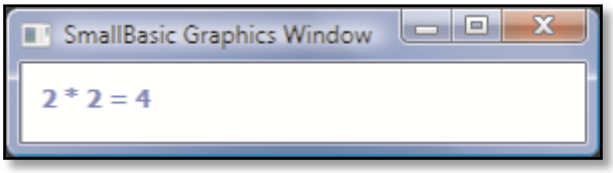


Рис. 4.15. Результат вычислений в *Графическом окне*

Поэтому мы будем действовать, как настоящие программисты: если нам будет достаточно только увидеть результат вычислений, то мы будем создавать *консольное приложение* с текстовым окном, а если потребуется графика - приложение *Windows* с *графическим* интерфейсом.

Напишите ещё несколько коротких программ, выводящих в графическое и текстовое окно строчки или результаты арифметических вычислений.

Урок 5. Какие бывают числа

Числа правят миром.

Пифагор

Как вы знаете из уроков математики, чисел бесконечно много, но все их можно разбить на отдельные *подмножества* по тем или иным признакам.

Самые первые числа, которые придумали ещё первобытные люди, называются **натуральными**. Они использовались для подсчёта различных предметов, например, яблок или палочек, на которых вы и сами учились считать в первом классе.

Папа спрашивает у сына: «Скажи, сколько будет, если к трём грушам прибавить ещё две груши?

Сын отвечает: «Не знаю, папа, мы в школе решаем задачи только про яблоки!

Множество натуральных чисел обозначается большой латинской буквой **N**, поэтому само множество можно записать так: N = {1, 2, 3, ...}. Иногда к множеству натуральных чисел относят и *ноль* (отсутствие предметов вообще): N = {0, 1, 2, 3, ...}. Множество натуральных чисел является подмножеством всех чисел и также бесконечно.

Если к натуральным числам добавить *отрицательные числа* (и нуль), то получится множество **целых чисел**. Оно обозначается большой латинской буквой **Z** = {... 0, -2, -1, 0, 1, 2, ...}. Нетрудно догадаться, что и целых чисел бесконечно много.

В арифметике обычно используют именно целые числа, но встречаются алгебраические и геометрические задачи, которые нельзя решить без *дробных чисел*.

Рациональные числа можно представить в виде *простой (обыкновенной) дроби* m/n, где

m - целое число;

n - натуральное число, не равное нулю (вы, конечно, помните, что на нуль делить нельзя!).

Множество рациональных чисел обозначается буквой **Q**. Если знаменатель дроби равен 1, то дробь равна числителю, то есть целому числу *n*. Таким образом, все целые числа являются в то же время и рациональными (множество целых чисел это подмножество рациональных). Но не наоборот!

Рациональные числа можно представить также в виде *конечной десятичной дроби* (1/2 = 0,5) или *бесконечной периодической десятичной дроби* (1/7 = 0,1428571...).

Продвигаемся дальше вглубь математики! **Иррациональные числа** не могут быть представлены в виде простой дроби (а также конечной или бесконечной десятичной периодической дроби), таким образом, иррациональным числом называют любое число, представимое в виде *бесконечной не периодичной десятичной дроби*. Примером такой дроби может служить корень квадратный из двойки. Иррациональность этого числа была известна уже древним математикам, которые доказали несоизмеримость стороны и диагонали квадрата.

Иррациональные числа обозначают буквой **I**.

Множество **действительных**, или **вещественных чисел** объединяет множества рациональных и иррациональных чисел. Их принято наглядно представлять в виде точки на *числовой прямой* (Рис. 5.1).

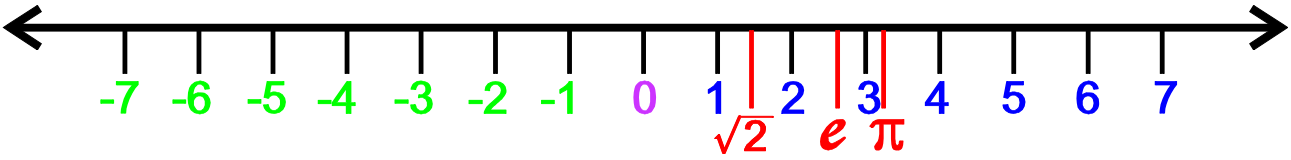


Рис. 5.1. Числовая прямая

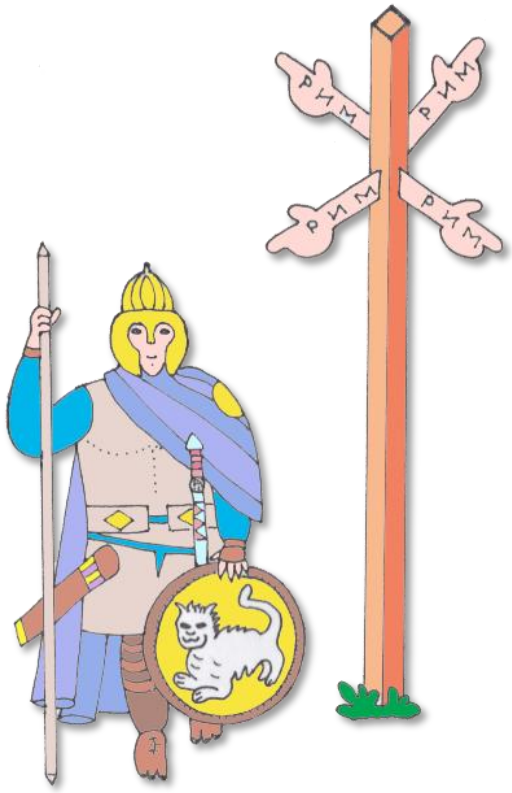
Множество действительных чисел принято обозначать буквой **R** (от их латинского названия *numerus realis*).

К иррациональным относятся знаменитые числа - *пи* (отношение длины окружности к диаметру) и *e* (основание натуральных логарифмов).

Иногда в занимательных задачах присутствуют *комплексные* числа, но нам они не понадобятся.

С множествами чисел мы разобрались, но нас, конечно, интересуют не все числа вообще, а «особенные». И начнём мы с натуральных чисел, но записанных *по-римски*.

Римские числа



Все дороги ведут в Рим.

Из ответа на уроке географии

Древние римляне создали огромную, могущественную империю, которую назвали в свою честь, но что касается чисел, то тут они изрядно начудили и придумали столь вычурную систему обозначения чисел буквами, что даже простейшие арифметические вычисления давались им с большим трудом. Существенно облегчили школярам тяготы учёбы арабские - но придумали их индийцы! - цифры, пришедшие на смену римским.

Римская империя давно пала под натиском варваров, а их цифры сохранились до сих пор в первозданном виде. К счастью, нам не нужно пользоваться ими на уроках математики, но как украшательство мы можем найти их в книгах для обозначения глав, для подсчёта столетий, царей, съездов и других исторических событий.

Поскольку компьютер не понимает не только римских цифр, но даже арабских, то мы напишем простую программу **Rome**, которая поможет нам разобраться в премудростях римской нумерации.

Как это будет по-римски?

- ***Parla italiano?***

Italiano = FALSE

Basic = TRUE

Из разговора двух компьютеров

Для ввода и вывода информации нам вполне хватит текстового окна, но мы его немного *раскрасим*. Но это потом, а пока давайте позаботимся о **переменных**, которые нам пригодятся в программе.

Переменные – это элементы программы, которые *могут изменять значение* в ходе её выполнения.

Физически переменная представляет собой область памяти компьютера, в которой хранится значение переменной. Все переменные должны иметь *имя* (*идентификатор*).

Запустите *СБ* и сохраните новый проект в папке ***Rome*** под тем же названием. Начало положено, но документ совершенно пуст, поэтому начнём наполнять его «контентом».

Тут нужно сразу упомянуть о странной особенности *СБ* - в нём нет именованных констант, поэтому роль констант в наших программах будут играть переменные. И это плохо, потому что значение переменной можно случайно изменить в программе, в то время как константы застрахованы от этой напасти. Впрочем, в этой программе мы вполне можем обойтись и без констант, но на будущее: имейте в виду, что все переменные в вашей программе можно изменить, даже непреднамеренно.

Константы – это элементы программы, которые *не могут изменять значение* в ходе её выполнения.

Все числа являются *числовыми константами*. Из чисел, знаков операций, идентификаторов методов и свойств объектов, а также круглых скобок можно составлять *константные выражения*:

```
123
 "Z"
Math.Cos(1)
“string1”
Text.GetCharacter(111-32)
```

Именованные константы должны иметь идентификатор (имя), как и переменные. Неименованные константы называют также *литералами*.

Более того, все переменные в программе на *СБ* - *глобальные*, а это значит, что они существуют с момента первого появления в программе и до её закрытия. Их нельзя уничтожить, поэтому старайтесь не объявлять лишних переменных.

Все глобальные переменные доступны из любого места программы, поэтому если вы присвоите значение переменной в одном месте, то она будет иметь то же самое значение и во всех других местах программы, где она используется.

Переменную в *СБ* можно *объявить* в любом месте программы, причём одновременно с её *инициализацией*, то есть ей сразу нужно присвоить *начальное значение*. Обычно глобальные переменные объявляются в начале программы, чтобы потом не разыскивать их по всему исходному коду. Поэтому мы соберём все переменные программы в её начале и предварим их список комментарием:

```
'variables
```

Все **комментарии** начинаются с символа *апострофа* (он похож на одиночную кавычку) и могут располагаться в начале строки или после оператора, но в любом случае его действие распространяется до *конца* строки, то есть после комментария нет смысла записывать операторы – они всё равно выполняться не будут. Комментарии выделяются ***зелёным наклонным шрифтом***, поэтому их легко распознать в исходном тексте.

Теперь запишем все *переменные* нашей программы:

```
'НАЙТИ РИМСКОЕ ЧИСЛО

'variables
number=0
sNumber=""
n=0
num[1]=1000
num[2]= 900
num[3]= 500
num[4]= 400
num[5]= 100
num[6]= 90
num[7]= 50
num[8]= 40
num[9]= 10
num[10]=9
num[11]=5
num[12]=4
num[13]=1

sNum[1]="M"
sNum[2]="CM"
sNum[3]="D"
sNum[4]="CD"
sNum[5]="C"
sNum[6]="XC"
sNum[7]="L"
sNum[8]="XL"
sNum[9]="X"
sNum[10]="IX"
sNum[11]="V"
sNum[12]="IV"
sNum[13]="I"
```

Проясним их *назначение*. Первая переменная *number* - это как раз то *арабское* число, которое мы будем переводить в *римское*. Как мы знаем, при объявлении переменной ей нужно присвоить какое-нибудь начальное значение. Если вам безразлично - какое, то всегда присваивайте переменной значение *нуль*!

Переменная *number* хранит числовое значение, а следующая - *sNumber* - строковое, о чём говорят двойные кавычки после знака присваивания. Все строки должны быть заключены в кавычки, иначе они будут восприниматься *СБ* как число или как значение переменной с таким именем, что приведет к ошибке.

К сожалению, в *СБ* одна и та же переменная может принимать и числовые, и строковые значения, например, переменной *number* легко присвоить строковое значение:

```
number= “строка”
```

Конечно, вы должны использовать переменные в соответствии с их первоначальным *типом*. Только в редких случаях вам может пригодиться универсальность переменных *СБ*.

Для контроля над типом переменной её название можно начинать с *префикса*. Например, ***i*** (*integer* - целое) или ***n*** (*number* - число) - для целых чисел, а ***s*** (*string* - строка) - для строковых.

Поскольку большинство переменных в программе *целочисленные*, то для них префиксы допустимо не указывать.

Мы вспомнили, что числа бывают натуральные, отрицательные целые и действительные. Перед отрицательным числом, как и в математике, нужно записать знак *минус*, а дробная часть действительного числа отделяется от целой части *десятичной точкой*, а не запятой:

fNumber= -1.12345678912345

fNumber= .12345678912345

Если вы хотите обозначить тип *действительной* переменной, то можете ставить префикс **f** (*float* - число с плавающей точкой).

Точность действительной переменной для однозначной целой части - 14 знаков после запятой. Нам должно хватить! Но с увеличением целой части точность числа уменьшается таким

образом, что всего в числе - 15 значащих цифр. Это касается и целых чисел. Если в числе цифр больше, то оно округляется – последние разряды обнуляются.

Если целая часть действительного числа меньше единицы, то ее можно не указывать, а сразу ставить десятичную точку: .5. Но если вы запишете число «по правилам» - 0.5, то результат получите тот же самый.

Однако вернёмся к строковой переменной *sNumber*. Она проинициализирована *пустой строкой* - для этого просто два раза нажмите на клавишу с кавычками. Пустая строка не содержит ни одного символа (буквы, цифры или знака препинания). Если начальное значение строковой переменной вас не интересует, всегда присваивайте ей пустую строку!

Переменная *n* - вспомогательная и служит для расчётов внутри программы. *Вспомогательные переменные* принято обозначать одной или несколькими буквами, а вот для «настоящих» переменных программы следует выбирать осмысленные имена, чтобы потом не гадать об их назначении.

Никогда не используйте одну и ту же переменную (за исключением, естественно, вспомогательных) для *разных* целей. Такая путаница часто приводит к ошибкам, которые очень трудно найти!

СБ позволяет использовать в качестве имён переменных и *русские* слова, поэтому вы вполне можете вместо переменной *number* объявить переменную *номер* и использовать её в программе. Если вы пишете программу для себя, то это вполне допустимо, но если вы планируете делиться своими наработками с любителями программирования по всему миру, то этот способ именования переменных применять не следует. Тем более что ключевые слова и объекты *СБ* всё равно придётся писать по-английски. Русский язык лучше оставить только для комментариев.

Не пытайтесь давать переменным имена, совпадающие с *ключевыми словами бейсика* – **If**, **For**, **While**, **Goto** - и названиями объектов – **TextWindow**, **Math**, **Array**. Первые выделяются

в тексте **синим** цветом, вторые – **сине-зелёным**. Идентификаторы переменных всегда **чёрные**.

Дальше - ещё интереснее! *Простые переменные*, о которых мы говорили, могут одновременно хранить *единственное значение*. Но нередко нужна переменная для одной и той же цели, но при этом она должна хранить *несколько значений*. Конечно, можно использовать нужное количество обычных переменных, подставляя после названия соответствующий номер:

```
num1=1000
num2= 900
num3= 500
```

Но для *СБ* это три *разные* переменные, поэтому вам придётся к каждой из них обращаться по имени-отчеству, а это бывает очень неудобно. В подобных случаях используют переменную типа **массив**. Она записывается, как обычная переменная, но затем в *квадратных скобках* указывают номер значения переменной (*индекс в массиве*). Причем *СБ* разрешает использовать для индексов *любые* значения (и числовые, и строковые), а не обязательно последовательные числа. Это, конечно, «круто», но осторожно пользуйтесь этой возможностью!

Массив – это набор объектов (*элементов*) одного и того же типа, каждый из которых имеет свой *номер* (*индекс*) в этом наборе. В *СБ* все переменные могут иметь *любой* простой тип, поэтому значения элементов одного и того же массива могут быть и строковыми, и числовыми.

Массивы относятся к *структурным типам* данных. Кроме типа массив, в *СБ* имеется еще один структурный тип данных – *стек.*

На самом деле в *СБ* все переменные имеют один и тот же тип *Primitive*, который перекочевал в *Small Basic* из другой программы фирмы *Microsoft - Visual Basic.* Он представляет собой *структуру* – тип данных, отсутствующий в *СБ*.

Вы легко найдёте в программе *Rome* два массива: один - *num[]* - для чисел, второй - *sNum[]* - для строк. В обоих массивах по 13 элементов, начальные значения которым мы сразу же и присваиваем.

Таким образом, эти переменные играют в программе роль *именованных констант,* поскольку у нас даже в мыслях нет изменять их значения в программе.

Если бы не краткость нашей жизни, можно было бы просто записать в массивы все числа от 1 до 3999 «по-арабски» и «по-римски», но поскольку это не так, то мы схитрим и в массиве *num* сохраним только *13* самых необходимых сочетаний арабских и римских чисел, а остальные нам придётся вычислять.

Мы разумно ограничим себя числом 3999 сверху и 1 снизу (а меньше у римлян и не было), потому что другие римские числа записывались с помощью непечатных символов.

Легко понять, почему нам понадобились именно эти числа, – остальные довольно просто собрать из них, как ожерелье - из бусин. И вот как это делается.

Сначала программа считывает с помощью метода *ReadNumber* число, которое пользователь ввёл с клавиатуры, и присваивает переменной *number* значение, равное этому числу:

```
'========================================
'            ОСНОВНАЯ ПРОГРАММА
'========================================
start:
  number=TextWindow.ReadNumber()
  sNumber=""
  n=0
```

Затем она проверяет введённое число, и если оно равно нулю, закрывает приложение:

```
' Если задан нуль, то работу с программой заканчиваем:
if (number = 0) then
```

```
  goto exit
EndIf
```

Для формирования строки с римским числом нам потребуются переменные *n* и *sNumber*. Вся премудрость конвертирования чисел таится в двух циклах *While*. Мы последовательно сравниваем заданное число *number* с теми числами, которые хранятся в массиве *num*, начиная с тысяч. *Внутренний цикл While* как раз и нужен для того, чтобы определять, сколько в *number* имеется тысяч, сотен, десятков и единиц (все остальные римские числа - 900, 500, 400, 90, 50, 9, 5, 4 - могут быть только в *единственном* числе). Если текущее значение *number* не меньше этих чисел, то из него число вычитаем, а в строку добавляем буквы, соответствующие этому числу в римской записи:

```
'Формируем строку с римским числом, равным заданному числу
number:
if (number < 1) or (number> 3999) then
  sNumber= " "Число должно быть от 1 до 3999!"
  goto print
else
  while number>0
    n= n+1
    while num[n] <= number
      sNumber= sNumber + sNum[n]
      number= number - num[n]
    EndWhile
  EndWhile
Endif
```

Как только от числа ничего не осталось (это контролирует *внешний цикл While*), преобразование числа заканчивается, и мы смело можем печатать на экране перевод римского числа на современный европейский математический язык:

```
'Печатаем результат вычислений на экране:
print:
  TextWindow.WriteLine(sNumber)
```

```
  Goto start

' Выход из программы:
exit:
```

Вижу, что вам не терпится проверить программу в деле! Тогда смело жмите на кнопку *Запуск*, вводите числа, затем нажимайте клавишу *ВВОД* - и всё у вас получится (Рис. 5.2).

Сделаем дополнительную проверку и зададим числа 9999 и -1. Программа отреагирует немедленно (Рис. 5.3).

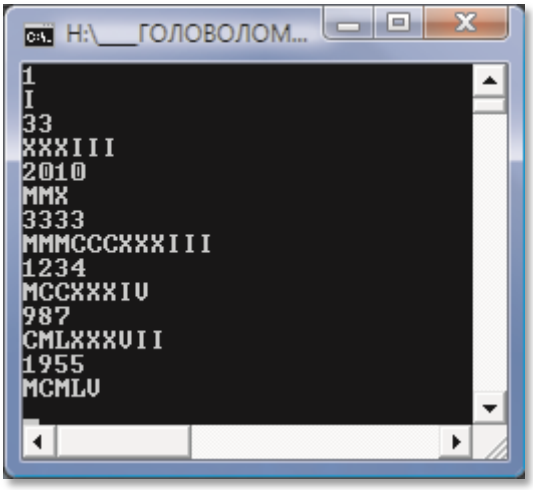


Рис. 5.2. Получилось не очень красиво, но программа работает правильно!

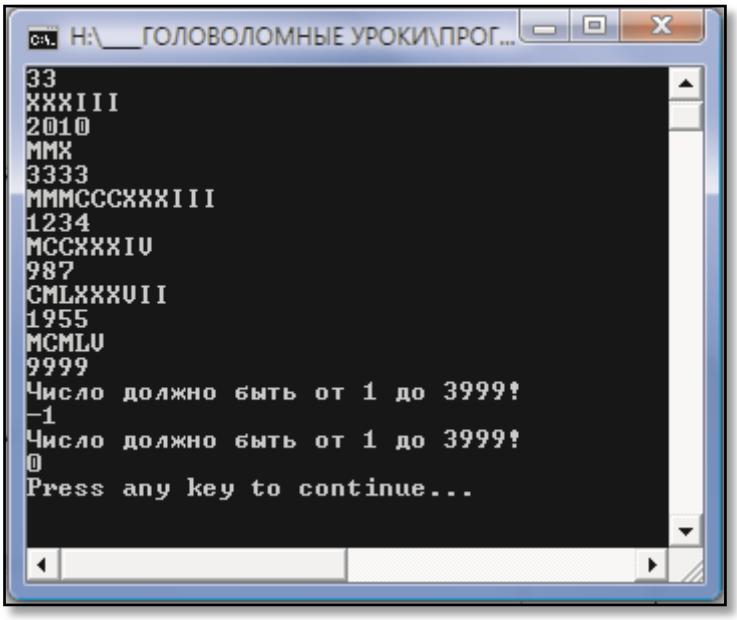


Рис. 5.3. А программа-то совсем не глупа!

Наконец вводим *нуль* – и работа с программой заканчивается.

Исходный код программы находится в папке **Rome**.

Раскрашиваем окна!

Но крашу, крашу я заборы,
чтоб тунеядцем не прослыть!
Песенка Царя из мультфильма
Вовка в Тридевятом царстве

Единственное порицание, которого заслуживает наша программа – какая-то она невыразительная, **серая**, поэтому давайте сделаем себе красиво!

Для «цветного» варианта программы запишем исходный текст в папку **Rome2** под аналогичным названием – теперь можно смело фантазировать!

Пусть цвет арабских чисел будет ***жёлтым***! Для этого нам достаточно свойству *ForegroundColor Текстового окна* присвоить значение "*Yellow*":

```
start:
  TextWindow.ForegroundColor="Yellow"
  number=TextWindow.ReadNumber()
  sNumber=""
  n=0
```

Дальше мы подробно рассмотрим консольные приложения, в том числе и цвета символов.

Римские числа мы сделаем ***зелёными***, а предупреждающую надпись – ***красной***:

```
TextWindow.ForegroundColor="Green"
```

```
'Формируем строку с римским числом, равным заданному числу number:
if (number < 1) or (number> 3999) then
  sNumber= "Число должно быть от 1 до 3999!"
  TextWindow.ForegroundColor="Red"
  goto print
else
  while number>0
    n= n+1
    while num[n] <= number
      sNumber= sNumber + sNum[n]
      number= number - num[n]
    EndWhile
  EndWhile
Endif
```

Вся остальная часть программы осталась без изменений, поэтому, добавив несколько строчек, запускаем программу (Рис. 5.4).

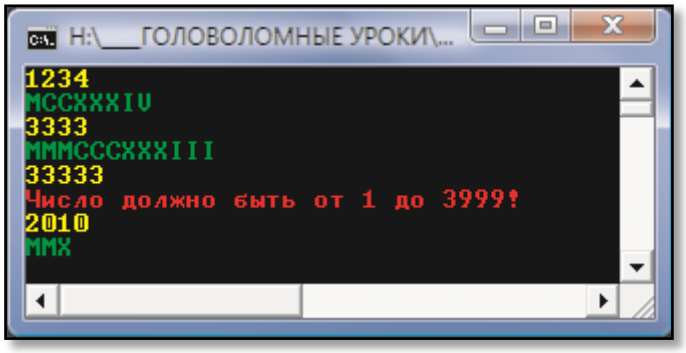


Рис. 5.4. Теперь вся информация выводится распрекрасно!

Исходный код программы находится в папке **Rome2**.

1. Переведите в римские числа ещё несколько арабских. Убедитесь, что программа работает верно.

2. Составьте небольшую программу, подобную той, что мы написали на чётвертом уроке для вывода в текстовое окно выражения «2 * 2». Она должна проводить сложные арифметические вычисления – с числами, знаками и скобками.

Урок 6. Консольные приложения

Мы уже познакомились с консольными приложениями и договорились использовать *Текстовое окно* в тех программах, которые не требуют графического интерфейса пользователя.

И хотя консольные приложения могут показаться устаревшими, но, например, в современном языке *C#* имеется класс *System.Console* для создания таких приложений. С их помощью обычно изучают непростой язык *C#*, поскольку в таких программах нет постороннего кода для обслуживания графического интерфейса, который, по сути, не имеет отношения к самому языку программирования. Вот так выглядит простейшее консольное приложение, написанное в *Microsoft Visual C# 2010* (Рис. 6.1).

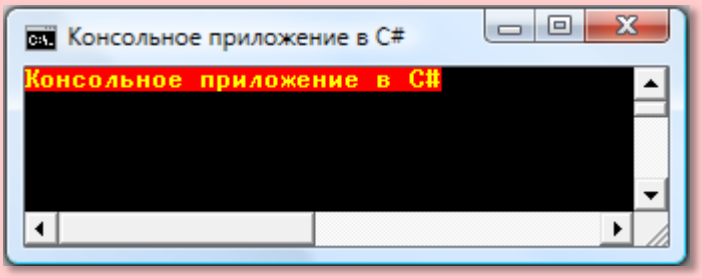


Рис. 6.1. Окно консольного приложения

Вы можете сравнить окно этого приложения с *текстовым окном СБ* – они практически неотличимы. Более того, и код программы совсем на другом языке почти совпадает с кодом на *СБ*:

```
Console.ForegroundColor = ConsoleColor.Yellow;
Console.BackgroundColor = ConsoleColor.Red;
Console.Title= "Консольное приложение в C#";
Console.WriteLine("Консольное приложение в C#");
Console.Read();
```

Справедливости ради следует отметить, что возможностей у класса *Console* в C# несколько больше, чем у аналогичного класса *TextWindow* в *СБ*.

Класс *Desktop* (*Рабочий стол*)

Поскольку окно приложения – и консольного, и графического – «лежит» на *Рабочем столе*, то давайте сначала познакомимся с классом **Desktop**, который в *СБ* как раз и описывает свойства *Рабочего стола*.

Класс *Desktop* - это один из самых «бедных» классов в *СБ* – у него всего два свойства и один метод.

Если вы взглянете на *Рабочий стол*, то сразу поймёте, что его самые главные свойства – это *ширина, высота* (конечно, при этом имеются в виду не сантиметры, что важно для нас, а *пиксели*, с которыми компьютер дружит больше) и *фоновая картинка* (почему-то называемая обоями!).

Ширина Рабочего стола хранится в свойстве

```
Desktop.Width,
```

а *высота* - в свойстве

```
Desktop.Height.
```

Оба свойства предназначены только для *чтения*, это значит, что вы *можете узнать* размеры стола, но *не можете их изменить*. Казалось бы, размеры *Рабочего стола* вообще изменить нельзя, поскольку он существует только на экране монитора, а тот «не резиновый». На самом деле это не так: мы измеряем экран в *пикселях*, но пиксели можно сделать и больше и меньше – в зависимости от *разрешения* экрана, а его изменить очень даже можно.

Метод

```
Desktop.SetWallPaper(fileName)
```

заменит картинку на *Рабочем столе* той, что вы указали в скобках. Файл с картинкой может находиться на диске вашего ком-

пьютера или на любом сайте (в этом случае нужно указать его полный адрес).

Поскольку обои меняют нечасто, то лучше это сделать более естественным способом!

Класс *TextWindow* (*Текстовое окно*)

Главное назначение *Текстового окна* – ввод и вывод текстовой информации в консольных приложениях.

Чтобы текстовое окно *появилось* на экране, нужно выполнить его метод

```
TextWindow.Show()
```

Если в программе имеются обращения к свойствам и методам *Текстового окна*, то *этот* метод использовать необязательно.

Противоположное действие оказывает метод

```
TextWindow.Hide()
```

Он делает окно *невидимым*, но приложение продолжает работать.

В этом легко убедиться, если добавить в программу строку

```
Program.Delay(10000)
```

Окно исчезнет с экрана, но приложение закроется только через 10 секунд.

По умолчанию *размеры* текстового окна равны примерно 680 х 340 пикселей (Рис. 6.2).

Программно *изменить размеры* окна нельзя, но в работающем приложении достаточно нажать среднюю кнопку в заголовке окна, чтобы его высота стала равной высоте *Рабочего стола*.

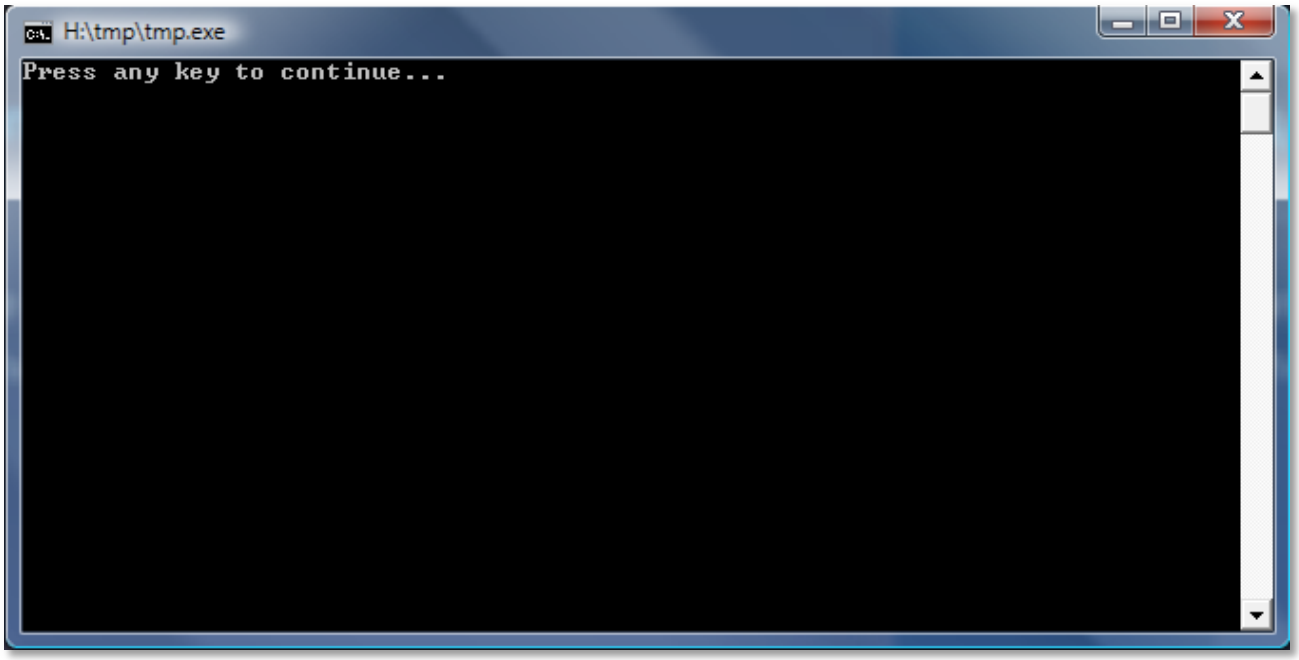


Рис. 6.2. Стандартное текстовое окно

Тот же самый эффект вы получите, если дважды щёлкнете по его заголовку, - окно вытянется во весь экран; повторный двойной щелчок вернёт ему прежние размеры.

Изменить размеры текстового окна также можно, потянув его мышкой за угол или за границу. При этом максимальная ширина окна остаётся постоянной, даже если кошка потянет за мышку, а Жучка - за кошку...

Такое «упрямство» *текстового окна* объясняется тем, что традиционно консоль имеет ширину в 80 символов. При ширине символов в *текстовом окне* 8 пикселей мы как раз и получаем 640 пикселей.

Текстовое окно всегда появляется в левом верхнем углу *Рабочего стола*, а свойства

```
TextWindow.Left
TextWindow.Top
```

позволяют задавать положение окна на экране непосредственно в приложении. Например, так:

```
TextWindow.Left = 200
TextWindow.Top = 300
```

Первое свойство задаёт расстояние в пикселях от левой границы окна до левой границы *Рабочего стола*. Второе свойство - от верхней границы окна до верхней границы *Рабочего стола*.

Начало координат на экране расположено в *верхнем левом углу*, а ось *Y* направлена *вниз* (Рис. 6.3).

Рис. 6.3. Окно приложения на *Рабочем столе*

Часто требуется установить окно приложения *по центру Рабочего стола*. Если мы знаем размеры *Рабочего стола*, то с *графическим окном* приложения проблем не будет:

```
GraphicsWindow.Left=  (- GraphicsWindow.Width) / 2
GraphicsWindow.Top = (Desktop.Height-GraphicsWindow.Height)/2
```

А вот у *Текстового окна* нет свойств *Width* и *Height*, как у *Графического окна*, но мы знаем, что по умолчанию размеры текстового окна равны примерно 680 x 340 пикселей, поэтому

```
TextWindow.Left= (Desktop.Width - 680) / 2
TextWindow.Top = (Desktop.Height - 340) / 2
```

Но если в новой версии *СБ* размеры *текстового окна* по умолчанию изменятся, то оно окажется совсем не там, где вы рассчитывали его увидеть.

Еще более опасно непосредственно задавать положение текстового окна так, как в примере:

```
TextWindow.Left = 200
TextWindow.Top = 300
```

На вашем *Рабочем столе текстовое окно* удобно расположится в самом его центре, но если вашу программу запустить на компьютере другого пользователя, с более скромным монитором, то оно может вообще скрыться за «горизонтом». Измените первую строчку:

```
TextWindow.Left = 2000
```

и запустите программу. Окно исчезнет с экрана, и хотя программа будет работать правильно, вы её не увидите, что, конечно, сильно огорчит кого угодно.

Эти свойства доступны и для записи, и для чтения, то есть вы всегда можете узнать истинные *координаты текстового окна* (точнее, его верхнего левого угла):

```
Left =  TextWindow.Left
Top =  TextWindow.Top
```

Если вы не позаботились об этом сами, то в *Заголовке окна* будет напечатано не название программы, как это принято, а полный *путь* к выполняемому файлу приложения, что неплохо при отладке, но совсем не обязательно знать пользователю вашей программы. Впрочем, вы легко можете изменить *Заголовок окна*, присвоив свойству нужное значение (Рис. 6.4):

```
TextWindow.Title = «Моя программа»
```

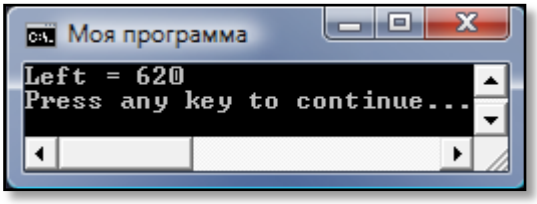


Рис. 6.4. Вот теперь заголовок «правильный»!

Установив окно на экране по своему желанию и украсив его заголовком, давайте заставим его работать, то есть сообщать пользователю важную информацию. Для этого предназначены методы

```
TextWindow.WriteLine(Данные)
TextWindow.Write(Данные)
```

Они выводят *Данные* (любые строковые и числовые выражения) в *текущую* строку текстового окна. Причем первый метод затем переводит вывод информации на *следующую* строку (Рис. 6.5).

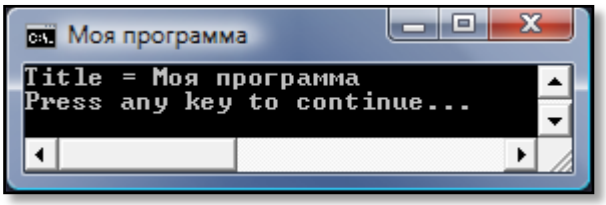


Рис. 6.5. Вывод информации методом *WriteLine*

А второй метод продолжает вывод в *ту же самую* строку (Рис. 6.6).

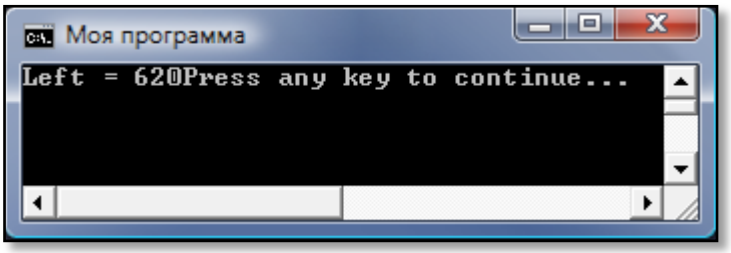


Рис. 6.6. Вывод информации методом *Write*

Чтобы сделать приложение по-настоящему *интерактивным*, нужно научить его *реагировать* на действия пользователя.

Метод

```
TextWindow.Read()
```

ждёт, пока пользователь наберёт какую-нибудь строку с клавиатуры и нажмёт клавишу *ВВОД*, после чего возвращает эту строку в программу. Её можно присвоить переменной и использовать в программе.

Наберите такие строчки:

```
TextWindow.Write("Введите строку и нажмите клавишу ВВОД! ")
str = TextWindow.Read()
TextWindow.WriteLine("str = " + str)
```

Здесь нам пригодился метод *Write*, чтобы объяснить пользователю, каких действий от него ожидает программа. Когда пользователь откликнется на просьбу и что-нибудь напишет в этой строке, а потом нажмёт клавишу *ВВОД*, значение строки будет присвоено переменной *str*, а затем напечатано в следующей строке методом *WriteLine* (Рис. 6.7).

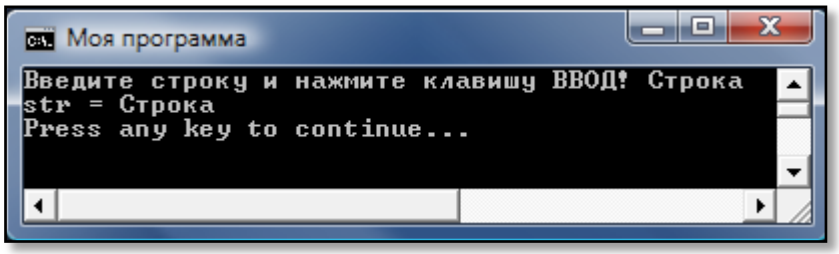


Рис. 6.7. Ввод информации методом *Read*

Надпись *Press any key to continue...* означает *Нажмите любую клавишу для продолжения*.

Метод

```
TextWindow.ReadKey()
```

возвращает *один символ*, соответствующий нажатой клавише (Рис. 6.8).

```
TextWindow.WriteLine("Нажмите символьную клавишу!")
str = TextWindow.ReadKey()
```

```
TextWindow.WriteLine("str = " + str)
```

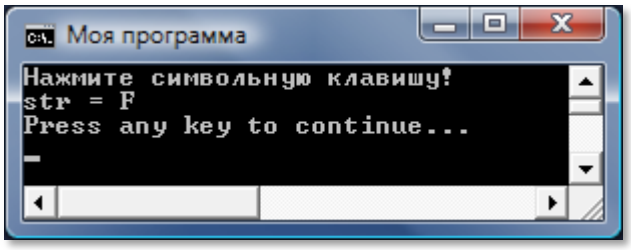


Рис. 6.8. Ввод информации методом *ReadKey*

Русские буквы не допускаются. Чтобы ввести ЗАГЛАВНУЮ латинскую букву, удерживайте нажатой клавишу *Shift*.

Метод

```
TextWindow.ReadNumber()
```

действует аналогично методу *Read*, но возвращает не строку, а *число*, поэтому нажатия на нецифровые клавиши игнорируются. Эта особенность метода *ReadNumber* облегчает работу программиста, так как контролирует ввод числа пользователем (Рис. 6.9):

```
TextWindow.Write("Введите число и нажмите клавишу ВВОД! ")
num = TextWindow.ReadNumber()
TextWindow.WriteLine("num = " + num)
```

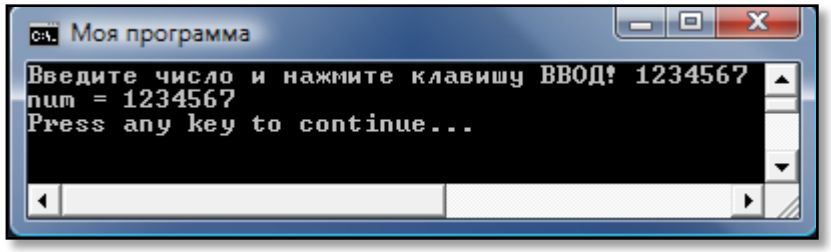


Рис. 6.9. Ввод информации методом *ReadNumber*

Свойства

```
TextWindow.CursorLeft и
TextWindow.CursorTop
```

Устанавливают текстовый *курсор* (текущий вывод) в заданную позицию *текстового окна*. При этом координаты задаются не в

пикселях, а в *символах*. Для самого первого символа в окне координаты равны *0, 0*. После нулевого символа идёт первый и так далее до 79-ого. Строки нумеруются сверху вниз. С помощью новой программы мы легко разбросаем цифры по всему *текстовому окну* (Рис. 6.10):

```
For i= 1 To 20
  x= Math.GetRandomNumber(20)
  TextWindow.CursorLeft=x
   y= Math.GetRandomNumber(20)
   TextWindow.CursorTop=y
   TextWindow.WriteLine(Math.GetRandomNumber(10)-1)
 EndFor

TextWindow.CursorLeft= 0
TextWindow.CursorTop= 21
```

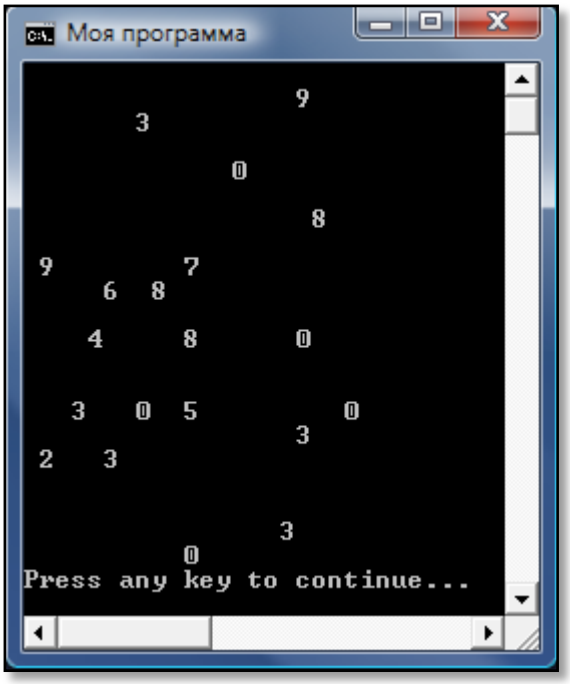


Рис. 6.10. Получилось!

И нам осталось поговорить о самом прекрасном, что есть в *Текстовом окне*, – о ***цветах.***

Метод

```
TextWindow.Clear()
```

просто *стирает* всю информацию с экрана, окрашивая его в чёрный цвет (Рис. 6.11, слева). Позиция вывода устанавливается в начало первой строки.

Если вы хотите окрасить фон *текстового окна* в цвет, отличный от чёрного, поступайте так (Рис. 6.11, справа):

```
TextWindow.BackgroundColor = "Blue"
TextWindow.Clear()
```

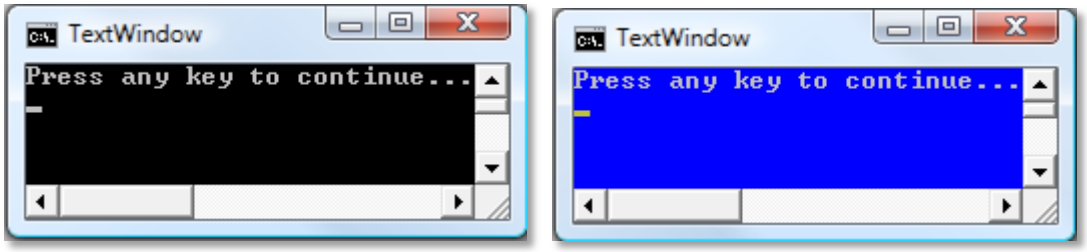


Рис. 6.11. Чистое *текстовое окно* – **чёрного** и **синего** цвета

Последние два свойства *Текстового окна* расцвечивают выводимые *символы* во все цвета радуги.

Свойство

```
TextWindow.ForegroundColor
```

устанавливает текущий *цвет символов*, а свойство

```
TextWindow.BackgroundColor
```

цвет фона, на котором эти символы печатаются.

Добавим пару строк к предыдущей программе:

```
TextWindow.ForegroundColor = "Yellow"

For i= 1 To 20
   = Math.GetRandomNumber(20)
   TextWindow.CursorLeft=x
   y= Math.GetRandomNumber(20)
   TextWindow.CursorTop=y
   TextWindow.WriteLine(Math.GetRandomNumber(10)-1)
 EndFor
```

```
TextWindow.BackgroundColor = "Red"
TextWindow.CursorLeft= 0
TextWindow.CursorTop= 21
```

Теперь картина получилась более живописная (Рис. 6.12)!

Чтобы вам легче было расцвечивать числа и слова, вот вам **таблица** всех цветов, которыми вы можете пользоваться в *Текстовом окне* (Рис. 6.13).

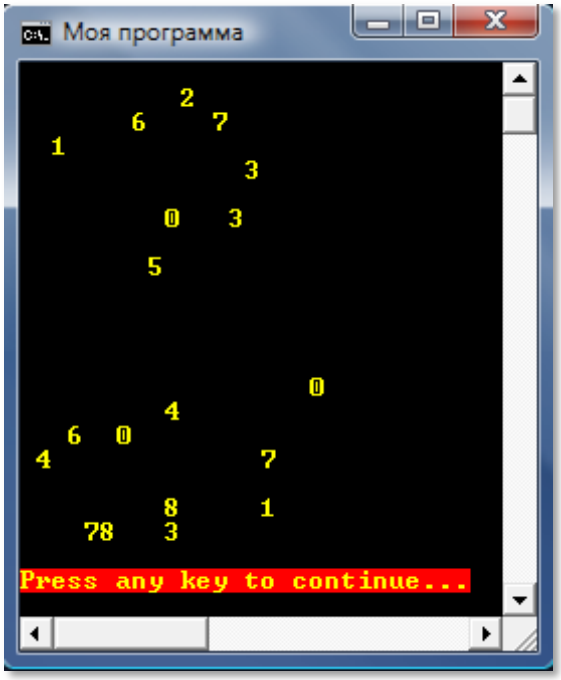


Рис. 6.12. Лепота!

Есть хорошие *способы* (*мнемонические приёмы*), помогающие запомнить последовательность цветов в спектре (или в радуге).

1. **Как Однажды Жак-Звонарь Городской Сломал Фонарь**.

2. **Каждый Охотник Желает Знать, Где Сидит Фазан**.

Мне больше нравится первый – он «складный».

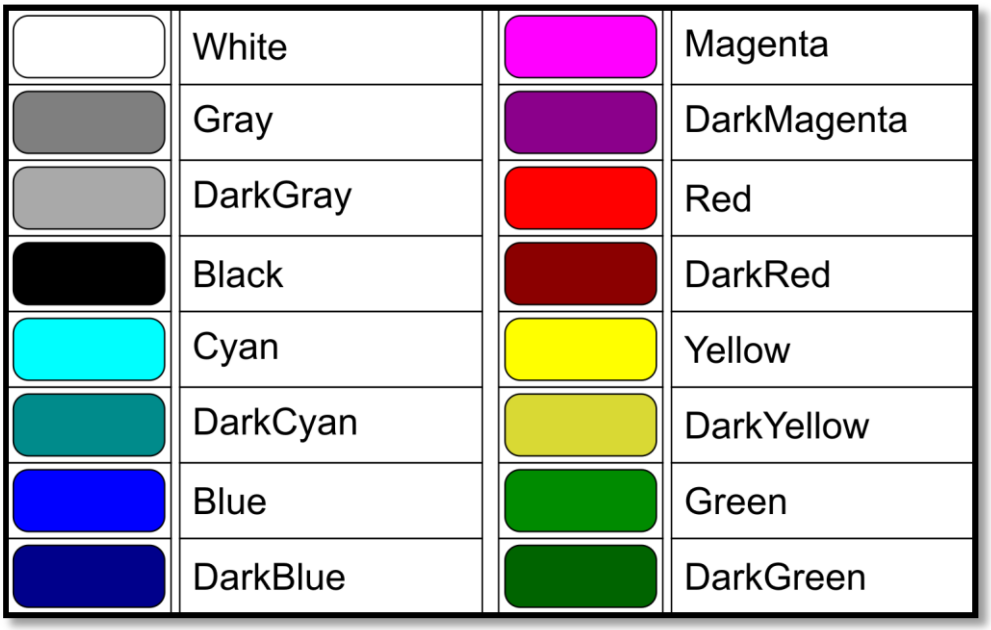


Рис. 6.13. Цветовая палитра *Текстового окна*

Урок 7. Цикл *While*

В программе *Rome* мы пользовались для формирования строки с римским числом *циклом While*, поэтому немного поговорим о **циклах**. Очень часто в программах встречаются ситуации, когда одни и те же действия нужно повторить многократно. Для таких случаев в языке *СБ* припасены два оператора циклов – ***For*** и ***While***.

While называется оператором цикла с *предусловием*. В других языках программирования есть ещё цикл *Repeat* (или *Do-While*) - *с постусловием*. Он отличается от цикла *While* только тем, что проверка завершения цикла проводится в конце цикла, а не в его начале. Это значит, что цикл *Repeat* всегда выполняется хотя бы один раз, а цикл *While* может вообще не выполняться ни разу. Поскольку различаются эти циклы очень мало, мы вполне сможем обойтись циклом *While*. Ещё один цикл – *For* - мы разберём на следующем уроке.

Примеры циклов мы можем найти и в повседневной жизни:
- времена года (циклически сменяют друг друга весна – лето – осень - зима),
- время суток (утро – день – вечер - ночь),
- фазы Луны,
- вращение планет вокруг Солнца и Солнечной системы вокруг центра Галактики,
- спортивные состязания,
- учебный год,
- режим дня,
- дыхание и кровообращение,
- часы,
- биоритмы активности человека – можно даже утверждать, что уникальны именно неповторяющиеся события, а не циклические.

Цикл **While** записывается так:

```
While условие_выполнения_цикла   ← Заголовок цикла
  оператор1                      ← Тело цикла
  оператор2
 . . .
  операторN
EndWhile                         ← Конец оператора цикла
```

А *работает* так:

1. Проверяется условие выполнения цикла.

2. Если оно *ложно* (не выполняется), то цикл заканчивается, и программа переходит на строчку, следующую за ключевым словом *EndWhile*.

3. Если оно *истинно*, то последовательно выполняются операторы *оператор1, оператор2, … , операторN*, которые называются *телом цикла,* и управление передаётся в *п.1*.

Поскольку условие проверяется *до* тела цикла, то операторы могут вообще не выполняться – если выражение с самого начала ложно.

Как мы видим, цикл заканчивается тогда, когда условие станет ложным. А почему *изменяется* значение *условного выражения* в заголовке цикла? – Да потому, что изменились значения переменных, входящих в это условие. И это может произойти только в теле цикла. Отсюда грустный вывод: цикл *While* может вообще никогда не закончиться, и программа зациклится навечно, что очень часто и бывает!

Закомментируйте (для этого поставьте в начале строки знак комментария - апостроф) строку *n= n+1*:

```
  while number>0
    'n= n+1
    while num[n] <= number
      sNumber= sNumber + sNum[n]
      number= number - num[n]
    EndWhile
  EndWhile
endif
```

Запустите программу, введите любое число и нажмите клавишу *ВВОД* – программа зациклится, и римское число вы не увидите никогда. Не томите компьютер и закройте программу.

Всегда следите за изменением *переменной* (или переменных), входящей в условие выполнения цикла! Если она не изменяется, то цикл либо не выполнится ни одного раза, либо будет выполняться бесконечно! Обратите внимание на отличие цикла *While* от цикла *For*, в котором, наоборот, переменную цикла изменять нельзя.

Применяйте цикл *For*, если число повторений известно заранее, а в противном случае лучше подойдёт цикл *While*.

Не забывайте ставить ключевое слово *EndWhile* в конце цикла!

СБ не позволяет записывать в строке больше одного оператора, поэтому запись *оператор1 оператор2* приведёт к ошибке.

Оператор присваивания

Самый «востребованный» оператор в любом языке программирования – это, безусловно, **оператор присваивания**.

Он записывается так:

переменная = выражение

Переменная - это простая переменная, элемент массива или свойство объекта. В результате выполнения этого оператора значение переменной станет равным значению выражения. Выражением может быть другая переменная, константа, константное выражение, свойство или метод объекта. Сложные выражения могут быть составлены из переменных и констант, соединённых знаками арифметических операций.

Например:

Оператор присваивания	Результат присваивания, значение переменной
Width = 100	100
Height = Width + 20	120
Mas[12] = «Двенадцать»	Двенадцать
Mas[13] = «Двенадцать» + 1	Двенадцать1
String = Mas[13] + Height	Двенадцать1120
GraphicsWindow.Width = 640 Width = GraphicsWindow.Width	640
max = Math.Max(1,11)	11

Различие между оператором присваивания и знаком равенства в математике можно показать на примере. Очень часто в программах можно встретить вот такие «ужасные» выражения:

```
x = 10
x = x +1
```

Первая строка не вызывает никаких возражений: согласно правилам алгебры, неизвестная величина *x* равняется 10. Если аналогично поступить со второй строкой, то алгебраические выкладки приведут к такому выражению:

```
0 = 1
```

Это верный путь к двойке! Но с точки зрения программирования, вторая строка абсолютно правильная! *Сначала* вычисляется значение выражения *справа* от знака присваивания, а *затем* оно *присваивается* переменной *слева* от знака присваивания. В нашем случае: в первой строке переменная *x* получила значение 10. Вычисляем значение выражения во второй строке: 10 + 1 = 11. После выполнения оператора присваивания переменная *x* будет равна 11.

В языке программирования *паскаль* знак присваивания записывается иначе - **:=**. Так, конечно, более понятно его назначение. Однако в большинстве языков программирования знак присваивания совпадает со знаком равенства.

Тот же самый знак равенства в *СБ* применяется и в условном операторе *IF* для *сравнения* двух выражений, что создаёт ещё большую путаницу. В языках *C++, C#* и некоторых других на этот случай знак равенства *дублируется* - **==**. А вот в бейсике один и тот же знак используют и в операторе присваивания, и в операторе сравнения.

Отладка программ

Лови жучков!
Программистская пословица

Отладка программы – самый трудный и ответственный момент в программировании. В самом деле, если хотя бы одна ошибка останется в готовом приложении, то оно иногда будет работать неправильно!

Вы, наверное, помните, сколько неприятностей доставил пользователям 2000-й год. А всё потому, что во многих старых программах для обозначения текущего года использовались только две последние цифры, то есть 1999-й год считался 99-м. После 99-го, естественно, наступил год 00. В некоторых программах он трактовался правильно – как 2000-й, в других

неправильно – как 1900-й. Эта ошибка объясняется тем, что в начале компьютерной эры программисты даже и не задумывались о том, что их программы будут использовать и в 2000-м году.

Другой курьёзный случай. Клиент одного банка ежемесячно получал требование: погасить долг в течение месяца. Однако сумма долга составляла 0 долларов, 0 центов. Конечно, оплачивать такой счет не имело никакого смысла. Этот пресловутый клиент именно так и думал. И каждый месяц получал новое предупреждение. Так продолжалось до тех пор, пока он не оплатил «долг», выслав банку чек на означенную сумму. А дело тут в том, что компьтерная программа в банке не учитывала, что нулевой долг таковым не является.

Ошибки, которые проявляются только при работе уже готовой программы принято называть словом *баг* – от английского *bug* – *жук*.

Этот термин возник в 1945 году, когда небольшая бабочка, незаконно проникшая в вычислительную машину, замкнула реле, чем вызвала ошибки в её работе. С тех пор слово *debugging*, то есть *извлечение скрытых багов*, означает отладку программы.

Все ошибки в программе можно условно разделить на синтаксические и логические.

Синтаксические (грамматические) ошибки появляются обычно из-за невнимательности или простой описки. Их нетрудно «обезвредить», поскольку *СБ* в процессе компиляции легко находит их и сообщает об этом в *Окне отладки*, которое появляется под окном *Редактора кода* (Рис. 7.1).

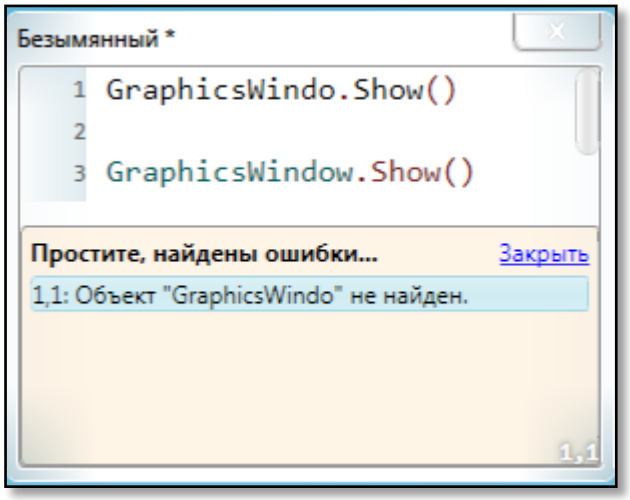


Рис. 7.1. Отладчик нашёл ошибку

Достаточно щёлкнуть по сообщению, и курсор установится в то место исходного кода, где найдена ошибка.

Номер строки и первого ошибочного символа указываются слева от описания ошибки (Рис. 7.2).

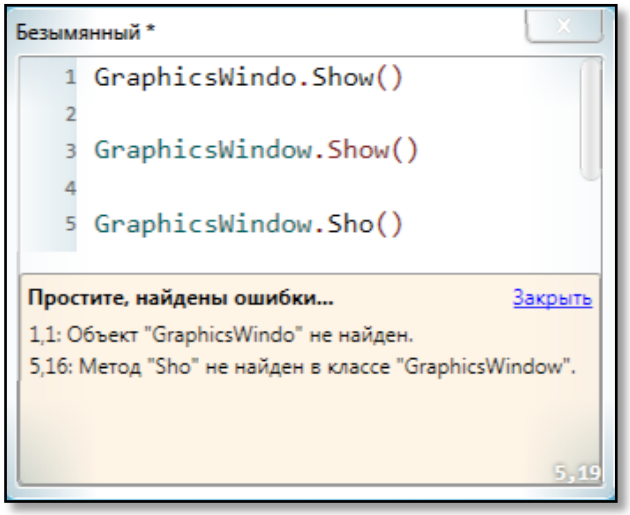


Рис. 7.2. Уточнение ошибки

Некоторые ошибки легко распознать благодаря *окрашиванию* элементов программы в разные цвета. Например, названия объектов **сине-зелёные**, их свойств и методов – **коричневые**, а имена переменных – **чёрные**:

```
TextWindow.WriteLine(str)
```

На рисунке 7.2 видно, что название объекта в первой строке *чёрного* цвета, значит, оно явно записано неправильно. То же самое касается и названия метода в пятой строке. Таким образом, при должной внимательности можно легко найти синтаксические ошибки.

Обычные синтаксические ошибки – неверно написанные идентификаторы, непарные скобки и кавычки (Рис. 7.3).

Логические ошибки возникают вследствие неправильного алгоритма или неверной записи операторов программы, которые компилятор *СБ* найти не может, потому что они не противоречат правилам языка. Логические ошибки найти очень трудно, а сам поиск таких ошибок называется *отладкой программы*.

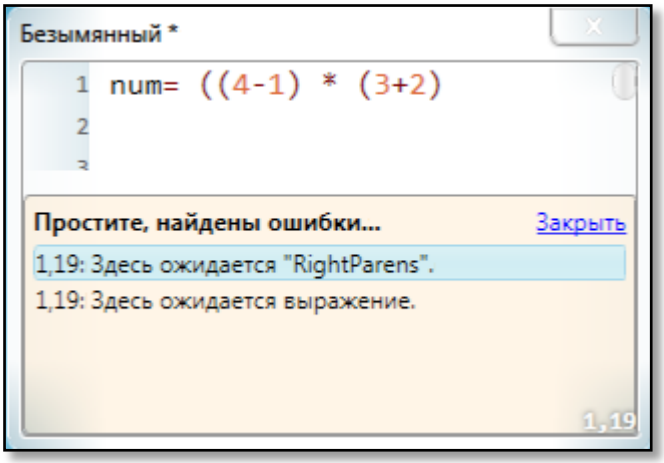


Рис. 7.3. Типичные синтаксические ошибки

В первую очередь, при написании программы следует помнить, что после добавления логически законченного фрагмента кода, сразу же следует проверить, правильно ли он работает. Никогда не пишите большие куски кода, иначе найти ошибки будет очень трудно.

Отлаженный промежуточный вариант программы всегда сохраняйте на диске, добавляя к названию программ номер версии: *MyProgram1, MyProgram2* и так далее. Конечно, последние изменения в тексте легко удалить с помощью кнопки *Отмена*, но сможете ли вы всегда вернуться к тому

коду, который предшествовал ошибке? – Поэтому не жалейте времени на сохранение отлаженного кода!

Поскольку в *СБ* нет «настоящего» отладчика, то в критических местах программы нужно вставлять операторы, которые выводят нужную информацию в окно программы.

И здесь нам как нельзя кстати пригодится *текстовое окно*. Действительно, зачем нам портить окно графической программы, когда у нас в распоряжении имеется *текстовое окно*, в которое очень удобно выводить отладочную информацию.

Напишем программу, выводящую разноцветные кружки в *Графическое окно*:

```
GraphicsWindow.Title = "Debug"

n= 0
m= 6
While (n < 10) And (m > 5)
  Draw()
EndWhile

Sub Draw
  height=GraphicsWindow.Height
  width= GraphicsWindow.Width
  radius=Math.GetRandomNumber(2)
  x = Math.GetRandomNumber(width-2*radius)
  y = Math.GetRandomNumber(height-2*radius-40) + 40
  r= Math.GetRandomNumber(163) + 92
  g= Math.GetRandomNumber(163) + 92
  b= Math.GetRandomNumber(163) + 92
  clr= GraphicsWindow.GetColorFromRGB(r,g,b)
  GraphicsWindow.BrushColor=clr
  GraphicsWindow.FillEllipse(x, y, 2*radius,2*radius)
EndSub
```

Если вы запустите программу, то она будет выполняться вечно, хотя мы в заголовке цикла записали условие окончания цикла:

```
(n < 10) And (m > 5)
```

Добавим в тело оператора цикла отладочную строку:

```
While (n < 10) And (m > 5)
  Draw()
  TextWindow.WriteLine("n = " + n + " m= " + m)
EndWhile
```

и снова запустим программу.

Пока в *графическом окне* бесконечно появляются точки, текстовое окно сообщает нам значения переменных *n* и *m*, которые опеределяют условие завершения цикла. И они *не изменяются*! Так бывает довольно часто – мы забыли проследить за переменными цикла (Рис. 7.4).

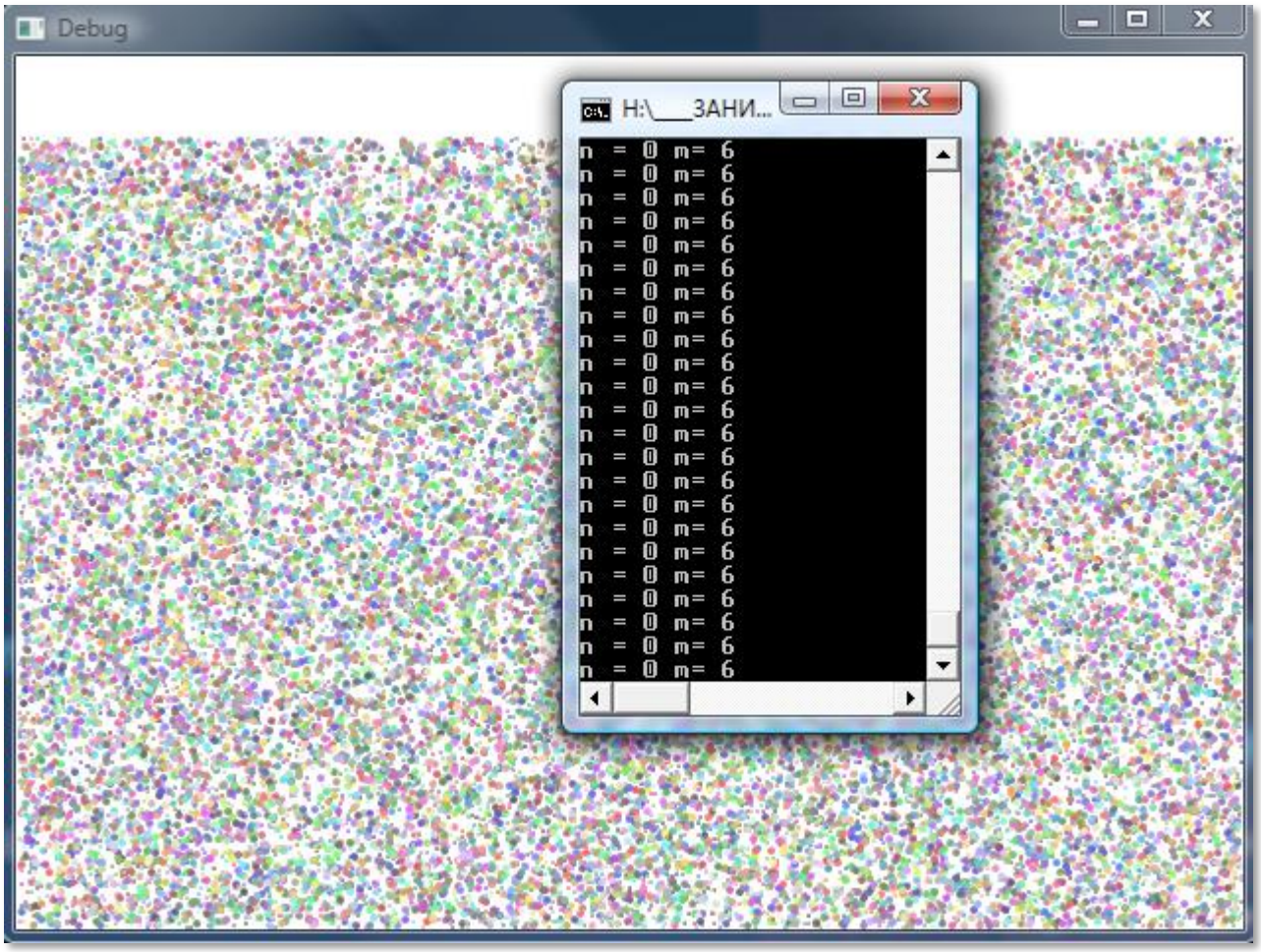


Рис. 7.4. Отладка цикла

Добавим ещё пару строк, в которых переменные цикла изменяют свои значения:

```
While (n < 10) And (m > 5)
  Draw()
  n= n+1
  m= Math.GetRandomNumber(20)
  TextWindow.WriteLine("n = " + n + " m= " + m)
EndWhile
```

Теперь мы видим, что цикл прекращается, если значение переменной *n* достигает 10, либо когда случайное значение переменной *m* становится меньшим или равным пяти (Рис. 7.5).

Так как именно эти условия мы и записали в заголовке цикла, то теперь эта часть кода работает верно.

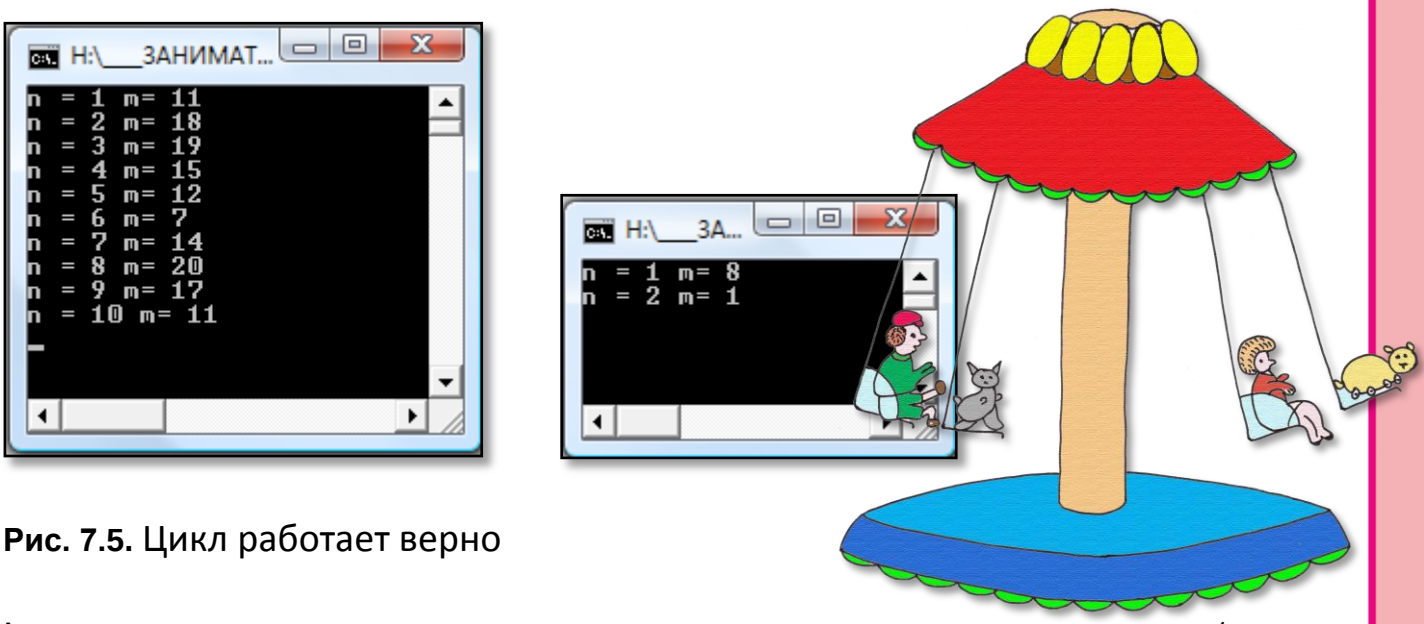


Рис. 7.5. Цикл работает верно

Аналогично вы можете контролировать значения любых переменных программы, вставляя отладочные операторы в те места кода, где переменные должны изменять свои значения.

После отлова всех жучков отладочные операторы можно удалить или закомментировать. Второй вариант предпочтительнее, так как исходный код может ещё вам пригодиться при написания другой программы, а в ней также могут понадобиться отладочные операторы. Их можно и снова написать, но раскомментировать строку гораздо быстрее и удобнее.

Урок 8. Признаки делимости

Амёбы умножаются делением.

Математический парадокс

Целые числа и их свойства изучает *теория чисел*, или *высшая арифметика* (есть и такая наука!).

В занимательных задачах (да и в жизни тоже) нередко нужно быстро определить, делится ли одно число на другое или нет. При этом сам результат деления неважен. У каждого натурального числа имеется, по крайней мере, два делителя - это 1 и само число. Если других делителей нет, то число называется *простым*. К ним мы вернёмся на следующем уроке, а сейчас вспомним *признаки* (то есть правила) *делимости*.

Признак делимости на 2

Самый простой признак: *число делится на 2 только тогда, когда его последняя цифра равна 0, 2, 4, 6 или 8*. Если число делится на 2, то оно называется *чётным*, если не делится - *нечётным*. Примеры чётных чисел: 2012, 92, 4, 76, 58.

Иногда этот признак формулируют проще: *число делится на 2 тогда и только тогда, когда его последняя цифра чётная.*

Признак делимости на 3

Число делится на 3 тогда и только тогда, когда сумма его цифр делится на 3. Например, число 2010 кратно 3, поскольку сумма его цифр равна 3: 2 + 1 = 3 (при подсчете нули не учитываем).

Если сумма цифр выражается не однозначным числом, то следует найти сумму *его* цифр. Если в результатае сложения цифр получится одно из чисел 3, 6 или 9, то число делится на 3. В противном случае – не делится. Например, сумма цифр числа 123456789 равняется 45. Число двузначное – опять находим сумму цифр: 4 + 5 = 9. Получили *девятку* – значит, исходное число 123456789 делится на три.

Признак делимости на 4

Очевидно, что числа кратные четырём, должны быть чётными. Но этого мало, поэтому мы оставляем от числа только *последние две цифры* и рассматриваем получившееся двузначное число.

Если число двузначное, то ничего отбрасывать не нужно. А если однозначное, то достаточно вспомнить таблицу умножения.

Если это двузначное число делится на 4, то и всё число также делится на четыре.

Почему достаточно рассмотреть только последние две цифры числа? – На этот вопрос легко ответить, если вспомнить, что сотня делится на 4 без остатка. Естественно, любое число сотен также разделится на 4, поэтому разряды сотен, тысяч и так далее в проверяемом числе можно не учитывать.

Можно упростить проверку. Сложите число десятков с половиной единиц. Если сумма четная, то исходное число делится на 4, в противном случае не делится. Опять проверим год 2010. Число из последних двух цифр равно 10. Оно на 4 не делится, значит 2010 не кратно четырем. Возьмем другое число - 4567896. Оставляем две цифры - 96. Складываем 9 с половиной от 6, то есть с тройкой и получаем 12. Это число кратно четырём, значит, число 4567896 делится на 4.

Признак делимости на 5

Это правило очень похоже на признак делимости на двойку. *Число делится на 5, если оно оканчивается на 0 или 5.*

Признак делимости на 6

Число делится на 6, если *одновременно* выполняются *признаки делимости на 2 и 3*.

Признак делимости на 7

Хорошего признака делимости чисел на 7 не существует, зато

имеется немало достаточно сложных и запутанных. Из них мы выберем один – самый простой и «универсальный».

Разбиваем заданное число, начиная с конца, на группы, состоящие из трёх цифр. Например, если мы проверяем число 4567896, то получим три группы цифр:

4 567 896

Кстати говоря, так в книгах зачастую и печатают длинные числа, чтобы легче было распознать разряды сотен, тысяч и так далее.

Теперь первое (считаем сзади!) число мы берём со знаком плюс, второе со знаком минус, третье – снова о знаком плюс. То есть знаки плюс и минус *чередуются*. Составлем из чисел с их знаками арифметическое выражение и вычисляем его значение:

896-567+4 = 333

Если результат делится на 7, то и всё число также делится на 7. В противном случае не делится.

В нашем примере число 333 на 7 не делится, значит, это вывод относится и к исходному числу 4567896.

Признак делимости на 8

Число делится на 8, если оно чётное. Кроме того, *необходимо, чтобы число, составленное из трёх последних цифр, делилось на 8.* Так как делить трёхзначное число на 8 тоже нелегко, то можно воспользоваться тем же приемом, что и в *признаке делимости на 4.*

Тысяча делится на 8 без остатка. Любое число тысяч также разделится на 8, поэтому разряды тысяч и далее в проверяемом числе можно не учитывать.

Рассмотрим три последние цифры. К числу, образованному первыми двумя цифрами, добавьте половину единиц, а затем к числу десятков добавьте половину единиц получившегося числа. Если результат чётное число, то исходное число делится на 8.

Проясним этот алгоритм на *примере*. Начнём с того же числа 2010. Последние три цифры дают двузначное число 10, которое на 8 не делится. Следовательно, не делится и число года. Возьмём другое число - 123457928. Оставляем для проверки трёхзначное число 928. Число из первых двух цифр равно 92. Складываем его с половиной единиц - 4 - и получаем 96. Дальше действуем, как в *признаке делимости на 4*: 9 + 3 = 12. Это число кратно двум, поэтому всё число 123457928 делится на 8.

Признак делимости на 9

Этот признак напоминает правило для тройки. *Число делится на 9 тогда и только тогда, когда сумма его цифр делится на 9.* Раньше мы установили, что число 2010 делится на 3 и сумма его цифр также равна трём. Поэтому, согласно признаку делимости, на 9 оно не делится.

Если сумма цифр выражается не однозначным числом, то следует найти сумму его цифр. То есть действовать так же, как и в признаке делимости на 3.

Признак делимости на 10

Еще проще, чем признак делимости на 5. *Число делится на 10 тогда и только тогда, когда оно заканчивается на 0.* Например, число 2010 делится на 10.

Признак делимости на 11

Самое любопытное правило; не все его знают, но оно помогает очень просто определить, делится ли, например, номер автобусного билета на 11.

Чтобы узнать, делится ли число на 11, нужно подсчитать отдельно сумму цифр, стоящих на *нечётных* и *чётных* местах в исходном числе. Если они равны, то число кратно 11. В противном случае нужно из первой суммы вычесть вторую. Если разность делится на 11, то и всё число делится на 11.

Например, число **123453** делится на 11, так как 1 + 3 + 5 = 2 + 4 + 3 = 9. А число **123456** не делится (проверьте сами!).

Другой признак делимости на 11 полностью совпадает с признаком делимости на 7, но делить сумму чисел нужно на 11.

Признак делимости на 12

Число делится на 12 тогда и только тогда, когда *одновременно выполняются признаки делимости на 3 и 4*.

Признак делимости на 13

Признак делимости на 13 тот же самый, что и для чисел 7 и 11 (второй способ), но делить сумму чисел нужно на 13. Поэтому я недаром назвал этот признак универсальным.

Интересно, что наименьшее число, которое одновременно делится на 7, 11 и 13, равняется 7 x 11 x 13 = 1001, то есть сказочному числу арабских ночей.

Интересные математические фокусы, связанные с признаками делимости, вы найдёте в книге Мартина Гарднера *Математические досуги*, глава 19 **[9]**.

Делится - не делится?

Компьютер должен считать,
а человек - думать.

Программистская пословица

Мы вспомнили признаки делимости чисел, без которых человеку обойтись трудно, а вот компьютеру они совсем не нужны, потому что он для того и сделан, чтобы считать. И надо сказать, делает он это охотно и быстро.

Давайте затеем новый проект. И начнём мы его не с пустого места, а загрузим исходный текст программы *Rome2* в *СБ* и сохра-

ним его в новой папке под именем **Delimost** (получилось коряво, но вы можете придумать и другое).

Почему мы действуем столь дерзко? - А мы всегда так будем поступать - чтобы не писать каждую программу снова да ладом. Если у нас в закромах и в сусеках уже есть похожая программа, то мы запросто можем её использовать как заготовку для следующей программы.

Итак, программа-шаблон загружена, начинаем её приспосабливать под свои нужды. Поскольку делить одно число на другое значительно проще, чем заниматься переводом чисел на чуждый нам язык, то из всех *переменных* мы оставим только две:

```
'ПРОГРАММА ДЛЯ ПРОВЕРКИ ДЕЛИМОСТИ ЧИСЕЛ

'variables
number= 0
sNumber=""
```

На что сразу следует обратить внимание: в первой же строке буквами ВО ВЕСЬ РОСТ объясняется *назначение программы*. Вы скажете, что всё и так понятно, зачем ещё бухгалтерию разводить? - Отвечу: сегодня понятно, а через месяц, когда у вас на диске скопится ворох таких программ, вспомните ли вы, для чего написали каждую из них? - Не вспомните! А если вы с кем-то поделитесь своими программами - как эти несчастные должны ими пользоваться? Вот так, по лени, вы можете потерять уважение своих товарищей.

Идем дальше. После апострофа записан *комментарий 'variables*. Вы можете вообще не писать в программе комментариев и напрочь забыть об апострофе, но тогда вы никогда не станете даже плохим программистом, а ваши товарищи отвернутся от вас навсегда.

Никогда не жалейте времени на *комментарии*! Конечно, не нужно объяснять каждую строчку программы, но *смысл* важных действий пояснять необходимо. Причины этой «писанины» изложены выше: через некоторое время вы и сами не

сможете разобраться в своей же собственной программе и опозоритесь перед друзьями, когда будете им объяснять, как ловко работает ваша программа.

Далее нам встретится ещё переменная цикла *i*. Её тоже можно внести в список переменных программы, но делать это необязательно, и вот почему: без циклов не обходится практически ни одна программа. В *СБ* все переменные - *глобальные*, но вы должны использовать переменные цикла только по назначению, чтобы не испортить программу. Другие *вспомогательные* переменные также допустимо не указывать в списке переменных. Такие переменные мы будем обозначать *одной* буквой (можно добавлять ещё цифру или букву), чтобы отличать их от более важных переменных, которым необходимо давать более *длинные* и обязательно *осмысленные* имена, чтобы не запутаться в их назначении.

Имена переменных в *СБ* могут быть очень длинными, например, *qwertzuiopasdfghjklyxcvbnm1234567890*, поэтому не жалейте букв! Если идентификатор состоит из нескольких слов, то их можно отделять друг от друга ПРОПИСНЫМИ буквами или знаком подчеркивания: *sobakaGryzlaKost, sobaka_gryzla_kost*. Если вы напишете все слова слитно и только строчными буквами, то получится абракадабра, которую будет трудно прочитать: *sobakagryzlakost*. Никогда так не делайте и другим не велите! Важно отметить, что *СБ* безразлично, строчными или ПРОПИСНЫМИ буквами записан идентификатор, поэтому переменная *sobakaGryzlaKost* это та же самая переменная, что и *sobakagryzlakost*, и даже *SoBaKaGrYzLaKoSt*. Имейте это в виду.

Как мы уже договорились, идентификаторы лучше записывать *латинскими* буквами. С другой стороны, не каждый ведь знает английский (или немецкий, к примеру) язык, поэтому вполне допустимо записывать русские слова латинскими буквами, как это принято делать в *Интернете*.

О правилах записи русских слов (*транслитерации*) можно прочитать на сайте *ru.wikipedia.org/wiki/Translit*. Вы можете использовать и свою систему перевода, главное, чтобы она была единой во всех ваших проектах.

Переходим к *основной части* нашей программы:

```
'========================================
'            ОСНОВНАЯ ПРОГРАММА
'========================================
'Расположить окно со смещением:
TextWindow.Left= 200
TextWindow.Top= 200
TextWindow.Show()
```

Если вы хотите разделить длинную программу на *отдельные смысловые составляющие*, то делайте это так, как показано выше, то есть начинайте строку со знака апострофа, как комментарий, а затем первую и третью строку заполните знаками равенства, звёздочками или тире. Во второй строке ЗАГЛАВНЫМИ буквами напишите название раздела программы.

Конечно, можно оформить надпись и по-другому, например, добавить *пустые строки*:

```
'========================================
'
'            ОСНОВНАЯ ПРОГРАММА
'
'========================================
```

Так заголовок будет выделен ещё лучше. Здесь же можно добавить пояснения о назначении и выполняемых действиях соответствующей части программы.

Обратите внимание на то, что все строки исходного текста *пронумерованы*. Это сделано только для удобства перемещения (*навигации*) по длинному тексту, в самой программе номера строк не используются. А в первых версиях бейсика номера строк были нужны не только для навигации, но и для перехода в нужное место программы с помощью операторов *goto* и *gosub*. В *СБ* для таких переходов используют *метки* (*label*).

Для ввода и вывода информации нам потребуется *текстовое окно*, которое вы можете расположить в любом месте экрана, а не только в верхнем левом углу, как это делается по умолчанию.

Для этого мы свойствам окна *Left* и *Top* присвоили значение 200.

Теперь пользователь должен ввести с клавиатуры *число*, делимость которого мы будем проверять, и нажать клавишу *ВВОД*. Значение переменной *number* станет равным этому числу:

```
start:
'считываем введенное число:
TextWindow.ForegroundColor="Yellow"
number=TextWindow.ReadNumber()
```

Здесь в верхней строке находится метка *start:*. Названия меткам выбирают точно так же, как и переменным, но после идентификатора должно стоять *двоеточие* (чтобы *СБ* не принял метку за переменную). Метка указывает на то место программы, в которое мы можем перейти с помощью любого оператора *goto*. В самом операторе *goto* нужно указать нужную метку, но уже *без* двоеточия.

```
'Если введён нуль, то работу с программой заканчиваем:
if (number = 0) then
  program.End()
EndIf
```

Закрыть программу можно, как обычно, то есть нажать кнопку с крестиком в верхнем правом углу текстового окна, но неплохо и в самой программе предусмотреть такую возможность.

Вряд ли кому-то придёт в голову проверять на делимость нуль, поэтому его вполне можно использовать как условный знак для прекращения работы с программой. Обратите внимание: чтобы проститься с программой, достаточно вызвать метод (в *СБ* методы называются *операциями*) *End* класса *Program*.

```
'Проверяем, делится ли введенное число на числа  2..31:
For i= 2 To 31
  If Math.Remainder(number, i) = 0 Then
    TextWindow.ForegroundColor="Green"
    sNumber= "Число делится на    "
  Else
    TextWindow.ForegroundColor="Red"
    sNumber= "Число не делится на "
```

```
  EndIf
  'выравниваем числа в строке:
  If i < 10 then
    sNumber= sNumber + " "
  endif
  TextWindow.WriteLine(sNumber + i)
EndFor
```

Если пользователь задал «правильное» число, то мы проверяем, делится ли оно на 2, 3, 4 и так далее, до 31. Как мы знаем из математики, если одно целое число делится на другое целое число, то остаток от деления должен быть равен *нулю*. А это легко проверить с помощью метода *Remainder* класса *Math*. При его вызове нужно в скобках через запятую записать два числа - *делимое* и *делитель*, а возвращает он *остаток* от деления. Вот и вся премудрость: если остаток нулевой, то заданное число делится на *i*, в противном случае - не делится.

Для наглядности давайте выведем информацию на экран *разными цветами*: заданное число пусть будет **жёлтым**, делители числа - **зелёные**, остальные числа - **красные**. В строковую переменную *sNumber* мы запишем результат проверки, а затем добавим один пробел для делителей меньше 10, чтобы вывод на экран был аккуратным.

Никогда не пренебрегайте *форматированием* вывода, иначе числа или слова могут быть прочитаны неверно!

Сформированную строку мы выводим на экран вместе с очередным делителем (как вы помните, он равен текущему значению переменной цикла *i*).

Ну, вот мы и управились с проверкой заданного числа (Рис. 8.1).

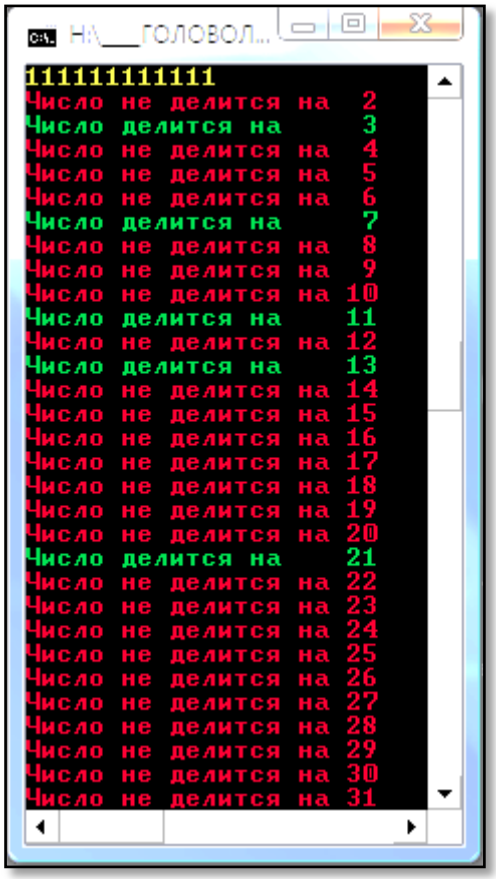


Рис. 8.1. Получилось очень красиво!

Нам осталось после всех «информативных» строк добавить *пустую строку*, чтобы отделить следующее число от уже имеющихся в окне, а затем перейти в то место программы, которое обозначено меткой *start*:

```
TextWindow.WriteLine("")
Goto start
```

Здесь пользователь может ввести новое число или, удовлетворив свое арифметическое любопытство, покончить с программой, введя нуль.

Вы можете проверять числа любой длины, но не забывайте, что *СБ* округлит его до 15-16 значащих цифр, поэтому результаты проверки вы получите неправильные!

Исходный код программы находится в папке **Delimost**.

Цикл *For*

Цикл For используется в программах тогда, когда диапазон изменения какой-либо переменной точно известен. В рассмотренном примере нам нужно было проверить, делится ли заданное число на другие последовательные числа от 2 до 31, поэтому мы поступили совершенно правильно, использовав именно этот вид циклов.

Цикл **For** записывается так:

```
For переменная_цикла=начальное_значение  To конечное _значение
                                          Заголовок цикла
  оператор1                          ← Тело цикла
  оператор2
 . . .
  операторN
EndFor                               ← Конец оператора цикла
```

А *работает* так:

1. Переменной цикла присваивается *начальное значение*.

2. Текущее значение переменной цикла сравнивается с её *конечным значением*. Если оно **меньше** конечного значения или **равно** ему, то последовательно выполняются операторы *оператор1, оператор2, … , операторN*, составляющие *тело цикла.* Когда программа дойдёт до ключевого слова *EndFor*, она снова вернётся в заголовок цикла, где переменная цикла получит следующее значение, на единицу больше текущего (то есть будет автоматически выполнен оператор присваивания *i = i + 1*). Значение переменной цикла увеличится на единицу, и цикл вернётся в начало *п.2*. Таким образом, переменная *i* последовательно принимает значения от 2 до 31, что нам и нужно. Если бы нам потребо-

валось перебрать первую *сотню* чисел, то мы бы записали заголовок цикла так: *For i = 1 To 100*.

3. Если **больше**, то выполнение цикла заканчивается и управление переходит к следующей за ключевым словом *EndFor* строке.

В качестве идентификатора *переменной цикла* часто выбирают короткие названия, часто из одной буквы – *i, j, k, l, m, n*.

С другой формой записи цикла *For* мы познакомимся на одном из следующих уроков.

Если начальное значение переменной цикла *больше* конечного, то цикл вообще не выполняется ни одного раза, а программа сразу перейдёт на строку, следующую за ключевым словом *EndFor*.

Если какой-либо оператор в теле цикла использует переменную *i*, то её значение равно текущему.

Никогда *не изменяйте* значение переменной цикла в теле цикла, иначе он может стать бесконечным или будет работать неверно!

Не забывайте ставить ключевое слово *EndFor* в конце цикла!

Программа у нас получилась дельная, но не на все случаи жизни, поэтому попробуйте усовершенствовать её!

1. Измените цикл *For* так, чтобы программа могла проверять делимость введённого числа, например, до 57.

2. Напишите программу *delimost2* (Рис. 8.2), в которой вместо переменной *number* заведите две переменные *dividend* (делимое) и *divisor* (наибольший делитель), чтобы пользователь мог самостоятельно выбирать конечное значение переменной цикла. Для ввода значений используйте операторы:

```
TextWindow.Write("Введите делимое:                ")
dividend=TextWindow.ReadNumber()
```

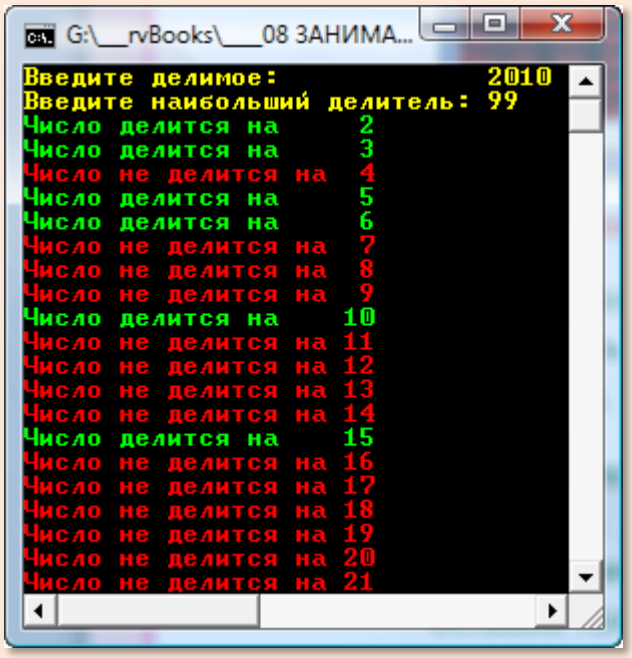


Рис. 8.2. Результат работы программы *delimost2*

В текстовом окне хранятся только 300 последних строк, поэтому есть смысл не печатать строки с *отрицательным* результатом.

Либо проверяйте, кратно ли делимое только *наибольшему* делителю. Для этого начальное значение переменной цикла задайте равным конечному или просто уберите цикл вообще, оставив только тело цикла.

Исходный код программы находится в папке **Delimost**.

3. Из *Правил делимости чисел*, которые мы рассмотрели в начале урока, легко вывести признаки делимости чисел на 15, 16, 18, 20, 22, 24, 25, 100, 1000. Попробуйте!

4. Докажите, что число, у которого число тысяч равно числу единиц, а число сотен равно числу десятков, делится на 11.

5. Докажите, что если *двузначное* число в 4 раза больше суммы цифр, то оно делится на 12.

Урок 9. Простые числа

Простыми называются натуральные числа, имеющие в точности *два разных* делителя. Из этого определения следует, что ни нуль, ни единица к простым числам не относятся. Также очень легко установить, что первое простое число - это *двойка*, потому что она делится на единицу и на саму себя (двойка – единственное *чётное* простое число). Дальше мы легко найдем тройку, пятёрку, семёрку. Нам не составит труда продолжить этот ряд: 11, 13, 17, 23, 29, 31. Чтобы отбросить многие *составные* числа, достаточно воспользоваться *признаками делимости*, которые мы рассмотрели на уроке математики. Но что вы скажете о числе 2011? Это год после нынешнего, 2010-ого, который уж точно не простой. Ни один признак делимости не действует, но это всё равно не позволяет нам однозначно сказать, простое это число или составное. Конечно, мы можем поделить 2011 на все подходящие числа, но если нам потребуется проверить число 2017? Или 514229? Или того пуще - 39916801! Вот в таких случаях правильнее один раз написать программу, чем каждый раз считать вручную. Этим мы сейчас и займёмся.

За основу нового проекта мы возьмем программу *delimost*, которую вы немедля должны загрузить в *СБ* и тут же записать в новую папку под именем **Prime**.

Переделать нам придётся совсем немного, поскольку действие обеих программ схоже. Нам потребуется только одна *переменная*, назначение которой понятно без слов:

```
'ПРОГРАММА ДЛЯ ПРОВЕРКИ ЧИСЕЛ НА ПРОСТОТУ

'variables
number=0
```

Дальше идут обычные *проверки*, на которых мы также не будем останавливаться:

```
'=======================================
'            ОСНОВНАЯ ПРОГРАММА
'=======================================
'Расположить окно со смещением:
```

```
TextWindow.Left= 200
TextWindow.Top= 200
TextWindow.Show()

start:
' считываем введённое число:
TextWindow.ForegroundColor="Yellow"
TextWindow.Write("Введите число: ")
number=TextWindow.ReadNumber()

' Если введён нуль, то работу с программой заканчиваем:
if (number = 0) then
  program.End()
EndIf

'Проверяем только числа, большие единицы:
If number < 2 Then
  TextWindow.ForegroundColor="Red"
  TextWindow.WriteLine("Число должно быть больше единицы!")
  Goto start
EndIf
```

А теперь - самое интересное: *главная часть* программы, которая все будет делать за нас (вы, наверное, помните, что Вовка в Три-девятом царстве хотел того же, но только для себя):

```
' Проверяем, делится ли введенное число на числа
'2..корень квадратный из числа
For i= 2 To Math.SquareRoot(number)
  If Math.Remainder(number, i) = 0 Then
    TextWindow.ForegroundColor="Red"
    TextWindow.WriteLine("Число составное")
    TextWindow.WriteLine("")
    Goto start
  EndIf
EndFor

TextWindow.ForegroundColor="Green"
TextWindow.WriteLine("Число простое")
TextWindow.WriteLine("")

Goto start
```

Обратите пристальное внимание на *заголовок цикла*: мы будем проверять делимость заданного числа на все числа из диапазона *2..корень квадратный* из заданного числа. Для двойки корень квадратный равен 1.414, то есть меньше начального значения переменной цикла, поэтому цикл выполняться не будет, а программа сразу перейдёт на строку, следующую за ключевым словом *EndFor*, где она сообщит нам, что двойка - простое число. С тройкой будет та же самая история. Для четвёрки начальное и конечное значения цикла совпадут, поэтому цикл выполнится 1 раз. Поскольку остаток от деления четвёрки на двойку равен нулю, то мы получим сообщение, что четвёрка - число составное.

Для следующих чисел *проверка* проходит так: заданное число последовательно делится на все числа, начиная с двойки и кончая корнем квадратным из заданного числа. Так как любое число делится на единицу и само на себя, то два разных делителя у него имеются в любом случае (кроме, единицы, разумеется). Если мы обнаружим ещё *хотя бы один* делитель, то это будет перебор: число заведомо составное - проводить испытания дальше смысла нет.

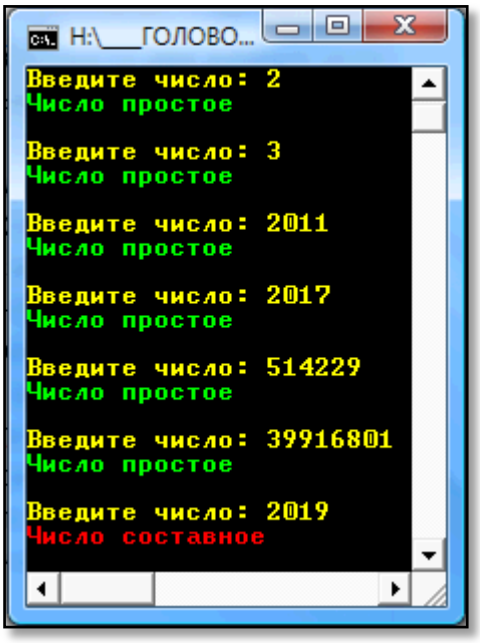


Рис. 9.1. Наша программа справляется с любыми числами!

Как видите, программа очень короткая и простая, но даже большие числа проверяет очень быстро (Рис. 9.1).

Помните, что *СБ* не умеет работать с ОЧЕНЬ БОЛЬШИМИ ЧИСЛАМИ, поэтому не мучайте программу глупыми вопросами.

Например, вы за несколько секунд узнаете, что

```
Введите число: 999999000001
Число простое
```

!

Ежели вам и этого мало, то придётся изучать «взрослые» языки программирования.

Исходный код программы находится в папке **Prime**.

Условный оператор *If*

Мы уже несколько раз встречались в своих программах с **условным оператором If**, и это не случайно: трудно себе представить программу, которая обошлась бы без него.

С условными операторами будущий программист знакомится уже в раннем детстве благодаря педагогическому усердию родителей. Мама скажет так: ***Если*** *ты будешь хорошо себя вести,* ***то*** *получишь большую сладкую конфету, а* ***иначе*** *не пойдёшь гулять во двор*. В переводе на скупой бейсиковский язык мамины посулы выглядели бы так:

If условие **Then**
  результат1
Else
  результат2
EndIf

Здесь:
условие – хорошее поведение;
результат1 – конфета;

результат2 – временное ограничение свободы.

Действует эта воспитательная конструкция – как в жизни: **если** условие соблюдено, **то** подопечный получает конфету, **не** соблюдено – принудительная домашняя отсидка.

Более лапидарный родитель, коим является отец семейства, изложил бы свои требования в более категоричной форме: ***Если*** *получишь двойку,* ***то*** *выпорю*. Тут уж никакой надежды на конфеты и другие сладостные изделия, то есть альтернатива жёсткая: принёс из школы двойку – получил ремня, не принёс – избежал ремня:

```
If условие Then
  результат
EndIf
```

В школе порка запрещена, поэтому там так мало настоящих педагогов-мужчин.

А теперь о том же, но – серьёзно!

Условный оператор If служит для того, чтобы изменять порядок выполнения операторов в программе в зависимости от некоторого логического условия. Он имеет две формы – *сокращённую* («папину»):

```
If условие  Then
  оператор1
  оператор2
 . . .
 операторN
EndIf
```

и *полную* («мамину»):

```
If условие Then
  оператор1
  оператор2
  . . .
  операторN
Else
  операторN+1
  операторN+2
  . . .
  операторN+M
EndIf
```

Условие в этих записях – обычное логическое выражение, в котором используются знаки операций сравнения =, <, > и другие. Результат логического выражения может быть либо *истинным* (условие выполняется), либо *ложным* (не выполняется).

Действует условный оператор так.

Если *условие* удовлетворено, **то** выполняются операторы после ключевого слова **Then**. Если **не** удовлетворено, то для сокращённой формы операторы пропускаются, а управление передаётся следующему за ключевым словом *EndIf* оператору. Для полной формы выполняются операторы между ключевыми словами *Else* и *EndIf*.

Например, если мы захотим найти большее (*max*) из двух чисел *n1* и *n2*, то легко сделаем это с помощью условного оператора:

```
If n1 > n2 Then
  max= n1
```

```
Else
  max= n2
EndIf
```

Довольно часто для наглядности выражение, определяющее условие, записывается в *скобках*:

```
If (n1 > n2) Then …
```

Особенно важно ставить скобки, если условие состоит из нескольких выражений, объединенных знаками *логических операций*:

```
if (a=1) and (b=1) then …
if (a=1) or (b=1) then …
```

Логические операции

AND - логическая операция **И**.
Результат операции тогда и только тогда будет *истинным*, если истинны одновременно *оба* операнда:

```
(7 > 13)  and (7 < 13)  → ложно
(7 <> 13) and (7 < 13)  → истинно
```

OR - логическая операция **ИЛИ**.
Результат операции будет *истинным*, если *хотя бы один* операнд истинен:

```
(7 > 13)  or (7 < 13)  → истинно
(7 <> 13) or (7 < 13)  → истинно
```

Редактор текста

Исходный текст программы – это обычный текст, который можно редактировать хотя бы в программе *Блокнот*, которая входит в состав операционной системы *Windows.*

Он состоит из отдельных строк, в каждой из которых записан *один* оператор или комментарий. *Пустые* строки не содержат ни одного символа и служат для отделения логических частей программы друг от друга.

Конечно, в *Блокноте* слова не будут выделяться цветом, как в *Редакторе кода* самого *СБ*, но в остальном текст в *Блокноте* ничем не отличается от исходного текста в *Редакторе кода СБ*. Это значит, что мы свободно можем копировать текст из *СБ* в любой текстовый редактор и наоборот.

Основные возможности Редактора кода СБ ничем не отличаются от таковых в других подобных программах.

Чтобы **выделить весь текст**, одновременно нажмите клавиши *CTRL + A*.

Чтобы **выделить фрагмент текста**, установите курсор в начало первой строки фрагмента, нажмите левую кнопку мышки и ведите курсор в его последнюю строку. Выделенный кусок текста окрасится в **синий** цвет:

```
while number>0
  n= n+1
  while num[n] <= number
    sNumber= sNumber + sNum[n]
    number= number - num[n]
  EndWhile
EndWhile
```

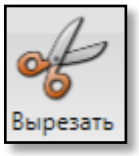
Если выделить *ключевое слово* или *идентификатор*, то автоматически он будет выделен ***жёлтым*** цветом во всей программе:

```
while number>0
  n= n+1
  while num[n] <= number
    sNumber= sNumber + sNum[n]
    number= number - num[n]
  EndWhile
EndWhile
```

Это поможет вам легко найти все строки, в которых находится нужное вам слово.

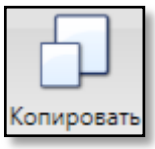
Удалить выделенный текст можно клавишами *Del* и *Backspace*.

Удалить выделенный текст **в буфер обмена** можно кнопкой *Вырезать*



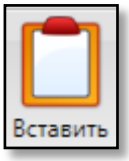
или клавишами *CTRL + X*.

Скопировать в буфер обмена можно кнопкой *Копировать*



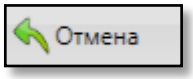
или клавишами *CTRL + C*.

Чтобы **вставить** текст из буфера обмена в позицию курсора, нажмите кнопку *Вставить*



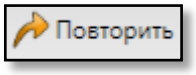
или клавиши *CTRL + V*.

Чтобы **отменить** удаление (или другую операцию редактирования), нажмите кнопку *Отмена*



или кнопки *CTRL + Z*.

Вернуть отмененную операцию можно кнопкой *Повторить*



или кнопками *CTRL + Y*.

Все кнопки редактирования находятся в группе команд *Буфер обмена*.

К сожалению, *Редактор текста СБ* не поддерживает операции по *перетаскиванию* выделенных фрагментов текста мышкой.

Если вы нажмёте *правую кнопку мыши*, то появится **всплывающее меню** (Рис. 9.2), в котором также имеются основные команды *редактирования* текста.

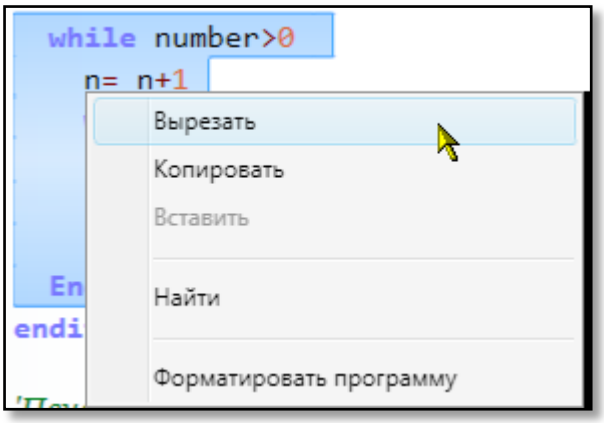


Рис. 9.2. Всплывающее меню *СБ*

Здесь же вы найдёте ещё две очень полезные команды.

Чтобы **отыскать** нужное слово в тексте, пользуйтесь командой всплывающего меню *Найти*. В появившемся диалоговом окне введите слово для поиска и нажмите кнопку *Найти* (Рис. 9.3).

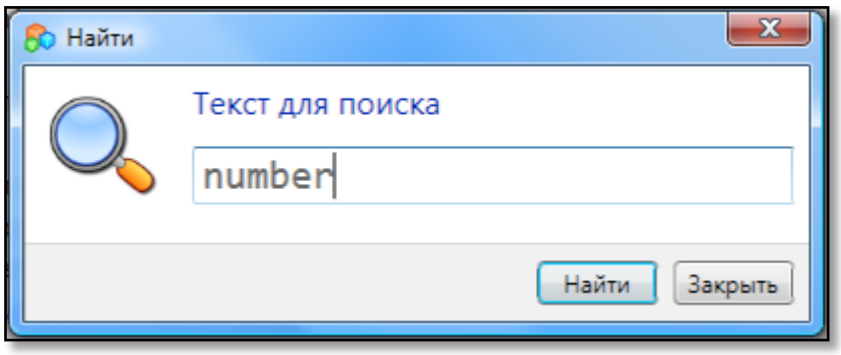


Рис. 9.3. Диалоговое окно поиска слова

Первое найденное слово от позиции курсора будет выделено ***синим*** цветом, а все остальные – ***жёлтым*** (Рис. 9.4).

Нажимая клавишу *F3*, вы последовательно будете переходить к следующему найденному слову.

```
while number>0
  n= n+1
  while num[n] <= number
    sNumber= sNumber + sNum[n]
    number= number - num[n]
  EndWhile
EndWhile
```

Рис. 9.4. Слова найдены!

Чтобы вызвать диалоговое окно *Найти*, достаточно нажать клавишу ***F3***.

И последняя по счету, но далеко не последняя по значимости команда всплывающего меню – *Форматировать программу*.

Запишем текст «всмятку»: так, чтобы все операторы были записаны от начала строки:

```
57 while number>0
58 n= n+1
59 while num[n] <= number
60 sNumber= sNumber + sNum[n]
61 number= number - num[n]
62 EndWhile
63 EndWhile
```

Теперь совсем непросто найти начало и конец вложенных циклов (если добавить еще несколько строк в тело циклов, то сделать это будет гораздо труднее, чем в этом коротком примере!). Но выполним команду *Форматировать программу* - и строчки будут напечатаны с *отступом* от края:

```
57   while number>0
58     n= n+1
59     while num[n] <= number
60       sNumber= sNumber + sNum[n]
61       number= number - num[n]
62     EndWhile
63   EndWhile
```

Именно так принято записывать исходный текст в языке паскаль, а теперь *паскалевский* способ применяется во всех языках программирования.

Вы можете и сами делать отступы, набирая сходный текст. Как обычно, для этого нужно нажать необходимое число раз клавиши *ПРОБЕЛ* или *TAB* (табуляции). Но иногда можно и запутаться, делая отступы вручную, а автоматическое форматирование работает быстро и без ошибок! Более того, оно поможет вам найти пропущенные ключевые слова. Например, удалим в строке 63 слово *EndWhile*, которое обозначает конец внешнего цикла *While*:

```
57 while number>0
58 n= n+1
59 while num[n] <= number
60 sNumber= sNumber + sNum[n]
61 number= number - num[n]
62 EndWhile
63
```

Пропущенное слово можно и не заметить, но давайте выполним команду *Форматировать программу*:

```
57   while number>0
58     n= n+1
59     while num[n] <= number
60     sNumber= sNumber + sNum[n]
61     number= number - num[n]
62     EndWhile
63
```

Теперь сразу видно, что у верхнего оператора *While* нет парного ключевого слова *EndWhile*.

Всегда записывайте программу с *отступами*! При этом ключевые слова, обозначающие начало и конец многострочных операторов, должны находиться на одинаковом расстоянии от начала строки – так вам будет гораздо проще контролировать правильность записи исходного текста. Для форматирования текста пользуйтесь командой всплывающего меню!

Некоторые возможности *Редактора кода* мы уже рассмотрели раньше. Поэтому только вкратце вспомним их.

1. Все строки исходного текста *пронумерованы* слева, что облегчает навигацию по длинному документу. Числа в левом нижнем окне *Редактора кода* показывают номер *текущей строки* и позицию текстового *курсора* в ней:

2. При наборе операторов и идентификаторов на экране появляется *подсказка*, в которой можно выбрать нужное слово, а также получить информацию о нём.

3. Ещё больше сведений предоставляет *Справочная панель* в правой части *Рабочего окна СБ*. Для того чтобы больше узнать о каком-либо элементе программы, щёлкните мышкой на нужном слове в тексте.

Способность *Редактора кода СБ* выделять цветом одинаковые слова мы можем использовать весьма необычным способом. В папке *Prime* вы найдете файл *OSH-W97.sb*. Щёлкните дважды по нему, и он загрузится в *СБ*.

Если бейсик уже запущен, загрузите файл командой *Открыть*. На самом деле это не программа, а просто текстовый файл:

```
1  АБАЖУР
2  АБАЗИНЕЦ
3  АБАЗИНКА
4  АББАТ
5  АББАТИСА
6  АББАТСТВО
7  АББРЕВИАТУРА
8  АБЕРРАЦИЯ
9  АБЗАЦ
10 АБИССИНЕЦ
```

Поскольку *СБ* позволяет загружать только файлы с расширением *.sb*, то пришлось расширение *.txt* заменить. Конечно, запускать такую «программу» не нужно, иначе вы получите несколько тысяч ошибок.

Этот файл представляет собой список существительных из словаря Ожегова и Шведовой. Допустим, нам нужно найти все слова, в которые входит слово *ТРИ*. Набираем его в первой строке – и получаем подсказку – все слова, начинающиеся с буквосочетания *ТРИ* (Рис. 9.5).

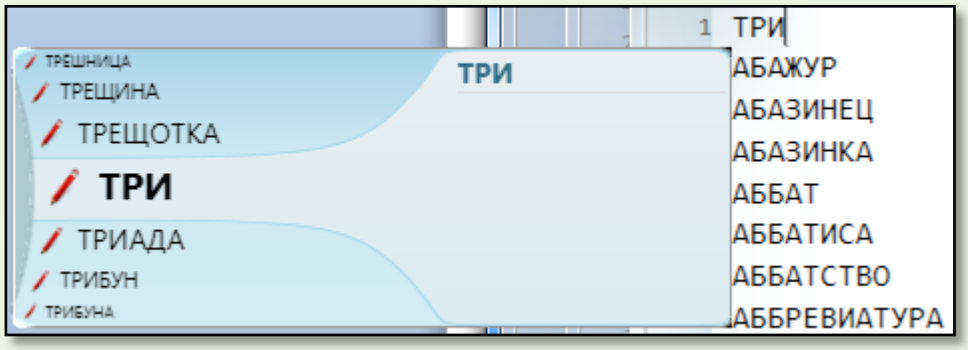


Рис. 9.5. *ТРИ*-слова в подсказке

Как обычно, вы можете прокрутить этот список в *подсказке*, но мы просто выделим слово *ТРИ* в *Редакторе кода*:

```
1  ТРИ
2  АБАЖУР
```

Теперь вы можете прокручивать окно *Редактора кода* и отыскивать слова, в которые входит *ТРИ* (Рис. 9.6).

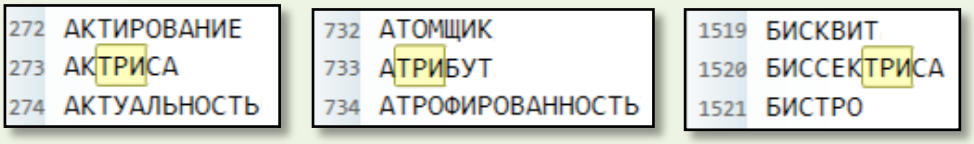


Рис. 9.6. *ТРИ*-слова найдены!

Ещё удобнее воспользоваться диалоговым окном *Найти*. Нажмите клавишу *F3* и задайте нужное слово (Рис. 9.7).

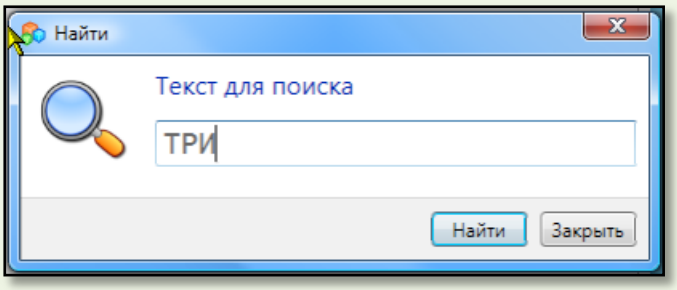


Рис. 9.7. Ищем *ТРИ*-слова

Все нужные слова будут выделены (Рис. 9.8).

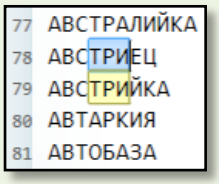


Рис. 9.8. *ТРИ*-слова выделены цветом

Нажимая клавишу *F3*, вы легко «огласите весь список».

Конечно, такой поиск можно выполнить и в любом другом редакторе текста, но этот пример показывает, что встроенный в *ИСР* редактор имеет неплохие возможности. Дальше мы рассмотрим другие способы поиска слов с помощью *СБ*, которые гораздо эффективнее этого.

А вам такое задание: хорошенько изучите все возможности *Редактора кода СБ*!

Урок 10. Файлы

Граждане, храните деньги в сберегательной кассе!

Жорж Милославский

Граждане, храните данные в файлах!

Программистская пословица

На предыдущем уроке мы научились ловко определять, простое ли заданное пользователем число или составное. Но представим себе ситуацию, что нам потребовался список всех простых чисел в каком-либо диапазоне, например от 2 до 10000. Не будем же мы тысячи раз вводить числа с клавиатуры и последовательно проверять их, а затем найденные простые числа заносить в список! Действительно, такими нудными делами должен заниматься компьютер, а наша задача – объяснить ему на простом, бейсиковском языке суть проблемы.

Так как поиск будет вестись в диапазоне чисел, то нам потребуются *две* переменные для хранения чисел, равных началу и концу диапазона.

```
'ПРОГРАММА ДЛЯ ПОИСКА ПРОСТЫХ ЧИСЕЛ
'В ЗАДАННОМ ДИАПАЗОНЕ

'variables
begin=0
end=0
```

Основную часть программы мы начинаем с ввода и проверки чисел, задающих начало и конец диапазона:

```
'=========================================
'         ОСНОВНАЯ ПРОГРАММА
'=========================================
'Расположить окно со смещением:
TextWindow.Left= 200
```

```
TextWindow.Top= 200
TextWindow.Show()

start:
'считываем дипазон чисел для проверки:
TextWindow.ForegroundColor="Yellow"
TextWindow.Write("Введите начало диапазона: ")
begin=TextWindow.ReadNumber()
'Если начало диапазона равно 0, то программу закрываем:
If begin=0 Then
  program.End()
EndIf
If begin < 2 Then
  begin = 2
EndIf

TextWindow.Write("Введите конец  диапазона: ")
end=TextWindow.ReadNumber()
If end < begin Then
  end = begin
EndIf
```

Здесь ничего нового для нас нет.

Закончив проверки, мы последовательно перебираем в цикле *For* числа из заданного диапазона, то есть переменная цикла *j* принимает значения от *begin* до *end*:

```
TextWindow.ForegroundColor="Green"
For j= begin To end
  'Проверяем, делится ли введенное число на числа
  '2..корень квадратный из числа
  For i= 2 To Math.SquareRoot(j)
    If Math.Remainder(j, i) = 0 Then
      Goto next
    EndIf
  EndFor
  TextWindow.WriteLine(j)
  File.AppendContents("primes.txt", j)
  next:
EndFor
```

```
TextWindow.WriteLine("")
Goto start
```

Во вложенном цикле *For* мы определяем, как и раньше, простоту числа *j*. Если оно составное, то мы переходим к проверке следующего числа. Если же простое, то выводим его на экран (Рис. 10.1).

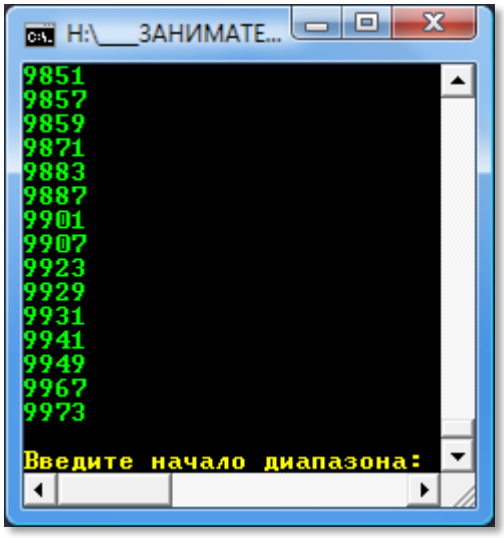


Рис. 10.1. Список простых чисел

Так в текстовое окно будут выведены все простые числа из заданного диапазона, например, 2..10000. Однако мы знаем, что в текстовом окне сохраняются только последние 300 строк, поэтому начало списка исчезнет навсегда. Конечно, можно взять маленький диапазон, чтобы все простые числа сохранялись в текстовом окне. Но и этот способ не очень хорош: всё равно придётся переписывать простые числа вручную, а это очень долго и чревато ошибками. Поэтому мы каждое новое простое число запишем в файл на диске с помощью метода класса *File*

```
AppendContents("primes.txt", j)
```

В скобках нужно указать *имя файла* вместе с расширением и *данные*, которые нужно записать в этот файл. Если файла с таким именем на диске нет, он будет создан. В противном случае вся информация в этом файле сохранится без изменений, а новая будет *добавлена* в конец файла. Это значит, что если вы будете пользоваться программой несколько раз, все данные будут запи-

саны в один и тот же файл. Если вас это не устраивает, то переименуйте файл на диске или каждый раз изменяйте в программе имя файла.

Исходный код программы находится в папке **Primes**.

Решето Эратосфена

Греческий математик Эратосфен придумал способ поиска простых чисел, который в его честь назвали *решетом Эратосфена*. Мы могли бы повторить его научный подвиг на бумаге, но лучше воспользоваться компьютером, тем более что алгоритм поиска незамысловатый, и мы без труда переведем его на язык бейсика.

В этом случае нам достаточно знать только конец диапазона чисел (переменная *end*), поскольку поиск всегда начинается с *двойки*, а сами натуральные числа мы будем записывать в массив *number*:

```
'РЕШЕТО ЭРАТОСФЕНА

'variables
number[0]=0
end=0
prime=0
```

Переменную *prime* мы отведем под текущее простое число


```
'========================================
'              ОСНОВНАЯ ПРОГРАММА
'========================================
'Расположить окно со смещением:
TextWindow.Left= 200
TextWindow.Top= 200
TextWindow.Show()

start:
TextWindow.ForegroundColor="Yellow"
TextWindow.Write("Введите конец  диапазона: ")
```

```
'Если конец диапазона равен 0, то программу закрываем:
end=TextWindow.ReadNumber()
TextWindow.WriteLine("")
If end < 2 Then
  end = 2
EndIf
If end=0 Then
  program.End()
EndIf
```

Дальше действуем по *алгоритму Эратосфена*:

1. Записываем все числа, начиная с двойки, в массив:

```
'Формируем массив натуральных чисел 2..end:
For i= 2 To end
  number[i]=  i
EndFor
```

2. Первое простое число равно двум:

```
prime=2
```

3. «Вычёркиваем» из массива число *prime*prime*, а затем все числа, начиная с этого числа, через *prime*:

4 – 6 – 8 - … для *prime= 2*,
9 – 12 – 15 - … для *prime= 3*.

И так далее:

```
nextPrime:
For i= prime*prime To end Step prime
  number[i]=0
EndFor
```

Конечно, мы не можем зачеркнуть число в массиве, поэтому присваиваем соответствующему элементу массива значение нуль, которое будет означать, что это число - *составное*.

4. Ищем первое «невычеркнутое» число – его значение в массиве должно отличаться от нулевого:

```
'ищем следующее простое число:
prime= prime+1
While (number[prime]=0)
  prime= prime+1
EndWhile
```

Теперь переменная *prime* содержит следующее простое число.

5. Если *prime* не превосходит корня квадратного из максимального числа *end*, то переходим к *п.3*.

```
If prime <= Math.SquareRoot(end) Then
  Goto nextPrime
EndIf
```

В противном случае все простые числа найдены, их значения в массиве - *ненулевые*. Перебираем весь массив и по этому признаку отыскиваем простые числа. Каждое из них печатаем в текстовом окне:

```
'Печатаем простые числа в текстовом окне:
s=""
perevodStroki=Text.GetCharacter(13) + Text.GetCharacter(10)
TextWindow.ForegroundColor="Green"
For j= 2 To end
  If number[j]<>0 Then
    TextWindow.WriteLine(number[j])
    'формируем строку для вывода в файл:
    s= Text.Append(s,j) + perevodStoki
  EndIf
EndFor
```

Как и в первом примере, мы могли бы параллельно записывать числа в файл:

```
File.AppendContents("resheto.txt", (number[j]))
```

Для этой программы так и нужно сделать, потому что с помощью решета Эратосфена не следует искать большие простые числа. Но в будущем вам вполне может пригодиться *другой* способ формирования строки для файла. Дело в том, что *СБ* крайне медленно считывает и записывает файлы стандартным способом, поэтому иногда лучше все данные объединить в *одну* строку и записать её

в файл *целиком*, а не выводить несколько сотен или тысяч *отдельных* строк.

Все простые числа мы запишем в строку *s*, которая сначала пустая. Затем мы последовательно добавляем к ней простые числа. Для этого мы используем метод *Text.Append(s,j)*, а не знак +, иначе все числа будут записаны вплотную друг за другом, без пробелов.

Чтобы отделить строки в файле друг от друга, между ними необходимо поставить символы перехода на другую строку и возврат к началу строки. Для удобства мы записали оба символа в переменную *perevodStroki*. В итоги мы получили одну-единственную длинную строку, содержащую более короткие строки с простыми числами. Нам остаётся

```
'записать простые числа в файл:
File.AppendContents("resheto.txt", s)
```

и вернуться к началу программы, чтобы пользователь мог «нарешетить» ещё простых чисел:

```
TextWindow.WriteLine("")
Goto start
```

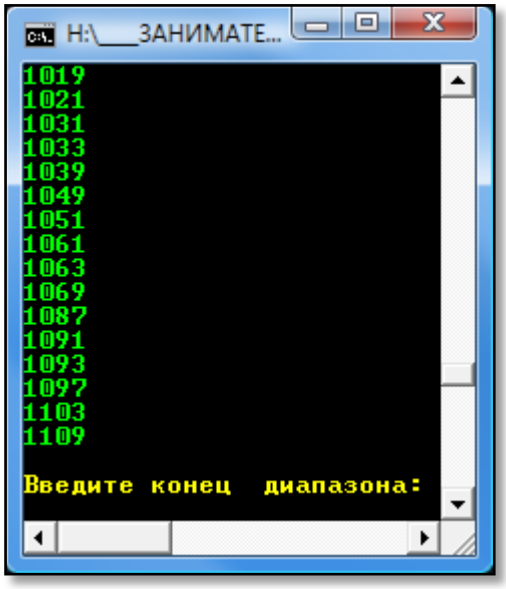


Рис. 10.2. Решето Эратосфена в действии!

Этот пример наглядно показывает нам, что для написания эффективной программы нужен хороший алгоритм. А если алго-

ритм имеется, то перевести его на любой язык программирования совсем несложно (Рис. 10.2).

Исходный код программы находится в папке **Resheto**.

Еще раз про цикл *For*

На восьмом уроке мы познакомились с первой формой записи цикла *For*. Она встречается в программах чаще всего. В этом случае переменная цикла автоматически увеличивается на *единицу* после каждого выполнения тела цикла (итерации). Но в программе *Решето Эратосфена* нам потребовалось увеличивать переменную цикла на число *prime*, которое не равно единице. Здесь нужно использовать *вторую форму записи цикла For*:

```
For переменная_цикла=начальное_значение To конечное _значение
Step шаг                    ← Заголовок цикла

  оператор1                 ← Тело цикла
  оператор2
 . . .
 операторN
EndFor                      ← Конец оператора цикла
```

Легко заметить, что она отличается от первой только тем, что к заголовку цикла добавилось ключевое слово *Step*. Число *шаг* после слова *Step* определяет *приращение* переменной цикла. Приращение может быть любым числом, в том числе и единицей - как в первой форме записи.

Если переменная цикла должна *уменьшаться*, задавайте *отрицательное* значение шага:

```
For i = 32 To 2  Step -2
```

В этом случае *i* изменяется от 32 до 2 с шагом 2: 32 – 30 - … - 4 – 2.

В цикле *While* мы самостоятельно изменяем значение переменной цикла, поэтому ключевое слово *Step* в нём не используется.

Вложенные циклы

Мы уже не раз встречались с ситуацией, когда внутри одного цикла располагался второй. Внутри второго цикла можно поместить третий – и так до бесконечности. Главное при организации таких вложенных циклов помнить *правило матрёшки*: всякий внутренний цикл должен целиком находиться внутри внешнего:

```
For …        ← Заголовок первого цикла

    For …    ← Заголовок второго цикла

        For …    ← Заголовок третьего цикла

        EndFor   ← Конец третьего цикла

    EndFor   ← Конец второго цикла

EndFor       ← Конец первого цикла
```

Как всегда, выделяйте отдельные циклы отступами!

Вложенных циклов может быть любое количество и это могут быть не только циклы *For*, но и *While* - в любых сочетаниях.

Олимпиадная задача

Давайте разберем *пример* решения задачи с помощью вложенных циклов.

В своё время на городской олимпиаде по математике мне пришлось решать такую задачу: *Подсчитайте, сколько раз пятёрка входит в представление чисел от 1 до 1000 в виде произведения простых чисел: 2, 3, 5, 7, 11,...* (например, 24 = 2 * 2 * 2 * 3; 25 = 5 * 5). Мы не на олимпиаде, поэтому пусть задачу решает компьютер, а мы составим для него простенькую программу:

```
'ОЛИМПИАДНАЯ ЗАДАЧА

'variables
n5=0 'счетчик пятерок

'=========================================
'                  ОСНОВНАЯ ПРОГРАММА
'=========================================

TextWindow.Show()

'проверяем все заданные числа:
for i= 1 to 1000
  n = i
  uslovie=1
  while uslovie=1
    'если число делится на 5,
    'увеличиваем счетчик пятерок:
    if math.Remainder(n, 5) = 0 then
      n= n / 5
      n5 = n5 + 1
    else
      uslovie=0
    EndIf
  EndWhile
EndFor

TextWindow.ForegroundColor="Green"
TextWindow.WriteLine("n5= " + n5)
TextWindow.WriteLine("")
```

```
TextWindow.ForegroundColor="Yellow"
```

Здесь мы легко найдём цикл *For*, в котором перебираются все заданные числа, и вложенный в него цикл *While*, который и подсчитывает их делители. Если число делится на 5 нацело, то мы увеличиваем счётчик на 1, иначе переходим к проверке следующего числа. Олимпиадный «подвох» этой задачи заключается в том, что полученное после деления на 5 число опять может быть кратно 5, то есть его заново нужно проверить. Если это обстоятельство не учесть, то число пятёрок можно было бы подсчитать мгновенно: 1000 : 5 = 200. Таким образом, в первой тысяче чисел ровно 200 делятся на 5. Ещё 40 делятся на 25 (5*5), 8 – на 125 (5*5*5) и одно – на 625 (5*5*5*5). Зная ответ, задачу легко решить, а я на олимпиаде намаялся (Рис. 10.3)!

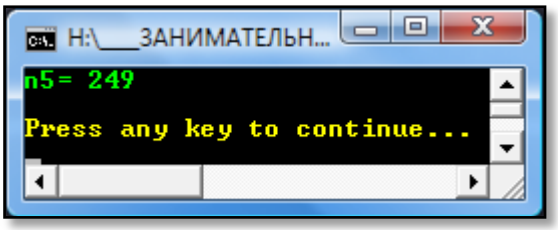


Рис. 10.3. Компьютер справился с олимпиадной задачей!

Исходный код программы находится в папке **1000**.

Добавьте в программу *Primes* переменную *fileName*, а при запуске программы пользователь пусть сам введёт имя файла для сохранения результатов работы программы. Для ввода строки используйте метод:

TextWindow.Read().

Исходный код программы находится в папке **Primes**.

Урок 11. Палиндромы

«Хоббит,
или Туда и обратно»

Толкиен

Стремление читать слова задом наперёд присуще даже самым отвратительным личностям, к коим мы, без тени сомнения, можем причислить Полиграфа Полиграфыча Шарикова из романа Михаила Булгакова *Собачье сердце*. Вспомните, как он начал своё восхождение к высотам низменных мыслей с причудливого слова *Абырвалг*, которое поначалу поставило в тупик доктора Борменталя и профессора Преображенского. Впрочем, они быстро догадались, что в оригинальном написании это была лишь вывеска магазина *Главрыба*, прочитанное Шариковым с конца.

Чему нас учит классика на данном примере? – Не всякое слово следует читать «по-арабски», дабы не смущать учёные умы!

Впрочем, истории известен и другой, весьма поучительный пример ретроградного чтения, опровергающий предыдущее утверждение. В порядком уже давние времена на Амуре село на мель судно под названием *Сунь Ят-сен* (китайский политик), его пытались стянуть за *корму*, но безуспешно. Это мероприятие продолжалось очень долго – до тех пор, пока один из матросов не повторил лингвистический подвиг Шарикова и не прочитал название судна с конца. Получилось *Не стянусь!* (только не придирайтесь к мелочам). Тогда попробовали зацепить трос за *нос* судна – и оно легко снялось с мели.

Кто мешает тебе выдумать
порох непромокаемый.

Козьма Прутков

Второй пример лучше первого потому, что в результате прочтения названия судна наоборот получилась вполне осмысленная фраза, обернувшаяся практической пользой. Конечно, это всего

лишь случайность, но кто запретит нам придумывать такие «двусмысленные» фразы?

Однако больше известны другие фразы – их можно без ущерба для смысла читать и слева направо, и справа налево. Их называют *палиндромами*. Пожалуй, самый известный палиндром *А роза упала на лапу Азора* придумал Афанасий Фет, но мы знаем этот палиндром только потому, что именно эту «волшебную» фразу диктовала Мальвина своему дубовому ученику Буратино. Легко проверить, что она читается точно так же и в обратную сторону. Потому и волшебная!

В литературе вы найдёте множество примеров фраз-палиндромов (или - более патриотично - *перевёртышей*), порой очень забавных и даже ненормативных. Но – придумывание таких афоризмов – настоящее искусство, которое совершенно не поддаётся алгоритмизации, поэтому мы поставим перед собой чисто техническую задачу – *отыскать слова, которые не изменяются при чтении наоборот*. Их тоже называют палиндромами, и вы наверняка знаете немало таких слов. Например, *РОТОР, ШАЛАШ, КАБАК*. Чтобы найти *все* такие слова, достаточно внимательно просмотреть словарь русского языка, но мы делегируем это пагубное занятие компьютеру.

Палиндромная программа

На этом уроке мы напишем программу **Palindrome** для поиска слов-палиндромов, но сначала давайте придумаем *алгоритм* программы.

Объявим переменные и присвоим им значения:

```
string1= "РОТОР"
string2= ""
```

Велико искушение присвоить переменной *string2* значение *перевёрнутой* строки *string1*. Например, так:

```
len = Text.GetLength(string1)
```

```
for i= 1 to len
  chr=Text.GetSubText(string1,len-i+1,1)
  string1= Text.Append(string2, chr)
endfor
```

Действительно, если слово не изменяется при чтении задом наперёд, то *string1 = string2*. Сравниваем строки и при их равенстве делаем вывод, что слово-палиндром найдено:

```
if string1 <> string2 then
  TextWindow.WriteLine("Не палиндром")
else
  TextWindow.WriteLine("Палиндром")
EndIf
```

Но можно поступить проще и ограничиться *одной* строковой переменной. Для этого достаточно заметить, что в слове-палиндроме одинаковые буквы расположены *симметрично* относительно середины слова, то есть нужно сравнить первую половину букв со второй (если в слове нечётное количество букв, то букву в середине слова ни с какой другой сравнивать не надо).

```
len = Text.GetLength(string1)
for i= 1 to len / 2
  if Text.GetSubText(string1,len-i+1,1) <>
Text.GetSubText(string1, i,1) then
    TextWindow.WriteLine("Не палиндром")
    Goto exitFor
  EndIf
EndFor

TextWindow.WriteLine("Палиндром"))
exitFor:
```

С одним словом всё понятно, но нам нужно просмотреть *все* слова русского языка. Обычно палиндромы ищут среди *существительных*, поэтому мы также можем ими ограничиться. Мы уже пользовались словарем Ожегова и Шведовой, в котором были бережно сохранены только существительные. Вы можете загрузить файл *OSH-W97.sb* ещё раз и убедиться, что в нём ровно 27407 слов.

Попробуем загрузить словарь из файла в массив слов. Объявляем *переменные*:

```
'ЗАГРУЖАЕМ СЛОВАРЬ ИЗ ФАЙЛА

'variables
spisok[0]=""    'массив-список слов
nWords=0        'число слов в списке
```

И считываем данные в массив:

```
'================================================

nWords=1
'загружаем словарь в массив:
fileName="OSH-W97.txt"
read()
```

Весь процесс загрузки файла вынесен в отдельную подпрограмму:

```
'================================================
Sub read
  For i= 1 To 27407
    'считываем очередное слово из файла:
    word=file.ReadLine(fileName,nWords)
    'словарь кончился:
    If word = "" Then
      Goto exitSub
    EndIf
    'заносим слово в список:
    spisok[nWords]= word
    'и показываем его в текстовом окне:
    TextWindow.WriteLine(nWords + " " + spisok[nWords])
    nWords= nWords+1
  EndFor
  exitSub:
EndSub
```

В данном случае мы точно знаем, что в словаре 27407 слов, поэтому нам достаточно цикла *For*, чтобы загрузить все слова из файла. Однако не всегда известно, сколько строк в файле, поэтому в подпрограмму введена дополнительная *проверка*: если бу-

дет считано *пустое* слово, значит, файл закончился. Дальше действие подпрограммы должно быть вам понятно и без дополнительных объяснений.

Процесс загрузки файла мы будем контролировать в текстовом окне, печатая каждое новое слово на экране. Конечно, словарь загрузится и без нашего наблюдения, но при отладке программы совсем неплохо «присмотреть» за её работой.

Чтобы ускорить вывод информации в текстовом окне, *уменьшите* его размеры!

Итак, программа написана, запускаем её! Слова сначала загружаются очень быстро, но потом скорость начинает падать и в конце на загрузку слов нельзя смотреть без слёз жалости!

Если бы нам пришлось загружать словарь один раз в день, то можно было бы и потерпеть, но ведь у нас впереди ещё много программ со словами, тут уж не натерпишься!

А всё-таки хорошо, что мы организовали просмотр загрузки словаря на экране, иначе нам трудно было бы понять причины столь медленной работы подпрограммы. А всё дело в строчке

```
word=file.ReadLine(fileName,nWords).
```

Как видите, именно здесь и считывается очередное слово из файла. В *СБ* этот процесс в принципе медленный, и вот почему. Для облегчения работы с файлами программист в *Смолл Бейсике* избавлен от многих нудных обязанностей. Например, вам не нужно открывать и закрывать файл и следить за ошибками, которые вполне могут при этом возникнуть. Вы просто указываете, из какого файла нужно считать строку номер такой-то. Но – за всё надо платить, особенно за удобство: программа на *СБ* для каждой строки открывает файл, находит нужную строку в нём, считывает её в переменную и закрывает файл.

В других языках программист сам открывает файл, сам его закрывает и сам реагирует на ошибки. Чтобы считать весь файл

или его часть, достаточно *один* раз файл открыть, а потом закрыть. Легко представить, сколько времени уходит на то, чтобы повторить эти процессы тысячи раз.

Однако время открытия и закрытия файла всегда одинаковое – и в начале считывания файла, и в его конце, почему же у нас скорость загрузки словаря уменьшается? – Потому что в файле нужно перейти к строке, номер которой мы указали. Чем она дальше от начала файла, тем дольше до неё добираться. По той же причине каждое слово дольше сохраняется в массиве *spisok*. В этом легко убедиться, если все слова загружать в первый элемент массива:

```
spisok[1]= word
```

Конечно, пользы от такой загрузки никакой, но теперь мы можем немного *ускорить* загрузку длинного файла. Для этого следует разбить длинный файл на несколько частей и загружать их последовательно. Здесь мы выиграем в скорости за счёт ускорения доступа к отдельным строкам файла. Для нас этот способ тем более хорош, что в большинстве словесных головоломок нужны слова определённой длины. Мы можем исходный словарь *OSH-W97.txt* преобразовать во «фракционный», в котором слова отсортированы не только по алфавиту, но и по длине:

```
ЯД
ЯК
ЯЛ
ЯМ
ЯР
АЗУ
АКР
АКТ
```

Мы разбиваем словарь на несколько частей по длине слов:

```
OSh_frc2-4.txt
OSh_frc5.txt
OSh_frc6.txt
OSh_frc7.txt
```

И так далее.

Теперь мы можем загружать только те файлы, которые нам нужны. Например, для поиска палиндромов достаточно загрузить только слова, в которых не более *семи* букв, поскольку достоверно известно, что в русском языке самое длинное слово-палиндром состоит именно из семи букв.

Нам нужно изменить только основную часть программы

```
nWords=1

fileName="OSh_frc2-4.txt"
read()

fileName="OSh_frc5.txt"
read()

fileName="OSh_frc6.txt"
read()

fileName="OSh_frc7.txt"
read()
```

и загружать файл *частями*. Как вы помните, товарищ Бендер не хотел брать частями, он хотел получить всё *сразу*. Но он плохо кончил. Поэтому не может служить нам примером!

Исходный код программы находится в папке **Palindrome**.

Подпрограммы

Разделяй и властвуй!

Пословица римских императоров

Никогда не пишите один
и тот же код дважды!

Программистская пословица

Обратите внимание: в программе *File2* подпрограмма *read* используется 4 раза. Если бы мы не перенесли часть кода в подпрограмму, то нам пришлось бы столько же раз повторить один и тот же текст в основной программе. Конечно, его легко скопировать в нужные места, но при этом программа станет длиннее и в ней будет сложно ориентироваться. Ещё хуже то, что в случае внесения изменений, нам придётся исправлять программу в нескольких местах, что ведёт к потере времени и увеличивает вероятность ошибки.

В принципе, можно не копировать часть кода, а повторять его в цикле. Однако, во-первых, это усложнит программу, поскольку нужно организовать этот цикл, а, во-вторых, одна и та же подпрограмма может использоваться в *разных* местах программы, и одним циклом обойтись всё равно не удастся.

Подпрограмма *записывается* так:

```
Sub имя
  оператор1        ← Тело подпрограммы
  оператор2
  . . .
  операторN
EndSub
```

Начинается она ключевым словом *Sub*, после которого нужно указать *имя* подпрограммы, а заканчивается ключевым словом *EndSub*.

В *СБ* подпрограмма называется *процедурой*, в отличие от других версий бейсика.

Между этими ключевыми словами находится *тело* процедуры, которое может состоять из любых операторов.

Внутри одной процедуры нельзя записать другую процедуру (то есть *вложенные* процедуры не допускаются), но одна процедура может вызывать другую процедуру или даже саму себя.

Чтобы исполнить код, который находится в теле процедуры, в нужном месте программы нужно записать имя процедуры со скобками: *имя()*. Почему нельзя вызвать процедуру просто по её имени? – Во-первых, такой вызов процедуры ничем не отличается от имени переменной, что приведет к ошибке при запуске программы. Во-вторых, вызов процедуры во многих языках программирования оформляется именно так, а в скобках в процедуру передают дополнительную информацию. Но в *СБ* все переменные являются глобальными, поэтому в подпрограмму ничего передавать не только не нужно, но и нельзя, то есть внутри скобок никаких символов быть не должно.

Подпрограммы используют также для того, чтобы разделить большую программу на отдельные части (*модули*), которые не зависят друг от друга, поэтому их легче отлаживать, чем программу необъятных размеров. Другое преимущество *модульного программирования* состоит в реализации важнейшего принципа: *Никогда не пишите дважды один и тот же код!* Отлаженные модули могут быть легко присоединены к любой программе и будут работать в ней без ошибок.

Примером модульного подхода может служить книга: она состоит из отдельных частей (томов), разбитых на главы. Сами главы разбиты на параграфы, абзацы, строки.

Любой механизм состоит из отдельных узлов, те, в свою очередь, - из деталей. И даже венец природы являет собой лучший образец модульного мышления Создателя нашего: человек – это совокупность отдельных органов (пищеварения, дыхания, кровообращения, слуха и других), что и позволяет лечить болезни узким специалистам (увы, не всегда успешно именно из-за своей узкой специализации).

В *СБ* модули могут быть выполнены в виде *процедур*. Основная часть программы при необходимости обращается к модулям, поэтому ядро программы можно сделать очень *небольшим*, что облегчит понимание логики работы программы, поскольку основной код получится очень коротким.

Так как программы на таких языках программирования, как бейсик или паскаль, строятся из модулей, как дома - из кирпичей или блоков, то они называются *процедурными*.

В отдельные модули обычно помещают процедуры общего назначения, которые могут потребоваться во многих программах. Например, процедура *Sound* может воспроизводить звуковые файлы, а процедура *Input* – вводить данные. У вас по мере написания программ также накопится множество таких полезных подпрограмм. Например, любая наша программа начинается с подготовки текстового окна. Давайте для примера напишем процедуру, которую затем вы сможете использовать в любых программах:

```
Sub prepareTextWindow
  'Расположить окно со смещением:
  TextWindow.Left= 200
  TextWindow.Top= 200

  TextWindow.ForegroundColor="Red"
  TextWindow.WriteLine(NAME_PROG)
  TextWindow.WriteLine("")
  TextWindow.ForegroundColor="Yellow"
  TextWindow.Show()
EndSub
```

А вот так можно использовать эту процедуру в своих программах (Рис. 11.1):

```
'const
NAME_PROG = " ИМЯ ПРОГРАММЫ"

'вызываем процедуру в начале программы:
prepareTextWindow()
```

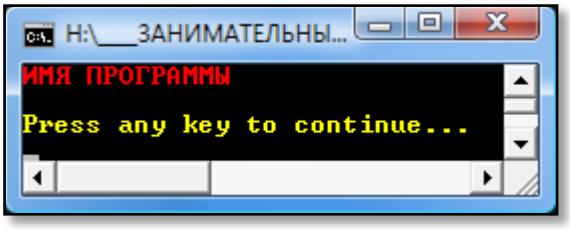


Рис. 11.1. Процедура работает!

В конкретной программе вам достаточно задать константе *NAME_PROG* название этой программы. В случае необходимости можно скорректировать цвет шрифта или позицию текстового окна на экране, но это уже совсем нетрудно.

Вы должны помнить, что в СБ все переменные являются *глобальными*, поэтому никогда не изменяйте в процедурах те переменные, которые используются в основной части программы. Лучше для каждой процедуры иметь свой собственный набор переменных.

Исходный код программы находится в папке **Sub**.

Фракционируем словарь

Как мы выяснили, довольно часто требуется список, в котором слова располагаются по длине (по числу букв): сначала двухбуквенные, затем трёхбуквенные и так далее. У нас пока имеется алфавитный список слов, который хранится в файле *OSH-W97.txt*. Мы можем непосредственно из него загрузить слова нужной нам длины, но тогда придётся просматривать весь файл, что не очень хорошо. Будет правильнее, если мы на его основе создадим такой

список, в котором слова отсортированы не только по алфавиту, но и по длине. Для временного хранения слов равной длины нам потребуется *двумерный массив*:

```
' ПРОГРАММА ДЛЯ СОЗДАНИЯ ФРАКЦИОННОГО СЛОВАРЯ

'variables
spisok[0][0]=""    'массив-список слов
nWords[0]=0        'число слов равной длины в массиве-списке
nAll=0             'общее число слов массиве-списке
```

Первый индекс отвечает за *длину* слов, второй – за *номер* слова в списке слов *равной* длины (с одним и тем же первым индексом).

Сначала слов в списках нет:

```
'================================================
nAll=0

For i= 2 to 30
  nWords[i]=0
EndFor
```

Затем мы загружаем слова из файла в двумерный массив так, чтобы они заняли своё место среди слов равной длины.

```
fileName="OSH-W97.txt"
read()
```

Затем мы записываем *фракционированный словарь* в новый файл:

```
fileNameOut="OSH-W97-frc.txt"
write()
```

Процедуру загрузки файла необходимо слегка изменить, чтобы слова записывались в словарь согласно их длине.

```
'Загрузить словарь в массив
Sub read
  For i= 1 To 100000
    word=file.ReadLine(fileName, nAll)
    len = Text.GetLength(word)
```

```
    If len = 0 Then
      Goto exitSub
    EndIf
    nWords[len]= nWords[len]+1
    nAll= nAll+1
    spisok[len][nWords[len]]= word
    TextWindow.WriteLine(n + " " + spisok[len][nWords[len]])
  EndFor
  exitSub:
EndSub
```

Всё! Осталось последовательно записать слова равной длины, начиная с двухбуквенных и кончая самым длинным словом у Ожегова – *ЧЕЛОВЕКОНЕНАВИСТНИЧЕСТВО*, в котором 24 буквы.

```
'Записать фракционный словарь в файл
Sub write
  For i= 2 To 30
    For j= 1 To  nWords[i]
      File.AppendContents(fileNameOut, spisok[i][j])
      TextWindow.WriteLine(i + " " + j+ " " + spisok[i][j])
    EndFor
  EndFor
EndSub
```

СБ понимает текстовые файлы только в формате *Юникод*, поэтому если у вас возникнут проблемы со словарём, записанным в других программах (например, *Блокнот* или *WordPad*), загрузите файл в *MS Word* и сохраните его в формате *Юникод (UTF-8)*. Тогда и *СБ*, и другие текстовые редакторы, умеющие работать с таким форматом, поймут его правильно.

Поскольку операции с файлами в *СБ* выполняются медленно, то запаситесь терпением, а также утешайтесь тем, что фракционный словарь вам придётся создавать нечасто!

Исходный код программы находится в папке **Palindrome**.

Вернёмся к нашим палиндромам!

Подготовив словари, мы можем целиком отдаться поиску палиндромов. Для этого достаточно загрузить словари в массив и проверить каждое слово в списке: если оно *симметрично*, то это палиндром, иначе – обычное слово.

Для эксперимента и развития собственной личности мы будем заполнять массив другим способом, чем мы до сих пор пользовались. Он более сложный, поэтому нам потребуется «куча» *переменных*:

```
'ПРОГРАММА ДЛЯ ПОИСКА ПАЛИНДРОМОВ

'variables
spisok[0]=""    'массив-список слов
nWords=0        'число слов в списке

txt=""          'вспомогательная переменная для считывания
файла
chr1=""         'первая буква для сравнения
chr2=""         'вторая буква для сравнения
index=0         'текущее значение индекса
beg=0           'индекс начала слова
len=0           'длина слова
```

Впрочем, сначала нас не ждут никакие сюрпризы:

```
'=========================================
'            ОСНОВНАЯ ПРОГРАММА
'=========================================
TextWindow.ForegroundColor="Red"
TextWindow.WriteLine("ИЩЕМ ПАЛИНДРОМЫ")
TextWindow.WriteLine("")
TextWindow.Show()

'считать файл 1:
fileName= "OSh_frc2-4.txt"
readFile()

'считать файл 2:
fileName= "OSh_frc5.txt"
readFile()
```

```
'считать файл 3:
fileName= "OSh_frc6.txt"
'readFile()

'считать файл 4:
fileName= "OSh_frc7.txt"
readFile()
```

Обратите внимание: файл с шестибуквенными словами мы не считываем, поскольку в словаре – проверено! – нет ни одного 6-буквенного палиндрома. Это сделано только для сокращения времени поиска, поэтому вы можете загрузить этот словарь и убедиться, что так оно и есть на самом деле.

Вся прелесть программы заключает в процедуре **readFile**:

```
'ПОДПРОГРАММА ДЛЯ СЧИТЫВАНИЯ СЛОВ ИЗ ТЕКСТОВОГО ФАЙЛА
Sub readFile
  txt= File.ReadContents(fileName)
  index=1
  beg=1
  'считываем слова, пока не закончился файл:
  While (text.GetCharacterCode(text.GetSubText(txt,index,1))>
0)

    'ищем конец текущего слова:
    While text.GetCharacterCode(text.GetSubText(txt,index,1))>
32
      index=index+1
    EndWhile

    'длина текущего слова:
    len= index-beg
    'заносим текущее слово в список:
    nWords= nWords+1
    spisok[nWords]= text.GetSubText(txt,beg,len)
    'переходим к следующему слову:
    index=index+2
    beg=index
  EndWhile
EndSub
```

Чтобы ускорить работу процедуры, мы сразу загрузим весь словарь в переменную *txt*, воспользовавшись методом *ReadContents* класса *File*. Всё бы хорошо, но в этой переменной все слова будут записаны друг за другом, а не построчно. При этом между ними находятся коды перехода на новую строку и возврата каретки. В этом легко убедиться, если посмотреть, как выглядит текст в файле. Для этого понадобится текстовый редактор, который может выводить информацию в 16-ричном виде. Не у каждого такой стоит на компьютере, поэтому начало файла с двухбуквенными словами выглядит так (Рис. 11.2).

```
00000000: C0 C4 0D 0A C0 C7 0D 0A|C0 D0 0D 0A C0 D1 0D 0A | АДАЗАРАС
00000010: C3 CE 0D 0A A8 C6 0D 0A|C8 CB 0D 0A CE CC 0D 0A | ГОЁЖИЛОМ
00000020: CE D0 0D 0A CF C0 0D 0A|D0 DD 0D 0A D1 D3 0D 0A | ОРПАРЭСУ
00000030: D3 C6 0D 0A D3 CC 0D 0A|D3 D1 0D 0A D9 C8 0D 0A | УЖУМУСЩИ
00000040: DE C3 0D 0A DE C7 0D 0A|DE D0 0D 0A DE D1 0D 0A | ЮГЮЗЮРЮС
00000050: DF C4 0D 0A DF CA 0D 0A|DF CB 0D 0A DF CC 0D 0A | ЯДЯКЯЛЯМ
00000060: DF D0 0D 0A C0 C7 D3 0D|0A C0 CA D0 0D 0A C0 CA | ЯРАЗУАКРАК
```

Рис. 11.2. Словарь в 16-ричном коде

На самом деле картина ещё страшнее, потому что *СБ* требует файлы в формате *Юникод*, а там все символы записываются двумя байтами, а не одним, как на рисунке. Можно посмотреть и в этом формате, но тогда символы справа будут совершенно непонятными!

В первых двух байтах записаны коды букв *А* и *Д*, которые образуют первое слово – *АД*. Затем идут коды *0D* и *0A*, которые разделяют в файле строки друг от друга. В десятичном виде они равны *13* и *10*, соответственно.

В переменной *txt* строки также будут отделяться этими кодами, поэтому наша задача: выделить строки из текста и заполнить ими массив *spisok*. Это легко сделать с помощью метода *GetSubText* класса *Text*:

```
spisok[nWords]= Text.GetSubText(txt,beg,len)
```

Он вырезает *len* символов текста в переменной *txt*, начиная с позиции *beg*.

Нам осталось определить *начало* и *длину* каждого слова. Мы знаем, что файл начинается с символов первого слова, поэтому начало первого слова нам известно:

```
beg=1
```

Поиск будем вести от начала файла:

```
index=1
```

В цикле *While* мы последовательно перебираем все символы в переменной *txt*. Пока их код больше 32 (это код пробела, коды всех печатных символов больше):

```
While text.GetCharacterCode(text.GetSubText(txt,index,1))> 32
      index=index+1
EndWhile
```

Цикл *While* закончится, когда встретится код 13 перехода на новую строку, тогда мы сможем определить длину слова:

```
len= index-beg
```

Записываем слово в массив и переходим к следующему слову в переменной *txt*:

```
index=index+2
beg=index
```

Когда текст в переменной *txt* закончится, а это условие мы проверяем во внешнем цикле *While*

```
While(text.GetCharacterCode(text.GetSubText(txt,index,1))> 0),
```

программа вернётся из процедуры *readFile* в строку основной части программы, которая следует за вызовом процедуры. Так последовательно загрузятся и будут обработаны все нужные нам файлы, и массив *spisok* будет содержать все слова из файлов. Способ, конечно, заковыристый, но работает довольно быстро!

Сам поиск палиндромов основан на уже рассмотренном нами алгоритме. Мы последовательно загружаем слова из массива *spisok*

в переменную *s* (для наглядности программы и некоторого ускорения работы программы; можно везде пользоваться переменной *spisok[i]*) и проверяем его на «палиндромность». Найденные слова-палиндромы выводим в текстовое окно:

```
TextWindow.WriteLine("")
TextWindow.ForegroundColor="Yellow"
'ищем палиндромы в списке слов:
for i=1 to nWords
  s= spisok[i]
  len= Text.GetLength(s)
  for j= 1 to len / 2
    chr1= Text.GetSubText(s,len-j+1,1)
    chr2=  Text.GetSubText(s, j, 1)
    if chr1 <> chr2  then
      goto break
    EndIf
  EndFor

  ''нашли палиндром:
  TextWindow.WriteLine(s)

  break:
EndFor

TextWindow.WriteLine("")
TextWindow.ForegroundColor="Red"
```

Обратите внимание на переменные *chr1* и *chr2*, в которые записываются символы очередного слова, симметричные относительно его середины. Без них также можно обойтись, но, согласитесь, так программа читается куда легче!

Вы, наверное, заметили, что палиндромы можно искать и в *обычном* словаре. Но он слишком велик для нашей программы, поскольку в русском языке самый длинный палиндром – *РОТАТОР* – состоит всего из *семи* букв. Наш фракционный словарь позволяет нам загружать только те слова, которые нам нужны для программы, так что мы не зря старались на этом уроке (Рис. 11.3)!

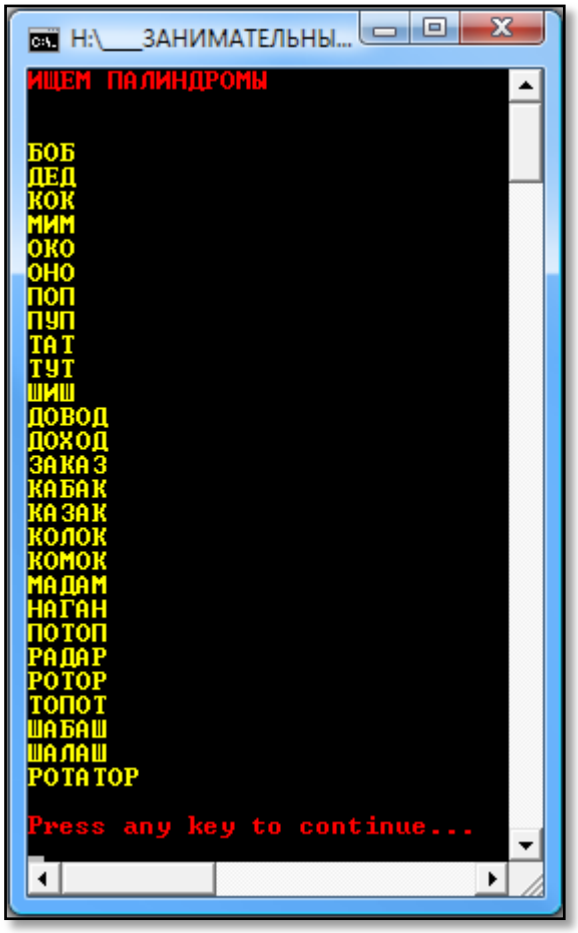


Рис. 11.3. Вот они - перевёртыши!

Исходный код программы находится в папке **Palindrome**.

Добавьте в программу *Palindrome* возможность записи в файл, чтобы найденные палиндромы сохранялись на диске.

Урок 12. Занимательная комбинаторика

Комбинаторика - это раздел математики, который изучает *множества* (совокупности, наборы) каких-либо элементов. Первая книга по комбинаторике вышла в 1666 году под названием *Рассуждения о комбинаторном искусстве*. Её написал известный немецкий математик Готфрид Вильгельм фон Лейбниц, который и придумал название *комбинаторика* этому разделу математики.

Как в жизни, так и в программировании очень часто встречаются *комбинаторные задачи*. Например, сколько различных слов можно составить из букв русского алфавита, сколько существует различных комбинаций при игре в кости двумя или тремя кубиками, сколько разных нарядов можно составить из трёх юбок и четырёх блузок и так далее.

Все комбинаторные задачи решаются с помощью комбинаторных конфигураций: *размещений, перестановок, сочетаний, композиций и разбиений*.

На этом уроке мы познакомимся с *перестановками* элементов. Возьмём множество, состоящее из трёх разных элементов, например, русских букв - *{К, О, Т}*. Всякое слово, составленное из этих букв, и называется перестановкой элементов множества.

Поскольку в множестве элементы *не упорядочены*, то мы выпишем их сначала в произвольном порядке, например так:

1. КОТ

Вот мы и получили первую перестановку, а значит, и первое слово – *КОТ*. Оно «случайно» совпало с настоящим русским словом. Чтобы найти вторую перестановку, поменяем местами вторую и третью буквы:

2. КТО

Тоже получилось неплохо, ведь *кто* - это русское местоимение.

Давайте поменяем теперь первую и вторую буквы:

3. ТКО

Такого слова нет (в старой речи была частица *–тко*, имеющая тот же смысл, что и современная -ка: *бери-тко, читай-тко*), а вот четвёртая перестановка снова удачная. Чтобы её получить, поменяем местами вторую и третью буквы:

4. ТОК

Снова меняем первую и вторую буквы и получаем пятую перестановку:

5. ОТК

Не ахти какое слово, но *Отдел технического контроля* тоже сгодится.

И последнюю перестановку мы получаем, переставив две последние буквы:

6. ОКТ

ОКТ – тоже сокращение от *Оптическая когерентная томография*, так что все наши перестановки из трёх букв *К, О, Т* оказались не совсем бессмысленными. Конечно, с другими буквами результат был бы другим.

Но почему мы можем утверждать, что нашли *все* перестановки множества из трёх элементов? – Давайте рассуждать логически. На первом месте в слове может стоять любая из трёх букв. На второе место можно поставить любую из двух оставшихся, а для последнего места останется только одна буква. Таким образом, всего можно составить 3 х 2 х 1 = 6 разных слов. Но мы ровно столько и составили, значит, других слов из этих букв составить нельзя (естественно, мы используем каждую букву только *один* раз!).

Если взять множество из четырёх разных элементов, то, рассуждая аналогично, мы придём к выводу, что из них можно составить 4 x 3 x 2 x 1 = 24 разных слова. Этот ряд легко продолжить сколь угодно далеко, а для произведения чисел от единицы до заданного (обозначим его *n*), придумано слово *факториал*. Обозначается факториал числа так:

n! (читаем: *эн-факториал*).

А чтобы его вычислить, нужно перемножить все числа от единицы до этого числа:

n! = 1 x 2 x . . . x (n-1) x n

Формула очень простая, но нетрудно догадаться, что для больших значений *n* факториал будет выражаться огромным числом.

Перемножение последовательных чисел можно доверить циклу *For*, поэтому мы с легкостью вычислим факториал любого числа, не превышающего 27 (на этом числе арифметические возможности *СБ* заканчиваются).

Факториал

Вычислять факториалы с помощью *СБ* – сплошное удовольствие!

Начинаем с *переменных*:

```
'ПРОГРАММА ДЛЯ НАХОЖДЕНИЯ
'ФАКТОРИАЛА ЗАДАННОГО ЧИСЛА

'variables
number=0
fact=  0
```

Проверяем действия пользователя:

```
start:
TextWindow.ForegroundColor="Yellow"
TextWindow.Write("Введите число 0..27 >  ")
number=TextWindow.ReadNumber()
```

```
'Проверяем заданное число

'Если задано отрицательное число,
'то работу с программой заканчиваем:
if (number < 0) then
  goto exit
EndIf

if (number > 27) then
  TextWindow.ForegroundColor="Red"
  TextWindow.WriteLine("Число должно быть от 0 до 27!" )
  goto start
EndIf
```

Если заставить программу вычислять очень большой факториал, то это вызовет ошибку, что охарактеризует нас как программистов не с лучшей стороны (Рис. 12.1). Поэтому придётся несколько ограничить свободу пользователя в выборе чисел.

```
TextWindow.WriteLine("")
TextWindow.ForegroundColor="Green"
'Вычисляем и выводим в текстовое окно
'факториал заданного числа number:

fact =  number
for i= number -1 to 2 step -1
  fact = fact * i
EndFor
If number=0 Then
  fact = 1
EndIf

'выводим факториал в текстовое окно:
TextWindow.WriteLine(number+"! = " + fact)

exit:
TextWindow.WriteLine("")
TextWindow.ForegroundColor="Yellow"
Goto start
```

Первое, на что следует обратить внимание: факториал нуля равен единице, это особый случай и его нужно учесть отдельно!

Второе: факториал вычисляется «задом наперед», то есть не как «прямое» произведение 2 * 3 * ... *number*, а как «обратное»: *number* * (*number*-1) * ... 2. Как известно, от перемены мест сомножителей произведение не изменяется, поэтому результат получится верным.

Умножать на единицу, конечно, смысла нет.

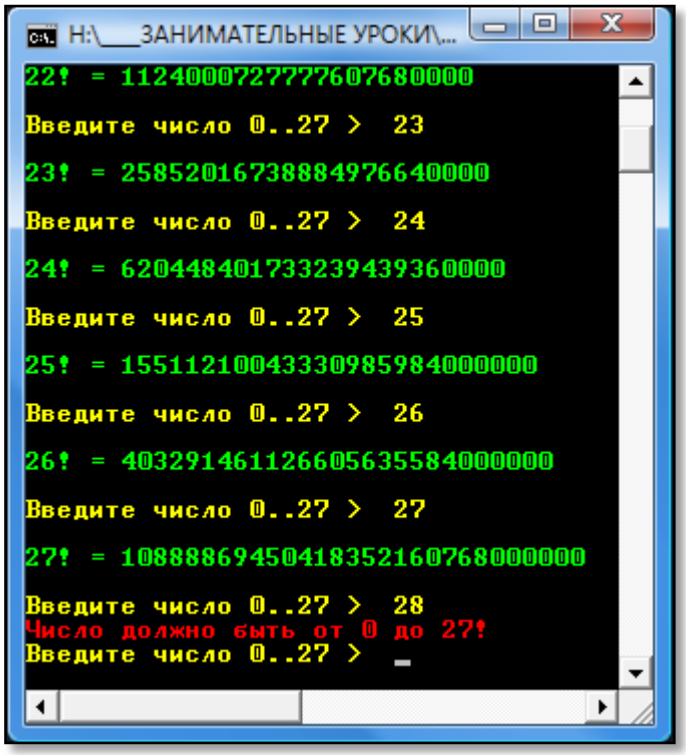


Рис. 12.1. Маловато будет!

Исходный код программы находится в папке **Factorial**.

Генерируем перестановки

Мы научились ловко узнавать общее число перестановок. Ну, пусть мы знаем, что из трёх разных букв можно составить 3! разных слов, но ведь программа *Factorial* не даёт ответа на очевид-

ный вопрос: а *как* составить эти перестановки? С множеством из трёх элементов мы легко справились, а если взять больше – четыре, пять, а то и восемь?

Оказывается, мы не первые, кто задался этим вопросом. Ещё в семнадцатом веке английские звонари научились выбивать на нескольких разных колоколах «мелодии», состоящие из всех перестановок этих колоколов. Например, для трёх колоколов нужно было сыграть такую мелодию:

Первый колокол – второй – третий.
Второй колокол – первый – третий.
Дальше вы и сами продолжите эту музыку.

Колоколов, конечно, было больше, а последовательность колокольных ударов нужно было держать в голове. Например, в *Книге рекордов Гиннесса* рассказывается о том, что в 1963 году за 17 с лишним часов удалось выбить на восьми колоколах все 8! = 40320 перестановок. В семнадцатом веке, конечно, эти музыкально-комбинаторные экзерсисы были короче, но, тем не менее, запомнить многие сотни перестановок было совсем непросто, поэтому звонари придумывали свои способы для «генерирования» всех колокольных перестановок. Один из таких способов мы и положим в основу компьютерной программы, которая быстро и правильно выпишет все перестановки элементов заданного множества.

Сначала программа выписывает первую перестановку. Например, для множества из четырёх элементов это будет

```
1 2 3 4
```

Затем она циклически меняет местами первый и второй, второй и третий, третий и четвёртый элементы. После этого снова - первый-второй, второй-третий, третий-четвёртый элементы, и так далее, пока не будут сгенерированы все перестановки (Рис. 12.2).

Начнём новый проект с загрузки исходного кода программы *Factorial*, сохраним его в папке **Permutation** и начнём добавлять новый код.

Нам вполне хватит трёх *переменных*:

```
'ПРОГРАММА ДЛЯ ГЕНЕРИРОВАНИЯ
'ВСЕХ ПЕРЕСТАНОВОК ЭЛЕМЕНТОВ МНОЖЕСТВА ЧИСЕЛ 1..nElem

'var
nElem=0   'число элементов в множестве
nPerm=0   'номер перестановки
p[0]= 0   'массив, содержащий очередную перестановку
```

Начало программы мало изменилось по сравнению с вычислением факториала. Главное – ограничить число элементов, иначе список перестановок будет очень длинным.

```
'=======================================
'             ОСНОВНАЯ ПРОГРАММА
'=======================================
start:
TextWindow.ForegroundColor="Yellow"
  TextWindow.WriteLine("")
TextWindow.Write("Введите число 1..8 >  ")
nElem=TextWindow.ReadNumber()

'Проверяем заданное число -->

'Если число меньше единицы,
'то работу с программой заканчиваем:
if (nElem <= 0) then
  Program.End()
EndIf

if (nElem > 8) then
  TextWindow.ForegroundColor="Red"
  TextWindow.WriteLine("Число должно быть от 1 до 8!")
  goto start
EndIf
```

Первая перестановка состоит из последовательного ряда чисел *1 .. nElem*:

```
For i= 1 to nElem
  p[i]= i
  c[i]=1
  pr[i]=1
endFor
nPerm=0
c[nElem]=0
' печатаем первую перестановку:
writePerm()
i= 1
```

Процедура *печати* перестановок очень простая:

```
' ПРОЦЕДУРА ПЕЧАТИ ОЧЕРЕДНОЙ ПЕРЕСТАНОВКИ
' ЭЛЕМЕНТОВ МНОЖЕСТВА
Sub writePerm
  nPerm= nPerm + 1
  TextWindow.ForegroundColor="Green"
  TextWindow.Write(nPerm + "> ")
  For wp_i= 1 to nElem
    TextWindow.Write(p[wp_i]+ " ")
  endFor
  TextWindow.WriteLine("")
endSub
```

Если вас интересуют только сами перестановки, без номера, закомментируйте строку

```
TextWindow.Write(nPerm + "> ")
```

Все остальные перестановки мы получаем из первой, меняя местами соседние элементы. Тут важно определить номера элементов для обмена. Как это работает в программе, проследите сами.

```
While (i < nElem)
  i= 1
  x= 0
  While (c[i] = nElem - i + 1)
    pr[i] = - pr[i]
    c[i] = 1
```

```
    If (pr[i]= 1) then
      x= x +1
    endIf
    i= i +1
  endWhile
  If (i < nElem) Then
    If (pr[i]= 1) then
      k= c[i] + x
    Else
      k= nElem - i + 1 - c[i] + x
    EndIf
    tmp= p[k]
    p[k]= p[k+1]
    p[k+1] = tmp
    ' печатаем очередную перестановку:
    writePerm()
    c[i]= c[i] + 1
  EndIf
endWhile
```

При желании можно сгенерировать и другие перестановки (Рис. 12.3):

```
Goto start
```

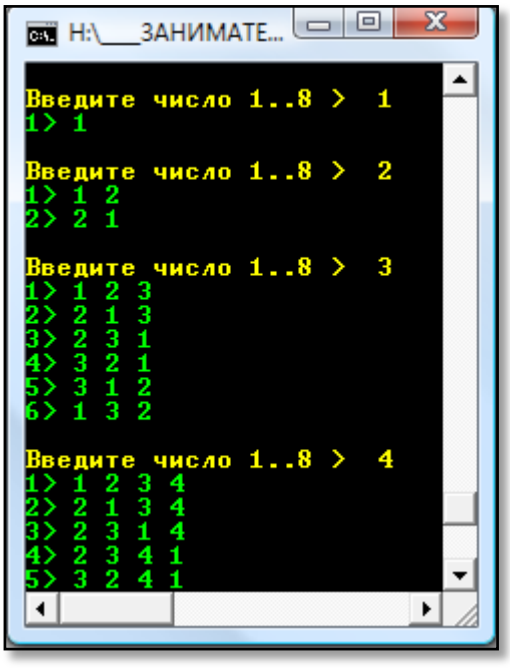


Рис. 12.2. Программа в работе!

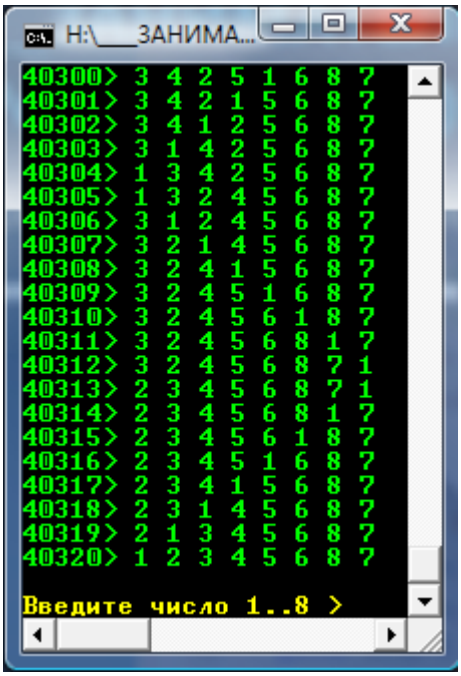


Рис. 12.3. Все перестановки множества 1..8 – звонить - не перезвонить!

Исходный код программы находится в папке **Permutation**.

Как собрать Дрим тим?

Решим ещё одну жизненно важную задачу. Представим себе, что на Чемпионат мира по футболу нужно послать сборную вашего класса, в котором 16 мальчиков. Как известно, футбольная команда состоит из 11 человек, поэтому нам нужно из 16 человек выбрать только 11.

Остальные мальчики класса, конечно, тоже поедут на чемпионат, но как запасные игроки, а девочки составят команду поддержки. Так что никто не будет обделён вниманием!

Для комбинаторики общее число мальчиков в классе - это число элементов в *множестве*, а число игроков в команде – число элементов *подмножестве*. Таким образом, нам *нужно найти все подмножества заданного множества*. Каждое подмножество какого-либо множества иначе называют набором из заданного числа элементов, или **сочетанием**. В сочетании порядок элементов не учитывается, поэтому мы отберём только 11 игроков в команду, а как между ними поделить номера на футболках (а, значит, и их амплуа на поле), это уже задача тренера.

За основу новой программы мы возьмём исходный код проекта для генерирования перестановок элементов множества и сохраним его в папке **Subset**.

Объявим новые *переменные*:

```
' ПРОГРАММА ДЛЯ ГЕНЕРИРОВАНИЯ
' ВСЕХ ПОДМНОЖЕСТВ k МНОЖЕСТВА ЧИСЕЛ n=1..nElem

'var
nElem=0     ' число элементов в множестве
k= 0        ' число элементов в подмножестве
a[0]= 0     ' массив, содержащий очередное подмножество
nSubset=0   ' номер подмножества
```

В этой программе пользователь должен ввести два числа, которые мы должны *проверить*, иначе расчеты окажутся неверными.

Важно учесть, что в подмножестве не может быть элементов больше, чем в множестве.

```
'========================================
'            ОСНОВНАЯ ПРОГРАММА
'========================================
start:
TextWindow.ForegroundColor="Yellow"
TextWindow.WriteLine("")
TextWindow.Write("Введите число элементов в множестве 1..30
>  ")
nElem=TextWindow.ReadNumber()
'Проверяем заданное число -->

'Если число меньше единицы,
'то работу с программой заканчиваем:
if (nElem <= 0) then
  Program.End()
EndIf

if (nElem > 30) then
  TextWindow.ForegroundColor="Red"
  TextWindow.WriteLine("Число должно быть от 1 до 30!")
  goto start
EndIf

TextWindow.Write("Введите число элементов в подмножестве 1.."
+ nElem + " >  ")
k= TextWindow.ReadNumber()
if (k < 1) Or (k > nElem) then
  TextWindow.ForegroundColor="Red"
  TextWindow.WriteLine("Число должно быть от 1 до "  + nElem +
"!")
  goto start
EndIf
```

Если данные пользователя успешно прошли проверку, то *генерируем* все подмножества:

```
For i= 1 to k
  a[i]= i
endFor
```

```
nSubset=0
p= k

While (p >= 1)
  'печатаем очередное подмножество:
  writeSubset()
  if (k = nElem) then
    Goto start
  endIf
  If (a[k]= nElem) then
    p= p - 1
  else
    p= k
  endIf

  If (p >= 1) Then
    For i= k to p step-1
      a[i]= a[p] + i - p + 1
    endFor
  EndIf
endWhile
```

Затем пользователь может ввести другие данные и продолжить работу с программой:

```
Goto start
```

Процедура *печати* очередного подмножества почти не отличается от процедуры печати перестановок:

```
'ПРОЦЕДУРА ПЕЧАТИ ОЧЕРЕДНОГО ПОДМНОЖЕСТВА
'ЭЛЕМЕНТОВ МНОЖЕСТВА
Sub writeSubset
  nSubset = nSubset + 1
  TextWindow.ForegroundColor="Green"
  TextWindow.Write(nSubset + "> ")
  For wp_i= 1 to k
    TextWindow.Write(a[wp_i]+ " ")
  endFor
  TextWindow.WriteLine("")
endSub
```

Запустив программу и введя наши данные: 16 элементов в множестве и 11 – в подмножестве, мы получим огромный список из 4368 разных сборных класса (Рис. 12.4)! Комбинаторную задачу мы решили, а вот какая из этих сборных действительно станет командой мечты, тут комбинаторика бессильна!

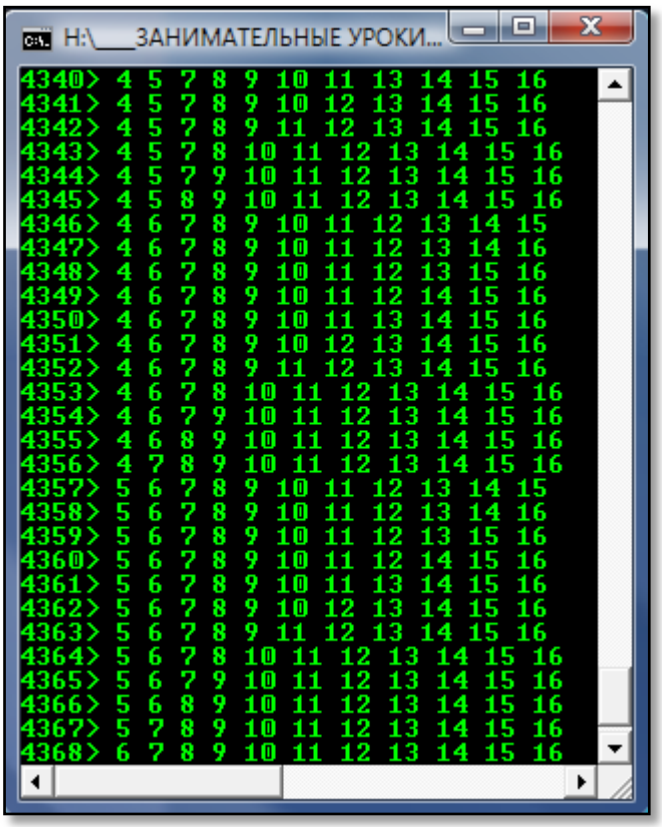


Рис. 12.4. Вот сколько сборных можно послать на чемпионат!

Иногда полезно заранее узнать, сколько же получится сочетаний при тех или иных исходных данных - не печатать же весь список подмножеств, если нам интересно узнать только их число! А на этот случай в комбинаторике припасена несложная формула:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

В нашей программе *n = nElem*.

Как видите, в комбинаторике без факториала не обойтись!

Исходный код программы находится в папке **Subset**.

1. Все перестановки большого числа элементов невозможно увидеть в текстовом окне, поэтому организуйте запись результатов работы программы в файл.

2. Подумайте, как вместо чисел использовать другие элементы, например, буквы.

3. Напишите программу, которая вычисляла бы и печатала в текстовом окне число сочетаний.

4. Переделайте программу *Factorial* так, чтобы она вычисляла факториал перемножением чисел от 2 до заданного.

Урок 13. Занимательная математика

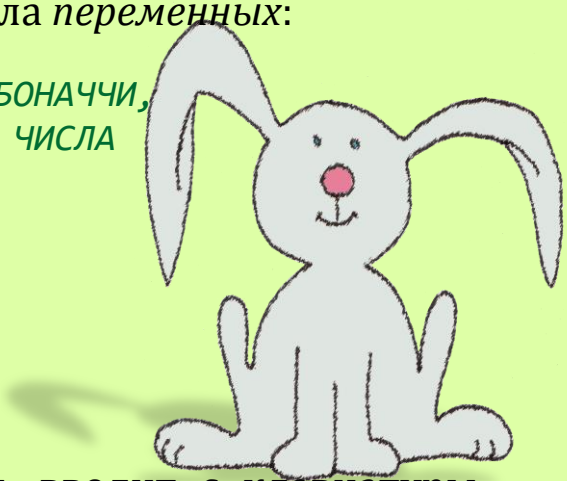
Предмет математики настолько серьёзен,
что полезно не упускать случаев
делать его немного занимательным.
Блез Паскаль

Давайте уделим немного нашего времени и внимания *числам Фибоначчи*, которые, как нетрудно догадаться, открыл Фибоначчи (он же Леонардо Пизанский), средневековый математик, автор *Книги абака*, которую он написал в 1202 году.

Впрочем, дотошные историки утверждают, что этот ряд чисел был известен в Индии, где он использовался при стихосложении, задолго до Фибоначчи.

В этой книге, среди прочих, есть и задача о размножении кроликов. Подробно мы её рассмотрим на уроке *Занимательное моделирование*, а на этом мы ограничимся только тем, что *найдём все числа Фибоначчи, которые не больше некоторого заданного числа n.*

Как обычно, начинаем программу с раздела *переменных*:



```
'ПРОГРАММА ДЛЯ НАХОЖДЕНИЯ ВСЕХ ЧИСЕЛ ФИБОНАЧЧИ,
'КОТОРЫЕ НЕ БОЛЬШЕ НЕКОТОРОГО ЗАДАННОГО ЧИСЛА

'variables
number=0
f=  0
f1= 0
f2= 0
```

После запуска программы пользователь вводит с клавиатуры число, ограничивающее поиски чисел Фибоначчи:

```
start:
  TextWindow.ForegroundColor="Yellow"
  number=TextWindow.ReadNumber()
```

Мы, естественно, должны *проверить* правильность ввода:

```
'Если задано отрицательное число,
'то работу с программой заканчиваем:
if (number < 0) then
  goto exit
EndIf

if (number < 1) or (number> 40000000) then
  TextWindow.ForegroundColor="Red"
  TextWindow.WriteLine("Число должно быть от 1 до 40000000!")
  goto start
EndIf
```

Если всё нормально, удовлетворяем запросы пользователя:

```
f=  0
f1= 1
f2= 1

TextWindow.WriteLine("")
TextWindow.ForegroundColor="Green"
'Вычисляем и выводим в текстовое окно
'числа Фибоначчи, не превышающие number:
TextWindow.WriteLine(0)
While f2 <= number
  TextWindow.WriteLine(f2)
  f2 = f1+f
  f  = f1
  f1 = f2
EndWhile

exit:
TextWindow.WriteLine("")
TextWindow.ForegroundColor="Yellow"
Goto start
```

Первые три числа задаём сразу:

```
f=  0
f1= 1
f2= 1
```

Первое число равно нулю, поэтому его нужно вывести сразу, потому что оно в дальнейших расчётах не участвует:

```
TextWindow.WriteLine(0)
```

Второе число равно 1, а все последующие равны сумме двух предыдущих, то есть третье число - 0 + 1 = 1, четвёртое - 1 + 1= 2, пятое - 1 + 2 = 3 и так далее, до бесконечности:

```
  f2 = f1+f
  f  = f1
  f1 = f2
```

Первое число обозначено одной буквой *f*, а второе буквой *f* и единицей. Логичнее было бы обозначить их как *f1* и *f2*, но нередко последовательность чисел Фибоначчи начинается с *единицы*, а не с нуля. Вот поэтому и возникла у нас такая несуразица!

В цикле *While* мы проверяем, не достигло ли очередное число Фибоначчи заданного нами предела. Как только программа закончит поиск (Рис. 13.1), она вернётся к началу, и пользователь сможет ввести новое число.

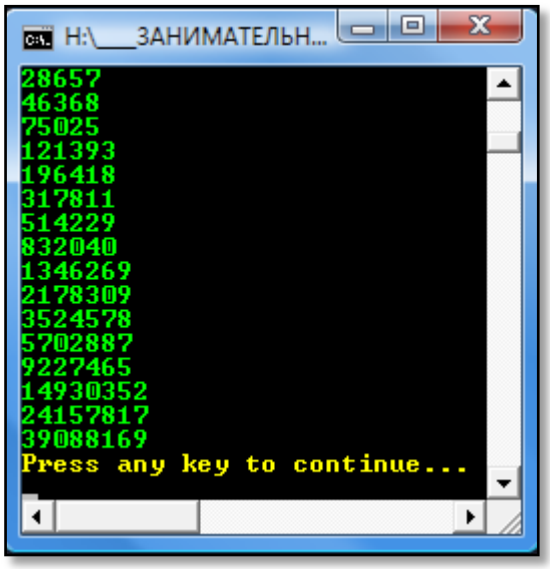


Рис. 13.1. Числа Фибоначчи

Исходный код программы находится в папке **Fibonacci**.

А теперь мы напишем две родственные программы из школьной жизни – для вычисления *НОД* (*наибольшего общего делителя* двух чисел) и *НОК* (*наименьшего общего кратного*). Они вам должны быть хорошо известны (если это не так, то сходите, наконец, в школу!).

Наибольший общий делитель (*НОД*)

Для вычисления *НОД* мы воспользуемся ускоренным алгоритмом Евклида для двух чисел - *number1* и *number2:*

```
'ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ НАИБОЛЬШЕГО
'ОБЩЕГО ДЕЛИТЕЛЯ ДВУХ ЧИСЕЛ (НОД)

'variables
number1=0
number2=0
NOD=0

'=========================================
'              ОСНОВНАЯ ПРОГРАММА
'=========================================
start:
TextWindow.ForegroundColor="Yellow"
TextWindow.Write("Введите первое число >  ")
number1 = TextWindow.ReadNumber()
'если задано отрицательное число,
'то работу с программой заканчиваем:
if (number1 < 0) then
  Program.End()
EndIf

TextWindow.Write("Введите второе число >  ")
number2 = TextWindow.ReadNumber()
```

Чтобы не обременять пользователя лишними заботами, мы позволим ему вводить числа произвольно, хотя для алгоритма важно, чтобы первое число было *больше* второго, поэтому выправляем ситуацию, если это необходимо:

```
'если первое число меньше второго,
```

```
'то меняем их значения:
if number1 < number2 then
n= number1
number1=number2
number2=n
EndIf
```

При равенстве чисел алгоритм сработает правильно.

Если меньшее из чисел (или оба) равны нулю, то за *НОД*, понятное дело, следует принять первое число:

```
if number2=0 then
  NOD = number1
else
  speed_euklid()
EndIf
```

Если же оба числа положительные, то мы начинаем действовать по ускоренному алгоритму Евклида:

```
Sub speed_euklid
  while number2 > 0
    n = Math.Remainder(number1, number2)
    number1 = number2
    number2 = n
  EndWhile
  NOD = number1
EndSub
```

В цикле *While* мы на каждой итерации находим остаток от деления первого числа на второе, значение второго числа присваиваем первому числу, а значение остатка - второму. Так мы продолжаем до тех пор, пока остаток от деления не станет равен *нулю*. Поскольку с каждым разом одно из чисел уменьшается, то рано или поздно это условие будет выполнено.

Вычисленное значение *НОД* мы печатаем в текстовом окне (Рис. 13.2):

```
TextWindow.WriteLine("")
TextWindow.ForegroundColor="Green"
'выводим НОД в текстовое окно:
TextWindow.WriteLine("НОД = " + NOD)

exit:
TextWindow.WriteLine("")
TextWindow.ForegroundColor="Yellow"
Goto start
```

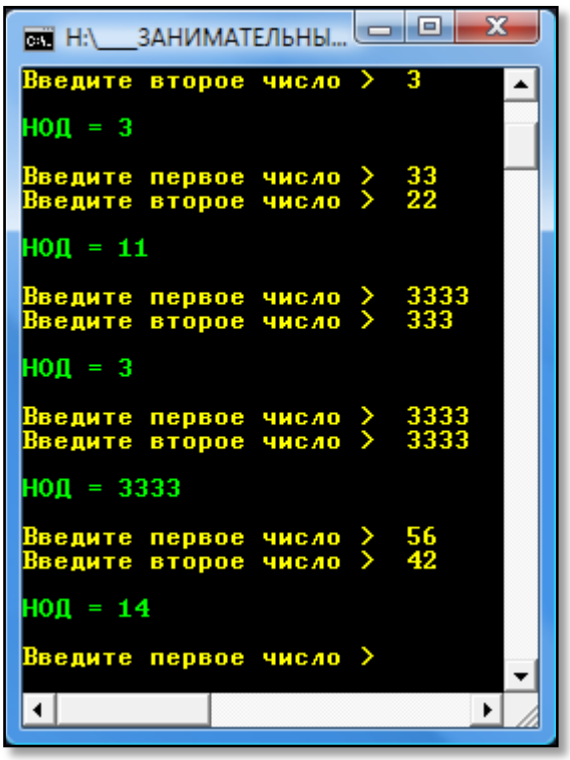


Рис. 13.2. Вычисляем *НОД*

Для большего понимания давайте найдём *НОД* для чисел

number1 = 42

number2 = 14

Так как второе число больше нуля, то находим остаток от их деления:

n = Math.Remainder(42, 14) = 0

И присваиваем новые значения переменным:

number1 = 14

number2 = 0

Второе число теперь равно нулю, значит, *НОД* найден – он равен второму числу, то есть 14.

Рассмотрим другой пример:

number1 = 56

number2 = 42

n = Math.Remainder(56, 42) = 14

number1 = 42

number2 = 14

После первого же цикла мы пришли к первому примеру.

Вы можете взять любые числа и убедиться, что алгоритм действительно работает очень быстро (Рис. 13.3)!

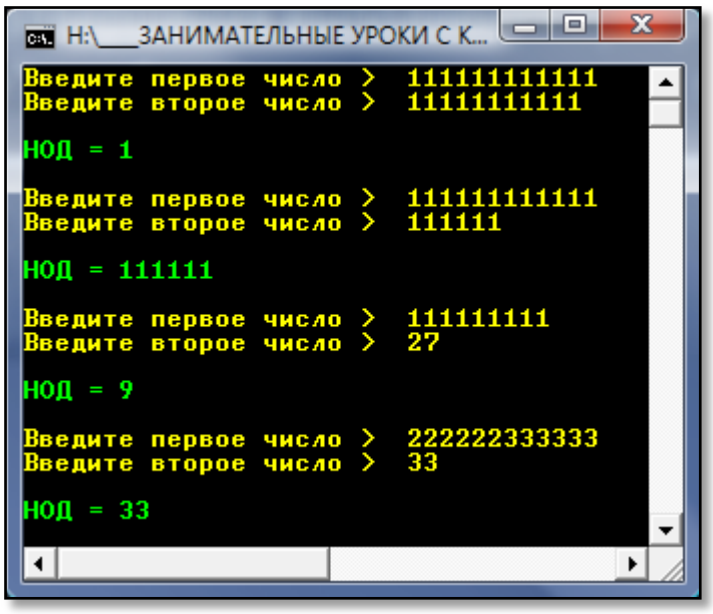


Рис. 13.3. Наша программа легко справляется и с БОЛЬШИМИ числами!

Исходный код программы находится в папке **NOD**.

Публикуем программу

Если вы хотите поделиться своей программой с друзьями или с другими любителями программирования, то в *СБ* вы можете сделать это одним щелчком по кнопке *Опубликовать* (Рис. 13.4).

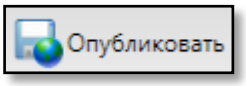


Рис. 13.4. Кнопка для публикации программы

Компьютер должен быть подключён к Интернету!

Пройдет совсем немного времени, и вы получите сообщение (Рис. 13.5).

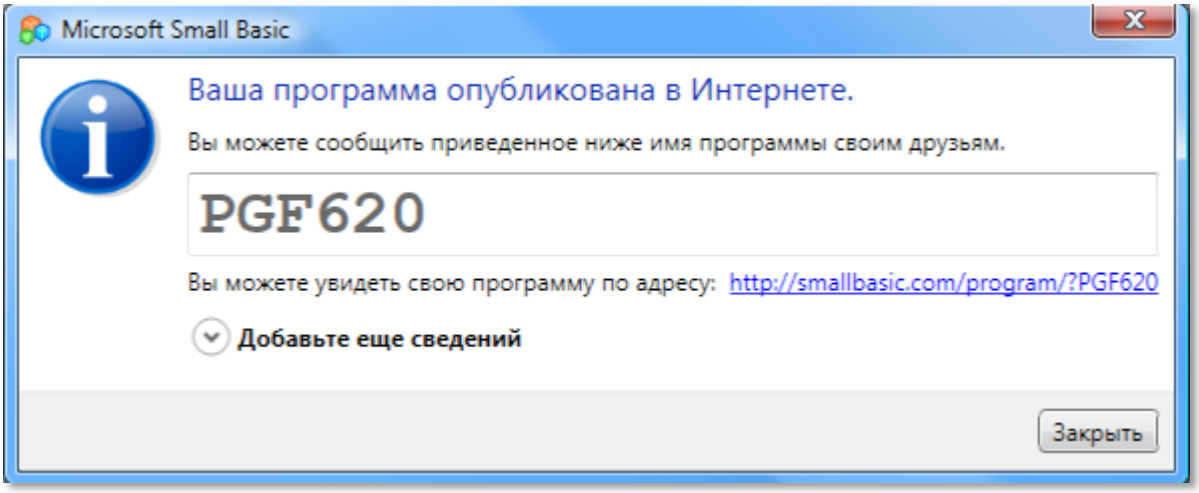


Рис. 13.5. Код вашей программы на сайте *smallbasic.com*

Обязательно запомните имя (код) программы, состоящий из трёх букв и трёх цифр! Зная его, каждый сможет легко загрузить исходный код вашей программы в *СБ* и запустить её на своём компьютере. Как это сделать, мы подробно разобрали на втором уроке.

А вот если у вашего приятеля *СБ* не установлен, то он может запустить вашу программу непосредственно в окне браузера, например, вполне сгодится и стандартный браузер *Internet Explorer* (Рис. 13.6).

Для этого нужно ввести адрес вашей программы на сайте:

smallbasic.com/program/?pgf620

Как видите, после вопросительного знака нужно указать имя программы.

Если ваша программа не появится в окне браузера, вам будет предложено установить плагин *Silverlight*. Обязательно сделайте это. Тогда вы сможете прямо в Интернете запускать и свои собственные, и любые другие программы, которые имеются на сайте *smallbasic.com*.

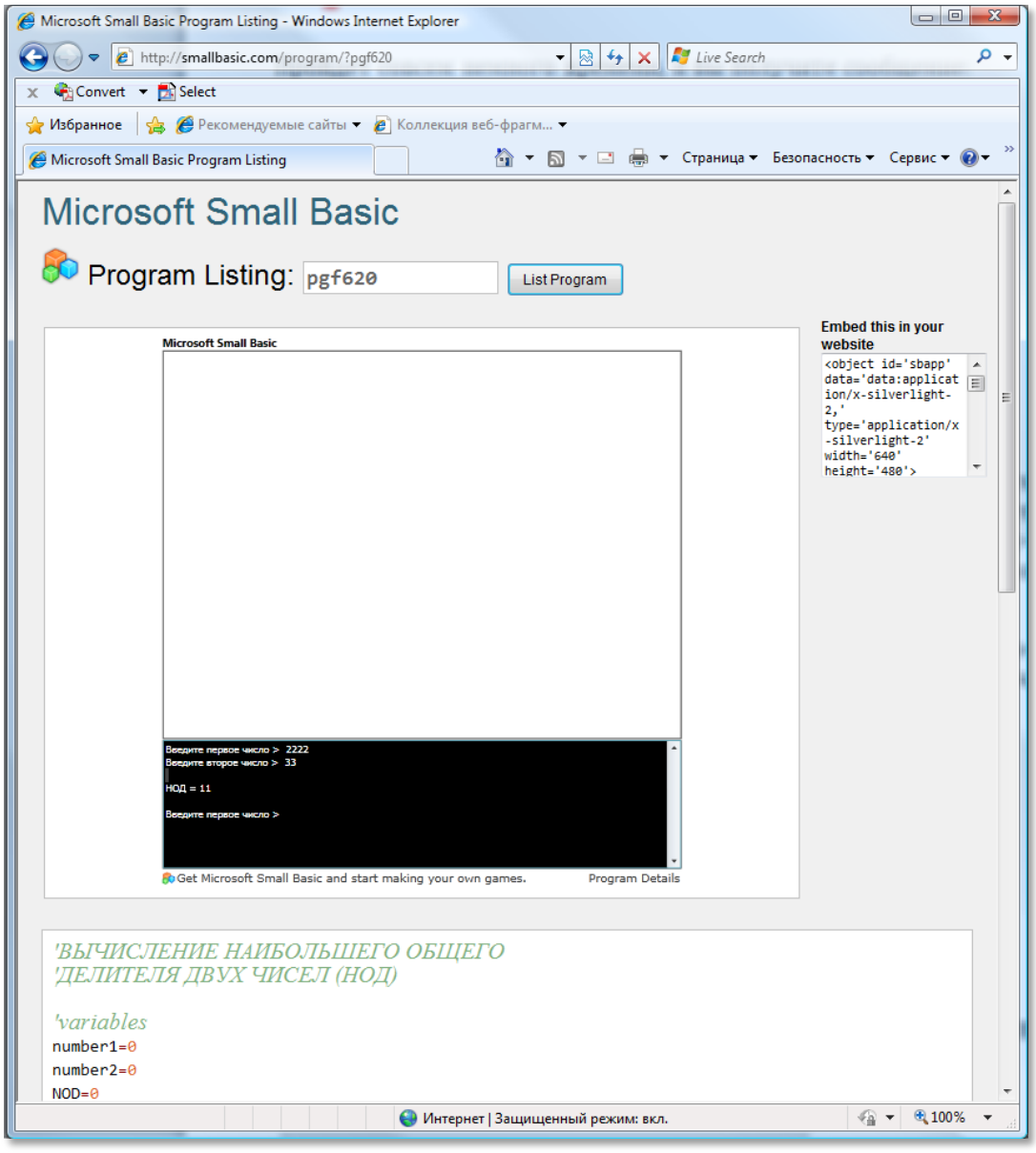


Рис. 13.6. Программа запущена в *Интернет Эксплорере*

В окне браузера вы не только сможете запустить программу, но также и посмотреть её исходный код. Это значит, что вы можете *хранить* отлаженные программы не только на своём компьютере, но и на сайте *smallbasic.com*.

Справа, под надписью *Embed this in your website* вы увидите список с кодом, который вы можете разместить на своей веб-странице (если он у вас есть, разумеется), чтобы вашу программу можно было бы запускать и с неё.

Если ваша программа загружает какие-либо файлы с диска (например, словари, картинки, звуки, музыку), то их придётся выложить на сайт отдельно, иначе программа будет работать неверно!

Наименьшее общее кратное (*НОК*)

Зная наибольший общий делитель двух чисел, мы оч ень просто вычислим наименьшее общее кратное по такой формуле:

НОК = число1 * число2 / НОД (число1, число2)

Возьмём за основу предыдущую программу и все вычисления *НОД* перенесем в подпрограмму *calcNOD*:

```
'Поцедура для вычисления НОД двух чисел -
'number1 и number2
'В переменной NOD возвращается результат
Sub calcNOD
  'если первое число меньше второго,
  'то меняем их значения:
  if n1 < n2 then
    n= n1
    n1=n2
    n2=n
  EndIf

  if n2=0 then
    NOD = n1
```

```
  else
    while n2 > 0
      n = Math.Remainder(n1, n2)
      n1 = n2
      n2 = n
    EndWhile
    NOD = n1
  EndIf
EndSub
```

Поскольку *СБ* не позволяет передавать в процедуру параметры, то мы должны помнить, что переменные *number1* и *number2* это те же самые переменные, что и в основной части программы. То же самое касается и вычисленного значения *NOD*! Поэтому в начале программы объявляем все *переменные* программы:

```
'ВЫЧИСЛЕНИЕ НАИМЕНЬШЕГО ОБЩЕГО
'КРАТНОГО ДВУХ ЧИСЕЛ (НОК)

'variables
number1=0
number2=0
NOD=0
NOK=0
```

А дальше мы просто вычисляем *НОК* по указанной выше формуле (Рис. 13.7):

```
'========================================
'               ОСНОВНАЯ ПРОГРАММА
'========================================
start:
TextWindow.ForegroundColor="Yellow"
TextWindow.Write("Введите первое число >  ")
number1 = TextWindow.ReadNumber()
'если задано отрицательное число,
'то работу с программой заканчиваем:
if (number1 < 0) then
  Program.End()
EndIf

TextWindow.Write("Введите второе число >  ")
number2 = TextWindow.ReadNumber()
```

```
if number1 * number2=0 then
  NOK = 0
Else
  'сохраняем введенные числа
  'от изменений в процедуре calcNOD:
  n1= number1
  n2= number2
  'вычисляем NOD:
  calcNOD()
  NOK= number1 * number2 / NOD
EndIf

TextWindow.WriteLine("")
TextWindow.ForegroundColor="Green"
'выводим НОК в текстовое окно:
TextWindow.WriteLine("НОК  = " + NOD)

exit:
TextWindow.WriteLine("")
TextWindow.ForegroundColor="Yellow"
Goto start
```

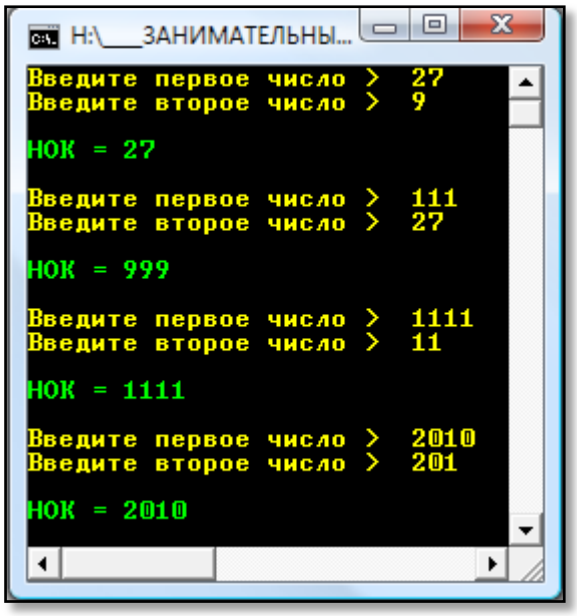


Рис. 13.7. Вычисляем *НОК*

Исходный код программы находится в папке **NOK**.

Урок 14. Графические приложения

С появлением операционной системы *Windows* основным интерфейсом пользователя стал *графический*, а текстовый вывод на консоль применяется только для быстрого получения результатов вычислений.

Вся информация – текстовая и графическая - при этой разновидности интерфейсов выводится в *графическое окно*, которое в *СБ* представлено классом **GraphicsWindow**.

Класс *GraphicsWindow* (*Графическое окно*)

Всякая программа с графическим интерфейсом начинается с создания и настройки параметров окна приложения (Рис. 14.1). В *СБ* графическое окно создается *автоматически* - достаточно хотя бы раз упомянуть его в программе. Например, так:

```
GraphicsWindow.Show()
Program.Delay(10000)
GraphicsWindow.Hide()
```

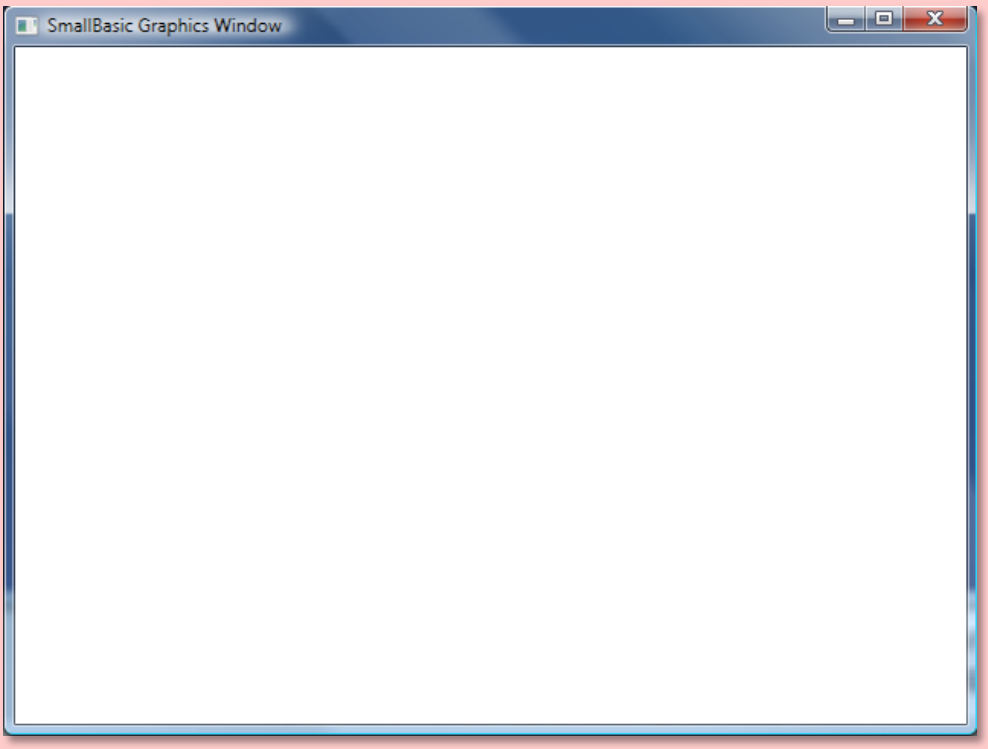


Рис. 14.1. *Графическое окно* создано!

Метод **Show** *показывает* графическое окно при запуске программы, а метод **Hide** делает его *невидимым* (при этом приложение продолжает работать). Немного изменим программу и посмотрим, как пульсирует окно:

```
For i= 1 To 10
  GraphicsWindow.Show()
  Program.Delay(1000)
  GraphicsWindow.Hide()
  Program.Delay(1000)
EndFor
```

Как видите, графическое окно *СБ* – это обычное окно *Windows*, оно имеет 4 стандартные кнопки (Рис. 14.2).

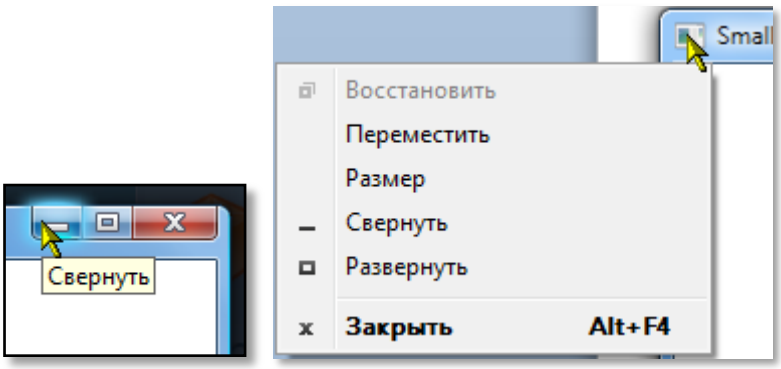


Рис. 14.2. Кнопки стандартного окна

С их помощью можно проделывать обычные «фортели»: сворачивать окно, разворачивать его, перемещать и, наконец, закрыть.

Окно имеет *заголовок* «SmallBasic Graphics Window», который легко заменить названием программы, задав свойству окна **Title** нужное значение (Рис. 14.3):

```
GraphicsWindow.Title="Моя программа"
```

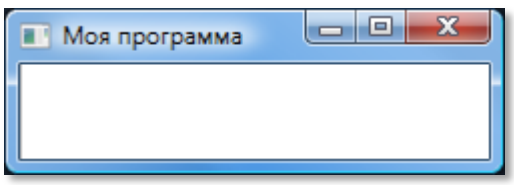


Рис. 14.3. Вот так-то лучше!

Если вы забудете, что написано, в заголовке окна, то легко освежите память, считав значение этого свойства в переменную:

```
s = GraphicsWindow.Title
```

Заголовок окна можно использовать и нестандартно, например, выводить в него число набранных в игре очков, время и другую полезную информацию:

```
time=0
Timer.Interval = 1000
Timer.Tick = OnTimer

Sub OnTimer
  time= time+1
  GraphicsWindow.Title= time
EndSub
```

Обычно пользователь может не только перетаскивать окно по экрану, но и изменять его размеры, потянув за края или углы. Иногда это приводит к тому, что графические элементы программы перемещаются, и вид программы становится неряшливым. Поэтому вы можете пресечь подобное «вольнодумство» пользователя и запретить ему изменять размеры окна, присвоив свойству **CanResize** нулевое значение:

```
GraphicsWindow.CanResize=0
```

Или значение "False":

```
GraphicsWindow.CanResize= "False"
```

При этом у окна останется только одна кнопка справа - *Закрыть*, а в системном меню исчезнут некоторые команды (Рис. 14.4).

Впрочем, вы можете сменить гнев на милость и снова вернуть пользователю свободу действий:

```
GraphicsWindow.CanResize= "True"
```

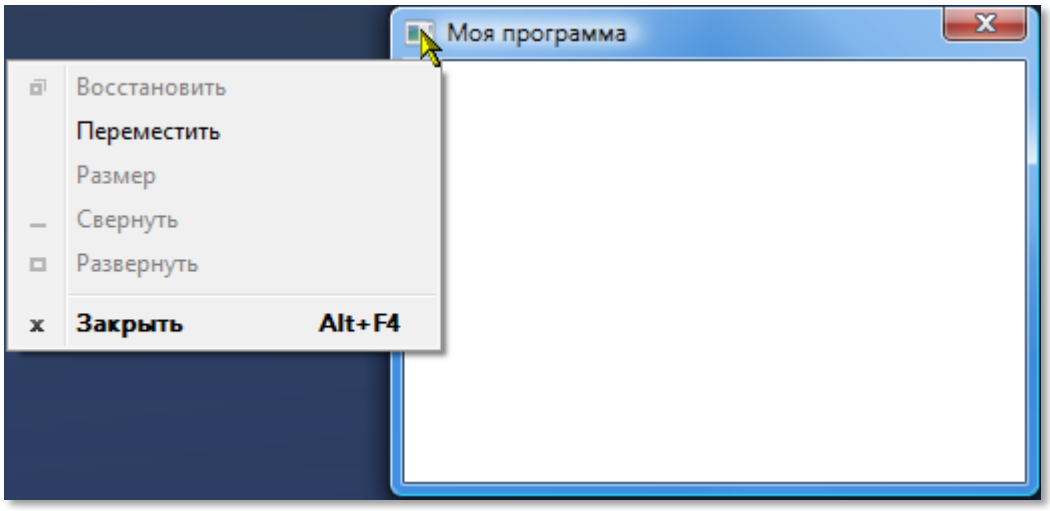


Рис. 14.4. Приложение строгих форм!

Это значение свойство *CanResize* имеет по умолчанию, поэтому, если вы не защитите программу, то пользователь безнаказанно сможет изменять размеры её окна.

Идём дальше по окнам. Если вы не позаботитесь сами, то окно будет иметь стандартные размеры 640 на 480 пикселей и располагаться у левого верхнего угла экрана. Этому горю легко помощь!

За *размеры окна* отвечают свойства с понятными названиями. Присвоив им нужные значения

```
GraphicsWindow.Width=400
GraphicsWindow.Height=200,
```

мы получим окно размером поменьше (Рис. 14.5).

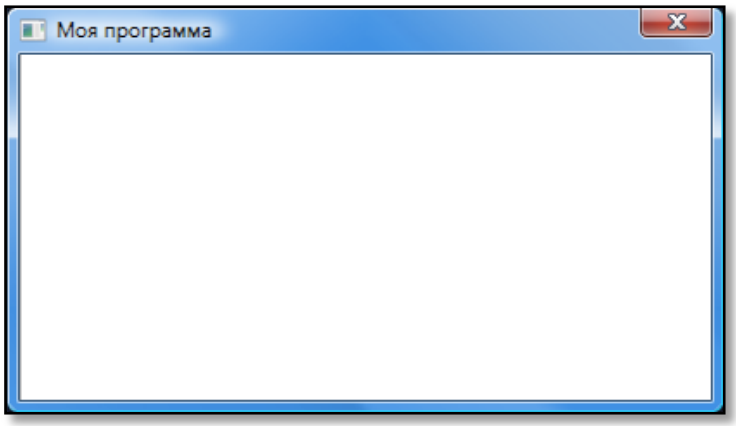


Рис. 14.5. Из окна мы сделали окошечко!

Можно сделать и «форточку» (Рис. 14.6).

```
GraphicsWindow.Width=0
GraphicsWindow.Height=0
```

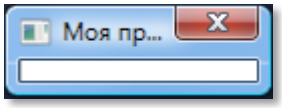


Рис. 14.6. Из окошечка мы сделали форточку!

Однако даже нулевые значения этих свойств не уменьшают окно до размеров точки!

Так же легко мы можем *установить окно* в любом месте экрана (Рис. 14.7), задав нужные координаты его левого верхнего угла:

```
GraphicsWindow.Left=400
GraphicsWindow.Top =400
```

Как установить окно приложения в центре *Рабочего стола* вы узнали на *Уроке 6*.

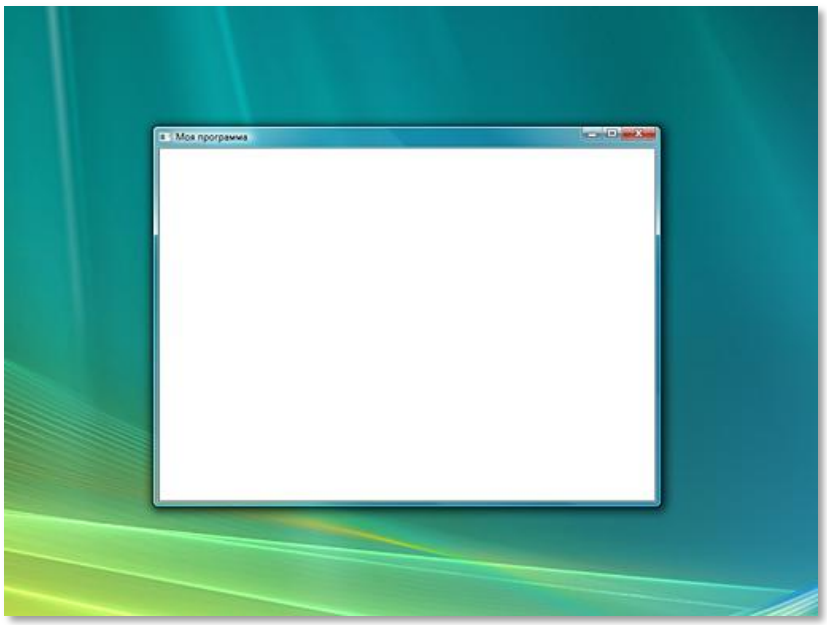


Рис. 14.7. В самое яблочко!

"Форменная" лихорадка

Давайте воспользуемся свойствами *Width* и *Height* окна и заставим его беспрерывно «ёрзать» по экрану. Сделать это проще простого. Для стимулирования окна нам потребуется объект *Таймер* (*Timer*). Задайте свойству *Interval* значение 200-400 миллисекунд и напишите процедуру для обработки срабатывания таймера:

```
Timer.Interval = 200
Timer.Tick = OnTimer
```

При срабатывании таймера, то есть через каждые 0,2-0,4 секунды будет вызываться процедура *OnTimer*:

```
rele=1
Sub OnTimer
  if rele=1 then
    GraphicsWindow.Left= GraphicsWindow.Left+ 5
    GraphicsWindow.Top= GraphicsWindow.Top+ 5
  else
    GraphicsWindow.Left= GraphicsWindow.Left- 5
    GraphicsWindow.Top= GraphicsWindow.Top- 5
  EndIf

  'переключить направление перемещения формы:
  rele= -rele
EndSub
```

В зависимости от значения переключателя *rele* окно немного сдвигается вниз-вправо, а потом возвращается на место. На статичном рисунке этого не покажешь, а в жизни бегающее окно выглядит очень забавно.

Заставьте перемещаться окно по всему экрану, чтобы разозлить пользователя не на шутку!

Исходный код программы находится в папке **GraphicsWindow**.

Элементы стандартного окна

Многие из них мы уже рассмотрели:

В верхней части окна находится полоска, которую обычно называют **заголовком**. Заголовок нужен для перемещения окна мышкой, а также на нём расположены:

- **Значок формы** – по умолчанию это стандартный значок *Windows* для приложений, и мы не можем заменить его своим.

- **Заголовок формы** – по умолчанию там выводится значение свойства *Title Графического окна*.

- **Кнопка сворачивания окна** (минимизации)  – окно превращается в кнопку на *Панели задач*, которая из нажатого положения (активное окно) переходит в отжатое. Чтобы активизировать окно, достаточно щёлкнуть по его кнопке на *Панели задач*.

- **Кнопка разворачивания окна** (максимизации)  – окно раскрывается во весь экран (за исключением *Панели задач*, если не изменены её свойства), а вместо этой кнопки появляется **кнопка восстановления окна**  , при нажатии на которую окно принимает прежние размеры.

- **Кнопка закрытия окна**  – окно исчезает с экрана и работа приложения завершается.

Системное меню окна – появляется, когда вы нажимаете правую кнопку мыши на заголовке окна. Оно дублирует уже рассмотренные нами команды управления окном.

Граница окна – весь контур окна позволяет изменять его размеры. Для этого нужно поставить курсор мыши на границу окна и, когда курсор примет вид двойной стрелочки  , потянуть её в нужную сторону.

Клиентская область – всё остальное пространство окна.

Светоформа

Вы должны были обратить внимание, что клиентская область окна окрашена в *белый* цвет, но некоторые любят и другие цвета. Что ж, очень просто удовлетворить желания самых изощренных любителей прекрасного. Для этого свойству **BackgroundColor** следует задать нужный цвет (Рис. 14.8):

```
GraphicsWindow.BackgroundColor="Green"
```

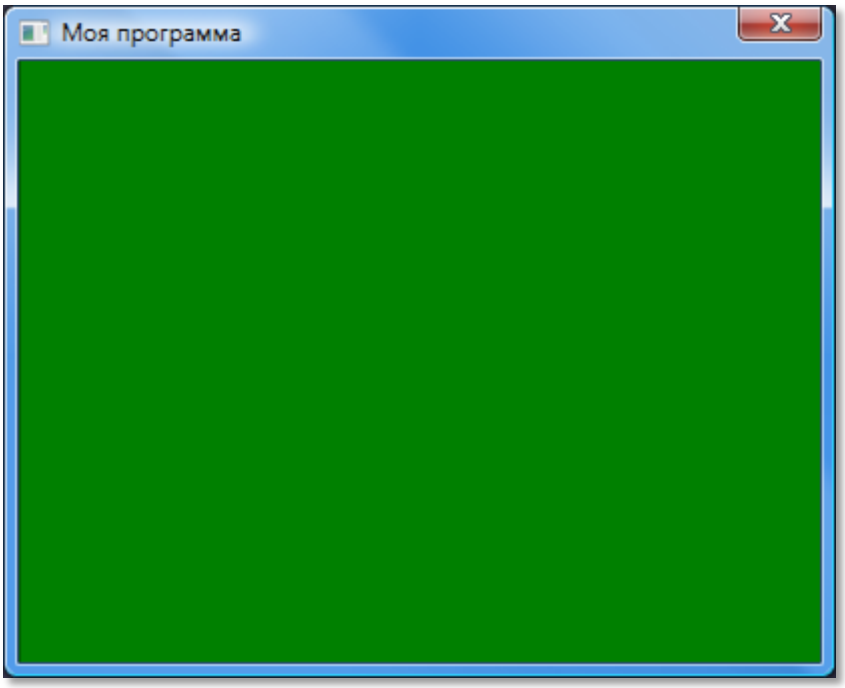


Рис. 14.8. Окно **позеленело**!

А теперь давайте превратим окно в **светофор** - чтобы оно переливалось всеми цветами радуги. Достаточно немного изменить процедуру из предыдущей программы, обрабатывающую «тиканье» таймера:

```
'Мигание окна
rele=1
Sub OnTimer
  GraphicsWindow.BackgroundColor=
GraphicsWindow.GetRandomColor()
  if rele=1 then
    GraphicsWindow.Show()
  else
    GraphicsWindow.Hide()
```

```
  EndIf
  rele=-rele

EndSub
```

После старта программы окно будет то исчезать, то вновь появляться на экране – и каждый раз в другой «одёжке».

Обратите внимание на новый метод *Графического окна* **GetRandomColor,** который возвращает «случайный» цвет. Если вам безразлична последовательность смены цветов, то это самый быстрый и простой способ получить произвольный цвет.

Ещё немного изменим программу, и окно будет не только перекрашиваться, но и *изменять свои размеры*:

```
Timer.Interval = 10
Timer.Tick = OnTimer

'Мигание  и изменение размеров окна
rele=1
n=1
Sub OnTimer
  if n= 100 Then
    rele=-rele
    n=0
    GraphicsWindow.BackgroundColor=
GraphicsWindow.GetRandomColor()
  else
    n=n+1
  EndIf

  GraphicsWindow.Width=GraphicsWindow.Width + rele
  GraphicsWindow.Height = GraphicsWindow.Height + rele
  GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width)
/ 2
  GraphicsWindow.Top = (Desktop.Height -
GraphicsWindow.Height) / 2

EndSub
```

Для задания цвета окна можно использовать и метод **GetColorFromRGB**, который преобразует составляющие цвета (красную - **Red**, зелёную - **Green** - и синюю - **Blue**) в новый цвет, который можно использовать в *СБ*:

```
'Изменение цвета окна
Sub OnTimer
  red= Math.GetRandomNumber(256)-1
  green= Math.GetRandomNumber(256)-1
  blue= Math.GetRandomNumber(256)-1

GraphicsWindow.BackgroundColor=GraphicsWindow.GetColorFromRGB(
red, green, blue)

EndSub
```

Исходный код программы находится в папке **GraphicsWindow**.

Свойство **LastKey** *Графического окна* содержит название последней нажатой на клавиатуре кнопки (Рис. 14.9):

```
While "True"
  GraphicsWindow.Title= GraphicsWindow.LastKey
EndWhile
```

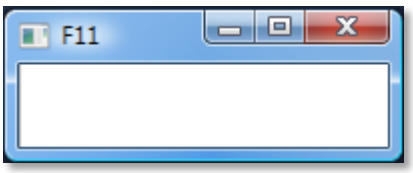


Рис. 14.9. Догадайтесь с трёх раз, какая клавиша была нажата?

Так как класс *Графического окна* имеет очень много методов и свойств, то его изучение мы продолжим на следующем уроке.

Урок 15. Текст в *Графическом окне*

Забавно было выводить информацию в заголовок окна, но куда красивее текст можно напечатать в самом окне. И для этого *СБ* предоставляет в наше полное распоряжение все, что нужно.

Вы можете печатать текст *любым* шрифтом, установленным на вашем компьютере. Достаточно присвоить свойству **FontName** название шрифта (Рис. 15.1):

```
GraphicsWindow.FontName="Arial"
```

Не пользуйтесь редкими шрифтами, если планируете передавать программу другим пользователям! Если у них такого шрифта не окажется, он будет заменён стандартным (Рис. 15.2).

Рис. 15.1. Надпись шрифтом *Arbat*

Рис. 15.2. Шрифт *Arbat* не установлен

Практически на всех компьютерах стоят такие шрифты (Рис. 15.3).

Вот программа, которая напечатала все эти надписи:

```
GraphicsWindow.BackgroundColor="Black"
GraphicsWindow.FontName="Arial"

For i= 0 To 6
  GraphicsWindow.FontSize= 12+ i*2
```

```
  GraphicsWindow.BrushColor= GraphicsWindow.GetRandomColor()
  GraphicsWindow.DrawText(10,10+i*30, "Text Text")
EndFor
```

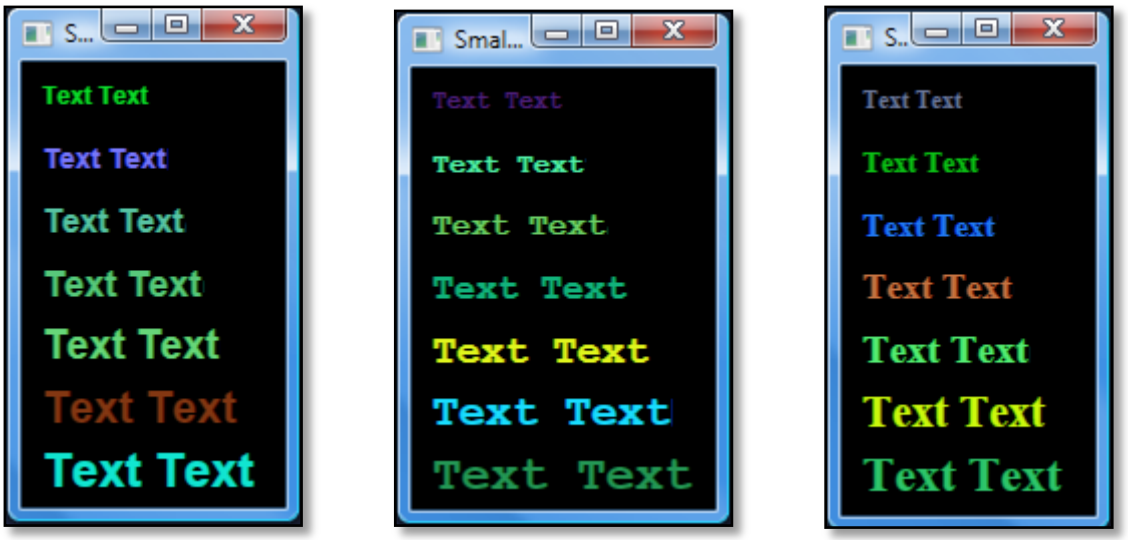


Рис. 15.3. *Arial* *Courier New* *Times New Roman*

Как видите, размер шрифта определяется свойством **FontSize**, а его цвет – свойством **BrushColor**. Сам же текст выводится методом **DrawText(x,y,строка)**.

Первое число в скобках – координата начала строки в пикселях, которая отсчитывается от левого края клиентской области окна.

Второе число – координата верхней границы строки, которая отсчитывается от верхнего края клиентской области. Затем следует указать числовую или текстовую информацию («строку») для ввода на экран.

По умолчанию все надписи выполняются **жирным** шрифтом. Если вам больше по душе тонкие буквы, измените свойство **FontBold** (Рис. 15.4) :

```
GraphicsWindow.FontBold= "False"
```

Буквы снова станут **жирными**, если вы вернёте этому свойству прежнее значение:

```
GraphicsWindow.FontBold= "True"
```

Рис. 15.4. Похудевший *Arial*

А такая строка заставит склониться все буквы в почтении (Рис. 15.5):

```
GraphicsWindow.FontItalic= "True"
```

Рис. 15.5. *Курсивный* шрифт

Но вы легко вернёте им самоуважение:

```
GraphicsWindow.FontItalic= "False"
```

Для печати текста «в рамку» предназначен метод **DrawBoundText(x,y, width, строка)**, который отличается от рассмотренного нами метода *DrawText* только тем, что он имеет до-

полнительный параметр *width*, который определяет максимальную ширину текста в окне.

Попробуем напечатать длинную строку цифр, ограничив её длину:

```
  GraphicsWindow.FontSize= 24
  GraphicsWindow.BrushColor= "Yellow"
  GraphicsWindow.FontBold= "False"
  GraphicsWindow.BrushColor= GraphicsWindow.GetRandomColor()
  GraphicsWindow.DrawBoundText(10,10+i*30, 150,
"123456789012345678901234567890")
```

Запустив программу, мы увидим только первые девять цифр и многоточие, которое сигнализирует нам о том, что часть строки напечатать не удалось (Рис. 15.6).

Рис. 15.6. Текст вышел за рамку!

Раздвинем границы рамки (Рис. 15.7):

```
GraphicsWindow.DrawBoundText(10,10+i*30,350,
"123456789012345678901234567890")
```

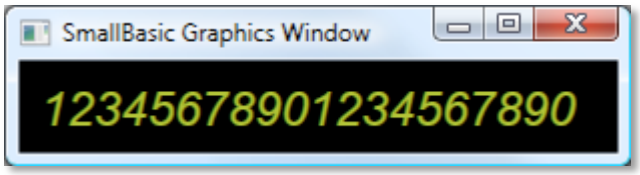


Рис. 15.7. А вот теперь все нормально!

Исходный код программы находится в папке **Текст в Графическом окне**.

События *Графического окна*

Операционная система *Windows* управляет всеми приложениями с помощью *сообщений*, которые она им посылает. Эти сообщения принимает окно приложения и реагирует на них. Например, если пользователь нажмёт какую-нибудь клавишу, система *Windows* посылает активному окну сообщение *WM_KEYDOWN*. Если это наше графическое окно, то в нём возникает событие *KeyDown*. Если нас это событие интересует, мы можем написать процедуру-обработчик для этого события и указать *Графическому окну* её название:

```
GraphicsWindow.KeyDown= OnKeyDown

Sub OnKeyDown
  GraphicsWindow.ShowMessage("Вы нажали клавишу", "СОБЫТИЯ")
EndSub
```

Запускаем программу, и как только мы нажмём клавишу, получим окно с приятным известием (Рис. 15.8).

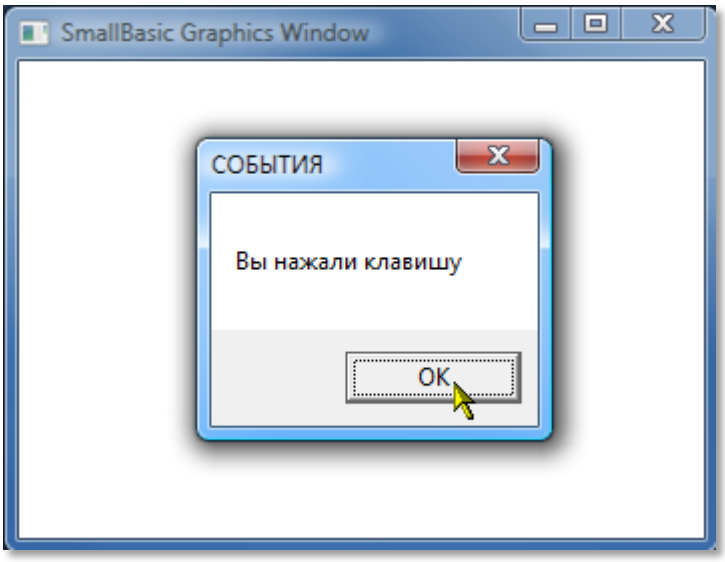


Рис. 15.8. Вам «письмо»!

Мы легко узнаем, какая именно клавиша была нажата, прочитав свойство *LastKey* (Рис. 15.9).

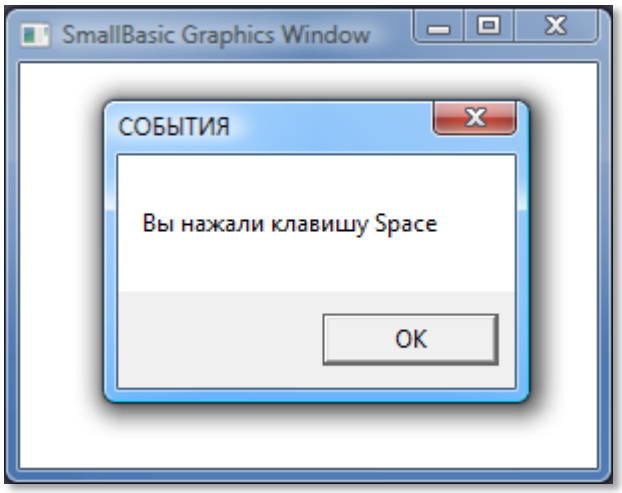


Рис. 15.9. Система знает всё!

При отпускании клавиши возникает событие *KeyUp*, которое мы также можем обработать с помощью процедуры (Рис. 15.10):

```
GraphicsWindow.KeyUp= OnKeyUp

Sub OnKeyUp
  GraphicsWindow.ShowMessage("Вы отпустили клавишу " +
GraphicsWindow.LastKey, " СОБЫТИЯ")
EndSub
```

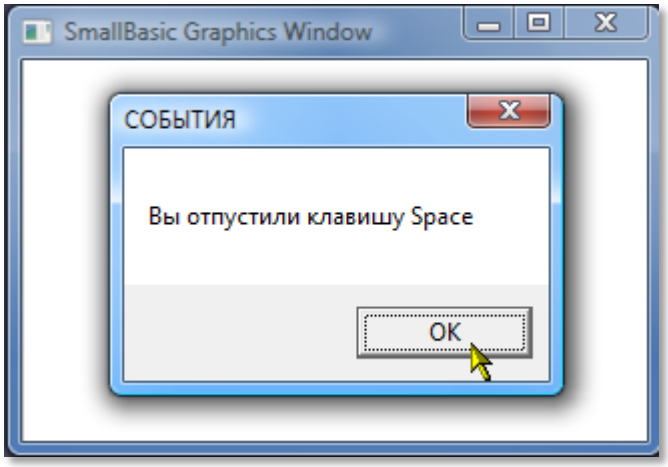


Рис. 15.10. Она и это знает!

Попутно мы познакомились с методом *Графического окна* **ShowMessage(строка, заголовок)**, который очень часто используется для вывода информации, когда нужно приостановить выполнение программы, например, при отладке.

Первый параметр этого метода – строка с сообщением, а *второй* – заголовок окна, который обычно сообщает его назначение.

Программа *Розыгрыш*

Обогатившись знаниями о событиях, мы можем написать небольшую программу, которая будет самым живым образом реагировать на действия доверчивого пользователя:

```
'Программа "РОЗЫГРЫШ""

GraphicsWindow.Width=490
GraphicsWindow.Height=60
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.Title= " Нажмите клавишу!"
GraphicsWindow.BackgroundColor="Black"
GraphicsWindow.FontSize= 24

GraphicsWindow.KeyDown= OnKeyDown
'GraphicsWindow.KeyUp= OnKeyUp

Sub OnKeyDown
  GraphicsWindow.Clear()
  GraphicsWindow.BrushColor= "Red"
  GraphicsWindow.DrawText(10,10," Не нажимайте клавишу, мне
больно!")
EndSub

Sub OnKeyUp
  GraphicsWindow.Clear()
  GraphicsWindow.BrushColor= "Green"
  GraphicsWindow.DrawText(10,10," Спасибо, что вы отпустили
клавишу!")
EndSub
```

Когда пользователь нажмет клавишу, в окне появится жалобное сообщение (Рис. 15.11).

Рис. 15.11. А, может, это правда?!

Гуманный пользователь тут же отпустит клавишу, за что получит благодарность (Рис. 15.12).

Рис. 15.12. Никогда не делайте больно!

Если пользователь долго держит клавишу нажатой и мучает *графическое окно*, значит, ему пора на урок психологии!

Обратите внимание на метод **Clear**, который очищает окно от любых надписей и рисунков, закрашивая клиентскую область в цвет *BackgroundColor*.

Как видите, ничего нового в программе нет, а получилось смешно!

Исходный код программы находится в папке **Текст в Графическом окне**.

Розыгрыши продолжаются!

Поскольку кроме клавиатуры, у пользователя всегда под рукой *мышка*, то мы вполне можем предположить, что *графическое окно* реагирует и на мышиную «возню». И мы не ошиблись в наших ожиданиях: реагирует, да ещё как!

Мы не можем и не должны останавливаться на констатации сего приятного факта, а наоборот, мы сразу же примемся использовать мышиные события для продолжения розыгрышей.

Снова будем выводить в окно смешные надписи. При нажатии и отпускании кнопки мыши в окне одна надпись будет сменяться другой. На этот раз мы возьмём замечательные выражения из «негритянских» рассказов юмориста Лукинского.

```
GraphicsWindow.Width=300
GraphicsWindow.Height=60
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.Title= "Нажми кнопку мышки!"
GraphicsWindow.BackgroundColor="White"
GraphicsWindow.FontSize= 32

GraphicsWindow.MouseDown= OnMouseDown
GraphicsWindow.MouseUp= OnMouseUp

Sub OnMouseDown
  GraphicsWindow.Title= "Отпусти кнопку мышки!"
  GraphicsWindow.Clear()
  GraphicsWindow.BrushColor= "Blue"
  GraphicsWindow.DrawText(10,10," С Новым годом!")
EndSub

Sub OnMouseUp
  GraphicsWindow.Title= "Нажми кнопку мышки!"
  GraphicsWindow.Clear()
  GraphicsWindow.BrushColor= "Green"
  GraphicsWindow.DrawText(10,10," Пошёл на фиг!")
EndSub
```

Отозвавшись на просьбу окна, выраженную в его заголовке, доверчивый пользователь нажмёт кнопку мышки и получит поздравление с праздником (Рис. 15.13).

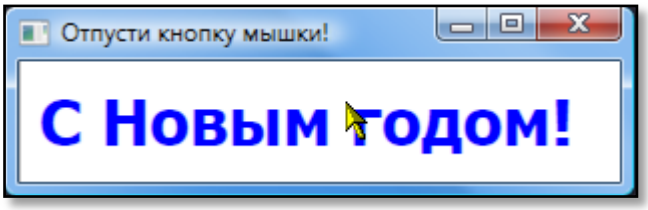


Рис. 15.13. Спасибо!

Теперь окно попросит его отпустить кнопку мышки и недобро пошлет его в другое приложение (Рис. 15.14).

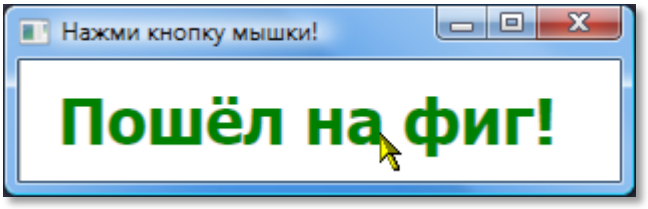


Рис. 15.14. Ну, Лукинский даёт!

Согласитесь, жить стало веселей. А вы можете повеселиться и пуще того, если заготовите множество таких фраз и подсунете их своим товарищам, близким и родственникам. Они непременно будут вам благодарны за доставленное удовольствие.

Исходный код программы находится в папке **Текст в Графическом окне.**

Прикольное окно

Попробуем сделать окно «прикольным». Прикол будет незатейливым: при попытке пользователя переместить окно, оно упрямо будет возвращаться в середину экрана (Рис. 15.15).

```
GraphicsWindow.Title=" Прикол"

GraphicsWindow.Width=420
GraphicsWindow.Height=60
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
```

```
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.FontSize= 32
GraphicsWindow.BrushColor= "Blue"
GraphicsWindow.CanResize="False"

GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2

GraphicsWindow.BackgroundColor="Black"
GraphicsWindow.DrawText(10,10," Передвинь меня в угол!")

Timer.Interval = 10
Timer.Tick = OnTimer

Sub OnTimer
  GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width)
/ 2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
EndSub
```

Рис. 15.15. Окно на приколе

Работает программа очень просто. Каждые 10 миллисекунд срабатывает таймер, и процедура-обработчик *OnTimer* возвращает окно на место.

Ввод текста в *Графическом окне*

Графическое окно реагирует ещё на одно событие, которое может пригодиться вам, например, для ввода информации с клавиатуры.

Когда пользователь нажимает клавишу, возникает событие *TextInput*, а свойство **LastText** содержит символ, соответствующий этой клавише, - букву, цифру, знак препинания и т.д. В процедуре-обработчике вы можете составить из этих символов строку (Рис. 15.16), а затем вывести на экран или использовать в программе как-то иначе.

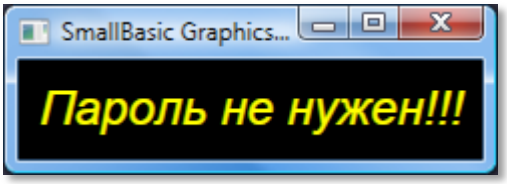


Рис. 15.16. Строка, собранная по буквам!

```
s=""
GraphicsWindow.FontSize= 24
GraphicsWindow.BrushColor= "Yellow"
GraphicsWindow.FontBold= "False"
GraphicsWindow.TextInput =OnText

Sub OnText
  GraphicsWindow.Clear()
  s= s+GraphicsWindow.LastText
  GraphicsWindow.DrawText(10,10, s)
EndSub
```

В данном примере после ввода каждой буквы приходится стирать всё окно, иначе надписи накладываются друг на друга, что выглядит неаккуратно. Скоро мы познакомимся с геометрическими фигурами, которые можно рисовать в *графическом окне*, и они помогут нам стирать любую часть клиентской области, не причиняя вреда окружающей среде.

Исходный код программы находится в папке **Текст в Графическом окне**.

Урок 16. Класс *Math*

В *СБ* над числовыми константами и переменными можно производить различные *арифметические действия*:

Знак операции	Операция
+	Сложение
-	Вычитание
*	Умножение
/	Деление

Вы, наверное, заметили, что некоторые знаки арифметических операций отличаются от тех, что вы используете в школе: знак умножения в виде крестика или точки здесь заменен *звёздочкой*, а двоеточие – знак деления – *дробной чертой* (её также называют *слэшем*). Это объясняется тем, что знака умножения нет на клавиатуре, а двоеточие используется для других целей.

Минус служит также для *изменения знака* выражения противоположным: 2 → -2; x → -x.

Рассмотрим *примеры*. Пусть

a = 6
b = 3

Действие	Значение переменной Num
Num = a + b	9
Num = a - b	3
Num = a * b	18
Num = a / b	2

При вычислении выражений учитывается *приоритет* операций – точно так же, как и в математике. Сначала слева направо выполняются операции умножения и деления, затем сложения и вычи-

тания. Изменить *порядок* вычисления можно с помощью *круглых скобок* (не квадратных и не фигурных!) – в полном согласии с законами арифметики:

Действия	**Значение переменной Num**
Num = (2 + 4 - 1) / 5 * 11	11
Num = (3.1415 – 1.0) / 7.78 * (45 - 7)	10,459768637532133676092544987
Num = -2.67 * 3.14 + -3/2 – 7.01	-16,8938
Num = 3 * 4 + 5 – 6/2	14

Квадратные скобки нужны для обозначения элементов массива, а фигурные скобки {} в *СБ* не применяются вообще.

Между числами и знаками операций может быть сколько угодно пробелов, но можно печатать их и вплотную друг к другу.

Для «научных» вычислений в *СБ* имеется специальный класс **Math**, который имеет единственной свойство *Pi*, значение которого, как легко догадаться, равно числу π. Зато методов у него вполне достаточно, чтобы выполнять довольно сложные расчеты.

Чтобы вам было удобнее пользоваться методами класса *Math*, мы сведём их в *таблицу*.

Большинство из этих функций вам известны по школе, поэтому мы не будем рассматривать их подробно, а напишем программу **Класс Math** с примерами вычислений. В случае необходимости вы сможете подставить свои значения в эти функции, чтобы лучше изучить их работу.

Pi	3,14159265358979
Abs(число)	Возвращает *абсолютную величину* числа
ArcCos(косинус)	Возвращает угол в радианах, соответствующего значению *косинуса*
ArcSin(синус)	Возвращает угол в радианах, соответствующего значению *синуса*
ArcTan(тангенс)	Возвращает угол в радианах, соответствующего значению *тангенса*
Ceiling(число)	*Округляет* заданное дробное число до большего целого.
Cos(угол)	Возвращает *косинус* угла, заданного в радианах
Floor(число)	*Округляет* заданное дробное число до меньшего целого.
GetDegrees(угол)	Пересчитывает угол, заданный в радианах, в *градусы*
GetRadians(угол)	Пересчитывает угол, заданный в градусах, в *радианы*
GetRandomNumber(максимальное число)	Возвращает *случайное число* от 1 до максимального числа, включительно. Выражение GetRandomNumber(30) вернет одно из чисел 1, 2, 3, … , 30
Log(число)	Возвращает *десятичный логарифм* заданного числа
Max(число1, число2)	Возвращает *большее* из двух чисел
Min(число1, число2)	Возвращает *меньшее* из двух чисел
NaturalLog(число)	Возвращает *натуральный логарифм* заданного числа
Power(число, степень)	Возвращает результат возведения числа в заданную *степень*
Remainder(число1, число2)	Возвращает *остаток от деления* первого числа на второе
Round(число)	*Округляет* заданное дробное число до ближайшего целого.
Sin(угол)	Возвращает *синус* угла, заданного в радианах
SquareRoot(число)	Возвращает *корень квадратный* из числа
Tan(угол)	Возвращает *тангенс* угла, заданного в радианах

```
GraphicsWindow.Title=" Класс Math"

'const
ps= Text.GetCharacter(13) + Text.GetCharacter(10)
BackgroundColor="#2F4F4F"

GraphicsWindow.Width= 300
GraphicsWindow.Height=400
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"
GraphicsWindow.BackgroundColor= BackgroundColor

txtMath=Controls.AddMultiLineTextBox(10,10)
Controls.SetSize(txtMath, GraphicsWindow.Width-20,
GraphicsWindow.Height-20)
```

Результаты вычислений мы будем выводить в *многострочное текстовое поле*. Это красиво, но нам нужно учесть, что поле хоть и многострочное, но строка для вывода должна быть *одна*. Возникает вопрос: а как же длинную строку разбить на отдельные, более короткие строки? – Очень просто: после каждой строчки нужно дописать символы перехода на новую строку. Так как эти символы непечатные, то их можно получить только по их коду:

```
ps= Text.GetCharacter(13) + Text.GetCharacter(10)
```

Формируем строку из пояснительного текста и результатов вычислений:

```
s= "Пи = " + Math.Pi + ps
s= s + "Abs(-2) = " + Math.Abs(-2) + ps
s= s + "ArcCos(0.5) = " + Math.ArcCos(0.5) + ps
s= s + "ArcSin(0.5) = " + Math.ArcSin(0.5) + ps
s= s + "ArcTan(0.5) = " + Math.ArcTan(0.5) + ps
s= s + "Ceiling(0.5) = " + Math.Ceiling(0.5) + ps
s= s + "Ceiling(1) = " + Math.Ceiling(1) + ps
s= s + "Cos(0.5) = " + Math.Cos(0.5) + ps
s= s + "Floor(0.5) = " + Math.Floor(0.5) + ps
s= s + "Floor(1) = " + Math.Floor(1) + ps
```

```
s= s + "GetDegrees(0.5) = " + Math.GetDegrees(0.5) + ps
s= s + "GetRadians(30) = " + Math.GetRadians(30) + ps
s= s + "GetRandomNumber(30) = " + Math.GetRandomNumber(30) +
ps
s= s + "Log(100) = " + Math.Log(100) + ps
s= s + "NaturalLog(100) = " + Math.NaturalLog(100) + ps
s= s + "Max(-1, -2) = " + Math.Max(-1, -2) + ps
s= s + "Min(-1, -2) = " + Math.Min(-1, -2) + ps
s= s + "Power(2, 32) = " + Math.Power(2, 32) + ps
s= s + "Remainder(15, 4) = " + Math.Remainder(15, 4) + ps
s= s + "Round(0.5) = " + Math.Round(0.5) + ps
s= s + "Round(1.6) = " + Math.Round(1.6) + ps
s= s + "Sin(0.5) = " + Math.Sin(0.5) + ps
s= s + "SquareRoot(2) = " + Math.SquareRoot(2) + ps
s= s + "Tan(0.5) = " + Math.Tan(0.5) + ps
```

Осталось напечатать её в *текстовом поле* (Рис. 16.1):

```
Controls.SetTextBoxText(txtMath, s)
```

Рис. 16.1. Все методы класса *Math* в одном окошке!

Методы класса *Math* представляют собой *функции*, которые *возвращают* результат вычислений. Это значит, что, например, выражение *Math.Sin(Math.GetRadians(30))* равно 0,5. Обычно результат, возвращаемый функцией, *присваивают* переменной:

```
sinus= Math.Sin(Math.GetRadians(30))
```

После выполнения этого оператора значение переменной *sinus* будет равно 0,5.

Иногда вызовы функций используют только для того, чтобы *сравнить* возвращаемое значение с каким-либо другим значением:

```
if (Math.Sin(Math.GetRadians(30)) = 0.5) Then
  . . . . .
EndIf
```

Можно сделать проверку и так:

```
sinus= Math.Sin(Math.GetRadians(30))

if (sinus = 0.5) Then
  . . . . .
EndIf
```

Но тогда потребуется лишняя переменная и лишняя операция присваивания. Впрочем, для удобства нередко так и делают, потому что составление длинных операторов может привести к ошибкам, которые непросто выявить.

Исходный код программы находится в папке **Класс Math**.

Вычисление числа *π*

А теперь давайте воспользуемся методами класса *Math*, чтобы вычислить значение числа *пи* (поскольку оно выражается бесконечной десятичной дробью, то мы сможем найти только приближенное значение).

Современное обозначение этого числа греческой буквой *пи* предложил в 1706 году английский математик У.Джонсон. Он воспользовался первой буквой греческого слова *periferia* (конечно, в оригинальном написании, а не более удобными латинскими буквами), что значит *окружность*. Но общепризнанным в научном мире этот символ стал после того как в 1736 году Леонард Эйлер использовал его в своих работах.

История вычислений числа пи, которое равно отношению длины окружности к её диаметру, началась много тысячелетий назад.

Так, египетские математики считали *пи* равным $(16/9)^2$, то есть примерно 3,1604938..., а индийские – $\sqrt{10}$ = 3,16227766... В третьем веке до нашей эры Архимед установил, что *пи* меньше, чем 3 1/7, и больше, чем 3 10/71 = 3,1428 ... 3,1408. Среднее значение этого диапазона равно 3,14185, что больше *пи* уже в четвертом десятичном знаке. Но ещё до Архимеда, в пятом веке до нашей эры китайский математик Цзу Чунчжи нашел более точное значение - 3,1415927... В первой половине пятнадцатого века ал-Каши, астроном и математик из Самарканда вычислил *пи* с точностью до 16 знаков после запятой. В 1615 году голландский математик Лудольф ван Цейлен довёл точность вычислений до 32 знаков. В 1873 году Вильям Шенкс после 20 лет расчётов нашёл 707 знаков числа *пи*, но в 1944 году Д.Фергюсон с помощью механического калькулятора выяснил, что верны только первые 527 знаков числа Шенкса. Для своих расчетов Шенкс использовал *формулу Дж. Мачина*:

$$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

А *арктангенс* вычислял по формуле

$$\arctan x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{2k+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

Сам Мачин еще в 1706 году по этой формуле вычислил 100 знаков числа *пи*.

С появлением ЭВМ скорость вычислений значительно выросла. В 1949 году электронная машина ЭНИАК за 70 часов работы вычислила более двух тысяч десятичных знаков числа *пи*. Через некоторое время были найдены 3000 знаков всего за 13 минут. В 1959 году компьютеры преодолели рубеж десяти тысяч знаков, а сейчас известно несколько десятков миллионов знаков числа *пи*. Чтобы их напечатать, потребуется несколько толстенных книг.

Формула Валлиса

$$\pi = 2\left(\frac{2}{1}\times\frac{2}{3}\times\frac{4}{3}\times\frac{4}{5}\times\frac{6}{5}\times\frac{6}{7}\times\frac{8}{7}\times\frac{8}{9}\cdots\right)$$

Джон Валлис вывел эту красивую формулу в 1655 году, когда вычислял площадь круга. Она представляет собой бесконечное произведение дробей. К сожалению, чтобы вычислить даже несколько правильных знаков числа *пи* по этой формуле, нужно затратить немало времени и сил. Однако, имея компьютер, мы можем облегчить себе задачу. Итак, пишем программу:

```
GraphicsWindow.Title=" Вычисление числа пи"

'const
ps= Text.GetCharacter(13) + Text.GetCharacter(10)
BackgroundColor="#2F4F4F"

GraphicsWindow.Width= 640
GraphicsWindow.Height=480
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"
GraphicsWindow.BackgroundColor= BackgroundColor

txtMath=Controls.AddMultiLineTextBox(10,10)
```

```
Controls.SetSize(txtMath, GraphicsWindow.Width-20,
GraphicsWindow.Height-20)
'Вычисление по формуле Валлиса:
pi=1
For n= 1 To 1000000
  pi= pi * (4*n*n/(2*n-1)/(2*n+1))
EndFor

s= "Пи = " + 2* pi + ps
Controls.SetTextBoxText(txtMath, s)
```

После миллиона итераций получаем ответ:

nu = 3,14159186819212071459647663 54

Да! Уже шестой знак после запятой неверный.

Ряд Лейбница

Попробуем зайти с другого конца и воспользуемся знакочередующимся рядом, который предложил немецкий математик Лейбниц. А ряд вот такой:

$$\pi = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \cdot \cdot \cdot\right)$$

Он хорош тем, что его легко приспособить под наши нужды:

```
'Вычисление по формуле Лейбница:
pi=0
For n= 0 To 1000001
  pi= pi + Math.Power(-1, n) / (2*n+1)
EndFor

s= "Пи = " + 4* pi + ps
Controls.SetTextBoxText(txtMath, s)
```

После миллиона итераций получаем результат:

Пи = 3,14159165359179323471265007 40

К сожалению, не лучше первого!

Всего существует не один десяток формул для вычисления *пи*, но они слишком сложны для занимательных уроков.

Исходный код программы находится в папке **Вычисление пи**.

Урок 17. Компьютерная графика

Лучше один раз увидеть,
чем сто раз услышать.
Программистская мудрость

Вся школьная геометрия построена из теорем, но мы их доказывать не будем, а просто собственными глазами убедимся, что из простых геометрических фигур с помощью компьютера можно создавать впечатляющие композиции.

Клиентская область *графического окна* выполняет ту же роль в *компьютерной графике*, что и холст в настоящей живописи. Нам уже довелось писать на ней разные словечки, а теперь мы возьмёмся за настоящее геометрическое творчество.

Пуантилизм, или Ставим точки

Мостовая пусть качнётся, как очнётся!
Пусть начнётся, что ещё не началось!
Вы рисуйте, вы рисуйте,
вам зачтётся...
Что гадать нам:
удалось - не удалось?

Булат Окуджава. *Живописцы*

Самым простым геометрическим объектом является *точка*. Она, как известно из геометрии, размеров не имеет. По той же причине она не имеет ни цвета, ни запаха. Таким образом, настоящую точку увидеть нельзя, а вот компьютерную - можно, хотя размером она примерно в четверть миллиметра. Называется такая точка **пикселем**. Пиксель мал да удал: его можно окрасить в миллионы цветов, а из множества пикселей мы сумеем нарисовать на экране монитора любую картину.

Метод **SetPixel(x, y, color)** *Графического окна* закрашивает пиксель клиентской области окна (для краткости мы будем назы-

вать её *холстом* или *канвой*) в заданный цвет. Цвет можно задать любым способом из тех, что мы уже рассмотрели, а вот с координатами всё не так просто, как в школьной геометрии.

Расположение координатных осей на канве может показаться странным: ось *Y* (ордината) направлена вниз, а не вверх, поэтому, чем *больше* значение *Y*, тем *ниже* будет точка на экране. Положительное направление оси *X* совпадает с нашими представлениями, а вот начало координат находится в *левом верхнем* углу канвы, а вовсе не в её центре, как мы могли бы того ожидать (Рис. 17.1).

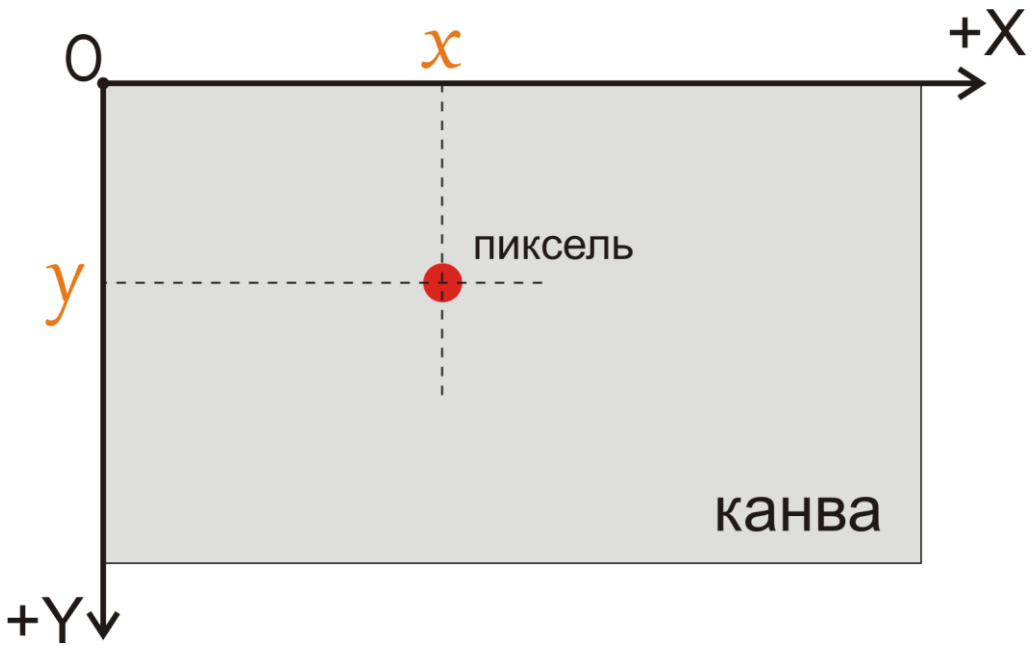


Рис. 17.1. Координатная система клиентской части окна приложения

Конечно, рисовать точками лучше в каком-нибудь графическом редакторе (*Microsoft Paint* или *Image Editor*, например, вполне годятся для этой цели), чем в программе на *СБ*, но поскольку все фигуры построены из точек, то давайте напишем простенькую программу, которая будет самостоятельно выводить на канву точки случайно выбранного цвета. Эстетического удовольствия мы не получим никакого, но зато хорошенько познакомимся с методом *SetPixel*. Программа **Пиксели** просто окрашивает точки

канвы в случайные цвета, поэтому картинка получается совершенно хаотическая (Рис. 17.2).

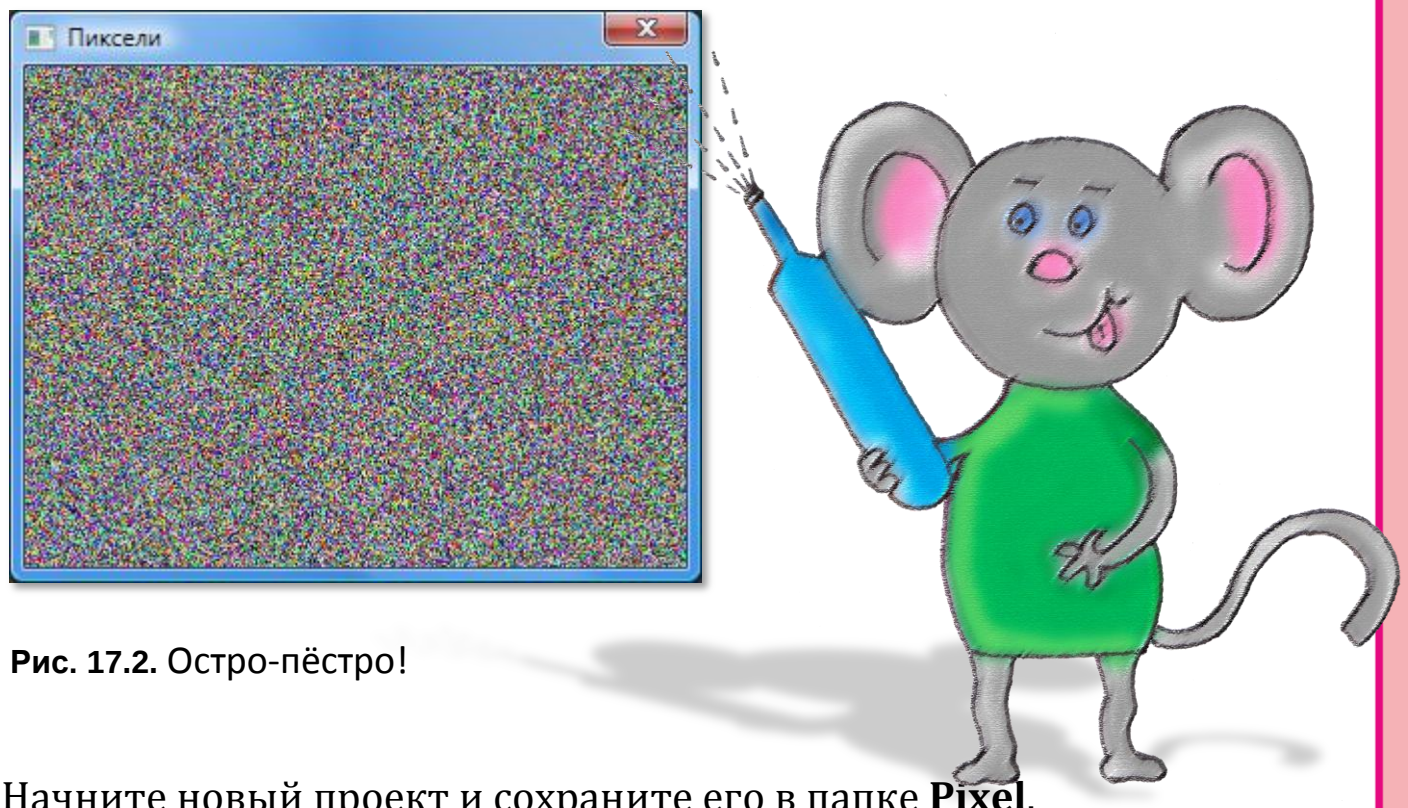


Рис. 17.2. Остро-пёстро!

Начните новый проект и сохраните его в папке **Pixel**.

Поскольку рисование точек на канве процесс довольно неспешный, то не задавайте большие размеры окна. Результат будет ничем не лучше, а времени вы потратите больше!

```
GraphicsWindow.Title=" Пиксели "

GraphicsWindow.Width= 320
GraphicsWindow.Height=240
GraphicsWindow.Left= (Desktop.Width-GraphicsWindow.Width)/ 2
GraphicsWindow.Top = (Desktop.Height-GraphicsWindow.Height)/ 2
GraphicsWindow.CanResize="False"
GraphicsWindow.BackgroundColor="Black"
```

В бесконечном цикле *While* программа последовательно перебирает все точки канвы и окрашивает их с помощью метода *SetPixel*:

```
' Окрашиваем пиксели канвы в разные цвета
height=GraphicsWindow.Height
width=GraphicsWindow.Width
While "True"
```

```
  for y=0 to height-1
    for x=0 to width-1
      'выбираем случайный цвет для пикселя:
      clr= GraphicsWindow.GetRandomColor()
      'и окрашиваем его:
      GraphicsWindow.SetPixel(x,y,clr)
    EndFor
  EndFor
EndWhile
```

Добавим к нашему проекту несколько строчек кода, чтобы получить интересный визуальный эффект: надпись *Пиксели* будет печататься *над* цветными точками (Рис. 17.3).

```
. . .
GraphicsWindow.FontSize= 56
GraphicsWindow.BrushColor = "Red"
. . .
GraphicsWindow.SetPixel(x,y,clr)
GraphicsWindow.DrawText(40,height/2-40,"Пиксели")
```

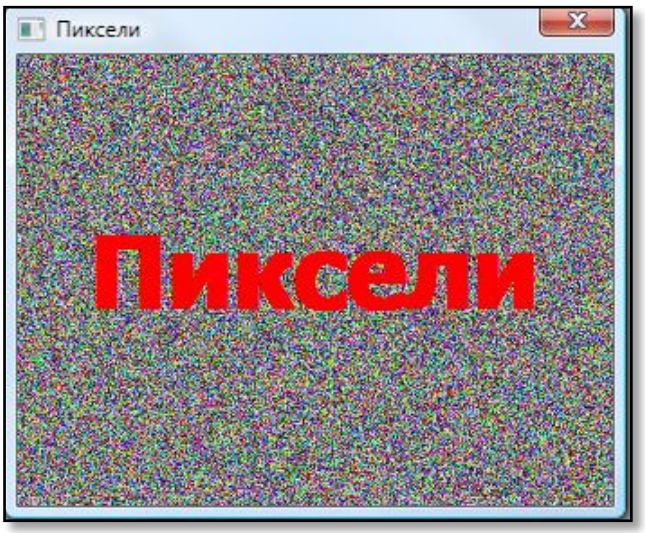


Рис. 17.3. Нестираемая надпись

Исходный код программы находится в папке **Pixel**.

«Пипетка»

Мы можем представить канву в виде двухмерного массива пикселей, каждый из которых характеризуется *координатами* (индексами массива) и *цветом*. Чтобы узнать, какой цвет имеет тот или иной пиксель канвы, достаточно обратиться к свойству *Графического окна* **GetPixel(x,y)**.

Начните новый проект и сохраните его в папке **Пипетка**.

Пипеткой в графических редакторах называют инструмент, который помогает узнать цвет пикселя под курсором. В этом режиме работы он как бы превращается в пипетку (Рис. 17.4)

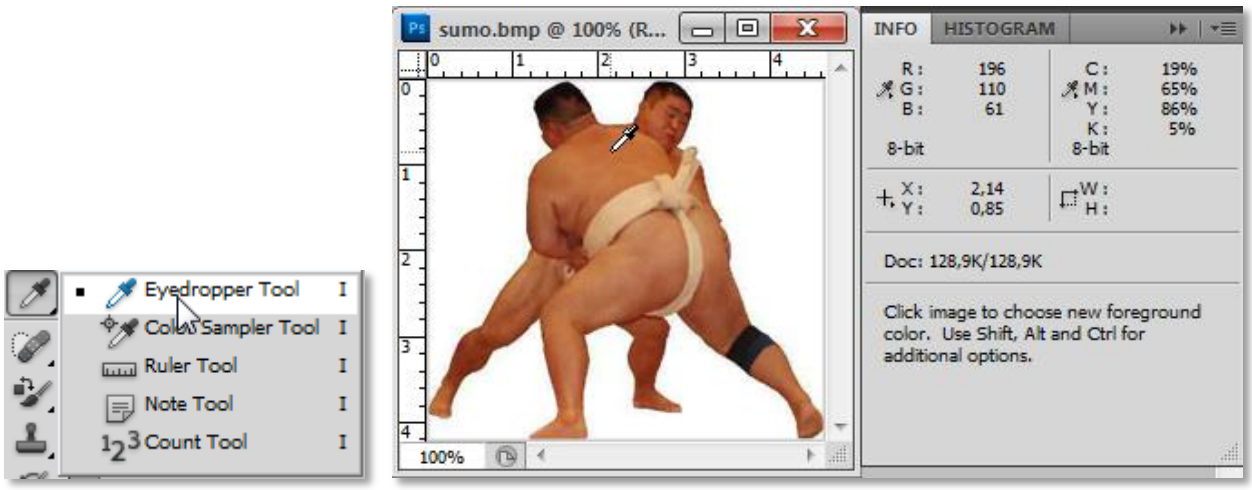


Рис. 17.4. Пипетка в *Фотошопе*

```
GraphicsWindow.Title="Цвет пикселя"

GraphicsWindow.Width= 320
GraphicsWindow.Height=240
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"
```

Сначала мы окрашиваем все пиксели канвы в разные цвета, а затем перемещаем мышку по канве и в строке заголовка читаем координаты курсора и цвет пикселя под курсором.

```
'Окрашиваем пиксели канвы в случайные цвета:
height=GraphicsWindow.Height
width= GraphicsWindow.Width
for y=0 to height-1
  for x=0 to width-1
    'выбираем случайный цвет для пикселя:
    clr= GraphicsWindow.GetRandomColor()
    'и окрашиваем его:
    GraphicsWindow.SetPixel(x,y,clr)
  EndFor
EndFor
```

Координаты курсора (точнее – его «горячей точки», которая обычно находится в верхнем левом углу) можно легко узнать по значению свойств **MouseX** и **MouseY** *Графического окна*.

Вся интрига этой программульки заключается вот в этих скупых строках:

```
'определяем цвет пикселя под мышкой:
While "True"
  x = GraphicsWindow.MouseX
  y = GraphicsWindow.MouseY
  clr=GraphicsWindow.GetPixel(x,y)
  GraphicsWindow.Title="Цвет пикселя (" + x + ","  + y + ") =
" + clr
EndWhile
```

Цвет пикселя под *горячей точкой* курсора (её координаты можно прочитать в заголовке окна) в 16-ричном виде указан там же (Рис. 17.5).

К сожалению, пиксели такие маленькие, что трудно определить их цвет на глаз, поэтому добавим в программу небольшое «окошко», которое будем закрашивать цветом текущего пикселя (Рис. 17.6).

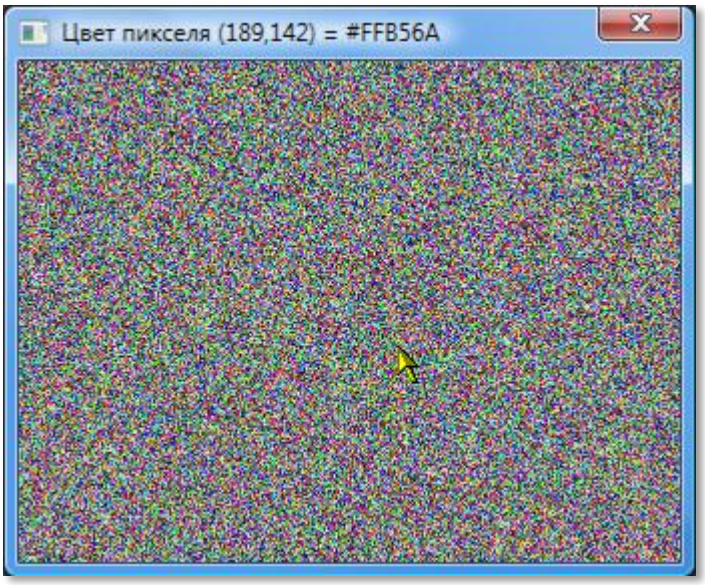


Рис. 17.5. Цветная пипетка в действии!

```
While "True"
  x = GraphicsWindow.MouseX
  y = GraphicsWindow.MouseY
  clr=GraphicsWindow.GetPixel(x,y)
  'окошко:
  GraphicsWindow.BrushColor=clr
  GraphicsWindow.FillRectangle(GraphicsWindow.Width-32-6, 20,
32,32)
  GraphicsWindow.Title="Цвет пикселя (" + x + ","  + y + ") =
" + clr
EndWhile
```

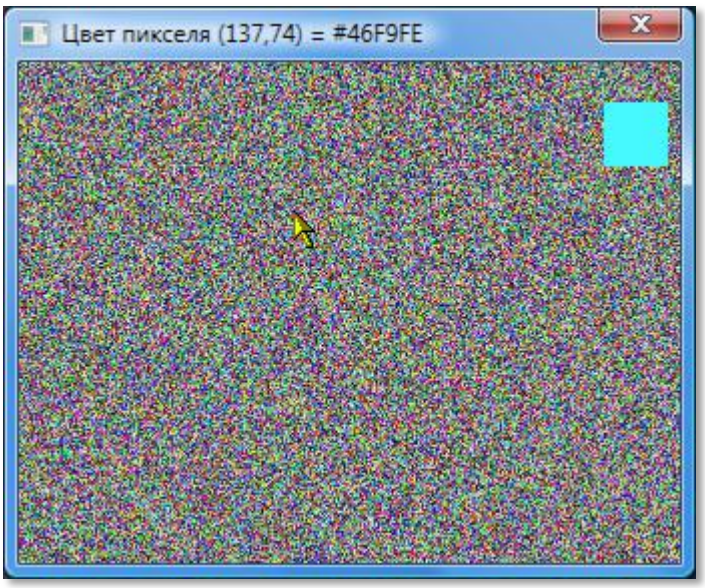


Рис. 17.6. А вот теперь каждый пиксель виден как на ладони!

Если вам понравится какой-нибудь цвет, запомните его код и применяйте в программах. Например, так:

GraphicsWindow.BrushColor= «#46f9fe»

Исходный код программы находится в папке **Пипетка**.

Многоточие

Раз уж компьютерные точки всё равно имеют размер, то мы сделаем их крупнее, чтобы лучше разглядеть. В этом деле нам потребуется не микроскоп и даже не увеличительное стекло, а новая программа **Многоточие**, которая умеет рисовать огромные точки, так что после её работы экран будет усыпан ими, как новогодний пол – конфетти (Рис. 17.7).

И в этом случае цвет и место точек задаются случайным образом, но теперь картина получается более «художественная».

Начало программы мы просто скопируем из предыдущих проектов:

```
GraphicsWindow.Title=" Многоточие"

GraphicsWindow.Width= 640
GraphicsWindow.Height=480
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"

height=GraphicsWindow.Height
width= GraphicsWindow.Width
```

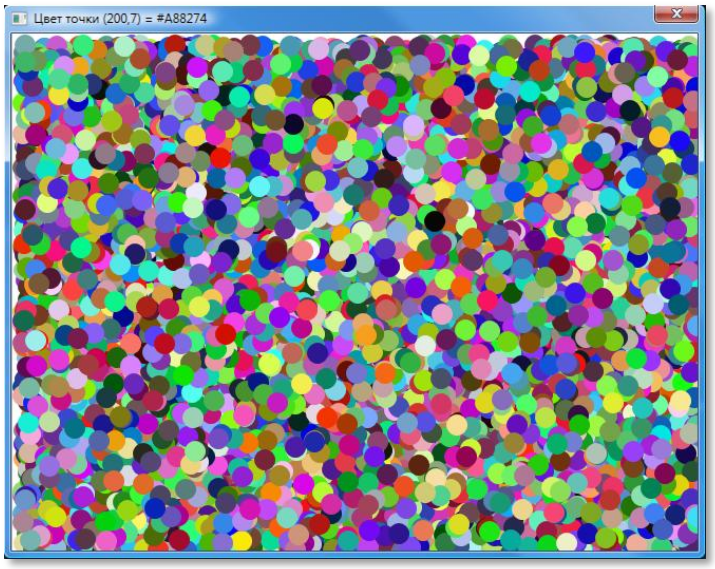


Рис. 17.7. Супер-пиксели!

Но теперь вместо отдельных пикселей мы рисуем «точки»:

```
' радиус "точки":
radius=10
for i= 1 to 10000
  'выбираем случайные координаты  "точки":
  x = Math.GetRandomNumber(width-radius)
  y = Math.GetRandomNumber(height-radius)
  'выбираем случайный цвет для  "точки":
  clr= GraphicsWindow.GetRandomColor()
  GraphicsWindow.BrushColor=clr
  GraphicsWindow.Title=" Цвет точки (" + x + ","  + y + ") = "
+ clr
  'рисуем цветную "точку":
  GraphicsWindow.FillEllipse(x, y, 2*radius,2*radius)
EndFor
```

На самом деле мы, конечно, схитрили и нарисовали не точки, а окрашенные *эллипсы* (а точнее - кружочки). Для этого нам понабился метод **FillEllipse(x, y, width, height).** Первая пара параметров – это *координаты* верхнего левого угла описанного прямоугольника, вторая пара – *ширина* и высота *эллипса* (Рис. 17.8). Цвет эллипса определяется свойством *BrushColor*.

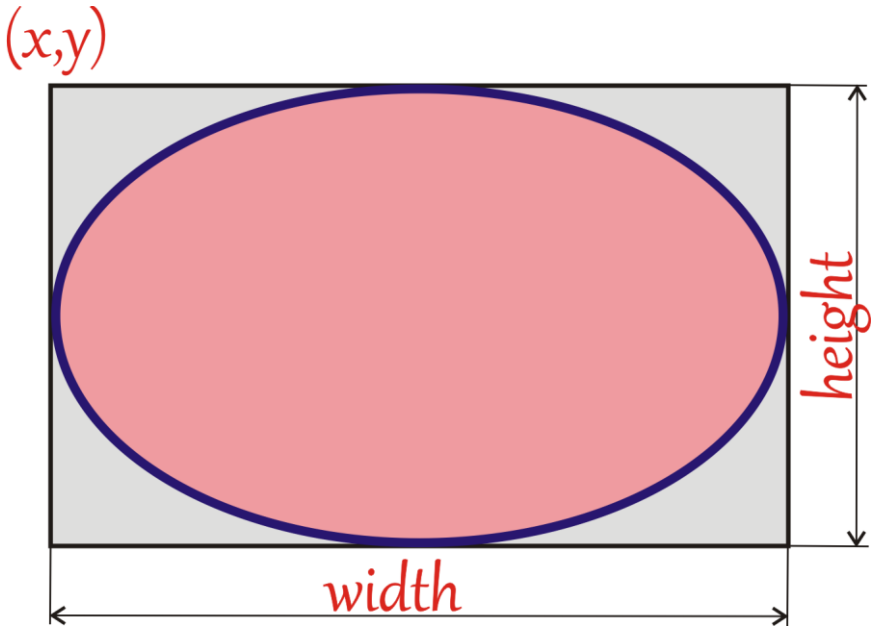


Рис. 17.8. Закрашенный эллипс

Для рисования *незакрашенного* эллипса нам пригодится метод **DrawEllipse(x, y, width, height)**. Так как он внутри пустой, то есть имеет цвет фона, то необходимо выделить контур эллипса, иначе мы его вообще не увидим. Цвет контура определяется значением свойства *PenColor*. Немного изменим код:

```
  'рисуем цветную "точку":
  'GraphicsWindow.FillEllipse(x, y, 2*radius,2*radius)
  GraphicsWindow.PenColor=clr
  GraphicsWindow.DrawEllipse(x, y, 2*radius,2*radius)
```

И точки-кружочки превращаются в элегантные окружности (Рис. 17.9).

Поскольку точки одного и того же размера вгоняют в тоску, то давайте усеем канву точками всевозможного калибра (Рис. 17.10). Сделать это проще простого. Достаточно радиус кружков задавать случайным образом с помощью метода *GetRandomNumber*:

```
height=GraphicsWindow.Height
width= GraphicsWindow.Width
for i= 1 to 10000
  radius= Math.GetRandomNumber(10)+4
  'выбираем случайные координаты  "точки":
  x = Math.GetRandomNumber(width-radius)
  y = Math.GetRandomNumber(height-radius)
```

```
  'выбираем случайный цвет для  "точки":
  clr= GraphicsWindow.GetRandomColor()
  GraphicsWindow.BrushColor=clr
  GraphicsWindow.Title=" Цвет точки (" + x + ","  + y + ") = " +
clr
  'рисуем цветную "точку":
  GraphicsWindow.FillEllipse(x, y, 2*radius,2*radius)
EndFor
```

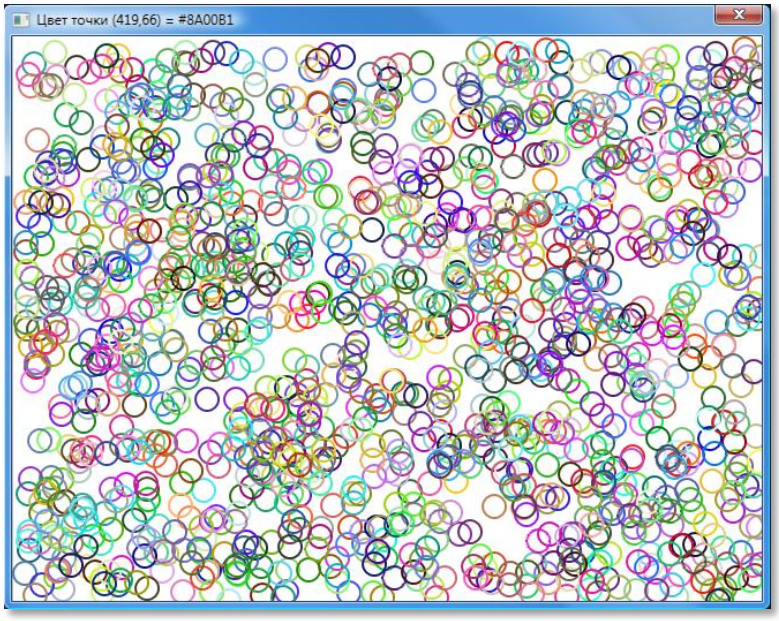


Рис. 17.9. Цветные колечки

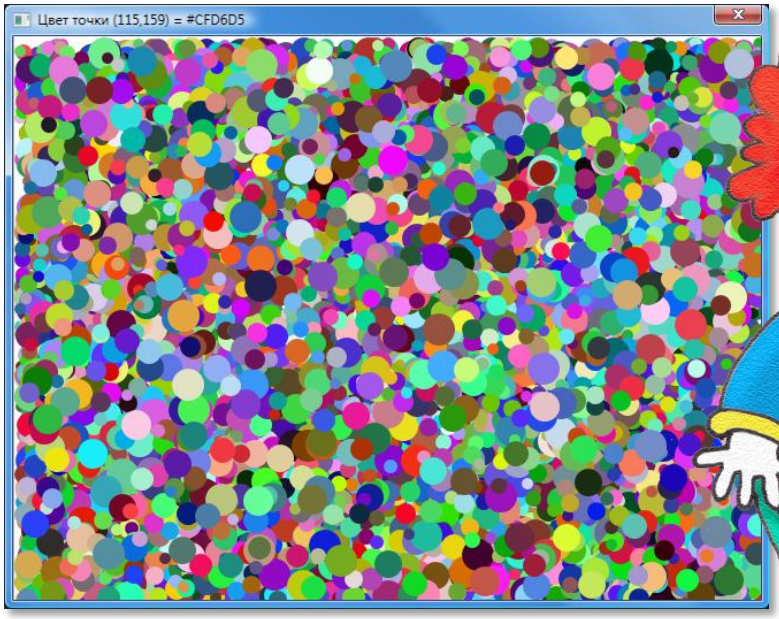


Рис. 17.10. Пёстренько, но со вкусом!

Исходный код программы находится в папке **Многоточие**.

Фильтруем снимки

Вам, без сомнения, не раз попадались на глаза необычные снимки, как бы составленные из цветных квадратиков или точек. Может быть, вы и сами делали такие картинки в *Фотошопе*, который имеет немало *фильтров* для придания фотографиям нового облика. Например, фильтр *Pointilize* (Рис. 17.11), который может увеличивать пиксели исходного изображения в несколько раз. В итоге получается «художественное» изображение, похожее на картины, написанные в технике *пуантилизма* (цветными точками) (Рис. 17.12).

Мы не будем замахиваться на достижения искусства или даже всеми нами любимого *Фотошопа,* а напишем простенький, но свой фильтр, который превратит любую фотографию в «мозаику».

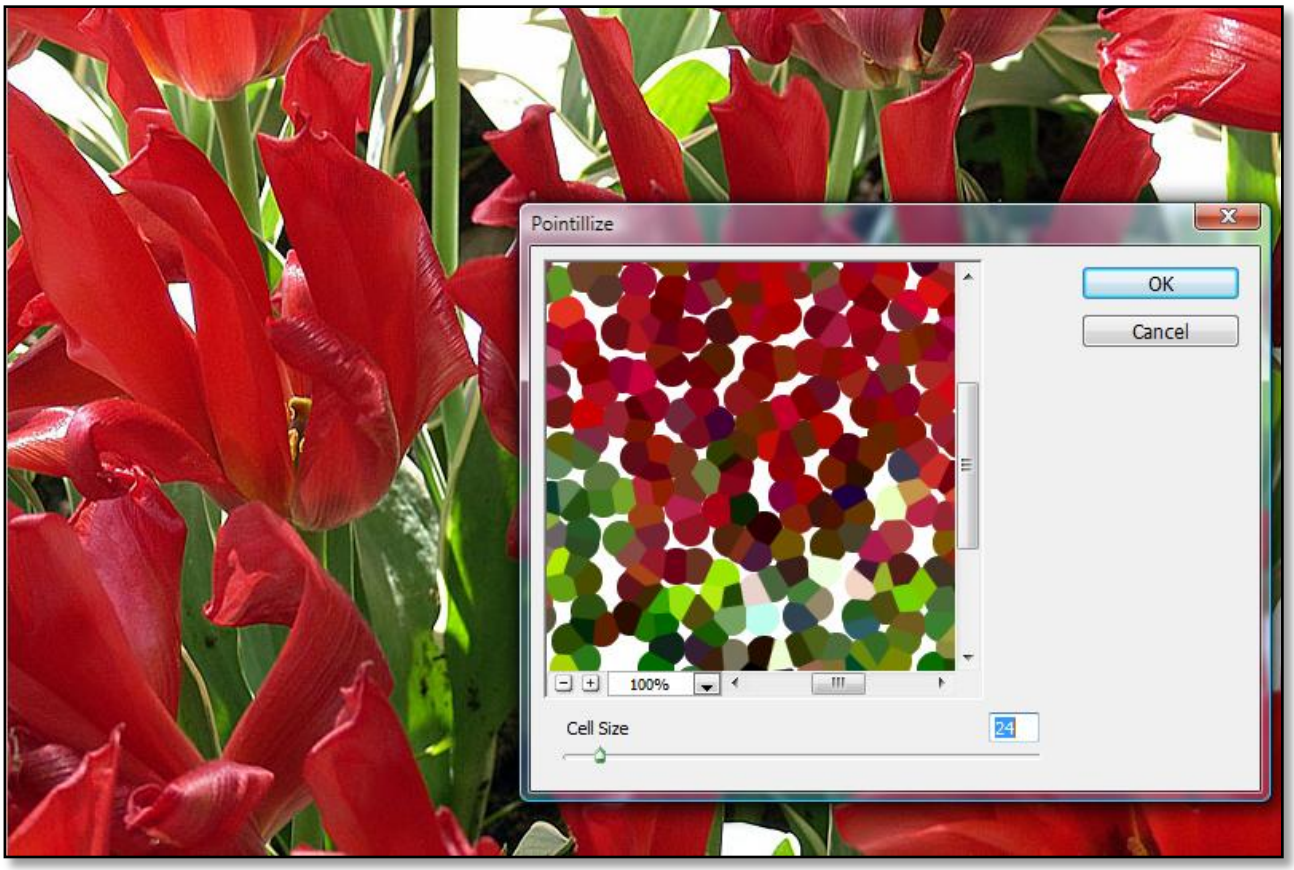


Рис. 17.11. Фильтр *Pointilize* в графическом редакторе *Фотошоп*

Для экспериментов всё-таки лучше взять не *любую* фотографию, а красочную и без мелких деталей, которые неизбежно исчезнут при *пикселизации*. Если вам приходилось видеть картины, вышитые крестиком, то смело берите пример с них.

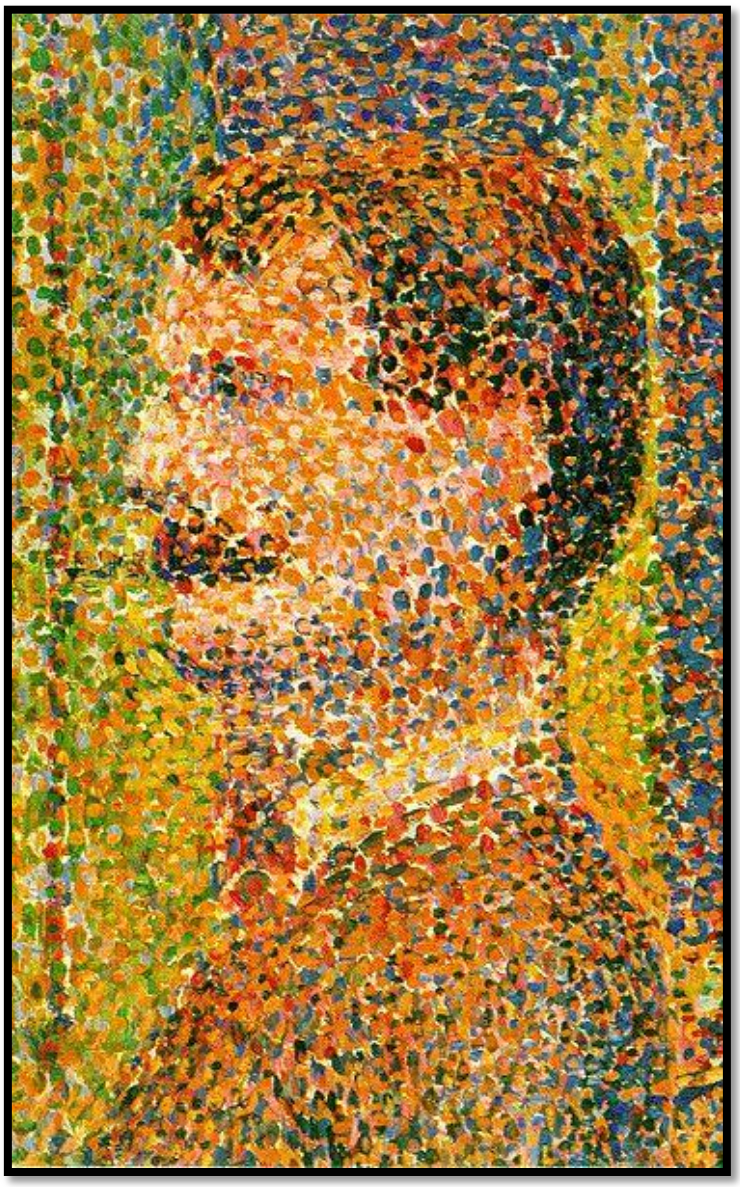


Рис. 17.12. Фрагмент картины Ж. Сёра

За основу проекта мы возьмём исходный код *Пипетки*, который сохраним в папке **Фильтр**.

В начало программы добавим единственную переменную *r*, которая у нас будет отвечать за величину точек отфильтрованного изображения. Размеры окна подгоните под выбранную вами фотографию. Поместите её в папку с программой и при запуске про-

граммы впечатайте ее в канву *графического окна*. В качестве примера я взял фотографию с красными тюльпанами, которая в *графическом окне* смотрится совсем неплохо (Рис. 17.13).

```
'ПРОГРАММА ДЛЯ ПИКСЕЛИЗАЦИИ ИЗОБРАЖЕНИЯ

'var
'радиус точки:
r=5

GraphicsWindow.Title="Фильтр"

GraphicsWindow.Width= 800-10
GraphicsWindow.Height=533-10
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"

height=GraphicsWindow.Height
width= GraphicsWindow.Width

background = ImageList.LoadImage(Program.Directory +
"/тюльпаны.jpg")
GraphicsWindow.DrawImage(background, 0, 0)

'применяем фильтр:
'Filter()
Filter2()

'определяем цвет пикселя под мышкой:
While "True"
  x = GraphicsWindow.MouseX
  y = GraphicsWindow.MouseY
  clr=GraphicsWindow.GetPixel(x,y)
  'окошко:
  GraphicsWindow.BrushColor=clr
  GraphicsWindow.FillRectangle(GraphicsWindow.Width-32-6, 20,
32,32)
  GraphicsWindow.Title="Цвет пикселя (" + x + ","  + y + ") =
" + clr
EndWhile
```

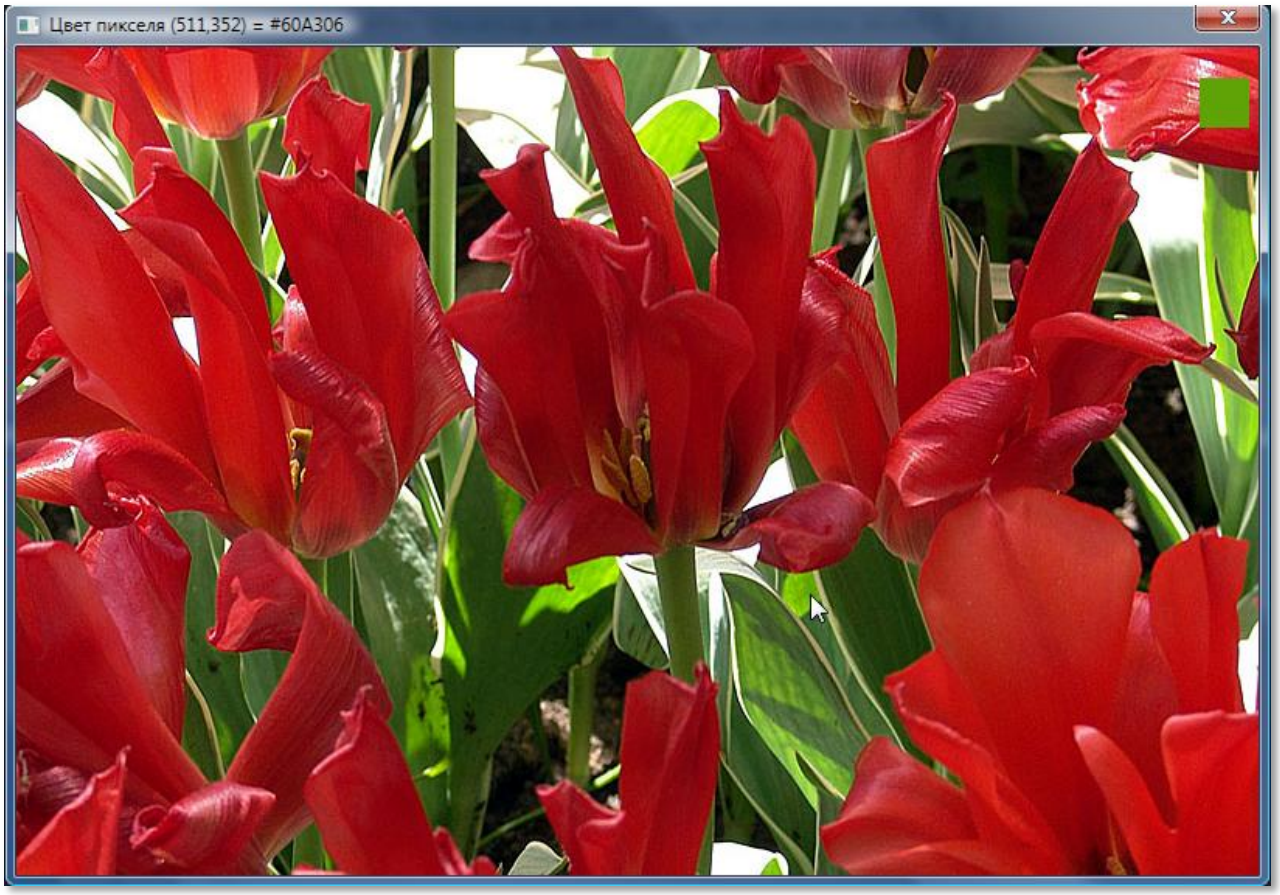


Рис. 17.13. Исходное изображение

Мы напишем *два* фильтра, которые отличаются друг от друга только тем, что в первом точки *квадратные*, а во втором - *круглые*. С точки зрения геометрии, разница небольшая, но в искусстве, как известно мелочей нет, поэтому пробуйте разные варианты, пока не добьётесь совершенства.

Первый фильтр действует очень просто. Мы разбиваем все изображение на квадратики со стороной *l* пикселей и с помощью двух циклов *For* сканируем изображение. Для каждого квадратика находим цвет пикселя в его центре, а затем всё изображение внутри квадратика заливаем этим цветом (Рис. 17.14).

```
'Процедура фильтрации изображения
Sub Filter
  l= 2*r
  For j= 1 To height/l
    For i= 1 To width/l
      'определяем цвет пикселя в центре точки:
      xc = (i-1)*l + r
      yc = (j-1)*l + r
```

```
      clr=GraphicsWindow.GetPixel(xc,yc)
      GraphicsWindow.BrushColor=clr
      'чертим квадрат:
      GraphicsWindow.FillRectangle(xc-r, yc-r, 1, 1)
    EndFor
  EndFor
EndSub
```

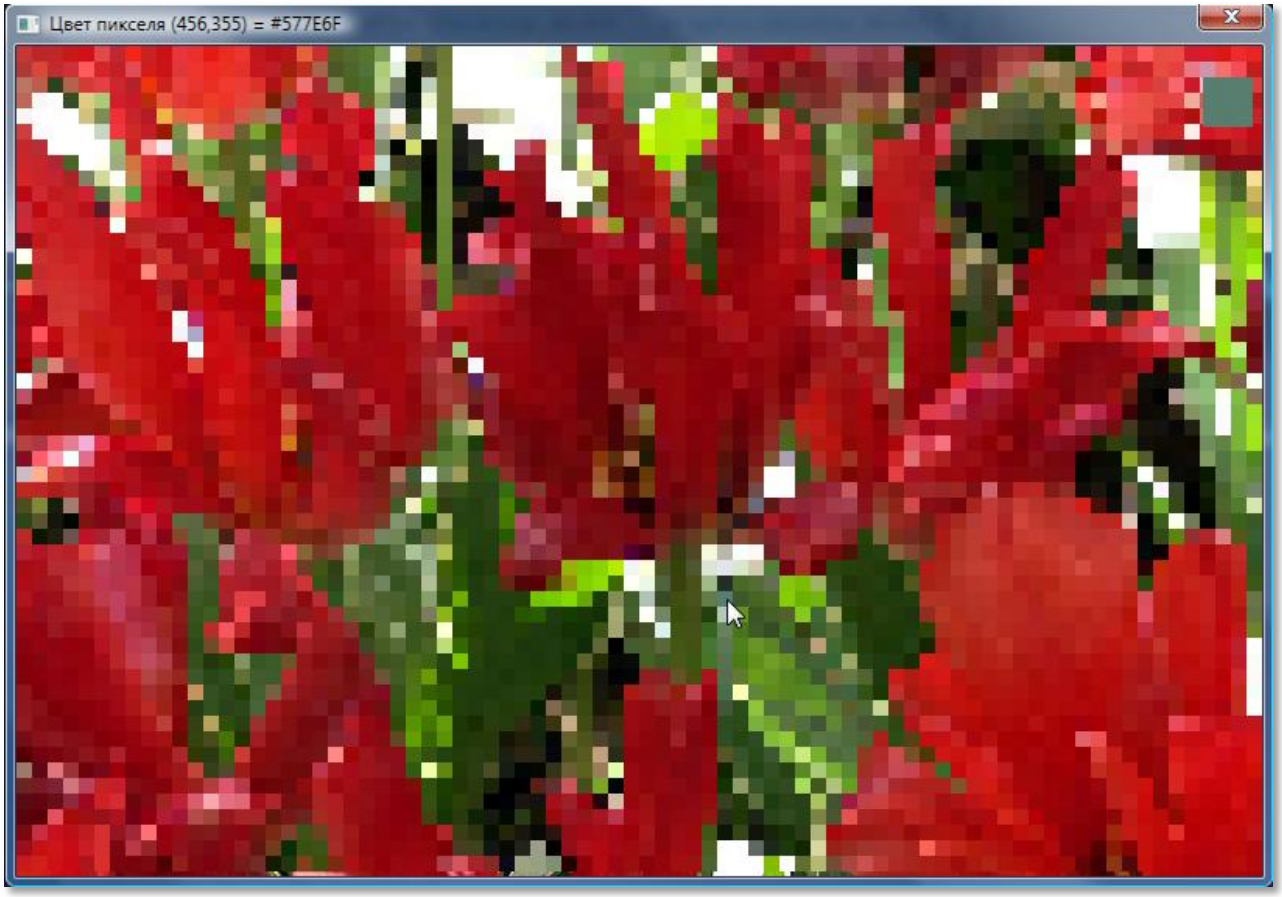


Рис. 17.14. Отфильтрованные тюльпаны

В результате изображение становится более грубым, как бы состоящим из огромных пикселей. Подобный эффект мы можем наблюдать при большом увеличении фотографий. Правда, в этом случае увеличиваются и размеры картинки, поэтому все пиксели исходного изображения остаются в целости и сохранности, наш же фильтр изменяет цвет большинства пикселей.

Второй фильтр более «изощрённый». Сначала мы запоминаем цвет пикселя в центре квадрата, затем закрашиваем его **чёрным** цветом.

Вы можете вообще не закрашивать квадрат, или закрашивать его другими цветом. Иногда таким простым способом удаётся получить хорошие результаты!

А уж потом, «по-чёрному» рисуем круг заданного цвета:

```
Sub Filter2
  l= 2*r
  For j= 1 To height/l
    For i= 1 To width/l
      'определяем цвет пикселя в центре точки:
      xc = (i-1)*l + r
      yc = (j-1)*l + r
      clr=GraphicsWindow.GetPixel(xc,yc)
      'рисуем квадрат:
      GraphicsWindow.BrushColor= "Black"
      'GraphicsWindow.BrushColor= "White"
      GraphicsWindow.FillRectangle(xc-r, yc-r, l, l)
      'рисуем круг:
      GraphicsWindow.BrushColor=clr
      GraphicsWindow.FillEllipse(xc-r, yc-r, l, l)
    EndFor
  EndFor
EndSub
```

Если каждый цветной кружок вышить крестиком (а лучше не полениться и вышить *двойным* крестиком), то получится прекрасная вышитая картина (Рис. 17.15). Особенно если к выбору исходного изображения подойти с полной ответственностью и бездной вкуса!

Исходный код программы находится в папке **Фильтр**.

Рис. 17.15. Фильтруем тюльпаны через круглые «дырочки»

1. В программе *Фильтр* мы устанавливали цвет точки по цвету пикселя в её центре. С увеличением размеров точки отфильтрованная картинка всё больше отдаляется от оригинала, поскольку мы не учитываем цвет других пикселей этой точки. Подумайте, как *усреднить* цвет точки в фильтре.

2. Добавьте к программе процедуру рисования *сетки* после применения *квадратного* фильтра (Рис. 17.16):

```
'применяем фильтр:
Filter()
drawGrid()

'Процедура рисования сетки
Sub drawGrid
  l= 2*r
  GraphicsWindow.PenColor="Black"
```

```
  GraphicsWindow.PenWidth=1
  'горизонтали:
  For j= 0 To height/l
    GraphicsWindow.DrawLine(0, j*l, width, j*l)
  EndFor
  'вертикали:
  For i= 0 To width/l
    GraphicsWindow.DrawLine(i*l, 0, i*l, height)
  EndFor
EndSub
```

Рис. 17.16. С сеткой мозаика выглядит не в пример лучше!

Урок 18. Полярная система координат

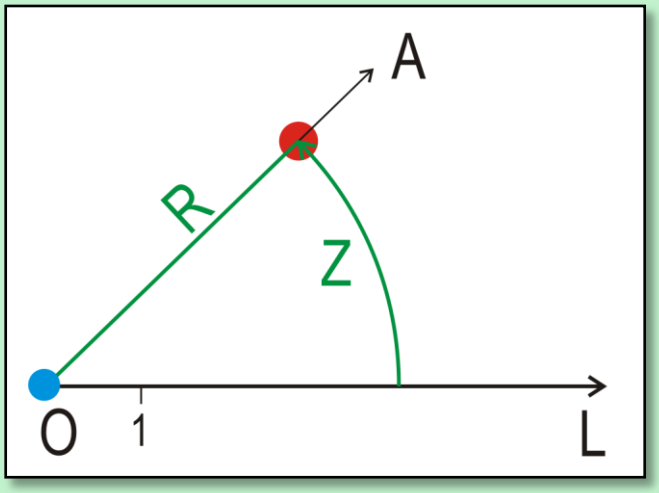
Трутся об ось медведи,-
Вертится Земля…

Песенка из комедии *Кавказская пленница*

Кроме прямоугольной системы координат, которая известна всем и каждому, существуют и другие. Если мы хотим поставить точку на плоскости, нам обязательно понадобятся *два* числа. В декартовой системе координат это абсцисса и ордината, а в **полярной**, которая нам понадобится на этом уроке, - *полярный радиус R* и *полярный угол Z*.

Полярный радиус называется также *радиус-вектором*. Длина радиус-вектора называется *модулем*.

Начало полярных координат находится, как легко догадаться, в *полюсе*. Из него мы можем провести в любом направлении луч, образующий с полярной осью *OL* угол **Z**, который отсчитывается *против* часовой стрелки. Если луч проходит через заданную точку, то её положение в полярной системе координат определяется значениями полярного угла **Z** и полярного радиуса **R**, который равен расстоянию от точки до полюса. Очень хорошо это видно на картинке (Рис. 18.1).



O – полюс
OL – полярная ось
OA – луч, проходящий через заданную точку на плоскости

R – полярный радиус точки
Z – её полярный угол

Рис. 18.1. Полярная система координат

Поскольку координаты пикселей на экране удобнее задавать в прямоугольной системе координат, то давайте выведем формулы, по которым можно пересчитать координаты точки в *полярной* системе в координаты в *прямоугольной* системе. Для этого совместим начало прямоугольной системы координат с полюсом, а ось абсцисс – с полярной осью (Рис. 18.2).

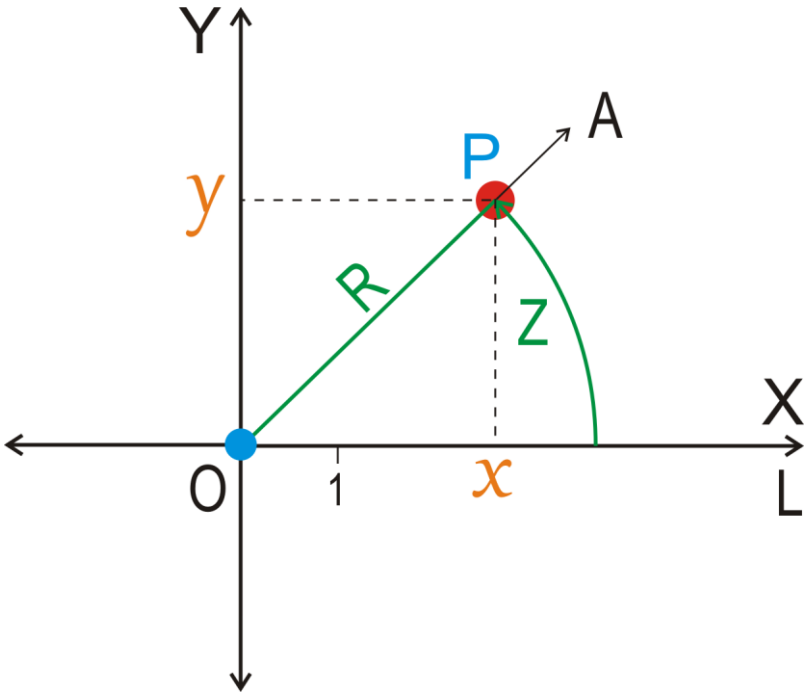


Рис. 18.2. Координаты точки в двух системах координат

Опустим перпендикуляры из точки *P* на ось абсцисс и ординат. Точки пересечения перпендикуляров обозначим буквами *x* и *y* и рассмотрим прямоугольный треугольник *OPx*, из которого легко найдём координаты точки *P*:

x = R * cos(Z)	(1)
y = R * sin(Z)	(2)

Возникает вопрос: если от полярных координат так просто перейти к прямоугольным, то для чего вообще их использовать? – Дело в том, что в полярных координатах некоторые функции задавать гораздо проще, чем в прямоугольных.

Например, *окружности* можно описать такими формулами:

R = 1 – для единичной окружности или
R = r – для окружности радиуса *r*

Те же самые окружности в прямоугольных координатах имеют совсем непростой вид:

$x^2 + y^2 - 2x = 0$ и
$x^2 + y^2 - 2rx = 0$

И многие другие кривые - спирали, улитки, розы, эллипсы, кардиоиды – удобнее задавать именно в полярных координатах. Чтобы убедиться в этом, давайте построим графики нескольких знаменитых кривых, описанных в полярных координатах.

Начните новый проект **Polar** и сохраните его в новой папке.

Для построения графиков (Рис. 18.3) нам потребуются три *переменные*. Назначение двух вполне очевидно, а флаг *res* будет указывать, находится ли очередная точка в пределах канвы или нет:

```
'ПРОГРАММА ДЛЯ ПОСТРОЕНИЯ ГРАФИКОВ
'В ПОЛЯРНЫХ КООРДИНАТАХ

'var
res="False"   'флаг ="False", если очередная точка выходит за
границы формы
R=0           'полярный радиус
Z=0           'полярный угол

GraphicsWindow.Title=" Графики в полярных координатах"

GraphicsWindow.Width= 320
GraphicsWindow.Height=320
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"

height=GraphicsWindow.Height
width=GraphicsWindow.Width
```

```
'координаты центра окна:
CX= width/2
CY= height/2

'цвет линий:
GraphicsWindow.PenColor= "Black"
GraphicsWindow.PenWidth=1
```

Чтобы построить график, мы изменяем полярный угол *Z* от нуля до 360 градусов (в радианах это – *0 ... 2*Math.Pi*). При дальнейшем увеличении полярного угла новые точки будут просто накладываться на уже существующие, поэтому такое ограничение угла вполне уместно.

Важно задать небольшой *шаг* (*Step 0.01*) изменения угла, чтобы точки графика создавали *цельную* кривую. В некоторых случаях не помогает и мелкий шаг, поэтому мы будем ставить не отдельные точки, а проводить *линии* из каждой последующей точки в предыдущую. Так мы застрахуем себя от разрывов в графике кривой. У самой первой точки нет предыдущей, поэтому мы должны учесть это исключение.

Построение всех кривых оформлено в виде *отдельных* процедур, что, конечно, гораздо удобнее, чем записывать соответствующий код одним куском. В данном примере мы построим кривую, заданную в процедуре *calc3*. Если мы захотим построить другой график, то просто заменим вызов процедуры *calc3*, например, вызовом *calc1* или *calc5*.

Закомментированный цикл *For* нужен только для процедуры *calc5 Звёздочка*.

```
'Строим график
'For i=0 to 7
for Z =0 to 2*Math.Pi+0.01 Step 0.01
  calc3()
  If (res="False") Then
    Goto nextZ
  EndIf

  'запоминаем координаты первой точки:
```

```
  If (Z=0) Then
    x1=x
    y1=y
  EndIf

  'ставим "точку"
  GraphicsWindow.DrawLine(x,y,x1,y1)
  nextZ:
  'TextWindow.WriteLine("x= " + x + "  y= " + y)
  x1=x
  y1=y
EndFor
'endFor
```

Все процедуры для построения кривых устанавливают флаг *res* в *"False"*, если очередную точку кривой *не нужно* рисовать, поскольку она выходит за границы канвы. Переменные *x* и *y* хранят координаты очередной точки кривой, пересчитанные из полярных координат в прямоугольные (оконные) по тем формулам (1) и (2), что мы вывели в начале урока.

Следующие процедуры просто вычисляют прямоугольные координаты точек по полярным формулам соответствующих кривых. Важно помнить, что некоторые формулы содержат параметры, при изменении которых график приобретает другой вид. Это значит, что с помощью этих формул вы можете начертить гораздо больше красивых кривых, чем показано в книге.

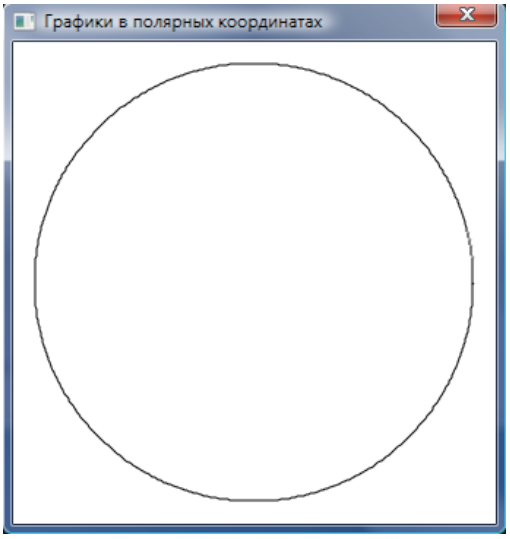
Например, по формуле R= sin(2*Z) (Рис 18.3б) будет вычерчен лист клевера, а по формуле R= sin(12*Z) (Рис 18.3в) – ромашка. В данном случае можно рассматривать формулу R= sin(k*Z) с параметром *k*, который задает число лепестков кривой.

Обязательно попробуйте поиграть параметрами или добавить в формулу новые члены, чтобы найти новые кривые!

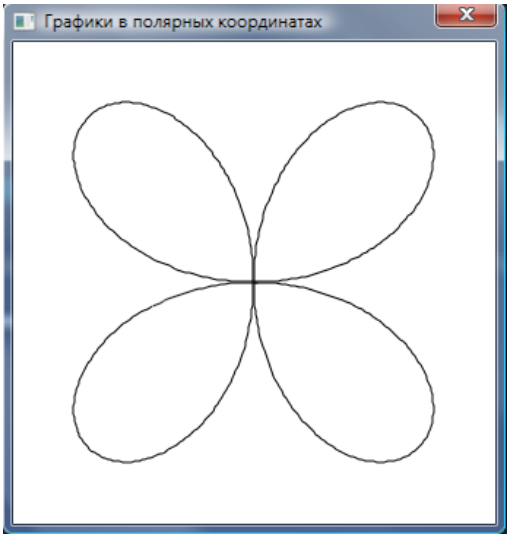
```
'Окружность
Sub calc1
  'радиус окружности:
  R= 150
  res="True"
```

```
  x= CX + Math.Cos(z)*R
  If (x<0) Or (x> width) Then
    res="False"
    Goto exit1
  EndIf
  y= CY + Math.Sin(z)*R
  If (y<0) Or (y> height) Then
    res="False"
    Goto exit1
  EndIf
  exit1:
EndSub

'Клевер, ромашки
Sub calc2
  k=12
  res="True"
  R= Math.Sin(k*Z)*160
  x= CX + Math.Cos(z)*R
  If (x<0) Or (x> width) Then
    res="False"
    Goto exit2
  EndIf
  y= CY + Math.Sin(z)*R
  If (y<0) Or (y> height) Then
    res="False"
    Goto exit2
  EndIf
  exit2:
EndSub

'Улитка Паскаля
Sub calc3
  k=2
  l=2
  res="True"
  R= (1+l*Math.Cos(k*Z))* 50'80'100
  x= CX + Math.Cos(z)*R '- width/4
  If (x<0) Or (x> width) Then
    res="False"
    Goto exit3
  EndIf
  y= CY + Math.Sin(z)*R
  If (y<0) Or (y> height) Then
```

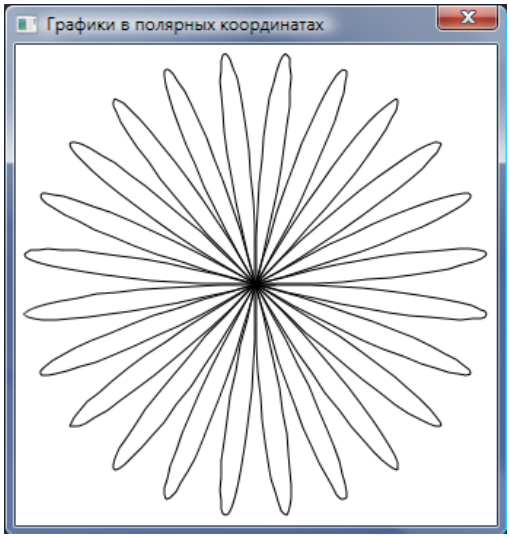
```
    res="False"
    Goto exit3
  EndIf
  exit3:
EndSub

'Спираль
Sub calc4
  k=5
  res="True"
  R= Z/k* 100
  x= CX + Math.Cos(z)*R
  If (x<0) Or (x> width) Then
    res="False"
    Goto exit4
  EndIf
  y= CY + Math.Sin(z)*R
  If (y<0) Or (y> height) Then
    res="False"
    Goto exit4
  EndIf
  exit4:
EndSub

'Звёздочка
Sub calc5
  k=5
  res="True"
  F= Z + i*Math.Pi*2
  R= 60/(2+ Math.Cos(F*k + ((Math.Floor(0.1*F)/5 -
Math.Floor(Math.Floor(0.1*F)/5)))))*2.5
  x= CX + Math.Cos(F)*R
  If (x<0) Or (x> width) Then
    res="False"
    Goto exit5
  EndIf
  y= CY + Math.Sin(F)*R
  If (y<0) Or (y> height) Then
    res="False"
    Goto exit5
  EndIf
  exit5:
EndSub
```



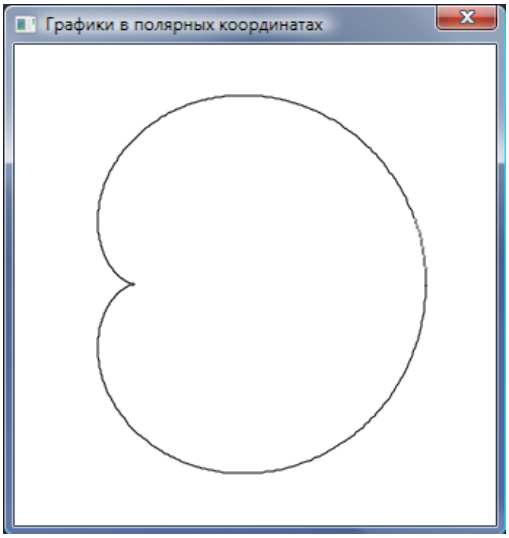
а) R = 1 - окружность



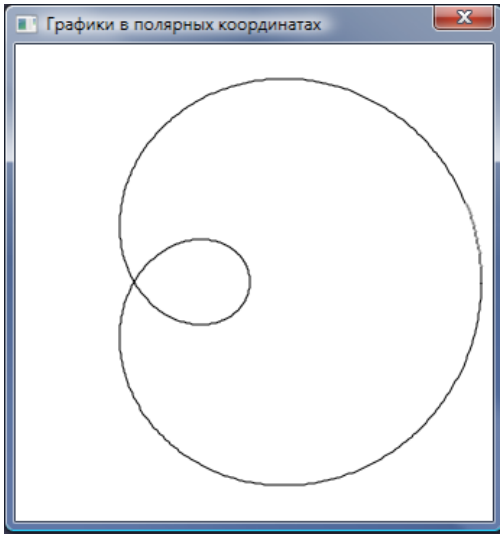
б) R= sin(2*Z) - клевер



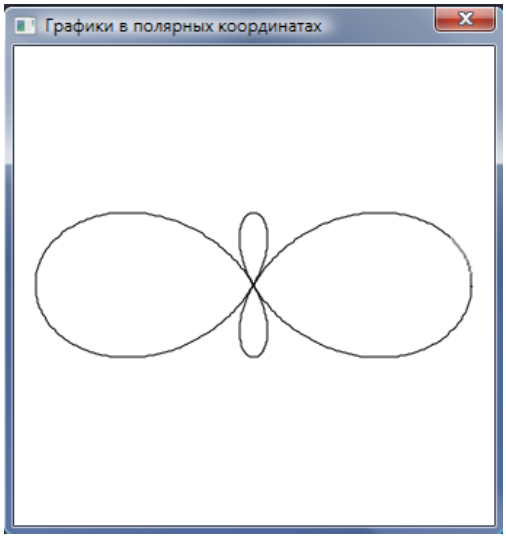
в) R= sin(12*Z) - ромашка



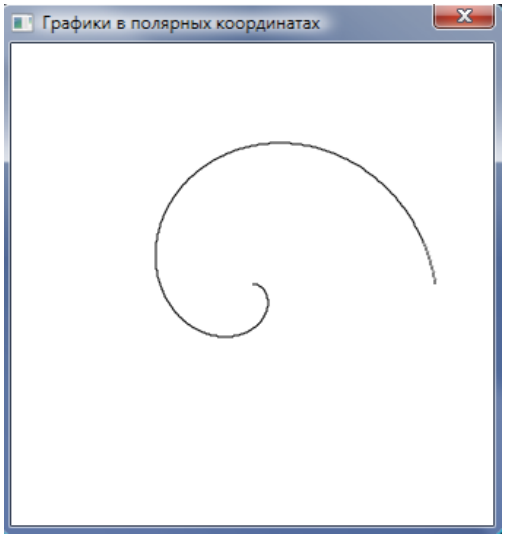
г) R= 1 + cos(Z) - кардиоида



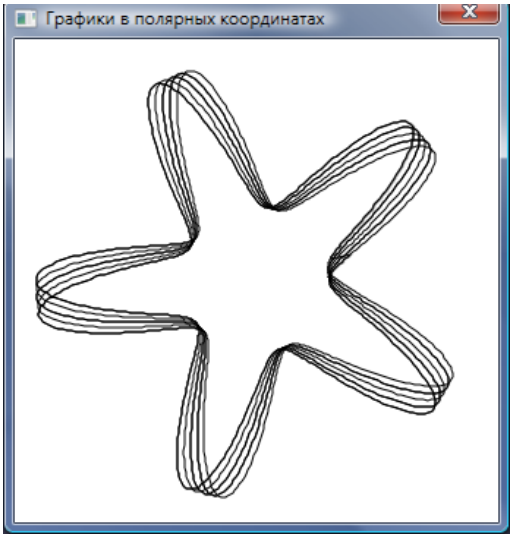
д) R= 1 + 2*cos(Z) – улитка Паскаля



е) R= 1 + 2*cos(2*Z) – петельное сцепление



ж) R= Z/k – спираль



з) Звездочка

Рис. 18.3. Графики кривых в полярных координатах

Исходный код программы находится в папке **Polar**.

1. Установите на форме *кнопки* для выбора нужной кривой.

Кнопки мы хорошенько изучим на уроке *Элементы управления*.

2. Поэкспериментируйте с различными значениями параметров, входящих в формулу. Так, например, можно получить ромашки с разным числом лепестков или улитки Паскаля разной формы.

3. Добавьте к программе возможность автоматического масштабирования графиков под размеры клиентской области окна приложения. Для этого перед началом построения графиков следует найти максимальные значения координат *x* и *y* и вычислить коэффициент масштабирования, на который затем умножить все значения текущих координат.

Урок 19. Занимательные игры с пикселями

Случайные точки, которыми мы «баловались» на уроке *Компьютерная графика*, создают хаотичный «узор», что не очень хорошо для уроков геометрии, поэтому теперь мы будем окрашивать пиксели по строгим математическим формулам. Они могут быть и довольно простыми, но узоры при этом давать расчудесные!

Синусоидные полоски

Начнём наш новый проект и сохраним его в папке **1D sine**. Рисунок здесь будут создавать горизонтальные строки одинаково окрашенных пикселей (проще говоря, линии, которые мы рисуем отдельными точками). Цвет пикселей изменяется по высоте клиентской области окна сверху вниз по совсем простой формуле:

```
clr = 255*(1+ Math.Sin(y/wl))/2
```
(1)

Поскольку в формуле присутствует синус угла, то и проект называется *Синусные полоски*. Частота горизонтальных волн зависит от длины волны *wl*, так что вы легко получите разные картинки, если измените значение этого параметра. Нам осталось узнать о назначении множителя 255 в формуле (1). Дело в том, что выражение в скобках изменяется в диапазоне 0..2, а делённое на два – в диапазоне 0..1. Цветные составляющие пикселя должны иметь значение от 0 до 255, откуда и вытекает необходимость введения в формулу этого множителя. Цвет очередного пикселя печатается в заголовке окна. Если вам эта информация не нужна, закомментируйте строчку

```
GraphicsWindow.Title="Цвет пикселя (" + x + ","  + y + ") = "
+ clr
```

Остальная часть кода не должна вызвать у вас никаких вопросов:

```
GraphicsWindow.Title="Синусоидные полоски"
GraphicsWindow.Width= 320
GraphicsWindow.Height=240
```

```
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"

height=GraphicsWindow.Height
width=GraphicsWindow.Width
'длина волны синусоиды:
wl= 10
for y=0 to height-1
  'цвет очередного пикселя:
  clr = 255*(1+ Math.Sin(y/wl))/2
  clr=GraphicsWindow.GetColorFromRGB(clr,clr, clr)
  GraphicsWindow.Title="Цвет пикселя (" + x + ","  + y + ") =
" + clr
  for x=0 to width-1
    'окрашиваем его:
    GraphicsWindow.SetPixel(x,y,clr)
  EndFor
EndFor
```

Картинка получилась славная, но **чёрно**-белая (Рис. 19.1).

Рис. 19.1. Синусоидные полоски

Но мы без труда окрасим волны в нужный цвет, слегка изменив параметры в методе *GetColorFromRGB* (Рис. 19.2 и 19.3).

Рис. 19.2. GetColorFromRGB(clr,0,0)

Рис. 19.3. GetColorFromRGB(0,0,clr)

Цветные волны ещё лучше!

Исходный код программы находится в папке **1D sine**.

Двойная волна

А теперь давайте пустим *две* волны – вертикальную и горизонтальную:

```
'длина горизонтальной волны:
wX= 20
'длина вертикальной волны:
wY= 20
```

Для этого в формулу для вычисления цвета пикселя добавим ещё один синус:

```
clr = 255*(1+Math.Sin(x/wX) * Math.Sin(y/wY))/2
```

Остальная часть программы ничуть не отличается от предыдущей:

```
for y=0 to height-1
```

```
  for x=0 to width-1
    'цвет очередного пикселя:
    clr = 255*(1+Math.Sin(x/wX) * Math.Sin(y/wY))/2
    clr=GraphicsWindow.GetColorFromRGB(0,clr,100)
    GraphicsWindow.Title=" Цвет пикселя (" + x + ","  + y + ")
= " + clr
    'окрашиваем его:
    GraphicsWindow.SetPixel(x,y,clr)
  EndFor
EndFor
```

Получилась интересная сетчатая структура (Рис. 19.4).

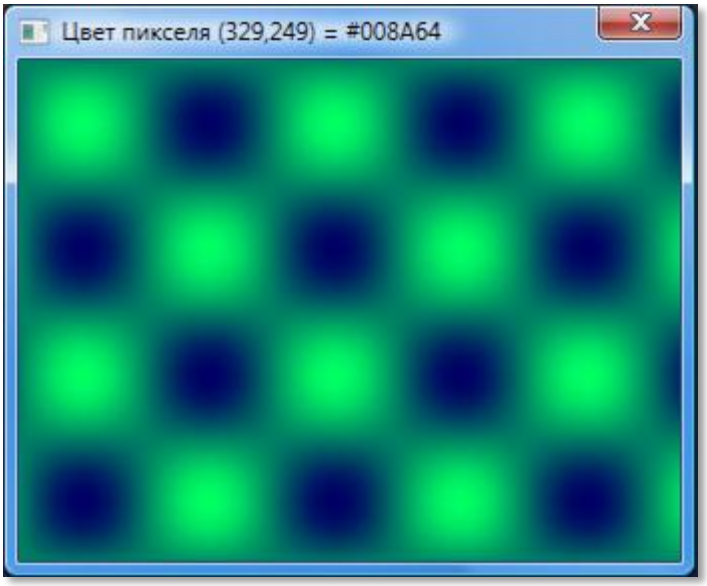


Рис. 19.4. Двойные волны

Исходный код программы находится в папке **2D sine**.

Лунки

В следующем проекте мы так изменим формулу для вычисления цвета пикселей, чтобы в ней участвовала *абсолютная величина* синусов:

```
'цвет очередного пикселя:
clr = 255*(1+ Math.Abs(Math.Sin(x/wX)) *
Math.Abs(Math.Sin(y/wY)))/2
```

Это приведет к тому, что все лунки приобретут *одинаковый* цвет, в отличие от проекта *Двойные волны*, где лунки окрашивались в *разные* цвета (Рис. 19.5).

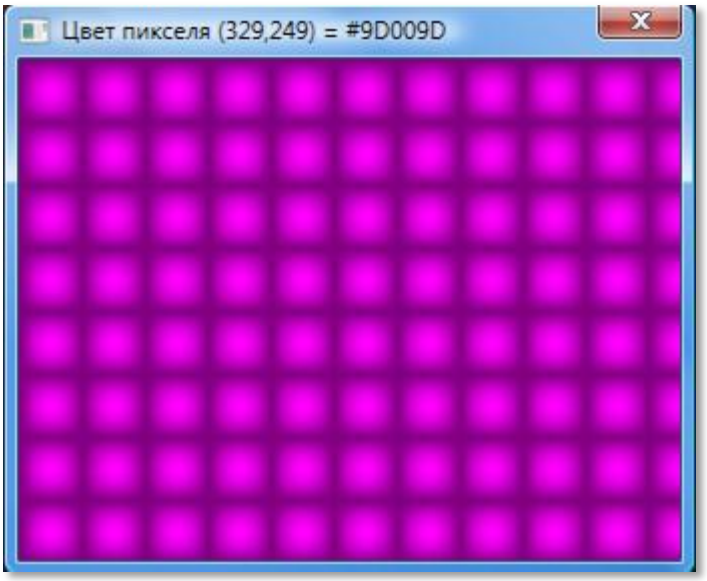


Рис. 19.5. Лунки

Исходный код программы находится в папке **Abs sine**.

Радиальные волны

Бросая в воду камешки, смотри на круги,
ими образуемые; иначе такое бросание
будет пустою забавою.

Козьма Прутков

Если вы бросали камни в воду, то, конечно, обратили внимание на то, что волны, которые разбегаются от места падения камня, имеют вовсе не форму прямых линий или лунок - они совершенно *круглые*. Чтобы загнуть волны в дугу, нам придётся перейти от прямоугольных координат к полярным, что мы и сделаем с помощью формулы:

```
rad = Math.SquareRoot((x-CX)*(x-CX) + (y-CY) * (y-CY)) / wl
```

Сама же формула для цвета пикселя останется без изменений:

```
clr = 255*(1+Math.Sin(rad))/2
```

Исходный код программы очень похож на наши прежние проекты:

```
GraphicsWindow.Title=" Радиальные волны"

GraphicsWindow.Width= 320
GraphicsWindow.Height=320
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"
height=GraphicsWindow.Height
width=GraphicsWindow.Width

'координаты центра волн:
CX= width/2
CY= height/2
'длина волны:
wl= 6
for y=0 to height-1
  for x=0 to width-1
    цвет очередного пикселя:
    rad = Math.SquareRoot((x-CX)*(x-CX) + (y-CY) * (y-CY)) /
wl
    clr = 255*(1+Math.Sin(rad))/2
    clr=GraphicsWindow.GetColorFromRGB(0, 0, clr)
    GraphicsWindow.Title=" Цвет пикселя (" + x + ","  + y + ")
= " + clr
    'окрашиваем его:
    GraphicsWindow.SetPixel(x,y,clr)
  EndFor
EndFor
```

Зато волны получились – как настоящие (Рис. 19.6)!

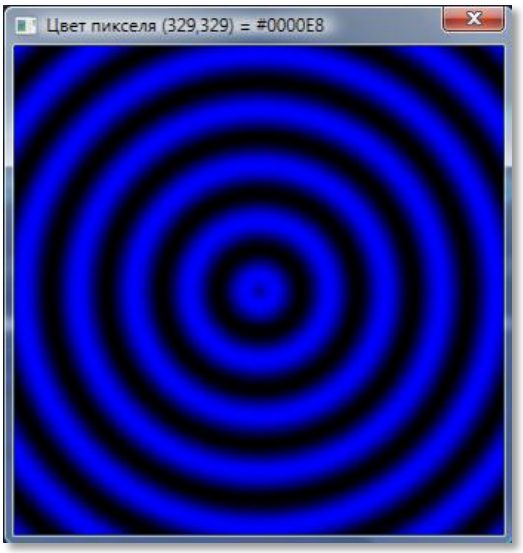


Рис. 19.6. «Полярные» волны!

Исходный код программы находится в папке **Polar sine**.

Ромбы

Применив для пикселестроения более хитроумную формулу, мы получим великолепный *ромбический узор*, от которого глаза трудно оторвать (Рис. 19.7).

```
GraphicsWindow.Title="Ромбы"

. . .

'длина волны:
wl= 60
for y=0 to height-1
  for x=0 to width-1
    'цвет очередного пикселя:
    clr = 255*2*(Math.Abs(math.Remainder(x, wl) -wl/2)  +
Math.Abs(math.Remainder(y, wl) -wl/2))/wl
    clr=GraphicsWindow.GetColorFromRGB(clr,0,clr)
    'крашиваем его:
    GraphicsWindow.SetPixel(x,y,clr)
  EndFor
EndFor
```

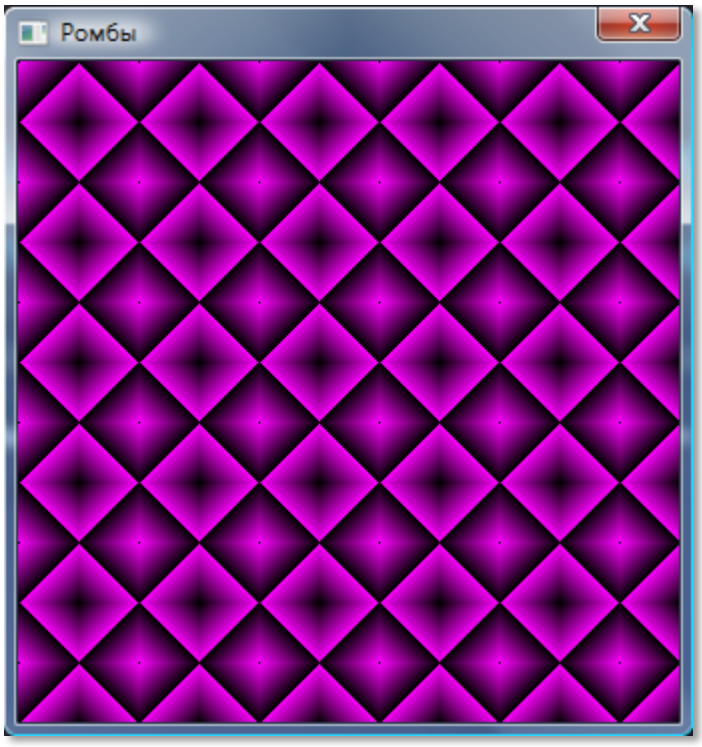


Рис. 19.7. Канва в ромбик

Исходный код программы находится в папке **Ромбы**.

Синусоиды Винни-Пуха

По никому неведомой причине неизвестный автор этих синусоид (Рис. 19.8) посвятил их нашему любимому Винни, прославившемуся своим пыхтеньем (следствие непомерного обжорства).

В этом примере 9 параметров, входящих в формулу для вычисления цвета пикселя, выбираются случайно в начале программы:

```
GraphicsWindow.Title="Синусы Пуха"

. . .

'длина волны:
wX= 10
wY= 10
For i= 1 To 9
```

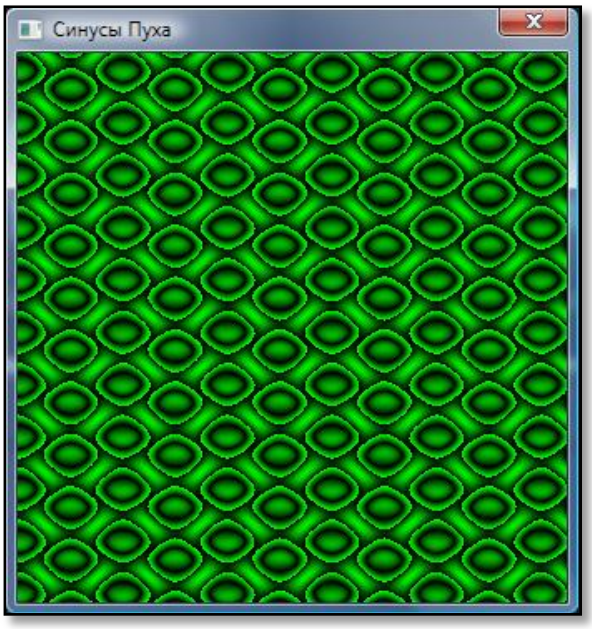
```
  a[i]= Math.GetRandomNumber(2)
EndFor
```

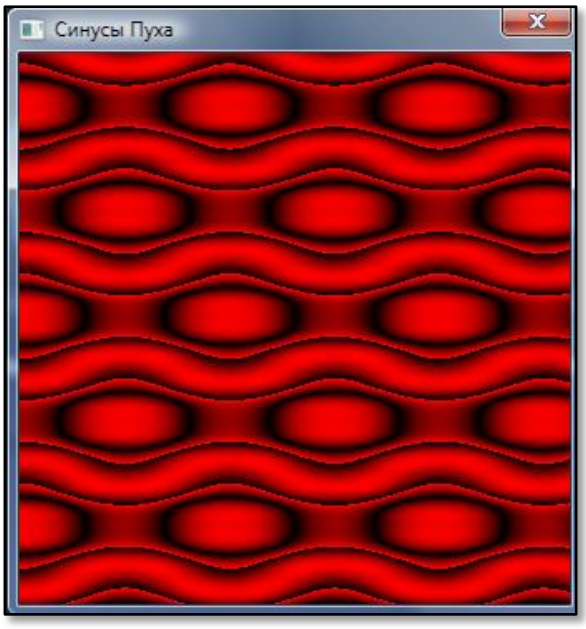
Это значит, что при каждом новом запуске программы вы будете получать новую картинку. Вот где простор для поиска новых узоров!

С другой стороны, случайный поиск не очень эффективен. Может быть, стоит попробовать подбирать коэффициенты вручную?

Формула для вычисления цвета пикселей просто ужасная, но компьютер мы ей не напугаем:

```
for y=0 to height-1
  for x=0 to width-1
    'цвет очередного пикселя:
    clr = 256*(1 + (a[4]*Math.Sin(a[0]*Math.Sin(a[6] * x/wX) +
a[1]*Math.Cos(a[7] * y/wY)) +
a[5]*Math.Cos(a[2]*Math.Cos(a[8]* x/wX) +  a[3]*Math.Sin(a[9]*
y/wY)))) /2
    clr=GraphicsWindow.GetColorFromRGB(0,0,clr)
    'окрашиваем его:
    GraphicsWindow.SetPixel(x,y,clr)
  EndFor
EndFor
```





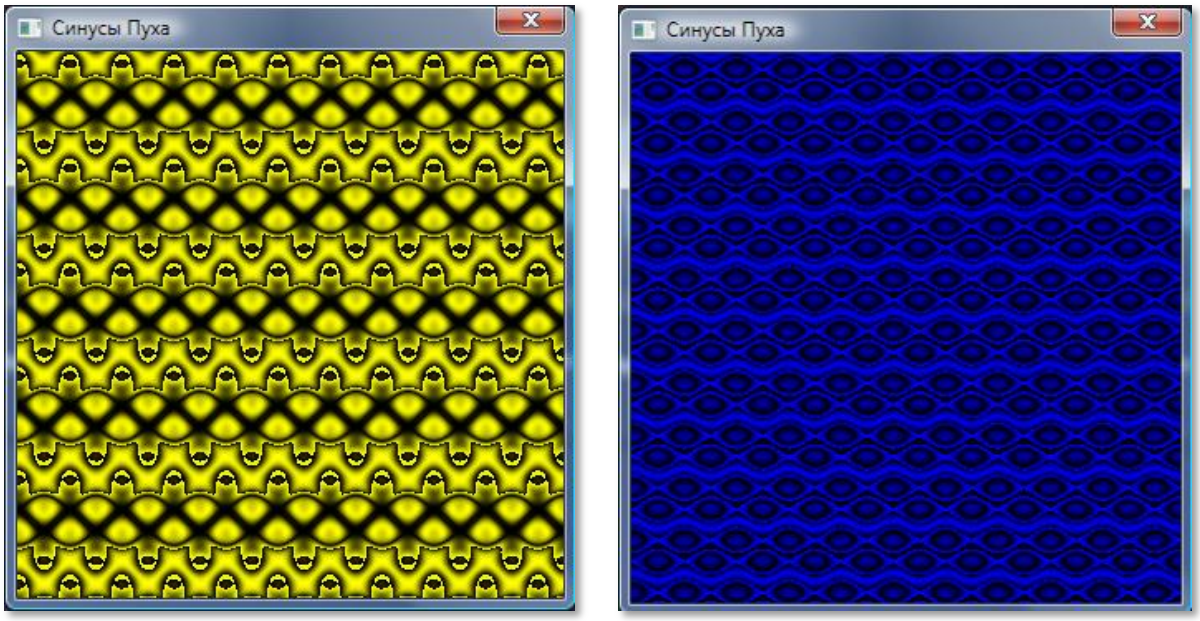


Рис. 19.8. Так как параметры для вычисления цвета пикселей выбираются случайно, то каждый раз вы будете получать другую картинку!

Исходный код программы находится в папке **Sine the pooh**.

Туманность

Аналогично, выбирая случайные параметры для формулы в этом примере, мы получим причудливые линии, похожие на туманности (Рис. 19.9):

```
GraphicsWindow.Title=" Туманность"
. . .
For i = 1 to 6
  a[i] = Math.Pi * (1-2*Math.GetRandomNumber(1000)/1000)
Endfor

While ("True")
  x = math.Sin (a[1]*a[6]) - math.Cos (a[2]*a[5])
  y = math.Sin (a[3]*a[5]) - math.Cos (a[4]*a[6])
  clr= GraphicsWindow.GetColorFromRGB(128*(x+y),
128*(x+y),255)
  GraphicsWindow.SetPixel(CX+x*100, CY+y*100,clr)
```

```
  a[5] = x
  a[6] = y
EndWhile
```

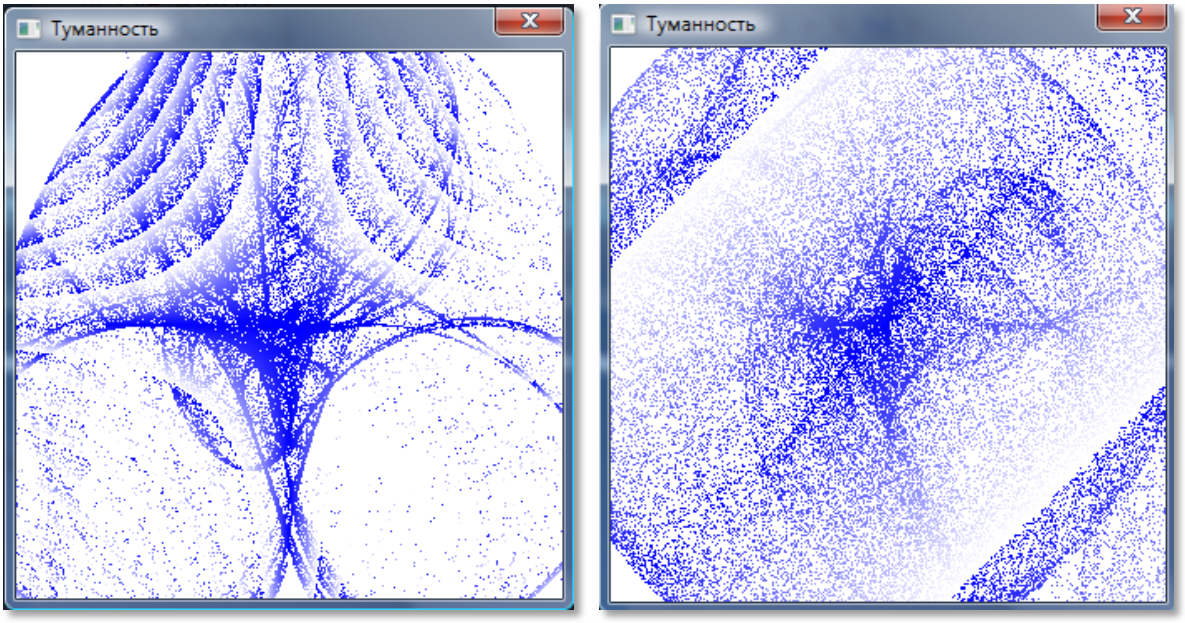


Рис. 19.9. Туманности

Исходный код программы находится в папке **Туманность**.

Урок 20. Занимательная прямолинейность

Если взять один кирпич, мало толку в нём,
Потому что из него не построишь дом.
Если пару кирпичей рядом положить,
Будет только две стены – неудобно жить.

Песенка Тыквы из мультфильма *Чиполлино*

Одна точка, как мы убедились на уроке *Компьютерная графика*, - унылое зрелище. А вот через *две* точки уже можно провести прямую, поэтому попробуем теперь порисовать *линиями*.

Для этого в классе *GraphicsWindow* припасён метод

```
GraphicsWindow.DrawLine( x1, y1, x2, y2 )
```

Точка с координатами *(x1, y1)* задаёт *начало* прямой, а точка *(x2, y2)* – её *конец*, то есть будет правильнее сказать, что этот метод вычерчивает *отрезок* прямой (Рис. 20.1).

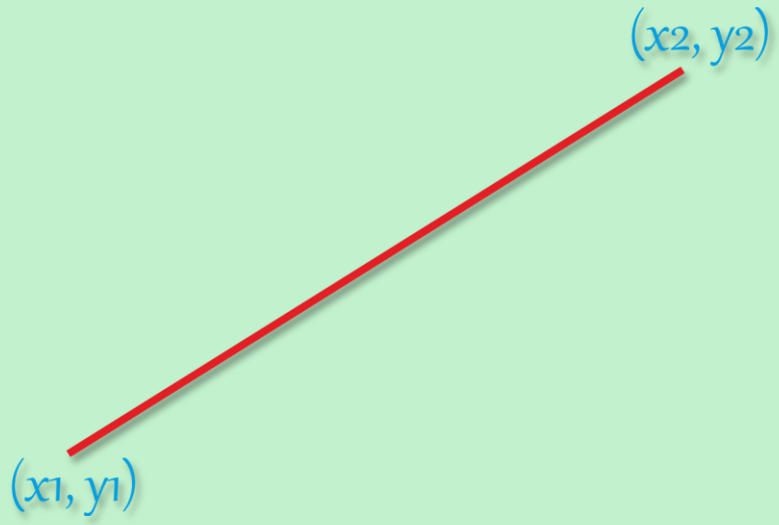


Рис. 20.1. Прямая линия

Толщина линий определяется свойством *PenWidth*, а её цвет – свойством *PenColor* того же класса *GraphicsWindow.*

Поскольку для рисования линий достаточно взять две точки, то мы начнём новый проект **Случайные линии** с того, что загрузим в *СБ* исходный код программы *Pixel* и сохраним его в новой папке.

Сама программа будет совершенно бесхитростной, поскольку рисовать прямые линии очень просто. Нам потребуется всего одна *переменная* – для хранения толщины линий:

```
' ПРОГРАММА ДЛЯ ВЫЧЕРЧИВАНИЯ
' СЛУЧАЙНЫХ ЦВЕТНЫХ ЛИНИЙ

'var
' толщина линий:
penWidth= 20

GraphicsWindow.Title=" Случайные линии"

GraphicsWindow.Width= 480
GraphicsWindow.Height=320
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"
GraphicsWindow.BackgroundColor="Black"
height=GraphicsWindow.Height
width=GraphicsWindow.Width
```

Вместо координат одной точки, как при рисовании пикселей, нам понадобятся координаты *двух* точек, которые мы затем и соединим прямой линией посредством вызова метода *DrawLine* с соответствующими параметрами:

```
GraphicsWindow.PenWidth = penWidth

' Чертим цветные линии
While "True"
  ' задаем случайные координаты начала и конца линии -->
  ' координаты первой точки:
  x1= Math.GetRandomNumber(width - penWidth / 2 - 1)
  y1= Math.GetRandomNumber(height - penWidth / 2 - 1)
  ' координаты второй точки:
```

```
  x2= Math.GetRandomNumber(width - penWidth / 2 - 1)
  y2= Math.GetRandomNumber(height - penWidth / 2 - 1)
  'выбираем случайный цвет линии:
  clr= GraphicsWindow.GetRandomColor()
  GraphicsWindow.PenColor= clr
  'проводим линию:
  GraphicsWindow.DrawLine(x1, y1, x2, y2)
  Program.Delay(30)
EndWhile
```

Запускаем программу – и тут уж ничего не добавишь: картина куда краше, чем с точками (Рис. 20.2). А мало этого, увеличьте толщину пера до 20 – и смотрите сами (Рис. 20.3):

```
penWidth= 20
```

Если толщина линий большая, то они больше напоминают прямоугольники.

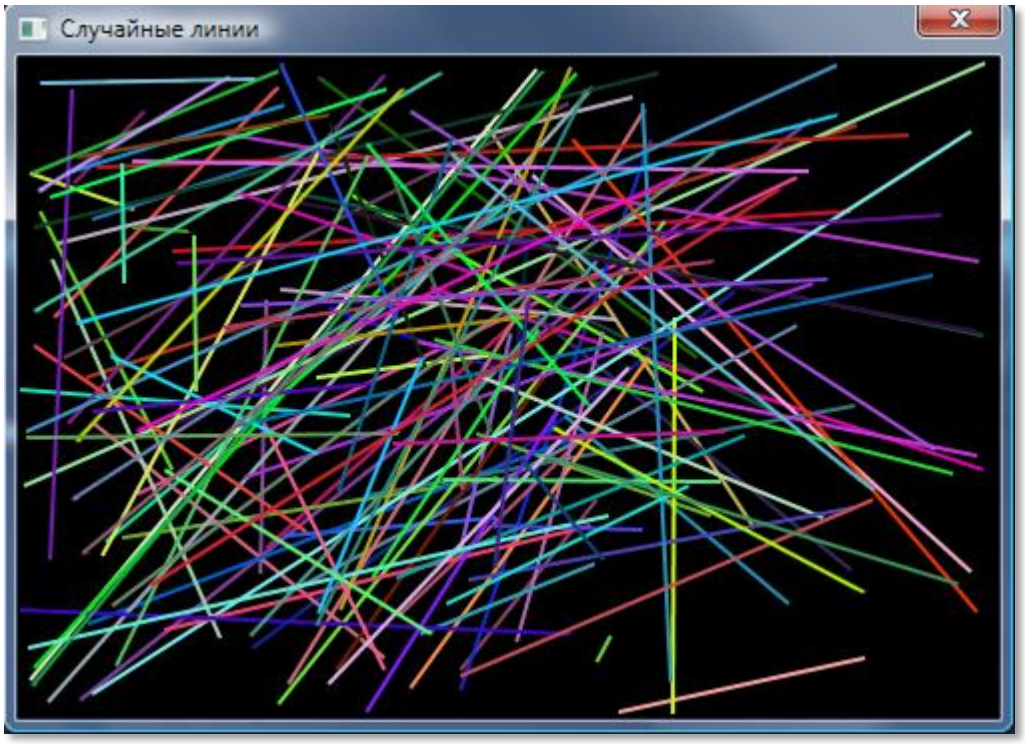


Рис. 20.2. Случайные линии

Исходный код программы находится в папке **Случайные линии**.

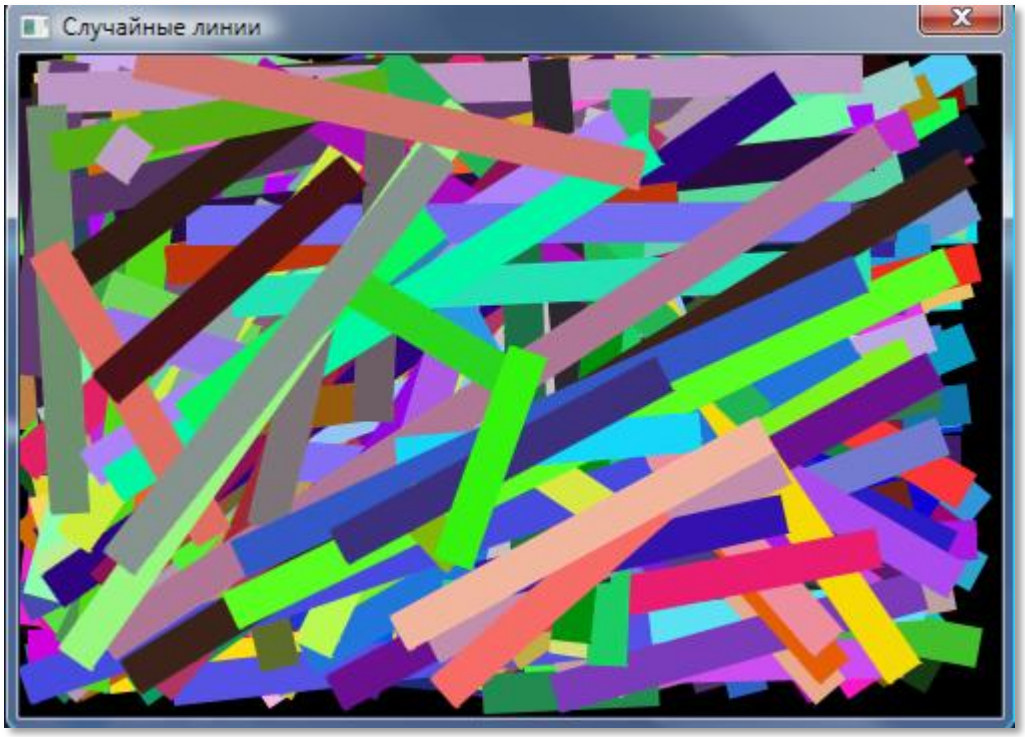


Рис. 20.3. Толстые случайные линии

Поскольку отрезки беспорядочно разбросаны по экрану, то глаз им не радуется. Однако есть много способов и с помощью одних только прямых линий нарисовать шикарные узоры. Сейчас мы напишем совсем короткую программу, которая «намалюет» великолепную картинку (Рис. 20.4).

```
'ПРОГРАММА "ЦВЕТНЫЕ ЛИНИИ"

GraphicsWindow.Title="Цветные линии"

GraphicsWindow.Width= 640
GraphicsWindow.Height=480
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"

height=GraphicsWindow.Height
width= GraphicsWindow.Width
'отношение высоты окна к ширине:
ratio= height / width
```

```
GraphicsWindow.PenWidth=1
GraphicsWindow.BackgroundColor="Black"
for x= 0 to width Step 10
  GraphicsWindow.PenColor= "Red"
  GraphicsWindow.DrawLine(0, x*ratio, width-x, 0)
  GraphicsWindow.PenColor= "Yellow"
  GraphicsWindow.DrawLine(0, (width-x)*ratio, width-x,
width*ratio)
  GraphicsWindow.PenColor= "Blue"
  GraphicsWindow.DrawLine(width-x, 0*ratio, width, (width-
x)*ratio)
  GraphicsWindow.PenColor= "Green"
  GraphicsWindow.DrawLine(width-x, width*ratio, width,
x*ratio)
EndFor
```

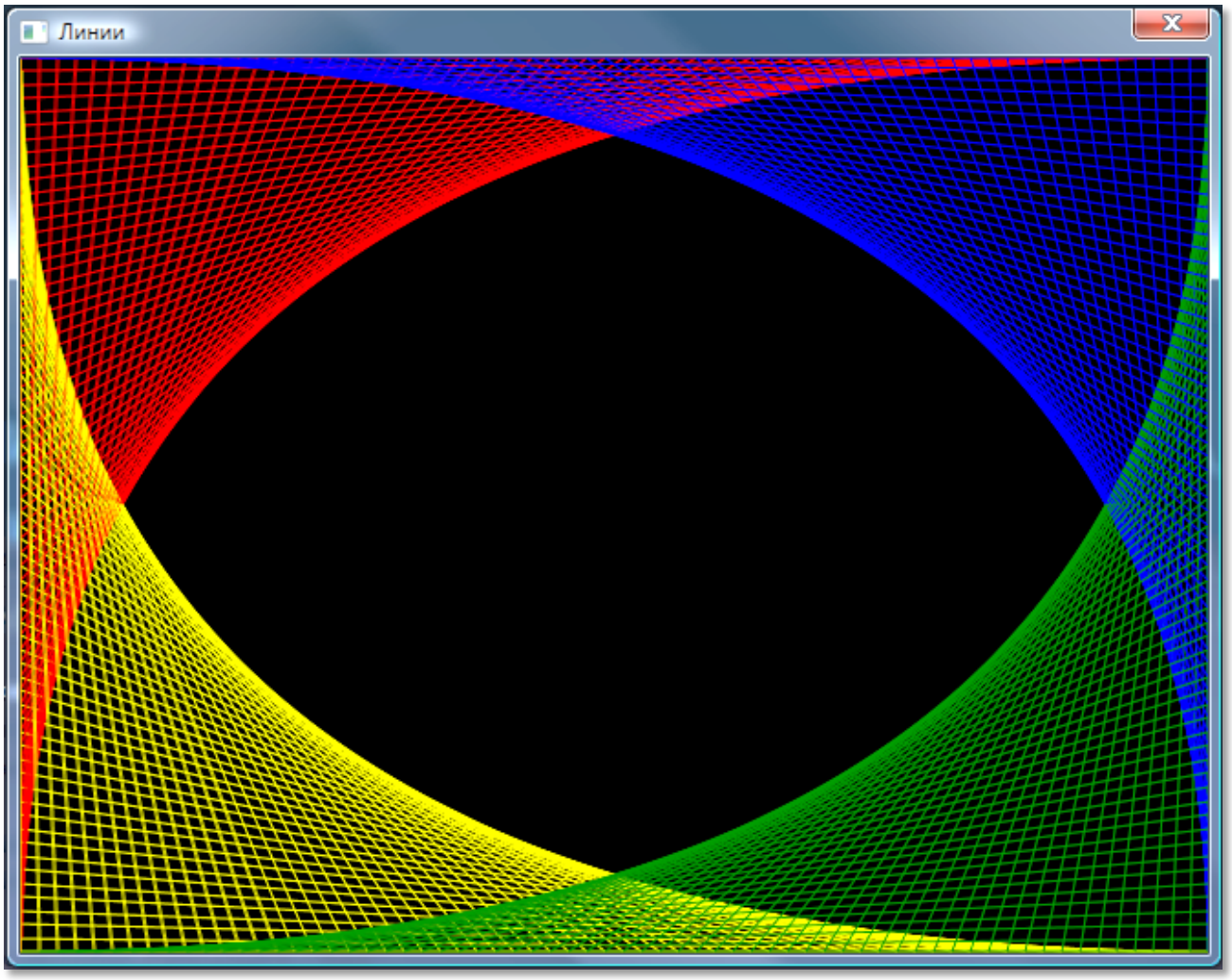


Рис. 20.4. Цветные линии

Чтобы каждый из четырёх наборов линий имел свой цвет, перед рисованием каждой прямой мы задаём её цвет методом *Графического окна PenColor*. Как изменяются координаты начала и конца линии, хорошо видно на рисунке – они скользят по границам клиентской области окна с заданным шагом *Step*.

Изменяя этот параметр, а также цвет линий, вы сможете получить и другие узоры.

Исходный код программы находится в папке **Линии**.

Градиентная заливка

Если чертить горизонтальные линии вплотную друг к другу, то можно получить картину, ласкающую взор. Сейчас мы испытаем модную нынче *градиентную* заливку, в которой один цвет *плавно* переходит в другой. Этот приём часто применяется в заставках, которые появляются на экране при установке программ, а также в компьютерной графике для раскрашивания кнопок, баннеров и других объектов.

Хорошим примером натуральной «градиентной заливки» может служить безоблачное полуденное или закатное небо!

Сохраните исходный код предыдущего проекта в папке **Градиент** и добавьте три *переменные* - для хранения **красной**, **зелёной** и **синей** составляющих текущего цвета линий:

```
'ПРОГРАММА ДЛЯ ГРАДИЕНТНОЙ
'ЗАЛИВКИ ПРЯМОУГОЛЬНИКА

'var
'толщина линий:
penWidth= 3
'составляющие цвета:
r=0
g=0
b=0
```

```
GraphicsWindow.Title=" Градиент"

. . .
```

«Теоретически» толщина линий должна быть равна одному пикселю, но тогда цвет заливки получается грязным, что обусловлено особенностями вычерчивания линий в *СБ*.

Всего наша программа сможет выполнить 6 различных градиентных заливок, поэтому нам потребуется именно столько *кнопок*, чтобы пользователь мог по своему желанию выбирать последовательность их просмотра. Для сокращения исходного кода все кнопки мы объединим в массив *btn*:

Кнопки мы хорошенько изучим на уроке *Элементы управления*.

```
'КНОПКИ

y= 10
dy=32
x= width-85
n=1
btn[n]=Controls.AddButton("Ж --> К", x, y+dy*(n-1))
Controls.SetSize(btn[n], 80, 24)
n=n+1
btn[n]=Controls.AddButton("Ж --> З", x, y+dy*(n-1))
Controls.SetSize(btn[n], 80, 24)
n=n+1
btn[n]=Controls.AddButton("Ц --> З", x, y+dy*(n-1))
Controls.SetSize(btn[n], 80, 24)
n=n+1
btn[n]=Controls.AddButton("Ц --> С", x, y+dy*(n-1))
Controls.SetSize(btn[n], 80, 24)
n=n+1
btn[n]=Controls.AddButton("Л --> С", x, y+dy*(n-1))
Controls.SetSize(btn[n], 80, 24)
n=n+1
btn[n]=Controls.AddButton("Л --> К", x, y+dy*(n-1))
Controls.SetSize(btn[n], 80, 24)

Controls.ButtonClicked=OnClick
```

Надписи на кнопках обозначают направление градиентного перехода, а начальный и конечный цвета указаны первой буквой названия цвета:

Ж – жёлтый

К – красный

Ц – циан

С – синий

Л – лиловый

После нажатия на любую из этих кнопок программа передаёт управление процедуре-обработчику *OnClick*, в которой выполняется подпрограмма, закреплённая за нажатой кнопкой:

```
'Начинаем закраску
Sub OnClick
  button= Controls.LastClickedButton
  If (button= btn[1]) Then
    YR()
  ElseIf (button= btn[2]) Then
    YG()
  ElseIf (button= btn[3]) Then
    CG()
  ElseIf (button= btn[4]) Then
    CB()
  ElseIf (button= btn[5]) Then
    MB()
  ElseIf (button= btn[6]) Then
    MR()
  EndIf
EndSub
```

Названия подпрограмм также обозначены первыми буквами цветов, образующих градиент, так что запутаться в них вам вряд ли удастся.

Поскольку сами процедуры, в отличие от заливок, весьма однообразны, то мы рассмотрим только две их них. Остальные вы можете посмотреть самостоятельно в исходном коде программы.

При переходе от **циана** к **синему** цвету (Рис. 20.5) *синяя* составляющая текущего цвета всегда равна 255 (максимальное значение), *красная* – нулю (отсутствует вообще), а *зелёная* составляющая постепенно уменьшает своё значение при перемещении линий сверху вниз от 255 до 0. Именно благодаря этому изменению и возникает плавный переход цвета по вертикали. Так как по горизонтали цвет остается без изменений, то достаточно провести горизонтальную линию текущего цвета:

```
' переход от циана  к синему - Cyan To Blue
Sub CB
  b = 255
  r = 0
  for i = 0 to height
    'вычислить интенсивность зелёной составляющей цвета:
    g= 255 * (1 - i / height)
    'задаём цвет линии:
    clr= GraphicsWindow.GetColorFromRGB(r,g,b)
    GraphicsWindow.PenColor= clr
    'проводим горизонтальную линию:
    GraphicsWindow.DrawLine(0, i, width-90, i)
    Program.Delay(1)
  EndFor
EndSub
```

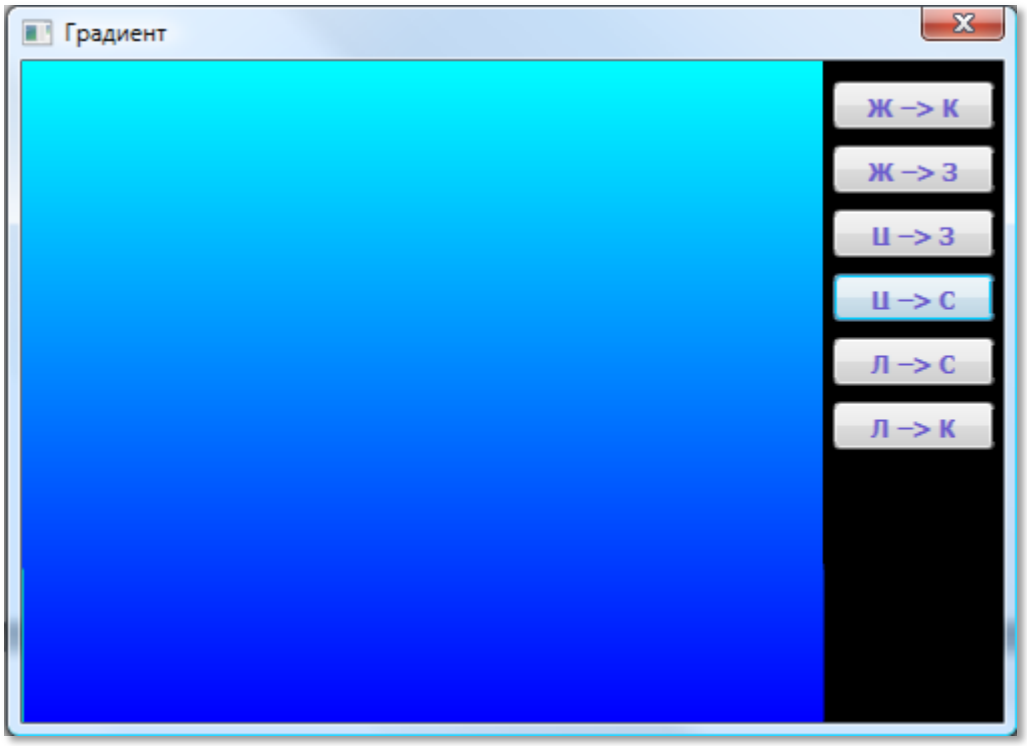


Рис. 20.5. Переход от **циана** к **синему**

Для создания градиентного перехода от **лилового** цвета к **красному** (Рис. 20.6) мы поступаем аналогично. От предыдущего градиента этот градиент отличается только составляющими цвета:

```
'переход от лилового к красному - Magenta To Red
Sub MR
  r = 255
  g = 0
  for i = 0 to height
    'вычислить интенсивность синей составляющей цвета:
    b= 255 * (1 - i / height)
    'задаем цвет линии:
    clr= GraphicsWindow.GetColorFromRGB(r,g,b)
    GraphicsWindow.PenColor= clr
    'проводим горизонтальную линию:
    GraphicsWindow.DrawLine(0, i, width-90, i)
    Program.Delay(1)
  EndFor
EndSub
```

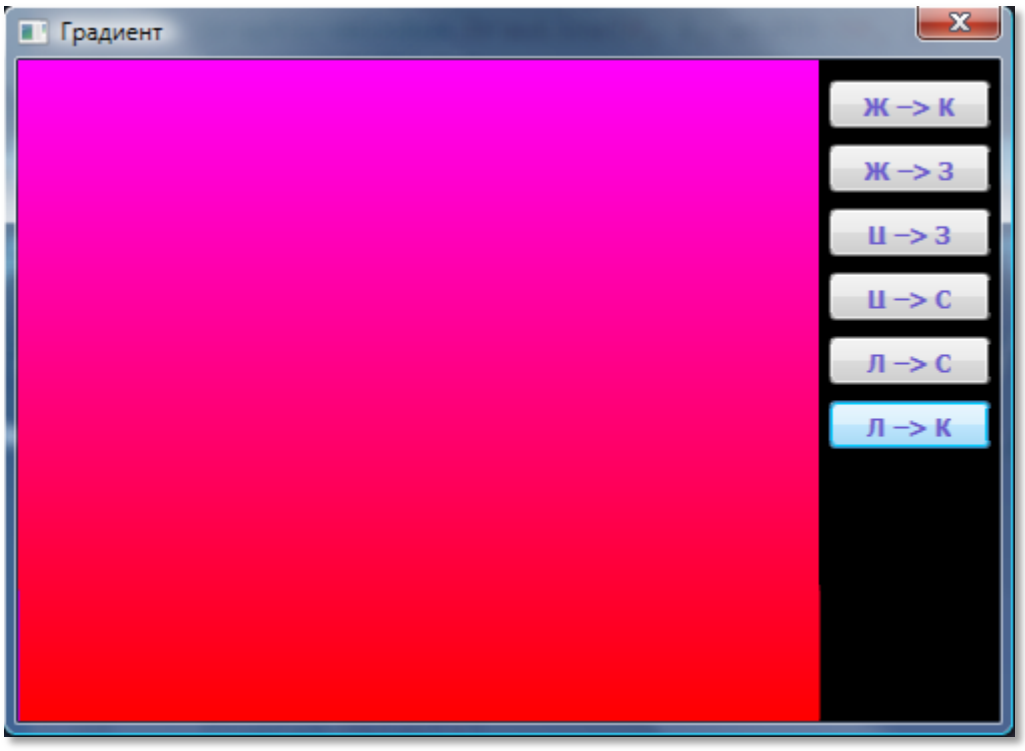


Рис. 20.6. Переход от **лилового** цвета к **красному**

Как видите, создать прямоугольную градиентную заливку совсем несложно, а результат получается впечатляющий! Запускайте

программу, жмите на кнопки и наслаждайтесь **цветовыми** переходами!

Исходный код программы находится в папке **Градиент**.

1. Измените программу *Градиент* так, чтобы она чертила линии вертикально (горизонтальный градиент) (Рис. 20.7)!

2. Подумайте, как запрограммировать двойной (Рис. 20.8), радиальный (Рис. 20.9) и квадратный (Рис. 20.10) градиенты.

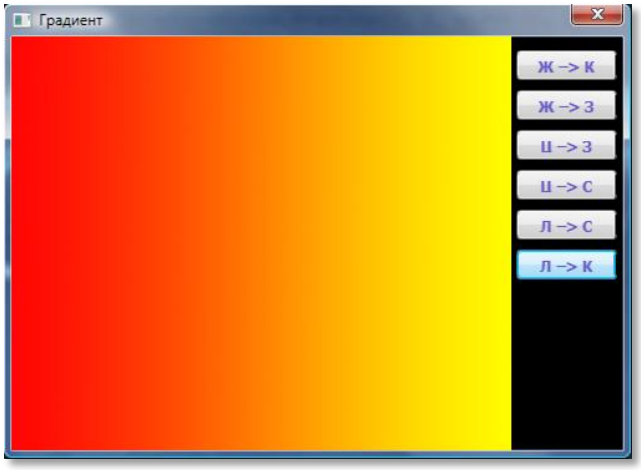


Рис. 20.7. Горизонтальный градиент

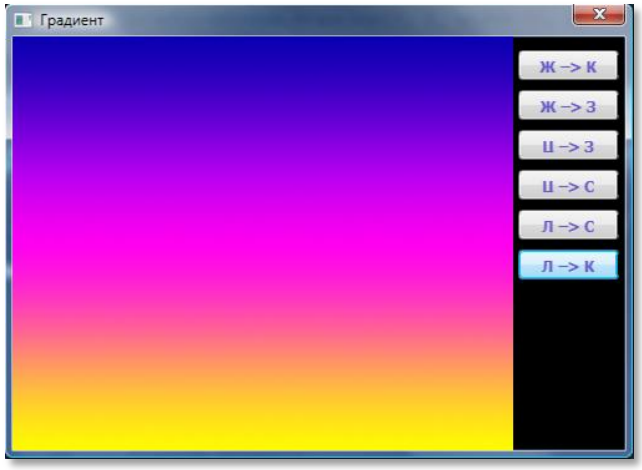


Рис. 20.8. Двойной градиент

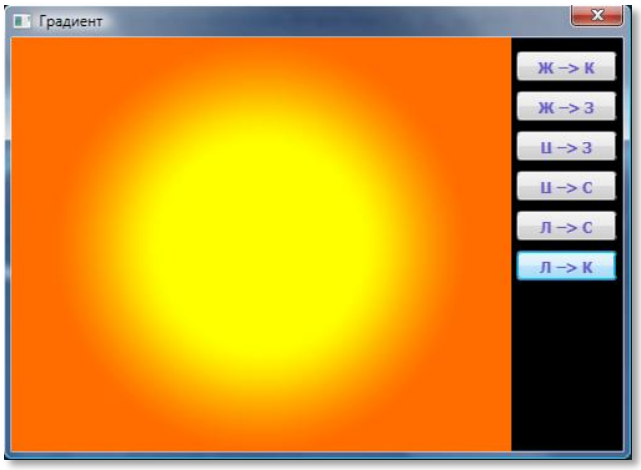


Рис. 20.9. Радиальный градиент

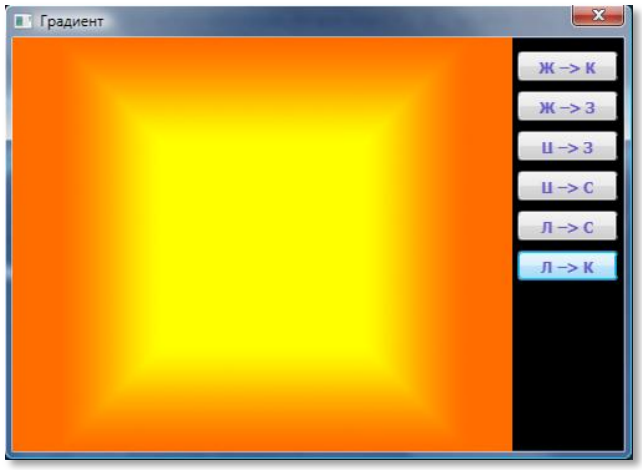


Рис. 20.10. Квадратный градиент

Урок 21. Геометрические фантазии

Из отрезков можно построить любые многоугольники, но для прямоугольников и треугольников в *СБ* имеются специальные методы.

Прямоугольники и квадраты

Полезный во всех отношениях метод

```
GraphicsWindow.DrawRectangle(x, y, width, height)
```

рисует *контурный прямоугольник*, верхний левый угол которого задаётся координатами (*X, Y*), а ширина и высота определяются параметрами *width* и *height*. Цвет контура устанавливается с помощью свойства *PenColor Графического окна*, а его толщина – с помощью свойства *PenWidth* (Рис. 21.1).

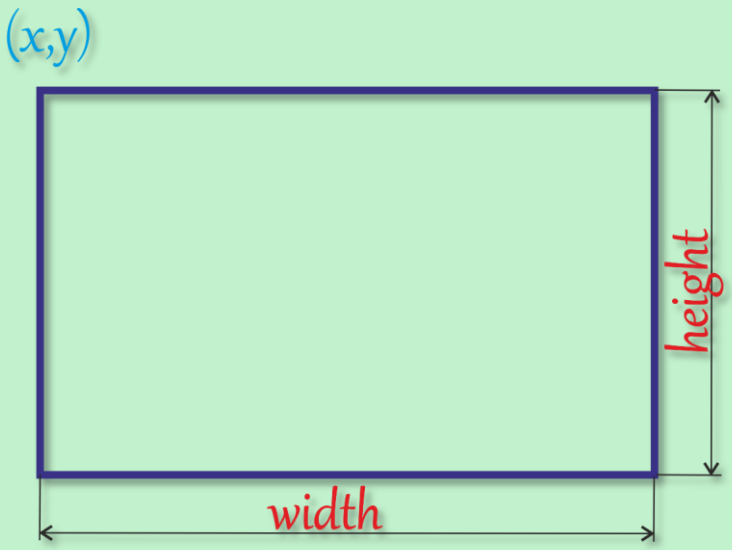


Рис. 21.1. *Контурный* прямоугольник

Для рисования *закрашенных прямоугольников* имеется метод

```
GraphicsWindow.FillRectangle(x, y, width, height)
```

Их размеры и положение на экране задаются точно так же, как и контурных прямоугольников, но они не имеют контура, а цвет заливки определяется цветом кисти *BrushColor Графического окна* (Рис. 21.2).

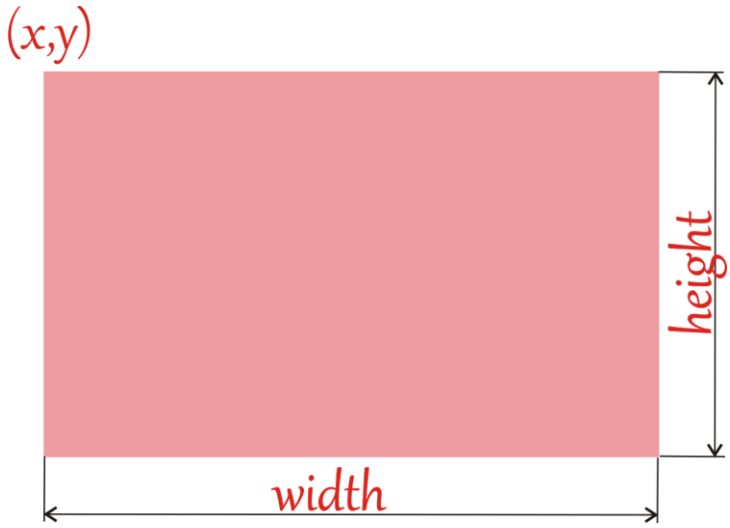


Рис. 21.2. *Закрашенный* прямоугольник

Если ширина прямоугольника равна высоте, то получится *квадрат*.

А теперь давайте позабавимся, рисуя случайные прямоугольники.

Сохраните исходный код программы *Случайные линии* в папке **Прямоугольники**.

Начнём мы с контурных прямоугольников. Нам придётся изменить только операторы в цикле рисования фигур. Теперь вместо прямых линий мы будем рисовать прямоугольники:

```
'ПРОГРАММА ДЛЯ ВЫЧЕРЧИВАНИЯ
'СЛУЧАЙНЫХ ПРЯМОУГОЛЬНИКОВ

'var
'толщина контура:
penWidth= 5
```

```
GraphicsWindow.Title="Случайные прямоугольники"

. . .

'Чертим цветные прямоугольники
While "True"
  'координаты левой верхней вершины прямоугольника:
  x= Math.GetRandomNumber(width - penWidth / 2 - 1)
  y= Math.GetRandomNumber(height - penWidth / 2 - 1)
  'размеры прямоугольника:
  w= Math.GetRandomNumber(width - x - penWidth / 2 - 1)
  h= Math.GetRandomNumber(height - y - penWidth / 2 - 1)
  'выбираем случайный цвет контура:
  clr= GraphicsWindow.GetRandomColor()
  GraphicsWindow.PenColor= clr
  'рисуем контурный прямоугольник:
  GraphicsWindow.DrawRectangle(x, y, w, h)
  Program.Delay(30)
EndWhile
```

Запускаем программу и визуально радуемся нашим «канвасным» достижениям (Рис. 21.3).

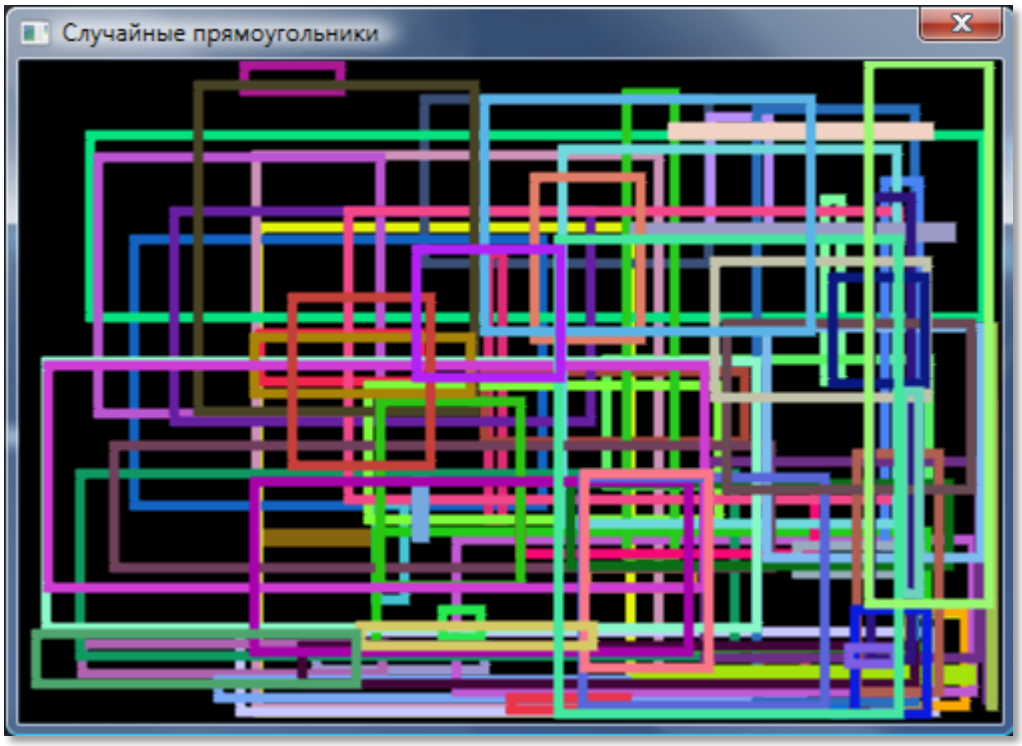


Рис. 21.3. *Контурные* прямоугольники

Совсем немного изменим программу, и вместо контурных прямоугольников на нас обрушится лавина прямоугольников, окрашенных во все **цвета радуги** (Рис. 21.4):

```
While "True"
  ' координаты левой верхней вершины прямоугольника:
  x= Math.GetRandomNumber(width)
  y= Math.GetRandomNumber(height)
  ' размеры прямоугольника:
  w= Math.GetRandomNumber(width - x)
  h= Math.GetRandomNumber(height - y)
  ' выбираем случайный цвет заливки:
  clr= GraphicsWindow.GetRandomColor()
  GraphicsWindow.BrushColor=clr
  ' рисуем закрашенный прямоугольник:
  GraphicsWindow.FillRectangle(x, y, w, h)
  Program.Delay(30)
EndWhile
```

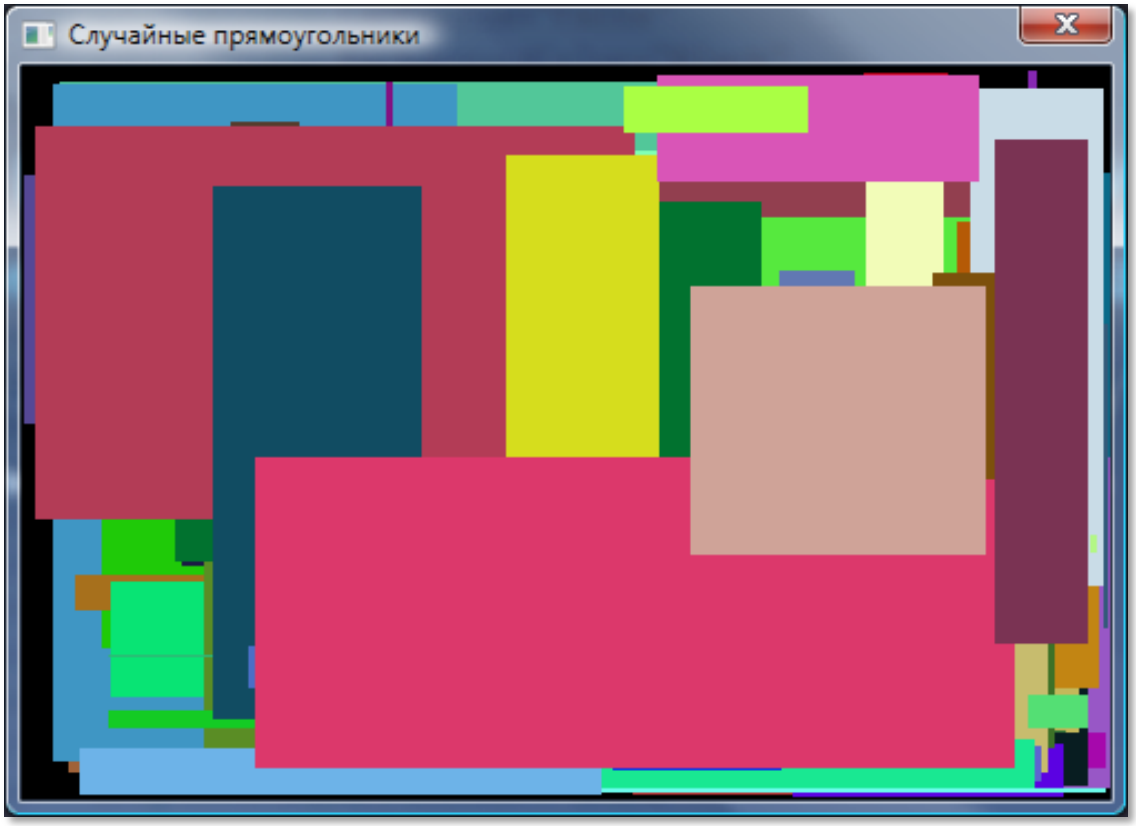


Рис. 21.4. *Закрашенные* прямоугольники

Эллипсы и круги

Для рисования *эллипсов Графическое окно* также предоставляет нам два метода, которые почти ничем не отличаются от «прямоугольных». Единственное различие состоит в том, что у эллипса нет вершин, поэтому его положение на канве задается координатами верхнего левого угла описанного прямоугольника.

Метод

```
GraphicsWindow.DrawEllipse(x, y, width, height)
```

рисует *контурный эллипс* (Рис. 21.5).

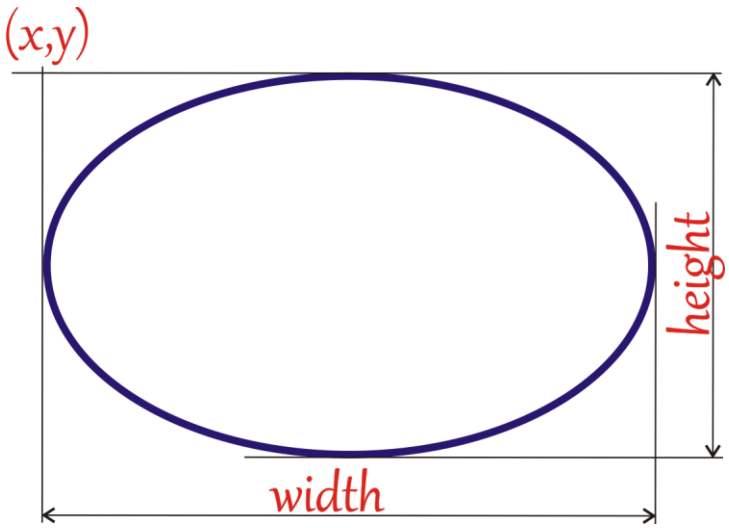


Рис. 21.5. *Контурный* эллипс

А второй метод

```
GraphicsWindow.FillEllipse(x, y, width, height)
```

рисует *закрашенный эллипс* (Рис. 21.6).

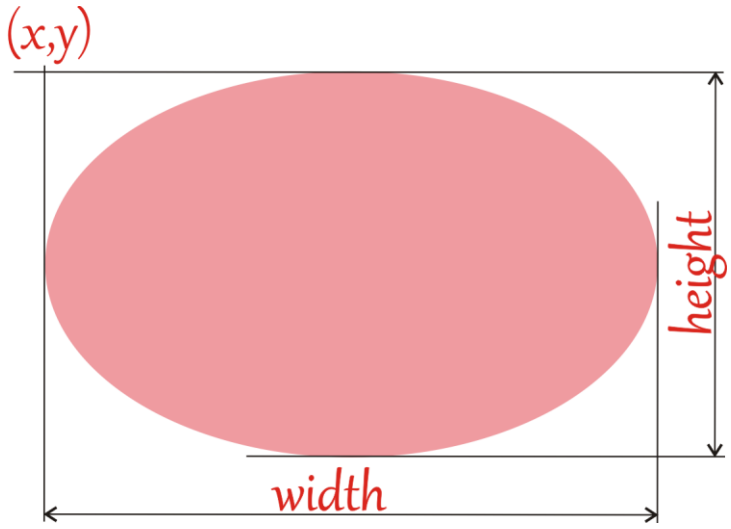


Рис. 21.6. *Закрашенный* эллипс

Если ширина эллипса равна высоте, то получится *круг*.

Измените обе программы для рисования прямоугольников так, чтобы они рисовали эллипсы!

Исходный код программы находится в папке **Прямоугольники**.

Треугольники

Вероятно, вы уже догадались, что и для вычерчивания треугольников *Графическое окно* также запаслось двумя методами (Рис. 21.7 и 21.8):

```
GraphicsWindow.DrawTriangle(x1, y1, x2, y2, x3, y3)

GraphicsWindow.FillTriangle(x1, y1, x2, y2, x3, y3)
```

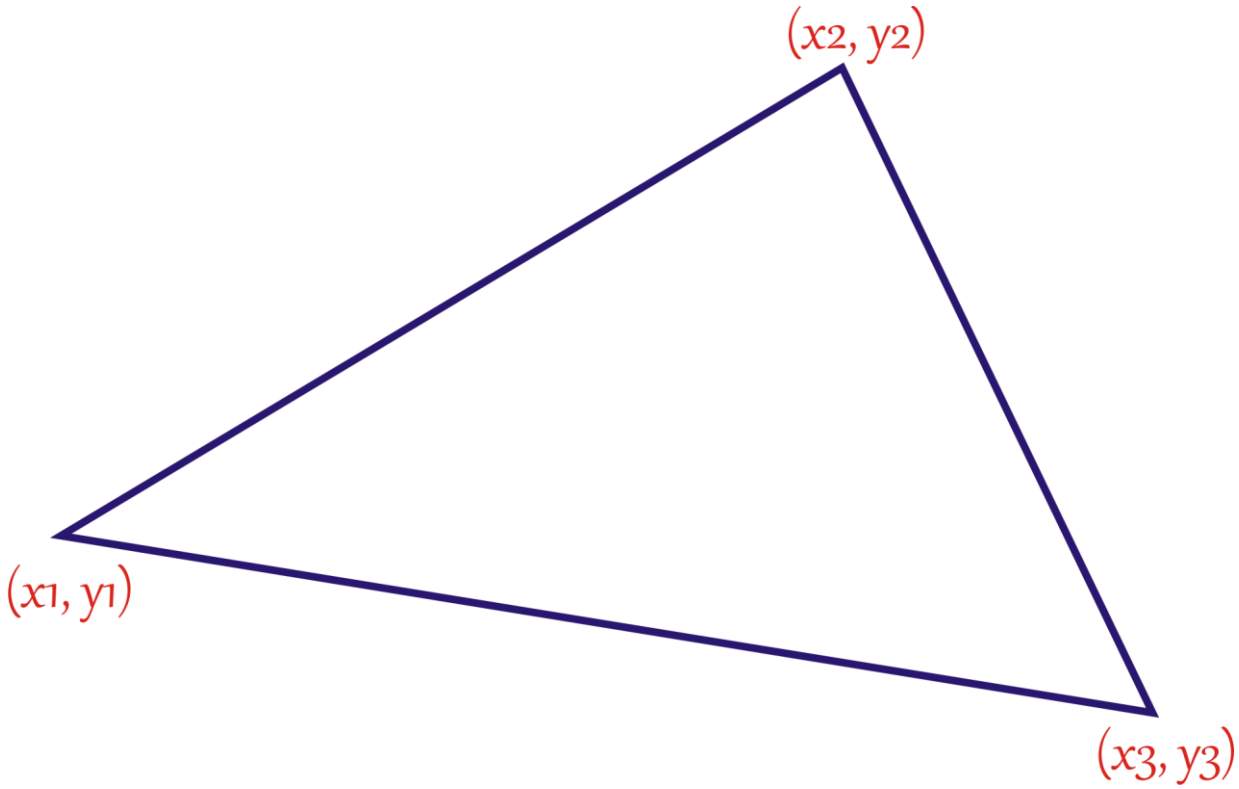


Рис. 21.7. *Контурный* треугольник

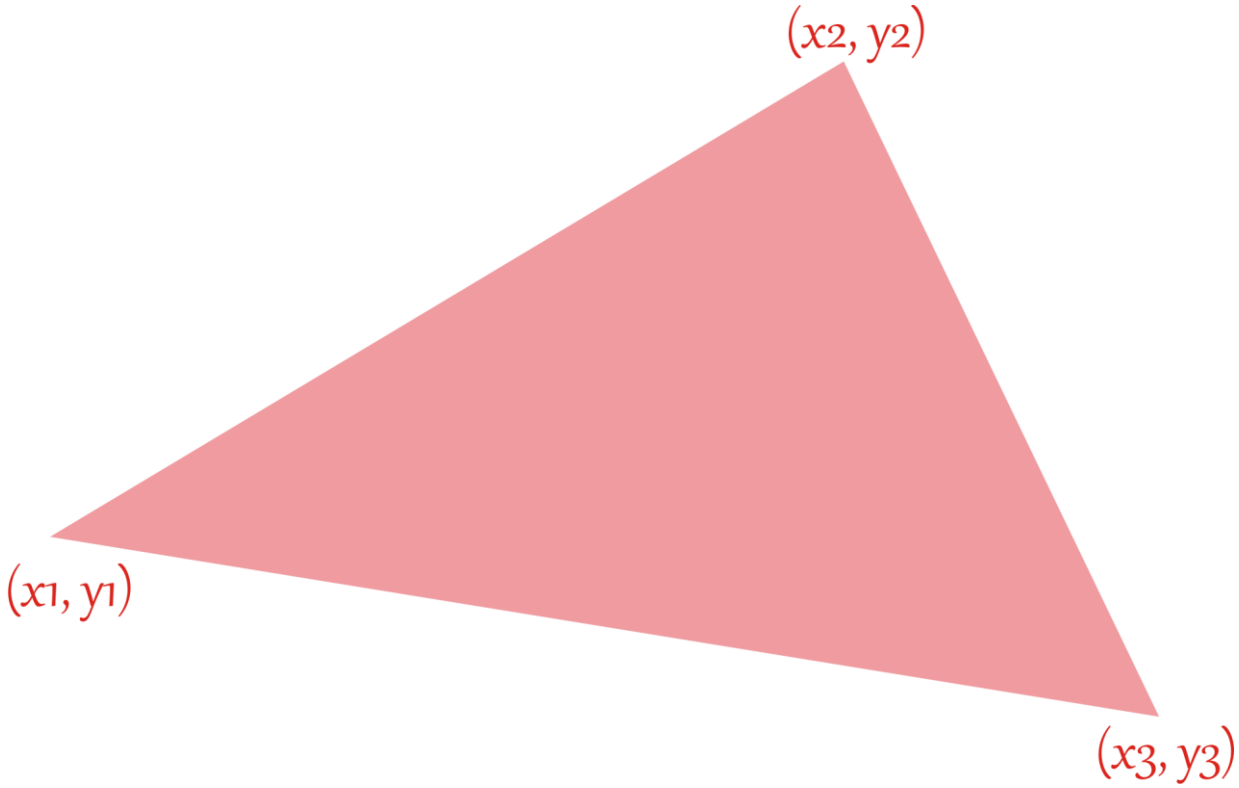


Рис. 21.8. *Закрашенный* треугольник

Проще всего задать размеры треугольника и его положение на канве координатами трёх его вершин, что и сделано в этих методах.

Дабы лишний раз не повторяться, мы не будем рисовать случайные треугольники, а построим из маленьких треугольников один большой. Это будет свежо и экспрессивно!

Загрузите в *СБ* проект *Прямоугольники* и сохраните его в папке **Треугольники**. Добавьте пару *переменных* – для хранения *длины* сторон правильных треугольников и *цвета* их контура. Также придётся увеличить высоту окна для гармонизации нашего произведения:

```
'ПРОГРАММА ДЛЯ ВЫЧЕРЧИВАНИЯ
'ТРЕУГОЛЬНИКА ИЗ ТРЕУГОЛЬНИКОВ

'var
'толщина контура:
penWidth= 2
'длина стороны треугольника:
size= 32
'цвет контура:
penColor= "Yellow"

GraphicsWindow.Title="Треугольники"

GraphicsWindow.Width= 480
GraphicsWindow.Height=440
. . .
```

Маленькие треугольники обращены вершиной вниз и образуют столбики, высота которых уменьшается по мере удаления столбиков от центра.

Первый столбик начинается треугольником с координатами левой вершины (*x1, y1*), а следующие столбики смещены влево и вправо на *size/4*3*(j)* пикселей. Кроме того, каждый следующий столбик начинается ниже предыдущего на *j*dy*1.5* пикселей.

Почему так происходит, хорошо видно на готовой картинке (Рис. 21.9)

```
GraphicsWindow.PenWidth = penWidth
GraphicsWindow.PenColor= penColor
'координаты левого угла верхнего треугольника:
x1= width/2- size/2
y1= 10
'высота треугольников:
dy= size*math.Sin(Math.GetRadians(60))
'запоминаем начальные значения:
yt=y1
xt=x1

'рисуем столбики из треугольников:
For j= 0 to 14
  'левый столбик:
  y1=yt+j*dy*1.5
  x1=xt- size/4*3*j
  For i= 1 to 15-1.5*j
    drawTre()
    y1= y1+dy
  endFor
  'правый столбик:
  y1=yt+j*dy*1.5
  x1=xt+ size/4*3*j
  For i= 1 to 15-1.5*j
    drawTre()
    y1= y1+dy
  endFor
endFor
```

В процедуре *drawTre* мы рисуем *один* треугольник:

```
'Чертим треугольник
Sub drawTre
  GraphicsWindow.DrawTriangle(x1,y1, x1+size, y1, x1 + size/2,
y1 + dy)
EndSub
```

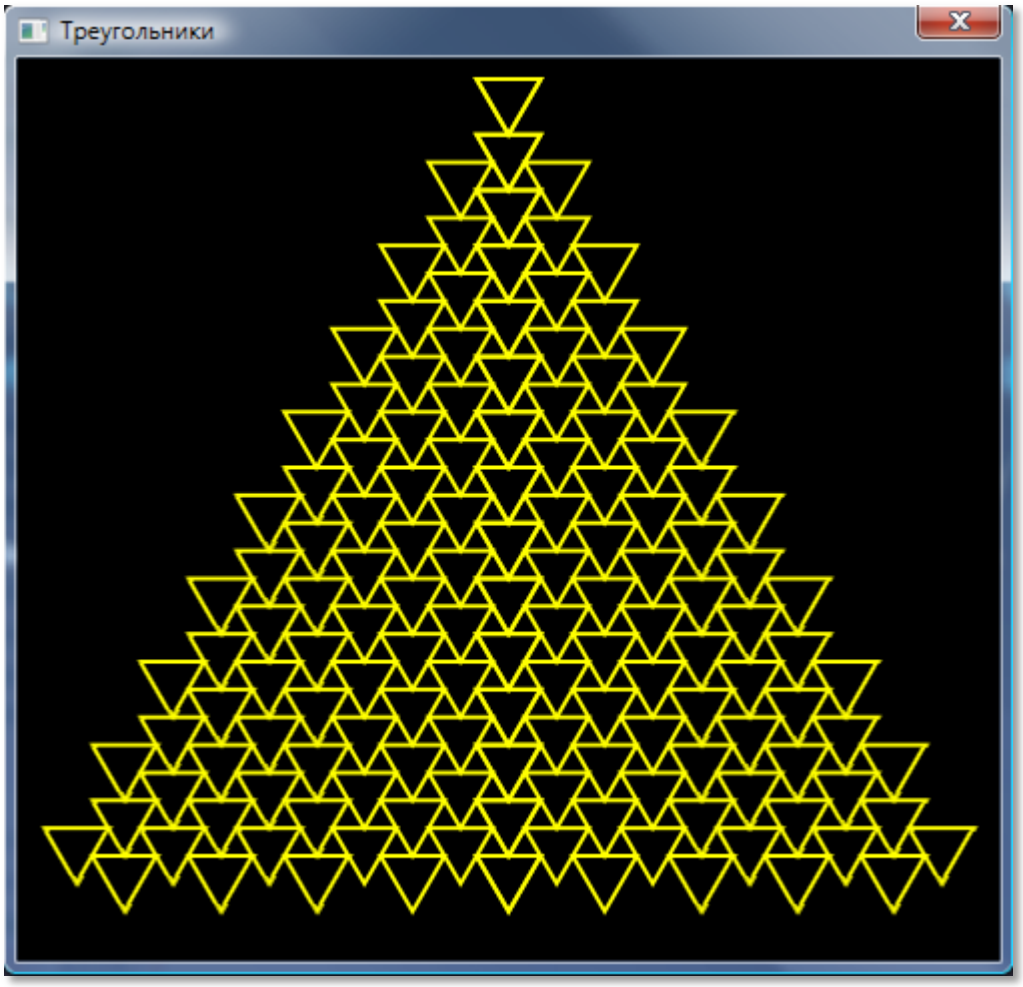


Рис. 21.9. Треугольный треугольник

Исходный код программы находится в папке **Треугольники**.

Живые картинки

До сих пор мы занимались *статическими* картинками: нарисовал их – и всё! А ведь гораздо интереснее наблюдать на экране за непрерывными изменениями геометрических фигур. Сейчас мы напишем программу, которая будет рисовать прямоугольники или эллипсы, непрерывно обновляющие картинку на экране.

Откройте проект *Прямоугольники* и сохраните исходный код в папке **Rectangular** (намёк на то, что картинка составлена из прямоугольников).

Поскольку эта программа не столько сложная, сколько заковыристая, то и переменных нам потребуется немало:

```
'ПРОГРАММА ДЛЯ ВЫЧЕРЧИВАНИЯ
'ДИНАМИЧЕСКИХ ПРЯМОУГОЛЬНИКОВ и ЭЛЛИПСОВ

'var
'толщина контура:
penWidth= 2
'начальные координаты прямоугольника:
x0=0
y0=0
'максимальные координаты прямоугольника:
xmax=0
ymax=0
'текущие координаты прямоугольника:
x1=0
y1=0
x2=0
y2=0
'фигура для рисования:
figura="Rect"

GraphicsWindow.Title="Rectangular"

GraphicsWindow.Width= 480 + 100
GraphicsWindow.Height=320
. . .
```

Обратите внимание, что переменные *(x1, y1)* и *(x2, y2)* задают координаты верхнего левого и правого нижнего углов прямоугольника, а при вызове метода *DrawRectangle* нам потребуется длина сторон прямоугольника, которую нам придётся вычислять дополнительно. Рисовать мы будет прямоугольники и эллипсы, а переменная *figura* будет хранить название текущей геометрической фигуры.

Чтобы переключаться между ними, установим пару *кнопок*:

```
'КНОПКИ

y= 10
```

```
dy=48
x= width-90
n=1
btn[n]=Controls.AddButton("Пр-ки", x, y+dy*(n-1))
Controls.SetSize(btn[n], 80, 32)
n=n+1
btn[n]=Controls.AddButton("Эллипсы", x, y+dy*(n-1))
Controls.SetSize(btn[n], 80, 32)
Controls.ButtonClicked=OnClick
```

После нажатия на одну из этих кнопок мы попадём в процедуру-обработчик *OnClick*:

```
'Выбираем фигуру для рисования:
Sub OnClick
  GraphicsWindow.BrushColor="Black"
  button= Controls.LastClickedButton
  If (button= btn[1]) Then
    GraphicsWindow.FillRectangle(x0, y0, xmax+4, ymax)
    GraphicsWindow.PenWidth = 2
    figura="Rect"
  Else
    GraphicsWindow.FillRectangle(x0, y0, xmax+4, ymax)
    GraphicsWindow.PenWidth = 1
    figura="Ellipse"
  EndIf
EndSub
```

Здесь мы, прежде всего, стираем все предыдущие кривые, устанавливаем разную толщину контура для прямоугольников и эллипсов (вы можете выбрать свои значения) и сообщаем основной части программы, какими фигурами мы желаем рисовать.

Так как справа мы разместили кнопки, то это обстоятельство следует учесть при рисовании:

```
'размеры области рисования:
x0= 0
y0= 0
xmax= width - 100
ymax= height
```

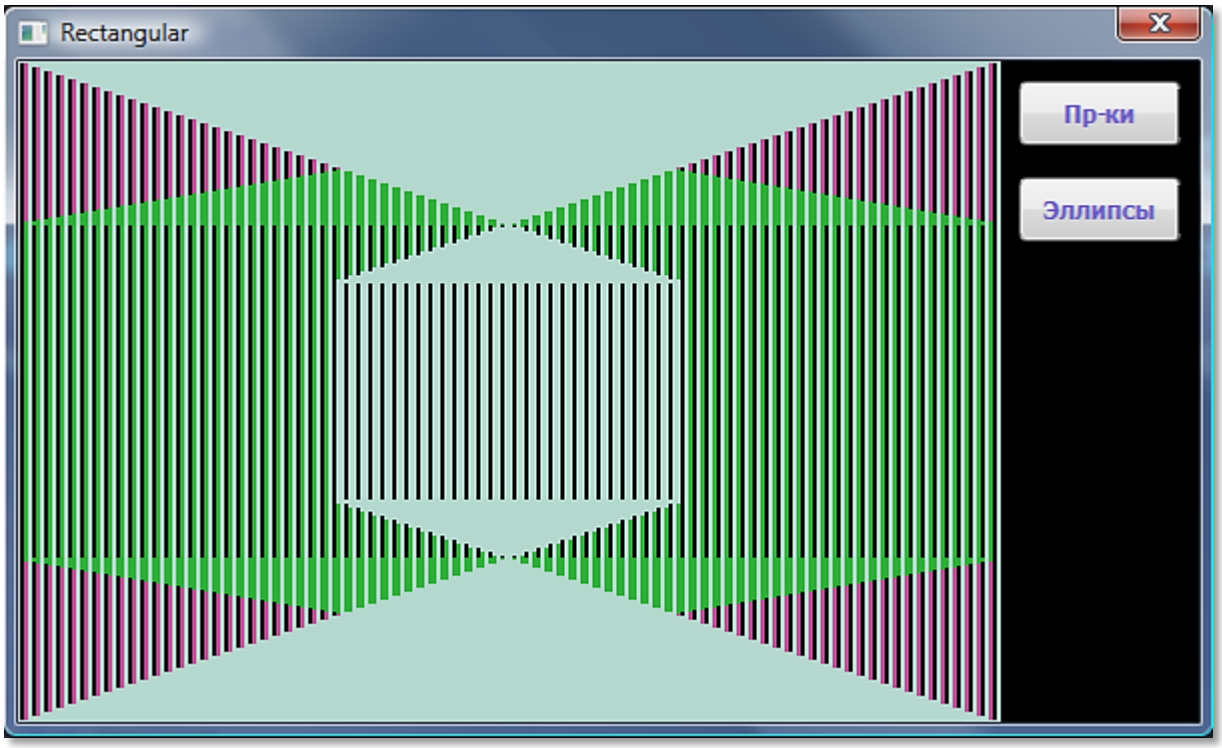


Рис. 21.10. Динамические прямоугольники

Далее, в бесконечном цикле *While* мы вычерчиваем прямоугольники или эллипсы. Начинаем с самой большой фигуры, размеры которой равны области рисования, затем постепенно координаты вершин прямоугольника всё больше и больше приближаются к центру и, наконец, наступает момент, когда левый верхний угол занимает место правее и/или ниже правого нижнего. Вот тут обязательно нужно поменять эти координаты в вызове метода *DrawRectangle*, иначе он будет работать неправильно!

```
GraphicsWindow.PenWidth = penWidth
Чертим цветные прямоугольники
While "True"
  for i= 6 To 1 Step -1
    for j= 0 to 5
      'стартовый прямоугольник:
      x1= x0
      y1= y0
      x2= xmax
      y2= ymax
```

```
      'выбираем случайный цвет контура:
      clr= GraphicsWindow.GetRandomColor()
      GraphicsWindow.PenColor= clr
      'корректируем координаты верхнего левого угла
прямоугольника:
      while (x1<= xmax) and (y1<= ymax)
        If (x1 > x2) Then
          x= x2
        Else
          x= x1
        EndIf
        If (y1 > y2) Then
          y= y2
        Else
          y= y1
        EndIf
        if (figura="Rect") Then
          'рисуем прямоугольник:
          GraphicsWindow.DrawRectangle(x, y, Math.Abs(x2-x1),
Math.Abs(y2-y1))
        Else
          'рисуем эллипс:
          GraphicsWindow.DrawEllipse(x, y, Math.Abs(x2-x1),
Math.Abs(y2-y1))
        EndIf
        Program.Delay(20)
        'новые координаты вершин прямоугольника:
        x1= x1+i
        y1= y1+j
        x2= x2-i
        y2= y2-j
      EndWhile
    EndFor
  EndFor
EndWhile
```

Правило преобразования фигур очень простое, но образующиеся при работе программы динамические узоры весьма любопытны, и разглядывать их можно не одну минуту (Рис. 21.10 и 21.11)!

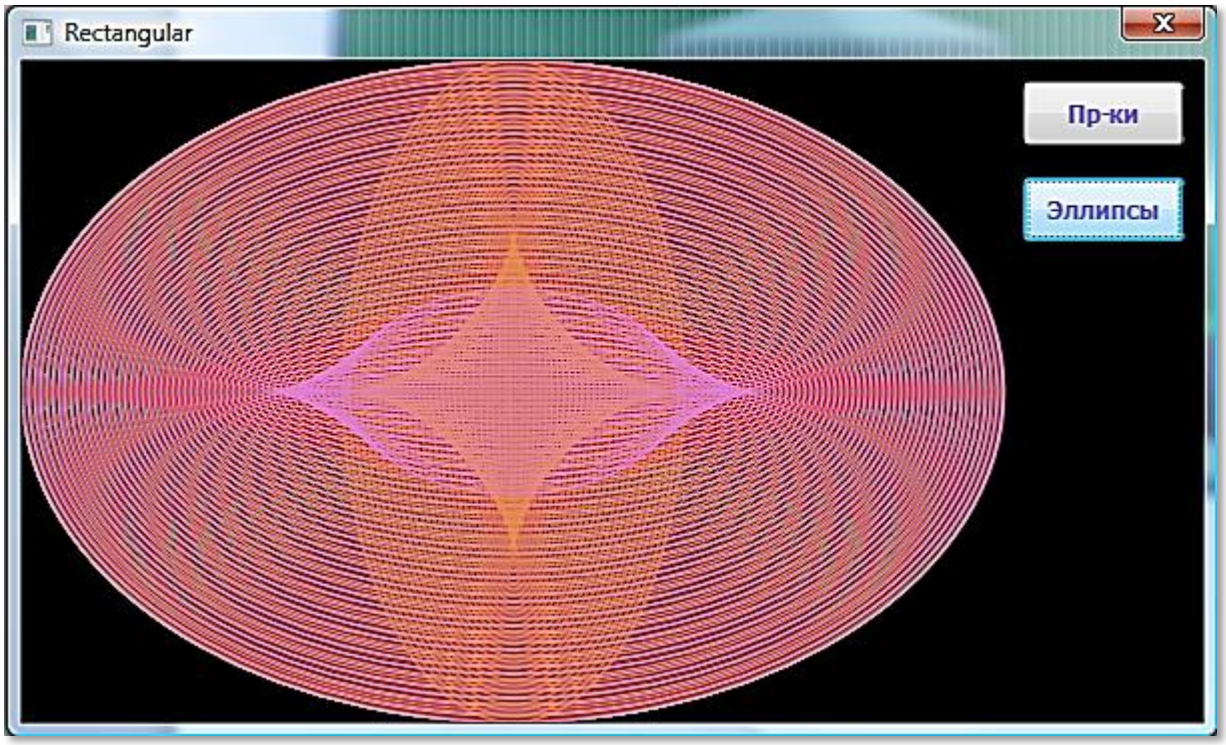


Рис. 21.11. Динамические эллипсы

Исходный код программы находится в папке **Rectangular**.

Урок 22. Черепашья графика

Тише едешь – дальше будешь!

Черепашье кредо

Мы уже научились рисовать картинки с помощью графических примитивов, которые предоставляет нам класс *GraphicsWindow*, - окружностей и эллипсов, прямоугольников и квадратов, треугольников и прямых линий. Их вполне достаточно, чтобы вычертить на канве любую фигуру. Для задания координат характерных точек этих примитивов мы используем прямоугольную систему координат канвы. Однако мы знаем, что в некоторых случаях удобнее перейти к полярным координатам, чтобы вычертить те или иные кривые (мы с ними хорошо познакомились на уроке *Полярная система координат*). Впрочем, нам всё равно пришлось пересчитывать полярные координаты точек кривых в прямоугольные, поскольку координаты пикселей канвы мы не можем задавать иначе. А ведь было бы, наверное, зд***о***рово рисовать кривые *сразу* в полярных координатах!

Впервые такая возможность появилась в языке программирования *Лого* (*Logo*), который был разработан в 1967 году *Сеймуром Пейпертом* и *Идит Харель*. Как и наш *Смолл Бейсик, Лого* создавался как средство для обучения детей дошкольного и младшего школьного возраста основам программирования. Главными объектами языка *Лого* были слова и предложения, что и определило выбор названия для самого языка – *лого* по-гречески значит *слово*. Но известность этому языку программирования принесла маленькая *Черепашка* (*Turtle*), которая умела выполнять несложные команды и вычерчивать линии. Именно благодаря *Черепашке* этот язык стал очень популярным в 70-80-е годы прошлого века, а *Черепашка* появилась и в других языках программирования – *LISP, SCHEME* и многих версиях бейсика. Есть *Черепашка* и в *Смолл Бейсике*, она «спрятана» в классе **Turtle**.

Чтобы увидеть *Черепашку* на экране, достаточно выполнить метод *Turtle.Show* (Рис. 22.1). Она всегда появляется в центре окна и

смотрит вверх. Направление её взгляда знать очень важно, потому что *Черепашка* может ползти только туда, куда глаза глядят!

Если вы хотите, чтобы *Черепашка* поползла в каком-либо направлении, сначала поверните её!

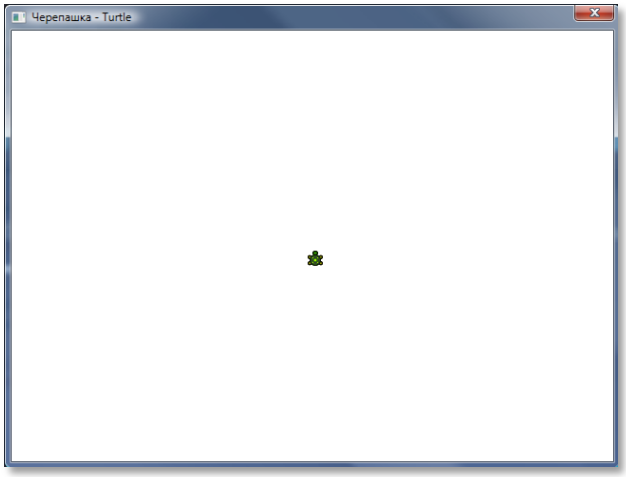


Рис 22.1. «Новорождённая» *Черепашка*

В разных системах программирования *Черепашки* отличаются по форме. Обычно это стилизованные черепахи, но встречаются и примитивные виды *Черепашек* – равносторонние треугольники и наконечники стрелок (Рис. 22.2). Их объединяет желание ползать по экрану и возможность указывать направление их перемещения.

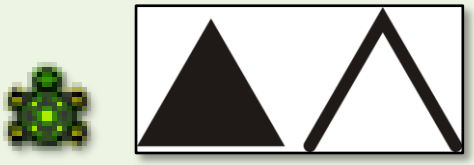


Рис 22.2. *Черепашки* разных видов

Если вы уменьшили размеры окна, то *Черепашка* появится либо у правой нижней границы окна, либо вообще не появится, поэтому лучше устанавливать *Черепашку* в нужном месте окна «принудительно», задавая свойствам *X* и Y *Черепашки* нужные значения координат.

Например, если мы хотим, чтобы она возникла в *центре* окна, то начнём программу с такого кода:

```
GraphicsWindow.Title="Черепашка - Turtle"
GraphicsWindow.Width= 320
GraphicsWindow.Height=320
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"

height=GraphicsWindow.Height
width=GraphicsWindow.Width

'координаты центра окна:
CX= width/2
CY= height/2

'цвет линий:
GraphicsWindow.PenColor= "Black"
GraphicsWindow.PenWidth=1

Turtle.Show()
' устанавливаем Черепашку в центре окна:
Turtle.X = CX
Turtle.Y = CY
```

Имейте в виду, что координаты *Черепашки* отсчитываются в системе координат канвы, то есть начало координат находится в левом верхнем углу, а ось *Y* направлена вниз!

Черепашка может выполнять несколько простых *команд*. Например, попросим её ползти вперед:

```
Turtle.Move(100)
```

Как и было обещано, *Черепашка* выполнит команду и проползёт *вперёд* (то есть *вверх*, потому что она туда смотрит!) 100 пикселей (Рис. 23.3).

А всё-таки наша *Черепашка* умеет двигаться и в противоположном взгляду направлении! Дайте ей команду

```
Turtle.Move(-100),
```

и она, смотря вперёд, попятится назад (Рис. 22.4).

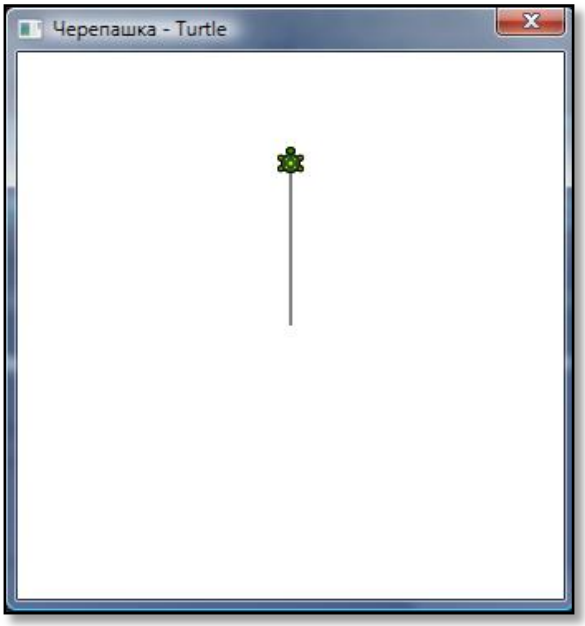


Рис. 22.3. Полный вперёд!

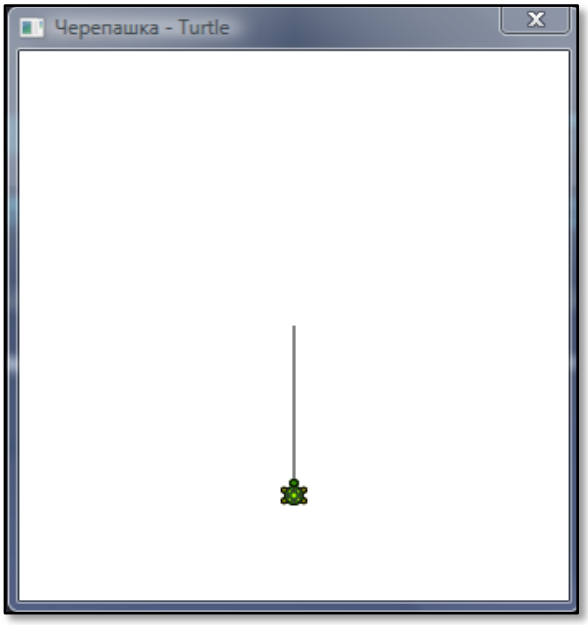


Рис. 22.4. Полный назад!

За ней остался след - тонкая прямая **чёрная** линия. Это всё потому, что наша *Черепашка* не простая, а вооруженная – карандашом! Вы можете дать *Черепашке* карандаш любого цвета, а также с грифелем большего диаметра, чтобы линии стали **толще**. Правда, для этого вам придётся обратиться к свойствам *Графического окна*, которое и выдаст нашей *Черепашке* нужный *карандаш*:

```
GraphicsWindow.PenColor= "Red"
GraphicsWindow.PenWidth=10
```

Теперь *Черепашка* проведет толстую **красную** линию (Рис. 22.5).

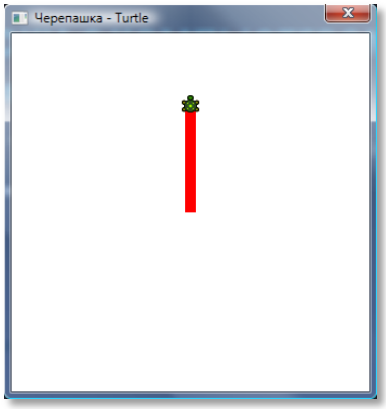


Рис. 22.5. *Черепашка* бороздит просторы *графического окна*

Но *Черепашка* при ползании *не всегда* прочерчивает линию. Дело в том, что по умолчанию карандаш находится в боевом положении, поэтому от него остаётся след. За положение карандаша отвечают методы *Черепашки PenDown* и *PenUp*. Первый *опускает* карандаш, а второй – *поднимает* его. Соответственно этому *Черепашка* либо чертит линию, либо просто перемещается в новое положение.

Немного исправим программу так, чтобы *Черепашка* чертила *пунктирную* линию (Рис. 22.6).

```
GraphicsWindow.PenColor= "Green"
GraphicsWindow.PenWidth= 4

For i= 1 to 10
  if (Math.Remainder(i,2)= 0) then
    Turtle.PenUp()
  Else
  Turtle.PenDown()
    EndIf
  Turtle.Move(10)
endFor
```

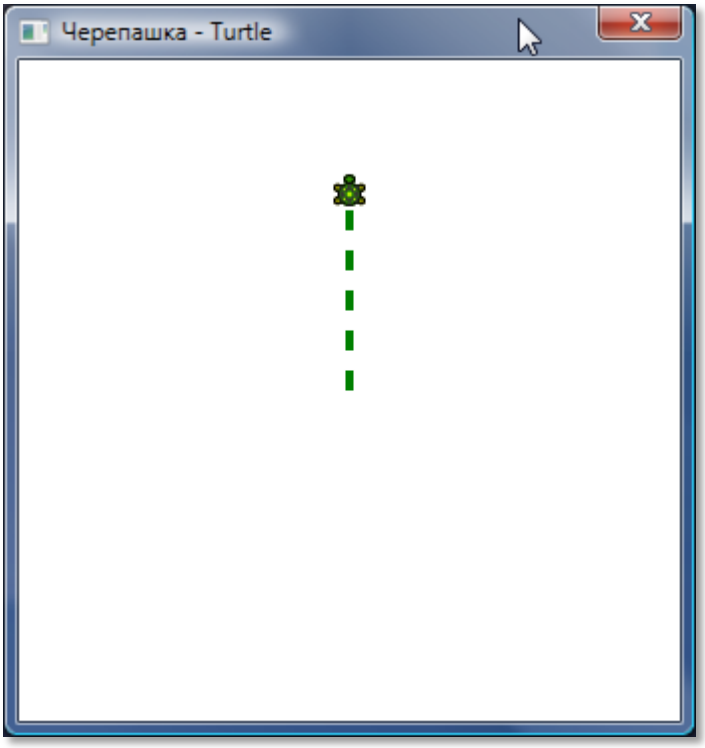


Рис. 22.6. *Черепашка* оставляет за собой следы

Иногда бывает полезно *спрятать Черепашку* после того как она выполнила свою работу, что часто бывает и в жизни: исполнители не должны заслонять руководящие кадры. Вызвав команду

```
Turtle.Hide(),
```

мы превратим *Черепашку* в *Черепашку-невидимку*. И что особенно приятно, она при этом не утратит способности вычерчивать линии (Рис. 22.7).

Несмотря на то, что *Черепашка*, как это у них, у черепашек, и водится, довольно медлительна, но, задав свойству *Turtle.Speed* значение десять или близкое к десяти, мы слегка приободрим её. Вы можете выбирать произвольные значения, но у *Черепашки* всего 10 скоростей – от 1 до 10.

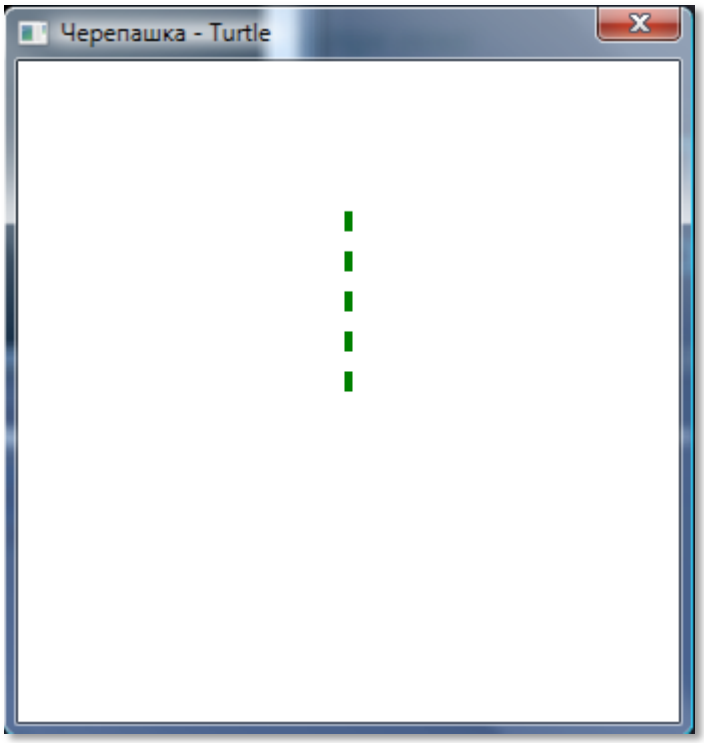


Рис. 22.7. *Черепашка* спряталась

Наша *Черепашка* может перемещаться в указанную точку канвы, то есть попросту чертить линию от точки, в которой она находится, до любой другой. Для этого при вызове метода

```
Turtle.MoveTo(x,y)
```

следует указать прямоугольные координаты новой точки.

После установки *Черепашки* в центре окна давайте-ка погоняем её по канве:

```
Turtle.X = CX
Turtle.Y = CY
Turtle.MoveTo(300,300)
Turtle.MoveTo(20,300)
```

Очень забавно наблюдать, как *Черепашка* самостоятельно поворачивается так, чтобы её глаза были направлены в заданную точку (Рис. 22.8). Значит, она не вслепую перемещается по канве, а именно так, как мы и договаривались в начале урока!

Таким образом, *Черепашка* неплохо ориентируется и в прямоугольной системе координат. Однако от неё было бы не много пользы, если бы она просто перемещалась от одной точки к другой в этой системе координат.

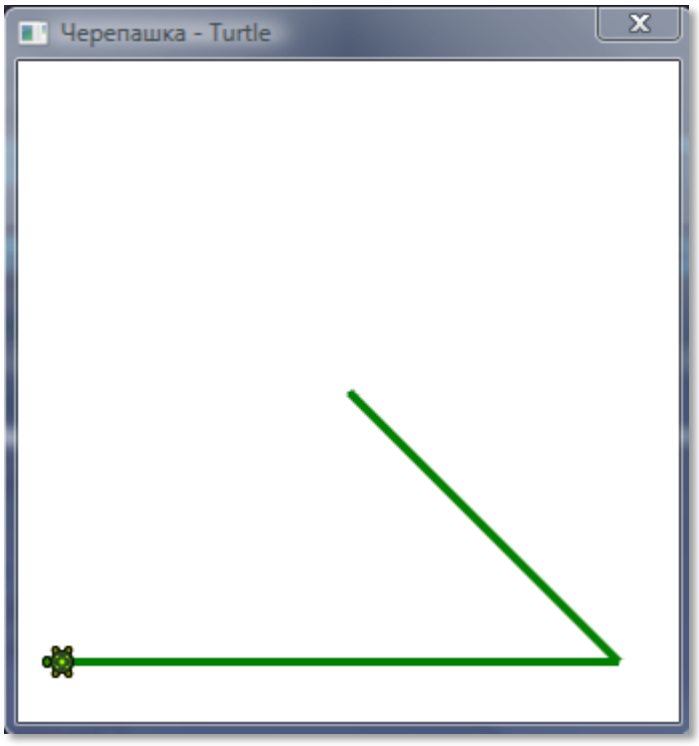


Рис. 22.8. *Черепашка* выполняет команды *MoveTo*

Главное достоинство *Черепашки* в том, что она может передвигаться и в *полярной* системе координат. Действительно, при за-

пуске программы мы можем установить *Черепашку* в центре окна. Будем считать, что это *полюс* полярной системы координат.

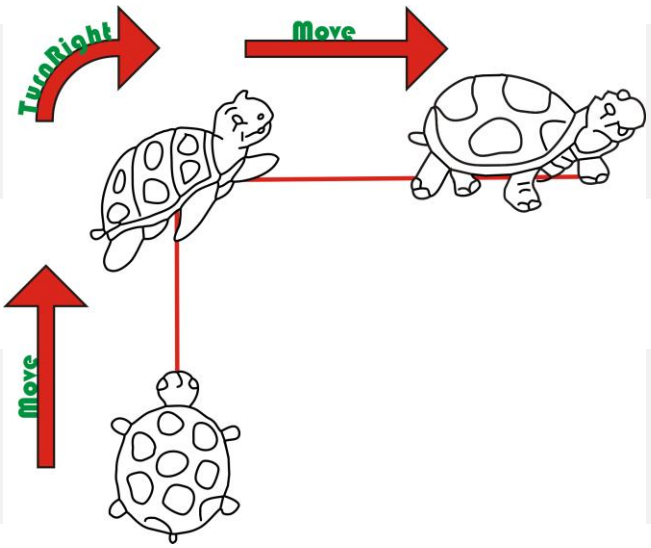
Три метода класса *Turtle*

```
  Turtle.TurnLeft()
  Turtle.TurnRight()
  Turtle.Turn(угол)
```

поворачивают *Черепашку* вокруг собственной оси. Первый метод – на 90 градусов *против* часовой стрелки, второй – на 90 градусов *по* часовой стрелке, третий – на произвольный угол. В последнем случае *Черепашка* повернётся против часовой стрелки, если угол отрицательный, и по часовой стрелке – если положительный.

Например, мы легко заставим нашу *Черепашку* танцевать брейк:

```
For i= 1 to 10
  Turtle.TurnLeft()
  Turtle.TurnRight()
endFor
```



Или так:

```
For i= 1 to 10
  Turtle.Turn(300)
  Turtle.Turn(-320)
endFor
```

С помощью этих методов мы легко сможем переместить *Черепашку* в любую точку, заданную в полярных координатах. Например, выбрав полярный угол и полярный радиус

```
Z=45
R=120
Turtle.Turn(Z)
Turtle.Move(R),
```

мы отправим *Черепашку* в точку, координаты которой заданы в полярной системе координат (Рис. 22.9).

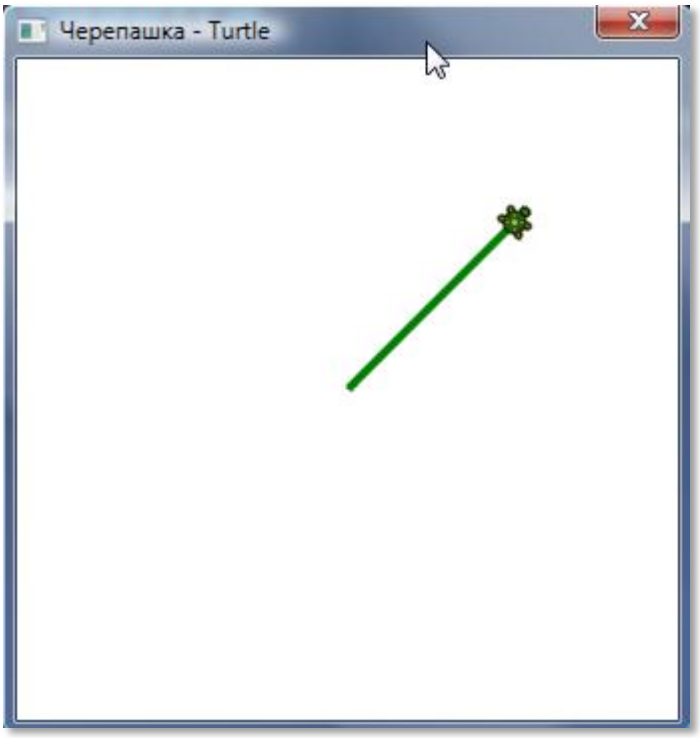


Рис. 22.9. Полярная *Черепашка*

Всякий раз возвращая *Черепашку* в начало координат и поворачивая её головой вверх, мы можем наставить сколько угодно точек в полярной системе координат:

```
R=120
Z=0
Turtle.Turn(Z)
For i= 1 to 8
  Turtle.PenUp()
  Turtle.Move(R-4)
  Turtle.PenDown()
  Turtle.Move(4)
  Turtle.PenUp()
  Turtle.MoveTo(CX,CY)
  Turtle.Angle = Z
  Turtle.Turn(360/8* i)
endFor
```

Например, мы можем расставить точки в вершинах правильного восьмиугольника (Рис. 22.10).

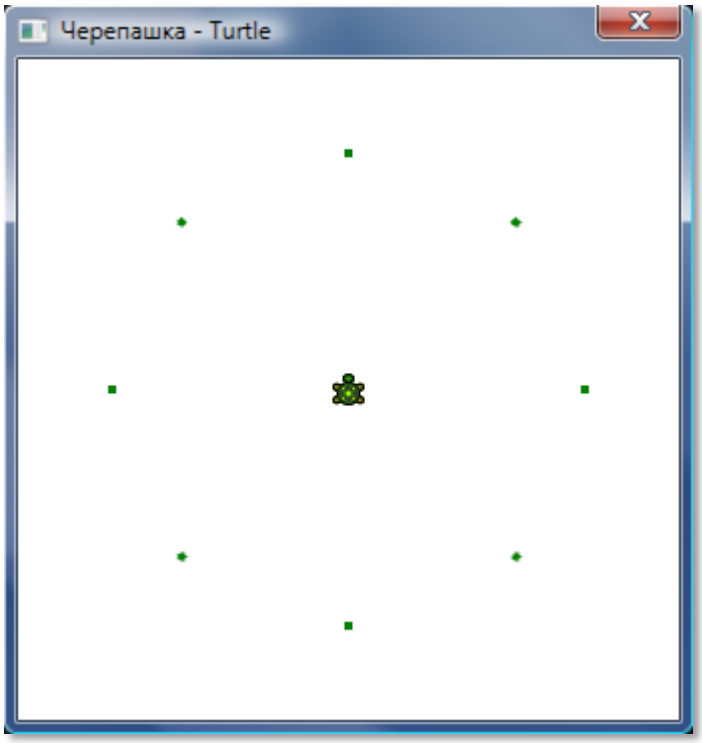


Рис. 22.10. *Черепашка* ставит точки в полярной системе координат

А что у нас получится, если мы продолжим перемещения *Черепашки*, не возвращая её в прежнее положение? – Тогда начало полярных координат переместится в точку, где находится *Черепашка*, а полярная ось повернётся по направлению взгляда *Черепашки*. Следующий поворот будет отсчитываться уже не от горизонтальной оси, а от *нового* положения полярной оси. Мы знаем, что все внутренние углы правильного треугольника равны 60 градусов, а внешние – 120, поэтому мы легко построим правильный треугольник (Рис. 22.11), если вычертим три его стороны, начиная каждую из конца предыдущей стороны. Так как длина всех сторон одинакова, то достаточно повернуть *Черепашку* на нужный угол:

```
R=120
Turtle.TurnRight()
For i= 1 to 3
  Turtle.Turn(-120)
  Turtle.Move(R)
endFor
```

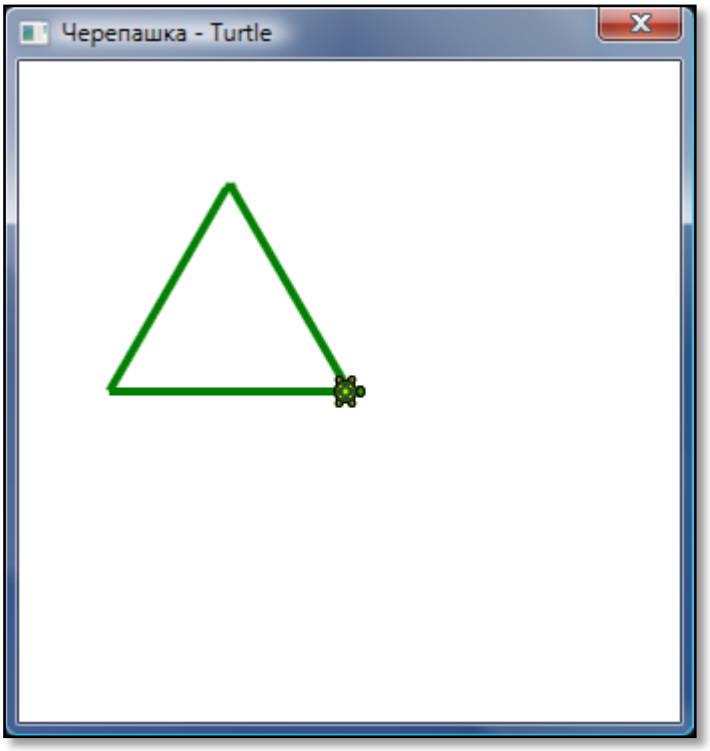


Рис. 22.11. *Черепашка* построила правильный треугольник

Мы сделаем формулу более универсальной, если введём параметр *n*, равный числу сторон правильного многоугольника:

```
Turtle.X = width-100
Turtle.Y = height-30
R=120
Turtle.TurnRight()
n=6
For i= 1 to n
  Turtle.Turn(-360/n)
  Turtle.Move(R)
endFor
```

Для построения *любого* правильного многоугольника (Рис. 22.12) нам потребовалось написать всего несколько строк кода и при этом – никаких расчётов координат вершин!

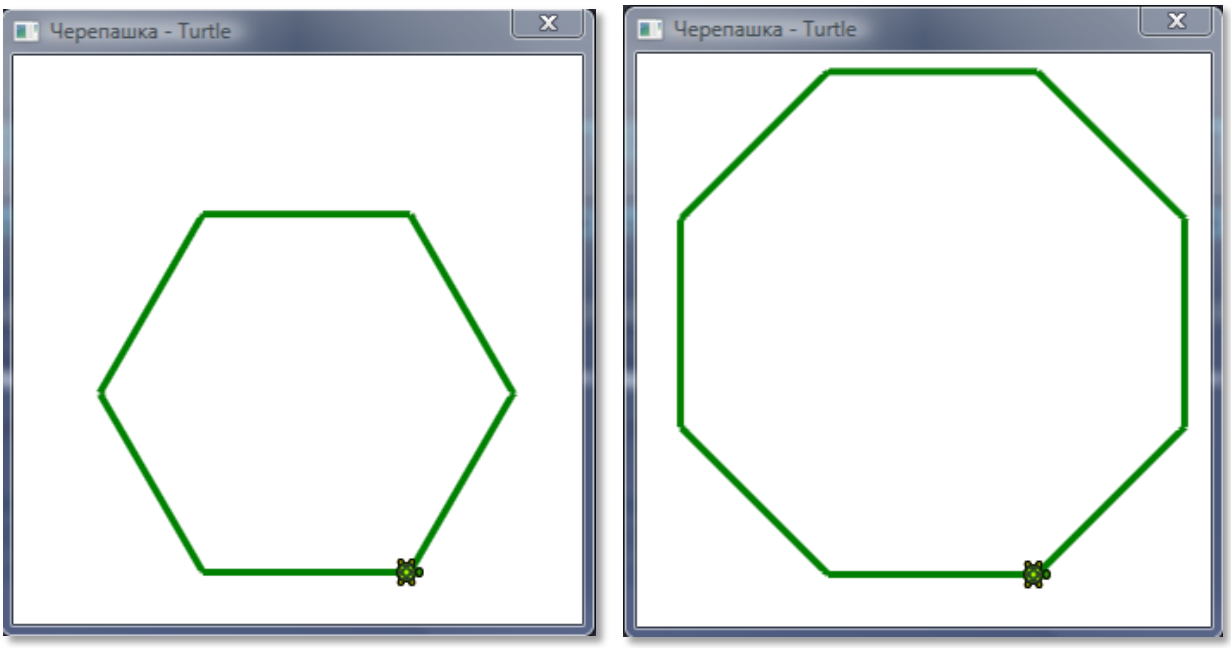


Рис. 22.12. *Черепашка* построила правильные многоугольники

Нетрудно догадаться, что если задать достаточно большое число вершин, то многоугольник превратится в *окружность* (Рис. 22.13).

```
Turtle.X = width-160
Turtle.Y = height-20
R=12
Turtle.TurnRight()
n=80
For i= 1 to n
  Turtle.Turn(-360/n)
  Turtle.Move(R)
endFor
```

Как это ни удивительно, но *Черепашке* нетрудно вычертить и *спираль* (Рис. 22.14).

```
'спираль:
Turtle.X = CX
Turtle.Y = CY
R=1
Turtle.TurnRight()
For i= 1 to 240
  Turtle.Move(R)
  Turtle.Turn(10)
```

```
  R = R + 0.1
endFor
```

Если при повороте *Черепашки* задать *отрицательный* угол

```
Turtle.Turn(-10),
```

то вместо правой спирали мы получим *левую* (она закручена в противоположном направлении).

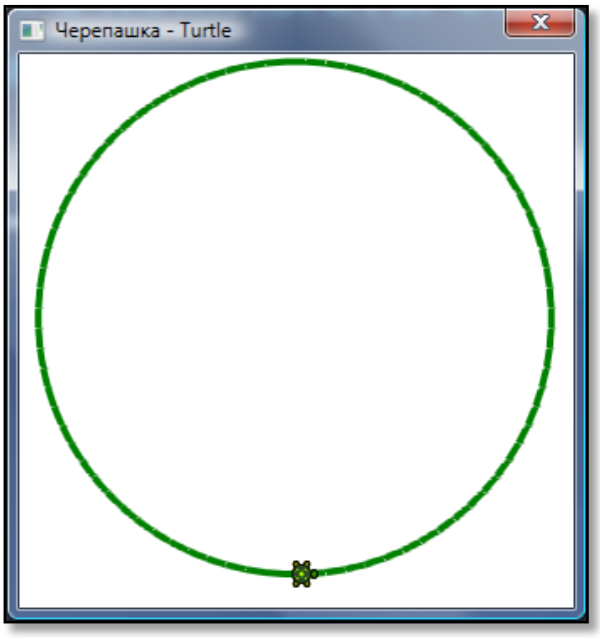


Рис. 22.13. Черепашка чертит окружность

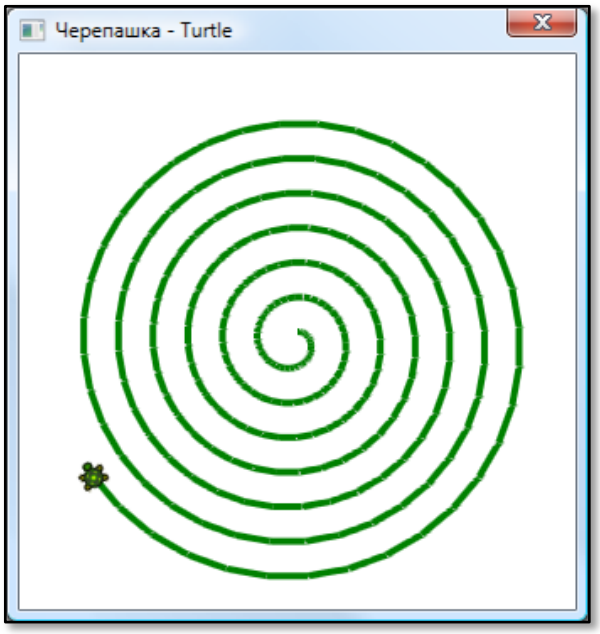


Рис. 22.14. Черепашка рисует спираль

А теперь давайте перейдём к более сложным кривым, которые наверняка убедят вас, что вы не напрасно познакомились с *Черепашкой.*

Для начала мы заставим многоугольники *вращаться* вокруг одной из своих вершин:

```
'Вращающеся фигуры
Sub polyRoll
  For i= 1 to 12
    polyStop()
    Turtle.Turn(angle2)
    endFor
EndSub
Sub polyStop
```

```
  ps_next:
  Turtle.Move(side)
  Turtle.Turn(angle1)
  _turn= _turn+angle1
  If (Math.Remainder(_turn, 360) <> 0) Then
    Goto ps_next
  EndIf
EndSub
```

Изменяя значения переменных *angle1* и *angle2*, мы получим разные многоугольники, вращающиеся вокруг центра окна (Рис. 22.15).

```
Turtle.X = CX
Turtle.Y = CY
GraphicsWindow.PenColor= "Black"
GraphicsWindow.PenWidth= 2
_turn=0
angle1=90
angle2=30
side=100

angle1=60
angle2=45
side=80
polyRoll()
TextWindow.Read()
```

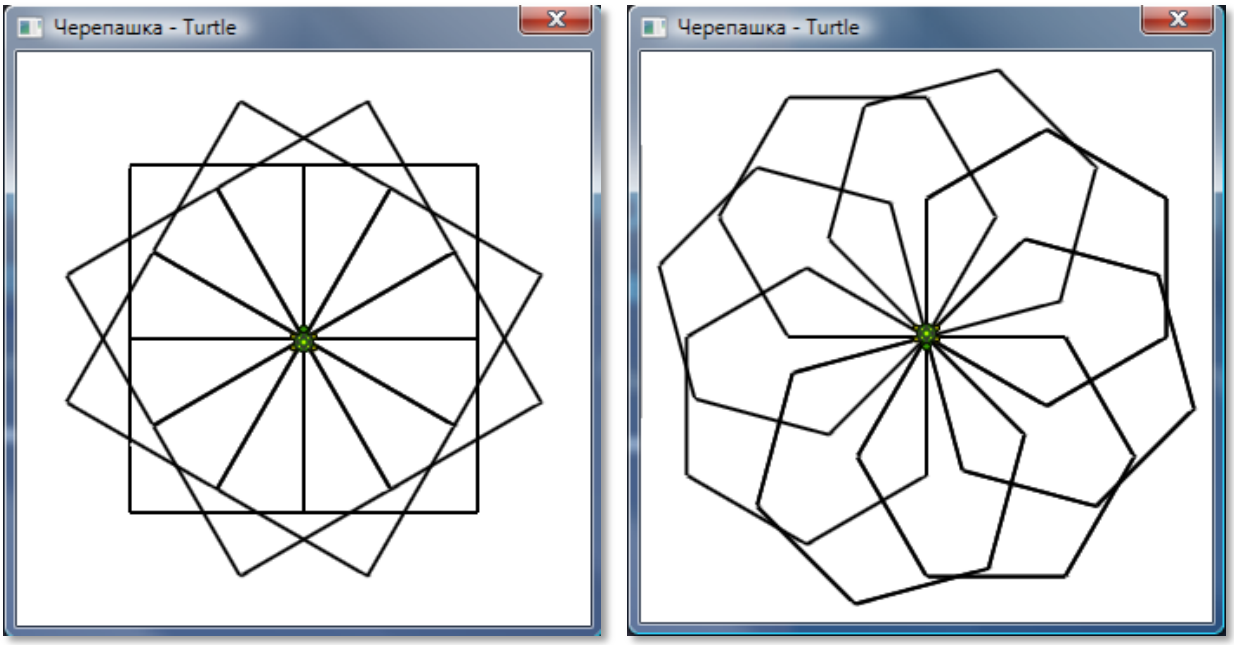


Рис. 22.15. Вращающиеся фигуры

Необыкновенно простая рекурсивная процедура *polySpi* при разных параметрах даёт потрясающее многообразие великолепных спиралей (Рис. 22.16)!

```
'Рекурсивные спирали
Sub polySpi
  If (side > maxside) Then
    goto ps_exit
  EndIf
  Turtle.Move(side)
  Turtle.Turn(angle)
  side = side + inc
  polySpi()
ps_exit:
EndSub
Turtle.X = CX -30
Turtle.Y = CY +30
GraphicsWindow.PenColor= "Black"
GraphicsWindow.PenWidth= 2
maxside=240
inc=1
side=60
angle=95
TextWindow.Read()
'квадратная спираль:
inc= 3
angle=90
angle=87
polySpi()
TextWindow.Read()
'треугольная спираль:
inc= 5
angle=120
Turtle.X = CX -20
Turtle.Y = CY +20
inc= 4
angle=117
maxside=280
TextWindow.Read()
inc= 1
angle=30
side= 3
polySpi()
TextWindow.Read()
```

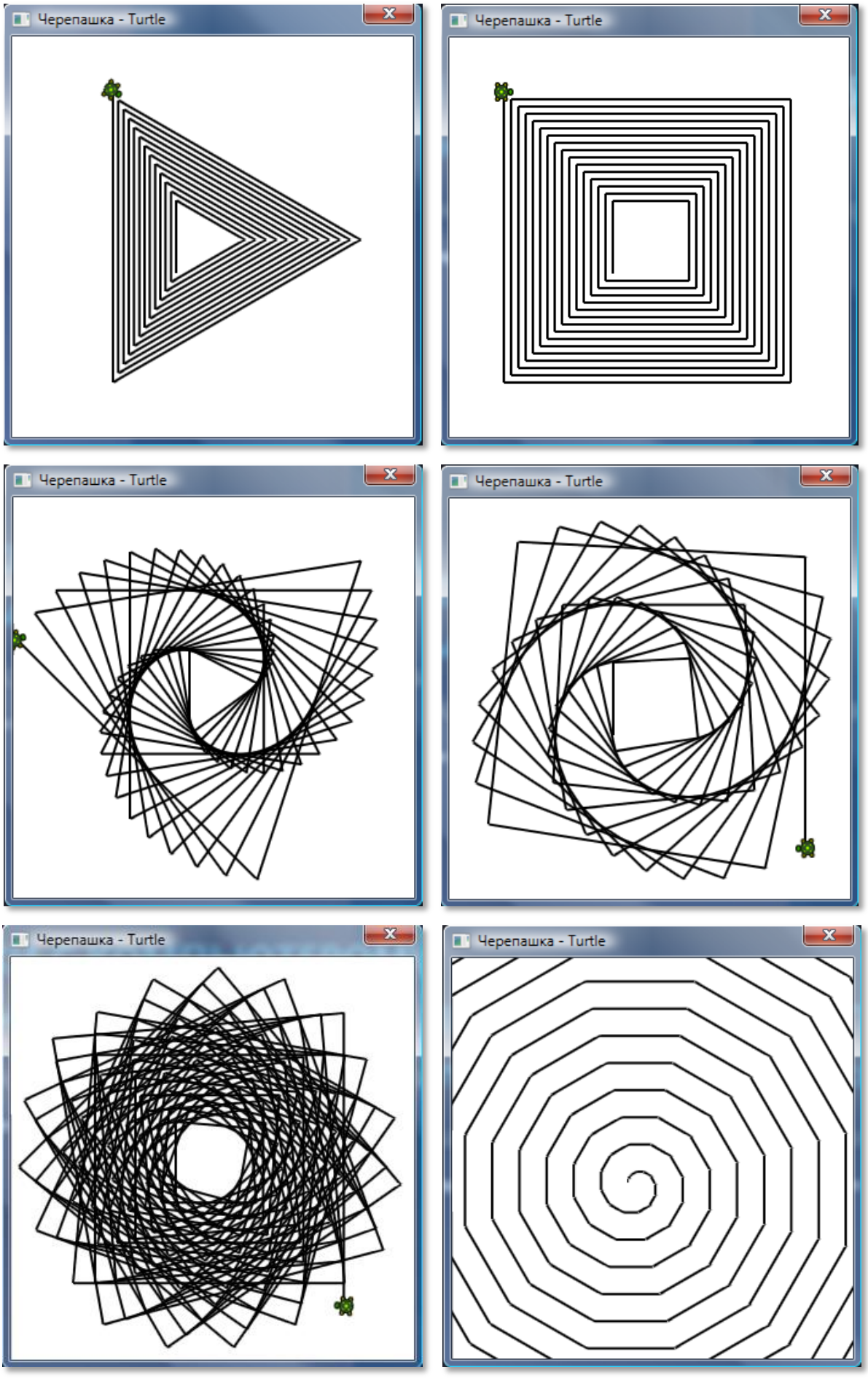


Рис. 22.16. Рекурсивные спирали

И последний проект этого урока – вычерчивание разнообразных *звёздчатых многоугольников* (Рис. 22.17).

```
'Звездчатые многоугольники
Sub newPoly
  For i= 1 to 24
    Turtle.Move(side)
    Turtle.Turn(angle)
    Turtle.Move(side)
    Turtle.Turn(2*angle)
  endFor
EndSub

Turtle.X = CX-30
Turtle.Y = CY-20
GraphicsWindow.PenColor= "Black"
GraphicsWindow.PenWidth= 2
side=80
angle=30
' звезда:
angle=144
newPoly()
TextWindow.Read()
GraphicsWindow.Clear()
'angle=45

' зубчатое колесо:
Turtle.X = CX-80
Turtle.Y = CY-100
Turtle.Angle=0
side=36
angle=125
newPoly()
TextWindow.Read()
GraphicsWindow.Clear()
```

Количество зубцов определяется углом *angle*.

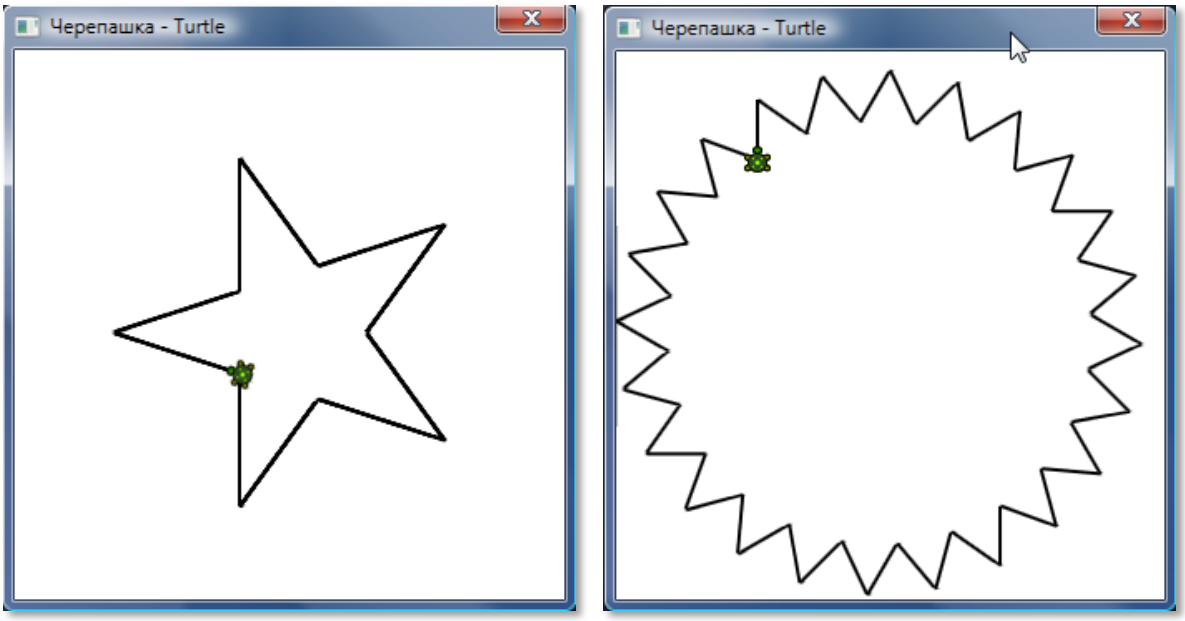


Рис. 22.17. Звёздчатые многоугольники

Исходный код программы находится в папке **Turtle**.

Напишите такие программы для *Черепашки*, чтобы она построила следующие фигуры (Рис. 22.18).

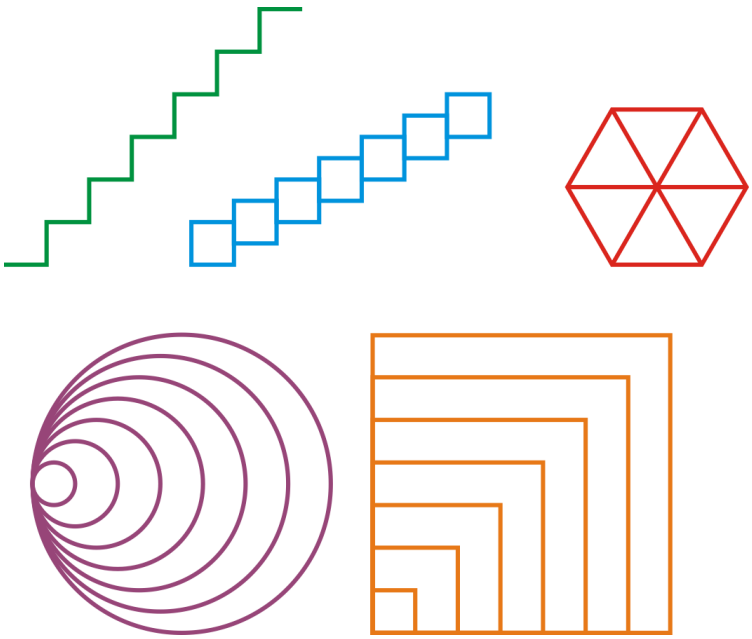


Рис. 22.18. Черепашьи фигуры

Урок 23. Фрактальная Киберчерепашка

На уроке *Черепашья графика* мы заставляли *Черепашку* выполнять наши команды, записанные на языке *Смолл Бейсик*. Но ведь в каждом живом существе, в его геноме записана информация о том, как оно должно расти и развиваться, а также инстинкты, управляющие поведением достаточно примитивных организмов. *Черепашка* также может выполнять не только внешнюю программу, но и внутреннюю – записанную в её геноме и реализуемую через инстинкты. Это урок мы и посвятим моделированию такой *кибернетической Черепашки*.

Начнём новый проект **Cyber-Turtle** с объявления переменных.

Черепашка рождается в условленном месте, обозначенном координатами *x0* и *y0*, а ее голова направлена в сторону света, заданную начальным углом *a0*:

```
'ПРОГРАММА ДЛЯ МОДЕЛИРОВАНИЯ
'КИБЕРЧЕРЕПАШКИ

'var
'начальное положение Черепашки:
x0=0
y0=0
a0=0
```

Основное предназначение *Черепашки* (судьба) - перемещение по плоскости и вычерчивание линий, поэтому нам необходимо задать её основные *свойства*:

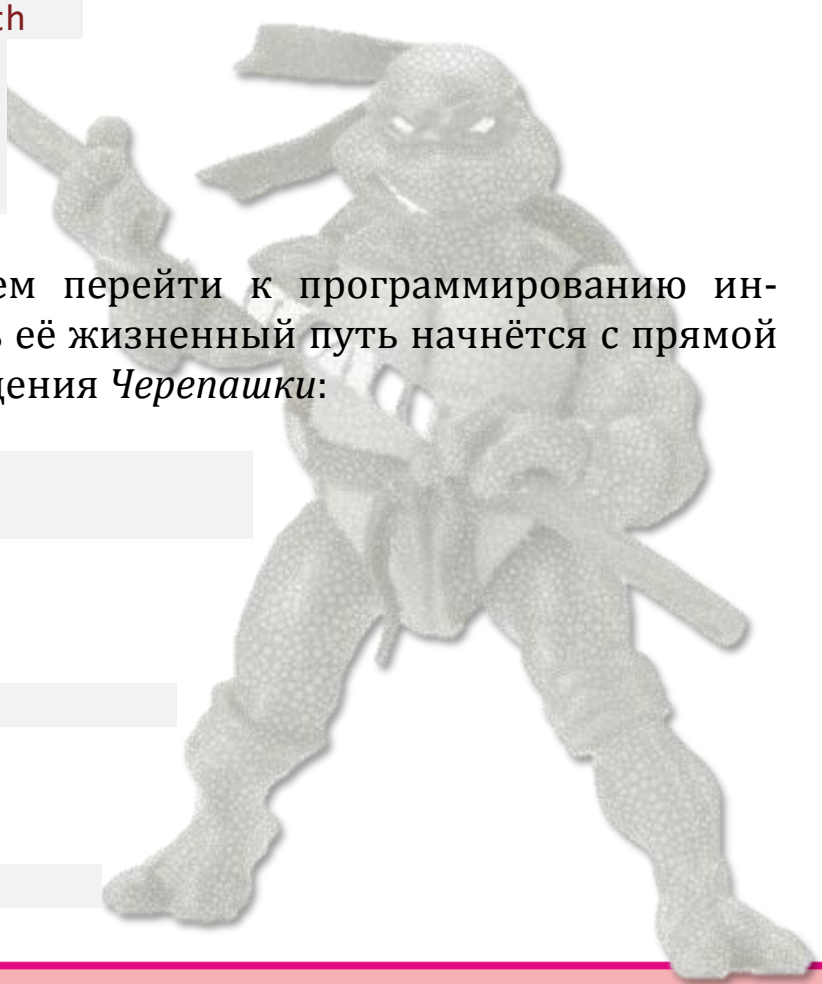
```
'длина единичного перемещения Черепашки:
size=0
'угол единичного поворота Черепашки:
teta=0
'скорость движения Черепашки:
speed= 9
'цвет линии:
penColor="Red"
'толщина линии:
```

```
penWidth=2
'список команд, которые должна выполнить Черепашка:
instinct=""
```

Здесь всё просто и понятно, кроме, пожалуй, *инстинкта*, который и определяет поведение *Черепашки*. Дальше мы зададим ей жизненную программу (инстинкт), и вы сможете сами программировать *Черепашку*.

Настройка *окна* приложения не должна вызвать у вас вопросов:

```
GraphicsWindow.Title="КиберЧерепашка - CyberTurtle"

GraphicsWindow.Width= 320
GraphicsWindow.Height=320
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"

height=GraphicsWindow.Height
width=GraphicsWindow.Width

'координаты центра окна:
CX= width/2
CY= height/2
```



Теперь мы вполне можем перейти к программированию инстинкта *Черепашки*. Пусть её жизненный путь начнётся с прямой линии. Задаём место рождения *Черепашки*:

```
x0= CX - 100
y0= CY + 100
```

Её направление:

```
a0= 0
```

Длину «шага»:

```
size=100
```

И самое главное – инстинкт:

```
'прямая линия:
instinct="F"
```

Процесс рождения *Черепашки* не зависит от самой *Черепашки*, а полностью лежит на совести её родителей, то есть на нас с вами:

```
'Черепашка занимает исходное положение
Sub start
  Turtle.Speed= speed
  GraphicsWindow.PenColor= penColor
  GraphicsWindow.PenWidth= penWidth
  Turtle.PenDown()
  Turtle.X= x0
  Turtle.Y= y0
  Turtle.Angle= a0
  Turtle.Show()
EndSub
```

В большинстве языков программирования, поддерживающих черепашью графику, движение *Черепашки* вперёд обозначается командой *Forward*. Первую букву этого слова мы будем использовать для той же цели. После того как *Черепашка* заняла исходное положение, она должна выполнить те действия, которые ей предписывает инстинкт. Для этого у неё имеется мозг, который умеет обрабатывать и исполнять команды инстинкта:

```
'Черепашка выполняет инстинкт
Sub execute
  'Черепашка считывает команды:
  For i= 1 To Text.GetLength(instinct)
    'очередная команда:
    cmd= Text.GetSubText(instinct,i,1)
   'двигаться вперед на один шаг:
    If (cmd="F") Then
      Turtle.Move(size)
    EndIf
  EndFor
EndSub
```

Пока *Черепашка* маленькая, она может только двигаться вперёд, но и этого вполне достаточно, чтобы дать ей путёвку в жизнь:

```
start()
execute()
```

Запускаем программу, и *Черепашка* проводит вертикальный отрезок длиной в *size* пикселей (Рис. 23.1).

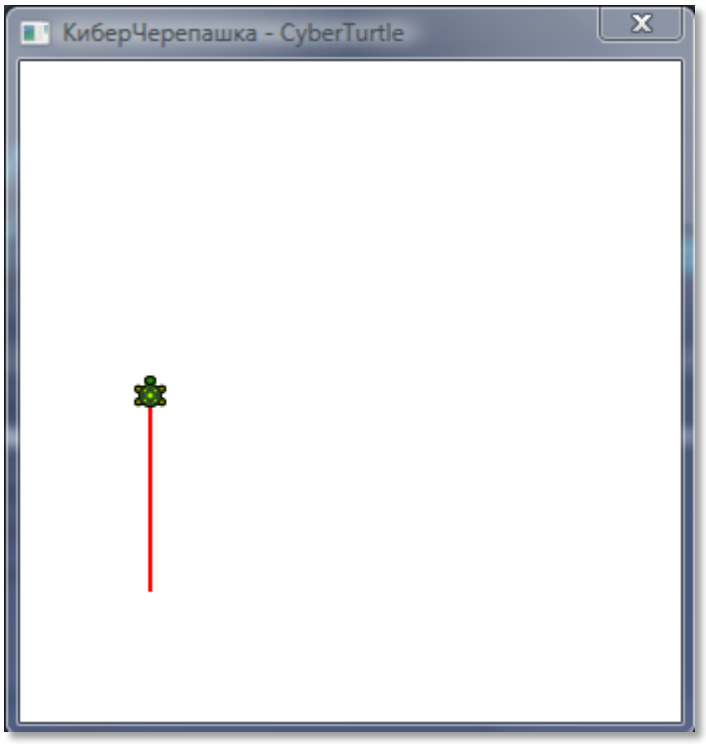


Рис. 23.1. *Черепашка* в ползунковом возрасте

На следующем этапе своей черепашьей жизни наша *Киберчерепашка* сможет выполнять команды + и -, то есть поворачиваться направо и налево на заданный угол *teta*. В её окрепшем мозгу появятся новые шишки для обработки более сложного инстинкта:

```
    If (cmd="+") Then
      Turtle.Turn(teta)
    EndIf
    If (cmd="-") Then
      Turtle.Turn(-teta)
    EndIf
```

Треугольник мы уже вычерчивали с помощью обычной *Черепашки*. Как вы помните, для этого *Черепашка* должна вычертить одну сторону, повернуться на угол 120 градусов, вычертить вторую сторону, опять повернуться на 120 градусов и, наконец, вычертить третью сторону. Отсюда следует, что угол единичного поворота равен 120 градусам, а инстинкт запишется строкой:

```
'треугольник:
teta=120
instinct="F+F+F+"
```

Для проверки нашей гипотезы запускаем программу. *Черепашка* нас не подвела – треугольник удался на славу (Рис. 23.2)!

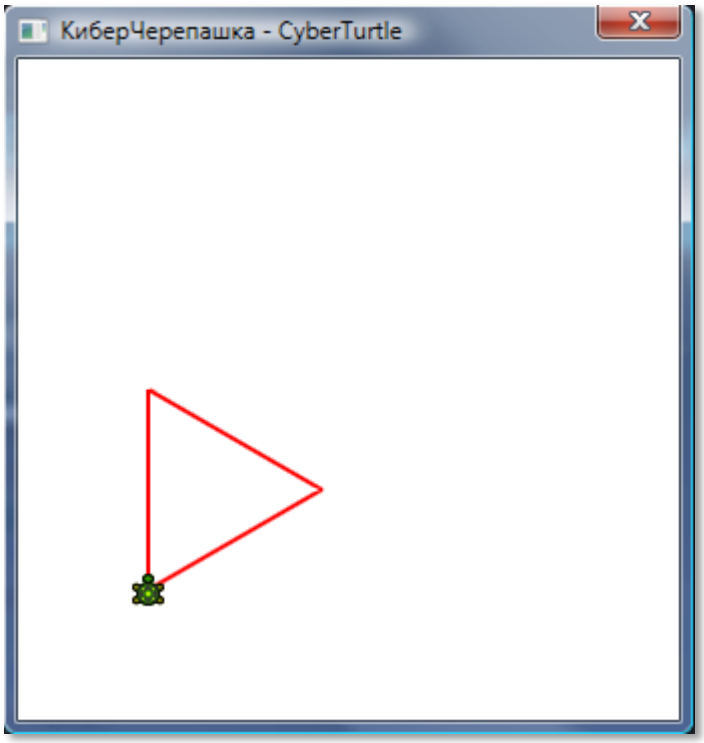


Рис. 23.2. Черепашковый треугольник

Совершенно аналогично мы можем сформировать инстинкты для вычерчивания любых правильных *многоугольников:*

```
'квадрат:
teta=90
instinct="F+F+F+F+"
'пятиугольник:
teta=72
```

```
instinct="F+F+F+F+F+"
'шестиугольник:
teta=60
instinct="F+F+F+F+F+F+"
```

Как вы видите, инстинкт становится всё более длинным, при этом одна и та же пара команд *F+* просто повторяется *n* раз, если через *n* обозначить число вершин правильного многоугольника. Давайте научимся выращивать инстинкты!

Пусть при рождении *Черепашка* получает простейший инстинкт, который принято называть *аксиомой*, а затем, с каждым годом инстинкт развивается и становится всё более сложным благодаря *правилам* роста. Для треугольника и других правильных многоугольников аксиома самая простая:

```
axiom="F"
```

Она означает, что *Черепашка* должна двигаться вперёд на расстояние *size*. Наша первая *Черепашка* только это и умеет делать.

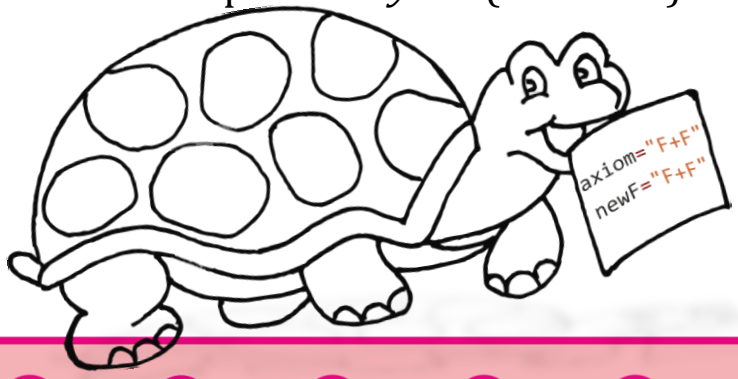
Поскольку все многоугольники строятся по одному и тому же сценарию, то и правило для них одно и то же:

```
newF="F+F"
```

А действует оно так. Через год (возраст *Черепашки* определяется переменной *iter*) каждая команда *F*, в соответствии с правилом роста, заменяется значением переменной *newF*, то есть *F+F*:

```
iter= 1 → axiom="F+F"
```

Это значит, что через год (а у *Черепашек* год короткий!) наша *Черепашка* научится вычерчивать *угол* (Рис. 23.3).



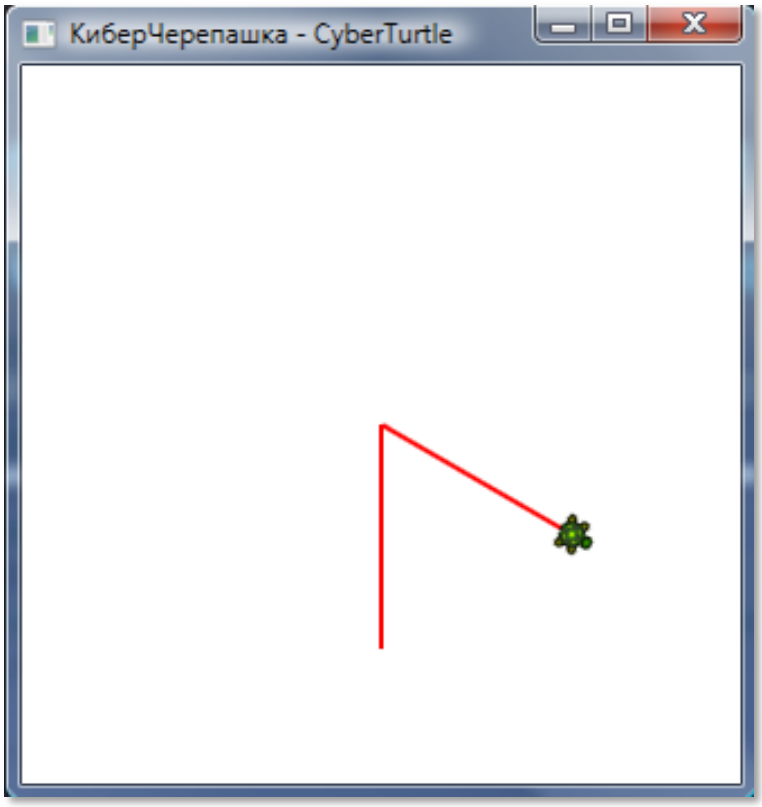


Рис. 23.3. Молодая *Черепашка* - «угловатая»

На второй год с поведение *Черепашки* усложнится:

iter= 2 → `axiom="(F)+(F)"` → `"(F+F)+(F+F)"` → `"F+F+F+F"`

Опять в аксиоме каждая буква *F* заменяется выражением *F+F*. Для удобства преобразований мы воспользовались скобками, но самой *Черепашке* они не нужны.

Вот теперь *Черепашка* может вычертить настоящий *треугольник* (Рис. 23.4).

Вы, должно быть, заметили, что для построения треугольника достаточно инстинкта `"F+F+F+"`, то есть одну сторону *Черепашка* чертит дважды. Увы, все инстинкты несовершенны, но зато достаточно просты.

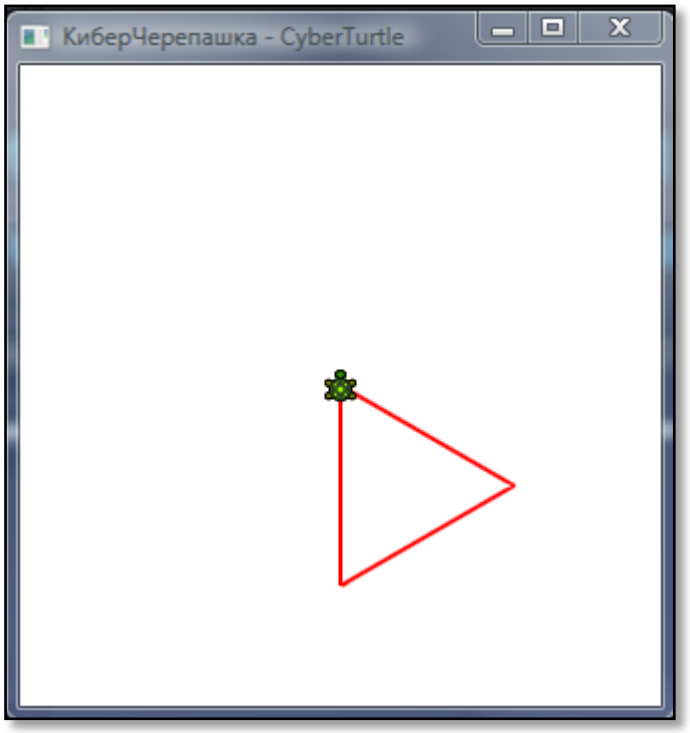


Рис. 23.4. Двухлетняя *Черепашка* умеет чертить треугольник

Инстинкт *Черепашки* формируется, конечно, не вручную - она имеет для этого в мозгу специальную процедуру:

```
'Черепашка формирует инстинкт
Sub createInstinct
  For i= 1 to iter
    instinct=""
    For j= 1 To Text.GetLength(axiom)
      'очередная команда аксиомы:
      cmd= Text.GetSubText(axiom,j,1)
      If (cmd="F") Then
        instinct= instinct + newF
      Else
        instinct= instinct + cmd
      EndIf
    endFor
    axiom=instinct
  endFor
endSub
```

Её действие мы уже разобрали. Каждая буква *F* заменяется значением переменной *newF*, а все остальные символы (команды поворота + и -) переходят в новый инстинкт без изменения. Так

как мы привыкли использовать при моделировании поведения *Черепашки* переменную *instinct*, то результатом работы процедуры *createInstinct* пусть будет значение этой переменной, хотя мы могли бы заменить её и переменной *axiom*.

Выделим «треугольный» инстинкт в отдельную процедуру

```
'Треугольник:
Sub Triangle
  teta=120
  size=100
  speed=5
  iter= 2
  axiom="F"
  newF="F+F"
endSub
```

и запустим программу:

```
Triangle()
createInstinct()
start()
execute()
```

Мы научились формировать простейший инстинкт, повинуясь которому *Черепашка* рисует треугольник. Достаточно изменить значение угла *teta,* и новый инстинкт – для построения *квадрата* – готов:

```
'Квадрат:
Sub Quadrat
  teta=90
  size=100
  iter= 2
  speed=10
  axiom="F"
  newF="F+F"
endSub
```

На третий год жизни *Черепашка* сумеет начертить пятиугольник, шестиугольник и восьмиугольник (Рис. 23.5).

```
'Пятиугольник:
Sub Pentagon
  teta=72
  size=100
  iter= 3
  speed=10
  axiom="F"
  newF="F+F"
endSub

'Шестиугольник:
Sub Hexagon
  teta=60
  size=100
  iter= 3
  speed=10
  axiom="F"
  newF="F+F"
endSub

'Восьмиугольник:
Sub Octagon
  teta=45
  size=100
  iter= 3
  speed=10
  axiom="F"
  newF="F+F"
endSub
```

Мы могли бы продолжать этот процесс и дальше, но совершенно понятно, что с годами *Черепашка* сможет выстраивать любые правильные многоугольники, поэтому давайте усложним поведение *Черепашки* с помощью более изощрённых инстинктов и научим её чертить *фракталы*.

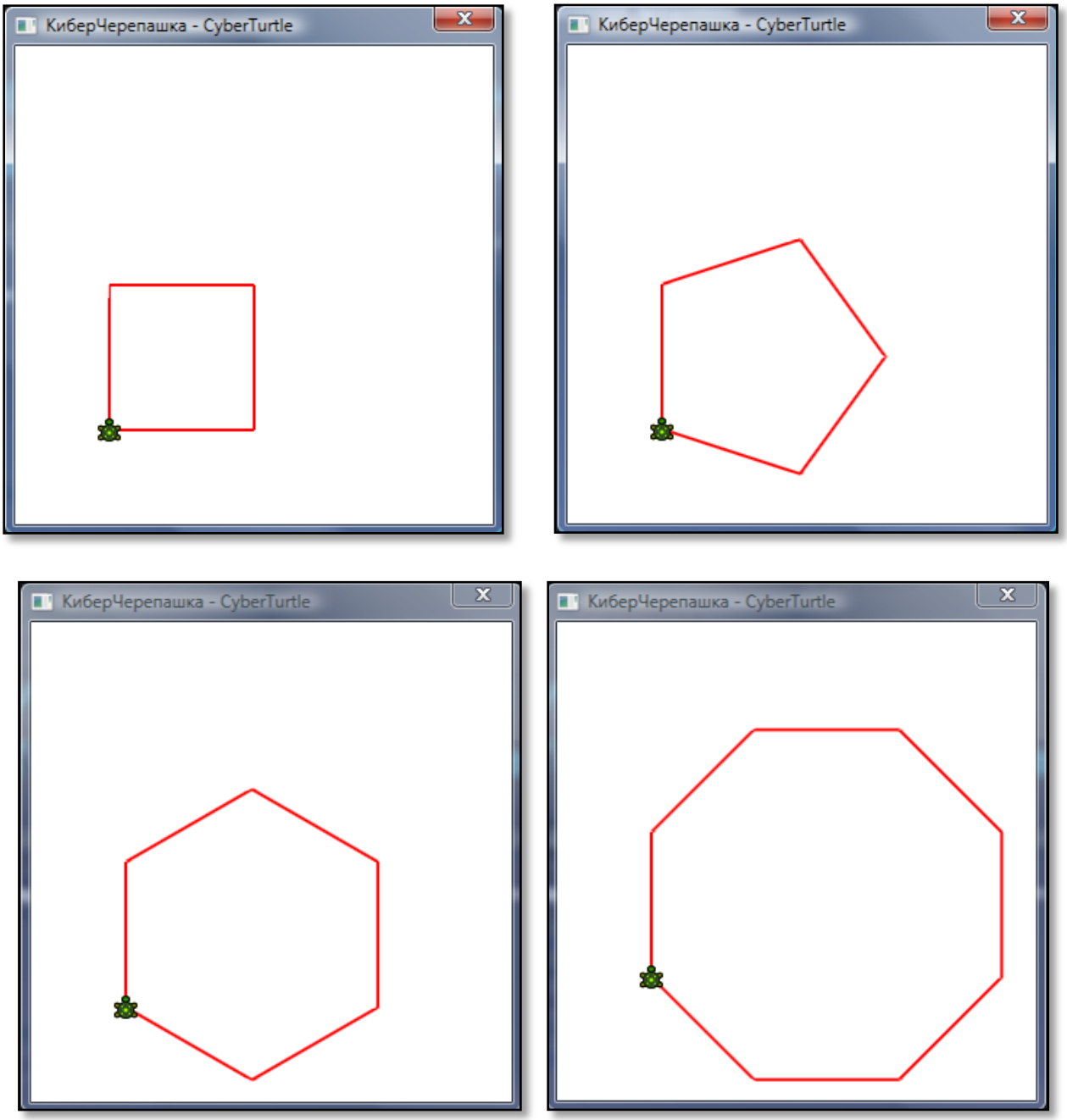


Рис. 23.5. Черепашковые многоугольники

Фракталы

Фракталы – это замечательные геометрические фигуры, составленные из точно таких же фигур, но меньшего размера. Например, из маленьких отрезков мы можем составить отрезок большей длины, из него – еще более длинный, и так до бесконечности (Рис. 23.6).

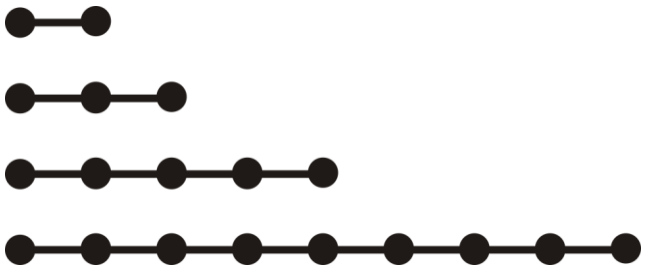


Рис. 23.6. Фрактальные отрезки

Аналогично из квадратиков мы можем построить большой квадрат (Рис. 23.7).

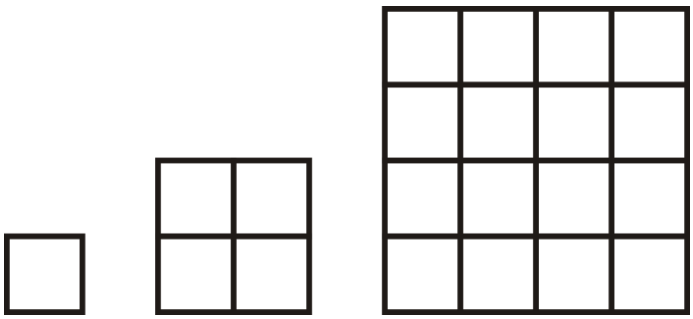


Рис. 23.7. Фрактальные квадраты

Примерами фракталов в природе могут служить: кровеносная система, береговая линия, горный рельеф, перистые облака (Рис. 23.8), разряд молнии, системы рек с притоками, трещины в почве, ветвистые растения, фьорды, прожилки на листьях (Рис. 23.9), ледяные узоры на стёклах…

Рис. 23.8. Перистые облака

Рис. 23.9. Жилкование листа

Снежинка Коха

Более сложный пример фрактальной кривой придумал в 1904 году *Гельг фон Кох*. Построение начинается с отрезка, который можно условно разделить на три равные части (Рис. 23.10а). Среднюю часть отрезка мы заменяем двумя отрезками, каждый из которых равен трети исходного отрезка (Рис. 23.10б). Затем мы повторяем эту операцию сколько угодно раз (Рис. 23.10в).

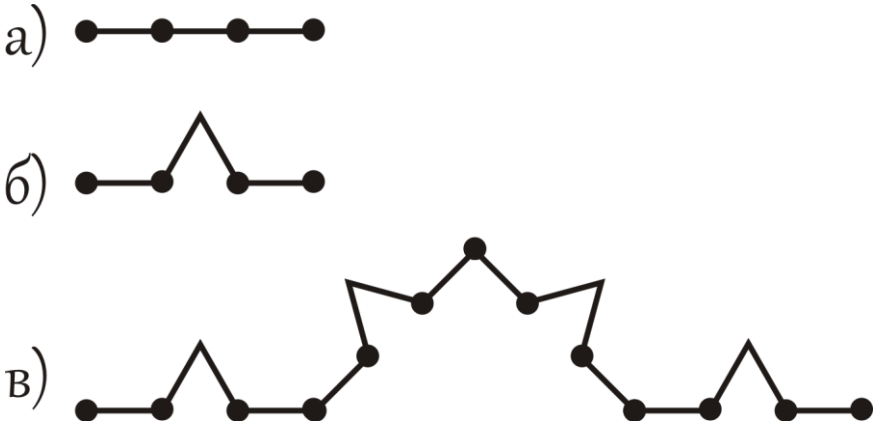


Рис. 23.10. Построение снежинки Коха

Если вы и дальше продолжите построение кривой Коха, то быстро убедитесь, что вручную её строить весьма утомительно. Поэтому лучше научить этому занятию нашу *Черепашку*, только строить она будет не кривую, а *снежинку Коха*, которая отличается от кривой только тем, что её звенья не выстраиваются в прямую линию, а поворачиваются на 60 градусов, вследствие чего кривая становится замкнутой. Итак, с углом *teta* мы определились. С аксиомой тоже можно справиться, а вот правило, порождающее инстинкт придётся позаимствовать у самого господина Коха:

```
'Снежинка Коха:
Sub Koch
  teta=60
  size=4
  iter= 1'4
  speed=10
  axiom="F++F++F"
  newF="F-F++F-F"
endSub
```

Запускаем программу:

```
Koch()
createInstinct()
start()
execute()
```

На первом году жизни *Черепашка* нарисует шестиконечную звезду, в которой легко узнать кривую Коха, повторенную 3 раза (Рис. 23.12 слева).

Вообще говоря, настоящие снежинки также имеют 6 лучей, но более сложной формы (Рис. 23.11).

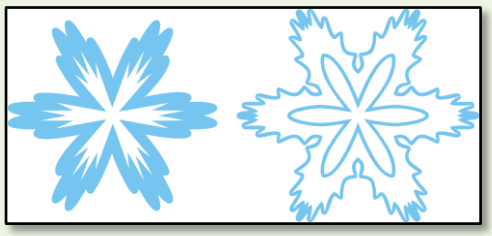


Рис. 23.11. Стилизованные снежинки

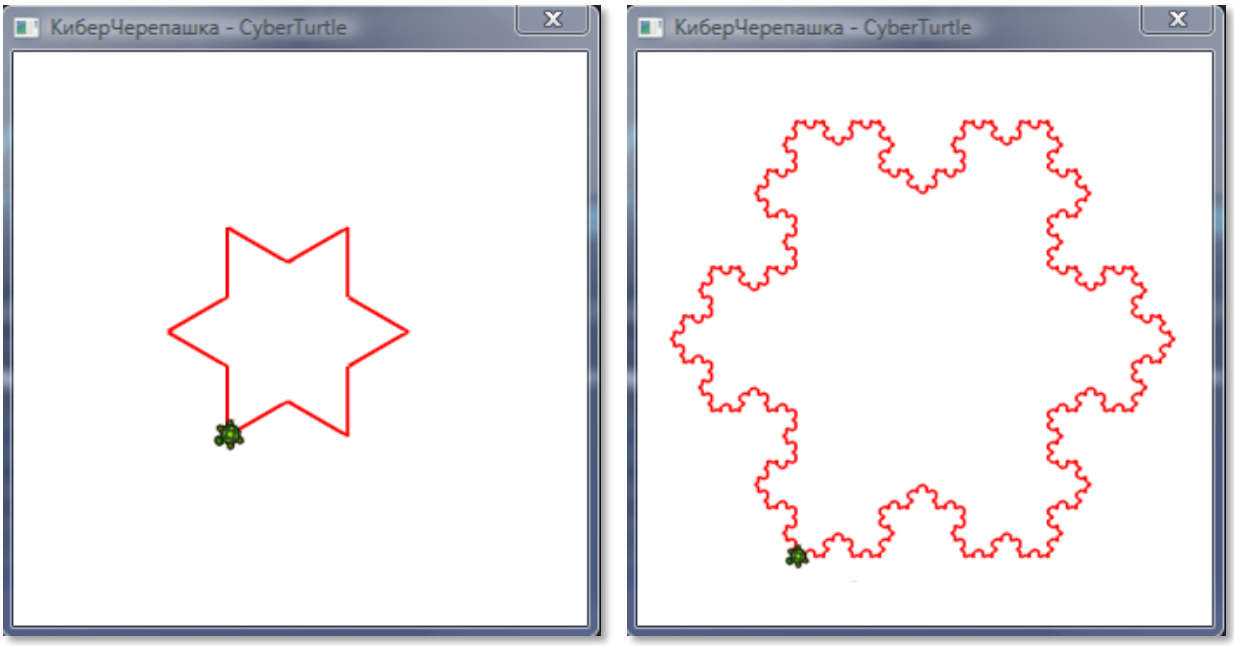


Рис. 23.12. Кривая Коха при *iter=1* и *iter=4*

С возрастом *Черепашка* рисует всё более и более сложные узоры, так что в четырёхлетнем возрасте снежинка у неё получается не хуже настоящей (Рис. 12 справа)!

В 1890 году *Джузеппе Пеано* построил функцию, график которой, названный *кривой Пеано*, покрывает квадратами (или, если угодно, ромбами) всю плоскость. И хотя кривая Пеано не является фракталом в полном смысле этого слова, но всё равно очень забавно наблюдать, как наша *Черепашка* её вычерчивает (Рис. 23.13). С годами (то есть с увеличением значения *iter*) она «квадрирует» все бо́льшую и бо́льшую поверхность.

```
'Кривая Пеано:
Sub Peano
  a0= 45
  teta=90
  size=10
  iter= 3
  speed=10
  axiom="F"
  newF="F-F+F+F+F-F-F-F+F"
endSub
```

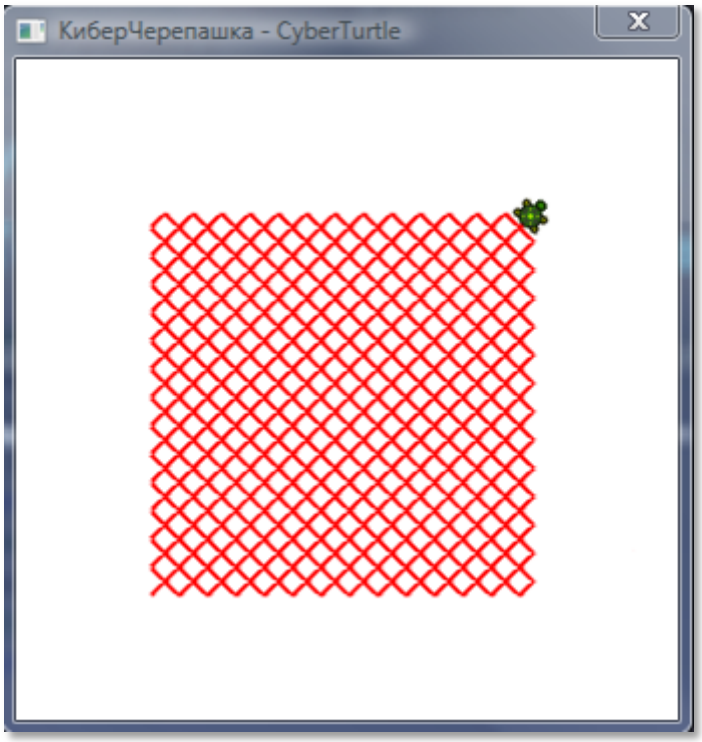


Рис. 23.13. Кривая Пеано при *iter= 3*

Ещё более усложнив инстинкт, мы научим *Черепашку* выделывать очень сложную кривую (Рис. 23.14).

```
' 32-сегментная кривая:
Sub sc32
  teta=90
  size=4
  iter= 2
  speed=10
  axiom="F+F+F+F"
  newF="-F+F-F-F+F+FF-F+F+FF+F-F-FF+FF-FF+F+F-FF-F-F+FF-F-
F+F+F-F+"
endSub
```

Добавим к инстинкту *Черепашки* новую команду, которая в генетическом коде обозначается буквой *b*. Получив такую команду, *Черепашка* переползает вперёд, но линию за собой не чертит.

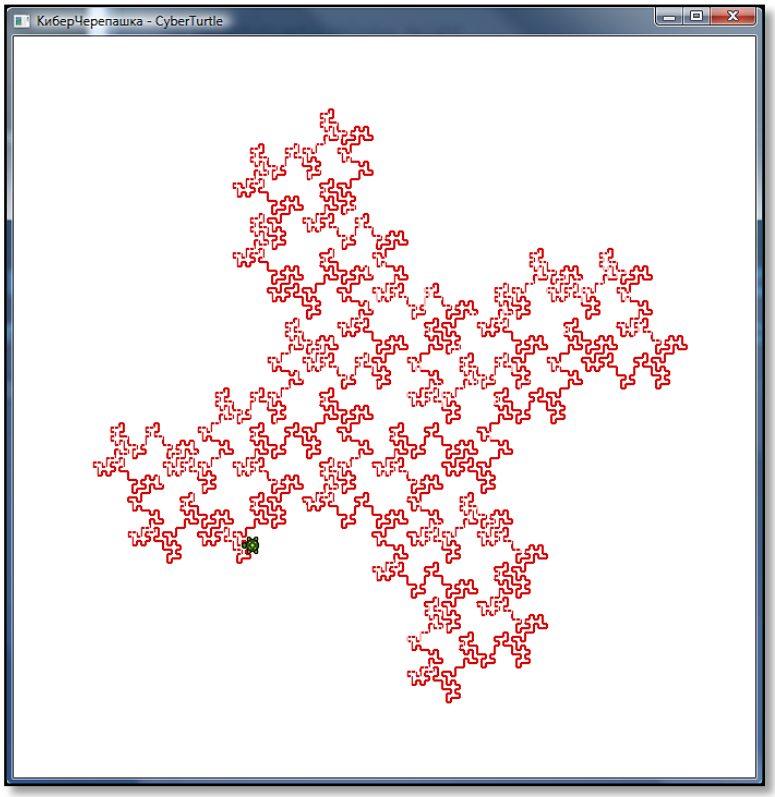


Рис. 23.14. 32-сегментная кривая при *iter= 2*

Для обработки нового гена мы добавим в процедуру *createInstinct* пару строк:

```
. . .
      ElseIf  (cmd="b") Then
        instinct= instinct + newb
. . .
```

То есть новый ген *b* входит в состав инстинкта аналогично гену *F*. А вот в процедуре *execute Черепашка* должна сначала поднять карандаш, а после перемещения снова опустить его:

```
    If (cmd="b") Then
      Turtle.PenUp()
      Turtle.Move(size)
      Turtle.PenDown()
    EndIf
```

С помощью этого гена *Черепашка* сможет рисовать фигуры, состоящие из нескольких не связанных между собой частей (Рис. 23.15).

```
'Мозаика:
Sub Mozaic
  a0= 0
  teta=90
  size=10
  iter= 2
  x0= CX
  y0= CY
  speed=10
  axiom="F-F-F-F"
  newF="F-b+FF-F-FF-Fb-FF+b-FF+F+FF+Fb+FFF"
  newb= "bbbbbb"
endSub
```

Обратите внимание: каждое вхождение нового гена *b* в инстинкт заменяется значением переменной *newb*!

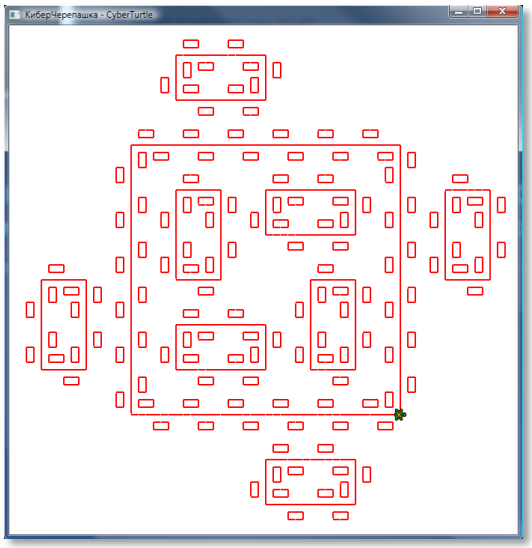


Рис. 23.15. Мозаика при *iter= 2*

Мы можем добавлять к инстинкту *Черепашки* и другие гены, которые обозначаются буквами латинского алфавита. Они участвуют в формировании инстинкта, но при его выполнении игнорируются.

Для каждого нового гена должно быть записано правило, по которому он заменяется в инстинкте другими генами. Нашей Чере-

пашке будет достаточно четырёх неисполняемых генов, которые формируют инстинкт в процедуре *createInstinct*:

```
      ElseIf  (cmd="W") Then
        instinct= instinct + newW
      ElseIf  (cmd="X") Then
        instinct= instinct + newX
      ElseIf  (cmd="Y") Then
        instinct= instinct + newY
      ElseIf  (cmd="Z") Then
        instinct= instinct + newZ
```

В процедуру *execute* никаких изменений вносить не следует, поскольку эти команды *Черепашка* не выполняет. Зато новые гены могут настолько усложнить инстинкт *Черепашки*, что она начнёт вычерчивать кривые удивительной красоты!

Помните, как на уроке *Геометрические фантазии* мы построили *Треугольный треугольник* (см. *Рис. 21.9*)? – А теперь мы обучим этому искусству нашу *Киберчерепашку*:

```
'Треугольники:
Sub triangles
  a0=30
  teta=120
  size= 20
  iter= 6
  speed=10
  axiom="bX"
  newb = "b"
  newF="F"
  newX="--FXF++FXF++FXF--"
endSub
```

Хотя задача оказалась непростой, но она справилась с заданием отлично (Рис. 23.16).

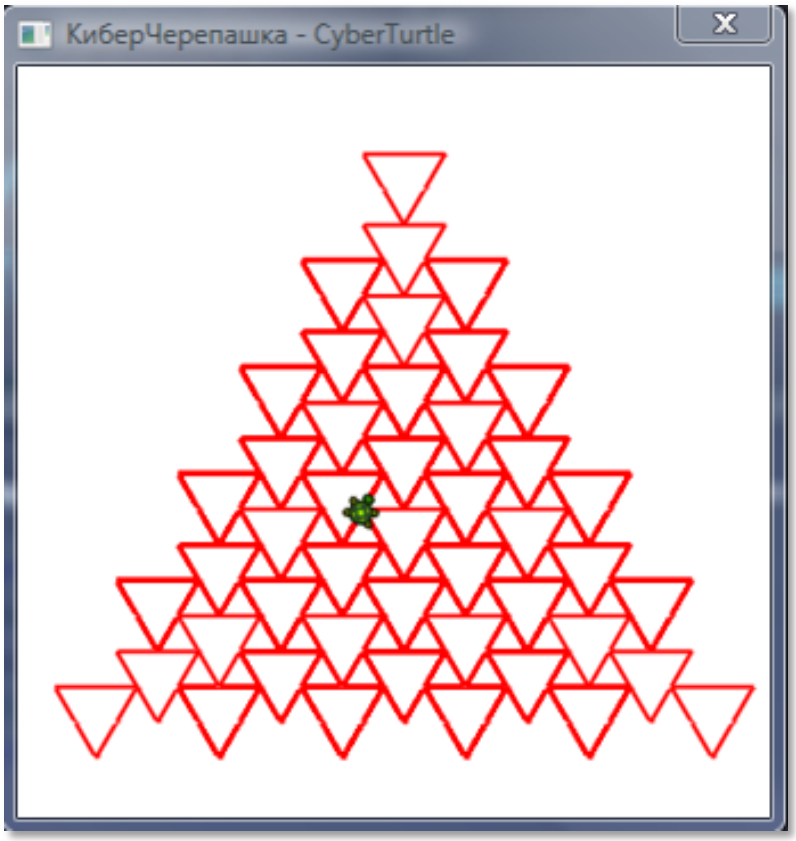


Рис. 23.16. Треугольники при *iter= 6*

С двумя новыми генами *Черепашка* построит кривую *Пеано-Госпера*, которая, как и кривая Коха, также напоминает снежинку, но заполненную внутри замысловатыми линиями (Рис. 23.17).

```
' Кривая Пеано-Госпера:
Sub PeanoGosper
  teta=60
  size=10
  iter= 4
  speed=10
  axiom="FX"
  newF="F"
  newX="X+YF++YF-FX--FXFX-YF+"
  newY="-FX+YFYF++YF+FX--FX-Y"
endSub
```

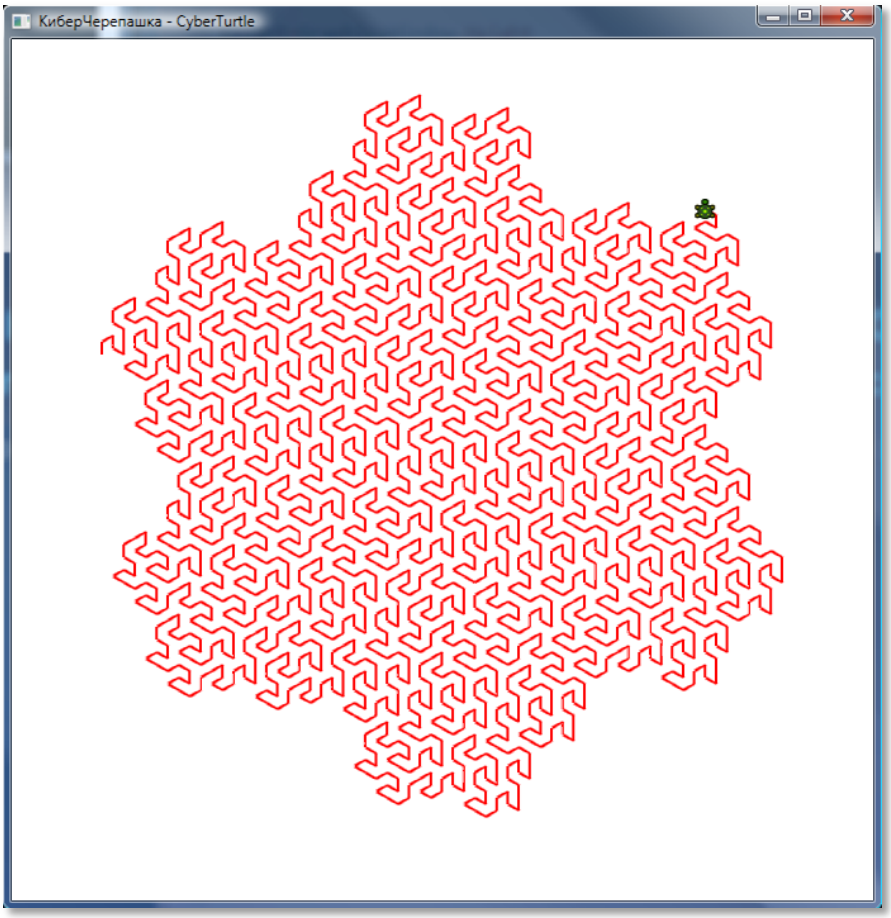


Рис. 23.17. Кривая Пеано-Госпера при *iter= 4*

А вот чтобы научить *Черепашку* рисовать почти настоящую снежинку, нам придётся наделить ее *памятью*. По команде *[* она будет запоминать свои координаты и угол поворота, а по команде *]* возвращаться в это состояние.

Поскольку эти гены не влияют на формирование инстинкта, то процедура *createInstinct* останется без изменений, а в процедуру *execute* нужно добавить строчки:

```
    If (cmd="[") Then
      Stack.PushValue(_stack, Turtle.Angle)
      Stack.PushValue(_stack, Turtle.X)
      Stack.PushValue(_stack, Turtle.Y)
    EndIf
    If (cmd="]") Then
      Turtle.Y= Stack.PopValue(_stack)
      Turtle.X= Stack.PopValue(_stack)
      Turtle.Angle= Stack.PopValue(_stack)
    EndIf
```

Как видите, память *Черепашки* устроена по принципу *стека*: *Черепашка* вспоминает *последние* запомненные события, после чего они стираются из её памяти.

Теперь *Черепашка* не только умеет выполнять команды инстинкта, но и обладает памятью. А вот так она строит почти настоящую снежинку (Рис. 23.18):

```
'Снежинка:
Sub Snowflake
  size=4
  iter= 2
  teta=60
  x0= CX
  y0= CY+100
  speed=10
  axiom="[F]+[F]+[F]+[F]+[F]+[F]"
  newF="F[++F][-FF]FF[+F][-F]FF"
endSub
```

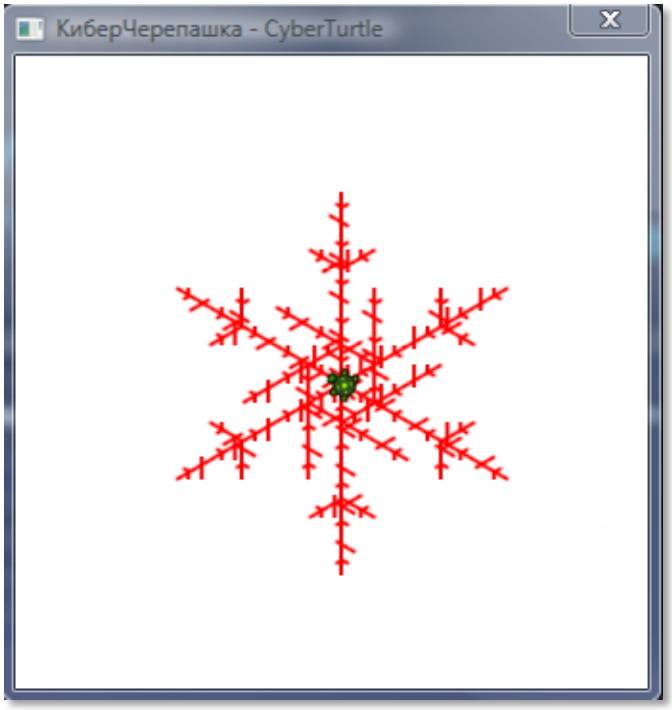


Рис. 23.18. Снежинка при *iter= 2*

С помощью очень простого инстинкта и памяти *Черепашка* нарисует очень красивую картинку, состоящую из ромбов (Рис. 23.19):

```
'Ромбы:
```

```
Sub Rhombes
  a0= 0
  teta=60
  size=10
  iter= 6
  x0= CX
  y0= CY
  speed=10
  axiom="F"
  newF="-F+F+[+F+F]-"
endSub
```

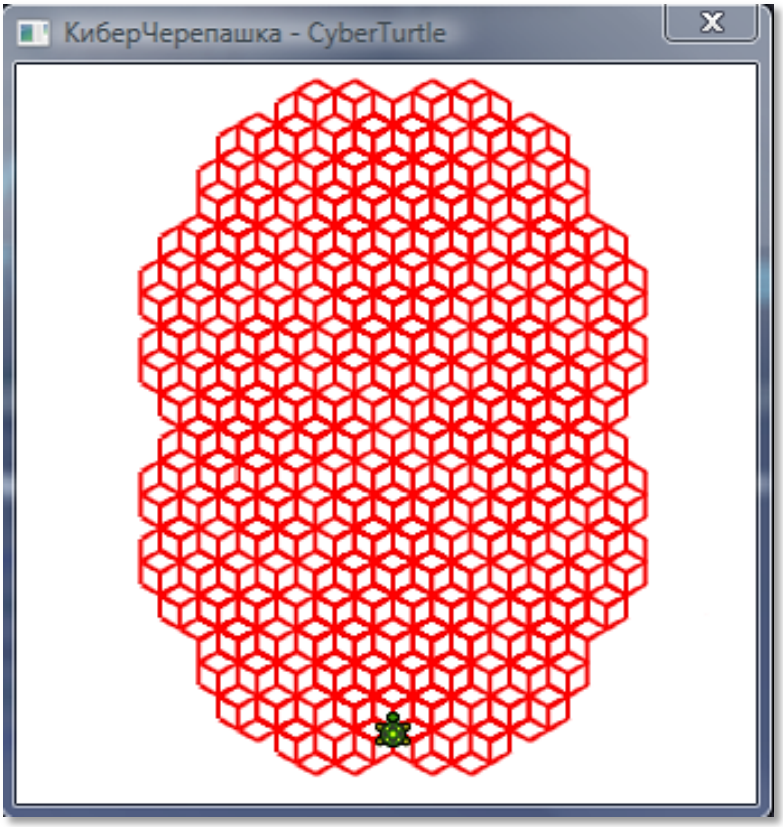


Рис. 23.19. Ромбы при *iter= 6*

Кривая Пенроуза (Рис. 23.20) также состоит из ромбов, но для её построения *Черепашке* понадобится и очень сложный инстинкт, и крепкая память:

```
'Кривая Пенроуза:
Sub Penrose
  teta=36
  size= 24
  iter= 4
```

```
  speed=10
  axiom="[Y]++[Y]++[Y]++[Y]++[Y]"
  newW="YF++ZF----XF[-YF----WF]++"
  newX="+YF--ZF[---WF--XF]+"
  newY="-WF++XF[+++YF++ZF]-"
  newZ="--YF++++WF[+ZF++++XF]--XF"
  newF=""
endSub
```

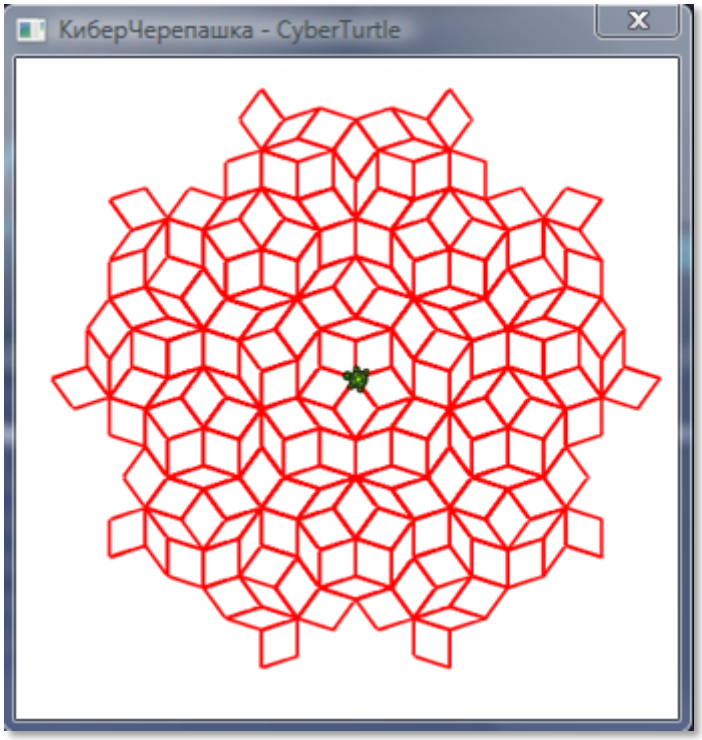


Рис. 23.20. Кривая Пенроуза при *iter= 4*

О *мозаиках Пенроуза* вы можете прочитать в книге Мартина Гарднера **[8]**.

И последнее, чему мы научим нашу *Черепашку*, - это строить кривую, которая очень напоминает настоящее *растение* (Рис. 23.21):

```
'Куст:
Sub Bush
  size=8
  iter= 4
  teta=180/8
  x0= CX
  y0= CY+100
  speed=10
```

```
  axiom="F"
  newF="-F+F+[+F-F-]-[-F+F+F]"
endSub
```

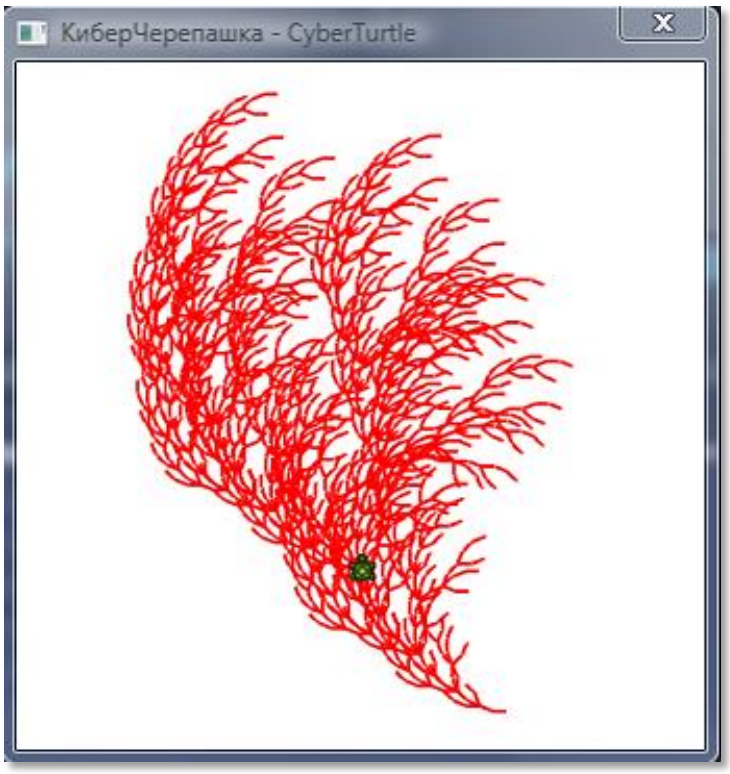


Рис. 23.21. Куст при *iter= 4*

L-системы

И вот, глядя на этот очаровательный «куст», мы можем, наконец, раскрыть тайну поведения нашей *Черепашки*. Оказывается, инстинкт *Черепашки* описывается с помощью *L-системы*, которая была предложена шведским биологом *Аристидом Линденмайером* (поэтому иначе они называются *системами Линденмайера*) для описания растений. Пример с кустом и следующий – с деревом (Рис. 23.22) - убедительно доказывают, что с помощью этой системы можно конструировать весьма правдоподобные растения (в некоторых графических редакторах они генерируются практически так же).

```
'Дерево:
Sub Tree
  size=8
```

```
  iter= 4
  teta=180/6
  x0= CX
  y0= CY+100
  speed=10
  axiom="F"
  newF="F[-F]F[+F][F]"
endSub
```

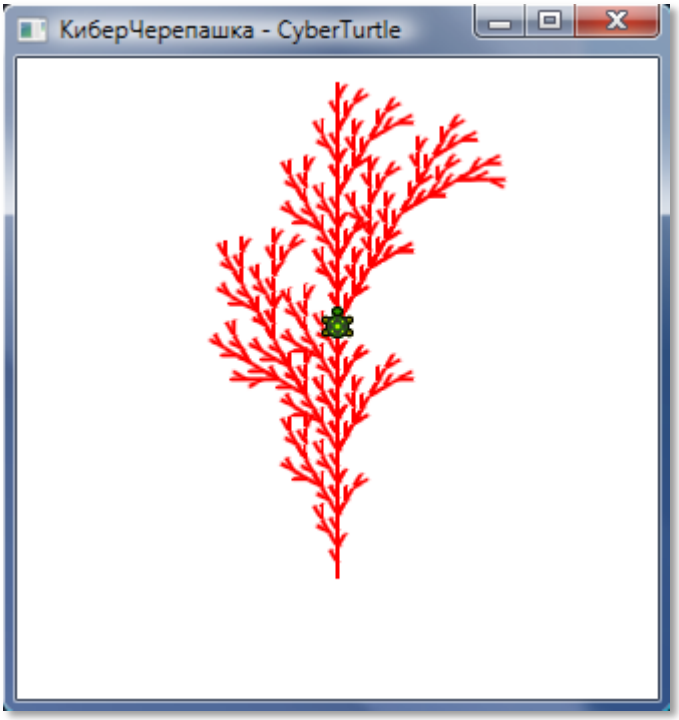
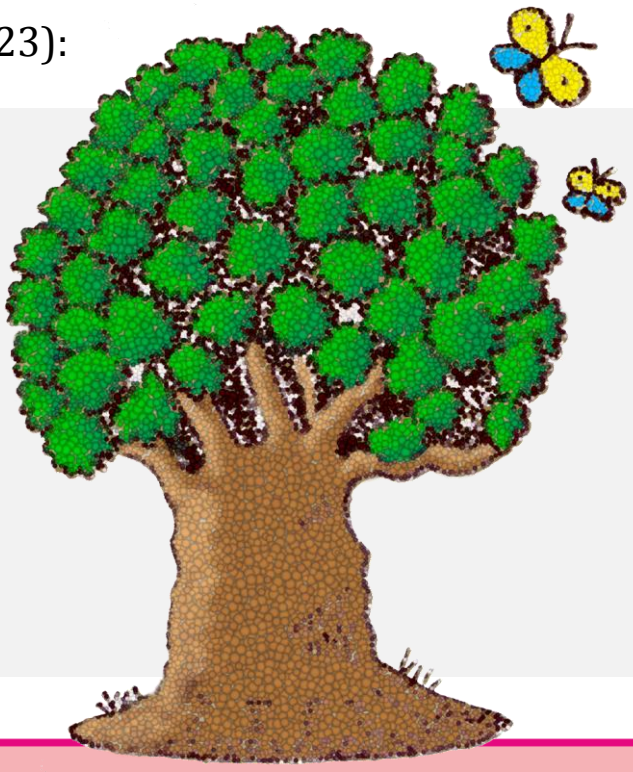


Рис. 23.22. Дерево

И ещё парочка деревьев (Рис. 23.23):



```
'Дерево2:
Sub Tree2
  size=4
  iter= 4
  teta=180/6
  x0= CX
  y0= CY+160
  speed=10
  axiom="F"
  newF="F[+F]F[-F]F"
endSub

'Дерево3:
```

```
Sub Tree3
  size=6
  iter= 4
  teta=180/6
  x0= CX
  y0= CY+160
  speed=10
  axiom="F"
  newF="FF-[-F+F+F]+[+F-F-F]"
endSub
```

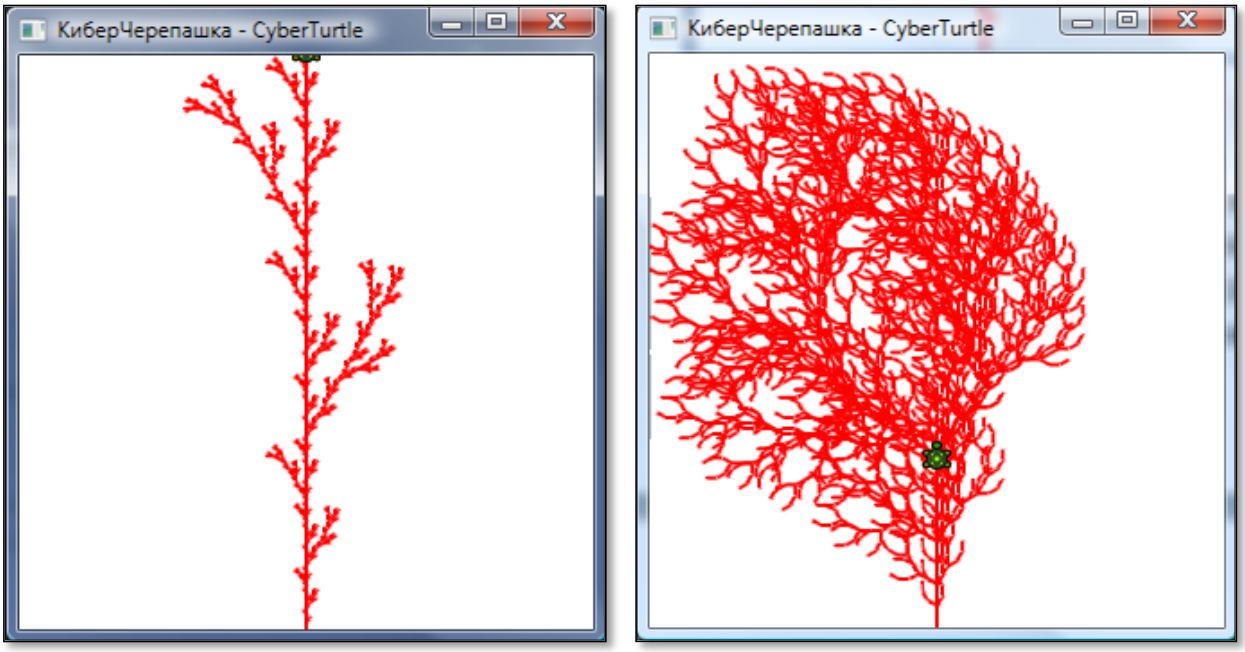


Рис. 23.23. Деревья

Все древовидные структуры, а также многие рассмотренные нами фракталы описываются аксиомой и несколькими правилами, которые с увеличением числа итераций *iter* могут создать кривые любой степени детализации (в том числе и бесконечной, если у вас есть бесконечно много времени и других ресурсов). Неожиданно оказалось, что построение кривых по системе Линденмайера очень удобно реализовать с помощью черепашьей графики, поскольку *Черепашка* умеет выполнять все команды, используемые в *L-системах*.

Исходный код программы находится в папке **CyberTurtle.**

Урок 24. Тьюрмиты

Чебурашка - это неизвестный науке зверь,
который живёт в жарких тропических лесах.

Э. Успенский, *Крокодил Гена и его друзья*

Если вам понравились кренделя, которые выделывает *Киберчерепашка*, то вряд ли вы останетесь равнодушными и к творческим порывам другого кибернетического исполнителя наших желаний, данных ему в виде инстинктов.

Зовут его **тьюрмитом** и, как всеми любимый *Чебурашка*, это совершенно неизвестный науке зверь, поскольку в глаза его никто не видел. Поговаривают, что он похож на большого муравья, а его «научное» название *тьюрмит* лингвисты производят от двух слов: **Тьюр**инг и те**рмит**.

Если термиты известны всем любителям живой природы, то причастность Тьюринга к *тьюрмитам* нуждается в пояснении. Английский математик и кибернетик Алан Тьюринг (1912 — 1954) в 1936 году придумал абстрактную вычислительную машину, которая была названа в его честь *Машиной Тьюринга*. Она может выполнять команды, записанные на бесконечной ленте, с помощью управляющего устройства, которое последовательно принимает одно из возможных состояний. Как мы увидим дальше, тьюрмит представляет собой некий примитивный вариант машины Тьюринга.

Тьюрмит также похож и на *Киберчерепашку*, так как может выполнять команды, записанные в его инстинкте. *Команд* всего четыре:

1. Перекрасить текущую клетку в заданный цвет.
2. Повернуть направо или налево на 90 градусов.
3. Переползти в соседнюю клетку в текущем направлении.
4. Перейти в новое состояние.

Тьюрмит живет в чистом поле, разделенном на клетки, которые первоначально окрашены в **чёрный** цвет. *Тьюрмит*, как и нейтрино, может существовать только в непрерывном движении,

которое заключается в переползании *тьюрмита* из одной клетки в другую, соседнюю с текущей по вертикали или горизонтали.

В каждый момент времени *тьюрмит* находится в каком-либо *состоянии*, обозначаемом буквами латинского алфавита. Например, при рождении *тьюрмит* находится в состоянии *A*. Также *тьюрмит* умеет определять цвет той клетки, на которой он стоит, и перекрашивать её в один из 16 цветов.

Мастерская тьюрмитов

Начнём новый проект и сохраним его в папке **Thurmits**. Сразу же объявим *константы*, которые пригодятся нам в программе.

Для удобства введём массив *iColor* и запишем в него все цвета, в которые могут быть окрашены клетки игрового поля:

```
'ПРОГРАММА ДЛЯ МОДЕЛИРОВАНИЯ
'ТЬЮРМИТОВ

nColor = 16   '- всего 16 разных цветов
'Цвета клеток:
iColor [0] = "Black"
iColor [1] = "Maroon"
iColor [2] = "Green"
iColor [3] = "Olive"
iColor [4] = "Navy"
iColor [5] = "Purple"
iColor [6] = "Teal"
iColor [7] = "Gray"
iColor [8] = "Silver"
iColor [9] = "Red"
iColor[10] = "Lime"
iColor[11] = "Yellow"
iColor[12] = "Blue"
iColor[13] = "Fuchsia"
iColor[14] = "Aqua"
iColor[15] = "White"
```

Тьюрмит может поворачиваться налево и направо, а также переползать вперёд - по направлению взгляда - в соседнюю слева, справа, сверху или снизу клетку (Рис. 24.1):

```
'направления движения тьюрмитов:
LEFT = -1           '- поворот/движение налево
FORWARD = 0         '- вперёд
RIGHT = 1           '- поворот/движение направо
UP = Forward        '- вверх
DOWN = 2            '- вниз
```

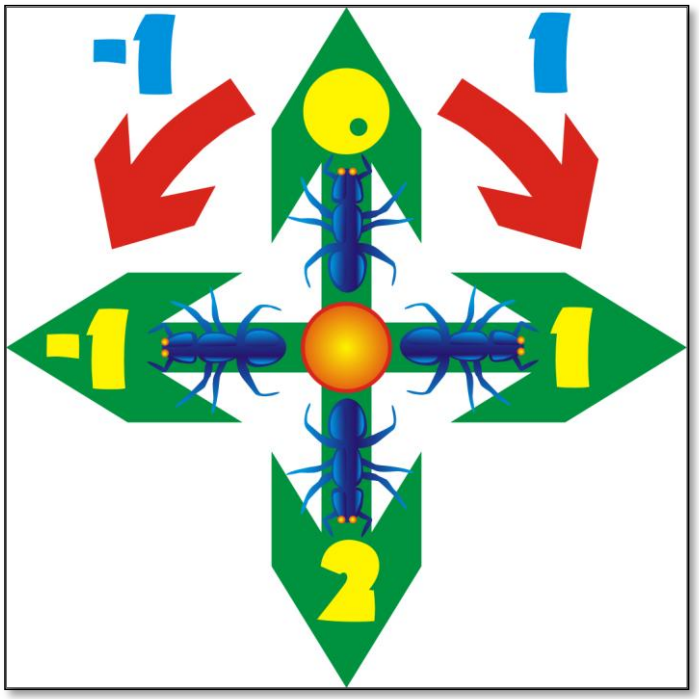


Рис. 24.1. Повороты и направления движения *тьюрмита*

Тьюрмит может переходить в одно из 26 *состояний*, которые обозначаются буквами латинского алфавита:

```
nStatus=26    '- всего разных состояний тьюрмитов - A..Z
```

Теперь займёмся *переменными*, которые описывают положение *тьюрмита* на поле и его состояние.

При рождении тьюрмит смотрит *вверх*, находится в состоянии *A* и стоит в клетке ***чёрного*** цвета:

```
'vars:
'направление движения тьюрмита:
```

```
Direction = Forward
'текущее состояние тьюрмита:
CurStatus="A"
'x-координата клетки тьюрмита:
xThurmit=0
'y-координата клетки тьюрмита:
yThurmit=0
'число ходов тьюрмита:
nMoves= 0
'инстинкт тьюрмита:
cmd[0]="END"
'массив команд:
commands[0][0]=0
```

Подготовим *поле*, на котором будет жить и творить наш *тьюрмит*:

```
'ширина клетки поля:
wCell = 10
'высота клетки поля:
hCell = 10
'отступ сетки поля от края окна по горизонтали:
OffsetX= 10
'отступ сетки поля от края окна по вертикали:
OffsetY= 60
'число строк в сетке поля:
nRow = 65
'число колонок в сетке поля:
nCol = 65
'число клеток в сетке поля:
nCell = nRow*nCol
'массив поля:
masPole[nCell]="Black"
```

Подготовку к рождению тьюрмита мы оформим в виде отдельной процедуры:

```
'=============================================

'Подготавливаем программу:
prepareProg()

'=============================================
```

Сначала мы выполняем все действия по созданию *окна* приложения.

Экзотичный *цвет фона* выбран только потому, что первоначально все клетки поля чёрные, но затем тьюрмит окрашивает их в разные цвета, в том числе и в чёрный, который будет неотличим от исходного чёрного цвета, что затруднит наши наблюдения за действиями *тьюрмита*.

```
Sub prepareProg
  'устанавливаем цвет фона:
  GraphicsWindow.BackgroundColor= "#58804E"
  'GraphicsWindow.BackgroundColor= "Black"
  GraphicsWindow.Clear()

  'заголовок окна:
  GraphicsWindow.Title="Тьюрмиты"

  'размеры окна:
  GraphicsWindow.Width  = OffsetX + (wCell*nCol)
  GraphicsWindow.Height = OffsetY+(hCell*nRow) + 60

  'окно помещаем  в центре Рабочего стола:
  GraphicsWindow.Top= (Desktop.Height - GraphicsWindow.Height)
/ 2
  GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width)
/ 2

  GraphicsWindow.CanResize="False"
  height=GraphicsWindow.Height
  width=GraphicsWindow.Width

  'координаты центра окна:
  CX= width/2
  CY= height/2
```

Для управления программой нам потребуются такие *элементы управления*:

```
  'КНОПКА
  btnPlay=Controls.AddButton("Play Turmit", 10,10)
  Controls.SetSize(btnPlay, 100,32)
  Controls.ButtonClicked= play
```

```
  'ТЕКСТОВЫЕ ПОЛЯ
  txtMove=Controls.AddTextBox(200,10)
  Controls.SetSize(txtMove, 120,26)
  'отладочное текстовое поле:
  txtDebug=Controls.AddTextBox(10,height-40)
  Controls.SetSize(txtDebug, width-20,32)
```

Число клеток, которые посетил *тьюрмит*, мы будем показывать на экране:

```
  'обнуляем число ходов:
  nMoves= 0
  'показываем число ходов в текстовом поле:
  ShowNumMoves()
```

Жизненное пространство *тьюрмита* разбиваем на клетки:

```
  'рисуем сетку:
  drawGrid ()

 'все клетки поля - чёрные:
  For i=1 To nCell
    masPole[i] = 0
  EndFor
EndSub
```

Так как *СБ* очень медленно работает с массивами, то на выполнение цикла *For*, в котором обнуляются элементы массива поля, потребуется некоторое время. Поэтому кнопку для старта игры лучше сначала *спрятать* командой

```
Controls.HideControl(btnPlay),
```

а по окончании цикла снова *показать*:

```
Controls.ShowControl(btnPlay).
```

Этот «трюк» не позволит игроку нажать кнопку до того как все элементы массива будут обновлены, иначе программа будет работать неверно.

Процедура для вывода информации о числе ходов *тьюрмита*:

```
'ПЕЧАТАЕМ ЧИСЛО СДЕЛАННЫХ ХОДОВ
Sub ShowNumMoves
  s= "Move: " + nMoves
  Controls.SetTextBoxText(txtMove, s)
EndSub
```

Следующая процедура расчерчивает поле на квадратики. Она очень простая и в комментариях не нуждается. Обратите только внимание на то, как выполняется обводка контура поля красными линиями:

```
'РИСУЕМ СЕТКУ
Sub drawGrid
  'устанавливаем размеры сетки:
  pgHeight = hCell*nRow
  pgWidth = wCell*nCol
  'и толщину линий:
  GraphicsWindow.PenWidth=2

  'чертим горизонтальные линии:
  pgYPosition= OffsetY
  For i=0 to nRow
    'устанавливаем цвет линий:
    If (i=0) or (i= nRow) Then
      GraphicsWindow.PenColor="#FF0000"
    Else
      GraphicsWindow.PenColor="#808080"
    EndIf
    GraphicsWindow.DrawLine(OffsetX, pgYPosition,
OffsetX+pgWidth, pgYPosition)
    pgYPosition= pgYPosition+  hCell
  endfor

  'чертим вертикальные линии:
  pgXPosition= OffsetX
  For i=0 to nCol
    'устанавливаем цвет линий:
    If (i=0) or (i= nCol) Then
      GraphicsWindow.PenColor="#FF0000"
    Else
      GraphicsWindow.PenColor="#808080"
    EndIf
```

```
    GraphicsWindow.DrawLine(pgXPosition, OffsetY, pgXPosition,
OffsetY+pgHeight)
    pgXPosition= pgXPosition+  wCell
  endfor
EndSub
```

Если мы теперь запустим программу, то увидим мир до рождения *тьюрмита* (Рис. 24.2).

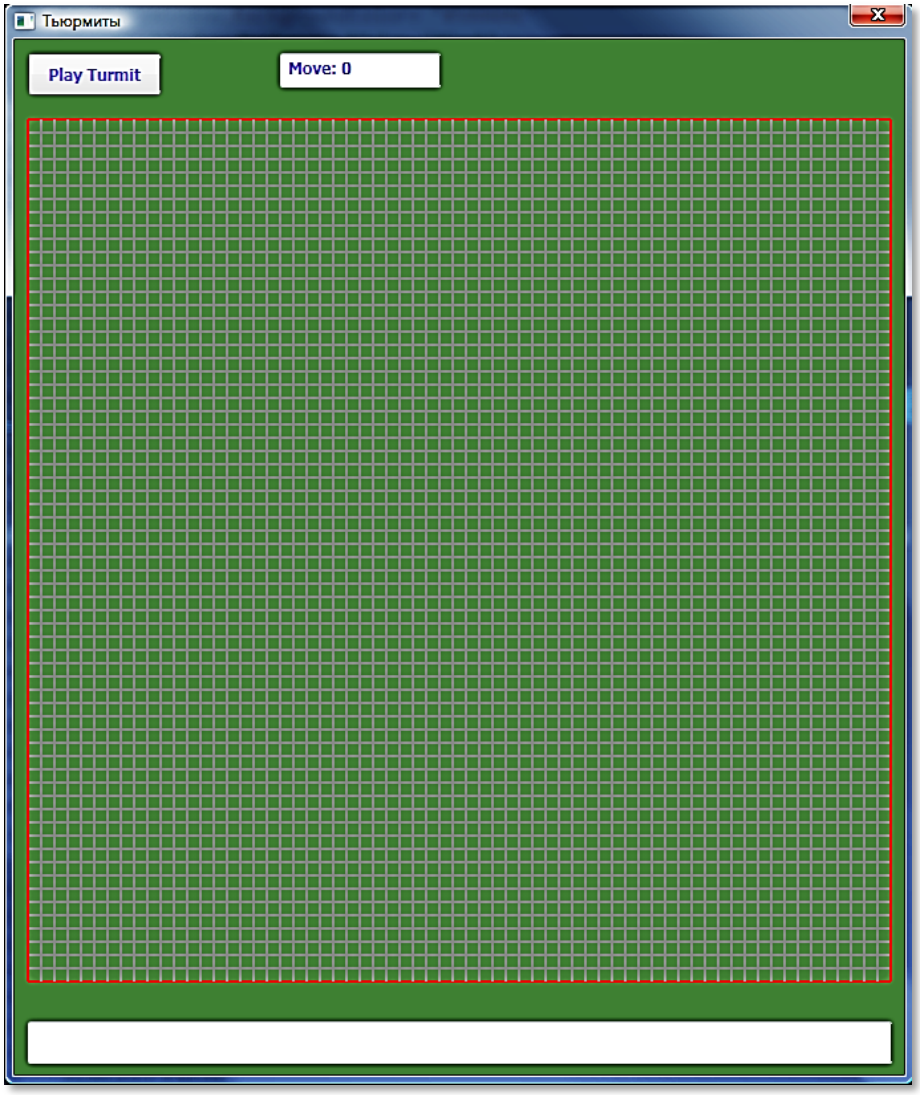


Рис. 24.2. Мир *тьюрмита* в начале времён

И вот в доселе пустом мире появляется *тьюрмит*!

```
'Рождение нового тьюрмита:
Sub prepareThurmit
  'обнуляем число ходов:
  nMoves= 0
```

```
  ShowNumMoves()

  'начальные свойства тьюрмита -->

  'тьюрмит появляется в центре поля:
  xThurmit = math.Floor(nCol/2)
  yThurmit = math.Floor(nRow/2)
  'начальный статус тьюрмита - A:
  CurStatus= "A"

  'направление движения - вперед:
  Direction = Forward

  'задаем тьюрмиту инстинкт:
  Thurmit1()
  readCommands()
EndSub
```

Весь акт творения очень прост, кроме двух последних строк, где *тьюрмит* получает инстинкт, определяющий всё его дальнейшее бренное существование.

В нашем распоряжении будет множество инстинктов, и каждый из них мы оформим в виде отдельной процедуры, так чтобы затем нам было просто создать нужного нам *тьюрмита*:

```
'////////////////////////////////////////////////
'//                                            //
'//           ИНСТИНКТЫ ТЬЮРМИТОВ              //
'//                                            //
'////////////////////////////////////////////////

Sub Thurmit1
  cmd[1]="A 0 15 1 A"
  cmd[2]="END"
EndSub
```

Как видите, инстинкт записывается в виде строк и хранится в массиве *cmd*. Последняя строка инстинкта всегда имеет значение "*END*" и отмечает конец списка.

Все остальные строки (Рис. 24.3) содержат *пять* **параметров**.

Первый параметр – буква, определяющая текущее состояние *тьюрмита*. Этот параметр обозначается прописными буквами латинского алфавита.

Второй параметр – число от 0 до 15 – код цвета той клетки, в которой находится тьюрмит. Конкретный цвет, соответствующий этому коду, определяется значением переменной типа массив *iColor*.

Третий параметр – число от 0 до 15 – код цвета той клетки, в которую *тьюрмит* должен перейти.

Четвёртый параметр – поворот тьюрмита. Обозначается числами:

-1 – поворот налево на 90 градусов
0 – не поворачиваться
1 – поворот направо на 90 градусов

Пятый параметр – буква, определяющая *новое* состояние *тьюрмита*. Также обозначается прописными буквами латинского алфавита. После выполнения строки *тьюрмит* переходит из текущего состояния в новое, которое становится текущим.

Рис. 24.3. Строка инстинкта *тьюрмита*

Сразу после рождения тьюрмит находится в состоянии *A* и занимает клетку ***чёрного*** цвета (Рис. 24.4). Таким образом, в его инстинкте первой обязательно должна быть строка, начинающаяся так: «*A 0*».

Для примера рассмотрим самый простой инстинкт:

```
Sub Thurmit1
```

```
  cmd[1]="A 0 15 0 A"
  cmd[2]="END"
EndSub
```

Последняя буква – *A* – совпадает с первой буквой, значит, при переходе в новую клетку состояние *тьюрмита* не изменяется. А поскольку весь инстинкт описывается единственной строкой, наш *тьюрмит всегда* будет находиться в одном и том же состоянии.

Тьюрмиты постоянно двигаются, поэтому и наш *тьюрмит* должен перейти в соседнюю клетку. Так как направление движения равно *0*, то он просто переползёт в ту соседнюю клетку, в сторону которой смотрит. Первоначально *тьюрмит* смотрит вверх (или, если хотите, на север), значит, и перейдёт он в соседнюю верхнюю клетку. Однако прежде он должен перекрасить текущую клетку в заданный цвет (Рис. 24.5). В инстинкте он обозначен числом *15*, что соответствует белому цвету.

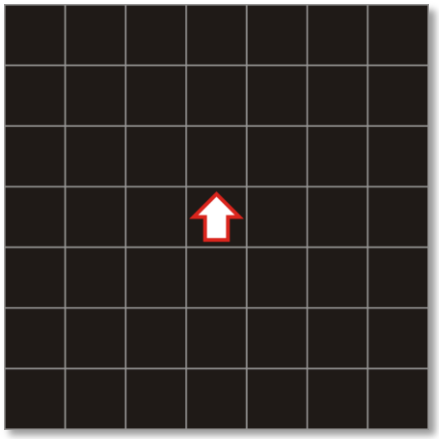


Рис. 24.4. *Тьюрмит* сразу после рождения

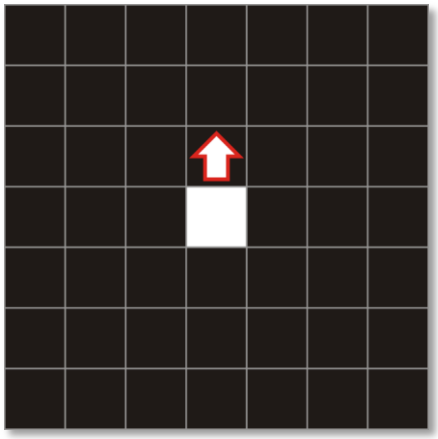


Рис. 24.5. *Тьюрмит* сделал первый шаг

Тьюрмит опять находится в состоянии *A* и стоит на **чёрной** клетке. Эту ситуации вновь описывает инстинкт в первой строке, то есть второй шаг *тьюрмита* будет в точности таким же, как и первый. Более того, мы можем утверждать, что тьюрмит будет бесконечно перемещаться вверх, окрашивая клетки в *белый* цвет (Рис. 24.6).

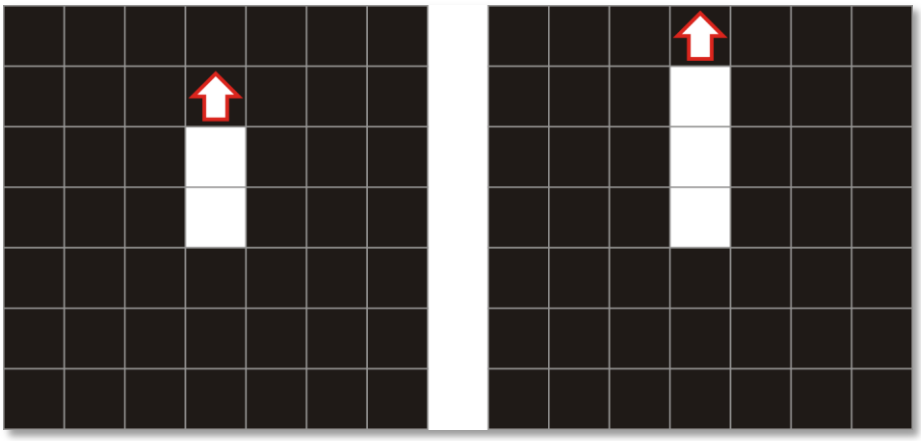


Рис. 24.6. Жизненный путь *тьюрмита*

В отличие от мира, в котором живёт *тьюрмит*, мы не можем создать ни в памяти компьютера, ни на экране монитора *бесконечный* мир, поэтому мы должны решить, чт***о*** нам делать, если *тьюрмит* покинет видимую нам часть мира.

Мы можем либо просто прекратить дальнейшие наблюдения за *тьюрмитом*, то есть прервать выполнение программы, либо вернуть его в наш мир, но уже с *обратной* стороны. В этом случае мир *тьюрмита* будет похож на поверхность *тора* (бублика) (Рис. 24.7).

Мы реализуем в программе оба сценария, но вы должны иметь в виду, что поведение *тьюрмита* в замкнутом мире будет деформировано, поскольку он может попасть на клетки, которые будут неверно истолкованы его инстинктом.

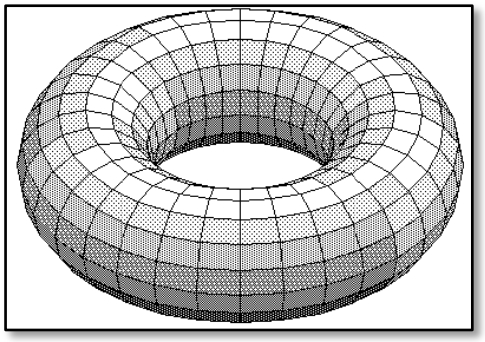


Рис. 24.7. Замкнутый «торный» мир

Ну а мы рассмотрим более сложный инстинкт, на описание которого потребовалось три строки:

```
'Зеленый квадрат 4 x 4:
Sub GreenSquare44
  cmd[1]="A 0 2 0 B"
  cmd[2]="B 0 2 0 C"
  cmd[3]="C 0 2 1 A"
  cmd[4]="END"
endSub
```

Путешествие по жизни новый *тьюрмит* начнёт с первой строки. Он перекрасит текущую клетку в **зелёный** цвет (код *2*), поднимется на клетку выше (направление равно нулю) и перейдёт в состояние *B*. (Рис. 24.8).

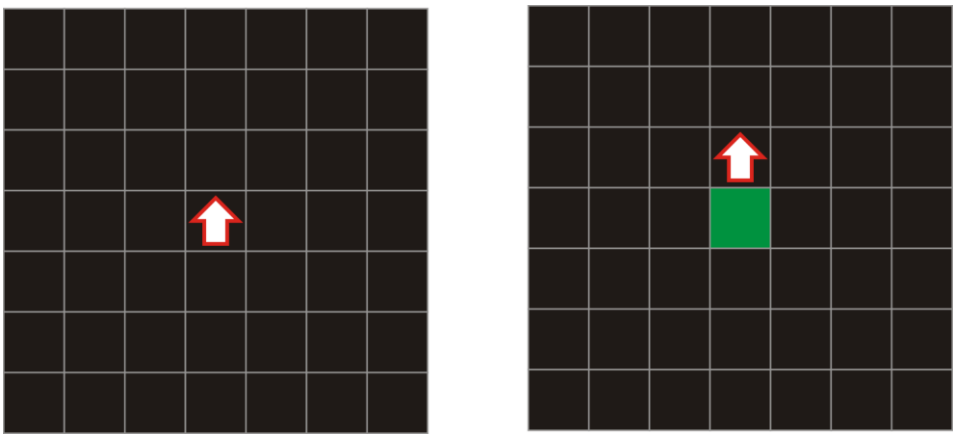


Рис. 24.8. Положение *тьюрмита* после первого хода

Теперь *тьюрмит* должен отыскать в своём инстинкте строку, начинающуюся с «*B 0*», так как он находится в состоянии *B* и занимает клетку **чёрного** цвета. Это вторая строка:

```
cmd[2]="B 0 2 0 C"
```

Повинуясь приказам инстинкта, *тьюрмит* и эту клетку покрасит в **зелёный** цвет и перейдёт в состояние *C* (Рис. 24.9).

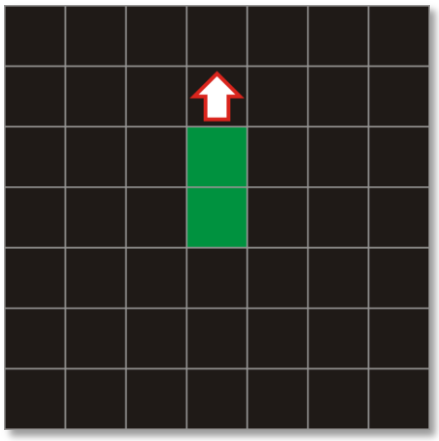


Рис. 24.9. Положение *тьюрмита* после второго хода

Далее *тьюрмит* находит строку

```
cmd[3]="C 0 2 1 A",
```

соответствующую его новому состоянию. Он вновь окрашивает клетку в **зелёный** цвет, но теперь переходит в клетку справа (код поворота равен единице). *Тьюрмит* возвращается в состояние *A* (Рис. 24.10).

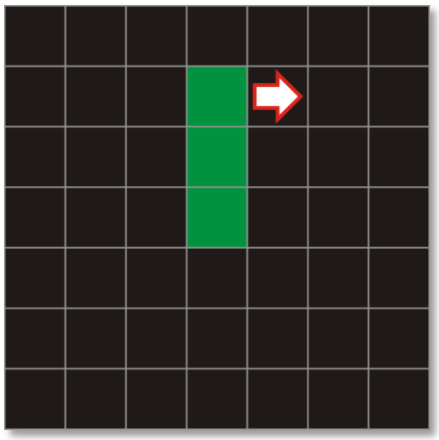


Рис. 24.10. Положение *тьюрмита* после третьего хода

Он опять должен выполнить команды, записанные в *первой* строке:

```
cmd[1]="A 0 2 0 B"
```

Но на этот раз *тьюрмит* смотрит *вправо*, поэтому перейдёт не в верхнюю клетку, а в *правую* (Рис. 24.11).

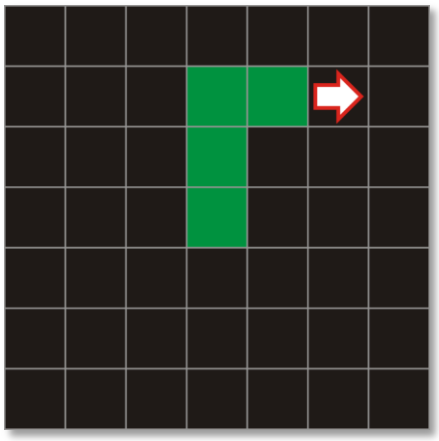


Рис. 24.11. Положение *тьюрмита* после четвёртого хода

Тьюрмит вторично выполняет *вторую* строку

```
cmd[2]="B 0 2 0 C",
```

при этом двигаясь вправо (Рис. 24.12).

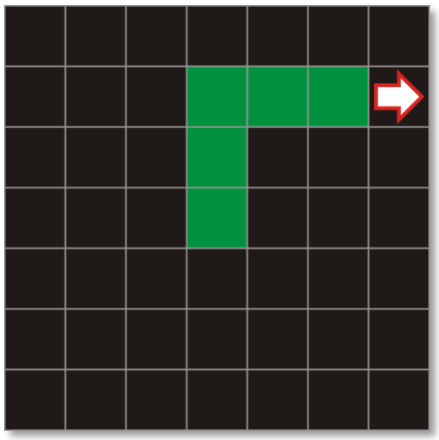


Рис. 24.12. Положение *тьюрмита* после пятого хода

Выполняя следующие команды

```
cmd[3]="C 0 2 1 A",
```

тьюрмит повернётся вправо и перейдёт в нижнюю клетку (Рис. 24.13).

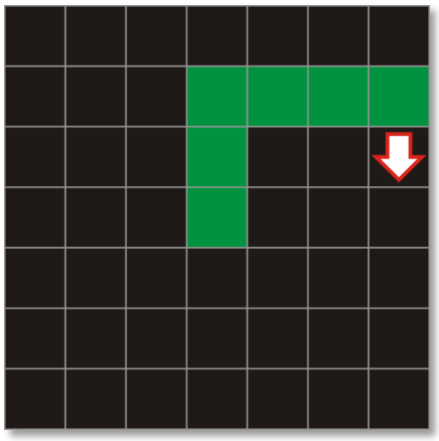


Рис. 24.13. Положение *тьюрмита* после шестого хода

Далее вы легко проследите, что тьюрмит выполнит строки

```
cmd[1]="A 0 2 0 B"
cmd[2]="B 0 2 0 C"
cmd[3]="C 0 2 1 A"
cmd[1]="A 0 2 0 B"
```

(Рис. 24.14, слева)
(Рис. 24.14, справа)
(Рис. 24.15, слева)
(Рис. 24.15, справа)

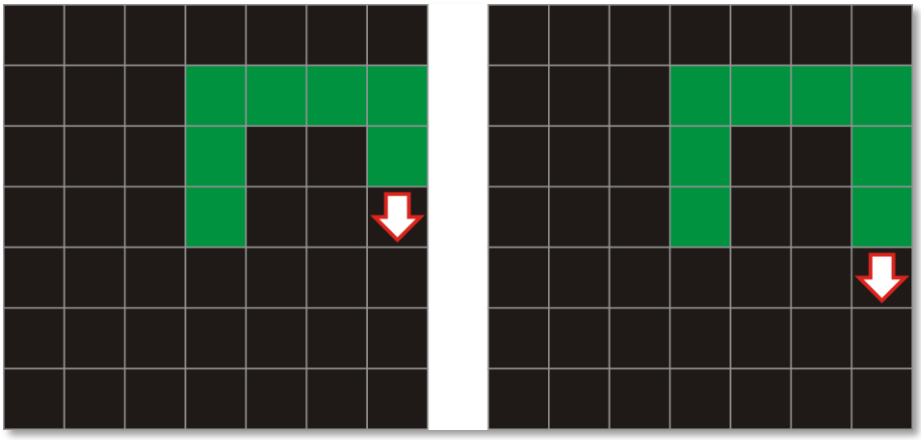


Рис. 24.14. Положение *тьюрмита* после седьмого и восьмого ходов

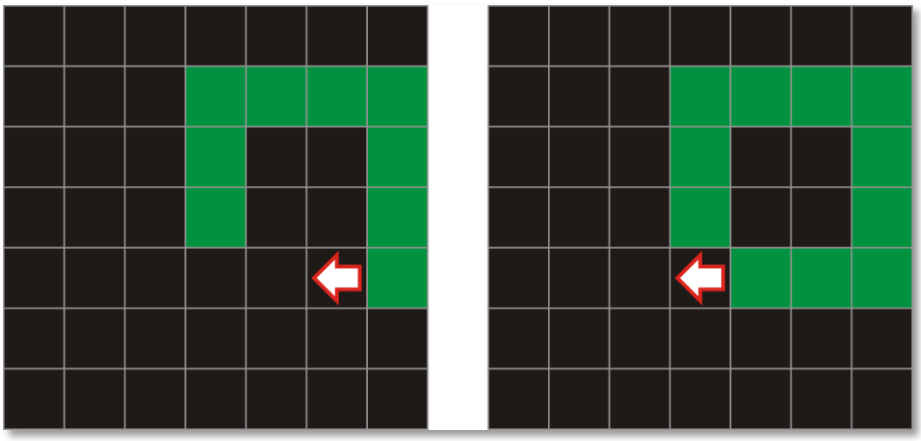


Рис. 24.15. Положение *тьюрмита* после девятого и десятого ходов

`cmd[2]="B 0 2 0 C"` (Рис. 24.16)

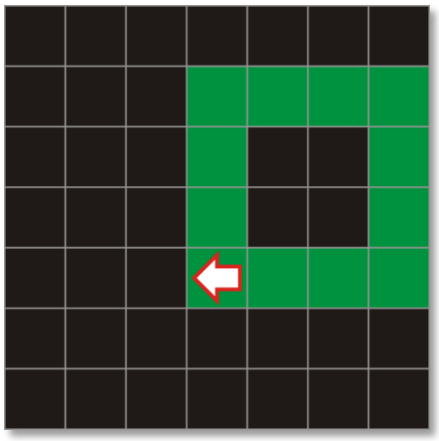


Рис. 24.16. Положение *тьюрмита* после одиннадцатого хода

В этом положении на поле *тьюрмит* находится на зелёной клетке в состоянии *C*, но в его инстинкте нет строки, начинающей с «*C 2*», поэтому *тьюрмит* погибает, а программа заканчивается.

Хотя *тьюрмитом* руководят команды, записанные в инстинкте, нам удобнее перевести их в двумерный массив команд, а не просматривать каждый раз строки инстинкта.

Оформим преобразование строк в виде отдельной процедуры:

```
'Считать строки инстинкта в массив команд
Sub readCommands
  i=1
  'обрабатываем строки, пока не достигнем конца списка:
```

```
  While (cmd[i] <> "END")
    'очередная строка инстинкта:
    str= cmd[i]
    'текущее состояние тьюрмита:
    commands[i][1]= Text.GetSubText(str, 1, 1)
    'его текущий цвет:
    n= 3
    clr=0
    While (Text.GetSubText(str, n, 1) <> " ")
      clr= clr*10 + Text.GetSubText(str, n, 1)
      n= n+1
    EndWhile
    commands[i][2]= clr
    'новый цвет тьюрмита:
    n= n+1
    clr=0
    While (Text.GetSubText(str, n, 1) <> " ")
      clr= clr*10 + Text.GetSubText(str, n, 1)
      n= n+1
    EndWhile
    commands[i][3]= clr

    'направление движения тьюрмита:
    n= n+1
    s= Text.GetSubText(str, n, 1)
    'отрицательное значение:
    If (s= "-") Then
      n= n+1
      s= s + Text.GetSubText(str, n, 1)
    EndIf
    commands[i][4]=  s
    'новое состояние тьюрмита:
    commands[i][5]= Text.GetSubText(str,
Text.GetLength(str),1)
    i = i + 1
  EndWhile
  'число строк с командами:
  nCommands= i-1
EndSub
```

Мы последовательно просматриваем строки, начиная с первой и заканчивая последней, в которой записано слово «*END*».

Формат строки нам известен: текущее состояние (буква) - текущий цвет (число) – новый цвет (число) – поворот (число) – новое состояние (буква). Для облегчения обработки строк мы будем считать, что все параметры разделяет в точности *один* пробел. В этом случае небольшие сложности возникнут только при обработке кода цвета, который может быть одно- и двузначным числом, и направления движения, которое может быть положительным и отрицательным.

Игра с *тьюрмитом* (а для него это жизнь!) начинается после нажатия на кнопку *Play Thurmit* и заключена в одной-единственной процедуре:

```
'ИГРА
Sub Play
  'производим на свет нового тьюрмита:
  prepareThurmit()
```

После рождения *тьюрмит* начинает выполнять команды первой строки, которая должна содержать его текущее состояние и цвет (то есть начинаться с *A 0*):

```
  'текущая строка в списке команд:
  curString=1
  While ("True")
    'новый статус тьюрмита:
    CurStatus= commands[curString][5]
    s=s+ " CurStatus= " + CurStatus
    'новый цвет клетки:
    NewColor= commands[curString][3]
    s=s+ " NewColor= " + NewColor

    'тьюрмит перекрашивает клетку в новый цвет:
    masPole[nCol*yThurmit+xThurmit]=NewColor
    'рисуем тьюрмита того же цвета, что и клетка:
    DrawTurmit()

    'направление движения тьюрмита:
    newDirection= commands[curString][4]
    s=s+ " newDirection= " + newDirection
    Controls.SetTextBoxText(txtDebug, s)
```

```
    Direction= Direction+newDirection
    'корректируем направление:
    If  Direction < LEFT Then
      Direction=DOWN
    EndIf
    If  Direction > DOWN Then
      Direction =LEFT
    EndIf

    'клетка, в которую должен перейти тьюрмит:
    If Direction = LEFT Then
      xThurmit= xThurmit-1
      yThurmit= yThurmit+0
    EndIf

    If Direction = UP Then
      xThurmit= xThurmit+0
      yThurmit= yThurmit-1
    EndIf

    If Direction = RIGHT Then
      xThurmit= xThurmit+1
      yThurmit= yThurmit+0
    EndIf

    If Direction = DOWN Then
      xThurmit= xThurmit+0
      yThurmit= yThurmit+1
    EndIf

    'тьюрмит  появляется с обратной стороны поля:
    'WRAP=1
    'программа заканчивается, если тьюрмит выходит за границу
поля:
    WRAP=0     if WRAP=0 Then
      If xThurmit = nCol Then
        s="  Too big X! "
        Goto exit
      EndIf

      If yThurmit = nRow Then
        s="  Too big Y! "
        Goto exit
      EndIf
```

```
      If xThurmit < 0 Then
        s="  Too small X! "
        Goto exit
      EndIf

      If yThurmit < 0 Then
        s="  Too small Y! "
        Goto exit
      EndIf
    EndIf

    'тьюрмит  появляется с обратной стороны поля:
    if WRAP=1 Then
      If xThurmit = nCol Then
        xThurmit = 0
      EndIf
      If xThurmit < 0 Then
        xThurmit = nCol-1
      EndIf

      If yThurmit = nRow Then
        yThurmit = 0
      EndIf
      If yThurmit < 0 Then
        yThurmit = nRow-1
      EndIf
    EndIf
```

Затем *тьюрмит* ищет в списке строку, соответствующую его новому состоянию и цвету текущей клетки. Если такой строки он не обнаружит, значит, он, как и мавр, сделал своё дело и должен уйти на покой:

```
    'тьюрмит ищет следующую строку:
    For i=1 to  nCommands
      if (commands[i][1]= CurStatus) and
(masPole[nCol*yThurmit+xThurmit]= commands[i][2]) then
        curString=i
        Goto break
      EndIf
    EndFor
```

```
    'нужной строки нет --> тьюрмит погибает:
    s= "    END of TURMIT    "
    Goto exit
    break:
```

В противном случае жизнь *тьюрмита* продолжается в бесконечном цикле *While*:

```
    'тьюрмит сделал очередной ход:
    nMoves= nMoves +1
    ShowNumMoves()
  EndWhile

  exit:
  'выдаем сообщение:
  GraphicsWindow.ShowMessage(s, "Тьюрмит")
EndSub
```

И в последней процедуре проекта нам осталось только нарисовать самого героя сей жизненной драмы – *тьюрмита*:

```
'Рисуем тьюрмита в текущей позиции
Sub DrawTurmit
  clr=iColor[masPole[nCol*yThurmit+xThurmit]]
  GraphicsWindow.BrushColor= clr
  x=OffsetX+xThurmit*wCell
  y=OffsetY+yThurmit*hCell
  GraphicsWindow.FillEllipse(x,y,8,8)
  'Program.Delay(50)
endsub 'DrawTurmit
```

Его внешний облик представлен в программе невыразительным кружком того же цвета, что и занимаемая им клетка. Таким образом, наш герой, как и персонаж *Рассказа о неизвестном герое* Самуила Маршака, совершенно незаметен в жизни, хотя и оставляет после себя яркие воспоминания в форме прелестно раскрашенного поля. Так воздайте же *тьюрмиту* должное и изобразите его в полной красе, как и подобает всякому труженику полей.

Рассказ о *тьюрмитах* оказался довольно продолжительным, но *тьюрмит* не должен разочаровать вас в столь долгих ожиданиях прекрасного. Сейчас вы убедитесь, что не зря изучали анатомию и психологию тьюрмитов. С помощью простых инстинктов мы научим *тьюрмита* делать столь экстравагантные па, что дух захватывает!

Однако начнём мы с простых экзерсисов, чтобы вы смогли проверить, что наши бумажные манипуляции с генетическим кодом зверушки полностью соответствуют его жизненным устремлениям. *Первый* инстинкт выполняется в полной мере и безукоризненно. *Тьюрмит* действительно уходит за верхнюю границу доступного нам мира (Рис. 24.17).

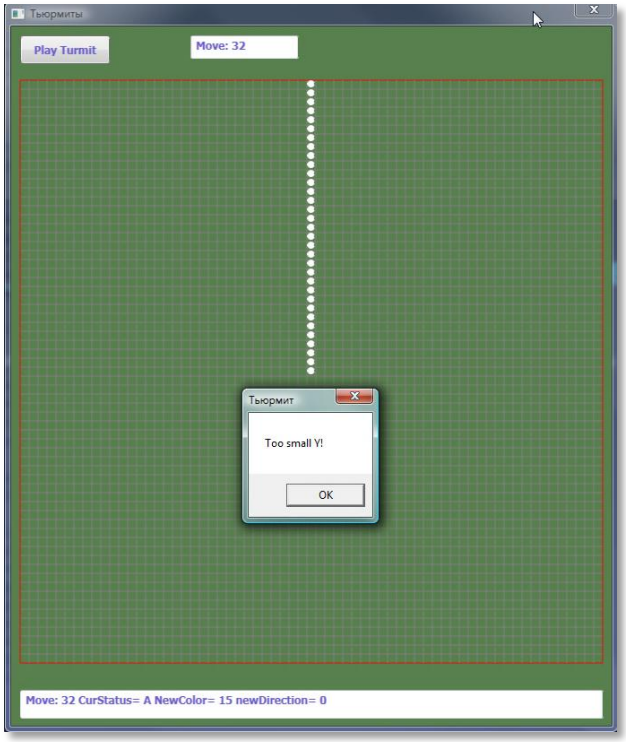


Рис. 24.17. *Тьюрмит* стремительно уходит вверх

Второй очень простой инстинкт заставляет *тьюрмита* нарисовать белый квадратик (Рис. 24.18):

```
Sub Thurmit2
  cmd[1]="A 0 15 1 A"
  cmd[2]="END"
EndSub
```

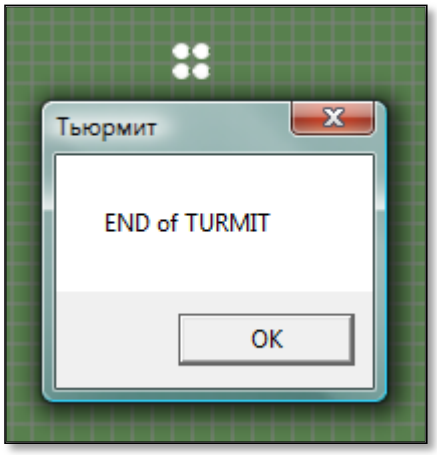


Рис. 24.18. *Тьюрмит* «оквадратился»

Далее *тьюрмит* интеллектуально прогрессирует и осиливает большие квадраты - 3 x 3 клетки (Рис. 24.19):

```
Sub GreenSquare33
  cmd[1]="A 0 2 0 B"
  cmd[2]="B 0 2 1 A"
  cmd[3]="END"
endSub
```

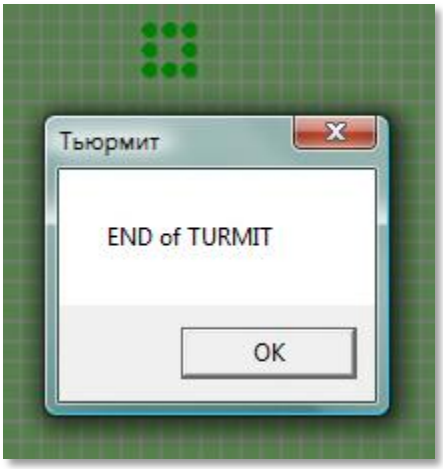


Рис. 19. Квадрат *тьюрмита* порядка 3

И 4 x 4 (этот пример мы разбирали вручную) (Рис. 24.20):

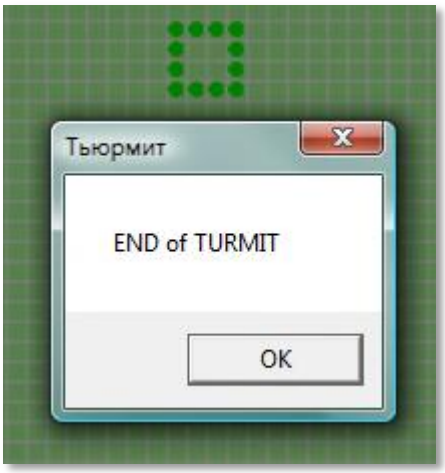


Рис. 24.20. Квадрат *тьюрмита* порядка 4

Тьюрмит может нарисовать и более сложную фигуру, например, *крест* (Рис. 24.21):

```
'Cross!
Sub Thurmit5
  cmd[1]="A 0 2 0 B"
  cmd[2]="B 0 2 0 C"
  cmd[3]="C 0 2 1 A"
  cmd[4]="A 2 1 -1 A"
  cmd[5]="END"
endSub
```

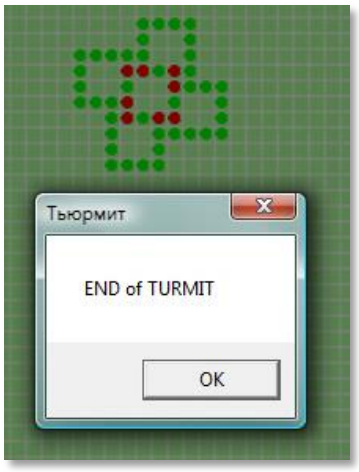


Рис. 24.21. Крест *тьюрмита*

Впрочем, статические рисунки, даже самые красивые, не могут сравниться с теми визуальными превращениями, которые пока-

зывает живой, полный энергии *тьюрмит*, стремительно носящийся по экрану. Наблюдать за ним – настоящее удовольствие! Поэтому достоинства двух следующих инстинктов (Рис. 24.22 и 24.23) трудно оценить по снимкам с экрана, их следует смотреть вживую.

```
'Спираль Тарка
Sub Thurmit4
  cmd[1]="A 0 2 -1 A"
  cmd[2]="A 2 0 0 B"
  cmd[3]="B 0 2 1 A"
  cmd[4]="B 2 2 1 A"
  cmd[5]="END"
endSub
```

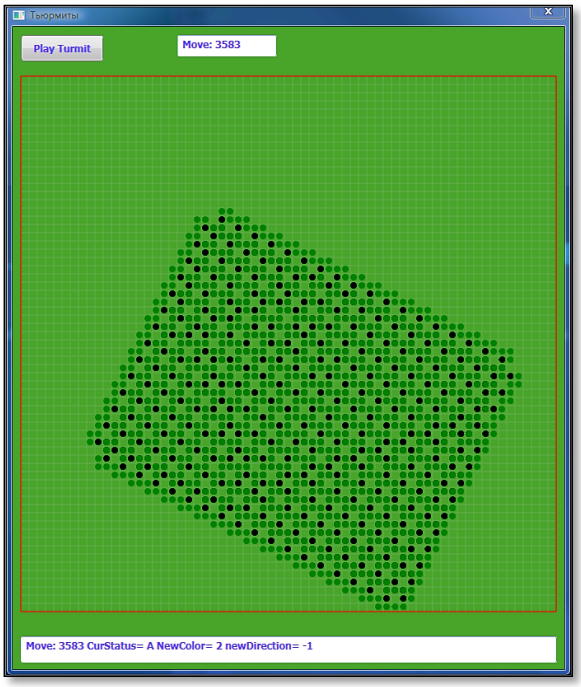


Рис. 24.22. Спираль Тарка

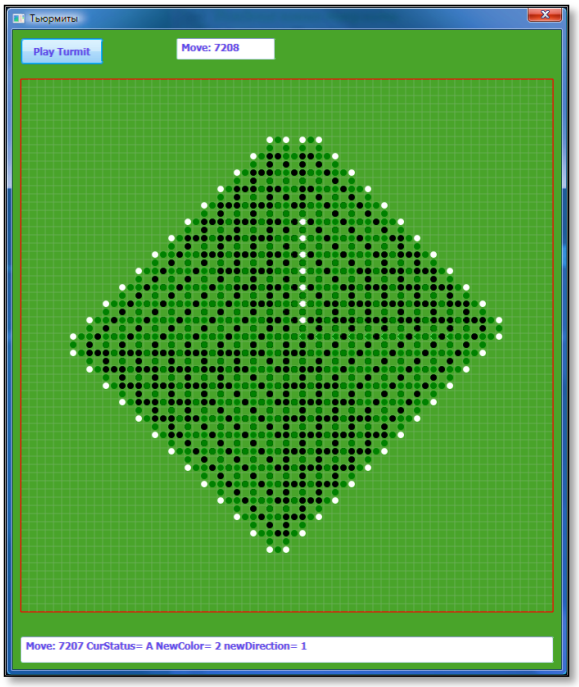


Рис. 24.23. Квадрат Кнопа

```
'Квадрат Кнопа
Sub KnopSquare
  cmd[1]="A 0 2 0 C"
  cmd[2]="A 2 0 0 B"
  cmd[3]="B 2 2 1 A"
  cmd[4]="B 15 2 1 A"
  cmd[5]="C 2 0 -1 A"
  cmd[6]="C 0 15 -1 A"
  cmd[7]="C 15 2 -1 A"
  cmd[8]="END"
```

```
EndSub
```

Следующий инстинкт – это более хитроумная вариация самого простого, первого инстинкта: он рисует не бесконечную белую линию, а бесконечную *шахматную доску* (Рис. 23.24):

```
'Шахматная доска
Sub ChessBoard2
  cmd[1]="A 0 15 -1 B"
  cmd[2]="B 0 9 0 C"
  cmd[3]="C 0 15 0 D"
  cmd[4]="D 0 9 0 E"
  cmd[5]="E 0 15 0 F"
  cmd[6]="F 0 9 0 G"
  cmd[7]="G 0 15 0 H"
  cmd[8]="H 0 9 1 I"
  cmd[9]="I 0 15 1 J"
  cmd[10]="J 0 9 0 C"
  cmd[11]="I 15 15 1 K"
  cmd[12]="K 9 9 1 L"
  cmd[13]="L 15 15 1 L"
  cmd[14]="L 9 9 -1 A"
  cmd[15]="A 15 15 1 A"
  cmd[16]="END"
EndSub
```

И, конечно, какая ж «песня» без *спирали* (Рис. 23.25)!

```
'GridSpiral
Sub GridSpiral
  cmd[1]="A 0 14 0 J"
  cmd[2]="J 0 13 -1 A"
  cmd[3]="A 14 13 0 A"
  cmd[4]="A 13 14 0 A"
  cmd[5]="J 14 13 0 A"
  cmd[6]="J 13 14 1 A"
  cmd[7]="END"
EndSub
```

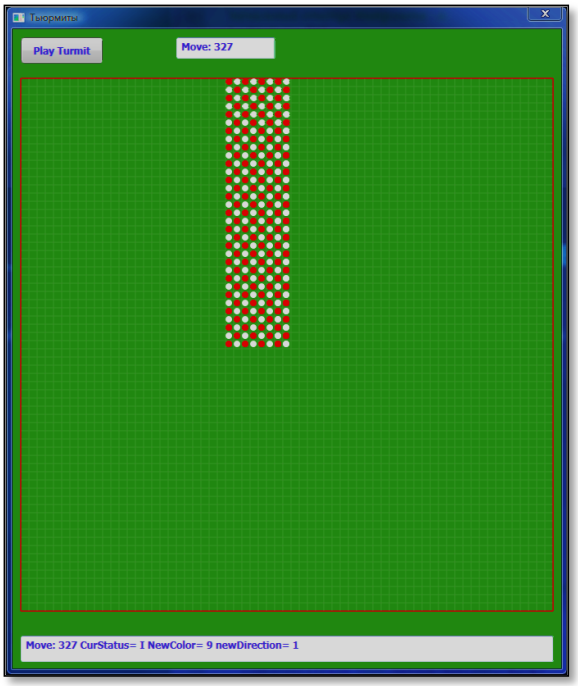


Рис. 24.24. Шахматная доска

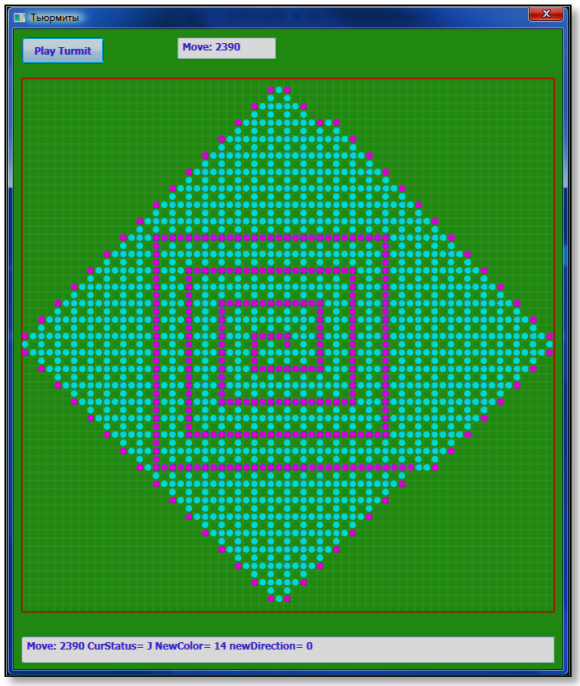


Рис. 24.25. Сетка-спираль

И последняя пара примеров весьма причудливых инстинктов - *клубок* (Рис. 24.26) и еще одна *спираль - ромбическая* (Рис. 24.27).

Ещё несколько инстинктов вы найдёте в исходном коде проекта.

```
' Клубок
Sub Klubok
  cmd[1]="A 0 15 1 B"
  cmd[2]="C 0 10 -1 A"
  cmd[3]="B 15 15 1 C"
  cmd[4]="A 15 15 1 B"
  cmd[5]="B 10 10 1 C"
  cmd[6]="B 0 15 1 A"
  cmd[7]="C 15 15 0 B"
  cmd[8]="END"
EndSub

' Ромбическая спираль
Sub RombGridSpiral
  cmd[1]="A 0 14 0 B"
  cmd[2]="B 0 14 0 C"
  cmd[3]="C 0 14 0 E"
  cmd[4]="E 0 14 0 F"
  cmd[5]="F 0 14 0 J"
```

```
  cmd[6]="J 0 13 -1 A"
  cmd[7]="A 14 13 0 A"
  cmd[8]="A 13 14 0 A"
  cmd[9]="J 14 13 0 A"
  cmd[10]="J 13 14 1 A"
  cmd[11]="END"
EndSub
```

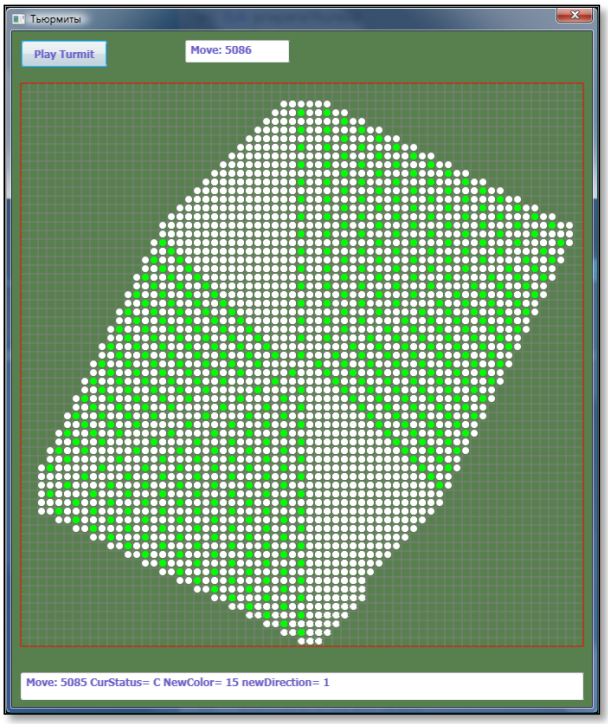


Рис. 24.26. Клубок

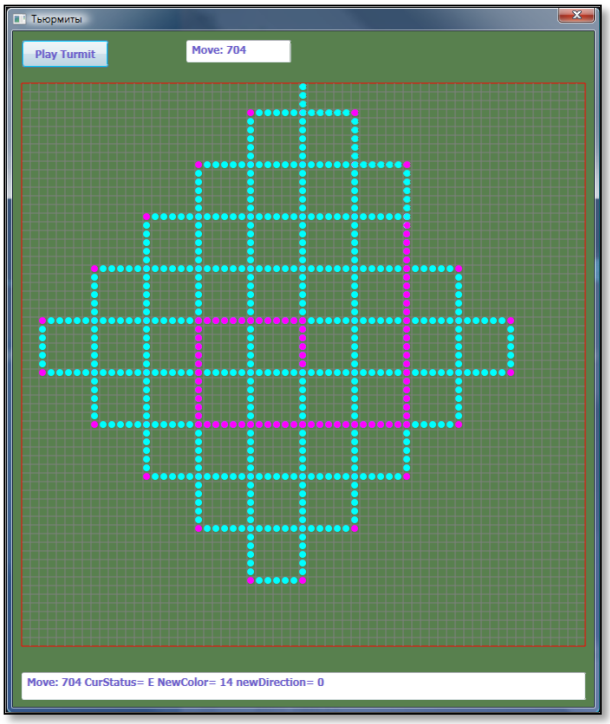


Рис. 24.27. Ромбическая спираль

Исходный код программы находится в папке **Thurmits**.

Мы уже упоминали, что обнуление массива поля – довольно медленная операция, поэтому нужно постараться вообще избежать её. Для этого можно узнавать цвет клетки непосредственно по цвету пикселей на экране, что совсем несложно. Тогда размеры поля можно значительно увеличить, вплоть до клеток величиной в один пиксель.

Урок 25. Элементы управления

До версии 0.9 интерфейс приложений, написанных на *СБ*, был очень скудным и не имел даже кнопок. Тут дело даже не во внешнем виде приложений, а в тех трудностях, которые испытывали пользователи при работе с подобными программами. К счастью, теперь в нашем распоряжении имеется несколько визуальных элементов управления (их можно видеть в клиентской области окна) и мы можем дополнить интерфейс приложений кнопками и другими элементами управления. Все элементы управления (ЭУ) создаются с помощью методов класса *Controls*.

Например, любое приложение *Windows* должно иметь хотя бы одну *кнопку*, чтобы запускать и останавливать выполнение программы. Давайте создадим приложение с кнопкой.

Для этого мы воспользуемся методом

```
Controls.AddButton(caption, left, top)
```

Параметр *caption* – это надпись на кнопке, а остальные два параметра задают положение кнопки в клиентской области.

Начнём нашу новую программу с того, что добавим *кнопку* (Рис. 25.1):

```
GraphicsWindow.Title=" Кнопки"

GraphicsWindow.Width= 320
GraphicsWindow.Height=240
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"

btnTimer=Controls.AddButton("Запустить таймер", 10,
GraphicsWindow.Height-48)
```

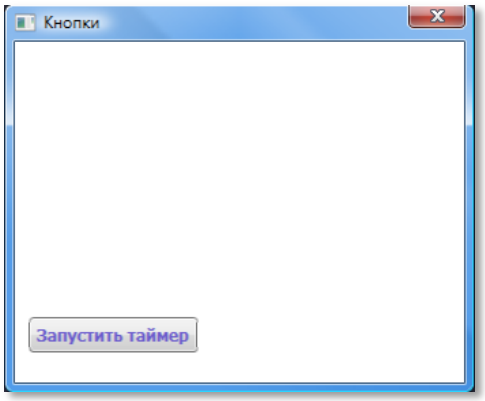


Рис. 25.1. Кнопка на месте!

Обратите внимание на то, что кнопке следует дать *имя*, по которому мы сможем потом к ней обращаться. Чтобы отличать названия кнопок от других идентификаторов программы, мы будем начинать названия кнопок с префикса *btn*. Это сокращение от английского слова *button*, что и означает *кнопка*.

Размеры кнопки по умолчанию будут такими, чтобы надпись поместилась на ней целиком. Если же мы захотим изменить размеры кнопки или любого другого элемента управления, то укажем его *имя* первым в методе

```
Controls.SetSize(control, width, height)
```

а затем – *ширину* и *высоту* кнопки. Добавим к программе одну строчку:

```
Controls.SetSize(btnTimer,164,32)
```

И наша кнопка станет более солидной (Рис. 25.2).

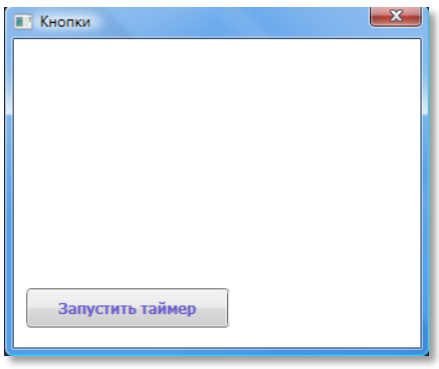


Рис. 25.2. Увеличенная кнопка

Иногда нужно *переместить* кнопку в другую позицию в окне приложения. Для этого имеется метод

```
Controls.Move(control, x, y)
```

Первый параметр нам уже известен, второй и третий – новые *координаты* левого верхнего угла элемента управления.

Воспользуемся этим методом, чтобы опять *разыграть* пользователя. Заставим кнопку беспрерывно ёрзать по экрану:

```
offset=-1
left=10
top= GraphicsWindow.Height-48
While "True"
  For i= 1 To 40
    left=left-offset
    top=top+offset
    Controls.Move(btnTimer, left, top)
    Program.Delay(10)
  EndFor
  offset= - offset
EndWhile
```

Вы уже, наверное, пробовали нажать нашу кнопку и были раздосадованы, что таймер не запускается, несмотря на обещание, написанное на кнопке. Правильно, обещания нужно выполнять, поэтому добавим к программе процедуру-обработчик события *ButtonClicked*, которое возникает при нажатии кнопки:

```
Controls.ButtonClicked = OnClick

Sub OnClick
  Controls.SetButtonCaption(btnTimer, "Остановить таймер")
EndSub
```

С помощью метода

```
Controls.SetButtonCaption(buttonName, caption)
```

мы легко можем *заменить надпись* на кнопке, чтобы она соответствовала новому состоянию программы (Рис. 25.3).

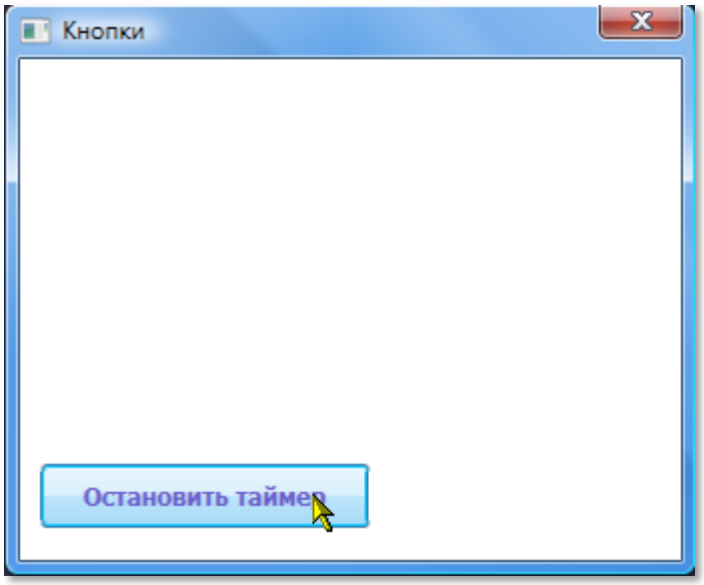


Рис. 25.3. Кнопка с новой надписью

Таким образом, нажав кнопку в первый раз, мы *запускаем* таймер, во второй раз – останавливаем его. Но вот незадача: после того как мы остановили таймер, должна вернуться первоначальная надпись, а этого не происходит. Исправляем оплошность:

```
Sub OnClick
  If Controls.GetButtonCaption(btnTimer) = "Запустить таймер"
then
    Controls.SetButtonCaption(btnTimer, "Остановить таймер")
  Else
    Controls.SetButtonCaption(btnTimer, "Запустить таймер")
  EndIf
EndSub
```

Теперь надписи на кнопке правильные. Осталось добавить *таймер*, который также является элементом управления, но *невизуальным (невидимым).*

```
Timer.Interval=1000
Timer.Pause()
Timer.Tick=OnTick
time=0
```

Свойство *Interval* отвечает за время срабатывания таймера. В данном случае мы установили 1000 миллисекунд, что равняется одной секунде. Это значит, что через каждую секунду будет про-

исходить событие *Tick*, которое мы сможем обработать в процедуре *OnTick*.

Поскольку значение свойства *Interval* задается в *миллисекундах*, а нам нужны секунды, то придётся его поделить на 1000. Затем прошедшее после нажатия на кнопку время мы выводим в заголовок окна приложения:

```
Sub OnTick
  time= time+ Timer.Interval/1000
  GraphicsWindow.Title= " Прошло: " + time
EndSub
```

Так как мы хотим, чтобы таймер начал отсчёт времени только после нажатия на кнопку, то сначала его нужно остановить методом *Pause*.

А вот подпрограмму-обработчик нажатия на кнопку придётся изменить:

```
Sub OnClick
  If Controls.GetButtonCaption(btnTimer) = "Запустить таймер"
then
    Controls.SetButtonCaption(btnTimer, "Остановить таймер")
    Timer.Resume()
  Else
    Controls.SetButtonCaption(btnTimer, "Запустить таймер")
    Timer.Pause()
  EndIf
EndSub
```

Метод *Resume* запускает таймер после остановки его методом *Pause*. Вот теперь наш таймер работает как часы (Рис. 25.4)!

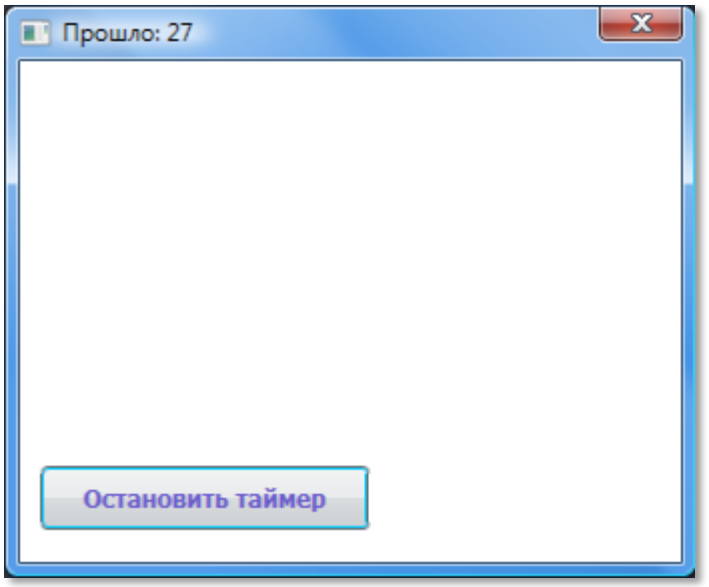


Рис. 25.4. Время пошло!

Выводить информацию, хотя бы и важную для нас, в заголовок окна вполне допустимо при отладке программы, но лучше использовать для этого элемент управления **текстовое поле** (или **метку**). Для того чтобы добавить *текстовое поле*, нужно вызвать метод

```
Controls.AddTextBox(left, top).
```

В скобках указывают координаты верхнего левого угла элемента управления:

```
txtTime=Controls.AddTextBox(10, 10)
```

Дополним таймерную процедуру одной строкой:

```
Sub OnTick
  time= time+ Timer.Interval/1000
  GraphicsWindow.Title= "Прошло: "  + time
  Controls.SetTextBoxText(txtTime, "Прошло: " + time)
EndSub
```

Метод

```
Controls.SetTextBoxText(textBoxName, text).
```

выводит текст в указанное *текстовое поле* (Рис. 25.5).

Помните, как нам пришлось стирать весь экран при выводе надписей в одно и то же место клиентской области окна? С *текстовым полем* у вас таких проблем не будет. Однако шрифт в *текстовом поле* изменить нельзя, поэтому для красивого вывода информации придётся использовать канву.

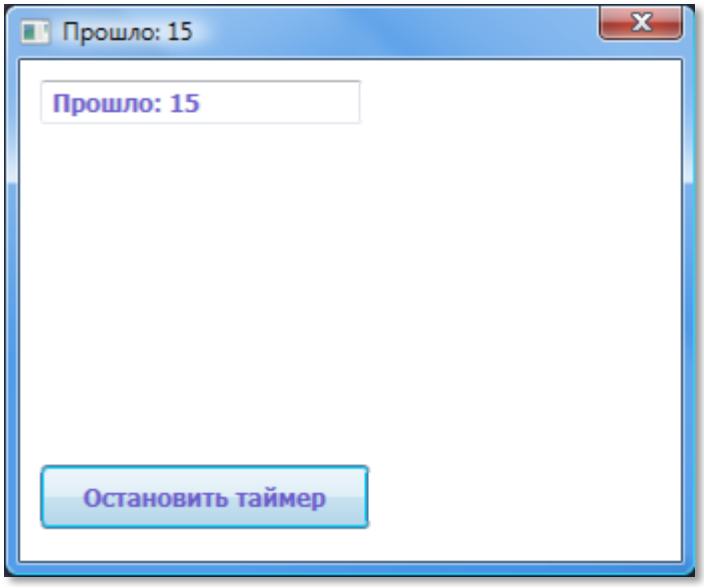


Рис. 25.5. Вот теперь совсем другое дело!

Вы можете установить *размеры текстового поля* такими же, как и кнопки:

```
Controls.SetSize(txtTime,164,32)
```

В *текстовое поле* можно *вводить информацию* с клавиатуры (Рис. 25.6).

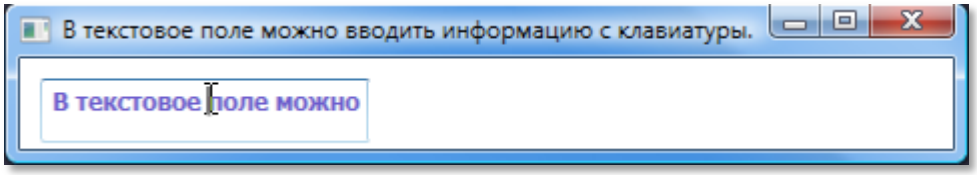


Рис. 25.6. Обратная связь с пользователем установлена!

А это очень важно, если мы хотим получить от пользователя программы какой-нибудь ответ или пароль.

Сейчас мы напишем маленькую программу, показывающую, как это можно сделать.

```
GraphicsWindow.Title= "Ввод в текстовое поле"

GraphicsWindow.Width= 320
GraphicsWindow.Height=240
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2

txtInfo=Controls.AddTextBox(10, 10)
Controls.SetSize(txtInfo,164,32)
Controls.TextTyped=OnText
```

Чтобы ввести текст, кликните мышкой внутри элемента управления. Так вы сделаете его *активным*, то есть передадите ему *фокус ввода*. При наборе текста возникает событие *TextTyped*, которое мы обрабатываем в процедуре *OnText*:

```
Sub OnText
  GraphicsWindow.Title= Controls.GetTextBoxText(txtInfo)
EndSub
```

Здесь с помощью метода *GetTextBoxText* мы получаем строку, записанную в *текстовом поле*, и выводим её в заголовок окна. Мы также можем присвоить это значение переменной

```
s = Controls.GetTextBoxText(txtInfo)
```

и затем использовать, как нам заблагорассудится.

Текстовое поле довольно «умное»: вы можете выделять в нём текст, редактировать его, копировать и вставлять в другое текстовое поле!

Вы, конечно, заметили, что в *текстовом поле* можно набрать строку любой длины, но при этом видна только ее часть. Чтобы прочитать длинную строку, придётся использовать клавиши со стрелками. Это не очень удобно.

```
txtInfo=Controls.AddMultiLineTextBox(10, 10)
Controls.SetSize(txtInfo,GraphicsWindow.Width-
20,GraphicsWindow.Height-20)
Controls.TextTyped=OnText

Sub OnText
  GraphicsWindow.Title= Controls.GetTextBoxText(txtInfo)
EndSub
```

Чтобы поместить элемент управления **многосторочное текстовое поле** на форму, мы воспользуемся методом *AddMultiLineTextBox(left, top)*, которому передаются два параметра – координаты верхнего левого угла элемента управления.

А скопируем-ка мы в него что-нибудь из *Пушкина* (Рис. 25.7).

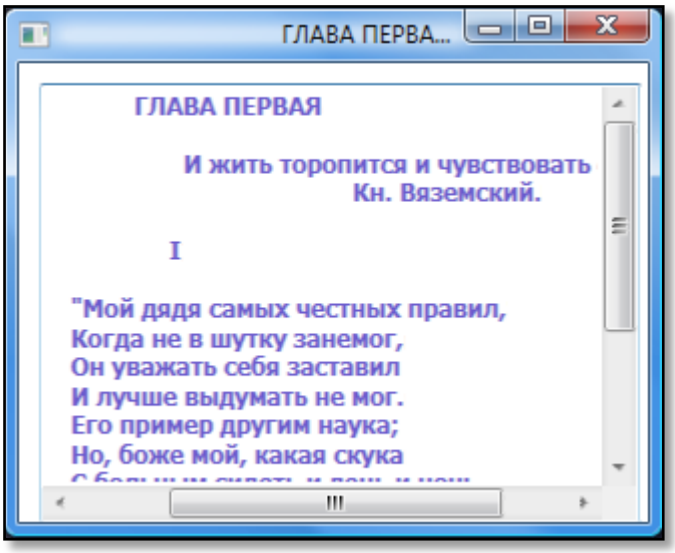


Рис. 25.7. Маловато будет!

Текст разбит на строки, но даже начало поэмы в *многострочное текстовое поле* не помещается. Но теперь весь текст легко просмотреть, пользуясь полосами прокрутки, что гораздо удобнее, чем клавишами со стрелками (Рис. 25.8).

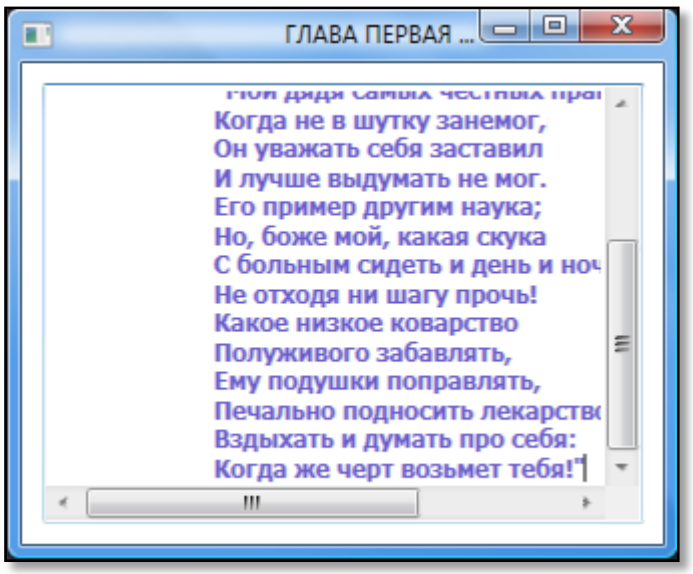


Рис. 25.8. Крути-верти!

Форматированный текст будет выведен, как обычный, поэтому забудьте о шрифтах, выравнивании и других изысках – здесь им места нет.

Дополнительные свойства и методы элементов управления

Заменим одну кнопку, которая отвечала за старт и остановку таймера, *двумя* (Рис. 25.9).

```
'Button1:
btnStart=Controls.AddButton("Запустить таймер", 8,
GraphicsWindow.Height-32)
'Button2:
btnStop=Controls.AddButton("Остановить таймер", 180,
GraphicsWindow.Height-32)
Controls.ButtonClicked=OnClick
```

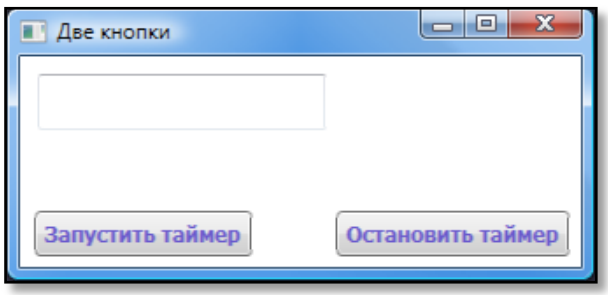


Рис. 25.9. Так удобнее?

Как говорится, одна кнопка хорошо, а две лучше. Однако правая кнопка сейчас лишняя, ведь таймер пока не запущен.

Дабы не искушать неискушенного пользователя, который обязательно нажмёт не ту кнопку и после будет роптать, уберём правую кнопку с глаз долой (рис. 25.10):

```
Controls.HideControl("Button2")
```

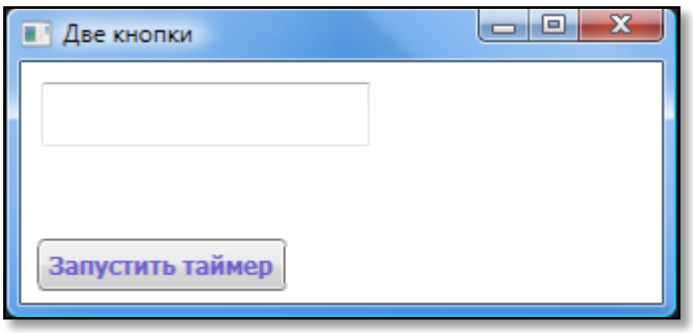


Рис. 25.10. Кнопка-невидимка

Для этого мы использовали метод *HideControl(controlName)*, которому передаётся название элемента управления. Как видите, оно не совпадает с тем именем, которое мы выбрали при создании кнопок. Все кнопки по умолчанию называются *Button#,* то есть они нумеруются от единицы в хронологическом порядке. Если сначала создать кнопку *btnStop*, а потом *btnStart*, то первой станет *btnStop*. Вы легко узнаете *название* кнопки, например, так:

```
  btn=Controls.LastClickedButton
  Controls.SetTextBoxText(txtTime," "Кнопка: " + btn)
```

Из кода видно, что для всех кнопок создаётся одна и та же процедура-обработчик *Controls.ButtonClicked=OnClick*. Возникает вопрос: как же мы узнаем, какая из двух кнопок была нажата? – Так как на кнопках *разные* надписи, то с помощью метода *GetButtonCaption* мы можем установить, что написано на кнопке, и так распознать её. Этим способом мы уже пользовались, поэтому поступим иначе. Свойство *LastClickedButton* содержит название последней нажатой кнопки: *Button1, Button2* и так далее. Зная это, перепишем обработчик для таймера:

```
Sub OnClick
  btn=Controls.LastClickedButton
  If btn="Button1" then
    Controls.HideControl("Button1")
    Controls.ShowControl("Button2")
    Timer.Resume()
  Else
    Controls.HideControl("Button2")
    Controls.ShowControl("Button1")
    Timer.Pause()
  EndIf
EndSub
```

В нём не только включается и выключается таймер, но и скрывается ненужная кнопка (Рис. 25.11).

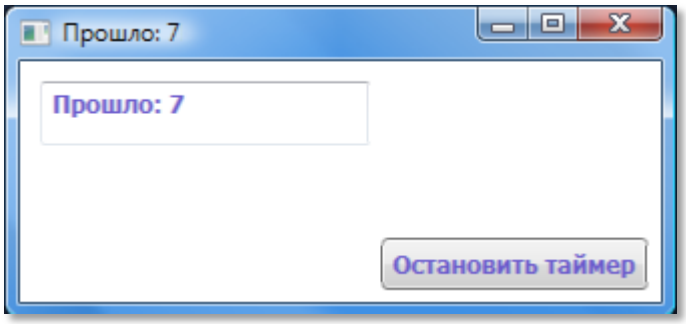


Рис. 25.11. Вторая кнопка-невидимка

Если на форме места для двух кнопок не хватает, можно их поместить одну над другой:

```
btnStop=Controls.AddButton("Остановить таймер", 8,
GraphicsWindow.Height-32)
```

Свойство *LastTypedTextBox* действует аналогично *LastClickedButton*, но при вводе в *текстовое поле*. Названия у текстовых полей такие: *TextBox1, TextBox2, …*

И последний метод *Remove(controlName) уничтожает* ненужный элемент управления в работающем приложении и восстановлению он не подлежит.

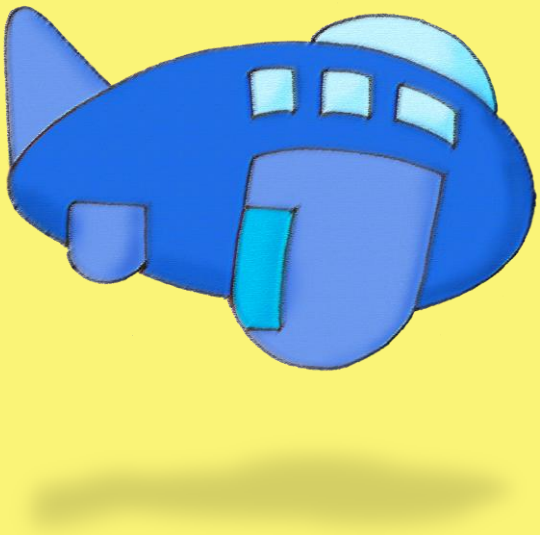
Исходный код программы находится в папке **Элементы управления**.

Урок 26. Занимательное моделирование

Природа говорит языком математики.

Галилео Галилей

Наверное, многим из вас приходилось собирать из отдельных деталей модели танков, самолётов, кораблей. Они похожи на настоящие образцы вооружений, но имеют меньший размер, сделаны из пластмассы и лишены многих полезных устройств и приспособлений, которыми обладают их прототипы. Однако эти модели умеют неплохо летать, плавать и преодолевать разнообразные препятствия на пересечённой местности. Таким образом, любая модель отражает не все, а только самые *важные* свойства и качества объекта. И чем точнее модель, чем она ближе к оригиналу, тем она сложнее и дороже (Рис. 26.1).



Подобные модели часто использовались при комбинированных съёмках кинофильмов. Несколько десятилетий назад, когда компьютеры ещё не годились для этих целей, приходилось мастерить «бутафорные» модели. Вы наверняка видели такие фильмы. В них летят под откос игрушечные составы, тонут в бассейне полуметровые корабли и горят в песочницах деревянные танки…

С давних времен модели применяются также в науке и технике. Прежде чем построить огромный мост, высоченную башню или скоростной самолёт, их уменьшенные копии трясут, мнут и обдувают потоком воздуха в аэродинамических трубах.

Но модели бывают не только реальные, сделанные из металла или пластмассы, но и виртуальные, которые существуют только в виде формул, которые описывают саму модель и её поведение. Такие модели называют *математическими*.

Поскольку компьютер с удовольствием считает по формулам, то мы вполне можем написать компьютерную программу, которая не только выдаст огромный массив чисел, показывающих поведение модели в различных условиях, но и покажет результаты этих расчётов в виде графиков, чертежей, рисунков и анимации. Такое представление модели в памяти компьютера и на экране монитора называется *компьютерным моделированием*.

Компьютерные модели более наглядны, чем математические, и позволяют найти новые закономерности, которые трудно выявить только по числовым данным эксперимента или расчетов по формулам.

На этом и следующем уроках мы создадим несколько простых, но занимательных биологических и физических моделей.

Рис. 26.1. Макет города Амстердама с двигающимися моделями самолетов

Кролики

Кролики - это не только ценный мех,
но и 4-5 килограммов вкусного,
легко усваиваемого мяса!

Юмористические Братцы-кролики

Первую математическую модель, связанную с кроликами, разработал известный нам Леонардо Фибоначчи в трактате *Книга абака*.

Он взял пару взрослых кроликов (точнее, кролика и крольчиху) и предположил, что они могут производить на свет потомство каждый месяц. Причем у них всегда рождается пара крольчат разного пола, у которых через два месяца также рождаются крольчата.

Поскольку все кролики приходятся близкими родственниками друг другу, то такая популяция обречена на быстрое вымирание, но в математической модели этот факт не учитывается.

Фибоначчи решил подсчитать, сколько будет кроликов через год, если за это время ни один кролик не умрет.

«Эксперимент» он начал в *январе*, когда у него была только *пара* кроликов.

В *феврале* семья пополнилась парой крольчат, итого кроликов стало *две пары*.

В *марте* родительская пара принесла ещё пару крольчат, а первым крольчатам исполнился месяц. Итак, в марте семья кроликов состояла из *трёх пар* кроликов.

В *апреле* родительская пара снова произвела на свет пару крольчат, а их первое потомство достигло половозрелого возраста и внесло свой вклад в дело размножения семьи, которая теперь насчитывала *5 пар*.

Просматривая этот сериал про семейство кроликов и дальше, мы сможем составить таблицу. Её часть вы можете видеть на Рис. 26.2. Полная таблица имеет невероятные размеры, ведь мы знаем, что в июле кроликов будет 21 пара, в августе – 34 пары, в сентябре – 55 пар, в октябре – 89 пар, в ноябре – 144 пары, в декабре – 233 пары. Поскольку цыплят считают по осени, а кроликов парами, то через год в вольере счастливое семейство кроликов будет насчитывать 233 пары кроликов. У первой пары кроликов появятся 11 пар внуков, 9 пар правнуков, и так далее.

Кролики и лисицы

- Гонялся, гонялся Братец Лис
за Братцем Кроликом, и так и этак ловчился,
чтобы его поймать.
А Кролик и так и этак ловчился,
чтобы Лис его не поймал.

Джоэль Харрис,
Братец Лис и Братец Кролик

А теперь мы разработаем более правдоподобную математическую модель, также связанную с кроликами.

Как мы видели, в модели Фибоначчи кроликам всегда было достаточно травы, чтобы питаться и неограниченно размножаться. Подобный случай наблюдался в середине XIX века, когда в Австралию завезли кроликов, у которых не оказалось ни серьёзных врагов, ни пищевых конкурентов.

Мы рассмотрим модель, достаточно благоприятную для кроликов, то есть будем считать, что травы для них вполне достаточно. Не будем ограничивать и продолжительность их жизни, но поселим рядом с ними *лисиц*, которые, в свою очередь, будут питаться кроликами.

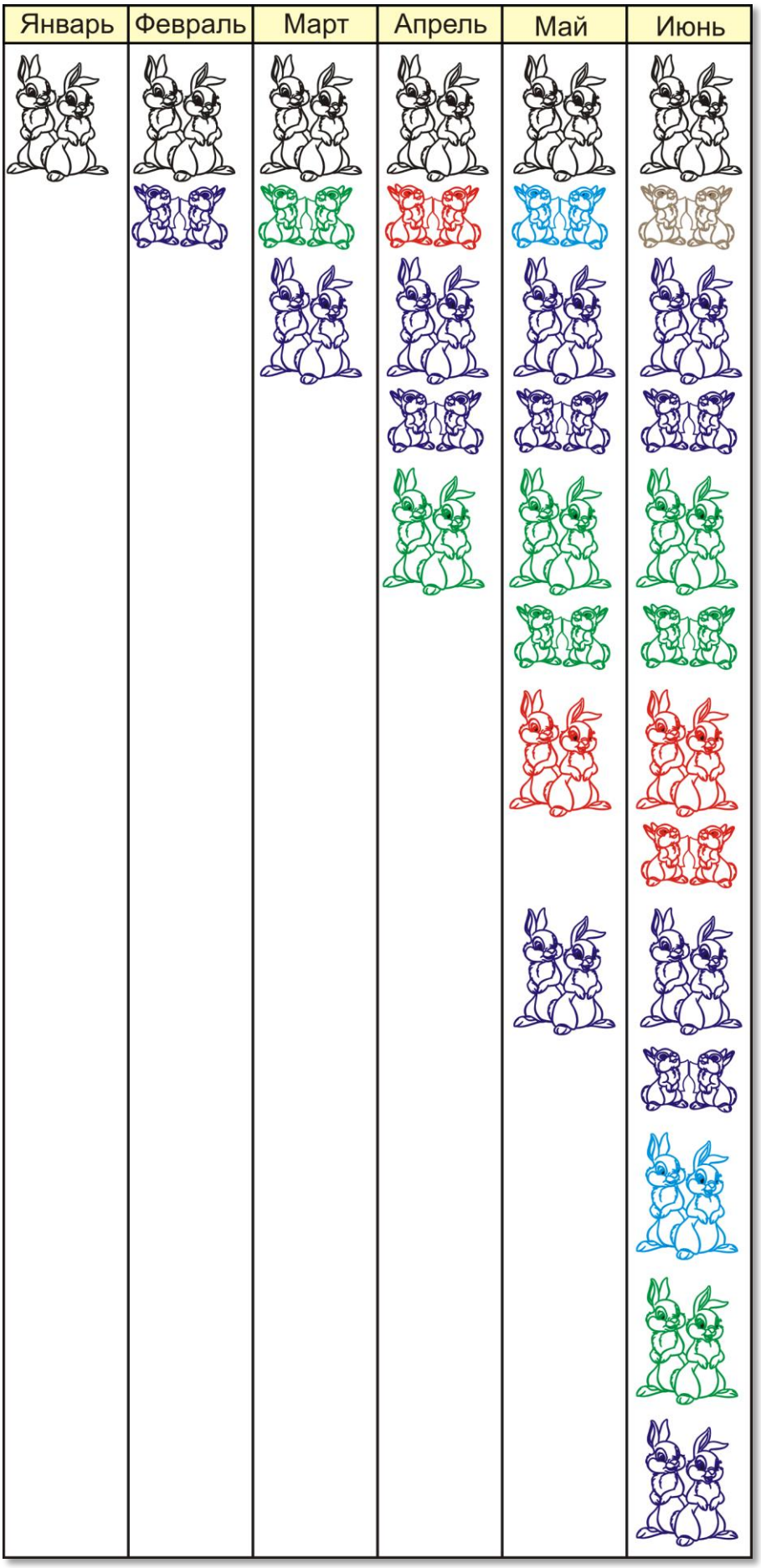


Рис. 26.2. Размножение фибоначчиевых кроликов

В начале эксперимента численность популяций кроликов и лисиц равна *nRabbit* и *nFox*, соответственно. Поскольку корма кроликам вполне хватает, они начинают усиленно размножаться, как им и предписано природой. Большое число кроликов приводит к тому, что ими может прокормиться и большее число лисиц, то есть вслед за ростом популяции кроликов начинает расти популяция лисиц. Они начинают поедать всё больше кроликов, а затем из-за недостатка «крольчатины» погибает всё больше лисиц. Но чем меньше становится лисиц, тем стемительнее размножаются кролики. Мы можем предположить, что численность популяций кроликов и лисиц будет изменяться *периодически*. Для проверки нашей гипотезы обратимся к математической модели этого процесса.

Она описывается системой дифференциальных уравнений, но это не должно вас пугать, так как компьютер решит их без проблем.

$$dR/dt = k_1GR - k_2RF$$
$$dF/dt = k_2RF - k_3F$$

Первое уравнение показывает, как со временем изменяется численность кроликов, а *второе* - лисиц.

Буквами *R*, *F* и *G* обозначено количество кроликов, лисиц и травы в неких условных единицах.

Коэффициенты k_1, k_2, k_3 определяют скорость размножения кроликов, скорость поедания кроликов лисицами и скорость вымирания лисиц из-за недостатка пищи.

В программе мы возьмём некоторые среднестатистические значения этих параметров, что ничуть не избавляет вас от необходимости проверить на практике и другие значения этих параметров.

```
'ПРОГРАММА ДЛЯ МОДЕЛИРОВАНИЯ
'ИЗМЕНЕНИЯ ЧИСЛЕННОСТИ ПОПУЛЯЦИЙ

'var
```

```
'коэффициенты:
Q1= 1   'k3/k2
Q2= 1   'k1/k2
'относительное количество травы:
Grass= 1
'относительное количество кроликов и лисиц:
nRabbit= 0.44
nFox= 0.40
'число циклов наблюдения:
D= 4000
'шаг по оси времени:
T=1

GraphicsWindow.Title="Кролики и Лисицы"

GraphicsWindow.Width= 1000
GraphicsWindow.Height=320
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
height=GraphicsWindow.Height
width=GraphicsWindow.Width
```

После объявления переменных и подготовки окна приложения мы готовим данные для изображения жизненного цикла популяций:

```
'начальные популяции кроликов и лисиц:
x= nRabbit
y= nFox
```

А эти переменные укажут нам максимальное и минимальное число особей каждого вида за время наблюдений:

```
minx= 1000000
maxx=0
miny= 1000000
maxy=0
```

Свои наблюдения мы начнём с определения количества точек, по которым будет построен график:

```
k= Q1/(Q2*Grass)
'число циклов:
n= 1 + d/t
```

Поскольку число циклов нам известно точно, то в цикле *For* мы определяем для каждого цикла численность обеих популяций. Из любопытства отслеживаем предельную численность популяций за всё время наблюдения:

```
'циклы наблюдения:
For j= 1 to n
  'вычисляем новые популяции:
  x2= x + (1 - y) * x * t/100
  'dt= t/100 - приращение времени при интегрировании
  y2= y - (1 - x) * y * k * t/100
  'новые популяции становится текущими:
  x= x2
  y= y2
  'печатаем текущие значения популяций:
  draw()

  If (x< minx) Then
    minx=x
  EndIf
  If (x> maxx) Then
    maxx=x
  EndIf
  If (y< miny) Then
    miny=y
  EndIf
  If (y> maxy) Then
    maxy=y
  EndIf
endFor
```

Числовые данные для наглядности преобразуем в график, а для его большей информативности добавляем линии, соответствующие начальным популяциям кроликов и лисиц, а также вспомогательные прямые для отсчёта текущих параметров популяций:

```
'чертим прямые, соответствующие начальным популяциям:
GraphicsWindow.PenWidth=3
```

```
GraphicsWindow.PenColor="Green"
GraphicsWindow.DrawLine(0, height-nRabbit*zoom, width, height-
nRabbit*zoom)
GraphicsWindow.PenColor="Red"
GraphicsWindow.DrawLine(0, height-nFox*zoom, width, height-
nFox*zoom)
'чертим прямые через 0.1 от начальной популяции:
GraphicsWindow.PenWidth=1
GraphicsWindow.PenColor="Gray"
For i= 0 to maxy Step 0.10
  GraphicsWindow.DrawLine(0, height-i*zoom, width, height-
i*zoom)
endFor

'печатаем в Текстовом окне макс. и мин. значения популяций:
TextWindow.WriteLine("minx= " + minx)
TextWindow.WriteLine("maxx= " + maxx)
TextWindow.WriteLine("miny= " + miny)
TextWindow.WriteLine("maxy= " + maxy)

'Строим график
Sub draw
  zoom=100
  'ставим "точки":
  GraphicsWindow.BrushColor="Green"
  GraphicsWindow.FillEllipse(j/4,height-x*zoom, 3,3)
  GraphicsWindow.BrushColor="Red"
  GraphicsWindow.FillEllipse(j/4,height-y*zoom, 3,3)
endSub
```

Запустив программу, мы воочию убедимся, что наше предположение о циклическом изменении численности популяции оправдалось (Рис. 26.3). Более того, на графике хорошо видно, что кривая численности лисиц смещена вправо относительно кривой для кроликов, то есть популяция лисиц растёт и уменьшается вслед за изменением популяции кроликов с некоторым опозданием, что также правильно объяснила наша модель.

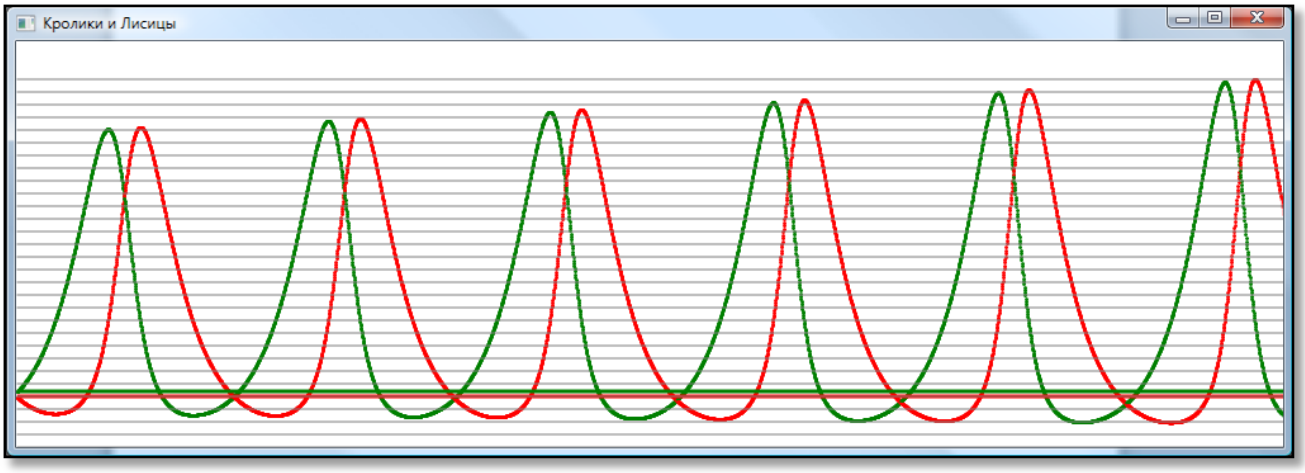


Рис. 26.3. Изменение численности кроликов (**зелёные** линии) и лисиц (**красные** линии)

В *текстовом окне* мы найдём информацию о предельной численности популяций за время наблюдения (Рис. 26.4). Как видите, численность и кроликов и лисиц уменьшается более чем в два раза, а растёт почти в семь раз. То есть размах колебаний численности популяций 14-кратный.

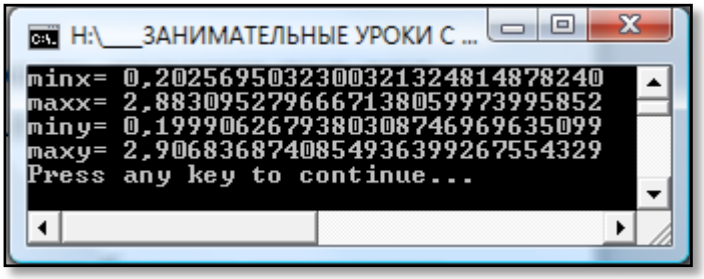
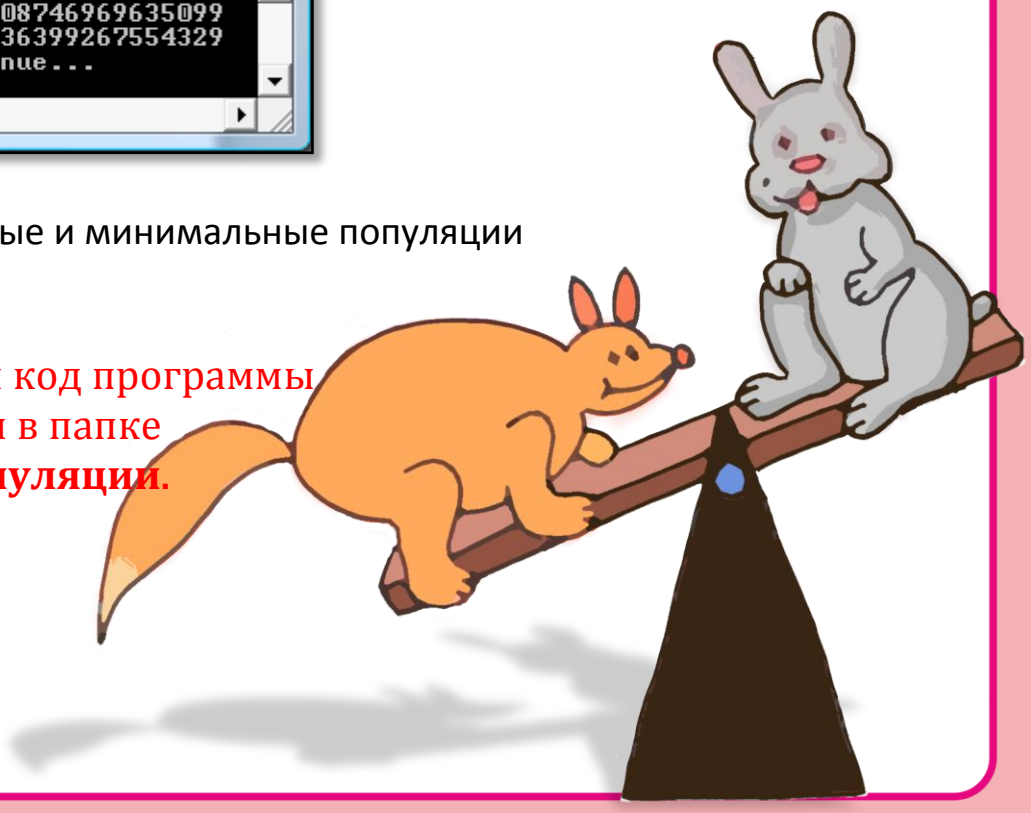


Рис. 26.4. Максимальные и минимальные популяции

Исходный код программы находится в папке **Популяции**.



Урок 27. Занимательная физика

Вплоть до XVII в. учёные были уверены, что скорость падения тел определяется их массой, то есть тяжёлые предметы падают быстрее лёгких.

Действительно, если мы бросим с одинаковой высоты железную гирю и скомканный кусок бумаги, то убедимся в правоте этого суждения, которое, кстати говоря, высказал великий учёный древности Аристотель. Его авторитет в научной среде был настолько велик, что никому и в голову не приходило тщательно проверить его утверждение экспериментально.

Так как тела движутся в атмосфере Земли, то при падении на них действует сила сопротивления воздуха, что и приводит к таким результатам. Однако если тела имеют одинаковую форму, сила сопротивления воздуха будет сопоставима, и падать они будут практически с одинаковой скоростью.

Так продолжалось более двух тысяч лет, пока великий итальянский учёный *Галилео Галилей* не опроверг теорию Аристотеля. Согласно легенде, он одновременно сбрасывал с Пизанской башни тяжёлое пушечное ядро и гораздо более лёгкую мушкетную пулю. Оказалось, что оба предмета одновременно достигали земли, то есть скорость их падения была одинакова.

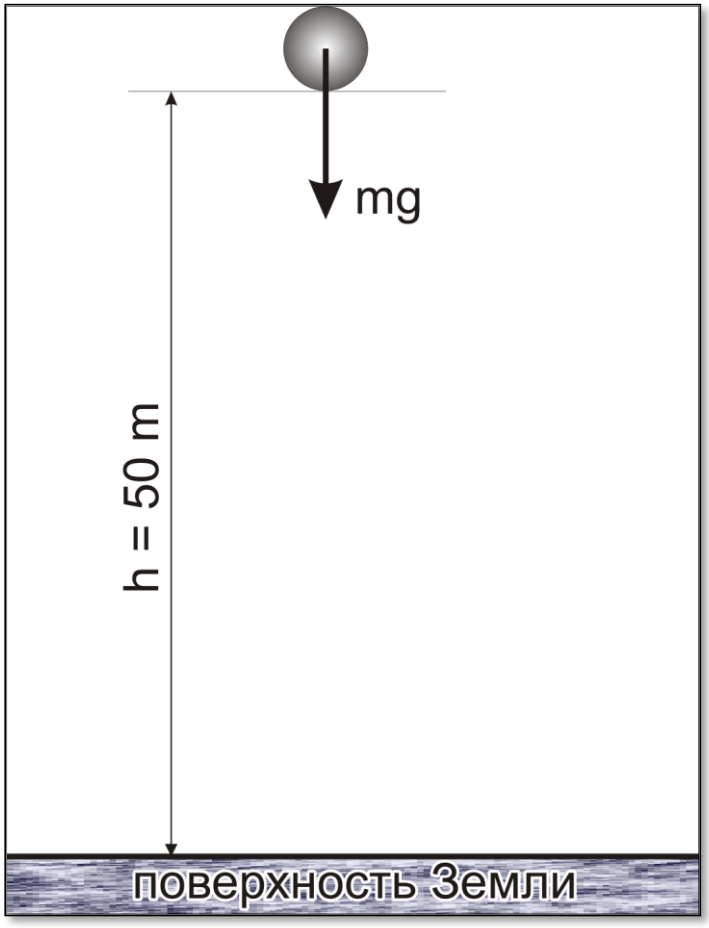
На самом деле Галилей провёл свой эксперимент гораздо «хитрее», ему не пришлось таскать тяжёлые ядра на вершину Пизанской башни. Он просто скатывал шары по наклонной доске и установил, что время скатывания шаров не зависит от их массы, причём этот вывод справедлив для разных углов наклона доски, из чего он и сделал вывод о том, что и при вертикальном падении объекты разной массы будут падать с одинаковой скоростью.

Более того, он опроверг и другое утверждение Аристотеля - что под воздействием силы тяжести тела движутся с постоянной скоростью. Его эксперименты показали, что шары катятся с *уско-*

рением. Если за первую секунду они прокатятся 1 метр (естественно, метров в то время ещё не было, но это не имеет значения), то за две - 4 метра, за три – 9 метров, и так далее, то есть скорость скатывания шаров увеличивается пропорционально времени.

Моделируем свободное падение

Мы не будем опровергать красивую легенду, а заберёмся на Пизанскую башню и сбросим оттуда шар. Общая высота башни составляет около 57 метров, но нам удобнее сбрасывать шары не с крыши, а с верхнего этажа, то есть с высоты примерно 50 метров (Рис. 27.1).



h – расстояние до поверхности Земли

mg – сила тяжести

Рис. 27.1. Шар в исходном положении

Для упрощения модели мы будем считать, что тело падает в безвоздушном пространстве, чтобы не учитывать сопротивление

воздуха при падении. Таким образом, наша модель больше годится для Луны, чем для Земли с её плотной атмосферой.

В этом случае все тела независимо от их массы должны падать с одинаковым ускорением, которое в физике принято обозначать буквой *g*. Его называют *ускорением свободного падения*. Вблизи поверхности Земли оно равно примерно 9,8 м/с^2.

Тогда скорость падения в зависимости от времени можно записать так:

$v_t = v_0 + gt$, где

v_0 – скорость падения тела в начале эксперимента, то есть на высоте *h* метров от поверхности Земли. Мы же собираемся просто выпустить шар из рук, поэтому начальная скорость будет равна нулю, а уравнение ещё больше упростится:

$v_t = gt$

Путь, пройденный телом при падении, легко рассчитать по формуле:

$S_t = v_0t + ½ gt^2$,

а при $v_0 = 0$:

$S_t = ½ gt^2$

Итак, в начальный момент времени $t = 0$ шар находится на расстоянии *h* метров от поверхности. Когда шар упадет на землю, он пролетит $S = h$ метров. Откуда

$½ gt^2 = h$, а *время падения* составит

$t = \sqrt{2h/g}$

Для высоты 50 метров это время равно 3,19 секунды.

Модель падения шара готова, приступаем к её компьютерной реализации.

Начните новый проект и сохраните его в папке **Fall**.

Нам потребуется в программе только одна *константа* – ускорение свободного падения:

```
'МОДЕЛЬ ПАДЕНИЯ ПРЕДМЕТОВ
'НА ЗЕМНУЮ ПОВЕРХНОСТЬ

GraphicsWindow.Title="Свободное падение"

'const
'ускорение свободного падения:
g= 9.8
```

Введём также несколько *переменных*, назначение которых очевидно:

```
'var
'смещение шара от края окна в пикселях:
xShar= 322
'начальная высота шара в пикселях:
hpix0= 176
'начальная высота шара в метрах:
h= 50
'диаметр шара в пикселях:
dShar= 20
'уровень поверхности Земли в пикселях:
yZemli= 660
```

Значения в пикселях выбраны, исходя из фотографии Пизанской башни, которую мы будем использовать в качестве фона (Рис. 27.2).

Далее мы настраиваем *окно* приложения и загружаем *фоновую картинку* из файла:

```
'окно приложения:
GraphicsWindow.Hide()
```

```
GraphicsWindow.Width= 438
GraphicsWindow.Height=667
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"

height=GraphicsWindow.Height
width= GraphicsWindow.Width

'фоновая картинка:
background = ImageList.LoadImage(Program.Directory +
"/pisa.jpg")
GraphicsWindow.DrawImage(background, 0, 0)

'шрифт:
GraphicsWindow.BrushColor="Blue"
GraphicsWindow.FontBold="True"
GraphicsWindow.FontSize=16
```

Добавим и толику *элементов управления*, чтобы придать компьютерному эксперименту больше наглядности:

```
'КНОПКА

'Начать эксперимент:
btnStart=Controls.AddButton("БРОСИТЬ!", 10, height-64)
Controls.ButtonClicked=OnClick

'ТЕКСТОВЫЕ ПОЛЯ

'Время падения:
txtTIme=Controls.AddTextBox(10, 10)
Controls.SetSize(txtTime,100,26)
GraphicsWindow.DrawText(120, 14, "<- Время падения")

'Пройденное расстояние:
txtRasst=Controls.AddTextBox(10, 48)
Controls.SetSize(txtRasst,100,26)
GraphicsWindow.DrawText(120, 52, "<- Пройденное расстояние")

GraphicsWindow.BrushColor="Red"
GraphicsWindow.Show()
```

На фотографии *уровень земли* виден не очень хорошо, поэтому прочерчиваем **толстую жёлтую линию**, которая будет обозначать поверхность Земли:

```
'рисуем поверхность Земли:
GraphicsWindow.PenWidth=5
GraphicsWindow.PenColor="Yellow"
GraphicsWindow.DrawLine(0, yZemli, 480, yZemli)
```

Осталось нарисовать *шар* и запустить программу.

```
'рисуем шар в исходном положении:
GraphicsWindow.BrushColor="Red"
shar=Shapes.AddEllipse(dShar,dShar)
drawShar()
```

К эксперименту мы подготовились отлично (Рис. 27.2), сам Галилей был бы доволен нашей работой!

Для рисования шара на экране мы используем фигуру *shar* - эллипс класса *Shapes*, так как его легко перерисовывать в разных положениях во время падения. Дополнительно мы будем отмечать *жёлтым кружком* текущее положение шара, чтобы нам было удобнее следить за его ускоренным падением.

```
Sub drawShar
  'ht - текущая высота шара в метрах
  'расстоянию в 50 метров соответствуют на экране htpix
пикселей:
  htpix= yZemli - hpix0 - dShar
  'коэффициент пересчета метров в пиксели =
  'число пикселей, приходящихся на 1 метр:
  kpix= htpix/h
  'текущая высота шара в пикселях на канве:
  ypix= hpix0 + kpix*s

  'рисуем шар в текущем положении:
  Shapes.Move(shar, xShar, ypix)

  'отмечаем текущее положение шара кружком:
  If (Math.Remainder(n,10)=0) Then
    GraphicsWindow.BrushColor= "Yellow"
```

```
    GraphicsWindow.FillEllipse(xShar+7, ypix, 6, 6)
  EndIf
  'время падения шара:
  ds_t= Math.Floor(t*100)/100
  Controls.SetTextBoxText(txtTime, ds_t)
  'пройденное расстояние:
  ds_s= g * t * t /2
  ds_s= Math.Floor(ds_s*100)/100
  Controls.SetTextBoxText(txtRasst, ds_s)
EndSub
```

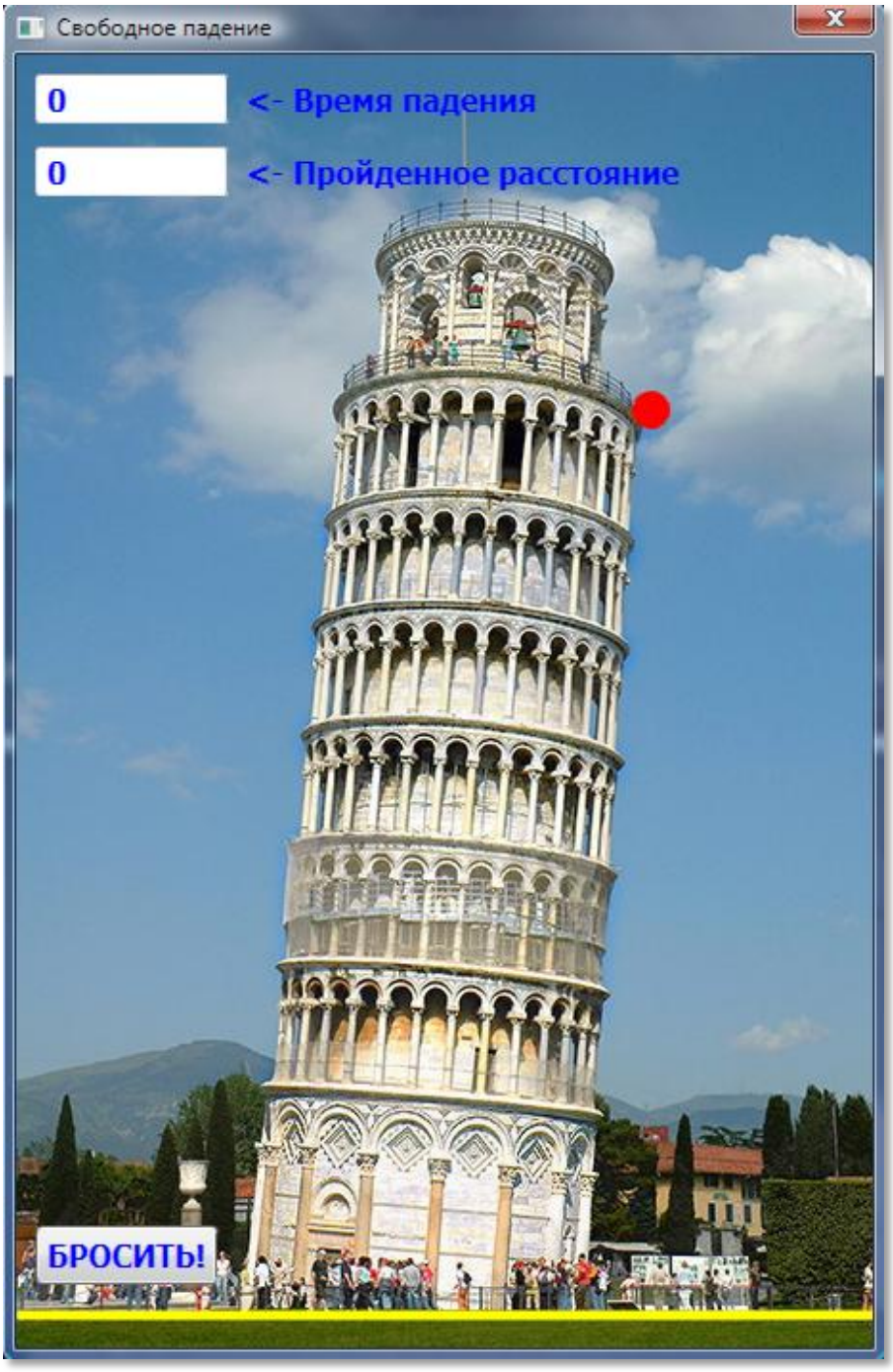


Рис. 27.2. Шар на Пизанской башне

Единственная сложность в процедуре рисования шара – это пересчёт метров реального мира в пиксели виртуального. Кроме того, следует учесть, что при падении шара его вертикальная координата *увеличивается*, хотя расстояние до земли *уменьшается*.

Однако пора нажать на кнопку!

```
'Бросаем шар
Sub OnClick
  fall()
EndSub
```

Шар начинает медленно, но с ускорением падать на землю:

```
'Падение шара
Sub fall
  t=0
  'приращение времени в секундах:
  dt= 0.01
  'текущая высота шара в метрах:
  ht= h
  n= 0
  While ("True")
    'пройденное расстояние:
    s= g*t*t/2
    If (s >= h) Then 'шар упал на землю
      s = h
      t= Math.SquareRoot(2*50/g)
      drawShar()
      Sound.PlayClick()
      Goto exit
    EndIf
    drawShar()
    t= t+ dt
    Program.Delay(10)
    n= n +1
  EndWhile
  exit:
EndSub
```

Каждую сотую долю секунды мы вычисляем новое положение шара и рисуем его на экране. Из-за того что реальный шар проле-

тит 50 метров, а наш, нарисованный, всего несколько сотен пикселей, нам придётся отмечать его текущее положение только один раз из десяти, иначе все кружочки сольются в одну линию и полностью лишат нас научной информации о ходе эксперимента. Для этого мы ввели вспомогательную переменную-счётчик *n*. Также в конце каждого цикла мы делаем задержку на ту же одну сотую долю секунды, чтобы наш шар падал синхронно с настоящим:

```
Program.Delay(10)
```

Когда пройденное шаром расстояние станет равным начальной высоте шара, мы должны закончить цикл, так как ниже поверхности земли шар двигаться не может (правда, вмятину в земле он сделает, но к нашей модели это отношения не имеет). Звук падения шара на землю мы имитируем вот таким звуком:

```
Sound.PlayClick()
```

Достаточно посмотреть на снимок экрана после завершения эксперимента (Рис. 27.3), чтобы убедиться: наша модель адекватно описывает свободное падение шара на землю.

Во время эксперимента никто не пострадал!

Исходный код программы находится в папке **Fall**.

Куда подальше!

Предположим, что мы не просто отпускаем шар в свободное падение, но и бросаем его с некоторой *горизонтальной* скоростью v_h. Поскольку вертикальная составляющая скорости при этом не изменится, то время падения шара останется прежним, то есть 3,19 секунды. Но в отсутствие воздуха шар за это время пролетит в горизонтальном направлении $v_h \times t$ метров, то есть упадёт не к

подножию башни, а на расстоянии v_h x t метров от него. При этом шар будет двигаться не по вертикальной прямой, а по нисходящей ветви *параболы* (Рис. 27.4).

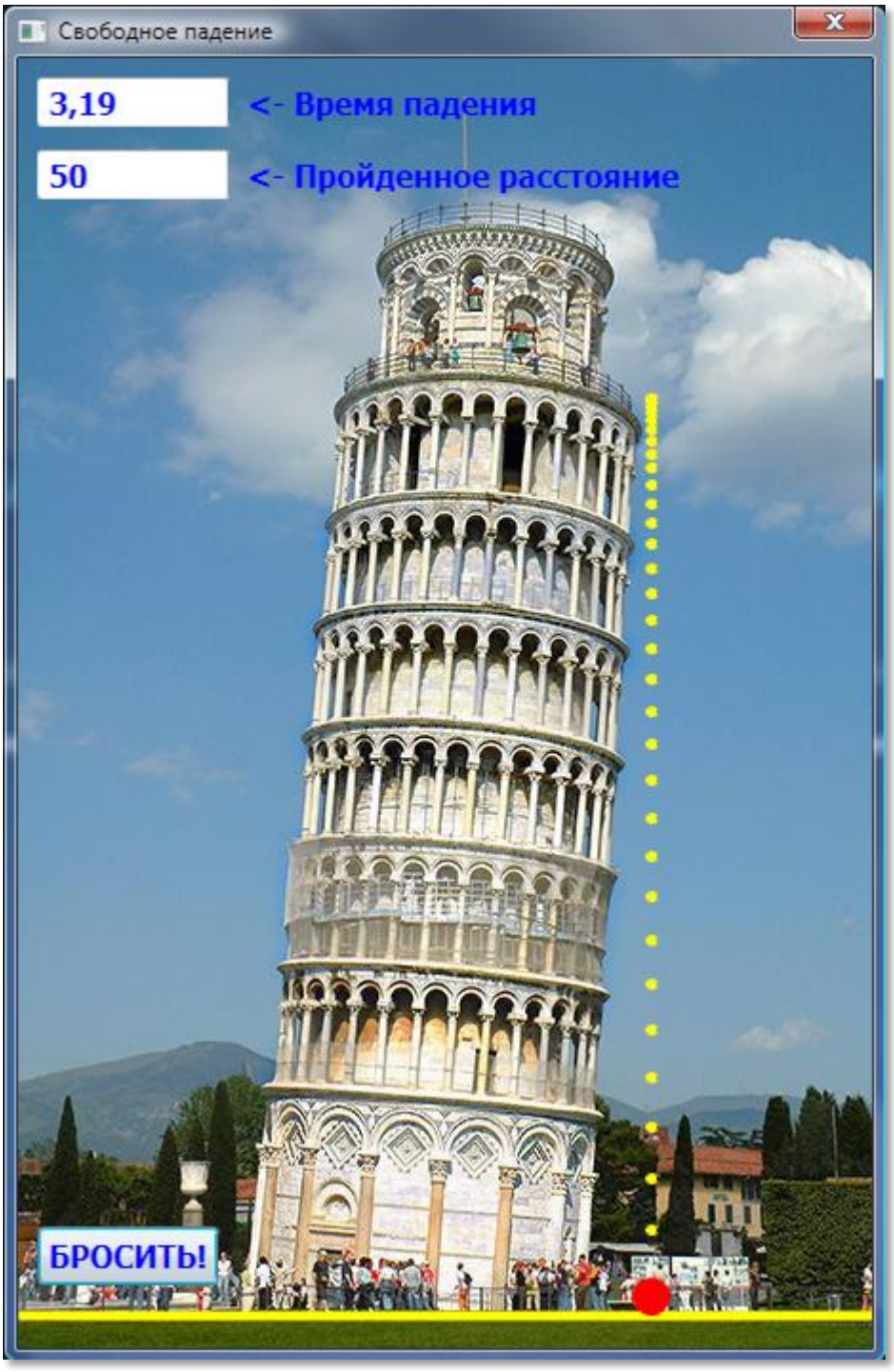


Рис. 27.3. Эксперимент закончен

Перепишем исходный код предыдущего проекта в папку **Fall2** и тут же примемся за дело.

Добавим новую *переменную*, отвечающую за горизонтальную скорость шара:

```
'горизонтальная скорость шара, метров в секунду:
vh= 10
```

Увеличим ширину окна приложения

```
GraphicsWindow.Width= 688
```

под новую картинку:

```
'фоновая картинка:
background = ImageList.LoadImage(Program.Directory +
"/pisa2.jpg")
```

В самом деле, если мы не добавим нашему шару жизненного пространства справа, то он просто улетит за пределы окна и эксперимент получится неполноценным!

Для слежения за расстоянием, которое пролетит шар в горизонтальном направлении, нам потребуется ещё одно *текстовое поле*:

```
txtRasstH=Controls.AddTextBox(10, 86)
Controls.SetSize(txtRasstH,100,26)
GraphicsWindow.DrawText(120, 90, "<- Пройденное расстояние по
горизонтали")
```

Продлим *линию* поверхности земли вправо:

```
GraphicsWindow.DrawLine(0, yZemli, 688, yZemli)
```

И последнее, что нам следует сделать, – дополнить процедуру рисования шара:

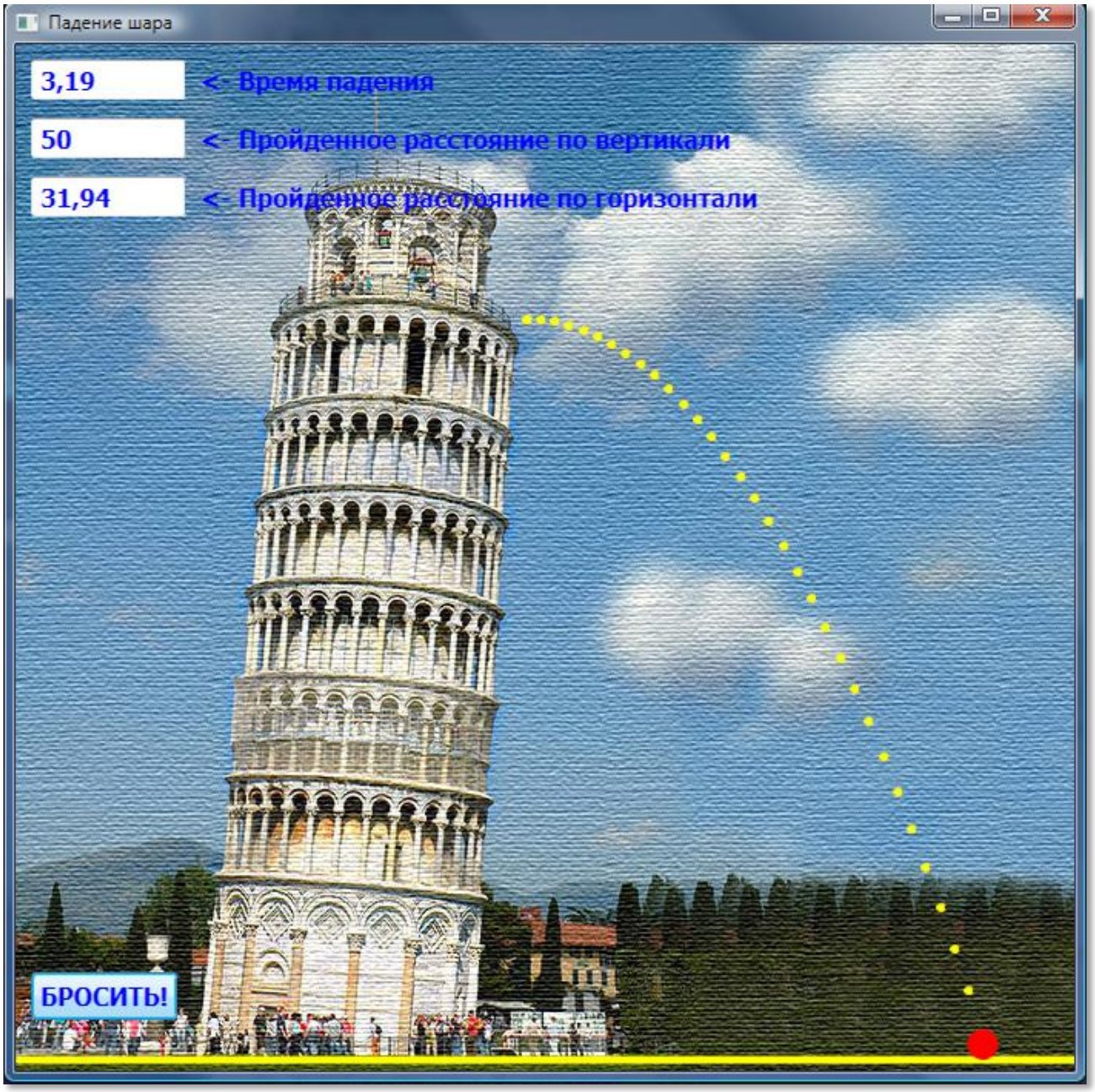


Рис. 27.4. Падение шара по параболе.

```
Sub drawShar
. . .

  'горизонтальная координата шара в пикселях:
  xpix= xShar + kpix* (vh* t)
  'рисуем шар в текущем положении:
  Shapes.Move(shar, xpix, ypix)

  'отмечаем текущее положение шара кружком:
  If (Math.Remainder(n,10)=0) Then
```

```
    GraphicsWindow.BrushColor= "Yellow"
    GraphicsWindow.FillEllipse(xpix+7, ypix, 6, 6)
  EndIf
. . .
  'пройденное расстояние по горизонтали:
  ds_h= vh * t
  ds_h= Math.Floor(ds_h*100)/100
  Controls.SetTextBoxText(txtRasstH, ds_h)
EndSub
```

Как хотите, но теперь шар падает значительно красивее. Бросал бы его и бросал!

Исходный код программы находится в папке **Fall2**.

Стреляем ядрами

Давайте ещё более усложним нашу модель: поднатужившись, бросим шар не горизонтально, а под некоторым *углом* вверх-вправо.

В новом проекте **Fall3** добавим *переменные* для этого случая:

```
'угол броска в градусах:
alpha= 60
'начальная скорость шара, метров в секунду:
v0= 15
```

Из прямоугольного треугольника (Рис. 27.5) мы легко найдём вертикальную и горизонтальную составляющие скорости шара в начальный момент времени:

```
'угол броска в радианах:
alphaR= Math.GetRadians(alpha)
'горизонтальная скорость шара, метров в секунду:
vh= v0 * Math.Cos(alphaR)
'начальная вертикальная скорость шара, метров в секунду:
vv= v0 * Math.Sin(alphaR)
```

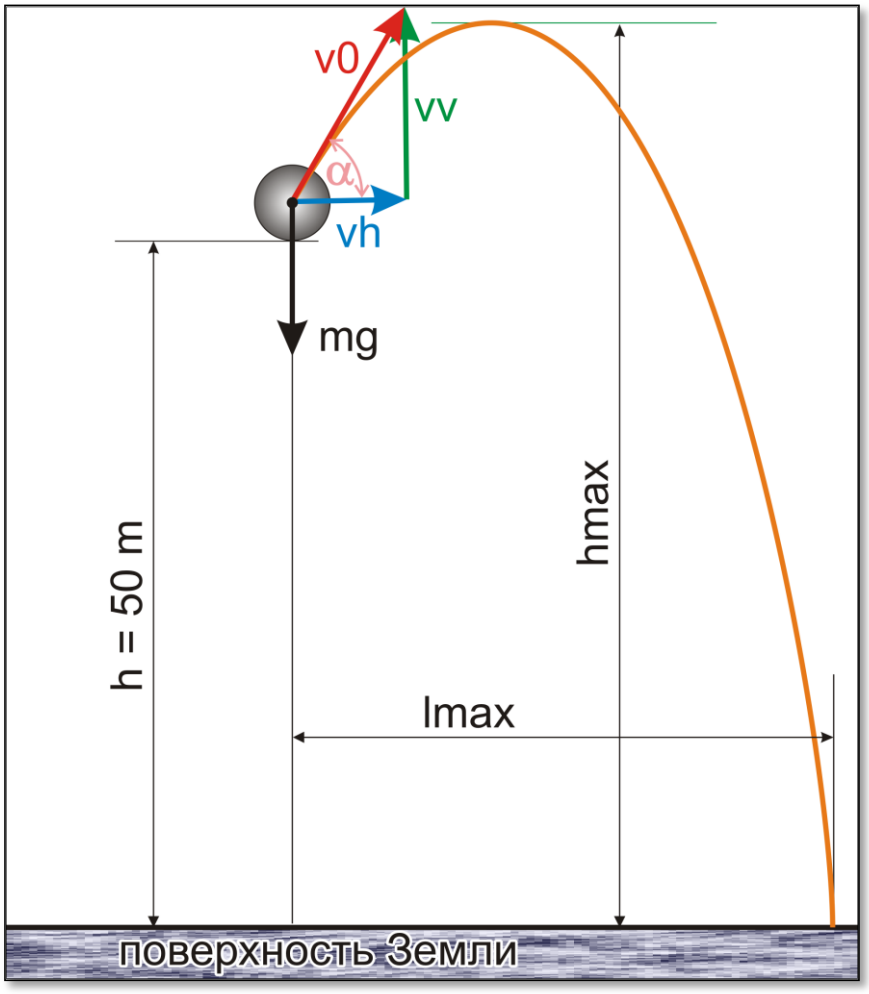


Рис. 27.5. Траектория полета шара, брошенного под углом к горизонту

```
'Падение шара
Sub fall
. . .
```

Поскольку теперь шар сначала поднимается вверх, а затем падает вниз с максимальной высоты, то прежняя формула расчёта вертикального пути не годится. Мы будем суммировать отдельные расстояния, которые пролетит шар по вертикали за время *dt*:

```
sv= sv + Math.Abs((vv - g*t) * dt),
```

принимая во внимание, что текущая вертикальная скорость равна

$vv_t = vv - gt$

```
  'путь по вертикали:
  sv=0
  While ("True")
```

Иначе придётся рассчитывать и текущую высоту шара над поверхностью земли:

```
    'текущая высота:
    curY= h - g*t*t/2 + vv*t
    sv= sv + Math.Abs((vv - g*t) * dt)
```

Условие падения шара на землю станет более естественным:

```
    If (curY <= 0) Then 'шар упал на землю
      curY = 0
      drawShar()
      Sound.PlayClick()
      Goto exit
    EndIf
    drawShar()
    t= t+ dt
    Program.Delay(10)
    n= n +1
  EndWhile
  exit:
EndSub
```

Так как мы изменили формулу для расчёта высоты шара, нам необходимо внести поправки и в процедуру рисования шара:

```
Sub drawShar
. . .
  kpix= htpix/h
  'текущая высота шара в пикселях на канве:
    ypix= yZemli- dShar - kpix* curY
  'горизонтальная координата шара в пикселях:
  xpix= xShar + kpix* (vh* t)
  'рисуем шар в текущем положении:
  Shapes.Move(shar, xpix, ypix)

  . . .
  'пройденное расстояние по вертикали:
  ds_s= sv
```

```
  ds_s= Math.Floor(ds_s*100)/100
  Controls.SetTextBoxText(txtRasst, ds_s)
  'пройденное расстояние по горизонтали:
  ds_h= vh * t
  ds_h= Math.Floor(ds_h*100)/100
  Controls.SetTextBoxText(txtRasstH, ds_h)
EndSub
```

Запускаем программу и убеждаемся, что модель работает верно (Рис. 27.6).

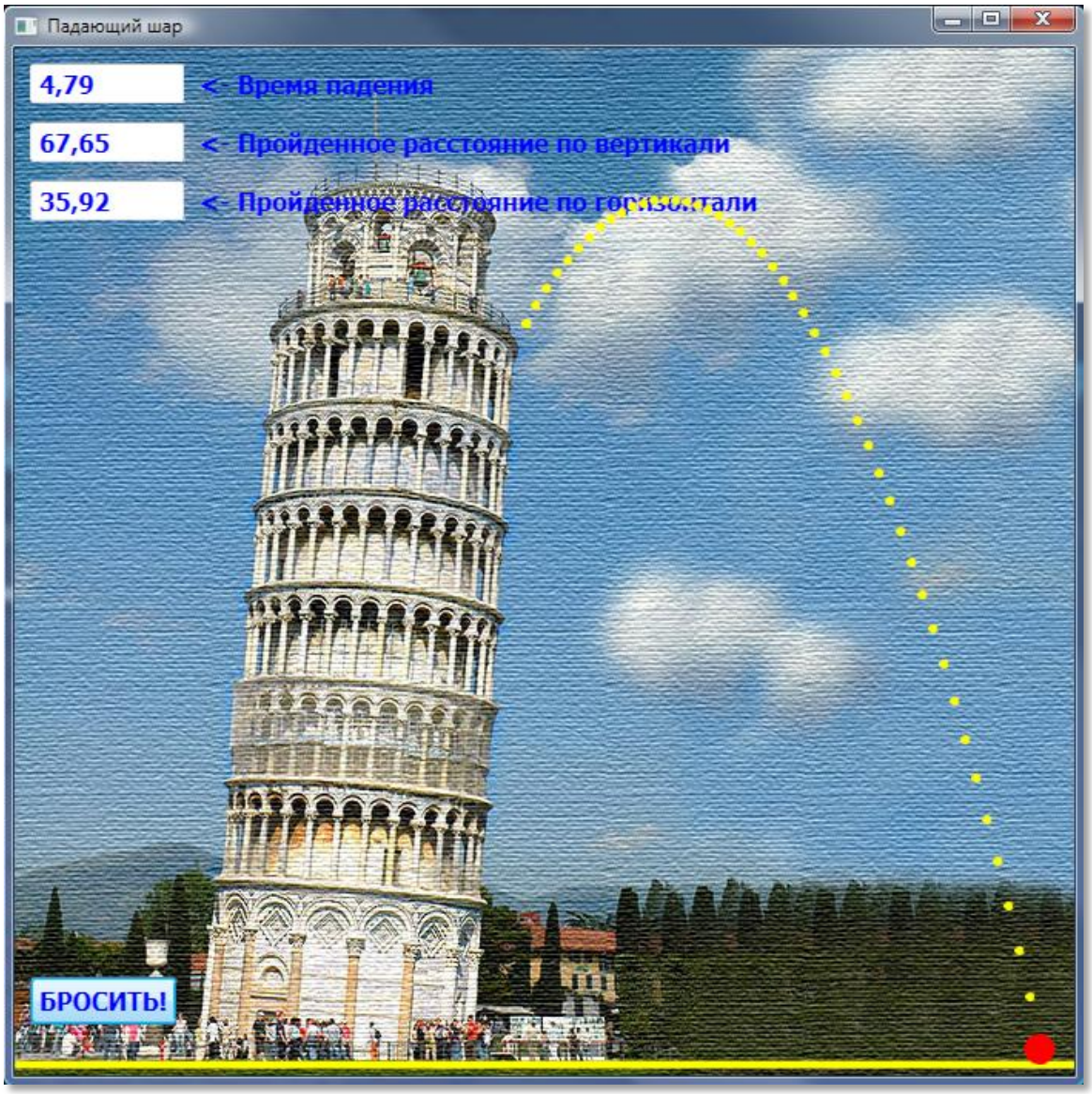


Рис. 27.6. Падение шара, брошенного под углом к горизонту

Исходный код программы находится в папке **Fall3**.

1. Ускорение свободного падения на Луне равно 1,624 м/с^2, а на Юпитере – 24,8 м/с^2. Смоделируйте падение шара на Луне, на Юпитере и на других планетах Солнечной системы!

2. Измените код программы *Fall3* так, чтобы шар можно было бы бросать с земли, а не с башни. Например, при начальных условиях

```
'угол броска в градусах:
alpha= 75
'начальная скорость шара, метров в секунду:
v0= 35
```

получится вот такая замечательная парабола (Рис. 27.7).

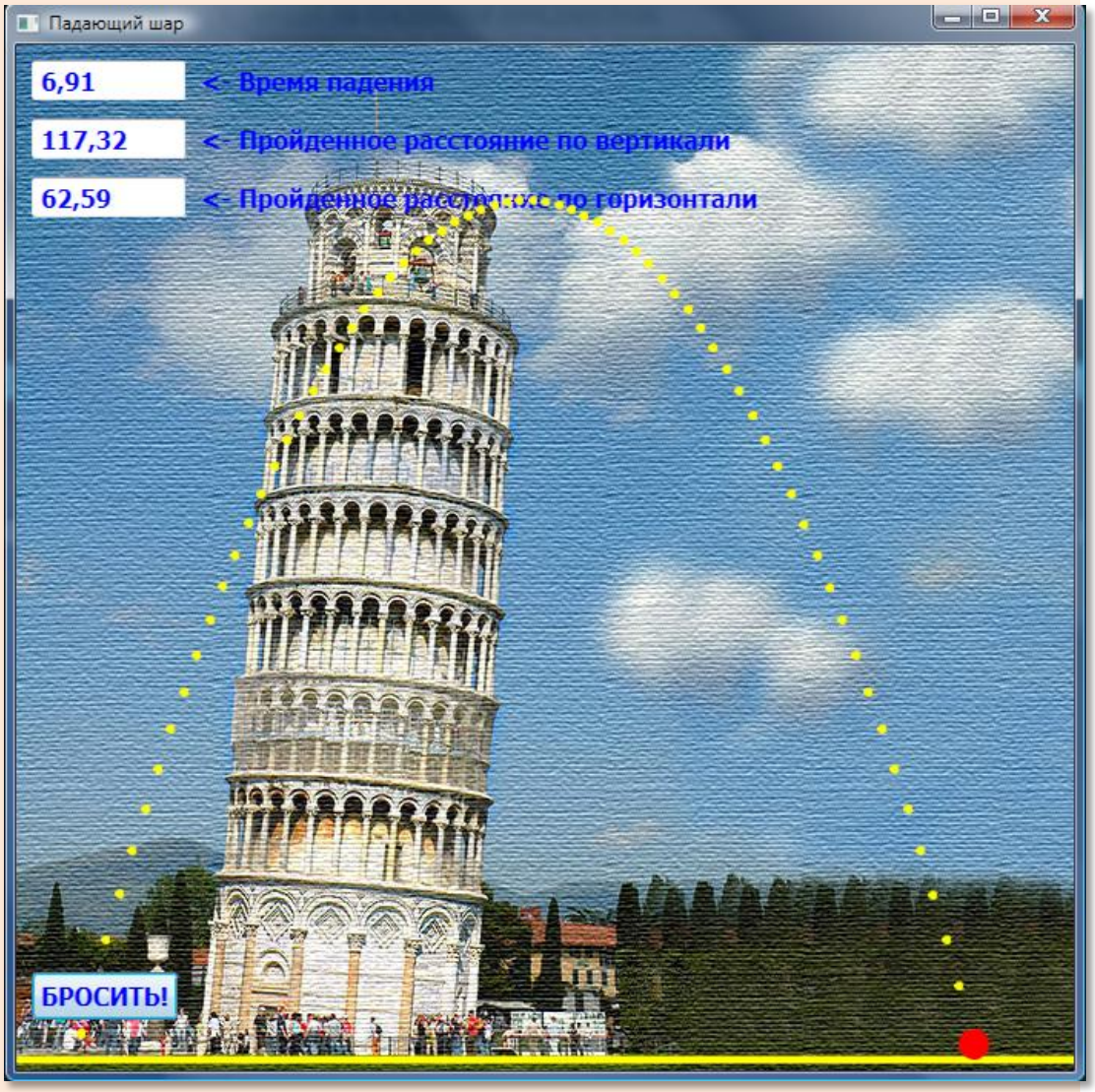


Рис. 27.7. Движение шара, брошенного с земли под углом к горизонту

Исходный код программы находится в папке **Fall3**.

3. Согласно легенде, однажды Ньютон лежал под деревом и о чём-то размышлял. Неожиданно на его голову упало яблоко, в результате чего он открыл закон всемирного тяготения. Определите, чему была равна скорость падения яблока, если оно висело в пяти метрах над землёй. Напишите программу, моделирующую падение яблока и других груш с дерева. Подумайте, какой бы закон открыл Ньютон, если бы на него упал арбуз!

4. Как вы помните, в кинокомедии *Бриллиантовая рука*, в той части, которая называется *Костяная нога*, Семён Семёныч Горбунков выпал из багажника автомобиля (Рис. 27.8). Всё бы ничего, но автомобиль был подвешен под вертолётом, который летел на высоте, будем считать, 75 метров. Определите, с какой скоростью Семён Семёныч врезался ногой в землю!

Рис. 27.8. Семён Семёныч вываливается из багажника

Урок 28. Тыблоки

Компьютерное тыблоко

Алексей Иванович Пантелеев в рассказе *Буква "ты"* учил одну маленькую девочку читать и писать. Иринушка, так звали девочку, быстро осваивала алфавит, пока они не дошли до последней буквы – *Я*, которую Иринушка упорно называла буквой *ТЫ*. Например, предложение *Якову дали яблоко* она читала так: *Тыкову дали тыблоко*. И так продолжалось до тех пор, пока Пантелеев не сказал Иринушке, что буква называется не *Я*, а *ТЫ*. Эта уловка помогла, и она стала читать все слова правильно, но мы поищем в русском языке именно *тыблоки*.

Мы возьмём за основу программу *Palindrome* и сохраним её в папке **Тыблоко** под тем же названием. Большинство переменных досталось нам по наследству, поэтому добавим только две новые *переменные*:

```
'ПРОГРАММА ДЛЯ ПОИСКА ТЫБЛОК

'variables
spisok[0]=""
nWords=0
txt=""
chr=""          ' буква
tybloko=""      ' слово-тыблоко
index=0
beg=0
len=0
```

Начало основной части программы также мало изменилось:

```
'========================================
'           ОСНОВНАЯ ПРОГРАММА
'========================================
TextWindow.ForegroundColor="Red"
TextWindow.WriteLine("ИЩЕМ ТЫБЛОКИ")
```

```
TextWindow.WriteLine("")
TextWindow.Show()

считать файл 1:
fileName= "OSh_frc2-4.txt"
readFile()
' считать файл 2:
'fileName= "OSh_frc5.txt"
'readFile()
' считать файл 3:
fileName= "OSh_frc6.txt"
'readFile()
' считать файл 4:
fileName= "OSh_frc7.txt"
'readFile()
```

Мы просто заполняем массив *spisok* словами нужной нам длины. Для примера загрузим только короткие слова.

```
TextWindow.WriteLine("")
TextWindow.ForegroundColor="Yellow"
' ищем слова, в которых есть буква Я:
for i=1 to nWords
  s= spisok[i]
  If Text.IsSubText(s, "Я")= "True" Then
    ' нашли  слово -->
    ' заменяем букву Я на ТЫ:
    tybloko=""
    len= Text.GetLength(s)
    For j= 1 to len
      ' очередная буква:
      chr= Text.GetSubText(s, j, 1)
      If (chr = "Я") Then
        tybloko= tybloko + "ТЫ"
      Else
        tybloko= tybloko + chr
      EndIf
    EndFor
    TextWindow.WriteLine(s + " - " + tybloko)
    File.AppendContents("tybloko.txt", s + " - " + tybloko)
  EndIf
EndFor

TextWindow.WriteLine("")
```

```
TextWindow.ForegroundColor="Red"
```

Затем в цикле *For* мы просматриваем все слова в списке. Если в них есть буква *Я* - а это очень просто узнать с помощью метода *IsSubText* класса *Text*, - то во вложенном цикле *For* мы находим каждую букву *Я* и подставляем вместо неё слово *ТЫ*. В изначально пустую строку *tybloko* мы последовательно добавляем буквы исходного слова, за исключением буквы *Я*, которую заменяет *ТЫ*.

Не забудьте скопировать в папку с программой все словари, иначе результат работы программы будет нулевой.

Исходное слово и слово-тыблоко выводим в *текстовое окно*, а также записываем в *файл* - для будущих чудачеств (Рис. 28.1).

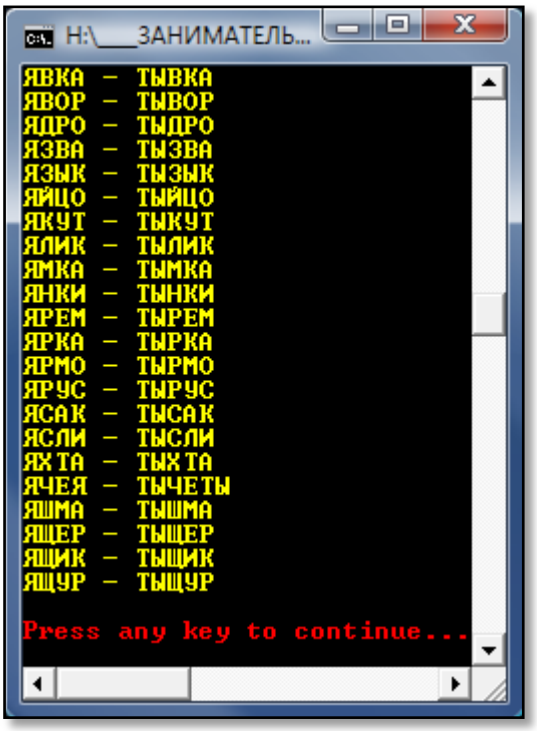


Рис. 28.1. Вот они, тыблочки наливные!

Иногда в тыблоки получаются обычными словами:

ЯК – ТЫК
ЯЛ – ТЫЛ
СОЯ – СОТЫ

БАНЯ – БАНТЫ
БУРЯ – БУРТЫ
СВАЯ – СВАТЫ
УНИЯ – УНИТЫ

Но попадаются и очень смешные тыблоки:

МЯЧ – МТЫЧ
ТУЯ – ТУТЫ
ЯЗЬ – ТЫЗЬ
БЯЗЬ – БТЫЗЬ
БЯКА – БТЫКА
ДОЯР – ДОТЫР
ДУЛЯ – ДУЛТЫ
ДЫНЯ – ДЫНТЫ
ЗАРЯ – ЗАРТЫ
ЗАЯЦ – ЗАТЫЦ
ЗМЕЯ – ЗМЕТЫ
ЗЮЗЯ – ЗЮЗТЫ
КЛЯП – КЛТЫП
КРЯЖ – КРТЫЖ
МАЯК – МАТЫК
ПЕНЯ – ПЕНТЫ
ПЛЯЖ – ПЛТЫЖ
ЯВОР – ТЫВОР
ЯДРО – ТЫДРО
ЯЗЫК – ТЫЗЫК
ЯЙЦО – ТЫЙЦО
ЯЛИК – ТЫЛИК
ЯНКИ – ТЫНКИ
ЯСЛИ – ТЫСЛИ
ЯХТА – ТЫХТА
ЯШМА – ТЫШМА
ЯЩЕР – ТЫЩЕР
ЯЩИК – ТЫЩИК

Исходный код программы находится в папке **Тыблоко**.

1. Можно заменять не только букву *Я* местоимением *ТЫ*, но и другие буквы какими-либо буквосочетаниями. Измените программу *Тыблоко* так, чтобы она искала такие слова.

2. Можно заменять не одну букву, а целое буквосочетание одной буквой или другим буквосочетанием.

3. Если взять не отдельные слова, а *текст*, в котором много слов с буквой *Я*, и заменить их на *тыблоки*, то получится очень смешной юмористический рассказ на *тыблочном языке* (Рис. 28.2). В данном случае следует отказаться от *текстового окна* и воспользоваться *графическим*, так как мы можем поместить на него элемент управления *многострочное текстовое поле*, в котором легко набирать текст (или просто копировать его из других источников). Если же к нему добавить ещё

одно *текстовое поле*, то в нём можно напечатать параллельный перевод текста на *тыблочный язык*.

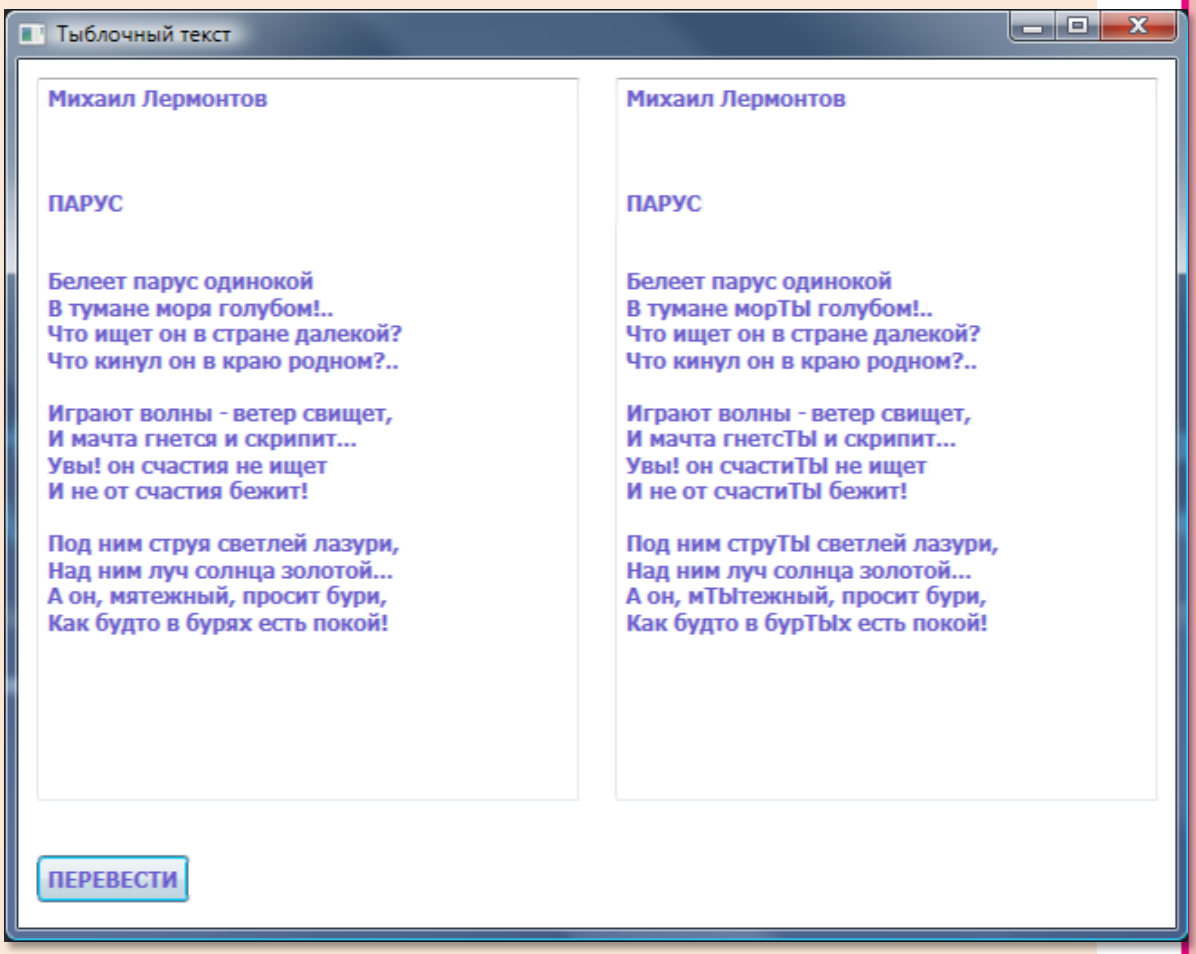


Рис. 28.2. Так может выглядеть интерфейс программы

Исходный код программы находится в папке **Тыблочный текст**.

РУССКИЙ ЯЗЫК

Урок 29. Занимательная логопедия

-Февочка, скажи «ыба»
- Селёдка!

Логопедический диалог из комедии
По семейным обстоятельствам

Некоторые люди неверно произносят отдельные буквы (точнее, звуки, обозначаемые этими буквами), а один *логопеф* не выговаривал даже половину букв русского алфавита, что ничуть не мешало ему учить других людей говорить правильно. Иногда это приводило к непониманию, например, названия улиц *Кировская* и *Киевская* в его исполнении звучали совершенно одинаково. В других случаях смысл его речей был понятен, но смешон. А «смешон» - это как раз то, что нам нужно!

Логопеф везде заменял букву **Д** буквой **Ф** (конечно, он произносил звуки, а не буквы, но у нас будут как раз буквы): вместо *девочка* он говорил *февочка,* вместо *будет – буфет*, ну и так *фалее*.

Если вы сделали (а сейчас мы это и проверим!) самостоятельное задание из урока *Тыблоки*, то переделать тыблочную программу в логопефическую вам будет проще простого!


```
'ПРОГРАММА ДЛЯ ПЕРЕВОДА ТЕКСТА НА
ЛОГОПЕФИЧЕСКИЙ ЯЗЫК

'variables
chr=""          'буква
logopef=""      'логопефическое слово
string=""       'строка с текстом
len=0           'длина текста
```

Понятное дело: без *графического окна* нам не обойтись, поэтому сразу же настроим его под Свои нужды:

```
GraphicsWindow.Title="Логопефический текст"
GraphicsWindow.Width= 800
GraphicsWindow.Height= 320
```

```
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
```

Добавляем к окну - по вкусу и нашим возросшим потребностям - *элементы управления* – два многострочных *текстовых поля* и одну *кнопку*:

```
txtSource=Controls.AddMultiLineTextBox(10, 10)
Controls.SetSize(txtSource,GraphicsWindow.Width/2-
20,GraphicsWindow.Height-80)
txtLogopef=Controls.AddMultiLineTextBox(GraphicsWindow.Width/2
+10, 10)
Controls.SetSize(txtLogopef,GraphicsWindow.Width/2-
20,GraphicsWindow.Height-80)

btnTLogopef=
Controls.AddButton("ПЕРЕВЕСТИ",10,GraphicsWindow.Height-40)
Controls.ButtonClicked= OnClick
```

В *левое текстовое поле* нужно скопировать текст (или набрать его вручную - при надлежащем трудолюбии и упорстве), затем нажать кнопку *ПЕРЕВЕСТИ*, которая выполнит процедуру *OnClick* и напечатает перевод сообщения в *правом текстовом поле*.

```
Sub OnClick
  string= Controls.GetTextBoxText(txtSource)
  len= Text.GetLength(string)

  ' заменяем букву Д буквой Ф:
  logopef=""
  For i= 1 to len
    ' очередная буква:
    chr= Text.GetSubText(string, i, 1)
    c = Text.ConvertToUpperCase(chr)
    If (c = "Д") Then
      logopef= logopef + "Ф"
    Else
      logopef= logopef + chr
    EndIf
  EndFor
```

```
  Controls.SetTextBoxText(txtLogopef, logopef)
EndSub
```

Важно: буква *Д* в тексте может быть и строчной, и ПРОПИСНОЙ. Для нас это одна и та же буква, а для компьютера – разные, поэтому нужно либо поверять обе буквы

```
If (chr = "Д" Or chr = "д") Then  …,
```

либо переводить очередной символ в верхний регистр, тогда все буквы будут ПРОПИСНЫМИ, что облегчит проверку. Однако, если вы хотите сохранить в тексте все буквы в их исходном состоянии, то позаботьтесь об их неприкосновенности.

Наш переводчик готов к работе. Для пущего смеху лучше подобрать текст, в котором много букв *Д*. Например, в Интернете можно найти «однобуквицу» про *деда Данилыча* (автор *OJIesYA_EviL_MonKey*), которая вполне годится для наших экспериментов (Рис. 29.1).

Тексты, все слова которых начинаются на одну и ту же букву, называются *тавтограммами*. Самая известная тавтограмма: ***Ч**етыре чёрненьких чумазеньких чертёнка чертили чёрными чернилами чертёж - чрезвычайно чисто.*

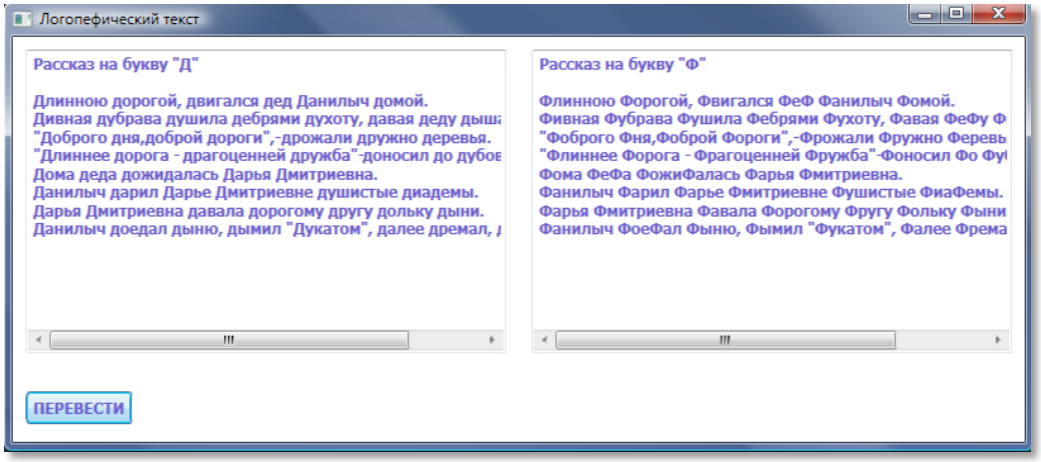


Рис. 29.1. Фолгая форога в фюнах!

Исходный код программы находится в папке **Логопеф**.

1. Подберите сами тексты с буквой *Д* и переведите их с помощью нашего *Логопефа*.

2. Попробуйте сами придумать подобный текст – с учётом того, что после перевода он должен стать очень смешным.

3. Добавьте к окну программы ещё одну кнопку – *СТЕРЕТЬ*, чтобы программой можно было бы пользоваться неоднократно.

4. Как вы помните, киношный логопеф не выговаривал букву *Р*, поэтому её тоже можно заменить буквой *Г* или *Й*. Или вообще выбросить из текста.

5. В некоторых русских говорах вместо буквы *Ц* произносят *Ч*: *цапля – чапля* (откуда и возникла русская фамилия Чаплин, которая не имеет к Чарли Чаплину никакого отношения, поскольку это наш Цаплин).

6. Писатель-юморист Семён Альтов знаменит не только своими смешными рассказами, но и особенностями их чтения: он глухие звуки произносит как звонкие, что очень смешно. Например, *канарейка* у него слала *ганарейгой*, *парикмахер – баригмакер*. Заставьте программу читать по-альтовски.

7. Как известно, москвичи (и мы вслед за ними) акают, то есть все безударные *О* произносят как *А*, то есть Москва и Россия по-московски - *МАсква* и *РАссия*. На слух мы уже привыкли к этому коверканью русской речи, но вот в напечатанном тексте это выглядит – ну, сами узнаете как, когда напишете программу для перевода русского текста на московский язык.

8. Помните песенку зайца из мультфильма *Ну, погоди*: «А ну-ка, давай-ка, плясать выходи!»? Вы легко можете ко всем (или только к некоторым) словам любого текста добавить частицу ка. Получится «заячий» текст.

9. Можно перевести текст и на «волчий» язык, если после каждого слова добавлять ну: *Ну, Заяц, ну, погоди, ну!*

Урок 30. Занимательная транслитерация

Мы уже встречались с транслитерацией на уроке *Признаки делимости*, где **транслитерация**, то есть запись русских слов латинскими буквами, была предложена как способ именования переменных программы. Давайте напишем приложение, которое будет автоматически переводить русские слова на «латынь».

Наш новый проект очень похож на программу *Тыблоко*, только теперь нам придётся заменять не только букву *Я*, но и все остальные. Как обычно, загружаем проект-шаблон *Тыблоко* и сохраняем его в папке **Транслитерация** под новым названием.

Дальше - всё просто. Заменяем одну *переменную* и – главное! - формируем *таблицу* для замены всех русских букв одной или несколькими латинскими:

```
' ПРОГРАММА ДЛЯ ТРАНСЛИТЕРАЦИИ РУССКИХ СЛОВ

'variables
spisok[0]=""
nWords=0
txt=""
chr=""
string=""       ' закодированное слово
index=0
beg=0
len=0

translit["А"] = "A"
translit["Б"] = "B"
translit["В"] = "V"
translit["Г"] = "G"
translit["Д"] = "D"
translit["Е"] = "E"
translit["Ё"] = "YO"
translit["Ж"] = "ZH"
translit["З"] = "Z"
translit["И"] = "I"
translit["Й"] = "Y"
translit["К"] = "K"
translit["Л"] = "L"
```

```
translit["М"] = "M"
translit["Н"] = "N"
translit["О"] = "O"
translit["П"] = "P"
translit["Р"] = "R"
translit["С"] = "S"
translit["Т"] = "T"
translit["У"] = "U"
translit["Ф"] = "F"
translit["Х"] = "KH"
translit["Ц"] = "TS"
translit["Ч"] = "CH"
translit["Ш"] = "SH"
translit["Щ"] = "SHCH"
translit["Ъ"] = "'"
translit["Ы"] = "Y"
translit["Ь"] = "J"
translit["Э"] = "E"
translit["Ю"] = "YU"
translit["Я"] = "YA"
```

А дальше – ещё проще:

```
...
TextWindow.WriteLine("")
TextWindow.ForegroundColor="Yellow"
'просматриваем все слова в списке:
for i=1 to nWords
  s= spisok[i]
  string=""
  len= Text.GetLength(s)
  For j= 1 to len
    'очередная буква:
    chr= Text.GetSubText(s, j, 1)
    string=string + translit[chr]
  EndFor
  TextWindow.WriteLine(s + " - " + string)
  File.AppendContents("translit.txt", s + " - " + string)
EndFor
```

Последовательно выбираем слова из массива *spisok* и каждую букву очередного слова заменяем «транслитерационными» символами. Перекодированное слово выводим в *текстовое окно* и в *файл* (Рис. 30.1).

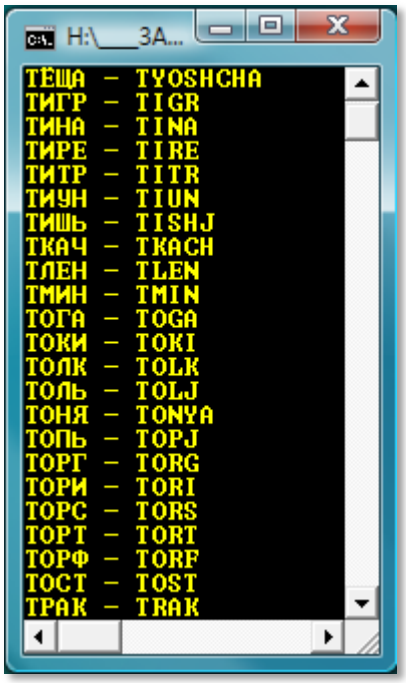


Рис. 30.1. Русско-латинский словарь

Если внимательно посмотреть на массив *translit*, то можно заметить, что некоторые буквы однозначно заменяются *транс-символами*. Например, *А, Б, В*. Другие, например, буква *Ё* требует *два* символа – *Y* и *O*, которые уже обозначают другие буквы – *Й* и *О*. Таким образом, обратный перевод (декодирование) нельзя провести однозначно.

Слово *ЙОГУРТ* при транслитерации превратится в *YOGURT*, а при переводе в обратную сторону мы получим два слова: правильное – *ЙОГУРТ* и неправильное – *ЁГУРТ*. В простых случаях достаточно просмотреть словарь и убедиться, что в нём имеется только первый вариант, который и является верным. Конечно, такая однозначность не всегда достижима.

Исходный код программы находится в папке **Транслитерация**.

Переделайте программу *Транслитерация* под *графическое окно* - так, как это сделано в программе *Занимательная логопедия*, и займитесь переводами известных произведений на «транслитерационный» язык.

Урок 31. Занимательная латиница

Moya tvoya ne ponimaet!

Из разговора в чате

Занимаясь транслитерацией, вы, конечно, не могли не обратить внимания на то, что некоторые кириллические буквы совпадают по написанию с латинскими, хотя и не обязательно обозначают один и тот же звук.

Предположим, что у нас имеются только латинские буквы. Сколько русских слов мы сможем напечатать?

Латиницей называют буквы латинского алфавита и слова, записанные такими буквами. *Кириллицей* – слова, записанные русскими буквами.

Для определённости будем считать, что только русские буквы **АВСЕНКМОРТХ** имеют аналоги в латинице.

За основу нового проекта возьмём программу *Тыблоки*, в которую добавим константу *latin*:

```
'ПРОГРАММА ДЛЯ ПОИСКА ЛАТИНСКО-РУССКИХ СЛОВ

'const
latin="АВСЕНКМОРТХ"
```

В этой строке находятся те русские буквы, которые мы решили считать «латинскими».

Часть новой программы, которая совпадает с исходной, мы пропускаем и сразу переходим к поиску нужных нам слов:

```
'просматриваем все слова в списке:
for i=1 to nWords
  s= spisok[i]
  len= Text.GetLength(s)
  'ищем слова, записанные латиницей:
  For j= 1 to len
```

```
    'очередная буква:
    chr= Text.GetSubText(s, j, 1)
    If Text.IsSubText(latin, chr) = "False" Then
      Goto next
    EndIf
  EndFor
  TextWindow.WriteLine(s)
  File.AppendContents("latin.txt", s)
next:
EndFor
```

Нам достаточно проверить, входит ли каждая буква очередного слова в строку *latin*. Если хотя бы одна буква «чисто русская», мы такое слово пропускаем и переходим к следующему. В противном случае выводим найденное слово в *текстовое окно* и записываем в *файл*.

Все проверки мы проводим в цикле *For* с помощью метода *IsSubText*:

```
Text.IsSubText(latin, chr) = "False"
```

Вот словарик четырёхбуквенных слов (Рис. 31.1).

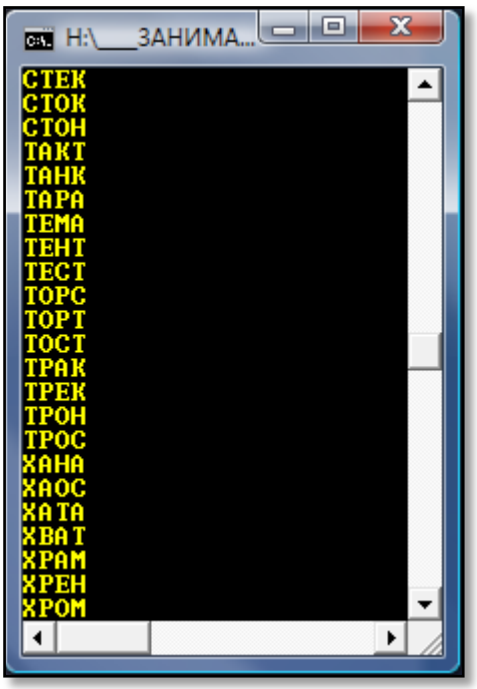


Рис. 31.1. Латинско-русский словарик составлен!

Если вы запасётесь терпением и словарями, то найдёте, что самые длинные русские слова, записанные латиницей, - это ВЕРХОВЕНСТВО, КОММЕРСАНТКА и ОРКЕСТРАНТКА.

Исходный код программы находится в папке **Латиница**.

Злые шутки с идентификаторами

Так как *СБ* позволяет записывать идентификаторы и русскими буквами, то можно легко разыграть начинающих программистов. Составьте идентификатор так, чтобы он мог быть записан русскими буквами, но при этом выглядел бы так, как будто записан латинскими. Например, *РАСА, СЕРА, АРХАР, СЕРСО*. При попытке воспользоваться этими идентификаторами будет возникать ошибка, поскольку все программисты уверены, что идентификаторы составлены из латинских букв.

Супернаборщик

Наверное, всем известна школьная игра *Наборщик*. В ней требуется из букв какого-либо одного заданного *длинного* слова составлять другие, более *короткие* слова, с условием, что они будут состоять только из тех букв, которые имеются в заданном слове. При этом, если в длинном слове, например, две буквы *А*, то и в каждом составленном слове их должно быть *не больше* двух. Побеждает тот игрок, который составит более длинный список слов. Очень хорошая игра, в которую интересно играть, особенно на уроках математики.

Мы сделаем себе небольшое послабление в правилах игры – не будем требовать, чтобы число одинаковых букв в искомых словах не превышало числа этих букв в заданном слове. Иначе говоря, любую букву этого слова можно использовать сколько угодно раз. Все остальные правила мы трогать не будем.

Проект мы назовем, конечно, **Супернаборщик**:

```
'СУПЕРНАБОРЩИК

'variables
…
string=""       'длинное слово
```

Нам потребуется всего одна новая «строковая» переменная *string*, в которой мы и будем хранить заданное длинное слово.

Как и во всех других словесных программах, сначала нужно загрузить словари. Опыт подсказывает нам, что на этот процесс потребуется время, и, чтобы игроки не отчаялись ждать, развлечем их сообщением:

```
'========================================
'            ОСНОВНАЯ ПРОГРАММА
'========================================
TextWindow.WriteLine("Подождите, я загружаю словари!")
TextWindow.Show()


…
fileName= "OSh_frc7.txt"
readFile()

TextWindow.Clear()
```

Затем приглашаем их задать слово, из букв которого мы будем составлять более короткие слова:

```
start:
TextWindow.ForegroundColor="Red"
TextWindow.BackgroundColor="Yellow"
TextWindow.WriteLine("Введите исходное слово и нажмите клавишу
ВВОД")
TextWindow.BackgroundColor="Black"
string= Text.ConvertToUpperCase(TextWindow.Read())
```

Заданное слово сразу же переводим в *верхний* регистр, потому что все слова в наших словарях записаны БОЛЬШИМИ буквами, и сохраним в переменной *string*.

Теперь нам осталось выбрать из списка такие слова, которые целиком состоят из букв длинного слова.

```
TextWindow.ForegroundColor="Yellow"
TextWindow.WriteLine("")
TextWindow.WriteLine("")
'просматриваем все слова в списке:
for i=1 to nWords
  s= spisok[i]
  len= Text.GetLength(s)
  'ищем слова, состоящие из букв заданного слова:
  For j= 1 to len
    'очередная буква:
    chr= Text.GetSubText(s, j, 1)
    If Text.IsSubText(string, chr) = "False" Then
      Goto next
    EndIf
  EndFor
  TextWindow.WriteLine(s)
next:
EndFor
```

Как видите, эти действия практически ничем не отличаются от тех, что мы рассмотрели в проекте *Латиница*, но теперь мы можем загадывать любые слова сколько угодно раз – пока не наиграемся!

```
TextWindow.WriteLine("")
TextWindow.BackgroundColor="Black"
TextWindow.ForegroundColor="Red"
Goto start
```

Если не учитывать времени на загрузку словаря, то сам поиск слов происходит очень быстро, и в текстовом окне мы тут же получаем все подсказки (Рис. 31.2). С таким помощником вы легко станете чемпионом школы по игре в *Супернаборщика*!

1. Если вы планируете использовать эту программу «для дома, для семьи», то позаботьтесь о том, чтобы придать ей более привлекательный вид, заменив текстовый интерфейс графиче-

ским.

2. Напишите программу для игры в классического *Наборщика*! За основу вы можете взять нашу программу, но процедуру поиска слов придётся изменить, потому что буквы заданного слова необходимо считать.

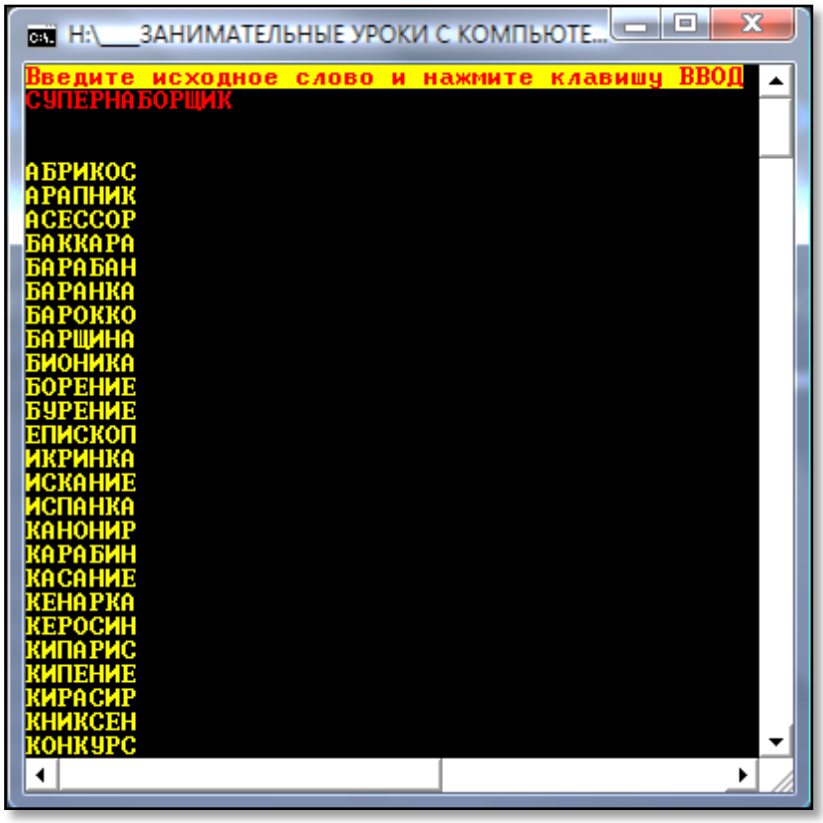


Рис. 31.2. А вы нашли бы столько слов?

Исходный код программы находится в папке **Суперна-борщик**.

Урок 32. Занимательная криптография

- Бамбарбия! Киргуду!

Загадочные слова из комедии
Кавказская пленница

Криптография в переводе с греческого означает *тайнопись*. Она возникла одновременно с письменностью - для того, чтобы утаить содержание письма или другого документа. Действительно, обычный текст может прочитать любой грамотный человек, а ведь иногда просто необходимо скрыть важную информацию от посторонних лиц.

В повести Фриды Абрамовны Вигдоровой *Дорога в жизнь*, в 21-ой главе *У вас ничего не выйдет* рассказывается о криптографической проделке воспитанника детского дома Андрея Репина, который вёл в тетради протокол собрания. И вот что у него получилось:

> *Богдащоричи: беврый одвят тефувид бо гдочорой, рдовой бо трову, дведий бо чегдщике...*

Он, конечно, хотел поразить воспитателей своим шифрованным письмом, но один из них, Алексей Саввич, быстро смекнул:

> *– Да, тут есть логика. Постойте... Беврый одвят... беврый одвят... да это же первый отряд! Так... тефувид – дежурит. Понимаете, он оставил гласные, а остальной алфавит разделил пополам и поменял согласные местами: вместо п – б, вместо в – р и наоборот...*

Андрей Репин был посрамлён, но не угомонился, о чём мы поговорим дальше, а пока давайте-ка поразмыслим над его *тайнописью*.

Этот способ шифрования сообщений был известен еще в Древней Руси и называется *простой литореей* (не путайте с лотереей!). В то время букв было больше, но и наши 33 буквы вполне годятся

для шифрования. Правда, букву *Ё* придётся исключить из списка, чтобы осталось чётное число букв.

Букве *Ё* вообще не очень-то везёт в русском языке. Её уже не раз пытались исключить из алфавита, но, к счастью, пока безуспешно. А в 2005 году в городе Ульяновске был открыт памятник этой славной букве (Рис. 32.1).

Рис. 32.1. Памятник букве *Ё*

Буква *Ё* появилась в русском алфавите в 1797 году, так что возраст у неё почтенный.

Простая литорея называется также *тарабарской грамотой*. А название литорея произошло от слова *литера – буква*.

Теперь расположим буквы в две строки, по 16 букв в каждой так, чтобы они образовали вертикальные пары:

А Б В Г Д Е Ж З И Й К Л М Н О П

Р С Т У Ф Х Ц Ч Ш Щ Ъ Ы Ь Э Ю Я

Чтобы закодировать слово, достаточно заменить в нём каждую букву парной в этом шифре. Например, слово *КРИПТОГРАФИЯ* превратится в *ЪАШЯВЮУАРДШП*. Пожалуй, не сразу и догадаешься, что за слово написано, хотя, как вы видите способ шифрования очень простой. Вот только вручную заменять одни буквы другими довольно утомительно, да и ошибиться можно не раз. Поэтому давайте напишем программу, которая в два счёта зашифрует любой текст.

Загрузите в *СБ* проект *Логопеф* и сохраните его в папке **Литорея**. Начинаем священнодействовать, то есть кодировать шифр!

```
'ПРОСТАЯ ЛИТОРЕЯ

'const
lit[1]=" АБВГДЕЖЗИЙКЛМНОП"
lit[2]=" РСТУФХЦЧШЩЪЫЬЭЮЯ"

'variables
chr=""          'буква
pos=0           'позиция буквы в массиве lit
litorea=""      'литорейный текст
string=""       'строка с текстом
len=0           'длина текста

GraphicsWindow.Title=" Простая литорея"
GraphicsWindow.Width= 800
GraphicsWindow.Height= 320
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) / 2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height) / 2
```

Для работы с программой нам понадобятся две *кнопки* и два *многострочных текстовых поля*:

```
'Добавляем элементы управления:
txtSource=Controls.AddMultiLineTextBox(10, 10)
Controls.SetSize(txtSource,GraphicsWindow.Width/2-
20,GraphicsWindow.Height-80)
txtLitorea=Controls.AddMultiLineTextBox(GraphicsWindow.Width/2
+10, 10)
```

```
Controls.SetSize(txtLitorea,GraphicsWindow.Width/2-
20,GraphicsWindow.Height-80)

btnLitorea=
Controls.AddButton("ПЕРЕВЕСТИ",10,GraphicsWindow.Height-40)
Controls.ButtonClicked= OnClick
btnClear=
Controls.AddButton("СТЕРЕТЬ",100,GraphicsWindow.Height-40)
```

Назначение каждого из них понятно и без пояснений.

Нажатие кнопок мы обрабатываем в процедуре *OnClick*. Так как на форме мы разместили *две* кнопки, то сначала нужно определить, какая из них была нажата.

Если это кнопка *СТЕРЕТЬ*, то мы очищаем оба *текстовых поля*. Для этого достаточно записать в них пустую строку.

Если же нажата кнопка *ПЕРЕВЕСТИ*, то мы считываем в переменную *string* весь текст из ЭУ *txtSource*, а затем последовательно просматриваем всю строку в цикле *For*.

С помощью метода *GetIndexOf* мы легко узнаем, имеется ли такой символ в массиве *lit*, а если имеется, то в какой из строк – в верхней или в нижней. В зависимости от результата проверки заменяем букву её парой. Букву *Ё* и все остальные символы (например, пробелы, латинские буквы, цифры и знаки препинания) просто копируем в новую строку *litorea*.

Закончив цикл, мы печатаем эту строку в правом *текстовом поле*:

```
Sub OnClick
  btn=Controls.LastClickedButton
  If (btn= btnClear) then
    Controls.SetTextBoxText(txtSource, "")
    Controls.SetTextBoxText(txtLitorea, "")
    Goto exit
  EndIf

  string= Controls.GetTextBoxText(txtSource)
```

```
  len= Text.GetLength(string)
  'кодируем - заменяем буквы парными:
  litorea=""
  For i= 1 to len
    'очередная буква:
    chr= Text.GetSubText(string, i, 1)
    c = Text.ConvertToUpperCase(chr)
    pos= Text.GetIndexOf(lit[1], c)
    If (pos > 0) Then 'верхняя буква
      'заменяем нижней:
      litorea= litorea + Text.GetSubText(lit[2], pos,1)
    Else
      pos= Text.GetIndexOf(lit[2], c)
      If (pos > 0) Then 'нижняя буква
        'заменяем верхней:
        litorea= litorea + Text.GetSubText(lit[1], pos,1)
      else 'буква Ё или другой символ:
        litorea= litorea + chr
      EndIf
    EndIf
  EndFor
  Controls.SetTextBoxText(txtLitorea, litorea)
  exit:
EndSub
```

Перевод проходит «без шума и пыли», но все буквы становятся ПРОПИСНЫМИ (Рис. 32.2).

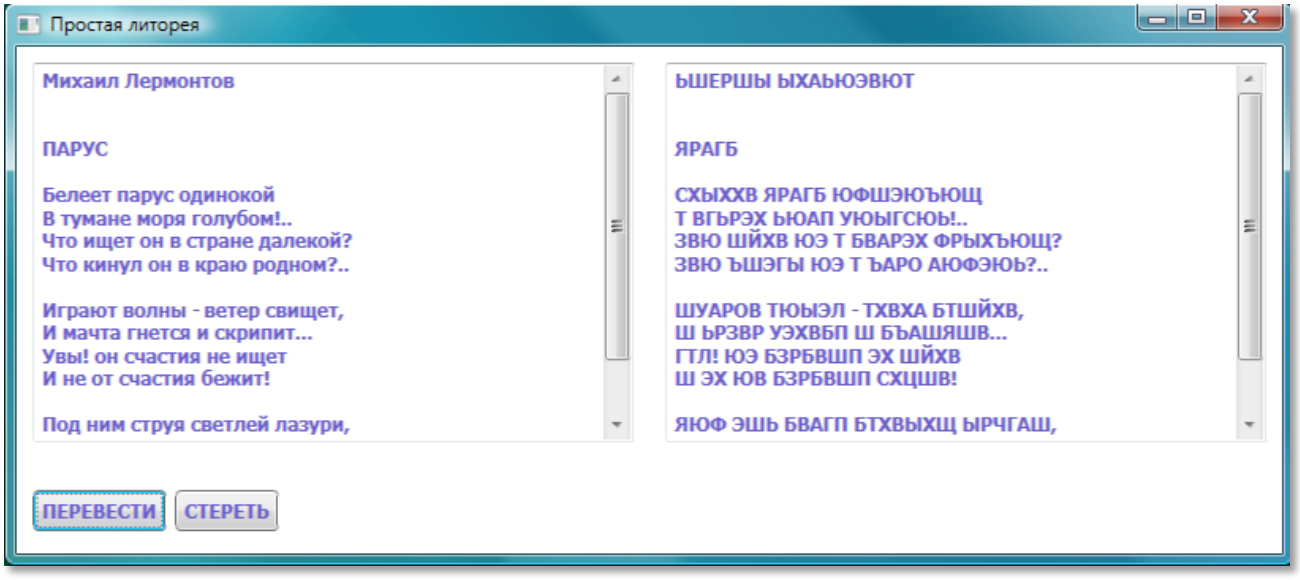


Рис. 32.2. Толковый перевод!

От этой порчи текста легко избавиться, если в массиве *lit* завести проверочные строки и для маленьких букв.

Декодирование текста легко провести в той же программе. Достаточно текст из правого поля перенести в левое и нажать кнопку *ПЕРЕВЕСТИ* (Рис. 32.3).

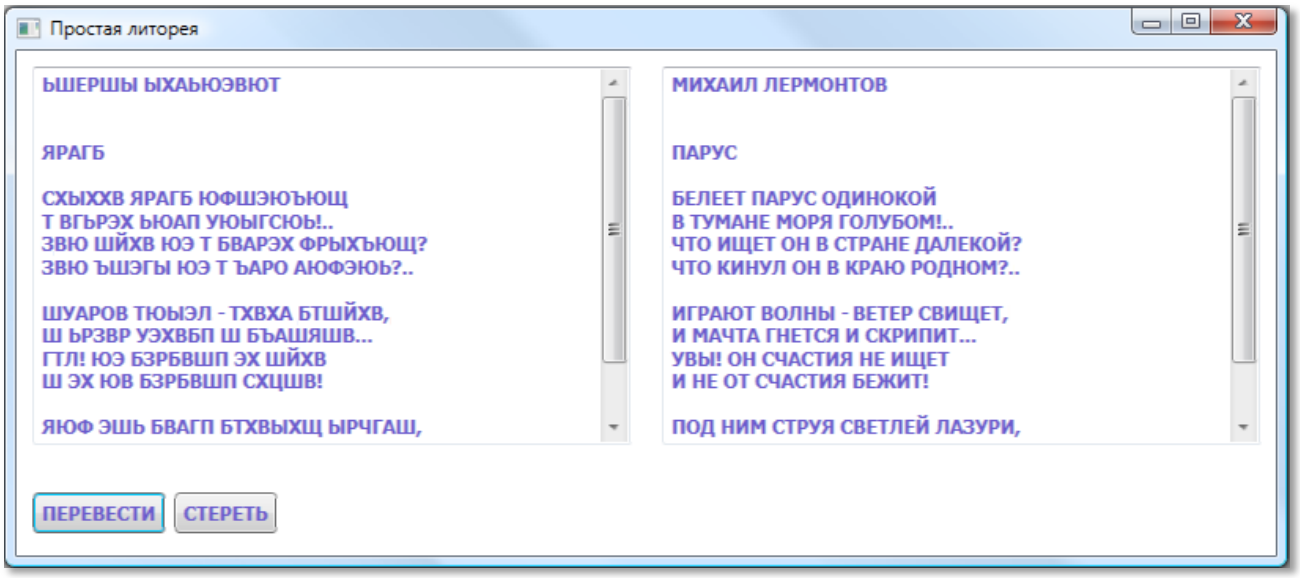


Рис. 32.3. Декодирование прошло успешно!

Исходный код программы находится в папке **Литорея**.

Литорея мудрая

В знаменитой повести А. Рыбакова *Кортик* мы также встречаемся с зашифрованной надписью, которая украшала ножны и сам кортик.

Вот что поведала нам Глава 53 *Ножны*:

> *Мальчики забежали за церковный придел, и Коровин вытащил из кармана ножны.*
> *Миша нетерпеливо выхватил их у него, повертел в руках, затем осторожно снял ободок и вывернул шарик.*

Ножны развернулись веером. Мальчики уставились на них, потом удивленно переглянулись…
На внутренней стороне ножен столбиками были нанесены знаки: точки, чёрточки, кружки. Точно так же, как и на пластинке кортика.

За разгадкой тайны кортика друзья обратились (Глава 56 *Литорея*) к директору школы Алексею Ивановичу, который и объяснил им, что надпись на кортике и ножнах представляет собой *литорею мудрую*, и что надписи на них можно прочитать, только соединив их вместе.

Так что же такое – литорея мудрая? – Давайте разбираться!

В Древней Руси 30 букв алфавита делили на три равные части, по 10 букв в каждой. В пределах первого десятка буквы последовательно обозначали **точками** (Рис. 32.4).

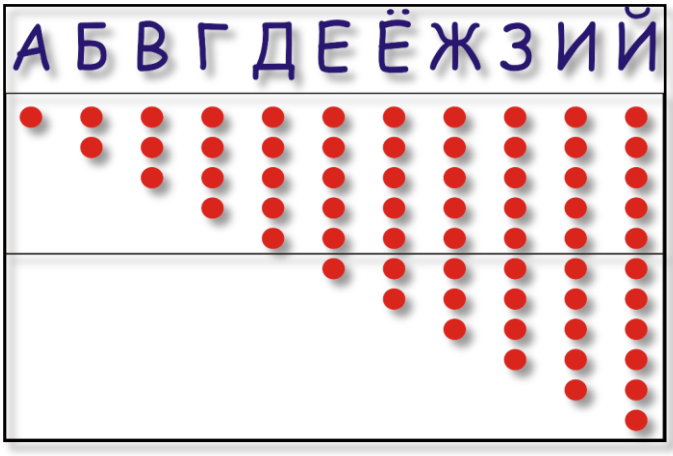


Рис. 32.4. Буквы-точки

Второго – **чёрточками** (Рис. 32.5), а третьего – **кружками** или крестиками (Рис. 32.6).

Как видите, знаки для каждой буквы записывались *столбиком*. А горизонтальная черта делила столбики пополам. Зная обозначение каждой буквы, мы можем составить любой текст, например, *Литорея мудрая* (Рис. 32.7).

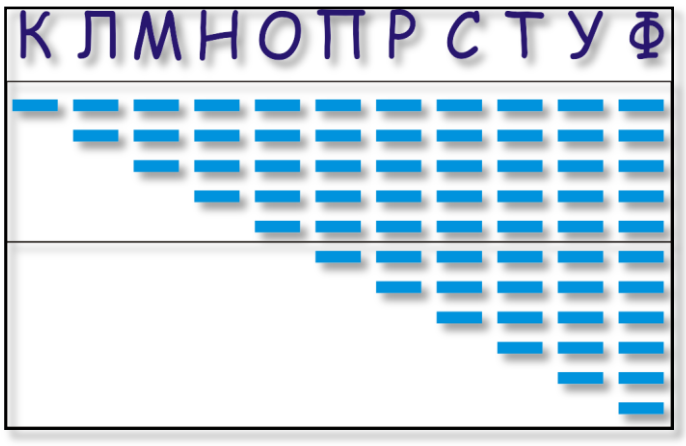


Рис. 32.5. Буквы-чёрточки

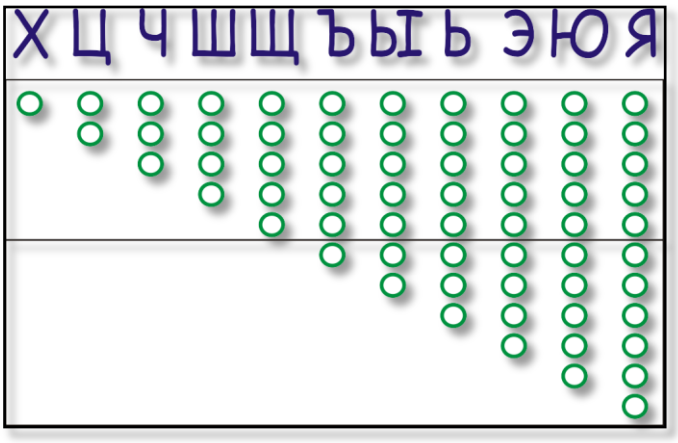


Рис. 32.6. Буквы-кружочки

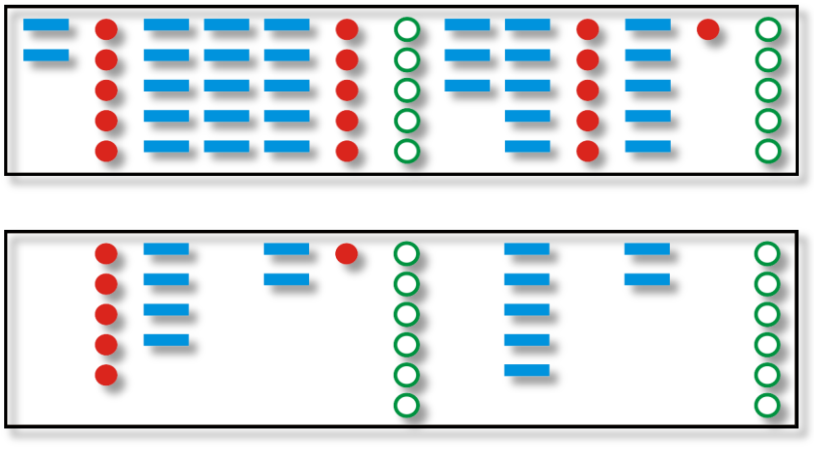


Рис. 32.7. Надпись литореей мудрой

Теперь закроем нижнюю часть шифровки и попробуем восстановить текст: **Л** Д-Й О-Ф О-Ф О-Ф Д-Й Щ-Я **М** О-Ф Д-Й О-Ф **А** Щ-Я. Удалось точно определить три буквы (они выделены **жирным**

шрифтом), для остальных мы нашли только диапазон возможных значений. Да, задачка оказалась совсем непростой!

Поскольку нам нужны не 30, а все 33 буквы современного русского алфавита, то у нас в каждой группе будет не по десять, а по одиннадцать букв. По этой причине максимальная высота столбиков с символами равна 11. Число нечётное, поэтому пришлось провести горизонтальную линию так, чтобы в верхней части было до пяти символов, а в нижней – все остальные.

Осталось разрезать полоску бумаги по горизонтальной линии на две части. Теперь, чтобы прочитать текст, нужно соединить обе части бумажки по линии разреза. Точно так же было зашифровано сообщение на кортике и ножнах. Если вы читали повесть А. Рыбакова, то помните, что на них было написано:

> *Сим гадом завести часы понеже проследует стрелка полудень башне самой повёрнутой быть.*

Несмотря на название, шифр не очень «мудрый»: некоторые буквы, на обозначение которых пошло от одного до четырех символов, остались незакодированными, что помогает расшифровать запись. Насколько помогает – определите сами.

Герои повести справились и с этой загадкой, а нам пора вернуться к тарабарской грамоте.

Тарабарская грамота

Как мы знаем, простая литорея называется также тарабарской грамотой, но есть и другие способы превратить русские тексты в тарабарщину. Этим достойным делом мы сейчас и займёмся.

Запишите исходный код программы *Литорея* в папку **Тарабарская грамота**. Давайте сразу же посмотрим, как работает программа (Рис. 32.8).

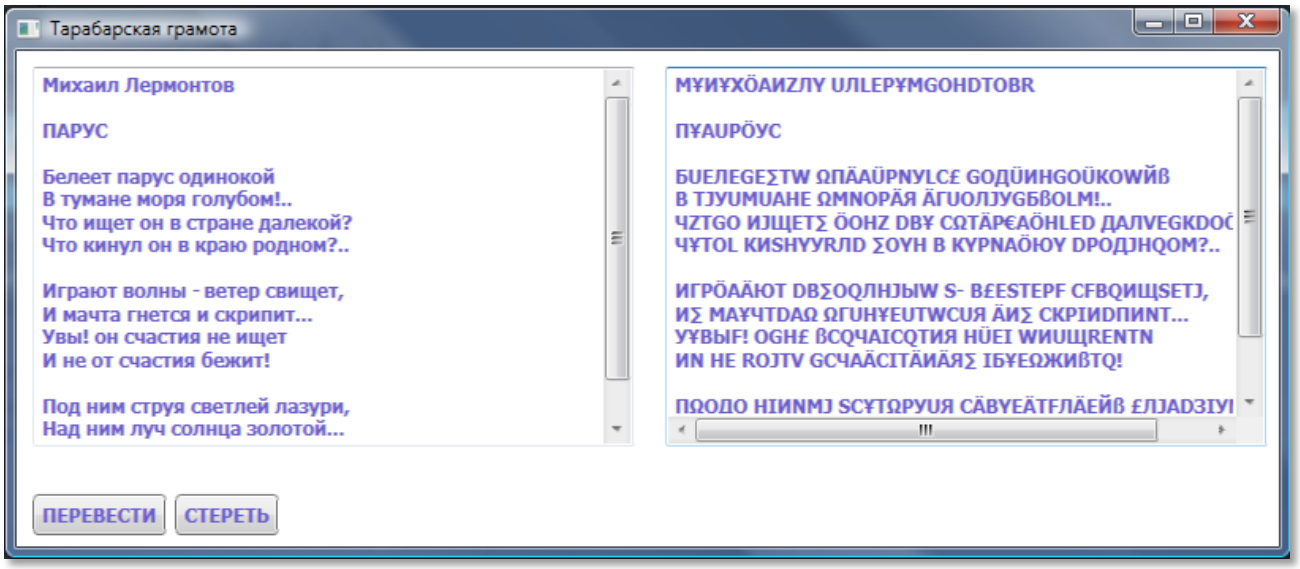


Рис. 32.8. И вправду полная тарабарщина!

Этот вид тайнописи наряду с обеими литореями применяли наши далекие предки, чтобы скрыть свои мысли и чаяния. На первый взгляд, текст справа написан не на русском языке, а на каком-то иноземном. Однако ни одна русская буква при этом гуманном эксперименте не пострадала! Их ровно столько же справа, сколько и слева, и трудно не заметить, что строчки в правом текстовом поле стали *длиннее*. Вот в этом-то весь секрет: нужно по вкусу добавить к русским буквам «тарабарские». Проще всего воспользоваться «европейскими» буквами и знаками, которых нет в русском языке.

Ну, вот, с шифром разобрались, а теперь за дело!

```
'ПРОГРАММА ТАРАБАРСКАЯ ГРАМОТА

'const
rus=" АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ"
bukvy="QWRZUIÜSDFGJLÖÄßYVN€£¥Ω∑"

'variables
chr=""          'буква
tarabar=""      'тарабарский текст
string=""       'строка с текстом
len=0           'длина текста

GraphicsWindow.Title=" Тарабарская грамота"
```

Нам потребуются также русские буквы и знак пробела. Зачем? – Мы будем вставлять инородные буквы только после русских, а для этого нам нужно находить среди всех символов текста, в котором могут быть знаки препинания, цифры и служебные символы, русские буквы, Пробел тоже пригодится, потому что он стоит перед первой буквой слова, иначе все слова будут начинаться с правильной буквы. А вот перед начальной буквой строки поставить другую букву сложнее. В этом случае стоит весь текст записать одной строкой.

Следующий фрагмент кода программы *Литорея* остался без изменений, но при желании вы можете заменить название правого поля более походящим *txtTarabar*.

В процедуре, обрабатывающей нажатие кнопок, мы перепишем только те строки, которые занимаются зашифровкой текста:

```
Sub OnClick
. . .
  'кодируем -  добавляем тарабарские буквы:
  tarabar=""
  For i= 1 to len
    'очередной символ текста:
    chr= Text.GetSubText(string, i, 1)
    c = Text.ConvertToUpperCase(chr)
    tarabar= tarabar + c
    'если это буква или пробел -->
    If (Text.IsSubText(rus, c) = "True") Then
      n= Math.GetRandomNumber(100)
      'если случайное число больше заданного,
      'то добавляем к тексту тарабарскую букву:
      If (n> 33) Then
        n=  Math.GetRandomNumber(Text.GetLength(bukvy))
        chr= Text.GetSubText(bukvy, n, 1)
        tarabar= tarabar + chr
      EndIf
    EndIf
  EndFor
  Controls.SetTextBoxText(txtTarabar, tarabar)
  exit:
EndSub
```

Если вы хотите, чтобы тарабарские буквы добавлялись чаще, то уменьшите число в выражении *n > 33*, а если реже – увеличьте его.

Исходный код программы находится в папке **Тарабарская грамота**.

1. А что же всё-таки написал в протоколе Андрей Репин? - Вы будете крепко озадачены, если раскодируете протокол в нашей программе. Вместо одной тарабарской грамоты вы получите другую:

СЮУФРЙЮАШЗШ: СХТАЛЩ ЮФТПВ ВХДГТШФ СЮ УФЮЗЮАЮЩ, АФЮТЮЩ СЮ ВАЮТГ, ФТХФШЩ СЮ ЗХУФЙШЪХ

Как же так: мы выяснили, что Андрей Репин для шифрования использовал простую литорею, а она не действует! А дело в том, что шифр Репина ещё проще, чем тот, который применяли мы. Он оставил без изменения все гласные буквы, а согласные записал в две строчки:

Б В Г Д Ж З К Л М Н
П Р С Т Ф Х Ц Ч Ш Щ

Легко заметить, что он исключил и некоторые другие буквы и знаки, но сути дела это не меняет.

Исправьте программу *Литорея*, чтобы она справилась, наконец, с дешифровкой протокола.

2. Способ Андрея Репина легко изменить так, чтобы разгадать шифровку было труднее. Например, можно записать вторую строку букв в обратном порядке:

Б В Г Д Ж З К Л М Н
П Р С Т Ф Х Ц Ч Ш Щ

3. Через некоторое время заведующий детским домом Семён Афанасьевич получил письмо вот такого содержания:

25 - 19,13 19, 2, 5, 13, 10, 13, 19 - 11, 8, 23, 9, 8 - 3, 11, 13, 18, 40 - 8, 18, 12, 2, 7, 2, 12, 40, 18, 25 -25 - 18, 1, 8, 10, 8 - 21, 14, 11, 21 - 21, 14, 11, 21, 12 - 17 - 5, 19, 8, 9, 17, 13 - 11, 10, 21, 9, 17, 13 – 25 - 11, 2, 7, 19, 8 - 4,50 - 21, 20, 13, 23 19, 8 - 5, 19, 13 - 4, 50, 23, 8 - 17, 19, 12, 13, 10, 13, 18, 19, 8 - 19, 2, 4, 23, 27, 11, 2, 12, 40 - 3, 2 - 7, 2, 5, 17 - 15, 10, 2, 7, 11, 2 - 7, 50 - 5, 19, 8, 9, 8, 9, 8 - 11, 8, 4, 17, 23, 17, 18, 40 - 19, 8 - 7, 18, 13 - 10, 2, 7, 19, 8 – 21 - 7, 2, 18 - 19, 17, 24, 13, 9, 8 - 19,13 7, 50, 14, 11, 13, 12 - 24, 13, 23, 8, 7, 13, 1 - 15, 10, 13, 6, 11, 13 - 7, 18, 13, 9, 8 - 16, 13, 19, 17, 12 - 18, 7, 8, 4, 8, 11, 21 - 18, 7, 8, 4, 8, 11, 2 - 11,23,25 - 19, 13, 9, 8 - 9, 23, 2, 7, 19, 8, 13 - 7, 50 - 22, 8, 12, 17, 12, 13 - 18, 11, 13, 23, 2, 12, 40 - 7,18,13 - 15, 8 - 11, 10, 21, 9, 8, 5, 21 - 19, 8 - 21 - 7, 2, 18 19, 17, 24, 13, 9, 8 - 19, 13 - 7, 50, 14, 11, 13, 12.

Он тут же хотел обратиться за помощью к Алексею Саввичу, разгадавшему первый шифр Репина (а вы, наверное, догадались, что письмо было именно от него), но потом решил, что и сам справится с новой головоломкой.

Заведующему очень помогла догадка: каждое число заменяет одну и ту же букву. Затем он обратил внимание на то, что некоторые числа встречаются *чаще* других, значит, им должны в русском языке соответствовать буквы, которые также встречаются в словах чаще, чем другие. Ну и наконец, иногда буквы образуют характерные *сочетания*, по которым можно найти короткие слова, а потом …

А вот что было потом, прочитайте в повести *Дорога в жизнь*. Там вы заодно и узнаете, почему глава называется *У вас ничего не выйдет*. А ещё лучше – постарайтесь сначала самостоятельно прочитать письмо. Для решения задачи компьютер вам

не понадобится, а вот *таблица частоты русских букв* вполне может пригодиться:

Пробел	0,2005		
О	0,0764	Ы	0,0143
Е	0,0732	Ь	0,0138
А	0,0629	З	0,0133
И	0,0577	Й	0,0125
Т	0,0549	Б	0,0114
Н	0,049	Ч	0,0094
Р	0,0459	Г	0,0083
С	0,0404	Ю	0,0081
В	0,0355	Ж	0,0079
П	0,033	Х	0,0048
К	0,0302	Щ	0,0042
Л	0,0299	Ф	0,0036
М	0,0275	Ш	0,0026
Д	0,0265	Э	0,0023
У	0,0222	Ц	0,0021
Я	0,0153	Ъ	0,0003

Как видите, в этой таблице буква *Ё* считается как *Е*, и вызвано это вовсе не пренебрежением к букве, а особенностями русского книгопечатания.

Вы можете отыскать и другие таблицы, в которых данные будут несколько отличаться. Это и естественно: подсчёты ведь можно проводить по-разному. Но в целом и эта таблица даёт правильное представление.

4. Разработайте проект *Ё-моё*. Из нашего словаря составьте список слов, в которые входит буква *Ё*. Задача несложная, но интересная: вы узнаете, сколько таких слов в русском языке.

5. Добавьте к программе *Тарабарская грамота* подпрограмму, *декодирующую* зашифрованный текст.

Урок 33. Занимательная биология

Поскольку биология не относится к числу точных наук, то совсем непросто подобрать интересные примеры использования компьютеров на уроках биологии, но три проекта, которые мы разработаем на этом уроке, и достаточно занимательны, и несомненно полезны, поскольку вы сможете доставить немало приятных минут своим близким, родственникам и друзьям, подвергнув их компьютерному тестированию. Не нужно относиться к его результатам излишне серьёзно, но и полностью отвергать их тоже не стоит.

Контрольное взвешивание, или Веское приложение

Хорошего человека должно быть много.

Коварное заблуждение

Сначала мы поможем себе и своим близким приблизиться к идеалу. Пока только в смысле веса.

Некоторые специалисты считают, что *идеальный вес* здорового человека можно определить по формулам:

P = (3 x A - 450 + B) x 0,25 + 45 - для мужчин,
P = (3 x A - 450 + B) x 0,225 + 40,4 - для женщин,

где под загадочными буквами *A* и *B* скрываются *рост* испытуемого (в сантиметрах) и *возраст* (в годах), соответственно.

Согласимся с ними и напишем чрезвычайно полезную для дома и семьи программу, которая позволит вам и всем окружающим оценить свои материальные (опять же только в смысле веса) достоинства.

Какой же может быть программа, основанная на этих незамысловатых формулах? - Ясно, что главную роль в ней должна играть

процедура, вычисляющая идеальный вес по заданным параметрам – полу «подопытного индивидуума», его возрасту и росту. С написанием процедуры проблем не будет, ведь достаточно перевести на компьютерный язык обычные математические выражения. Чтобы процедура «знала» исходные величины, введём *переменные*, в которых будем хранить текущие значения *возраста* и *роста*:

```
ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ ИДЕАЛЬНОГО ВЕСА
GraphicsWindow.Title="Вес"

'var
ves=0
rost=0
vozrast=0
```

Размер *окна* и картинку для *фона* можно выбрать по своему вкусу и настроению:

```
GraphicsWindow.Hide()
GraphicsWindow.Width= 600
GraphicsWindow.Height=390
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"

height=GraphicsWindow.Height
width= GraphicsWindow.Width

background = ImageList.LoadImage(Program.Directory +
"/mw.jpg")
GraphicsWindow.DrawImage(background, 0, 0)

'шрифт:
GraphicsWindow.BrushColor="Red"
GraphicsWindow.FontBold="True"
GraphicsWindow.FontSize=16
```

Нам нужно позаботиться о том, чтобы пользователь мог быстро и удобно передавать в программу необходимые значения. Так как

пол может быть либо мужской, либо женский, то поставим на форму две *кнопки* с буквами *Эм и Жо*, как говаривал Лёлик из комедии *Бриллиантовая рука*. Пользователь выберет ту кнопку, которую пожелает (например, кто-то захочет узнать свой идеальный вес при перемене пола или больше узнать о девочках).

```
'КНОПКИ

'Вычисляем вес для мужчин:
btnM=Controls.AddButton("М", 290, height-40)
Controls.SetSize(btnM, 48, 32)
'Вычисляем вес для женщин:
btnW=Controls.AddButton("Ж", 10, height-40)
Controls.SetSize(btnW, 48, 32)
Controls.ButtonClicked=OnClick
```

Возраст и рост могут принимать большое количество значений, поэтому их придётся вводить с клавиатуры в два *текстовых поля*:

```
'ТЕКСТОВЫЕ ПОЛЯ

'Возраст:
txtVozrast=Controls.AddTextBox(180, 210)
Controls.SetSize(txtVozrast,160,26)
GraphicsWindow.DrawText(350, 214, "<- Возраст в годах")

'Рост:
txtRost=Controls.AddTextBox(180, 248)
Controls.SetSize(txtRost,160,26)
GraphicsWindow.DrawText(350, 252, "<- Рост в сантиметрах")
```

Нам потребуется ещё одно *текстовое поле* – для того, чтобы предъявить пользователю его идеальный вес:

```
'Идеальный вес:
txtVes=Controls.AddMultiLineTextBox(180, 290)
Controls.SetSize(txtVes,300,50)
```

Вся премудрость нашей программы заключена в процедуре-обработчике нажатия кнопки *OnClick*:

```
'Печатаем результат тестирования
Sub OnClick
  vozrast= Controls.GetTextBoxText(txtVozrast)
  rost=    Controls.GetTextBoxText(txtRost)

  s=""
  If (vozrast = "") Or (vozrast < 1) Or (vozrast > 120) Then
    s= "Проверьте возраст!"
  ElseIf (rost = "") Or (rost < 60) or (rost > 240) Then
    s= "Проверьте рост!"
  EndIf

  If (s<>"") Then
    Goto print
  EndIf

  btn= Controls.LastClickedButton
  If (btn= btnM) Then
    ves = (3 * rost  - 450 + vozrast) * 0.25 + 45    '-  для
мужчин
  Else
    ves = (3 * rost  - 450 + vozrast) * 0.225 + 40.4 '- для
женщин
  EndIf

  'корректируем отрицательные значения:
  if ves< 0 then
    ves = 4
  EndIf

  GraphicsWindow.BrushColor="Blue"
  s= "Идеальный вес равен " + Math.Round(ves) + " кг"
 print:
  Controls.SetTextBoxText(txtVes, s)
EndSub
```

Действует-злодействует (любимое занятие Бабы-Яги) она так.

Считываем введённые пользователем данные в переменные *vozrast* и *rost*, после чего проверяем биологическую грамотность пользователя или злобный юмор шутника. Если данные выходят за пределы разумения, то расчёты вести глупо, поэтому мы сразу же печатаем последнее предупреждение.

Если пользователь успешно преодолел ввод анкетных данных, то по имени нажатой кнопки мы определяем пол «персонажа», в соответствии с которым и выбираем одну из двух формул.

На всякий случай заменяем отрицательный вес более осмысленным. Это вызвано тем, что формулы для расчёта веса не универсальны, то есть действительны только для разумных сочетаний параметров, поэтому идеальный вес Кощея Бессмертного или дядьки Черномора по ней вычислять нельзя.

Найдя идеальный вес, округляем его до целого числа (формулы, естественно, не настолько точны, чтобы рассчитывать ещё и граммы), и выводим в *текстовое поле txtVes* строку с полезной информацией (Рис. 33.1).

Рис. 33.1. Пора худеть?

Исходный код программы находится в папке **Вес**.

Жиропонижающее средство

90-60-90

Фигурка мирового класса

Продолжаем исследования рода человеческого с помощью компьютера. Немного подправив программу *Вес*, мы легко определим *жирность* тела любого гражданина (или гражданки).

Дотошная наука выяснила, что тело молодых здоровых мужчин содержит около 15%, а тело женщин – около 22% жира. Жирность произвольно выбранного «тела» приблизительно оценивается по формуле:

Ж = (Вес - Р) : Вес х 100 + 15 - для мужчин,
Ж = (Вес - Р) : Вес х 100 + 22 - для женщин,

где *Р* – идеальный вес, который определяется по предыдущим формулам.

Не будем целиком переписывать всю программу *Вес*, а адаптируем её к «новым реалиям». Для этого создадим новую папку **Жир**, в которую перепишем исходный текст программы *Вес* под новым названием.

Добавим ещё одну *переменную* – для хранения жирности тела:

```
zhir=0
```

И *текстовое окно* – для ввода *действительного* веса пользователя. Заодно придётся подправить и другие *текстовые поля*:

```
'Возраст:
txtVozrast=Controls.AddTextBox(180, 190)
Controls.SetSize(txtVozrast,160,26)
GraphicsWindow.DrawText(350, 194, "<- Возраст в годах")

'Рост:
txtRost=Controls.AddTextBox(180, 224)
Controls.SetSize(txtRost,160,26)
```

```
GraphicsWindow.DrawText(350, 227, "<- Рост в сантиметрах")

'Вес:
txtVes=Controls.AddTextBox(180, 259)
Controls.SetSize(txtVes,160, 26)
GraphicsWindow.DrawText(350, 262, "<- Вес в килограммах")

'Жирность:
txtZhir=Controls.AddMultiLineTextBox(180, 300)
Controls.SetSize(txtZhir,300,50)
```

Конечно, нужно потрудиться и над обработкой данных пользователя с учётом новых формул:

```
'Печатаем результат тестирования
Sub OnClick
  vozrast= Controls.GetTextBoxText(txtVozrast)
  rost=    Controls.GetTextBoxText(txtRost)
  v=       Controls.GetTextBoxText(txtVes)
  s=""
  If (vozrast = "") Or (vozrast < 1) Or (vozrast > 120) Then
    s= "Проверьте возраст!"
  ElseIf (rost = "") Or (rost < 60) or (rost > 240) Then
    s= "Проверьте рост!"
  ElseIf (v = "") Or (v < 10) or (v > 240) Then
    s= "Проверьте вес!"
  EndIf

  If (s<>"") Then
    Goto print
  EndIf

  btn= Controls.LastClickedButton
  If (btn= btnM) Then
    ves = (3 * rost  - 450 + vozrast) * 0.25 + 45    '-  для
мужчин
    zhir= (v - ves) / v * 100 + 15
  Else
    ves = (3 * rost  - 450 + vozrast) * 0.225 + 40.4 '-  для
женщин
    zhir= (v - ves) / v * 100 + 22
  EndIf
```

```
  'корректируем  значения:
  if (zhir< 0) then
    zhir= 0
  elseIf (zhir > 100) then
    zhir= 100
  EndIf

  GraphicsWindow.BrushColor="Blue"
  s= "Жирность равна " + Math.Round(zhir) + "%"
  print:
  Controls.SetTextBoxText(txtZhir, s)
EndSub
```

Нужно добавить проверку для введённого веса, а также расчёт жирности по формулам, после чего можно переходить к контрольному взвешиванию (Рис. 33.2).

Рис. 33.2. А девочки-то «жирнее» мальчиков!

Исходный код программы находится в папке **Жир**.

Сколько кожи на человеке?

Кожа да кости!

Требование Славы Зайцева к моделям

Не лезьте из кожи!

Царевна Лягушка

Составим ещё одну познавательную программу по биологии. На этот раз мы вычислим площадь поверхности человеческого тела (то есть кожи). С точки зрения геометрии, фигура ка весьма «причудлива», поэтому не существует точных формул для определения площади тела.

Мы воспользуемся формулой *Бойде*, которая позволяет приближённо вычислить нужную нам величину (не пугайтесь - формула заковыристая):

$$S = (P \times 1000)^{(\lg(1/P)+35,75)\,/\,53,2} \times H^{0,3} : 3118,2,$$

где

S – площадь кожи в квадратных метрах

P – вес человека в килограммах

H – его рост в сантиметрах

Так как для вычисления площади тела нужно знать *рост* и *вес* человека, то за основу нашей программы мы возьмём предыдущее наше творение - проект *Вес*, скопировав его в новую папку **Кожа**.

```
' ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ ПЛОЩАДИ КОЖИ ЧЕЛОВЕКА

GraphicsWindow.Title="Кожа"

'var
ves=0
rost=0
s=0
```

Подправим *элементы управления*:

```
' КНОПКА

' Вычисляем площадь кожи:
btnKozha=Controls.AddButton("Вычислить!", 180, height-40)
Controls.SetSize(btnKozha, 120, 32)
Controls.ButtonClicked=OnClick
```

```
' ТЕКСТОВЫЕ ПОЛЯ

' Возраст:
txtVes=Controls.AddTextBox(180, 210)
Controls.SetSize(txtVes,160,26)
GraphicsWindow.DrawText(350, 214, "<- Вес в килограммах")

' Рост:
txtRost=Controls.AddTextBox(180, 248)
Controls.SetSize(txtRost,160,26)
GraphicsWindow.DrawText(350, 252, "<- Рост в сантиметрах")

' Площадь кожи:
txtKozha=Controls.AddMultiLineTextBox(180, 290)

Controls.SetSize(txtKozha,360,50)
```

Довольно странно, но формула Бойде никак не учитывает половую принадлежность испытуемого, хотя пропорции мужского и женского тела заметно отличаются. Однако доверимся науке и уберём лишнюю кнопку.

Вычислять площадь тела, как обычно, мы будем в процедуре *OnClick*:

```
' Печатаем результат вычислений
Sub OnClick
  ves= Controls.GetTextBoxText(txtVes)
  rost=    Controls.GetTextBoxText(txtRost)
  str=""
  If (ves = "") Or (ves < 10) Or (ves > 240) Then
    str= " Проверьте вес!"
  ElseIf (rost = "") Or (rost < 60) or (rost > 240) Then
    str= " Проверьте рост!"
  EndIf

  If (str<>"") Then
    Goto print
  EndIf

  ' вычисляем площадь кожи по формуле Бойде:
  s= (Math.Log(1 / ves) + 35.75) / 53.2
```

```
  s= Math.Power((ves * 1000), s) * Math.Power(rost, 0.3) /
0.31182

  str= " Площадь кожи равна " + Math.Round(s) + " кв. см"
  print:
  Controls.SetTextBoxText(txtKozha, str)

EndSub
```

Переводим формулу Бойде на математический диалект бейсиковского языка и выводим на экран площадь тела в квадратных *сантиметрах* (отбрасываем десятичные знаки с помощью метода *Round*). Как говорил Удав: *А в попугаях я гораздо длиннее!* (Рис. 33.3).

Обратите внимание, что буквой *s* мы обозначили площадь тела, как это обычно и принято в математике, поэтому идентификатор строки для вывода результатов пришлось изменить - *str*.

С помощью этой программы вы сможете вычислить площади всех доступных вам тел, к вящему удовольствию их «владельцев» (если они ещё не потеряли вкус к жизни после первых двух экспериментов).

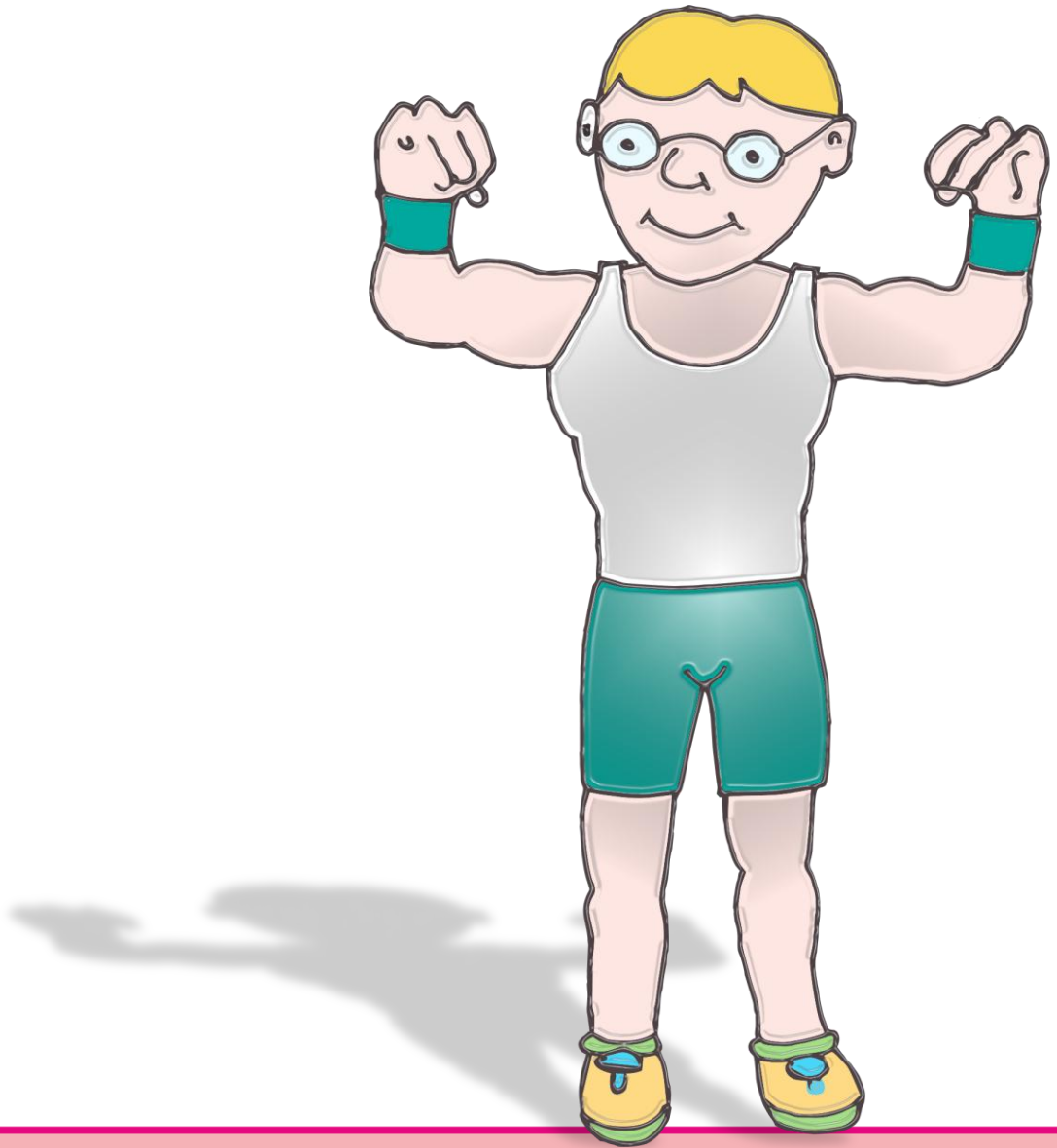
Рис. 33.3. С наукой не поспоришь!

Исходный код программы находится в папке **Кожа**.

1. Перед тем как предлагать программу *Вес* другим людям, очень полезно испытать её на себе. Запустите программу, введите данные, оцените все достоинства и недостатки интерфейса. И – непременно! – постарайтесь её улучшить.

2. Протестируйте программу *Жир* на ком только сможете. Наверняка это доставит и вам, и всем испытуемым массу удовольствия и запомнится на всю жизнь.

3. Посчитайте, сколько татуировок сможет разместить на себе Владимир Винокур, если средняя площадь татуировки равна 100 квадратным сантиметрам.



Урок 34. Занимательная психология

Очень часто в популярных тестах по психологии встречаются задания, в которых нужно быстро и правильно *сосчитать* какие-нибудь предметы: цветки, бабочек, якоря – да что угодно. Только не думайте, что это пустая забава!

В замечательной книге *Вам – взлёт!* Анатолий Маркуша рассказал о таком случае из фронтовой жизни.

Когда в нелётную погоду все лётчики полка скучали и слонялись по аэродрому, старший лейтенант Нико Ломия играл в спички: бросит несколько штук, быстро взглянет на них, сгребёт в кулак, снова бросит... И так просиживал он целые дни. Конечно, товарищи начали переживать за него – не сошёл ли он с ума от тоски? Или гадает на хорошую погоду? Дошла эта история и до командира полка. Он вызвал лейтенанта, а тот объяснил ему, что таким необычным способом он тренирует зрительную память: бросит горсть спичек и пытается с одного взгляда определить, сколько их. Он уже справлялся с дюжиной спичек, но цель у него была – 50 спичек.

А через полгода он стал лучшим воздушным разведчиком фронта. На большой скорости пролетая над вражеской территорией, он мгновенно подсчитывал и запоминал: орудий столько, самолетов – столько, танков – столько. И никогда не ошибался в своих подсчётах.

На следующей странице вы найдёте одно из таких заданий (Рис. 34.1). Вы можете просто пересчитать на время, сколько там разных цветков и листьев, но еще лучше - сначала прикиньте, а потом проверьте, насколько вы ошиблись в своих предположениях.

Вы и сами легко можете сделать такие задачки – для себя и своих товарищей по классу. Возьмите чистый лист белой бумаги и в беспорядке нарисуйте какие-нибудь предметы. Можно использовать и красивые наклейки, тогда решать такие задачки будет ещё интереснее.

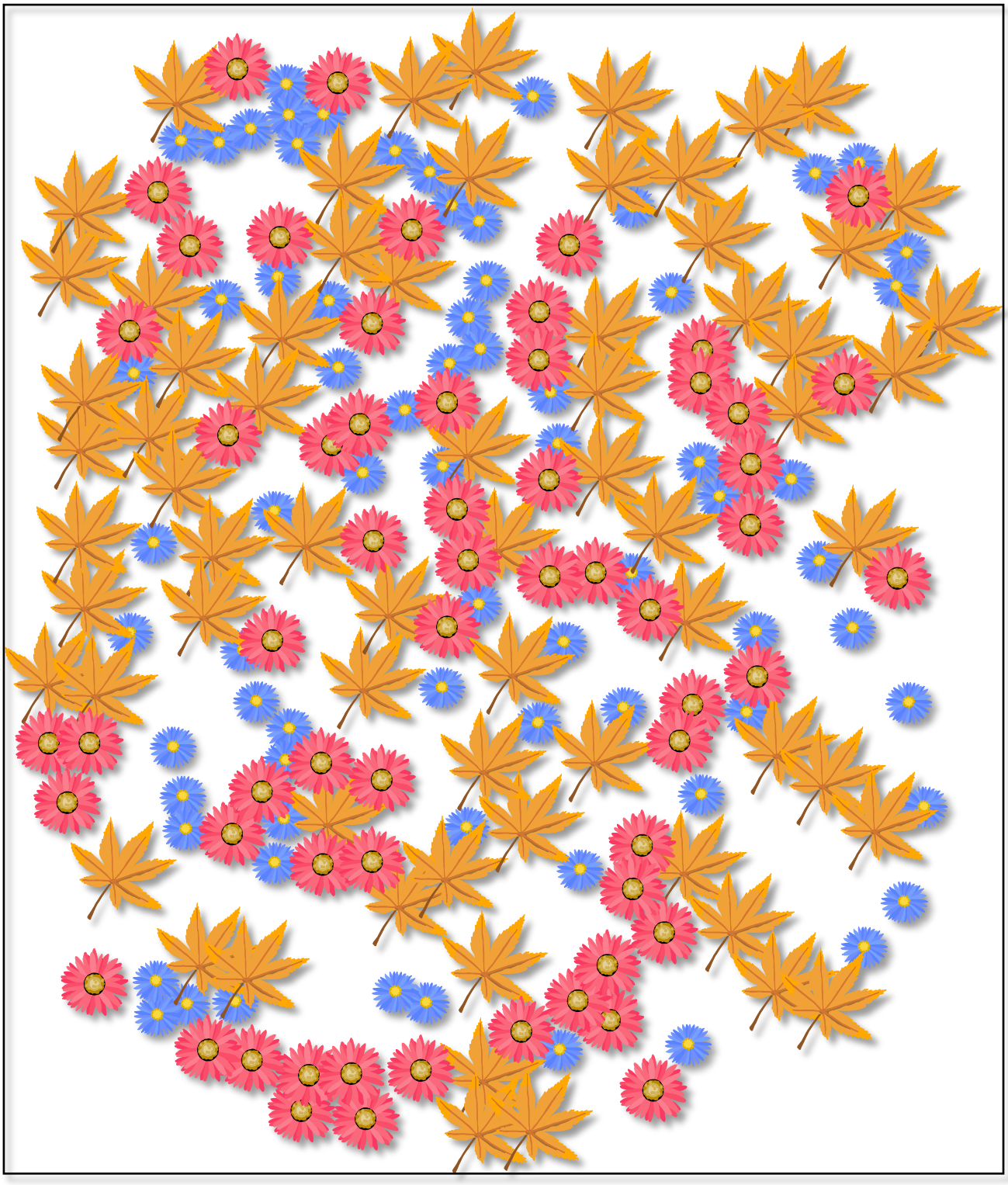


Рис. 34.1. Сосчитайте как можно быстрее, сколько здесь

1. Синих цветов _____
2. Красных цветов _____
3. Желтых листьев _____

Психологическая считалка

Но самоделки - это потом, а сейчас мы поступим иначе – напишем программу, которая будет сама делать за нас такие задачки. Для почина мы будем рисовать разноцветные кружочки, но вы можете заменить их (или добавить к ним) квадратами или треугольниками. На другом уроке мы усовершенствуем программу так, чтобы она могла выводить на экран любые картинки, а не только простые геометрические фигуры.

А начнём мы новый проект **Психологическая считалка** с того, что уже хорошо знаем, то есть создадим *графическое окно* для нового приложения:

```
GraphicsWindow.Title="Психологическая считалка"

'const
BackgroundColor="#2F4F4F"

GraphicsWindow.Hide()
GraphicsWindow.Width= 640
GraphicsWindow.Height=480
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"
GraphicsWindow.BackgroundColor= BackgroundColor

height=GraphicsWindow.Height
width= GraphicsWindow.Width
```

Для определения времени, затраченного на выполнение теста, нам потребуется *таймер*, который будет срабатывать каждую секунду.

```
'таймер:
Timer.Interval=1000
Timer.Pause()
Timer.Tick=OnTick
```

Процедура-обработчик выводит время в *текстовое поле*:

```
'Отсчитываем время тестирования:
Sub OnTick
  time= time+ Timer.Interval/1000
  Controls.SetTextBoxText(txtTime,"Время: " + time)
EndSub
```

Его нужно создать:

```
'ТЕКСТОВЫЕ ПОЛЯ

'Время тестирования:
txtTime=Controls.AddTextBox(340, 10)
Controls.SetSize(txtTime,100,26)
Controls.HideControl(txtTime)
```

Не удивляйтесь комментарию ТЕКСТОВЫЕ ПОЛЯ: нам придётся создать ещё одно *текстовое поле* – для ввода ответа игрока.

```
'Проверка ответа:
txtOtvet=Controls.AddTextBox(180, 10)
Controls.SetSize(txtOtvet,32,26)
Controls.HideControl(txtOtvet)
```

Не обойтись нам и без пары *кнопок*:

```
'КНОПКИ

'Button1 - начинаем тест:
btnStart=Controls.AddButton("Начать тестирование", 10, 10)
'Button2 - проверка:
btnProv=Controls.AddButton("Проверить", 220, 10)
Controls.ButtonClicked=OnClick
Controls.HideControl("Button2")
```

На этом создание нехитрого интерфейса программы закончено - показываем окно на экране (Рис. 34.2):

```
GraphicsWindow.Show()
```

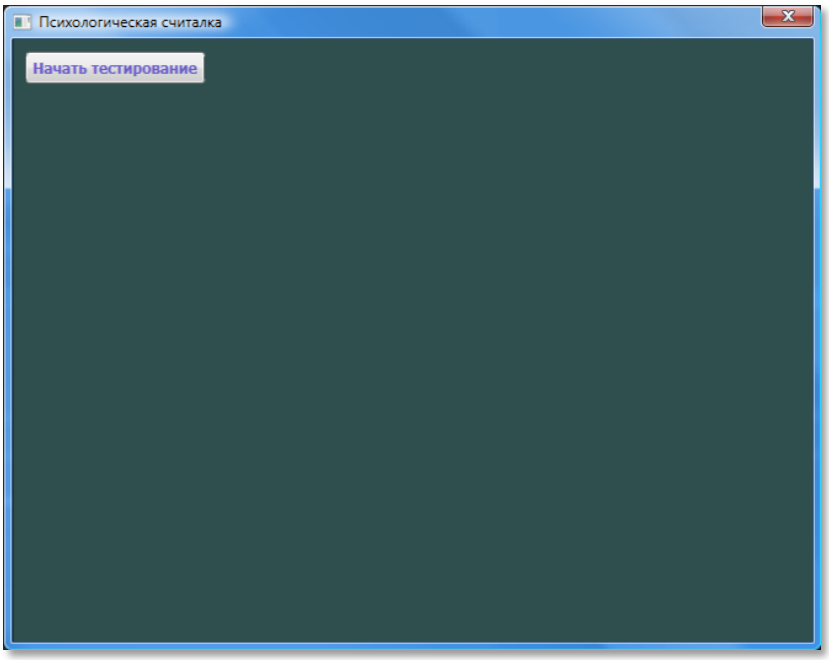


Рис. 34.2. Окно приложения

Одинокая кнопка не даст пользователю ни единого шанса запутаться в интерфейсе. Поскольку выбора у него нет, он нажмёт кнопку *Начать тестирование*, и, как мы видим по коду, попадёт в процедуру-обработчик нажатия на кнопку *OnClick*:

```
Sub OnClick
  btn=Controls.LastClickedButton
  If btn="Button1" then 'начать тестирование
    Controls.HideControl(btnStart)
    Controls.ShowControl(btnProv)
    createTest()
    Controls.SetTextBoxText(txtTime,"")
    Controls.ShowControl(txtTime)
    Controls.SetTextBoxText(txtOtvet,"")
    Controls.ShowControl(txtOtvet)
    time=0
    Timer.Resume()
  Else   'проверить ответ
    Controls.HideControl("Button2")
    Controls.ShowControl("Button1")
    Timer.Pause()
    If nKrug= Controls.GetTextBoxText(txtOtvet) then
      s="ПРАВИЛЬНО!"
      GraphicsWindow.BrushColor= "Yellow"
    Else
```

```
      s="НЕПРАВИЛЬНО!"
      GraphicsWindow.BrushColor= "Red"
    endif
    GraphicsWindow.FontSize= 48
    GraphicsWindow.DrawText(140,height/2-24, s)
  EndIf
EndSub
```

Так как нажатие всех кнопок в нашей программе обрабатывает одна и та же процедура, то нам нужно, в первую очередь, узнать, какую именно кнопку нажал пользователь:

```
btn=Controls.LastClickedButton
```

Если это *первая* кнопка, то мы начинаем тестирование, если *вторая* - проверяем ответ пользователя. Для тестирования нам нужно создать тест с помощью процедуры *createTest,* а также спрятать одни элементы управления и показать другие, чтобы пользователь в них не запутался.

Итак, кнопка нажата – создаём новый тест. Для этого мы просто разбрасываем по клиентской области окна **разноцветные** кружочки:

```
Sub createTest
  'очистить окно:
   GraphicsWindow.BrushColor=BackgroundColor
   GraphicsWindow.FillRectangle(0,40,width,height-40)
  'число кружков::
  nKrug=Math.GetRandomNumber(30)+ 20
  'отладка:
  'GraphicsWindow.Title= nKrug
  for i= 1 to nKrug
    'выбираем случайный радиус кружка:
    radius=Math.GetRandomNumber(16)+ 20
    ' выбираем случайные координаты  кружка:
    x = Math.GetRandomNumber(width-2*radius)
    y = Math.GetRandomNumber(height-2*radius-40) + 40
    ' выбираем случайный цвет для  кружка:
    clr= GraphicsWindow.GetRandomColor()
    GraphicsWindow.BrushColor=clr
    'рисуем цветной кружок:
    GraphicsWindow.FillEllipse(x, y, 2*radius,2*radius)
```

```
  EndFor
EndSub
```

Прежде всего, нужно подготовить рабочее место, то есть убрать с канвы всё лишнее:

```
'очистить окно:
 GraphicsWindow.BrushColor=BackgroundColor
 GraphicsWindow.FillRectangle(0,40,width,height-40)
```

Метод *GraphicsWindow.Clear* не проходит, потому что под его горячую руку попадет всё, что уже нарисовано на канве, в том числе и все элементы управления! Поэтому мы нарисуем закрашенный цветом фона окна прямоугольник нужных нам размеров. Глядя на первую картинку, можно подумать, что канва и без того чистая, но вот что с ней станет после нажатия на кнопку (Рис. 34.3).

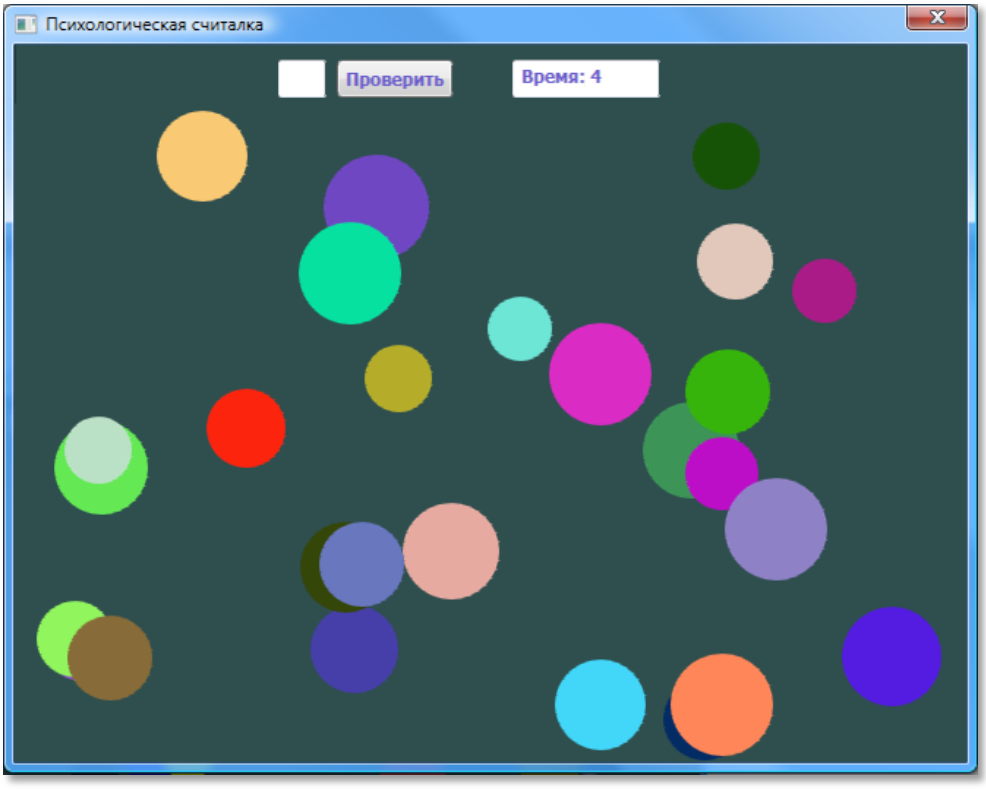


Рис. 34.3. А вот и цветные кружочки!

Если мы захотим продолжить тестирование, то следующие кружочки будут нарисованы поверх прежних, и мы вместо теста получим кашу!

Число кружочков для каждого тестирования нужно выбирать другим, иначе будет неинтересно. В данном примере их будет от 21 до 50:

```
nKrug=Math.GetRandomNumber(30)+ 20
```

Вы легко можете выбрать другой интервал, например, для младших школьников лучше подойдет диапазон от 3 до 11.

При отладке программы очень важно знать, сколько кружков на поле, чтобы не пересчитывать их всякий раз заново, ведь программу придётся доводить до ума довольно долго:

```
  'отладка:
  'GraphicsWindow.Title= nKrug
```

Всегда вставляйте в свои программы *отладочные операторы*. А когда они станут не нужны, просто закомментируйте их.

Теперь можно нарисовать на экране заданное число кружков. Их размеры, цвет и положение также выбираем случайным образом, чтобы задания отличались друг от друга:

```
radius=Math.GetRandomNumber(16)+ 20
```

Радиус кружков задайте по своему вкусу!

Выбирать место для нового кружка следует так, чтобы он целиком помещался на канве, а также не вторгался в область канвы, где расположены элементы управления программой:

```
    x = Math.GetRandomNumber(width-2*radius)
    y = Math.GetRandomNumber(height-2*radius-40) + 40
```

Осталось нарисовать кружок на экране:

```
   GraphicsWindow.FillEllipse(x, y, 2*radius,2*radius)
```

Когда построение теста закончено, программа начинает отсчёт времени:

```
    Controls.SetTextBoxText(txtTime,"")
    Controls.ShowControl(txtTime)
    Controls.SetTextBoxText(txtOtvet,"")
    Controls.ShowControl(txtOtvet)
    time=0
    Timer.Resume()
```

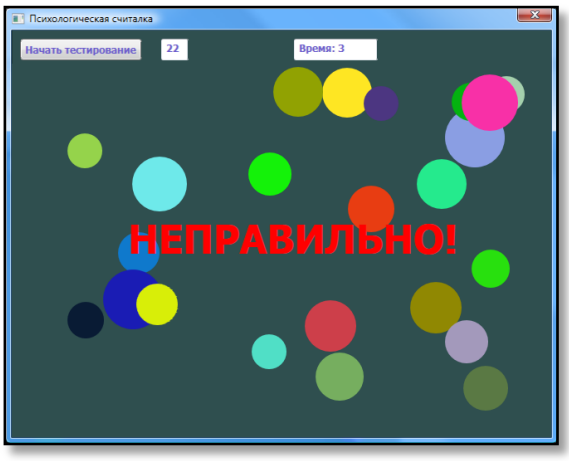
Испытуемый, подсчитав кружки на поле, нажимает кнопку *Проверить*, после чего программа переходит к обработке этого события в процедуре *OnClick*.

Таймер останавливается и показывает затраченное на тестирование время:

```
    Timer.Pause()
```

Затем в процедуре сравнивается число, введённое пользователем с клавиатуры в *текстовое поле txtOtvet*, с действительным числом кружков, и выдаёт *результат* теста (Рис. 34.4):

```
    If nKrug= Controls.GetTextBoxText(txtOtvet) then
      s="ПРАВИЛЬНО!"
      GraphicsWindow.BrushColor= "Yellow"
    Else
      s="НЕПРАВИЛЬНО!"
      GraphicsWindow.BrushColor= "Red"
    endif
    GraphicsWindow.FontSize= 48
    GraphicsWindow.DrawText(140,height/2-24, s)
  EndIf
```



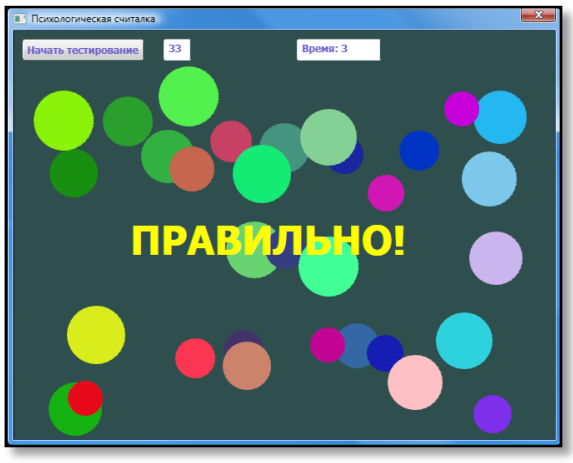


Рис. 34.4. Пользователь погорячился! Результат тренировок налицо!

На рисунках видно (и по исходному тексту тоже!), что снова появилась призывная кнопка *Начать тестирование - Психологическая считалка* снова в бою!

Исходный код программы находится в папке **Психологическая считалка**.

1. В случае неправильного ответа программа только сообщает пользователю, что он неверно подсчитал кружочки. Добавьте к строке *правильное* число кружков, чтобы пользователь мог оценить, насколько он ошибся.

2. Так как цвет кружков выбирается случайно, то он может совпасть с цветом фона *BackgroundColor="#2F4F4F"* и тогда будет невидим для пользователя. Вероятность такого события очень невелика, но её следует учесть, например, так:

```
nextColor:
clr= GraphicsWindow.GetRandomColor()
If (clr=BackgroundColor) Then
  Goto nextColor
EndIf
GraphicsWindow.BrushColor=clr
```

3. Другая неприятность заключается в том, что большой кружок может полностью закрыть маленький, который появился на экране раньше. То, что некоторые кружки частично перекрывают друг друга, это неплохо, так как это делает задачу более трудной. Однако полностью закрытый кружок вообще не видно, поэтому результат тестирования будет неверным. Чтобы этого избежать, будем проверять, куда попадает центр нового кружка. Если на пустое место, то всё нормально, кружок можно рисовать, иначе нужно повторить попытку:

```
n=0
nextXY:
```

```
    radius=Math.GetRandomNumber(20)+ 35
    x = Math.GetRandomNumber(width-2*radius)
    y = Math.GetRandomNumber(height-2*radius-40) + 40
    If (n< 100) and
(GraphicsWindow.GetPixel(x+radius,y+radius) <>
BackgroundColor) Then
      n=n+1
      Goto nextXY
    EndIf
```

Для чего введена переменная *n*? – Она подсчитывает число неудачных попыток нарисовать кружок в чистом поле. После 100 попыток кружок будет нарисован наудачу. Действительно, на поле может возникнуть ситуация, когда место для нового кружка будет трудно найти (а если кружков много и они большого размера, то вообще не найти) и программа будет тратить много времени на создание теста (или вообще зациклится!). Конечно, такую ситуацию мы допустить не должны.

Если размеры кружков очень сильно различаются, то и в этом случае может произойти полное «затмение» некоторых кружков со всеми вытекающими отсюда последствиями. Можно несколько улучшить ситуацию, - что мы и сделаем на одном из следующих уроков,- но полностью предотвратить её нельзя. Поэтому ограничивайте количество кружков и не делайте их слишком разными «по росту»!

Исходный код программы находится в папке **Психологическая считалка**.

АСТРОНОМИЯ

Урок 35. Звёздное небо

Открылась бездна, звезд полна…

М.В.Ломоносов

Трудно быть богом

Братья Стругацкие

Если бы мы жили в центре Галактики, то звёздное небо очень бы напоминало картинку из *Психологической считалки*. Но нам в смысле звёздной красоты повезло значительно меньше, поскольку мы живем на периферии, на забытой богом спирали. Куда ни глянь: звёзды мелкие и **жёлтые**.

На самом деле звёзды БОЛЬШИЕ и **разноцветные**, просто они нам такими кажутся из-за огромного расстояния до них. Правда, даже в самый сильный телескоп звёзды всё равно выглядят точками, хотя и **цветными**.

Если усыпать канву маленькими кружочками, то мы получим картину, вполне *напоминающую* настоящее небо.

Чтобы небо выглядело натурально, нам придётся задать координаты *всех* видимых глазом звёзд, а это несколько тысяч штук! Также нам будет необходимо учесть их цвет и блеск (звёздную величину).

Поскольку нам не нужно заботиться о перекрывании звёзд и прочих премудростях, с которыми мы столкнулись в психологическом тесте, то программа получится очень простая.

Звёзды и на самом деле закрывают друг друга, например, затменные переменные звёзды делают это периодически.

Начнём новый проект **Звёздное небо** с заголовка окна и цвета неба (Рис. 35.1, слева):

```
GraphicsWindow.Title="Звёздное небо"

'const
BackgroundColor="MidnightBlue" 'цвет неба
```

Если хотите, можете сделать небо совсем **чёрным** (Рис. 35.1, справа), тогда звёзды будут казаться более яркими:

```
BackgroundColor="Black"
```

Нам будет достаточно одной *кнопки*:

```
btnStart=Controls.AddButton("Новое небо", 10, 10)
Controls.ButtonClicked=OnClick
```

Вот так, с помощью единственной кнопки, мы будем менять картину мира в галактических масштабах!

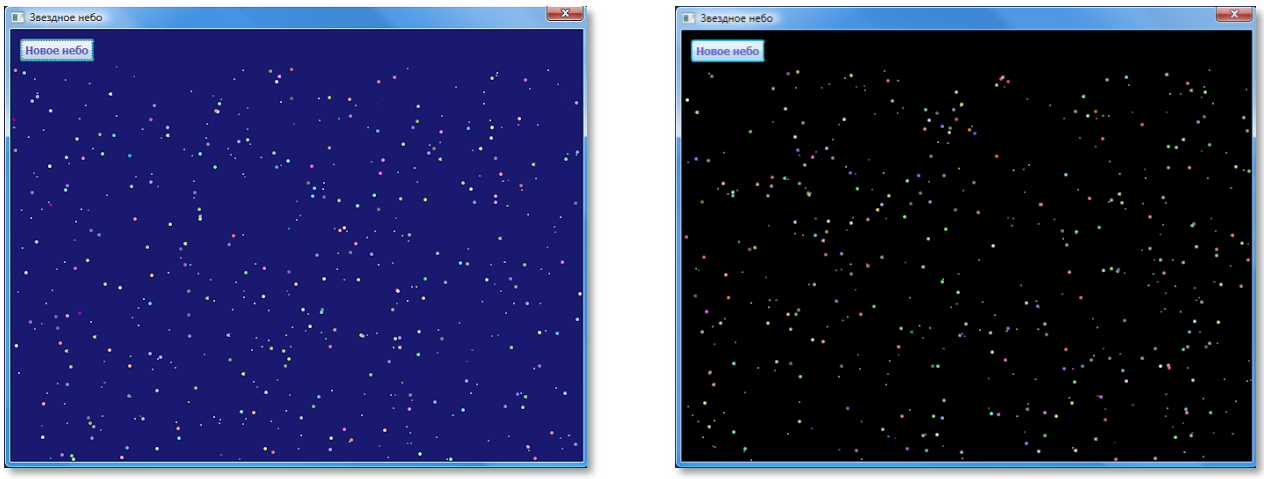


Рис. 35.1. Звёзд много и все они разные, но нет на искусственном небе наших родных созвездий Ориона, Лиры, Пегаса, Лебедя и обеих Медведиц!

Как только мы нажмём заветную кнопку творца

```
Sub OnClick
  createSky()
EndSub,
```

будет вызвана процедура *createSky*, где произойдет *Большой взрыв*, и небо украсится мириадами звёзд:

```
Sub createSky
  'очистить небо:
  GraphicsWindow.BrushColor=BackgroundColor
  GraphicsWindow.FillRectangle(0,40,width,height-40)
  'число звезд:
  nStar=Math.GetRandomNumber(100)+ 400
  for i= 1 to nStar
    'выбираем случайный радиус звезды:
    radius=Math.GetRandomNumber(2)
    'выбираем случайные координаты  звезды:
    x = Math.GetRandomNumber(width-2*radius)
    y = Math.GetRandomNumber(height-2*radius-40) + 40
    'выбираем случайный цвет для  звезды:
    r= Math.GetRandomNumber(163) + 92
    g= Math.GetRandomNumber(163) + 92
    b= Math.GetRandomNumber(163) + 92
    clr= GraphicsWindow.GetColorFromRGB(r,g,b)
    GraphicsWindow.BrushColor=clr
    'рисуем звезду:
    GraphicsWindow.FillEllipse(x, y, 2*radius,2*radius)
  EndFor
EndSub
```

С точки зрения программирования, ничего таинственного в этом созидательном акте нет. Единственное, на что следует обратить внимание: мы выбираем случайный цвет звезды не с помощью метода *GetRandomColor*, что было бы проще. Нам нужны достаточно яркие звезды (иначе они потеряются на небе), поэтому мы задаём такие составляющие цвета *r, g, b*, чтобы они были от 93 (самые тусклые) до 255 (самые яркие), а далее формируем из них цвет в методе *GetColorFromRGB:*

```
clr= GraphicsWindow.GetColorFromRGB(r,g,b)
```

Если вы останетесь недовольны видом созданного вами неба, смело жмите на кнопку *Новое небо*!

Исходный код программы находится в папке **Звёздное небо**.

Кроме одиночных и кратных звёзд, с рисованием которых наша программа неплохо справляется, на небе можно увидеть и *звёздные скопления* – группы звезд, связанных общим происхождением и силами гравитации. Они бывают *рассеянные* и *шаровые*.

Самое известное рассеянное звёздное скопление в наших широтах – это *Плеяды*, или *Стожары*. Оно находится в созвездии Тельца и сравнительно недалеко от Солнца, поэтому многие звёзды этого скопления можно видеть невооружённым глазом.

Давайте добавим к нашему звёздному небу несколько рассеянных скоплений (Рис. 35.2). Для нас важно знать, что в них не очень много звёзд и они белого и **голубого** цвета.

Для создания рассеянных звездных скоплений (РЗС) нам потребуется ещё одна *кнопка*:

```
btnAddOpenCluster=Controls.AddButton("Добавить  РЗС",
100, 10)
```

Потребуется переписать и процедуру-обработчик:

```
Sub OnClick
  btn=Controls.LastClickedButton
  If (btn= btnStart) then
    createSky()
  EndIf
  If (btn= btnAddOpenCluster) then
    createOpenCluster()
  EndIf
EndSub
```

Но самое главное – в новой процедуре:

```
'СОЗДАТЬ РАССЕЯННОЕ ЗВЕЗДНОЕ СКОПЛЕНИЕ
Sub createOpenCluster
  'число звезд:
  nStar=Math.GetRandomNumber(10)+ 7
  'диаметр скопления:
  rOC=Math.GetRandomNumber(20)+20
  'левый верхний угол скопления:
  xOC = Math.GetRandomNumber(width-2*rOC)
  yOC = Math.GetRandomNumber(height-2*rOC-40) + 40
```

```
  for i= 1 to nStar
    'выбираем случайный радиус звезды:
    radius=Math.GetRandomNumber(2)
    'выбираем случайные координаты  звезды:
    x = Math.GetRandomNumber(2*rOC)+xOC
    y = Math.GetRandomNumber(2*rOC-40) + 40 + yOC
    'выбираем случайный цвет для  звезды:
    clr=Math.GetRandomNumber(4)
    If (clr= 1) Then
    clr="White"
    EndIf
     If (clr= 2) Then
    clr="AliceBlue"
    EndIf
     If (clr= 3) Then
    clr="LightSkyBlue"
    EndIf
     If (clr= 4) Then
    clr="DeepSkyBlue"
    EndIf
    GraphicsWindow.BrushColor=clr
    'рисуем звезду:
    GraphicsWindow.FillEllipse(x, y,
2*radius,2*radius)
  EndFor
EndSub
```

Пользуясь этими подсказками, напишите новый вариант программы (Рис. 35.3)!

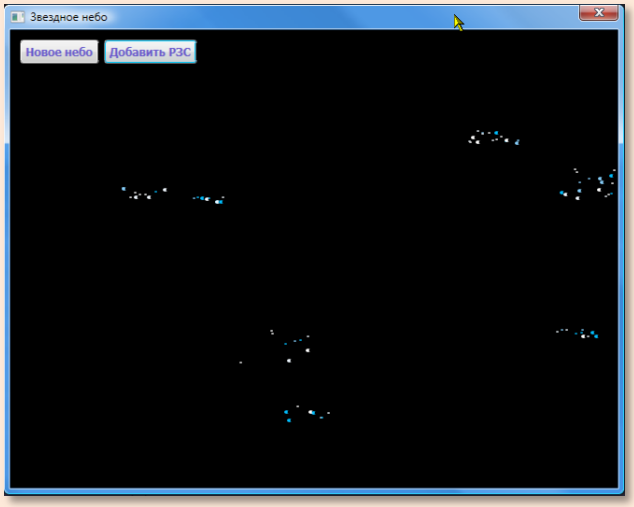


Рис. 35.2. Семь рассеянных скоплений

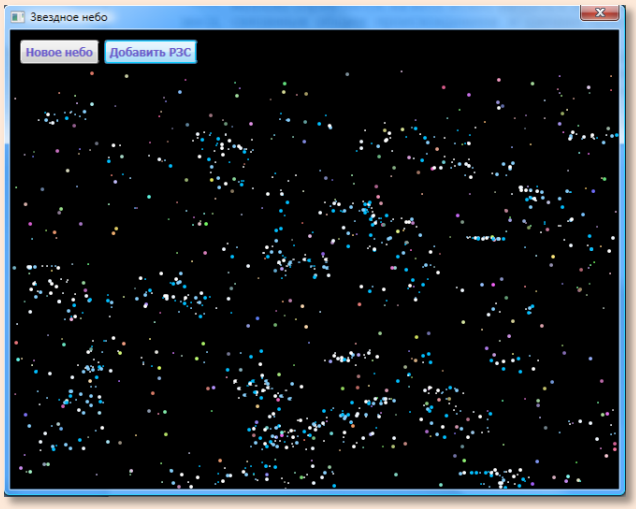


Рис. 35.3. Звёздное небо стало живописнее!

2. Любуясь звёздами, вы, конечно, не могли забыть о *шаровых* звёздных скоплениях, в которых звёзд гораздо больше, чем в рассеянных, и форма у них сферическая, причем концентрация звёзд увеличивается к центру скопления (Рис. 35.4).

Рис. 35.4. Шаровое звёздное скопление

Цвет звёзд в таких скоплениях – от белого до **красного**, то есть можно брать такие же цвета, как для одиночных звёзд в процедуре *createSky*.

Исходный код программы находится в папке **Звёздное небо**.

Урок 36. С первого взгляда!

Когда видишь деньги,
не теряй времени!

Кинокомедия *Бриллиантовая рука*

На уроке *Занимательная психология* мы говорили о задачах, в которых требуется очень быстро, с первого взгляда определить количество предметов. Пример такого психологического теста вы найдёте на следующей странице (Рис. 36.1).

Конечно, решая задание в книге, вы можете и схитрить - подглядывать или считать монетки пальцем. Да и задание только одно – решил и сказочке конец. А ведь так хочется продлить удовольствие… И тут нам опять поможет компьютер!

Начинаем проект **С первого взгляда!**, сохраняем его в папке, настраиваем окно программы и объявляем константы:

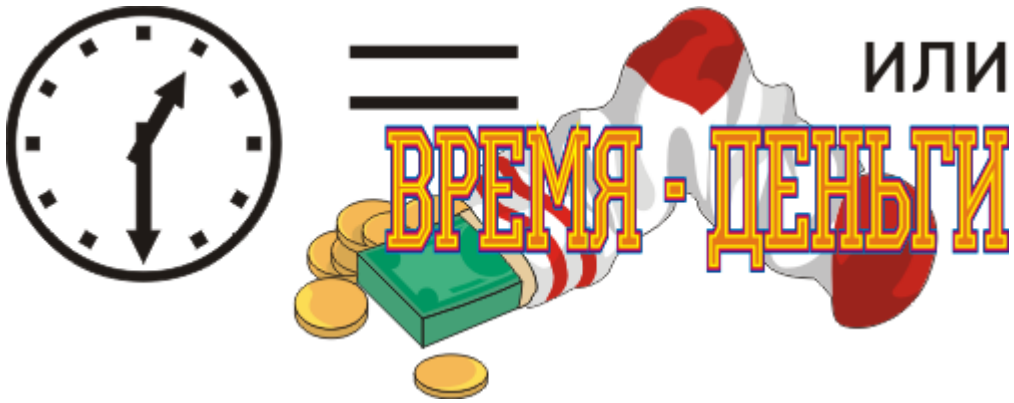
```
GraphicsWindow.Title=" С первого взгляда!"

'const
BackgroundColor="#FFFFF0" 'Ivory
N_IMAGE=6

. . .
```

Нам потребуется *таймер* для отсчёта заданного времени

```
'Таймер:
Timer.Interval=1000
Timer.Pause()
Timer.Tick=OnTick
```

и *картинки* с монетами. Согласно константе *N_IMAGE*, их должно быть шесть штук. Хранятся они в папке с выполняемым файлом, поэтому путь к ним легко найти:

```
path= Program.Directory +"/"
```



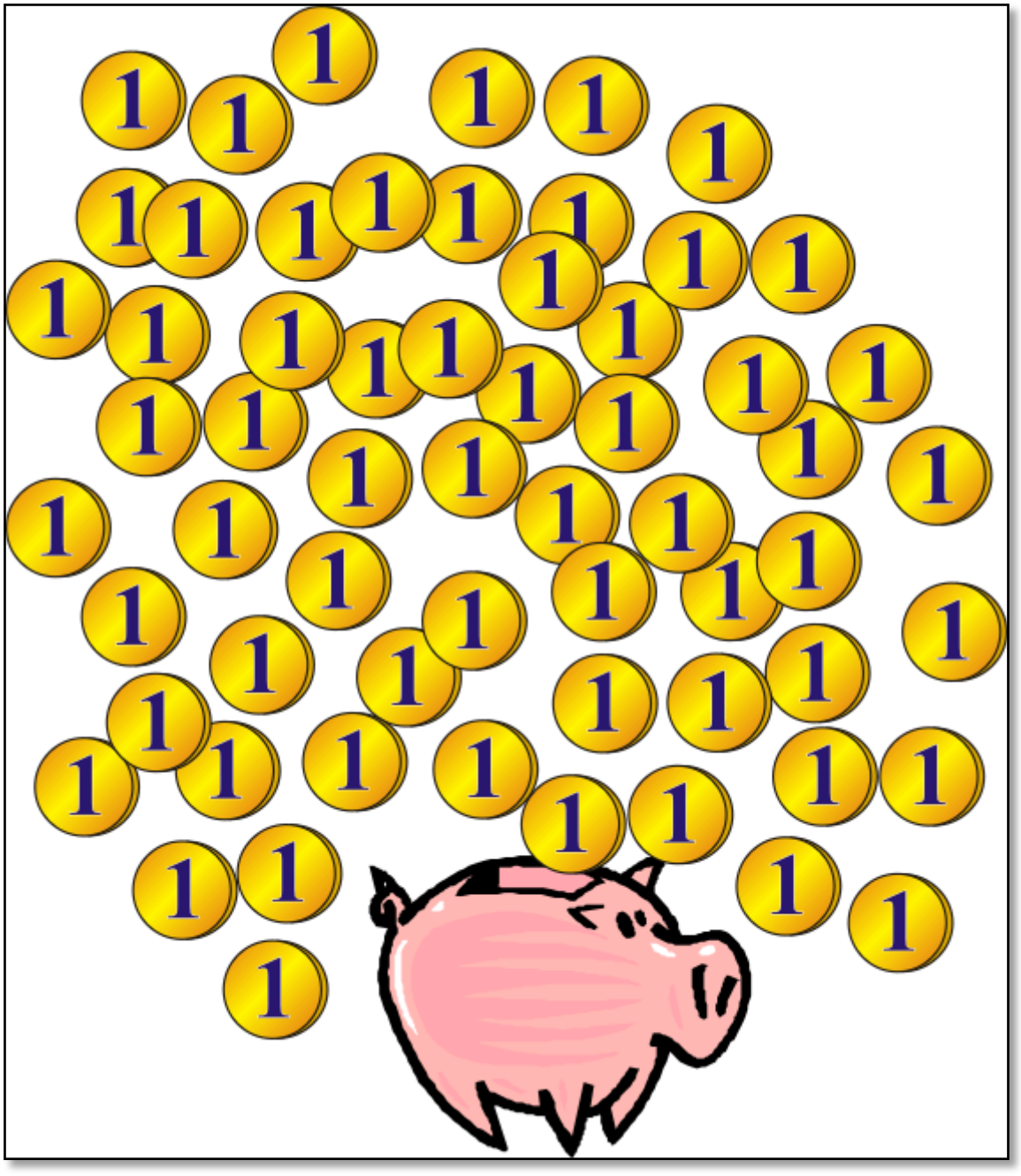


Рис. 36.1. Определите с первого взгляда, **сколько здесь монет**

Загружаем картинки в объект *ImageList* (*Список рисунков*) с помощью метода *LoadImage:*

```
img[1]=ImageList.LoadImage(path+"1.png")
img[2]=ImageList.LoadImage(path+"2.png")
img[3]=ImageList.LoadImage(path+"3.png")
img[4]=ImageList.LoadImage(path+"4.png")
img[5]=ImageList.LoadImage(path+"5.png")
img[6]=ImageList.LoadImage(path+"6.png")
```

Всё – картинки у нас в кармане! Теперь мы можем рисовать их на канве - где угодно и сколько угодно.

Элементы управления - точно такие же, как в программе *Психологическая считалка 2*, как и б**о**льшая часть этой программы. Вывод сделайте сами!

Процедура *OnClick* изменилась незначительно:

```
Sub OnClick
  btn=Controls.LastClickedButton
  If btn="Button1" then 'начать тестирование
    Controls.HideControl(btnStart)
    Controls.HideControl(txtOtvet)
    createTest()
    time=0
    Timer.Resume()
    Controls.SetTextBoxText(txtTime,"Время: 0")
    Controls.ShowControl(txtTime)
  Else    'проверить ответ
    Controls.HideControl("Button2")
    Controls.HideControl(txtTime)
    Controls.ShowControl("Button1")
    If nImage= Controls.GetTextBoxText(txtOtvet) then
      s=" ПРАВИЛЬНО - " + nImage + "!"
      GraphicsWindow.BrushColor= "Yellow"
    Else
      s="НЕПРАВИЛЬНО - " + nImage + "!"
      GraphicsWindow.BrushColor= "Red"
    endif
    GraphicsWindow.FontSize= 48
```

```
    GraphicsWindow.DrawText(90,height/2-24, s)
  EndIf
EndSub
```

Обратите внимание на то, как элементы управления появляются на экране и удаляются с него!

Тест формируется из загруженных *картинок*, а не из простых геометрических фигур, поэтому для программы можно использовать любые изображения!

```
Sub createTest
  GraphicsWindow.BrushColor=BackgroundColor
  GraphicsWindow.FillRectangle(0,40,width,height-40)
  'число картинок на экране:
  nImage=Math.GetRandomNumber(30)+ 20
  'размеры картинок:
  w= ImageList.GetWidthOfImage(img[1])
  h= ImageList.GetHeightOfImage(img[1])
  'отладка:
  'GraphicsWindow.Title= nImage
  for i= 1 to nImage
    'выбираем случайную картинку:
    nImg=Math.GetRandomNumber(N_IMAGE)
    'ыбираем случайные координаты  картинки  -->
    n=0
    nextXY:
    'задаем случайные координаты  картинки:
    x = Math.GetRandomNumber(width-w)
    y = Math.GetRandomNumber(height-h-40) + 40

    'центр картинки:
    if (GraphicsWindow.GetPixel(x+w/2,y+h/2) <>
BackgroundColor) Then
      Goto nextattempt
    EndIf
    'углы:
    If (GraphicsWindow.GetPixel(x,y) <> BackgroundColor)  Then
      Goto nextattempt
    EndIf
    If (GraphicsWindow.GetPixel(x+w,y) <> BackgroundColor)
Then
      Goto nextattempt
```

```
    EndIf
    If (GraphicsWindow.GetPixel(x,y+h) <> BackgroundColor)
Then
      Goto nextattempt
    EndIf
    If (GraphicsWindow.GetPixel(x+w,y+h) <> BackgroundColor)
Then
      Goto nextattempt
    EndIf
    Goto place
    nextattempt:
    If (n< 100) Then
      n=n+1
      Goto nextXY
    EndIf

    place:
    'рисуем картинку:
    GraphicsWindow.DrawResizedImage(img[nImg],x,y,w,h)
  EndFor
EndSub
```

Здесь важно позаботиться о том, чтобы картинки не закрывали друг друга. Для этого мы 100 раз пытаемся вывести картинку на свободное место канвы и только потом бросаем монетку наудачу. При небольшом числе монет этот способ создания теста действует очень хорошо.

Время тестирования ограничим *десятью* секундами:

```
'Отсчитываем время тестирования:
Sub OnTick
  time= time+ Timer.Interval/1000
  Controls.SetTextBoxText(txtTime,"Время: " + time)
  If (time>= 10) Then
    GraphicsWindow.BrushColor=BackgroundColor
    GraphicsWindow.FillRectangle(0,40,width,height-40)
    Timer.Pause()
    Controls.ShowControl(btnProv)
    Controls.ShowControl(txtOtvet)
  EndIf
EndSub
```

Когда время истечёт, все монеты исчезнут с экрана, а в окне приложения появится *кнопка* и *текстовое поле* (Рис. 36.2). Тестируемый должен ввести число – соответственно тому, сколько монет он успел насчитать, и нажать кнопку *Проверить*.

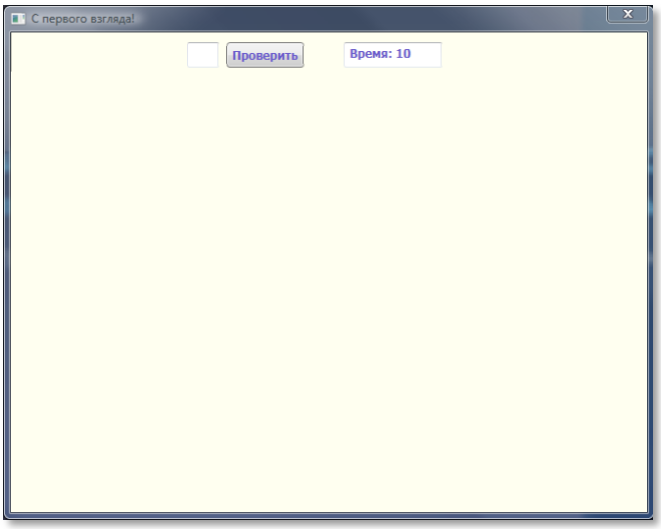


Рис. 36.2. Время вышло!

Как говорится, вердикт не заставит себя долго ждать (Рис. 36.3).

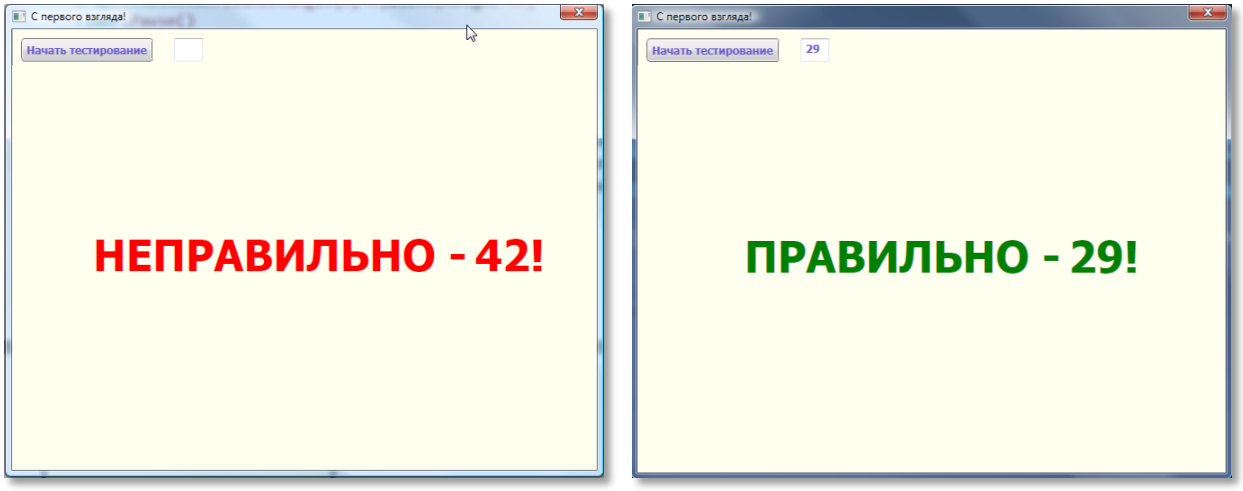


Рис. 36.3. Оттестировались по-разному!

Ну вот, к тесту мы подготовились. Запускаем программу, нажимаем кнопку *Начать тест* и стараемся уложиться в десять секунд (Рис. 36.4).

Если вам это удаётся, вы смело можете идти в банкиры и считать чужие деньги!

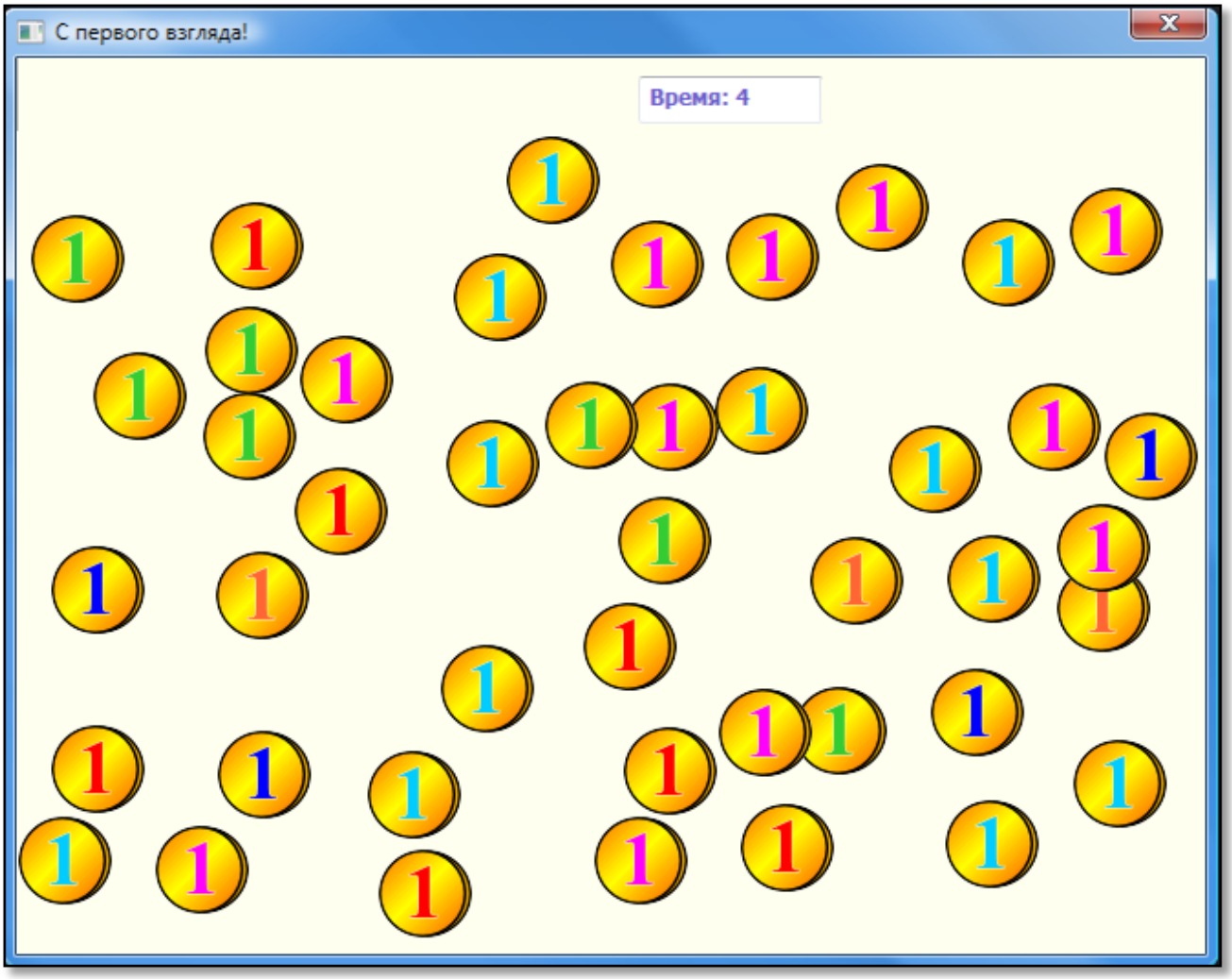


Рис. 36.4. Осталось 6 секунд… Ма-ма!

Исходный код программы находится в папке **С первого взгляда!**

Урок 37. Тараканьи бега по методу Монте-Карло

Вдруг из подворотни
Страшный великан,
Рыжий и усатый
Та-ра-кан!
Таракан, Таракан, Тараканище!

Корней Чуковский, *Тараканище*

Тараканьи бега – любимое развлечение *аристократов и дегенератов*, как выразился бы Лёлик из комедии *Бриллиантовая рука,* - проводятся так.

Берут в нужном количестве больших - длиной до 10 сантиметров - тараканов с острова Мадагаскар (Рис. 37.1). Конечно, тараканам всё равно куда бежать, поэтому их помещают в направляющие желобки, избавляющие их от тяжкого бремени выбора пути. Длина забега 1,5 – 2 метра. Зрители и прочие ротозеи делают ставки на полюбившегося им «спортсмена», после чего все тараканы весело бегут к финишу, подбадриваемые дружеским свистом публики.

Рис. 37.1. И вправду тараканище!

Если вам не довелось воочию увидеть это жуткое зрелище, то вы можете прочитать его художественное описание в пьесе *Бег* Михаила Булгакова (Рис. 37.2) и в романе Алексея Толстого *Похождения Невзорова, или Ибикус*.

Рис. 37.2. Призовые гонки тараканов в фильме *Бег* (1970)

Более изощрённый вариант тараканьих бегов показан в фильме *Китайский сервиз*, где тараканы по условному сигналу подносили шулеру карты, приклеенные к спине (Рис. 37.3).

Рис. 37.3. Пляшущие тараканы из фильма *Китайский сервиз*

Наверное, вам встречались и компьютерные игры, симулирующие тараканьи бега (или лошадиные, крысиные, собачьи и поросячьи). Мы сейчас тоже напишем программу, но не игровую, а вполне серьёзную, реализующую простую *компьютерную модель* вообще любых подобных состязаний.

Чтобы не отвлекаться на анатомические подробности разных видов «спортсменов», будем считать их *эллипсами*. Эта фигура вполне адекватно заменяет тушки бегунов. Их количество можно

принять равным шести, но при желании это число можно изменить в любую сторону.

Перед стартом все участники выстраиваются в одну линию, после чего следует сигнал к началу гонок. Тараканы бегут вперёд, пока первый из них не пересечёт финишную черту. Он объявляется победителем, а все игроки, сделавшие на него ставки, получают призовые деньги.

Если бы в забегах всегда побеждал один и тот же таракан, то состязание потеряло бы всякий смысл, поскольку все зрители ставили бы на него - без всякой надежды получить выигрыш больше той суммы, что они поставили. Значит, в гонках тараканы должны бежать не с одинаковой скоростью, более того, они должны совершать рывки или отставать. На результаты гонок могут влиять и другие факторы, например, физическое состояние участников, питание и настроение.

Поскольку учесть все эти факторы невозможно, то в нашей модели мы будем считать, что скорость бега каждого таракана изменяется по дистанции *случайным* образом. Кроме того, все остальные факторы мы объединим в одну группу и обозначим их как текущее *здоровье* таракана.

Таким образом, наша компьютерная модель будет описывать случайные процессы. Неудивительно, что для этого используют *метод Монте-Карло*. Как вы знаете, в этом городе находится самый известный в мире игорный дом, а рулетку вполне можно считать хорошим генератором случайных чисел. Действительно, шарик в рулетке случайно «выбирает» одно из 37 чисел (от 1 до 36 и 0 - зеро) на колесе.

Важно, чтобы числа выпадали именно *случайно*, без всякой закономерности, ибо всякая закономерность может быть подмечена внимательным игроком. Такой случай описал Джек Лондон в рассказе *Смок и Малыш*, где героям удалось выиграть неплохую сумму денег на рулетке, которая стояла рядом с печкой и потому со временем рассохлась. Колесо крутилось неравномерно, и одни номера выпадали чаще других.

На самом деле эта история - не вымысел автора. Английский инженер Джозеф Джаггерс в 1973 году с помощниками изучил поведение рулеток одного из казино в Монте-Карло и определил, что на одном из них числа выпадают неравномерно. За четыре дня он выиграл несколько сотен тысяч долларов, после чего руководство казино догадалось проверить колесо этой рулетки. С тех пор «система Джаггерса» не действует, так как во всех казино строго следят за исправностью рулеток и время от времени опять же случайным образом меняют местами колёса.

В романе Фёдора Михайловича Достоевского *Игрок* описаны все перипетии игры на рулетке.

Метод Монте-Карло разработали в 1948 году американские математики Джон Нейман и Станислав Улам. Кстати говоря, и раньше некоторые задачи решали с помощью случайных выборок, но до появления электронно-вычислительных машин применение этого метода было очень трудоёмким.

Теперь же в любом языке программирования имеется процедура, генерирующая случайные числа (точнее, *псевдослучайные*, поскольку они вычисляются по формуле). Есть такая процедура и в *СБ* – это метод *GetRandomNumber* класса *Math*.

Теперь мы знаем достаточно для того, чтобы приступить к написанию программы.

Как мы и договорились, в забегах будут принимать участие 6 тараканов, которых мы раскрасим в разные ***цвета***:

```
'ПРОГРАММА, МОДЕЛИРУЮЩАЯ ТАРАКАНЬИ БЕГА

GraphicsWindow.Title="Тараканьи бега"

'const
BackgroundColor="#2F4F4F"
N_TARAKAN=6   'число тараканов
'цвета тараканов:
color[1]="Red"
```

```
color[2]="Yellow"
color[3]="Blue"
color[4]="Green"
color[5]="SandyBrown"
color[6]="LavenderBlush"
'перенос строки:
perevodStroki=Text.GetCharacter(13) + Text.GetCharacter(10)
```

Также нам понадобятся *переменные* для хранения координат тараканов и их здоровья на момент старта:

```
'var
'координаты тараканов:
x[1]=0
y[1]=0
zdor[1]=0  'здоровье таракана
startTime=0'время старта
'флажки начала и окончания забега:
flgStart="False"
flgFinish = "False"
'координаты стартовой позиции:
xStart= 10
yStart=40
'расстояние между дорожками по вертикали:
dy= 32
'размеры тараканов:
wTar= 32
hTar= 16
```

Размеры *окна* вы можете сделать и больше, чтобы продлить удовольствие:

```
GraphicsWindow.Hide()
GraphicsWindow.Width= 640
GraphicsWindow.Height=480
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"
GraphicsWindow.BackgroundColor= BackgroundColor

height=GraphicsWindow.Height
```

```
width= GraphicsWindow.Width
```

Дополним интерфейс программы *элементами управления*. Каждые 10 миллисекунд (0,01 с) будет срабатывать *таймер*:

```
'ТАЙМЕР:
Timer.Interval= 10
Timer.Pause()
Timer.Tick=OnTick
```

Время, прошедшее со старта, мы будем выводить в *текстовое поле txtTime* в процедуре-обработчике таймера *OnTick*:

```
'Отсчитываем время гонок:
Sub OnTick
  t= Clock.ElapsedMilliseconds - startTime
  s= "Время: " + t/1000
  Controls.SetTextBoxText(txtTime,s)
EndSub
```

Обратите внимание, что для точного подсчёта времени мы используем свойство *ElapsedMilliseconds* класса *Clock*.

Тараканы помчатся вперёд после нажатия на *кнопку Начать забег!*:

```
'КНОПКА

'Начать забег:
btnStart=Controls.AddButton("Начать забег!", 10, height-32)
Controls.ButtonClicked=OnClick
```

Для вывода информации мы воспользуемся услугами двух *текстовых полей*. Первое будет показывать время, второе – *многострочное* – результаты забега:

```
'ТЕКСТОВЫЕ ПОЛЯ

'Время забега:
txtTime=Controls.AddTextBox(340, 10)
Controls.SetSize(txtTime,100,26)
Controls.HideControl(txtTime)
```

```
'Победитель:
txtWin=Controls.AddMultiLineTextBox(10, 300)
Controls.SetSize(txtWin,320,100)
Controls.HideControl(txtWin)
```

Перед первым стартом мы сбрасываем флажки и расставляем тараканов на стартовой позиции:

```
flgStart="False"
flgFinish = "False"
prepare()
```

В процедуре *prepare* мы создаём новых тараканов заданного цвета и размера. В нашей модели роль тараканов будут исполнять эллипсы. Почему эллипсы? – Во-первых, эллипсы куда приятнее для глаз, чем тараканы. Во-вторых, эллипсы более универсальны: под ними можно понимать и тараканов, и крыс, и жеребцов, что для нашей компьютерной модели тоже является плюсом. Ну и наконец, вы очень просто можете заменить эллипсы любыми картинками (хотя бы своих одноклассников), если загрузите их из файла, и метод *AddEllipse* замените методом *AddImage.*

Здесь же ветврач оценивает боеготовность каждого таракана и делает соответствующие записи в массиве *zdor*.

Для большего правдоподобия проводим *две черты – стартовую* и *финишную*, чтобы ни у одного таракана даже не возникла мысль о фальстарте!

```
'Готовим тараканов к старту
Sub prepare
  For i= 1 to N_TARAKAN
    'цвет таракана:
    clr= color[i]
    GraphicsWindow.BrushColor= clr
    GraphicsWindow.PenColor=clr
    'скрываем старых тараканов:
    Shapes.HideShape(tar[i])
    'создаем новых тараканов:
    tar[i]=Shapes.AddEllipse(wTar, hTar)
    'и расставляем их у стартовой черты:
```

```
    x[i]= xStart
    y[i]= yStart + dy * (i-1)
    Shapes.Move(tar[i], x[i], y[i])
    'здоровье тараканов:
    zdor[i]= Math.GetRandomNumber(30)/100 + 1.2
  endFor
  'чертим линию старта:
  GraphicsWindow.PenColor="Red"
  GraphicsWindow.DrawLine(xStart+wTar,yStart, xStart+wTar,
y[N_TARAKAN]+ hTar)

  'чертим линию финиша:
  GraphicsWindow.PenColor="Green"
  xFinish=width-wTar
  GraphicsWindow.DrawLine(xFinish, yStart, xFinish,
y[N_TARAKAN]+ hTar)
  'прячем инофрмационное окно:
  Controls.HideControl(txtWin)
endSub
```

Тараканы на старте – можно начинать *игровой цикл*:

```
'игровой цикл:
While (flgFinish = "False")
  'нажата кнопка "Начать забег!":
  If flgStart="True" then
    'вычисляем новые координаты тараканов:
    calcNewCoords()
    'и перемещаем их туда:
    For i= 1 to N_TARAKAN
      Shapes.Move(tar[i], x[i], y[i])
    endFor
    'устанавливаем скорость работы программы:
    Program.Delay(20)
    'проверяем, не финишировал ли таракан:
    testFinish()
  endIf
endWhile
```

Он будет продолжаться до тех пор, пока один из тараканов не достигнет финиша – тогда флаг *flgFinish* будет установлен в *True*.

Однако тараканы будут послушно стоять на старте, если флаг *flgStart* равен *False*. Как вы помните, в начале программы мы задали ему именно это значение. А вот когда пользователь или другой естествоиспытатель нажмёт кнопку *Начать забег!*, ситуация резко изменится:

```
'Начинаем забег
Sub OnClick
  Controls.HideControl(btnStart)
  Controls.ShowControl(txtTime)
  'обнуляем время:
  time=0
  Controls.SetTextBoxText(txtTime,"Âðåìÿ: " + 0)
  'запускаем таймер:
  Timer.Resume()
  'засекаем время старта по системным часам:
  startTime=Clock.ElapsedMilliseconds
  'стреляем из стартового пистолета:
  flgStart="True"
EndSub
```

В процедуре-обработчике *OnClick* мы готовим к бою секундомер и понуждаем тараканов к бегу громким звуком.

Условие *flgStart="True"* в игровом цикле выполнено, и тараканы помчались вперёд. В процедуре *calcNewCoords* мы определяем текущее положение каждого таракана.

Поскольку тараканы обязаны бежать неравномерно, то каждый раз мы *случайно* задаём каждому таракану то расстояние, которое он должен пробежать в данный момент. Обратите внимание на то, что мы учитываем и физическое состояние каждого таракана. Недомогающие особи будут медленнее шевелить ногами:

```
'Вычисляем новые координаты тараканов
Sub calcNewCoords
  For i= 1 to N_TARAKAN
    'случайное перемещение:
    dx= Math.GetRandomNumber(43)/10 + 0.9
    'новое положение таракана на дистанции:
    x[i] = x[i] + dx*zdor[i]
```

```
  endFor
endSub
```

Переместив всех тараканов в их новое положение, мы должны проверить, не пересёк ли хотя бы один из них финишную черту, иначе один за другим они скроются за пределами окна приложения, так и не выявив победителя. В том, что рано или поздно тараканы доползут до финиша, сомневаться не приходится, поскольку они бегут только вперёд.

Нам осталось рассмотреть последнюю процедуру *testFinish*, в которой мы и определяем, закончился ли забег, и если закончился, то какой таракан пришел первым. Здесь важно учесть, что финишировать могут и *несколько* тараканов. Тогда придётся проводить «фотофиниш»: победителем будет признан тот таракан, который убежал дальше за линию финиша на момент проверки:

```
'Проверям финиш
Sub testFinish
  xEnd=0
  n=0
  For i= 1 to N_TARAKAN
    if (x[i] >= xFinish-wTar) Then
      flgFinish="True"
      Timer.Pause()
      If (x[i] > xEnd) then
        'определяем номер победителя:
        n= i
        xEnd= x[i]
      EndIf
    endif
  endFor
  If (flgFinish="True") then
    s= "Победил таракан # " + n + "!"+ perevodStroki
    t= Clock.ElapsedMilliseconds - startTime
    Controls.SetTextBoxText(txtTime," Время:" + t/1000)
    s=s+ " Его время: " + t/1000 + " сек."
    flgStart="False"
    Controls.SetTextBoxText(txtWin, s)
    Controls.ShowControl(txtWin)
    Controls.ShowControl(btnStart)
    'ждем нового старта:
```

```
    While (flgStart="False")
    EndWhile

    'начинаем новый забег:
    flgFinish ="False"
    Controls.SetTextBoxText(txtTime," Время " + 0)
    'готовимся к новой игре:
    prepare()
  endIf
endSub
```

После финиша следует объявление чемпиона, оглашение победного времени (Рис. 37.4) и народные гуляния. Затем все болельщики, пьяные от счастья и шампанского, вновь запускают свежих тараканов по скользкому пути азарта.

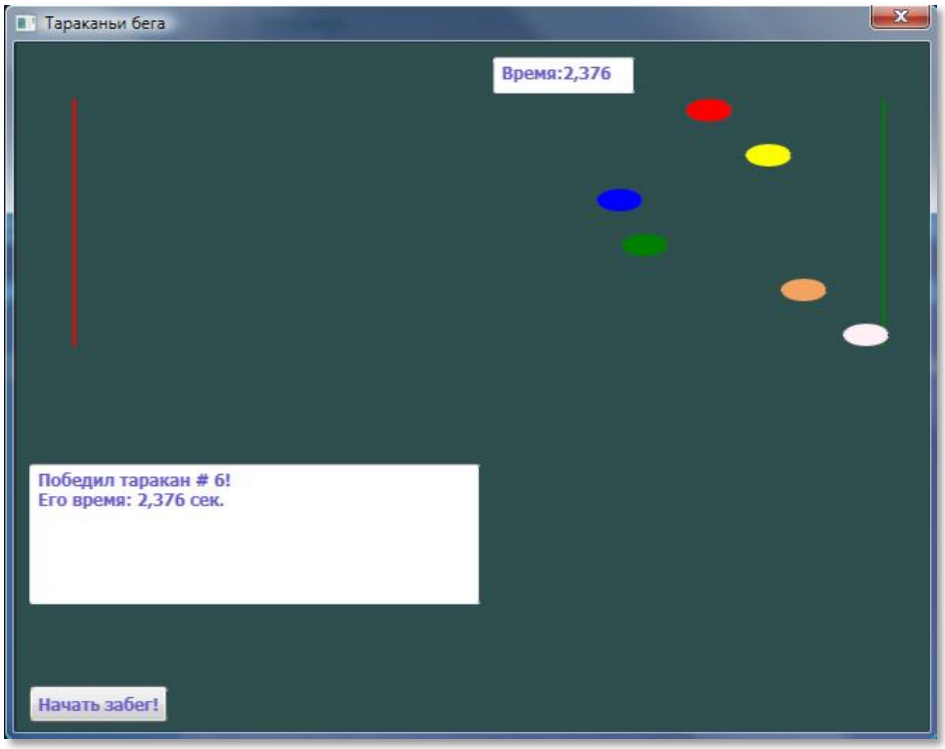


Рис. 37.4. Делайте ставки, господа!

Кстати говоря, наша компьютерная модель, несмотря на свою простоту, вполне адекватно отражает процесс тараканьей беготни, за которой наблюдать весьма любопытно. А если добавить к игре ещё и возможность делать ставки, то и на ипподром ходить не надо!

Исходный код программы находится в папке **Тараканьи бега.**

Электронная гадалка

Прошедший в 2010 году Чемпионат мира по футболу запомнился многим болельщикам и просто зрителям не только отвратительными африканскими дуделками (вувузелами), но и блестящими предсказаниями результатов матчей, которые делал немецкий *осьминог Пауль* (Рис. 37.5).

Рис. 37.5. Карикатурная обложка журнала *Тайм*, который признал Пауля лучшим осьминогом года

Он угадал результаты всех семи игр сборной Германии, а также финала. Итого - 8 игр. Для гадания ему предлагали два прозрач-

ных ящичка с флагами стран-участниц очередного матча, из которых он выбирал один – открывал его щупальцами.

Поскольку встречи в группах могли закончиться и вничью, то само гадание было не совсем корректным. Впрочем, все гадания прошли успешно, потому что сборная Германии ни разу вничью не сыграла.

В Испании, чья футбольная команда завоевала Кубок мира, сделали бронзовую статую осьминога и подарили её океанариуму немецкого города Оберхаузена. Также он стал почётным гражданином испанского города Карбальино, а американец Пэрри Грипп посвятил ему песню *Осьминог Пауль*. В самой Германии, правда, из него хотели сварить суп, но, в конце концов, его простили, поскольку он выступил лучше сборной Германии, и отправили на заслуженный отдых.

Вот так осьминог Пауль *благополучно* завершил свою карьеру предсказателя, что с предсказателями бывает нечасто.

Однако давайте подсчитаем, так ли уж велики заслуги этого головастого головоногого моллюска перед мировым сообществом. Если в каждом матче выигрывает одна из двух команд, то вероятность правильного предсказания результата одной игры равна ½, двух – $(½)^2$ = ¼, а всех восьми - $(½)^8$ = 1/256. То есть из 256 осьминогов один почти наверняка угадал бы все результаты. Так что подвиг Пауля не столь уж значителен, особенно если учесть, что некоторые люди выигрывают и в Спортлото (вероятность угадывания 5 чисел из 36 равна 1/45 239 040, а 6 чисел из 49 - 1/10 068 347 520).

Поскольку результаты футбольных матчей в большой мере случайны, а учесть все факторы невозможно, то будем вслед за Паулем полагать, что у каждой команды *равные* шансы на победу.

И нам осталось написать простенькую программу **Осьминог Пауль**, которая моделирует простейшего *электронного оракула*.

Для ввода названий команд нам потребуются два *текстовых поля* и столько же *переменных*.

```
'ЭЛЕКТРОННЫЙ ОРАКУЛ

GraphicsWindow.Title=" Осьминог Пауль "

'const
'перенос строки:
perevodStroki=Text.GetCharacter(13) + Text.GetCharacter(10)

'var
'названия команд:
team1=""
team2=""

GraphicsWindow.Hide()
GraphicsWindow.Width= 440
GraphicsWindow.Height=270
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"

height=GraphicsWindow.Height
width= GraphicsWindow.Width

background = ImageList.LoadImage(Program.Directory +
"/paul.jpg")
GraphicsWindow.DrawImage(background, 0, 0)

'шрифт:
GraphicsWindow.BrushColor="Red"
GraphicsWindow.BrushColor="Green"
GraphicsWindow.FontBold="True"
GraphicsWindow.FontSize=16
```

Отдавая должное нашему герою Паулю, мы поместим его фото на задний план окна (Рис. 37.6).

Перейдем к *элементам управления*. Запишите рецепт: возьмите одну *кнопку* и три *текстовых поля* и разместите их в окне по вкусу. Например, так, как на Рис. 37.6:

```
' КНОПКА

' Предсказать результат:
btnStart=Controls.AddButton("ПРОГНОЗ!", 10, height-32)
Controls.ButtonClicked=OnClick

' ТЕКСТОВЫЕ ПОЛЯ

' Команды:
txtTeam1=Controls.AddTextBox(10, 10)
Controls.SetSize(txtTeam1,160,26)
GraphicsWindow.DrawText(180, 14, "<- Первая команда")

txtTeam2=Controls.AddTextBox(10, 48)
Controls.SetSize(txtTeam2,160,26)
GraphicsWindow.DrawText(180, 52, "<- Вторая команда")

GraphicsWindow.BrushColor="Red"

' Победитель:
txtWin=Controls.AddMultiLineTextBox(10, 180)
Controls.SetSize(txtWin,300,50)

GraphicsWindow.Show()
```

Нажимаем кнопку – и получаем предсказание на интересующую нас игру:

```
' Печатаем результат
Sub OnClick
  s1= Controls.GetTextBoxText(txtTeam1)
  s2= Controls.GetTextBoxText(txtTeam2)
  If (s1="") Or (s2="") Then
    s= "Введите названия команд!"
  ElseIf (team1=s1) And (team2=s2) Then
    s= "Повторно не гадаю!"
  ElseIf (s1=s2) Then
    s= "Одна и та же команда!"
  Else
```

```
    team1=s1
    team2=s2
    If (Math.GetRandomNumber(2)=1) Then
      s= "Победит " + team1
    Else
      s= "Победит " + team2
    EndIf
  EndIf
  GraphicsWindow.BrushColor="Blue"
  s= s + perevodStroki + "Давайте погадаю еще раз!"
  Controls.SetTextBoxText(txtWin, s)
EndSub
```

Бо́льшую часть процедуры занимают проверки и беседа с пользователем, но зато так наш предсказатель выглядит более мудрым. А само предсказание занимает пару строк и основано всё на том же методе *GetRandomNumber*, что и в предыдущем проекте.

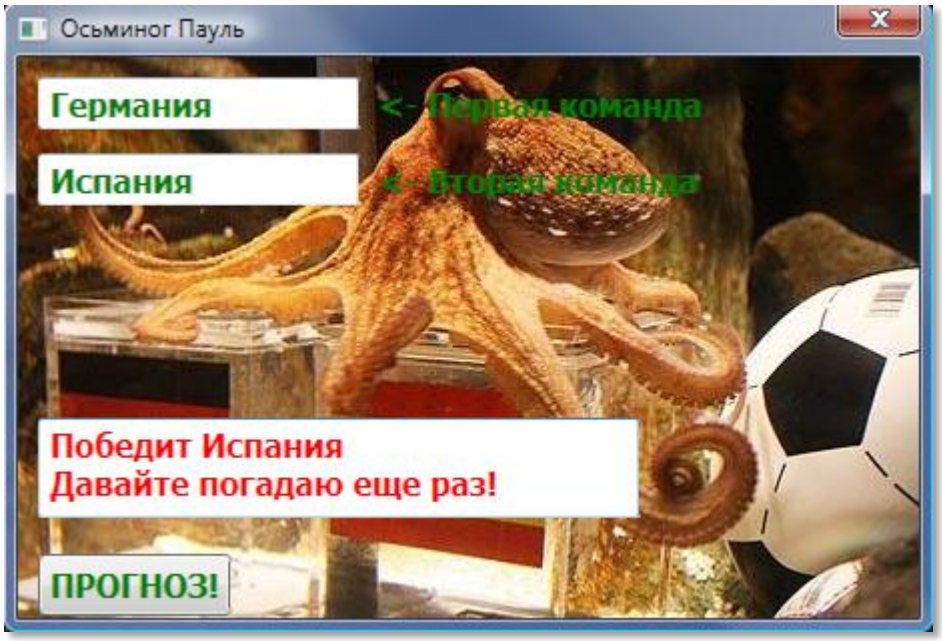


Рис. 37.6. И наш электронный Пауль – парень не промах!

Конечно, наша математическая модель слишком проста, но её можно улучшить, если вычислять вероятность исхода матча с учетом рейтинга ФИФА для сборных команд. Для гаданий на Чемпионат России, например, вполне уместно оценивать по-

ложение команд в турнирной таблице и статистику личных встреч. Но сколько бы мы факторов ни учитывали, а всё равно не угадаем!

Исходный код программы находится в папке **Осьминог Пауль**.

Число *пи*, или Метод научного тыка

А теперь давайте рассмотрим пример более серьёзного применения метода Монте-Карло.

Пусть нам нужно вычислить площадь *S* какой-либо криволинейной фигуры, которая задана графически (нарисована) или аналитически (формула).

Впишем её в квадрат (Рис. 37.7) со сторонами *L* сантиметров (или миллиметров, или метров) и начнём совершенно случайно «тыкать» пальцем так, чтобы все «тыки» приходились в квадрат (можно тыкать и мимо, но тогда неудачные тыки мы просто не учитываем).

Поскольку интересующая нас фигура целиком и полностью лежит *внутри* квадрата, то ей тоже достанется. И вероятность того, что тык попадёт именно в фигуру прямо пропорциональна её площади.

Если мы тыкнули *N* раз (**красные** и **жёлтые** точки на Рис. 37.7, справа), а в фигуру угодили *P* раз (**красные** точки там же), то мы можем вычислить площадь фигуры так. Площадь квадрата равна L^2, а нашей фигуры – *S*, значит:

$L^2 / S = N / P$, откуда находим S:

$$S = L^2 * P / N \qquad (1)$$

Нетрудно понять, что при малом количестве тыков точность вычислений по формуле (1) будет невелика, однако, как показывает практика, при больших N погрешность вычислений уменьшается.

Естественно, тысячи раз тыкать пальцем (а лучше карандашом) в чертёж, да ещё при этом надеяться на случайное распределение тыков по площади квадрата, было бы неразумно, поэтому мы поступим правильнее, если научим этому нехитрому занятию компьютер. Он всё сделает быстро и аккуратно.

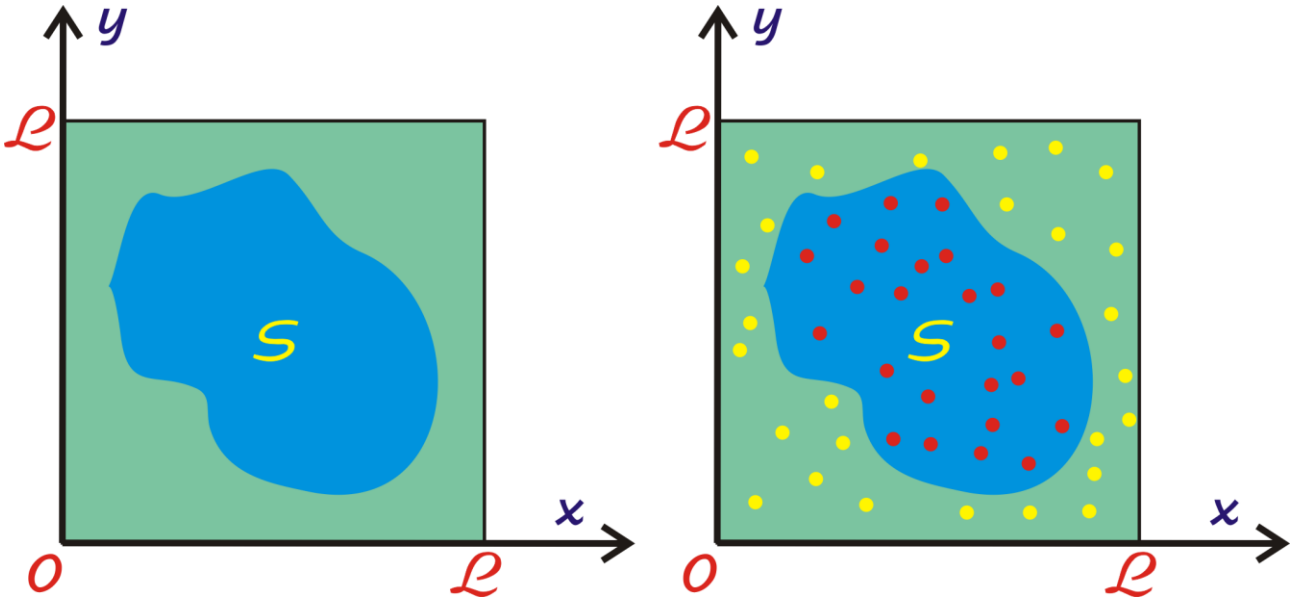


Рис. 37.7. Метод научного тыка в действии

Попробуем этим способом определить число *пи*. Мы знаем, что площадь любой плоской фигуры прямо пропорциональна квадрату её характерного размера. Для собственно квадрата это длина стороны, а для круга – *диаметр*. Коэффициент пропорциональности для квадрата равен единице, а для круга – числу *пи,* значение которого мы пока не знаем.

Нанесём по квадрату и вписанному в него кругу N случайных выстрелов, из которых P попадут в круг (Рис. 37.8).

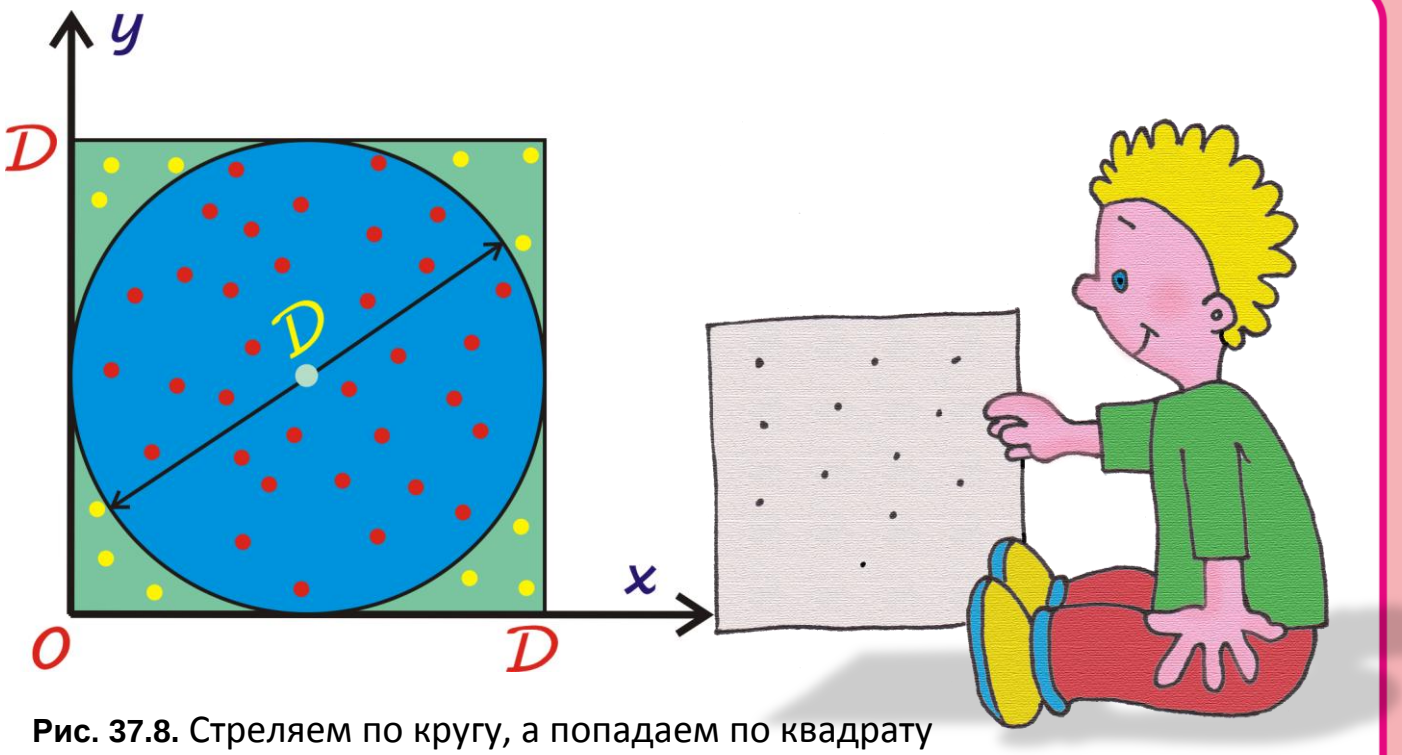


Рис. 37.8. Стреляем по кругу, а попадаем по квадрату

Отношение чисел *P* и *N* пропорционально площадям фигур:

$$\frac{P}{N}=\frac{\pi D^2}{4D^2}$$

Из этой формулы мы легко найдем *пи*:

$$\pi=\frac{4P}{N}$$

Новую программу **Метод Монте-Карло**, как обычно, мы начнём с объявления *констант* и *переменных*. После теоретической подготовки здесь трудностей у нас быть не должно:

```
'ВЫЧИСЛЕНИЕ ПИ МЕТОДОМ МОНТЕ-КАРЛО

'const
BackgroundColor="MidnightBlue" 'цвет фона
'смещение фигур от левого верхнего края окна:
dx=10
dy=50

'var
'общее число точек:
```

```
N=10000
'число точек, попавших в круг:
P=0
'диаметр круга:
D=400
D2= D*D
'радиус круга:
R= D/2
'центр круга:
xc=0
yc=0
```

Задаём разумные *размеры* окна и фигур, хотя в нашем случае размер имеет значение – чем больше, тем лучше!

```
GraphicsWindow.Title="Метод Монте-Карло"
GraphicsWindow.Hide()
GraphicsWindow.Width= 410
GraphicsWindow.Height=450
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"
GraphicsWindow.BackgroundColor= BackgroundColor

height=GraphicsWindow.Height
width= GraphicsWindow.Width
```

Из *элементов управления* нам понадобится *кнопка*, чтобы начать процесс. *Текстовое окно* для вывода текущего значения числа *пи* и *таймер*, который и обеспечит нас актуальной информацией о ходе эксперимента:

```
'КНОПКА
btnStart=Controls.AddButton("Начать!", 10, 10)
Controls.ButtonClicked=OnClick

'ТЕКСТОВОЕ ПОЛЕ
'Пи:
txtPi=Controls.AddTextBox(100, 10)
Controls.SetSize(txtPi,200, 24)
```

```
' ТАЙМЕР
Timer.Interval= 100
Timer.Pause()
Timer.Tick=OnTick

GraphicsWindow.Show()
```

Рисуем на канве окна героев нашего эксперимента – *квадрат* и вписанный в него *круг*:

```
' рисуем квадрат:
clr="Green"
GraphicsWindow.BrushColor = clr
GraphicsWindow.PenColor = "Black"
GraphicsWindow.FillRectangle(dx, dy, D, D)
' рисуем круг:
clr="Blue"
GraphicsWindow.BrushColor = clr
GraphicsWindow.FillEllipse(dx, dy, D, D)
' вычисляем координаты центра круга:
xc= D/2 + dx
yc= D/2 + dy
```

Для эксперимента всё готово. Без тени сомнения нажимаем кнопку *Начать!* – и *процесс пошёл*, как говаривал первый Президент СССР.

Правда, со словом *начать* у него были орфоэпические проблемы.

```
Sub OnClick
  ' запускаем таймер:
  Timer.Resume()
  ' ставим точки:
  P=0
  setPoints()
EndSub
```

Главные события сей научной драмы разворачиваются в процедуре *setPoints*:

```
'Ставим точки
Sub setPoints
```

```
  for i= 1 to N
    'выбираем случайные координаты  точки:
    x = Math.GetRandomNumber(D) + dx - 1
    y = Math.GetRandomNumber(D) + dy - 1
    xx= xc - x
    yy= yc - y
    'попали в круг?
    If (xx*xx + yy*yy <= R*R) Then 'попали
      p=p+1
      clr= "Red"
    Else 'не попали
      clr= "Yellow"
    EndIf

    'устанавливаем цвет точки:
    GraphicsWindow.BrushColor=clr
    'рисуем точку:
    GraphicsWindow.FillEllipse(x, y, 2,2)
  EndFor
EndSub
```

Чтобы проверить, попадает ли точка с координатами *(x, y)* внутрь круга или нет, достаточно по теореме Пифагора вычислить длину гипотенузы *r* прямоугольного треугольника (Рис. 37.9):

$$r = \sqrt{xx^2 + yy^2}$$

Поскольку *r >= 0*, то обе части равенства можно возвести в квадрат:

$$r^2 = xx^2 + yy^2$$

Теперь достаточно сравнить *r* с *R*. Если *r <= R*, то точка находится *внутри* круга, иначе – *снаружи*.

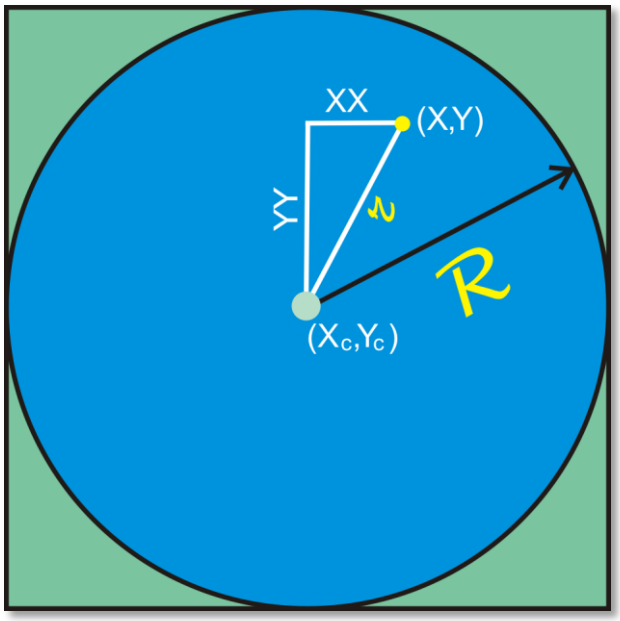


Рис. 37.9. Геометрия – это наука!

В *компьютерной* модели проверку можно проводить и иначе. Мы знаем, что круг окрашен в **синий** цвет, поэтому достаточно проверить цвет пикселя в точке с координатами *(x,y)*. Если он **синий**, то точка находится внутри круга, в противном случае – снаружи.

Однако, после того как в круге появятся точки **красного** цвета, нужно будет учитывать и их. Поскольку мы окрашиваем точки вне круга в **жёлтый** цвет, то проверка будет очень простой.

Вопреки теоретическим изысканиям, наша модель не уточняет значения *пи* с увеличением числа испытаний, что хорошо видно на Рис. 37.10. Зато генератор случайных чисел в *СБ* работает совсем неплохо, поскольку на правом рисунке почти все точки закрашены (при его идеальной работе были бы закрашены все 160 000 точек, а это не так).

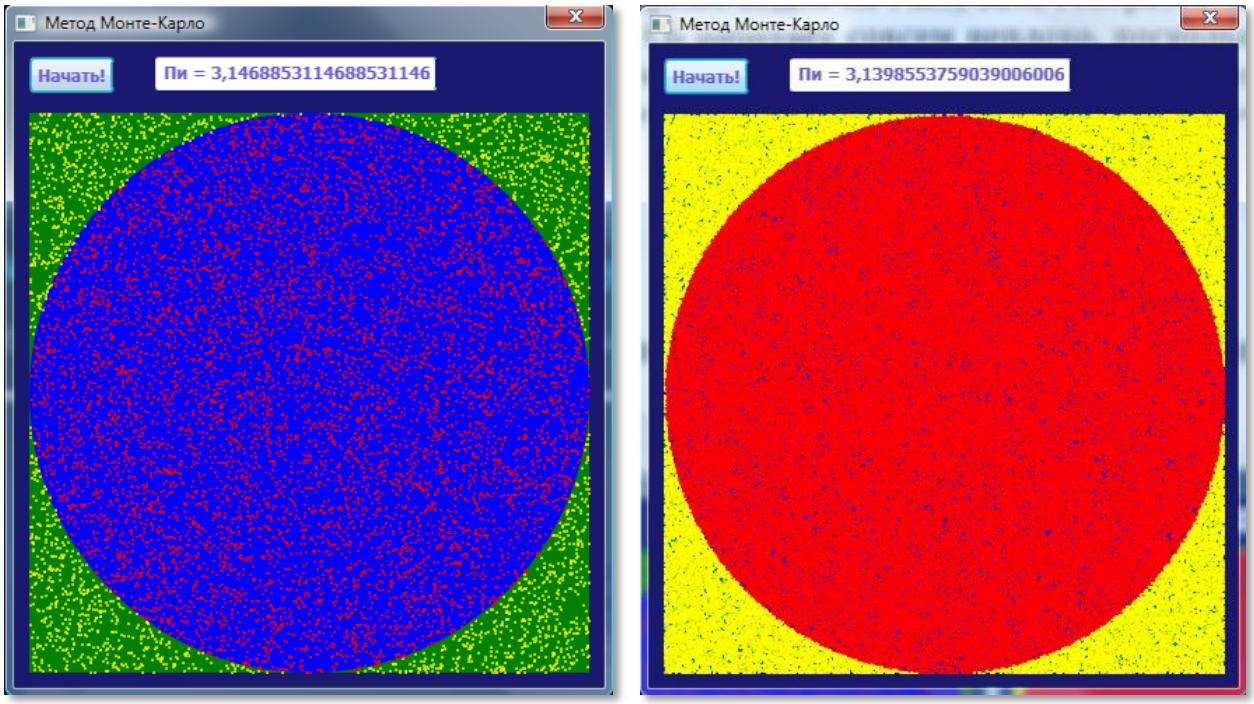


Рис. 37.10. Результаты эксперимента при N= 10 000 и N= 160 000

И хотя в *текстовом поле* мы видим множество цифр после запятой, точность вычислений ограничивается уже *вторым* знаком. Несколько улучшить ситуацию можно, если провести серию испытаний, а затем вычислить среднее значение. Принципиальный же источник ошибок здесь в том, что реальные фигуры состоят из *бесконечного* числа точек, а мы закрашиваем пиксели, которых совсем немного. Конечно, если отказаться от визуализации процесса и вычислять значение *пи* по формулам, то эта проблема будет решена. Другая проблема состоит в том, что все случайные числа, которые генерирует метод *GetRandomNumber*, - *целые*, а координаты точек внутри фигур скорее рациональные, чем целые. Можно генерировать последовательности цифр после запятой, чтобы получать рациональные случайные числа, но это уже детали реализации математической модели, которые не могут поколебать её основ.

Исходный код программы находится в папке **Метод Монте-Карло.**

1. Напишите программу для бросания монеты. Для этого достаточно слегка изменить проект *Осьминог Пауль*.

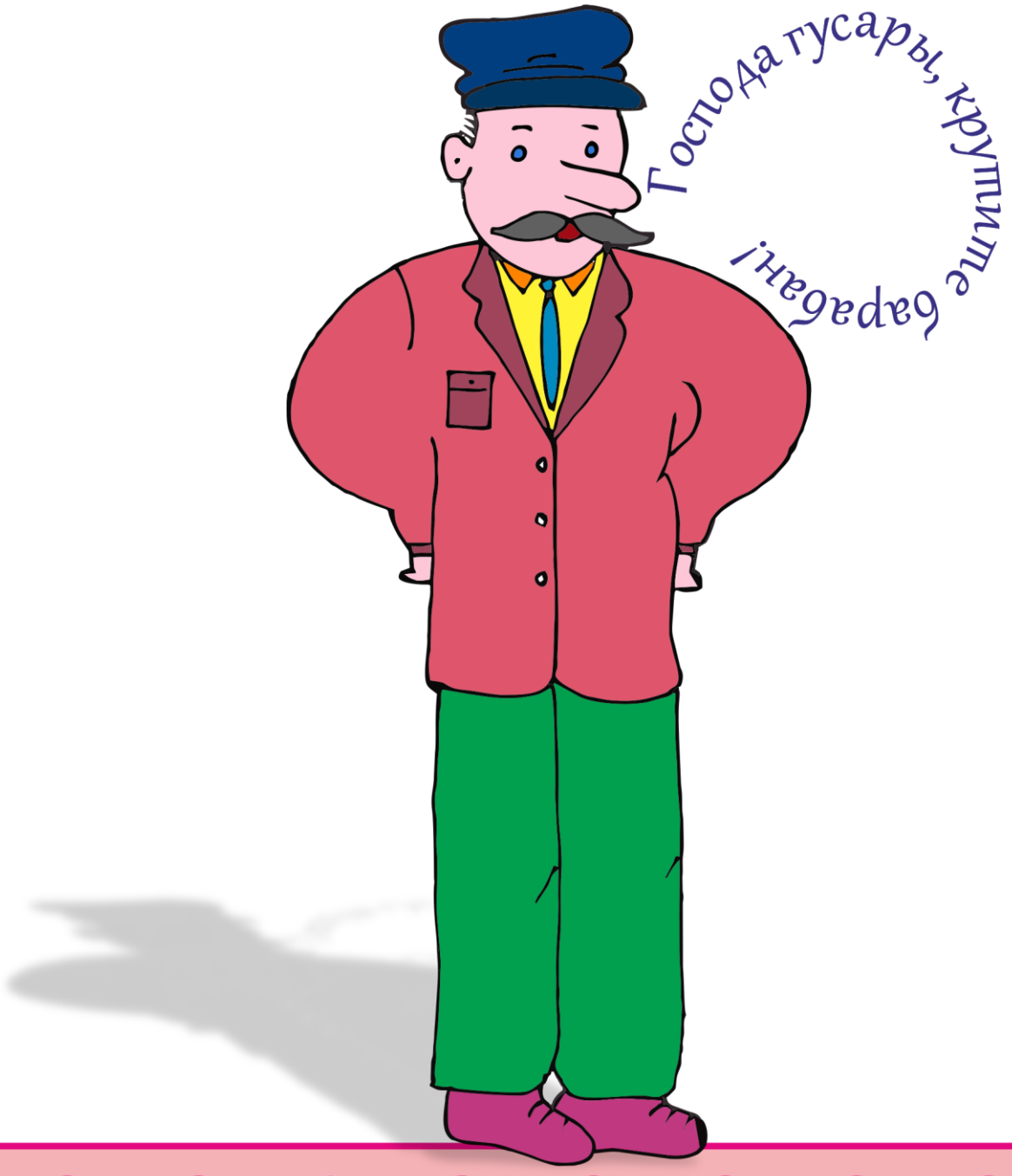
2. Немного труднее научить программу бросать кубик, вытаскивать карту из колоды или бочонок из мешка в игре лото.

3. Экстремальным вариантом рулетки можно считать *русскую рулетку (*иначе она называется *гусарской рулеткой)*. «Играют» в неё так. В барабан шестизарядного револьвера вставляется один патрон (из соображений гуманности и здравомыслия будем считать, что *холостой*). Затем барабан прокручивается несколько раз так, чтобы «игроки» не знали его положения относительно ствола пистолета. Мы, не обсуждая все разновидности этой игры, будем считать, что после каждого спуска крючка барабан прокручивается вновь. Таким образом, вероятность выстрела будет постоянной на протяжении всей игры и равна 1/6. Соответственно, вероятность благоприятного исхода после первого выстрела равна 1 - 1/6 = 5/6 = 0,833. После второго $(5/6)^2$ = 0,694, после шестого - $(5/6)^6$ = 0,335, и так далее, приближаясь к *нулю* (вот почему так важно учить теорию вероятностей!). Математическая модель этой игры ничем не отличается от моделирования бросания игрального кубика, у которого в точности шесть граней.

4. *Слот-машина*, больше известная под названием *однорукий бандит*, представляет собой три или пять барабанов, которые вращаются независимо друг от друга, наподобие колеса рулетки. На боковую поверхность барабанов нанесены различные символы, комбинация которых после остановки колёс и определяет сумму выигрыша или проигрыша игрока, сделавшего ставку (Рис. 37.11). Справедливое прозвище этих автоматов *однорукий бандит* восходит к тому времени, когда колёса раскручивались рычагом на боковой поверхности автомата. Этот рычаг и придавал ему такой запоминающийся облик.

Сейчас больше распространены видео-слоты, которые обходятся без механических барабанов, поэтому рычаг им не нужен. Поскольку в новых машинах комбинация символов задаётся генератором псевдослучайных чисел, то достаточно просто написать программу, имитирующую их бандитскую деятельность.

Рис. 37.11. Удачная комбинация!



Урок 38. Перебор с возвратами

Мы рассмотрим сначала простую задачу: *найти все варианты расстановки восьми ладей на шахматной доске так, чтобы они не били друг друга*. Когда речь идет о поиске *всех* вариантов, то обычно для решения задачи используют *перебор вариантов*, или *метод перебора с возвратами* (по-английски *backtracking*).

Обычная шахматная доска имеет 8 клеток в ширину и столько же в высоту, что довольно много для предварительных изысканий, поэтому мы начнём с самой маленькой доски – 1 x 1 клетку (Рис. 38.1). Ясно, что на одну клетку можно поставить *единственную* ладью, которая других ладей бить не сможет - из-за отсутствия соперников. Итак, на такую доску можно поставить одну ладью одним способом.

Рис. 38.1. Ладья на поле 1 x 1

Мы будем рассматривать только *квадратные* доски - на прямоугольных досках мы не найдём ничего нового.

Возьмём доску побольше – 2 x 2 клетки. Первую ладью можно поставить на первую горизонталь, в самую первую клетку (Рис. 38.2).

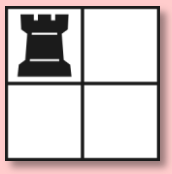


Рис. 38.2. Ладья на поле 2 x 2

Поскольку других ладей на поле нет, то никакие проверки не нужны. Из шахматных правил следует, что ладья бьёт все поля, находящиеся с ней на одной вертикали и горизонтали. Для убедительности закрасим битые поля (Рис. 38.3).

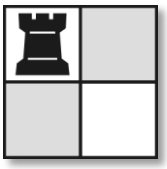


Рис. 38.3. Ладья бьёт две клетки

Теперь ясно видно, что на каждой горизонтали может стоять только *одна* ладья, потому что она держит под контролем *всю* горизонталь, в которой находится. Учтём это на будущее и попробуем поставить *вторую* ладью.

Первая горизонталь уже занята, поэтому поставим вторую ладью в первую клетку второй горизонтали. Проверка показывает, что на этой вертикали уже находится первая ладья, значит, такая позиция невозможна (Рис. 38.4).

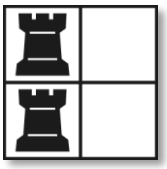


Рис. 38.4. Первая ладья бьет вторую

Передвигаем вторую ладью **вправо** (Рис. 38.5).

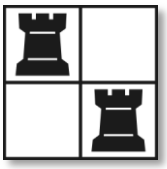


Рис. 38.5. Вторая ладья уходит из-под боя

Теперь всё нормально, обе ладьи заняли допустимые позиции, а мы нашли *первый* вариант расстановки. Но, возможно, есть и другие - ведь нам нужно найти *все* варианты. Мы не можем вторую ладью передвинуть еще раз вправо, потому что горизонталь закончилась. Нам остаётся убрать вторую ладью, вернуться назад к первой ладье и попробовать *её* передвинуть вправо. Справа от неё располагается пустая клетка, так что такая операция возможна (Рис. 38.6).

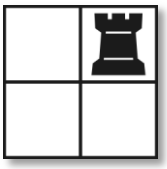


Рис. 38.6. Первая ладья идет вправо

Ищем место для второй ладьи от начала второй горизонтали. Подходит уже первая клетка (Рис. 38.7).

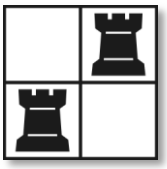


Рис. 38.7. Второе решение найдено!

Мы нашли *второй вариант* решения задачи. А есть ли ещё решения? – Для проверки снова передвигаем вторую ладью вправо (Рис. 38.8).

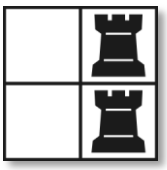


Рис. 38.8. Вторая ладья опять под боем!

В этой позиции вторая ладья оказывается под боем первой ладьи. Передвинуть её ещё вправо невозможно, поэтому снимаем вторую ладью с доски и пытаемся передвинуть первую ладью вправо. Это также невыполнимая операция. Убираем первую ладью с поля и убеждаемся, что на поле не осталось больше ни одной ладьи. Значит, мы нашли *все* расстановки двух ладей на поле 2 x 2 клетки. Всего таких расстановок *две*.

Если у вас есть желание и терпение, вы аналогично можете исследовать доску 3 x 3 клетки, но мы уже знаем *алгоритм поиска* всех решений, поэтому нам ничего не стоит перевести его на язык *СБ*, а компьютер найдёт нам решения всех ладейных задач.

Начнём новый проект **Ладьи** и сразу введём все *переменные* программы:

```
' ПРОГРАММА РАССТАНОВКИ ЛАДЕЙ

'var
cols=8          ' ширина поля в клетках (число вертикалей)
rows=8          ' высота поля в клетках (число горизонталей)
nLad=8          ' число ладей
n=0             'номер уже выставляемой ладьи
vert[0] = 0     ' = 1..rows - номер вертикали, на которой стоит
ладья
v=0             ' текущая вертикаль
resolution= "False"  ' флаг: если = True, то можно ставить
очередную ладью
nVar=0          ' число найденных вариантов расстановки
```

В первую очередь, нам понадобятся переменные *cols* и *rows* для хранения размеров шахматного поля и переменная *nLad.* В ней мы будем хранить число ладей, которые необходимо выставить на поле. В нашем примере, все они равны восьми, но вы легко можете изменить как размеры поля, так и число ладей, присвоив переменным нужные значения. Например, отладку программы следует проводить на поле 2 x 2, чтобы убедиться в правильности работы программы на таком поле. Затем можно постепенно довести размеры поля до 8 x 8. Так по ходу отладки мы подсчитаем варианты расстановки ладей на квадратных полях разных размеров.

Буквой *n* мы обозначим переменную для хранения номера выставляемой на поле ладьи. Во всех горизонталях должно стоять по одной ладье, при этом каждая из них занимает свою вертикаль. Номер вертикали, на которой стоит та или иная ладья, будем хранить в массиве *vert*. Например, первая ладья занимает вертикаль *vert[1]* в первой горизонтали (всегда считаем горизонтали сверху вниз!). Вторая - *vert[2]* во второй горизонтали, и так далее. Пока место для очередной ладьи не найдено, мы запоминаем номер вертикали *v*, на которую мы хотим её поставить.

Переменная *resolution* играет роль флага. Если он установлен (равен *True*), то очередную ладью *n* можно поставить на вертикаль *v*. Проверять допустимость выставления ладьи на вертикаль *v* мы будем в процедуре с недвусмысленным названием *test*.

И последняя переменная – *nVar* – послужит нам для подсчёта найденных вариантов.

Начинаем кодировать наш алгоритм.

В исходном положении ни одна ладья не выставлена на поле:

```
'ни одна ладья еще не выставлена:
n=0
```

Ставим *следующую* ладью. После нулевой это будет, конечно, *первая*. Если одну ладью мы уже поставили, то следующей будет вторая, и так далее. Поскольку ладей нужно выставить несколько, то обозначим меткой *nextLad* строку кода, в которой мы выставляем следующую ладью:

```
nextLad:
n=n+1
```

Если выставлены уже все ладьи, то найден очередной вариант расстановки, который нужно записать в файл или вывести на экран:

```
If (n > nLad) Then
  Goto writeVar
EndIf
```

Каждую новую ладью пытаемся выставить на первую вертикаль, затем на вторую, и так до тех пор, пока не выйдем за границу поля. Как вы помните, эта операция проводится многократно для каждой ладьи, поэтому нужно запомнить начало кода для движения ладьи *вправо*, чтобы в нужный момент перейти к нему:

```
v=0
nextVert:
```

```
v=v+1
If (v > cols) Then
  Goto back
EndIf
```

Если места для очередной ладьи не нашлось, значит, нужно вернуться *назад*, к предыдущей выставленной ладье. Эти строки кода обозначены меткой *back*.

```
test()
If resolution="False" then
  Goto nextVert
EndIf
vert[n]= v
Goto nextLad
```

Итак, очередную ладью *n* мы хотим поставить на вертикаль *v*. Имеем ли мы право сделать это, узнаём по результатам проверки в процедуре *test*. Если нет (процедура *test* сбросила флаг *resolution*), то возвращаемся на строку *nextVert*, чтобы передвинуть ладью вправо. Если же проверка прошла успешно и процедура *test* установила флаг *resolution*, то уверенно ставим ладью *n* на эту вертикаль, а само приятное для нас событие отмечаем в массиве *vert*. Далее мы возвращаемся на метку *nextLad*, чтобы выставить следующую ладью.

Найденное решение мы выводим в *текстовое поле* в самом простом виде – выписываем номера вертикалей для каждой ладьи:

```
writeVar:
nVar = nVar +1
TextWindow.ForegroundColor="Red"
TextWindow.WriteLine("Вариант # " + nVar)
TextWindow.ForegroundColor="Yellow"
For i= 1 To nLad
  TextWindow.Write(vert[i]+ " ")
EndFor
TextWindow.WriteLine("")
TextWindow.WriteLine("")
```

Для поля 4 x 4 клетки «протокол» выглядит так (Рис. 38.9).

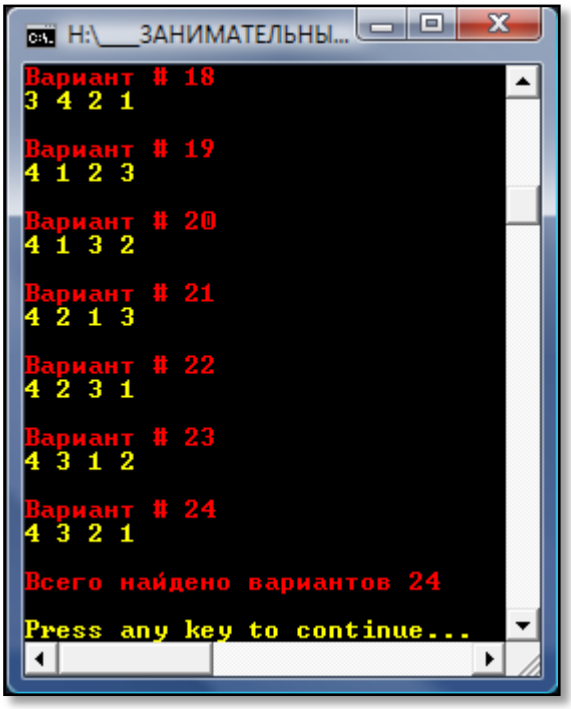


Рис. 38.9. Все решения для доски 4 x 4

Совсем нетрудно вывести решение и в более *наглядном* виде (Рис. 38.10).

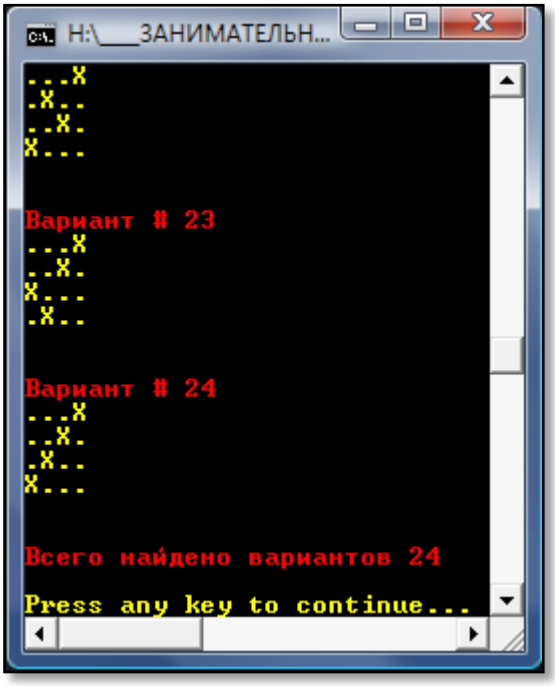


Рис. 38.10. Те же решения, но более наглядно

Можно даже красиво нарисовать решение в графическом окне. Однако для больших полей решений окажется слишком много,

чтобы вырисовывать каждое из них на экране. Правильнее - сохранять все решения в *файле*.

Если для очередной ладьи места на поле не нашлось, значит, нужно *вернуться назад*, к предыдущей ладье и передвинуть её вправо:

```
back:
n=n-1
If (n = 0) Then
  Goto end
EndIf
v= vert[n]
Goto nextVert
```

Как только программа вернётся к *нулевой* ладье, мы можем рапортовать о завершении работы – поиск вариантов закончен. Сообщаем пользователю число расстановок и ждём дальнейших указаний:

```
end:
TextWindow.ForegroundColor="Red"
TextWindow.WriteLine("Всего найдено вариантов " + nVar)
TextWindow.WriteLine("")
TextWindow.ForegroundColor="Yellow"
```

Нам осталось написать процедуру проверки **test**:

```
'Проверка: можно ли поставить очередную
'ладью n на текущую вертикаль v
Sub test
  resolution= "True"
  If (n = 1) Then
    Goto exit
  EndIf
  For i= 1 To n-1
    If (vert[i]=v) then
      resolution= "False"
      Goto exit
    EndIf
  EndFor
  exit:
```

```
EndSub
```

Она очень простая. Сначала мы устанавливаем флажок *resolution* и, если выставляется первая ладья, сразу же завершаем процедуру, поскольку первую ладью *всегда* можно выставить на любую вертикаль. Для каждой ладьи, начиная со второй, следует проверить, не занята ли вертикаль *v* одной из предыдущих ладей. Если занята, мы *сбрасываем* флажок. В противном случае завершаем процедуру с *установленным* флажком.

Запустив программу при разных размерах поля, мы получим такие результаты:

Размер	Число решений
1 x 1	1
2 x 2	2
3 x 3	6
4 x 4	24
5 x 5	120
6 x 6	720
7 x 7	5040
8 x 8	40320

Если эти числа вам ничего не напоминают, то возвратитесь к уроку, на котором мы подсчитывали факториалы. Действительно, если размеры поля равны *n x n*, то первую ладью можно выставить на любую из *n* вертикалей, для второй остается *(n-1)* вертикаль, для третьей - *(n-2)...* и, наконец, для последней - только одна-единственная. Общее число расстановок ладей мы легко найдем, перемножив эти числа, то есть, другими словами, найдя факториал *n!* Вот так неожиданно факториал оказался связан с шахматными ладьями!

Исходный код программы находится в папке **Ладьи**.

Повторяющиеся сходные действия, из которых и состоит перебор с возвратами, можно оформить в виде *рекурсии*. Программа станет насколько короче, настолько же и труднее в понимании.

Напишите программу, которая генерирует одну произвольную (случайную) расстановку ладей на поле.

Урок 39. Занимательная Гауссиана

Кроме ладей, можно расставлять на шахматной доске и другие фигуры так, чтобы они не били друг друга. Самая известная из таких шахматных задач, безусловно, *головоломка о восьми ферзях*.

Впервые задача о расстановке восьми ферзей появилась в 1848 году в немецкой шахматной газете *Schachzeitung* (бьюсь об заклад, что вы никогда не догадаетесь, чт***о*** значит её название!). Автором задачи был Макс Беццель. В 1854 году в этой же газете были напечатаны 40 решений задачи. Однако ещё 21 сентября 1850 года в другой немецкой газете Франц Н***а***ук опубликовал все 92 решения. Правда, сам он тогда ещё не был уверен, что это *все* возможные решения.

Впрочем, эта задача стала одной из самых известных головоломок мира только потому, что она связана с именем великого немецкого математика Карла-Фридриха Гаусса, который летом 1850 года нашел 76 решений этой задачи.

Он придумал очень интересный способ поиска решений. Прежде всего, Гаусс установил, что в каждой вертикали и горизонтали может стоять только один ферзь. Это очевидно вытекает из условия задачи. Таким образом, все решения можно представить в виде *перестановки* чисел *1 2 3 4 5 6 7 8*. Действительно, если ставить ферзей на доску, начиная с первой горизонтали, то каждый из них займет в ней место, отличное от мест других ферзей. Поскольку всего ферзей восемь, и все они стоят на разных вертикалях и горизонталях, то все решения можно записать в виде перестановки восьми чисел.

Количество перестановок восьми разных чисел нам хорошо известно: 8! = 40320. Но, в отличие от ладей, ферзи не могут занимать одну и ту же *диагональ*, поэтому из общего числа перестановок следует убрать те, при которых на одной и той же диагонали окажутся два ферзя или больше.

Гаусс придумал остроумный *перестановочный метод* для проверки диагоналей. Для простоты возьмём доску 5 x 5 клеток и любую перестановку из пяти чисел, например, *1 3 4 5 2*. Сразу и не скажешь, бьют ли ферзи друг друга по диагонали или нет, поэтому подпишем под перестановкой числа от 1 до 5 (число горизонталей доски) в прямом и в обратном порядке и вычислим *сумму* каждой вертикальной пары чисел:

1 3 4 5 2	1 3 4 5 2
1 2 3 4 5	5 4 3 2 1
2 5 7 9 7	6 7 7 7 3

Красным цветом отмечены *равные* суммы для каждого из двух случаев (у нас они совпали, но это необязательно).

В первой расстановке совпали *третья* и *пятая* суммы, это значит, что ферзи в соответствующих горизонталях стоят на ***восходящей*** диагонали. Во второй расстановке мы имеем три одинаковые суммы, поэтому ферзи на *третьей, четвертой* и *пятой* горизонталях расположены на ***нисходящей*** диагонали. Эту позицию легче проследить по рисунку (Рис. 39.1).

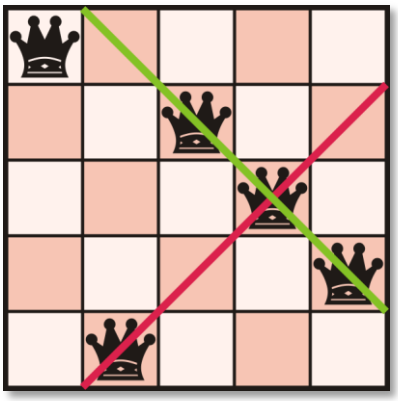


Рис. 39.1. Позиция на доске, соответствующая расстановке *1 3 4 5 2*

Как видите, способ Гаусса чисто механический. Достаточно выписать все перестановки чисел 1..8 (или 1..5, если мы хотим решить задачу на доске 5 x 5 клеток), затем подписать под ними номера горизонталей в прямом и обратном порядке, найти суммы и

сравнить их между собой. Задача осложняется только тем, что для доски 8 x 8 клеток эти действия придётся повторить несколько десятков тысяч раз.

А вот для компьютера подобные задачи – самые простые. В самом деле, если одна и та же операция совершается многократно, а сама по себе она несложная, то и программу для компьютера можно написать за несколько минут. Мы, естественно, не будем кодировать способ Гаусса, который ничего не знал о компьютерах, а просто приспособим программу для расстановки ладей под свои нужды.

Решаем задачу Гаусса

Дописав к нашей программе *Ладья* всего несколько строк, мы решим знаменитую задачу о расстановке ферзей.

Для нас ферзи отличаются от ладей только тем, что они бьют не только вертикаль и горизонталь, но и две *диагонали*, проходящие через клетку с ферзем. Это легко учесть в процедуре проверки *test:*

```
'Проверка: можно ли поставить очередного
'ферзя n на текущую вертикаль v
Sub test
  resolution= "True"
  If (n = 1) Then
    Goto exit
  EndIf
  For i= 1 To n-1
    'проверяем вертикаль:
    If (vert[i] =v ) then
      resolution= "False"
      Goto exit
    EndIf
    'проверяем диагонали:
    If (vert[i]-i=v-n) then
      resolution= "False"
      Goto exit
    EndIf
```

```
    If (vert[i]+i=v+n) then
      resolution= "False"
      Goto exit
    EndIf
  EndFor
  exit:
EndSub
```

Проверка стала сложнее из-за того, что ферзь более «убойная» фигура, чем ладья. Нам нужно проверить, не бьёт ли новый ферзь других ферзей, выставленных ранее, не только по горизонтали и вертикали, но и по восходящей и нисходящей диагоналям. Постарайтесь разобраться, как проходит проверка, потому что в задачах нередко приходится проверять диагонали, а это несколько труднее, чем вертикали и горизонтали.

Запустив нашу программу для поиска расстановок ферзей на стандартной шахматной доске, мы быстро получим список из всех 92-х вариантов (Рис 39.2).

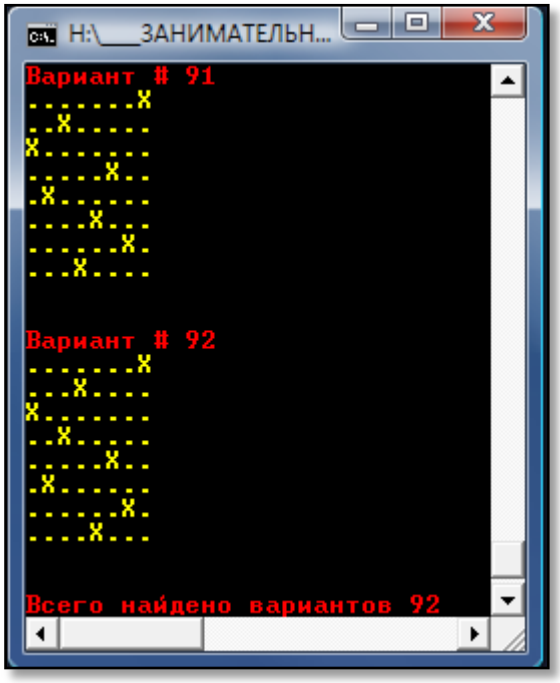


Рис. 39.2. Все расстановки ферзей получены!

Как и в случае с ладьями, вы можете исследовать доски других размеров, даже 10 x 10 клеток и больше:

Размер	Число расстановок
1 x 1	1
2 x 2	0
3 x 3	0
4 x 4	2
5 x 5	10
6 x 6	4
7 x 7	40
8 x 8	92
9 x 9	352
10 x 10	724

Как видите, для ферзей нет такой простой формулы для подсчёта числа решений, как для ладей.

Исходный код программы находится в папке **Ферзи**.

Из 92 расстановок ферзей принято выделять 12 *уникальных*, то есть таких, которые невозможно получить из других расстановок путем поворотов на 90 градусов и зеркальных отражений доски (Рис. 39.3).

Из 10-ой расстановки можно получить только 3 новых решения, из остальных – по 7. С учётом уникальных решений как раз и получаются все 92 расстановки ферзей.

Возьмём, для примера, *первую* расстановку. Поворачивая доску каждый раз на 90 градусов по часовой стрелке, мы получим ещё три новых решения (Рис. 39.4, верхний ряд). Переворачивая (отражая) доску по горизонтали и вертикали, мы получим ещё 4 новых решения (Рис. 39.4, нижний ряд). Вместе с исходным они и дадут 8 расстановок.

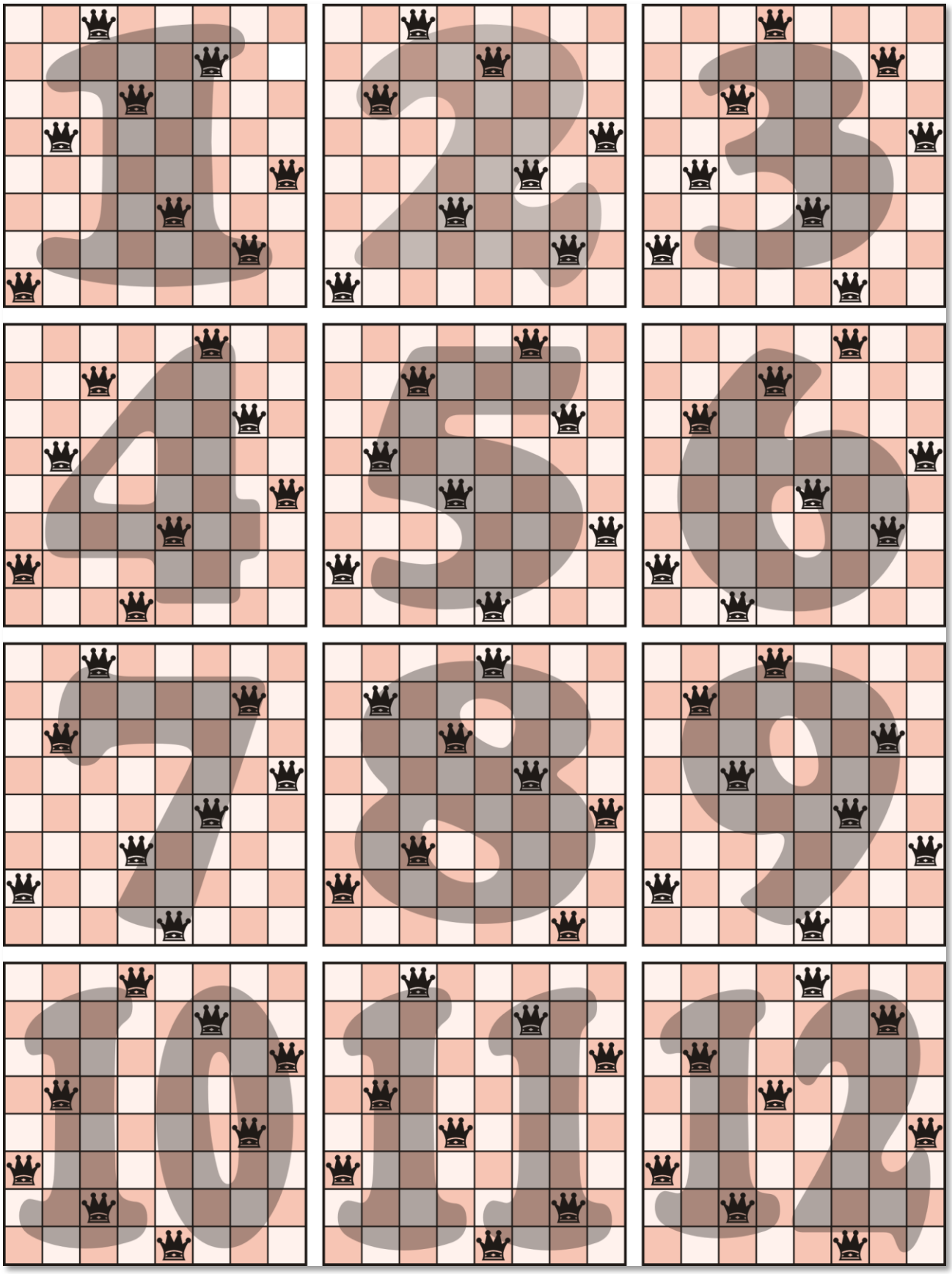


Рис. 39.3. 12 уникальных расстановок ферзей.

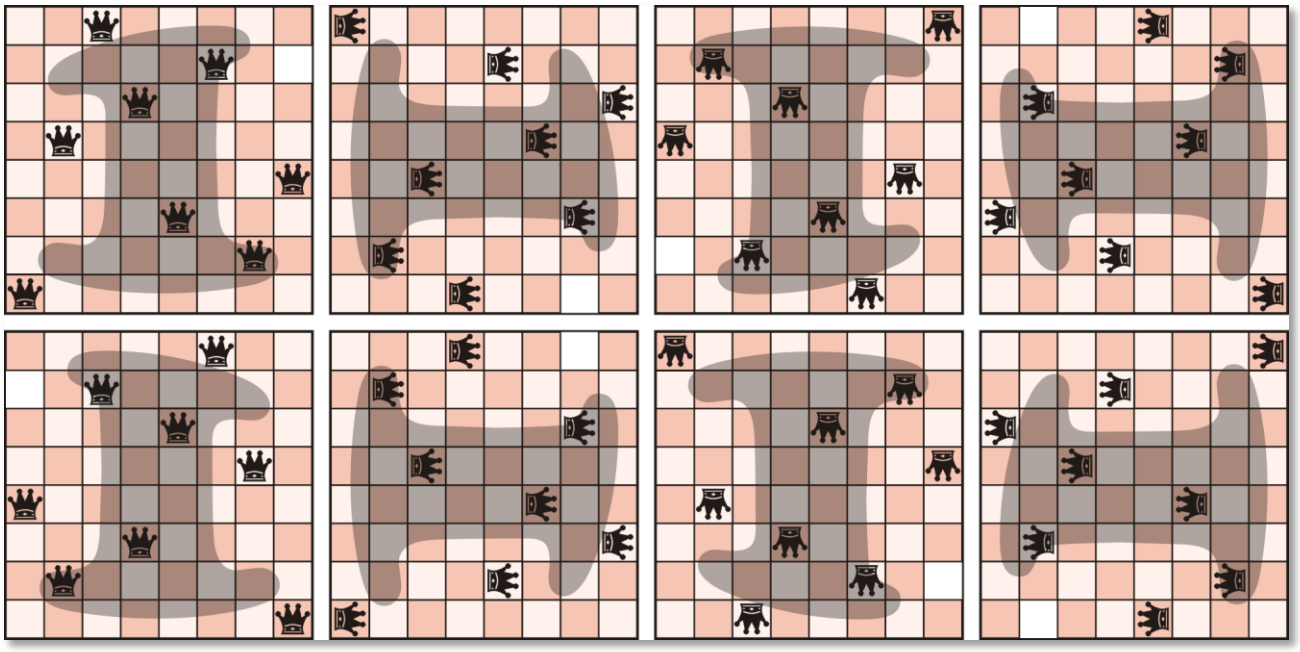


Рис. 39.4. Все расстановки ферзей, полученные из первого решения

На рисунках и ферзи, и номер уникальной расстановки оставлены в том положении, в котором они оказались после поворотов и отражений, потому что так легче проследить, как они были получены.

Исследуя все решения, английский изобретатель головоломок Генри Дьюдени нашёл, что среди 12 уникальных решений, имеется только одно, в котором ни через какую тройку ферзей нельзя провести прямую линию (естественно, не диагональ!). Это 11-ая расстановка на Рис. 39.3.

Напишите программу, которая генерирует одну произвольную (случайную) расстановку ферзей на поле.

Урок 40. Полный перебор

Рассматривая задачи о расстановке ладей и ферзей на шахматной доске, мы прибегли для их решения к методу *перебора с возвратами.* Этот метод требует предварительных размышлений, чтобы построить быстрый алгоритм. Однако представим себе такую ситуацию, что вам нужно решить какую-либо задачу – и забыть о ней. В жизни такие задачи встречаются очень часто. Например, вы хотите решить вот такой *числовой ребус* на сложение двух чисел, зашифрованных буквами:

```
 ОДИН
  ДВА
 ----
 АНОД
```

Вообще говоря, *числовые ребусы* (*числобусы, арифметические ребусы, криптарифмы*) предназначены для *логического* решения, но практически ни из них не решается без перебора вариантов.

В этом случае было бы неоправданной роскошью выстраивать хороший алгоритм – ведь он нам больше никогда не понадобится. Если задача это позволяет, то её следует решить **методом полного перебора**. Английское название *brute force*, что в переводе значит *грубая сила*, более эмоционально выражает суть этого метода программирования. Коли нам нужно единожды решить задачу, то решим её *самым простым* способом. Конечно, алгоритм наверняка получится очень медленным, но зато мы сэкономим время на разработке самого алгоритма. И всё бы хорошо, но нередко число всех возможных вариантов решения задачи столь велико, что экономия времени при написании программы совершенно ничтожна по сравнению со временем поиска решений, которое может достигать многих сотен лет.

По-английски метод называют также *exhaustive search - исчерпывающий поиск*.

Вернёмся к задаче о расстановке ферзей. Общее число возможных способов размещения 8 ферзей на 64-клеточной доске равно 64*63*62*61*60*59*58*57 = 178 462 987 637 760. Действительно, первого ферзя можно поставить на любое из 64 полей, для второго ферзя остается 63 поля, и так далее. Конечно, совершенно немыслимо перебирать все эти варианты, поэтому метод *полного* перебора к этой задач неприменим.

А вот числовые ребусы вполне успешно можно решать методом полного перебора. В *криптарифме* каждая цифра заменена буквой, но, поскольку цифр всего десять, то каждая буква может принимать только 10 разных значений - от 0 до 9. В любом ребусе не более десятка разных букв, то есть в худшем случае нам придётся проверить 10 000 000 000 вариантов. Современным компьютерам это вполне по силам. Впрочем, так бездумно компьютер не используют, поэтому даже при полном переборе следует разумно ограничивать число вариантов. Обычно сделать это очень просто, поскольку некоторые способы лежат на поверхности и не требуют глубоких размышлений. В случае с ферзями достаточно заметить, что на одной горизонтали может стоять только *один* ферзь, чтобы число проверяемых вариантов расстановки уменьшилось до 8*8*8*8*8*8*8*8 = 16 777 216, что совсем немного.

В ребусах также легко найти простой ограничитель числа вариантов – достаточно учесть тот очевидный факт, что все буквы должны иметь *разное* значение. Это следует из условия самой головоломки: разным буквам соответствую разные цифры. Вот теперь можно смело решать любой числовой ребус. Но мы, конечно, решим не любой, а тот самый, что озадачил нас в самом начале урока.

Программа, как это обычно и бывает при полном переборе, очень простая. Записываем столько вложенных циклов *For,* сколько разных букв в ребусе. В нашем примере всего 6 разных букв, поэтому и циклов тоже будет шесть. Начиная со второй буквы, проверяем, чтобы её цифровое представление не совпало с преды-

дущими буквами. Найдя значение каждой буквы, проверяем условие:

```
1000*O + 100*D + 10*I + n + 100*D + 10*V + A <> 1000*A + 100*N
+ 10*O + D
```

Если оно выполняется (то есть левая часть выражения *не равна* правой), то продолжаем поиск решения, в противном случае выписываем найденное решение в *текстовом окне* и ищем другие решения:

```
'ПРОГРАММА ДЛЯ РЕШЕНИЯ ЧИСЛОВОГО РЕБУСА
'МЕТОДОМ ПОЛНОГО ПЕРЕБОРА

'var
nVar=0          'число найденных вариантов решения

TextWindow.Show()

For A= 1 to 9

  For V= 0 to 9
    If (V=A) Then
      Goto nextV
    EndIf

    For D=1 to 9
      If (D=A) Or (D=V) Then
        Goto nextD
      EndIf

      For I= 0 to 9
        If (I=A) Or (I=V) Or (I=D) Then
          Goto nextI
        EndIf

        For N=0 to 9
          If (N=A) Or (N=V) Or (N=D) Or (N=I) Then
            Goto nextN
          EndIf

          For O=1 to 9
            If (O=A) Or (O=V) Or (O=D) Or (O=I) Or (O=N) Then
```

```
            Goto nextO
          EndIf
          If (1000*O + 100*D + 10*I + n + 100*D + 10*V + A
<> 1000*A + 100*N + 10*O + D) Then
            Goto nextO
          Else
            writeSolution()
          EndIf
          nextO:
        endFor
        nextN:
      endFor
      nextI:
    endFor
    nextD:
  endFor
  nextV:
endFor
TextWindow.ForegroundColor="Red"
TextWindow.WriteLine("Всего найдено вариантов " + nVar)
TextWindow.WriteLine("")
TextWindow.ForegroundColor="Yellow"

'Печатаем решение ребуса:
Sub writeSolution
  nVar = nVar +1

  TextWindow.ForegroundColor="Red"
  TextWindow.WriteLine("Вариант # " + nVar)
  TextWindow.ForegroundColor="Yellow"

  s=""
  s= Text.Append(s, O)
  s= Text.Append(s, D)
  s= Text.Append(s, I)
  s= Text.Append(s, N)
  TextWindow.WriteLine(s)
  s=" "
  s= Text.Append(s, D)
  s= Text.Append(s, V)
  s= Text.Append(s, A)
  TextWindow.WriteLine(s)
```

```
  s=""
  s= Text.Append(s, A)
  s= Text.Append(s, N)
  s= Text.Append(s, O)
  s= Text.Append(s, D)
  TextWindow.WriteLine(s)

  TextWindow.WriteLine("")
  TextWindow.WriteLine("")
EndSub
```

В итоге мы найдём все *четыре* решения этого ребуса (Рис. 40.1). Для учебной задачи это вполне допустимо, но правильно составленный ребус должен иметь *единственное* решение!

В программе учтено условие, что первая цифра числа не может быть нулём, поэтому циклы для букв *A, D, O* начинаются с *единицы*. Если немного подумать, то можно ещё более сузить круг поисков. Например, очевидно, что *D > 4*, а *H <> 0*. Нетрудно и дальше улучшать алгоритм, но тогда мы потеряем все преимущества метода полного перебора, который как раз для того и нужен, чтобы решить задачу без долгих раздумий.

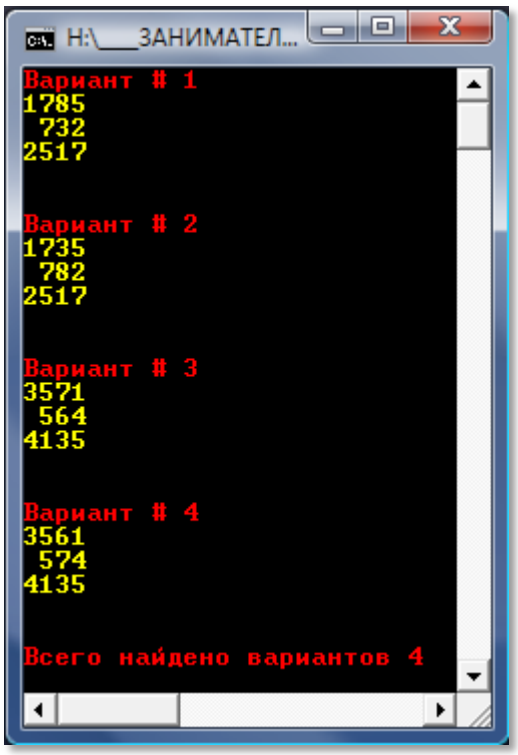


Рис. 40.1. Задача решена!

Исходный код программы находится в папке **Alphametics**.

Взлом паролей

- Пароль!
- Яблоко!
- Не яблоко, а груша! Проходи.

Анекдотичный пример взлома пароля

Некоторые задачи невозможно решить иначе, как только методом полного перебора вариантов, причем сократить перебор принципиально невозможно. Одна из таких задач – *взлом паролей.*

Мы, конечно, не будем взламывать чужих паролей. Наоборот, мы постараемся защитить свои пароли от посягательств.

Наверняка вам много раз приходилось выбирать себе пароль при регистрации на разных сайтах. Обычно для пароля разрешено использовать все 10 цифр и буквы латинского алфавита в верхнем и нижнем регистрах. Итого получается 10 + 26 + 26 = 62 символа. Из них можно составить 62 односимвольных пароля. Естественно, такой пароль разгадать очень просто. Попробуем составить пароль из двух символов: 62 x 62 = 3844 – это уже лучше, но всё равно мало, поэтому минимальная длина пароля должна составлять 6 символов, что дает уже 62^6= 56 800 235 584 варианта. Максимальная длина пароля также ограничивается – 16 символами. Поскольку 62^{16} = 4 767 240 170 682 353 345 026 330 816, то его невозможно отгадать за разумное время.

Почему же тогда хакеры так часто подбирают чужие пароли? – На самом деле пароли чаще крадут, чем взламывают. Если вы храните пароли в файле на компьютере, а антивирусная программа у вас не установлена или плохого качества, то получить доступ к вашей информации совсем нетрудно, а, значит, и ваши пароли могут украсть.

Что касается взлома паролей, то, как мы убедились, длинный пароль, составленный из *случайной* последовательности символов, разгадать нельзя. Но ведь такой пароль, - например, *A8xGzOAgm7* – и запомнить очень сложно, поэтому пользователи нередко выбирают пароли попроще – их запомнить легче.

Предположим, что некий нехороший человек, хакер Редиска решил взломать ваш сайт, то есть получить доступ к файлам на сервере провайдера, подобрав пароль. Редиска знает, что длинный хаотичный пароль от не разгадает никогда, поэтому он полагается на вашу беспечность при выборе пароля. Он очень быстро перебирает все короткие пароли.

Хакер не знает не только пароля, но чаще всего даже его длину.

Если ни один из них не подошёл, значит, ваш пароль состоит из большего числа символов. Полный перебор длинных паролей не годится, но ведь вы вполне могли в качестве пароля выбрать какое-либо осмысленное слово. Например, свою фамилию, имя, кличку собаки, город. Наконец, просто русское слово, записанное латинскими буквами. При всем богатстве выбора так можно загадать не более миллиона паролей, которые легко перебрать в считанные секунды.

Естественно, хакер должен иметь список подходящих слов, но его несложно составить, просеяв тексты в Интернете.

Итак, знание метода полного перебора вариантов помогает нам сделать правильный выбор пароля: он должен состоять не менее чем из шести символов, взятых в случайном порядке. Другой хороший способ выбора пароля – шифрование осмысленных слов одним из методов, рассмотренных нами на уроке *Занимательная криптография*. Тогда и пароль вы легко запомните, и разгадать его будет невозможно.

Естественно, для серьёзного шифрования наши методы не годятся!

А теперь попробуйте составить несколько паролей, беря наугад произвольные буквы и цифры. Довольно утомительное занятие! Все неприятные дела следует поручать компьютеру, поэтому мы сейчас научим его составлять пароли.

Откройте любой проект, в котором есть кнопки и текстовые поля и запишите его в папку **Пароль**.

Так как программа очень простая, то нам понадобятся две *константы* и одна *переменная*. В строке *SYMBOLS* мы будем хранить все символы, из которых можно составлять пароли. Если хотите, можете добавить в неё и что-нибудь от себя.

```
'ПРОГРАММА ДЛЯ ГЕНЕРИРОВАНИЯ ПАРОЛЕЙ

'const
SYMBOLS ="1234567890abcdefghijklmnopqrstuvxyzABCDEFGHIJKLMNOPQRSTUVXYZ"
MAX_LEN= 16    'макс. длина пароля

'var
pass= ""    'пароль

GraphicsWindow.Title="Генератор паролей"

GraphicsWindow.Hide()
GraphicsWindow.Width= 320
GraphicsWindow.Height=140
GraphicsWindow.Left= (Desktop.Width - GraphicsWindow.Width) /
2
GraphicsWindow.Top = (Desktop.Height - GraphicsWindow.Height)
/ 2
GraphicsWindow.CanResize="False"
GraphicsWindow.BackgroundColor= "Black"

height=GraphicsWindow.Height
width= GraphicsWindow.Width
```

Для удобства пользования программой украсим её интерфейс *элементами управления – кнопкой* и двумя однострочными *текстовыми полями*:

```
' КНОПКИ
```

```
'Button:
btnGen=Controls.AddButton("Составить!, 10, 80)
Controls.ButtonClicked=OnClick

' ТЕКСТОВЫЕ ПОЛЯ
GraphicsWindow.FontSize= 12
' длина пароля:
txtLen=Controls.AddTextBox(10, 10)
Controls.SetSize(txtLen,30,20)
Controls.SetTextBoxText(txtLen, 6)
GraphicsWindow.DrawText(70,12, " Введите длину пароля 1.." +
MAX_LEN)
' пароль:
txtPass=Controls.AddTextBox(10, 40)
Controls.SetSize(txtPass,140,20)
Controls.SetTextBoxText(txtPass, "***************")
```

Прежде всего, «шифровальщик» должен задать *длину* пароля. По умолчанию она равна шести символам, но можно выбрать и другую – от одного до шестнадцати символов. Нажав кнопку *Составить!*, пользователь в нижнем *текстовом окне* получает свежий пароль. Если он не годится или нужны и другие пароли, то опять же следует нажать эту кнопку.

А вот так просто наша программа составляет пароли нужной длины:

```
' Генерируем пароль
Sub OnClick
  ' длина пароля:
  len=Controls.GetTextBoxText(txtLen)
  ' проверить:
   if (len < 1) Then
    len = 1
  EndIf
  if (len > MAX_LEN) Then
    len = MAX_LEN
  EndIf
  Controls.SetTextBoxText(txtLen, len)
  pass=""
  For i= 1 To len
    ' выбираем случайную букву из строки символов:
    n= Math.GetRandomNumber(Text.GetLength(SYMBOLS))
```

```
    ch= Text.GetSubText(SYMBOLS,n,1)
    pass= Text.Append(pass, ch)
  EndFor
  Controls.SetTextBoxText(txtPass, pass)
EndSub
```

После очевидных проверок ввода пользователя процедура *OnClick* последовательно выбирает из строки символов *случайный* и добавляет её к «строковой» переменной *pass*. Когда длина пароля достигнет заданной, он будет напечатан в *текстовом поле* (Рис. 40.2).

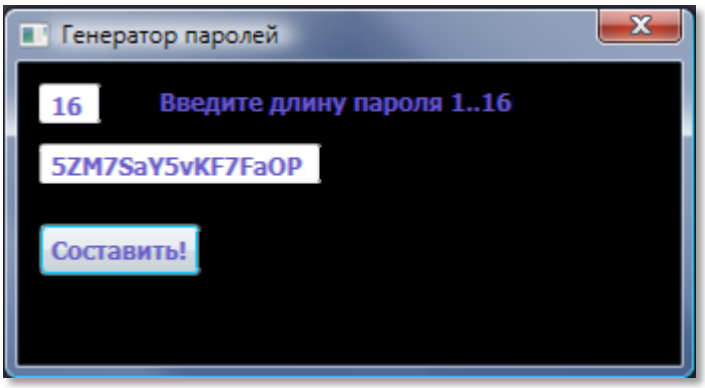


Рис. 40.2. Такой пароль хакеру не по зубам!

Исходный код программы находится в папке **Пароль**.

Решите самостоятельно такие ребусы:

BOR • JOD = ARGON

РАМА • 6 = ОКНО

ЖЕЛЕЗО + ЖЕЛЕЗО = МЕТАЛЛ

Каждый ребус имеет *единственное* решение.

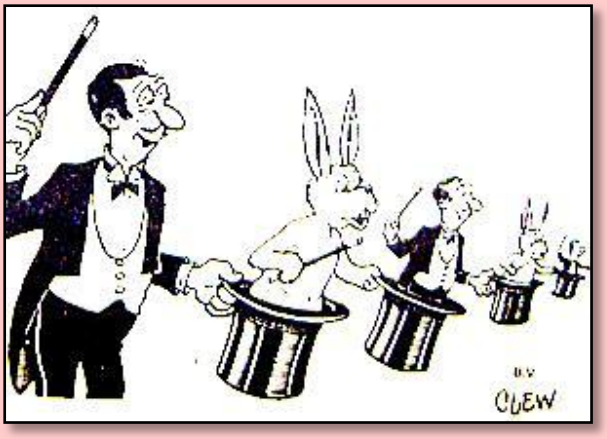
Ответы для проверки решения:

1. 283 x 189 = 53487
2. 1343 x 6 = 8058
3. 304072 + 304072 = 608144

Урок 41. Рекурсия, или Сказочка про белого бычка

- Сказать ли тебе сказку про белого бычка?
- Скажи.
- Ты скажи, да я скажи,
да сказать ли тебе сказку про белого бычка?
- Скажи.
- Ты скажи, да я скажи, да чего у вас будет,
да докуль это будет!
Сказать ли тебе сказку про белого бычка?

Докучная сказка



В теле процедур могут быть записаны вызовы других процедур, то есть из одной подпрограммы можно вызвать другую подпрограмму. Но наиболее интересный случай таких вызовов – это когда подпрограмма вызывает саму себя.

Этот приём программирования называется **рекурсией**.

Хорошим примером рекурсии в *литературе* могут служить *докучные сказки*, которые находчивый русский народ придумал для того, чтобы изводить ими своих врагов, поскольку они никогда не заканчиваются (и те, и другие). Такую рекурсию называют *бесконечной*. В программах она вызывает «зависание» компьютера до тех пор, пока не будут полностью исчерпаны его ресурсы. Естественно, такое поведение приложения совершенно недопустимо, поэтому для рекурсивных подпрограмм необходимо предусмотреть *условие выхода* из рекурсии.

Попробуем запрограммировать другую докучную сказку – про то, как поп убил собаку.

Число повторений сказки будет ограничивать переменная *n*. Она просто указывает программе, сколько раз мы хотим услышать

эту душещипательную историю. Это и будет тот счётчик повторений, который избавит нас от бесконечной рекурсии.

А вот и сама сказочка:

```
'РЕКУРСИВНАЯ ПРОГРАММА ДЛЯ РАССКАЗЫВАНИЯ СКАЗОЧЕК

TextWindow.ForegroundColor="Red"
TextWindow.WriteLine("Сказочка про попа и его собаку")
TextWindow.WriteLine("")
TextWindow.Show()

'n - число повторов сказочки
n= 3
pop()

TextWindow.ForegroundColor="Red"

'Процедура печати сказочки в Текстовом окне
Sub pop
  'n - число повторов

  if n <= 0 then
    Goto exit
  endif
  TextWindow.ForegroundColor="Yellow"
  TextWindow.WriteLine("У попа была собака, он её любил,")
  TextWindow.WriteLine("Она съела кусок мяса, он её убил.")
  TextWindow.WriteLine("В землю закопал и надпись написал:")
  TextWindow.WriteLine("")

  n= n-1
  pop()

  exit:
EndSub
```

В начале процедуры как раз и находится обязательное условие, гарантирующее завершение сказочки. Как только переменная *n* станет равной *нулю*, процедура закончится. Но до этого сказочка будет напечатана 1 раз, а затем переменная *n* уменьшится на единицу

```
n= n-1
```

и процедура *pop* снова вызывает себя

```
pop()
```

для повторения сказки на 1 раз меньше, чем при первом вызове процедуры:

```
n= 3
pop()
```

Таким образом, процедура *pop* будет выполнена 3 раза (если вы не изменили начальное значение), а значение переменной всякий раз будет уменьшаться на единицу, пока не достигнет *нуля*, после чего вызовы прекратятся:

```
n = 3
n = 2
n = 1
n = 0
```

На последнем вызове сработает условие *n <= 0*, и страдания читателя сих милых строк закончатся (Рис. 41.1).

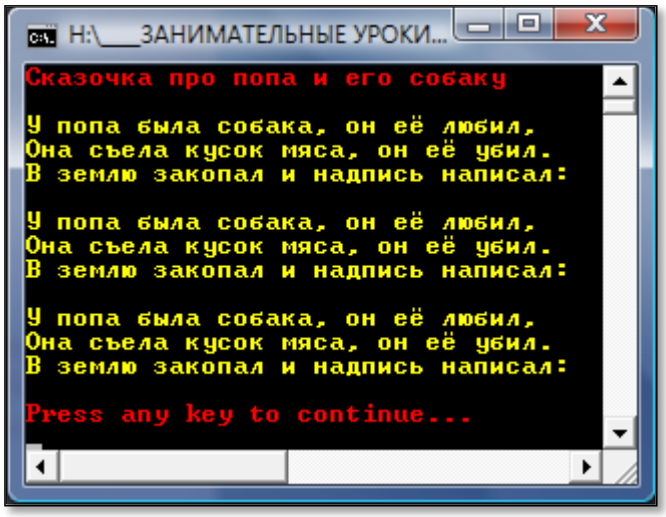


Рис. 41.1. Рекурсивная сказочка при n=3

Действие процедуры *pop* напоминает действиве оператора цикла *While*, поэтому давайте напишем *нерекурсивную* процедуру, что иногда бывает полезно:

```
Sub popWhile
  'n - число повторов
  While n > 0
    TextWindow.ForegroundColor="Yellow"
    TextWindow.WriteLine("У попа была собака, он её любил,")
    TextWindow.WriteLine("Она съела кусок мяса, он её убил.")
    TextWindow.WriteLine("В землю закопал и надпись написал:")
    TextWindow.WriteLine("")
    n= n-1
  EndWhile
EndSub
```

Заменим вызов процедуры *pop* вызовом процедуры *popWhile*:

```
n= 3
'pop()
popWhile()
```

Результат рассказывания сказки будет точно такой же.

Конечно, не всегда так просто избавиться от рекурсии, как в этом примере.

Исходный код программы находится в папке **Pop**.

Передача параметров через стек

В большинстве языков программирования при вызове подпрограмм в скобках указываются параметры, которые передаются в подпрограмму. Вызов процедуры *pop* выглядел бы так:

```
n= 3
pop(n)
```

или так:

```
pop(3)
```

В *СБ* параметры в процедуру передавать нельзя, поэтому приходится пользоваться глобальными переменными, что нередко приводит к трудностям при написании программ. Это связано с тем, что значения глобальных переменных изменяются в подпрограммах, поэтому большинство рекурсивных процедур не будет работать в *СБ*.

Давайте выясним, почему так происходит. При вызове процедуры *pop(3)* значение параметра (то есть тройка в нашем случае) помещается на стек, а в самой процедуре значение снимается со стека и присваивается локальной переменной процедуры. Какое бы имя она ни имела, она не может изменить значение переменной с таким же именем вне процедуры. Все локальные переменные процедуры уничтожаются при её завершении.

Поскольку мы затронули способ передачи параметров в процедуру через стек, то нам нужно поближе познакомиться с работой стека.

Класс *Stack*

Класс **Стек** служит для хранения различных объектов «*стопкой*». Поясним это на примере с книгами.

Сначала стол пустой, на нем книг нет. Затем мы кладём на него первую книгу (Рис. 41.2).

Рис. 41.2. На столе лежит *одна* книга

Затем на первую книгу – вторую, и так далее (Рис. 41.3-6).

Рис. 41.3. В стопке *две* книги

Рис. 41.4. В стопке *три* книги

Рис. 41.5. В стопке *четыре* книги

Рис. 41.6. В стопке *пять* книг

Эти действия можно продолжать до тех пор, пока в доме не закончатся книги или стопка не развалится.

Компьютер не может выстраивать стопки из книг или других материальных объектов. Вместо них он оперирует данными - числами, строками и другими «виртуальными» объектами. В остальном аналогия полная.

Мы можем раскладывать книги в несколько стопок (например, по каждому школьному предмету отдельно), и *СБ* позволяет нам создавать любое число стопок. Для этого в нём имеется *класс Stack*. В переводе с английского *Stack* как раз и значит *стопка*. По-русски и сам класс, и объекты этого класса называют **стеком**.

Поскольку стеков может быть несколько, их нужно отличать друг от друга. Как обычно, для этого каждому стеку присваивается единственное и неповторимое *имя*.

Например, так:

```
'var
_stack="Stack"
```

Дальше мы будем исследовать стек, имя которого хранится в переменной *_stack*. Вначале стек, как и наш стол, пустой, в чем легко убедиться, если вызвать метод

```
Stack.GetCount(stackName),
```

который возвращает число элементов на стеке с заданным именем *stackName*.

В нашем случае после выполнения оператора

```
n= Stack.GetCount(_stack)
```

переменная *n* примет значение равное нулю.

Вы можете указать и полное имя стека:

```
n= Stack.GetCount("Stack")
```

Результат, естественно, будет то же самый, поскольку мы обращаемся к одному и тому же объекту.

Чтобы поместить какой-нибудь объект на стек, нужно вызвать метод

```
Stack.PushValue(stackName, value)
```

Например, положим на наш стек число 1:

```
Stack.PushValue(_stack, 1)
```

Теперь он имеет один элемент, который находится на *вершине* стека.

Аналогично мы можем добавить к стеку сколько угодно объектов:

```
For i= 1 to 12
  Stack.PushValue(_stack,i)
endfor
Stack.PushValue(_stack,"Вершина стека")
```

Важно помнить, что последний объект, который мы кладем на стек, всегда находится на вершине стека – точно так же, как и книги на столе. Таким образом, при добавлении объектов стек растет *вверх*. Из стопки мы можем взять только ту книгу, которая лежит *наверху* (если не хотим уподобиться Трусу из комедии *Операция Ы и другие приключения Шурика*, который из «стека» с горшками вытянул нижний, отчего весь стек рухнул со страшным звоном), из компьютерного стека также нельзя вытянуть объект, не лежащий на вершине. А чтобы снять со стека верхний объект, достаточно воспользоваться методом

```
Stack.PopValue(stackName)
```

Он вернёт тот объект, который находится на вершине стека. При этом число элементов на стеке уменьшится на единицу, объект будет удален из стека и его место на вершине займёт тот объект, который был вторым сверху. Проясним эту ситуацию примером.

Положим на наш стек 12 чисел и строку:

```
For i= 1 to 12
  Stack.PushValue(_stack,i)
endfor
Stack.PushValue(_stack," Содержимое стека:")
```

Посмотрим, как они расположились на стеке:

```
writeStack()

Sub writeStack
  TextWindow.ForegroundColor = "Red"
  TextWindow.WriteLine("Содержимое стека:")
```

```
  TextWindow.ForegroundColor = "Yellow"
  n= Stack.GetCount(_stack)
  For i = 1 to n
    TextWindow.WriteLine(Stack.PopValue(_stack))
  EndFor
  TextWindow.WriteLine("")
EndSub

TextWindow.ForegroundColor = "Red"
```

Как и следовало ожидать, наверху лежит строка, а затем идут числа в обратном порядке – от большего к меньшему (Рис. 41.7).

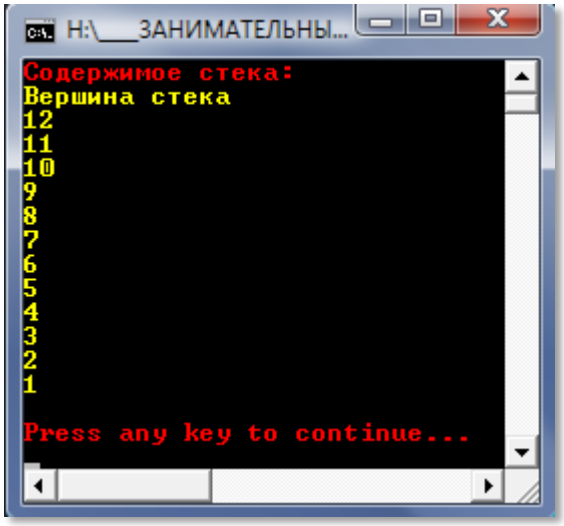


Рис. 41.7. Стек

Вопрос: сколько теперь элементов на нашем стеке? – При печати мы воспользовались методом *PopValue*, чтобы узнать содержимое стека. Этим самым мы сняли со стека *все* элементы, и, значит, он теперь *пустой*. Таким образом, наше любопытство разрушило стек. Если вы хотите контролировать стек, не портя его, создайте ещё один стек

```
_stack2="Stack2"
```

и каждый объект из первого стека укладывайте во второй:

```
Sub writeStack
  TextWindow.ForegroundColor = "Red"
  TextWindow.WriteLine("Содержимое стека:")
  TextWindow.ForegroundColor = "Yellow"
```

```
  n= Stack.GetCount(_stack)
  For i = 1 to n
    obj= Stack.PopValue(_stack)
    TextWindow.WriteLine(obj)
    Stack.PushValue(_stack2, obj)
  EndFor
  TextWindow.WriteLine("")
  For i = 1 to n
    obj= Stack.PopValue(_stack2)
    Stack.PushValue(_stack, obj)
  EndFor
EndSub
```

Естественно, порядок элементов в нём будет противоположным по отношению к первому стеку, в чем легко убедиться, если выполнить программу (Рис. 41.8).

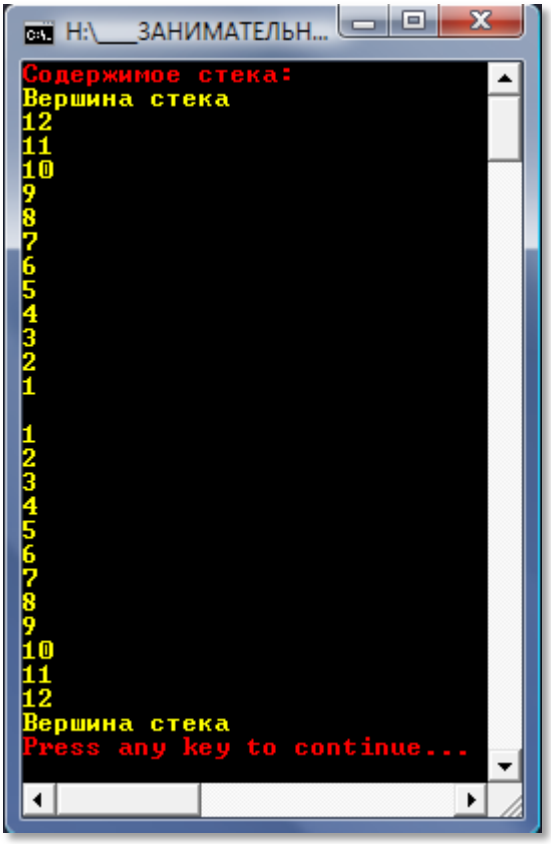


Рис. 41.8. Два стека

Иногда бывает полезно переписать элементы стека в *обратном* порядке (например, если нам нужно инвертировать слово или строку) , но обычно нужно сохранить стек точно таким же, каким

он был до выполнения процедуры печати. Легко догадаться, что повторив операцию копирования стека ещё раз, в противоположном направлении, мы восстановим первый стек:

```
For i = 1 to n
  obj= Stack.PopValue(_stack2)
  Stack.PushValue(_stack, obj)
EndFor
```

Если бы нам был безразличен порядок объектов на стеке, мы могли бы поочередно использовать первый и второй стеки, введя переменную

_stack3

и присваивая ей нужное значение:

_stack3 =_stack
_stack3 =_stack2

Переменная *_stack3* всегда указывала бы на полный стек, независимо от того, первый это стек или второй.

Исходный код программы находится в папке **Stack**.

Старая сказочка на новый лад

Для лучшего понимания работы стека давайте перепишем сказочку так, чтобы число повторов передавалось в процедуру *pop* через стек.

Объявим новую переменную для хранения имени стека:

```
'РЕКУРСИВНАЯ ПРОГРАММА ДЛЯ РАССКАЗЫВАНИЯ СКАЗОЧЕК
'СО СТЕКОМ

'var
_stack= "Stack"
```

```
TextWindow.ForegroundColor="Red"
TextWindow.WriteLine("Сказочка про попа и его собаку")
TextWindow.WriteLine("")
TextWindow.Show()
```

Значение переменной *n* кладём на стек, а затем вызываем процедуру *pop*:

```
'n - число повторов сказочки
n= 3
Stack.PushValue(_stack, n)
pop()

TextWindow.WriteLine("n= " + n)

TextWindow.ForegroundColor="Red"
```

Если число повторов задать *константой*, то на стек нужно положить просто число:

```
Stack.PushValue(_stack, 3)
```

В процедуре *pop* мы снимаем значение со стека и присваиваем его переменной *n*. Затем печатаем сказочку и кладём на стек число на единицу меньше - *n-1* – и вновь вызываем процедуру:

```
'Процедура печати сказочки в Текстовом окне
Sub pop
  'n -  число повторов
  n= Stack.PopValue (_stack)
  if n <= 0 then
    Goto exit
  endif
  TextWindow.ForegroundColor="Yellow"
  TextWindow.WriteLine("У попа была собака, он её любил,")
  TextWindow.WriteLine("Она съела кусок мяса, он её убил.")
  TextWindow.WriteLine("В землю закопал и надпись написал:")
  TextWindow.WriteLine("")

  Stack.PushValue(_stack, n-1)
  pop()

  exit:
```

```
EndSub
```

Строка

```
TextWindow.WriteLine("n= " + n)
```

покажет нам, чему стало равно значение переменной *n* после выполнения процедуры (Рис. 41.9).

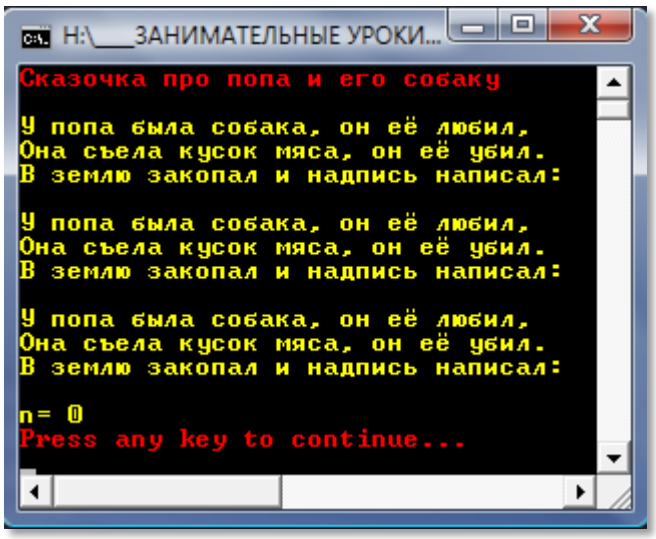


Рис. 41.9. Стековая сказочка

Как и следовало ожидать, оно равно *нулю*, потому что было изменено в самой процедуре.

Если вы хотите сохранить значение глобальных переменных основной части программы, то можно сделать так.

1. Положите значение переменной на стек *два* раза. Одно значение будет использовано в подпрограмме, а второе останется лежать на стеке, откуда его можно снять и вновь присвоить переменной *n*:

```
n= 3
Stack.PushValue(_stack, n)
Stack.PushValue(_stack, n)
pop()
n= Stack.PopValue(_stack)
```

2. Вместо стека используйте вспомогательную переменную:

```
n= 3
m= n
Stack.PushValue(_stack, n)
pop()
n= m
```

3. Введите в подпрограмме «локальную» переменную:

```
Sub pop
  pop_n= Stack.PopValue(_stack)
  if pop_n <= 0 then
    Goto exit
  endif
  TextWindow.ForegroundColor="Yellow"
  TextWindow.WriteLine("У попа была собака, он её любил,")
  TextWindow.WriteLine("Она съела кусок мяса, он её убил.")
  TextWindow.WriteLine("В землю закопал и надпись написал:")
  TextWindow.WriteLine("")

  Stack.PushValue(_stack, pop_n-1)
  pop()

  exit:
EndSub
```

Этот способ наиболее близок к «естественному» - за исключением того, что переменная *pop_n* не является локальной. Однако если задавать локальным переменным имена с префиксом - названием процедуры, то она будет *почти* локальной: просто не используйте в других частях программы такие переменные.

Исходный код программы находится в папке **Stack**.

Рекурсивные программы

Перейдём теперь к несказочной рекурсии – к классическому примеру вычисления *факториала*.

Интерфейс программы мы позаимствуем у проекта *Factorial*, в котором мы вычисляли его *обычным* способом. Начало программы мы оставим без изменений, добавим только *переменную* для хранения имени стека:

```
'ПРОГРАММА ДЛЯ НАХОЖДЕНИЯ
'ФАКТОРИАЛА ЗАДАННОГО ЧИСЛА
'МЕТОДОМ РЕКУРСИИ

'variables
number=0
fact=0
_stack= "Stack"


'========================================
'            ОСНОВНАЯ ПРОГРАММА
'========================================
start:
TextWindow.ForegroundColor="Yellow"
TextWindow.Write("Введите число 0..27 >  ")
number=TextWindow.ReadNumber()

'Проверяем заданное число -->

'Если задано отрицательное число,
'то работу с программой заканчиваем:
if (number < 0) then
goto exit
EndIf

if (number > 27) then
TextWindow.ForegroundColor="Red"
TextWindow.WriteLine("Число должно быть от 0 до 27!")
goto start
EndIf
```

Заданное пользователем число *number* мы кладём на стек, вызываем процедуру *Factorial*, которая через стек возвращает вычисленное значение факториала. Присваиваем его переменной *fact* и печатаем результат на экране. Затем пользователю будет предложено ввести новое число (Рис. 41.10):

```
Stack.PushValue(_stack, number)
Factorial()
'вычисленное значение факториала:
fact = Stack.PopValue(_stack)

TextWindow.WriteLine("")
TextWindow.ForegroundColor="Green"

'Выводим в Текстовое окно
'факториал заданного числа number:
TextWindow.WriteLine(number + "! = " + fact)

exit:
TextWindow.WriteLine("")
TextWindow.ForegroundColor="Yellow"
Goto start
```

В процедуре *Factorial* мы снимаем число со стека и анализируем его. Если оно равно *нулю*, то кладём на стек единицу и заканчиваем вычисления. Если *единице*, то кладем на стек 1 и 0, после чего выполняем процедуру *Factorial* для числа 0. Этот случай мы уже рассмотрели – на стек будет положена единица.

Кладём на стек произведение двух верхних значений на стеке. А сейчас это 1 x 1 = 1. Для единицы на этом вычисления будут закончены, а на стеке будет лежать факториал единицы.

```
'Процедура рекурсивного вычисления
'факториала числа number
Sub Factorial
  'Возвращает вычисленное значение через стек

  'снимаем заданное число со стека:
  n=Stack.PopValue(_stack)
  'если число равно нулю,
  'то его факториал равен 1:
  If (n = 0) Then
    Stack.PushValue(_stack,1)
  Else
    'иначе вычисляем факториал по формуле
    'Factorial = n * Factorial(n-1):
```

```
    Stack.PushValue(_stack,n)
    Stack.PushValue(_stack,n-1)
    'выполняем процедуру при n-1:
    Factorial()
    'сохраняем на стеке значение  n * (n-1)!:
    Stack.PushValue(_stack,     Stack.PopValue(_stack) *
Stack.PopValue(_stack))
  EndIf
EndSub
```

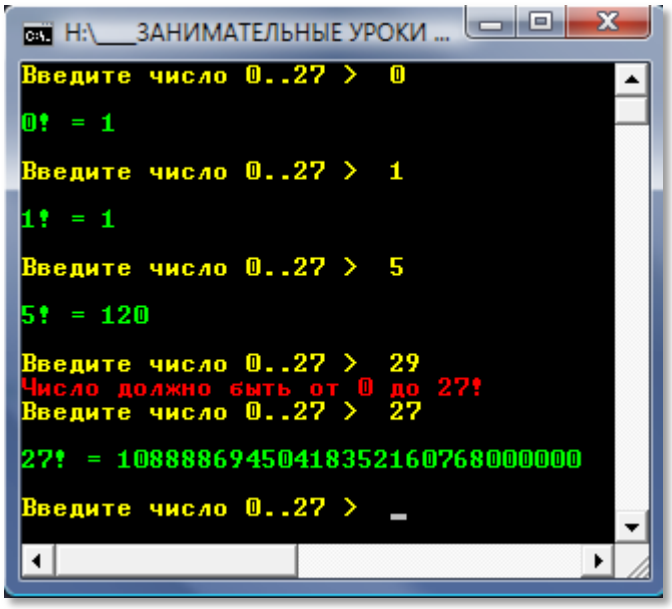


Рис. 41.10. Рекурсивное вычисление факториала

Разберём выполнение процедуры для *n=2*. При первом обращении к процедуре *Factorial* на стеке лежит число 2, которое снимается со стека и присваивается переменной *n*. Стек пуст.

Так как *n > 2*, то выполняется условие *Else*: кладём на стек числа 2 и 1. С вершины стека мы видим числа в обратном порядке: 1 2.

Выполняем процедуру *Factorial*. Снимаем со стека верхнее значение (1) *n = 1*. На стеке осталась двойка.

Так как *n > 1*, то снова выполняется условие *Else*: кладём на стек числа 1 и 0. С вершины стека мы видим числа: 0 1 2.

Выполняем процедуру *Factorial*. Снимаем со стека верхнее значение (0) *n = 0*. На стеке остались числа 1 2. Теперь условие *n = 0* выполняется. Поэтому на стек добавляется единица: 1 1 2.

Второй вызов процедуры *Factorial* закончен. Теперь выполняется оператор

```
Stack.PushValue(_stack, Stack.PopValue(_stack) *
Stack.PopValue(_stack))
```

Со стека снимаются два верхних числа, а на стек возвращается их произведение - 1 2.

Закончен первый вызов процедуры *Factorial*. Онять со стека снимаются два верхних числа, а на стек укладывается их произведение: 2.

На этом выполнение процедуры *Factorial* полностью закончено, и управление программой возвращается в строку

```
'вычисленное значение факториала:
fact = Stack.PopValue(_stack)
```

Затем вычисленное значение печатается на экране.

Очень важно разобраться в механизме «самовызова» подпрограмм, так как рекурсивные процедуры обычно записываются очень коротко, но при этом выполняют множество действий, последовательность которых не так-то просто понять.

Добавим в нашу программу отладочную процедуру *writeStack*:

```
Sub writeStack
  TextWindow.ForegroundColor = "Yellow"
  ws_n= Stack.GetCount(_stack)
  If (ws_n=0) Then
    TextWindow.WriteLine("Стек пуст!")
  Else
    For i = 1 to ws_n
      obj= Stack.PopValue(_stack)
```

```
      TextWindow.Write(obj + " ")
      Stack.PushValue(_stack2, obj)
    EndFor
    TextWindow.WriteLine("")
    For i = 1 to ws_n
      obj= Stack.PopValue(_stack2)
      Stack.PushValue(_stack, obj)
    EndFor
  EndIf
EndSub
```

Это немного изменённая версия из проекта *Stack*.

Теперь в те места процедуры *Factorial*, где записаны стековые операции, мы добавим отладочные операторы, которые позволят нам наблюдать за стеком:

```
Sub Factorial
  n=Stack.PopValue(_stack)
  TextWindow.WriteLine("")
  TextWindow.WriteLine("n= " + n)
  TextWindow.Write("1. ")
  writeStack()
  If (n = 0) Then
    Stack.PushValue(_stack,1)
    TextWindow.Write("2. ")
    writeStack()
  Else
    Stack.PushValue(_stack,n)
    Stack.PushValue(_stack,n-1)
    TextWindow.Write("3. ")
    writeStack()
    Factorial()
    Stack.PushValue(_stack, Stack.PopValue(_stack) *
Stack.PopValue(_stack))
    TextWindow.Write("4. ")
    writeStack()
  EndIf
EndSub
```

Добавьте также *переменную* с именем второго стека:

```
_stack2="Stack2"
```

И *переменную* для подсчёта вызовов процедуры:

```
start:
vyzov=0
```

Запускаем программу и наглядно видим все метаморфозы, происходящие со стеком (Рис. 41.11).

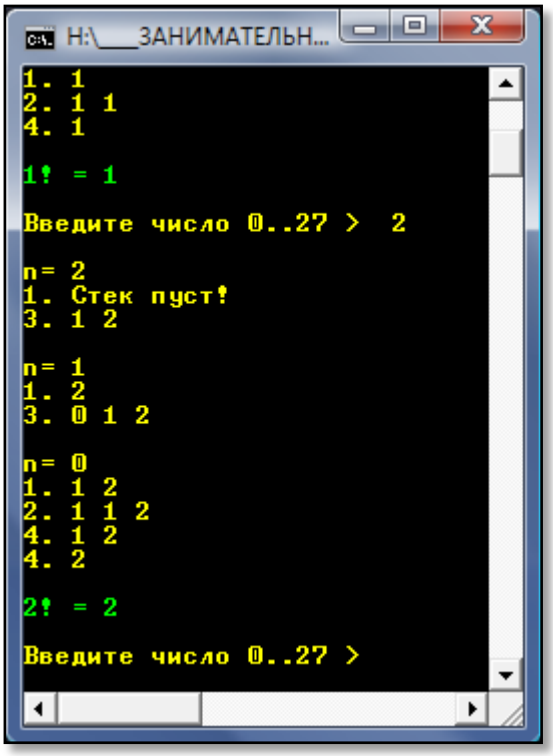


Рис. 41.11. Содержимое стека

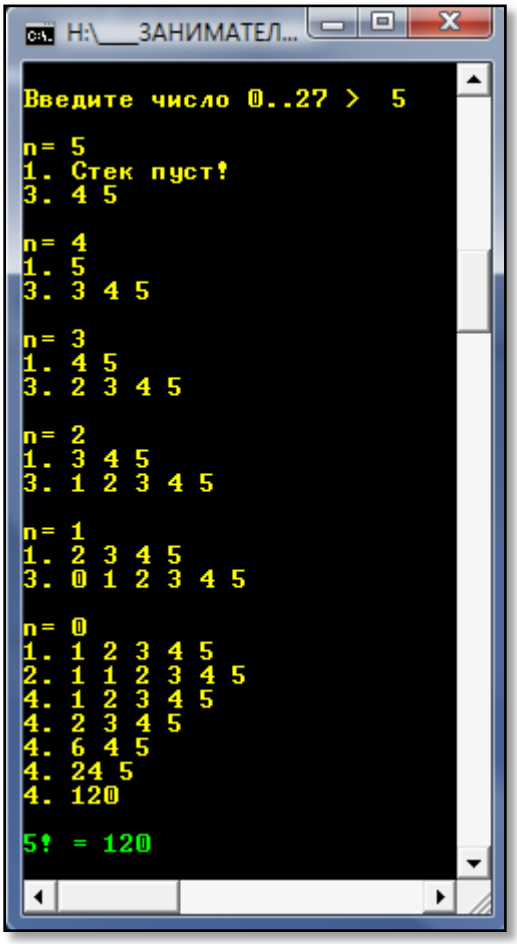


Рис. 41.12. Этапы большого рекурсивного пути

Если для двойки мы ещё можем без труда следить за стеком, то для пятёрки это сделать без ошибок уже практически невозможно, а вот наша программа справляется со «слежкой» без труда (Рис. 41.12).

Исходный код программы находится в папке **FactorialR**.

Переделайте программу для подсчёта заданного числа *Фибоначчи* рекурсивным методом. Вычисляйте только заданное число, а не все числа, не превосходящие заданного. Поскольку глубина рекурсии в этом случае большая (Рис. 41.13), то скорость программы очень низкая, поэтому рекурсивный способ получения чисел Фибоначчи не имеет практического значения.

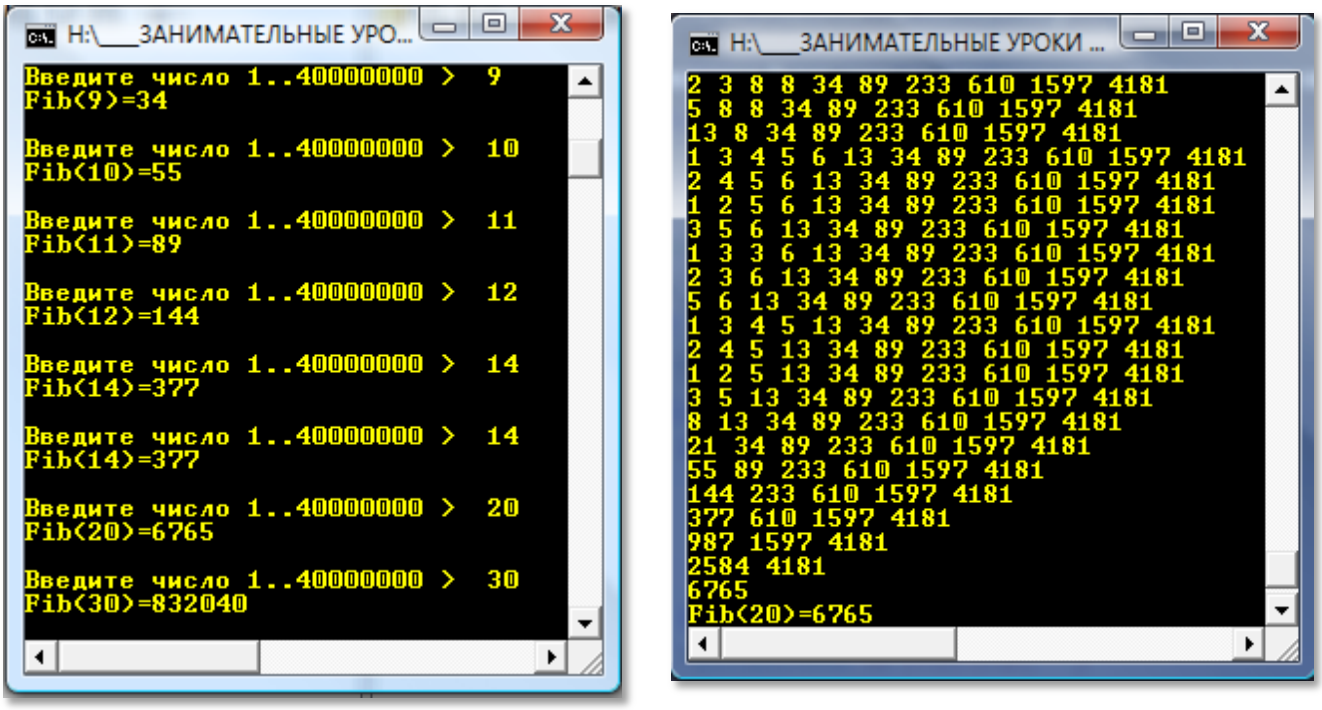


Рис. 41.13. Стек при Number = 20

Исходный код программы находится в папке **FibonacciR**.

Урок 42. Занимательная география

Снятся людям иногда
Их родные города
Кому Москва, кому Париж…

Песня Эдуарда Хиля *Голубые города*

Помните, как Доцент и его команда в кинокомедии *Джентльмены удачи* играли в *Города*? – Впрочем, игра всем известна с детства и без кинокомедии. Суть её заключается в том, чтобы последовательно называть города так, чтобы первая буква каждого следующего слова совпадала с последней буквой последнего названного. Проигрывает тот участник, который не сможет на своём ходу назвать походящего слова.

Вот пример такой игры: *Москв**а*** – ***А**рхангельс**к*** – ***К**иев* – ***В**олгогра**д*** – ***Дел**и* – ***И**ркутс**к*** - … Поскольку городов на земле очень много, то игра может и затянуться. В этом случае можно ограничить выбор городов какой-нибудь страной или часть света.

Можно предложить и такой вариант этой игры, который более «географичен», а потому и более полезен. Возьмём карту мира и совершим вояж по городам, придерживаясь основного принципа игры, то есть названия городов должны составлять такую же цепочку слов, как и в основном варианте игры.

Если карта мира небольшая по размерам (а для игры она подойдет, конечно, лучше, чем карта в полстены!), то на ней будут отмечены только *большие* города. Например, мы можем воспользоваться вот такой таблицей и для определённости считать, что путешествовать мы будем только по ним.

Город	Население	Площадь ($км^2$)	Страна
Мехико	18 841 916	1 485	Мексика
Шанхай	16 492 800	289,44	КНР
Карачи	18 140 670	3530	Пакистан
Стамбул	8 566 823	1538,9	Турция
Токио	8 742 995	621,9	Япония
Мумбаи	13 922 125	603	Индия
Буэнос-Айрес	13 356 715	4000	Аргентина

Дакка	12 725 000	815	Бангладеш
Манила	12 285 000	636	Филиппины
Дели	11 954 217	1483	Индия
Москва	10 562 099	1081	Россия
Сеул	10 421 782	605,4	Республика Корея
Киншаса	10 076 099	10 550	ДР Конго
Сан-Паулу	10 037 593	1520	Бразилия
Лагос	9 360 883	999	Нигерия
Джакарта	8 576 788	660	Индонезия
Нью-Йорк	8 140 993	1214	США
Лима	8 057 397	804,3	Перу
Каир	7 947 121	210	Египет
Пекин	7 712 104	1370	КНР
Лондон	7 581 052	1580	Великобритания
Богота	7 137 849	1590	Колумбия
Гонконг	7 102 354	1104	КНР
Тегеран	6 897 547	660	Иран
Лахор	6 747 238	1010	Пакистан
Багдад	6 431 839		Ирак
Рио-де-Жанейро	6 193 265	1180	Бразилия
Бангкок	5 698 435	1100	Таиланд
Бангалор	5 180 533	230	Индия
Сантьяго	5 090 824		Чили
Калькутта	5 021 458		Индия
Сингапур	4 974 232	699	Сингапур
Янгон	4 886 305		Мьянма
Эр-Рияд	4 606 888		Саудовская Аравия
Санкт-Петербург	4 600 310	1439	Россия
Ченнай	4 562 843		Индия
Чунцин	4 406 788		КНР
Сиань	4 305 536		КНР
Ухань	4 262 236	400	КНР
Александрия	4 247 414	2680	Египет
Ибадан	4 149 487		Нигерия
Сидней	4 114 710	1212	Австралия
Кано	4 082 050		Нигерия
Хайдарабад	3 980 938	170	Индия
Лос-Анджелес	3 975 590	1290	США
Анкара	3 882 639	2500	Турция
Чэнду	3 915 259		КНР
Абиджан	3 900 546		Кот-д’Ивуар
Пусан	3 899 140	771	Республика Корея
Ахмадабад	3 867 336	190	Индия
Тяньцзинь	3 682 177		КНР
Омдурман	3 667 982		Судан
Берлин	3 599 000		Германия
Чикаго	3 573 721		США
Мельбурн	3 528 690		Австралия

Если вы с этим выбором не согласны, ничто не мешает вам отобрать города по собственному желанию. Для нас важно только одно – найти такой маршрут, чтобы посетить *как можно больше городов*, поскольку совершенно очевидно, что все города в одну цепочку уложить не удастся. Задача сама по себе непростая, но ещё труднее доказать, что найденная цепочка городов и есть самая длинная из всех возможных. Так или иначе, но лучше решение найти (или только проверить) с помощью компьютера.

Компьютерный навигатор

Прежде всего, нам потребуется список городов, по которым мы и проложим наш маршрут. Названия городов из выделенной колонки (см. Табл.) мы перевёдем в ВЕРХНИЙ регистр (это можно сделать и в самой компьютерной программе, и в текстовом редакторе), затем скопируем в *Блокнот* и сохраним файл *города.txt* в кодировке *UTF-8*.

Теперь можно браться за программу **Города**, которая и составит нам цепочку слов. Поскольку мы сохранили список названий городов в файле (а это гораздо удобнее, чем присваивать значения элементам массива в исходном тексте программы!), то этот список нам нужно загрузить с диска. Эта необходимость подталкивает нас взять за основу нового проекта любой из уже готовых, в котором мы загружали словарь. Например, подойдёт *Латиница*.

Часть *переменных*, отвечающих за загрузку файла в массив *spisok*, мы оставим без изменения:

```
'ПРОГРАММА ДЛЯ ПОИСКА ЦЕПОЧЕК СЛОВ

'variables
spisok[0]=""    'массив-список названий городов
nWords=0        'число слов в списке
chr=""          'буква
len=0           'длина слова
txt=""          'вспомогательная переменная для считывания
файла
index=0         'текущее значение индекса
```

```
beg=0            'индекс начала слова
```

Однако нам их маловато будет – только загрузить список, поэтому добавим переменные конкретно для нашей программы:

```
flgChain[0] = "False" 'если флаг равен True, значит, слово
стоит в цепочке
begLetter[0]="" 'массив начальных букв слов
endLetter[0]="" 'массив конечных букв слов
n=0             'число уже выставленных слов
maxChain=0      'макс. длина цепочки
idWord[0]=0     'номер слова в списке, которое стоит в цепочке
id=0            'текущий номер слова в списке для слова,
                'которое мы ставим в цепочку
```

Об их назначением мы подробно поговорим по ходу написания программы, которую начнём с *загрузки* списка слов в массив *spisok*:

```
'========================================
'            ОСНОВНАЯ ПРОГРАММА
'========================================
TextWindow.ForegroundColor="Red"
TextWindow.WriteLine("ИЩЕМ ЦЕПОЧКИ ГОРОДОВ")
TextWindow.WriteLine("")
TextWindow.Show()

'считать файл:
fileName= "города.txt"
readFile()
```

Процедуру *readFile* мы обсуждать не будем, потому что мы уже много раз использовали её в других проектах. Для нас только важно помнить, что теперь все слова находятся в массиве *spisok*, а их число надёжно хранится в переменной *nWords*.

Дальше мы поступим хитро и обзаведёмся двумя *массивами* - для первых *begLetter* и последних *endLetter* букв всех слов в списке. Действительно, слова в цепочке цепляются друг за друга так, что первая буква следующего слова должна быть точно такой же, как и последняя буква предыдущего (у первого слова цепочки пред-

шественника нет, поэтому его можно ставить в цепочку безбоязненно). Можно каждый раз извлекать эти буквы из слов и сравнивать их между собой, но эти действия потребуют больше времени, чем в случае с двумя массивами. Остальные буквы слов нас вообще не интересуют, так что для составления цепочки нам будет достаточно только этих массивов. Их заполнение – дело совершенно незамысловатое, но мы должны учесть специфику нашей программы: каждое слово можно использовать в цепочке только один раз. Значит, мы как-то должны отмечать *цепочечные* слова. Проще всего для этого завести массив *flgChain* и заносить для каждого слова значение *False*, если слово в цепочке отсутствует, и *True* - в противном случае.

До построения цепочки все слова, естественно, находятся вне цепочки:

```
'заполняем массивы начальных и конечных букв:
for i=1 to nWords
  begLetter[i]= Text.GetSubText(spisok[i],1,1)
  len= Text.GetLength(spisok[i])
  endLetter[i]= Text.GetSubText(spisok[i],len,1)
  'ни одно слово не стоит в цепочке:
  flgChain[i] = "False"
EndFor
```

Сначала в цепочке нет ни одного слова:

```
'цепочка пустая:
maxChain=0
'ни одно слово еще не выставлено:
n=0
```

Вспомним условие задачи: построить цепочку слов *максимальной* длины. Для того чтобы определить, какая цепочка самая длинная, мы должны составить *все* возможные цепочки и выбрать из них нужную. Это типичная задача *перебора с возвратами*, поэтому в построении алгоритма мы будем опираться на *ладейную* головоломку, но с учётом того, что слова следует выставлять в *одну* цепочку, а не на *разные* горизонтали шахматного поля.

Начало – традиционное для переборных программ: ставим *следующее* слово в цепочку: первое (оно тоже следующее – для пустой цепочки), второе и так далее, насколько возможно. Аналог слова, которое мы добавляем к цепочке, - это новая ладья.

```
'ставим следующее слово в цепочку:
nextWord:
n=n+1
```

Итак, нам нужно выбрать слово из списка, которое отсутствует в цепочке. Начинаем поиск очередного слова для цепочки от начала массива *spisok*. По смыслу это действие совпадает с перемещением ладьи вправо по горизонтали. Если список закончился, а ни одного нового слова в цепочку поставить не удалось, то нужно возвращаться назад. В этой программе мы, конечно, вернёмся не к предыдущей выставленной ладье, а к предыдущему слову в цепочке.

```
idWord[n]=0
id=0
nextId:
id= id+ 1
If (id > nWords) Then
  Goto back
EndIf
```

Опять поступаем аналогично ладейной программе: проверяем, возможно ли очередное слово из списка присоединить к цепочке. Если нет, то переходим к следующему слову в списке:

```
test()
If resolution="False" then
  Goto nextId
EndIf
```

Что касается процедуры проверки, то здесь она совершенно другая, поскольку именно проверка существенно зависит от особенностей конкретной задачи, в то время как алгоритм перебора изменяется не столь радикально. Но, как и в случае с ладьями или ферзями, первое слово можно ставить в цепочку без дополни-

тельной проверки, так как оно никак не зависит от других слов. Нельзя поставить в цепочку и уже выставленное слово, и, наконец, главная проверка – на совпадение букв:

```
'Проверка: можно ли поставить очередное
'слово в цепочку
Sub test
  resolution= "True"
  If (n = 1) Then
    Goto exit
  EndIf
  'слово уже стоит в цепочке:
  If (flgChain[id] = "True") Then
    resolution= "False"
    Goto exit
  EndIf
  'первая буква очередного слова должна совпадать
  'с конечной буквой предыдущего слова:
  If (begLetter[id] <> endLetter[idWord[n-1]]) Then
    resolution= "False"
    Goto exit
  EndIf
  exit:
EndSub
```

В результате проверки переменная *resolution* принимает значение *False*, если слово не подходит, и *True* – если годится.

Первое слово в нашем списке – *Мехико*. Оно и начнёт цепочку. В программе это действие выполняется так. В массиве *idWord* мы запоминаем номер слова из списка, которое дополнило цепочку, а также отмечаем его в массиве *flgChain* как использованное, чтобы у программы не было соблазна использовать его вторично. Нам в этой программе не нужны вообще все цепочки, которые можно составить из слов списка. Нет – нам нужна только *самая длинная*. Её длина хранится в переменной *maxChain*, которую мы обнулили в самом начале программы. Естественно, уже первое слово создаст цепочку *единичной* длины. Поэтому максимальная длина станет равной единице, и мы сможем напечатать все слова, из которых цепочка состоит, после чего постараемся удлинить её ещё на одно слово.

```
'поставили слово в цепочку:
idWord[n]=id
flgChain[idWord[n]]= "True"
'проверяем длину цепочки:
If (n > maxChain) Then
  maxChain= n
  writeChain()
EndIf

Goto nextWord
```

По условию *n > maxChain* будет напечатана только *первая* цепочка максимальной длины. Иногда таких цепочек может оказаться и *несколько*. Если вы хотите лицезреть их все, то исправьте условие на *n >= maxChain*.

В процедуре печати мы выводим в *текстовое поле* все *n* слов цепочки из списка. Номера слов в списке легко узнать по значению элемента массива *idWord[i]*, соответствующего *i*-тому слову в цепочке.

```
Sub writeChain
  TextWindow.ForegroundColor="Red"
  TextWindow.WriteLine("Длина цепочки = " + n)
  TextWindow.ForegroundColor="Yellow"
  For i= 1 To n
    TextWindow.WriteLine(spisok[idWord[i]])
  EndFor

  TextWindow.WriteLine("")
  TextWindow.WriteLine("")
  TextWindow.Read()
endsub
```

Поставив слово *МЕХИКО* в цепочку, мы переходим к поиску второго слова: *n=2*. Просматриваем список городов с самого начала: *id=1*. Проверка возвращает отрицательный результат: слово *МЕХИКО* уже выставлено. Переходим к следующему слову: *id= 2*. Это *ШАНХАЙ*. Это слово не использовано, но первая буква – *Ш* – не совпадает с последней буквой – *О* – предыдущего слова. Результат опять отрицательный. И так мы будем просматривать весь

список, пока не дойдём до слова *ОМДУРМАН* в самом конце списка. Добавляем его к цепочке и ищем третье слово. Легко находим, что это слово *НЬЮ-ЙОРК*. Затем присоединяем *КАРАЧИ – ИБАДАН*, и стоп: больше ни одного слова, начинающегося на букву *Н* в списке нет. Нужно возвращаться *назад*!

```
back:
'переходим к предыдущему слову:
n=n-1
If (n = 0) Then
  Goto end
EndIf

id= idWord[n]
'убираем слово  n из цепочки:
flgChain[idWord[n]]= "False"
Goto nextId
```

Почти дословно эта процедура повторяет *ладейную*: переходим к предыдущему выставленному слову и, если слов в цепочке больше не осталось, то наши поиски закончены на самом интересном месте, иначе убираем его из цепочки (поэтому его снова можно использовать!) и продолжаем поиски со следующего слова в списке.

Наша программа, как и большинство других переборных программ, совсем несложная, но неминуемо находит самую длинную цепочку, какую только можно составить из слов заданного списка. Но если список очень длинный, то придётся запастись недюжинным терпением!

Для нашего виртуального путешествия самый длинный маршрут состоит из 15 городов (Рис. 42.1).

Исходный код программы находится в папке **Города**.

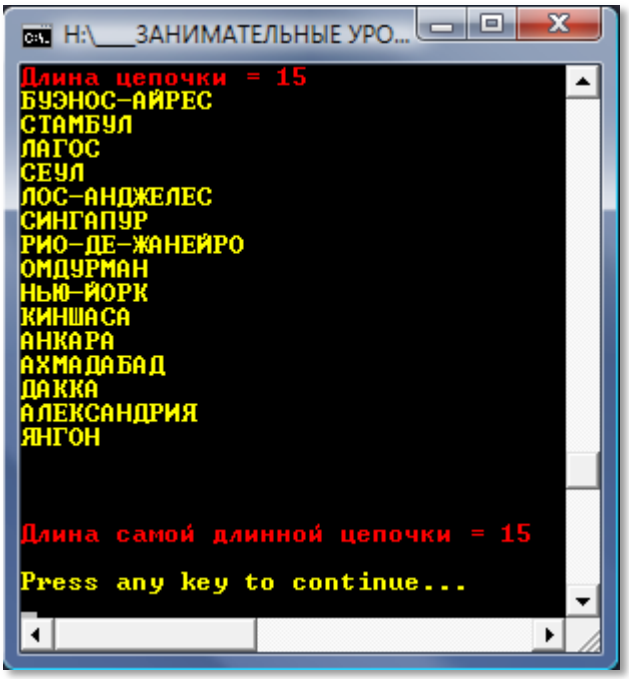


Рис. 42.1. А вы сумели проложить такой маршрут?

Чайнворды

Полученная нами цепочка городов в *головоломковедении* называется *чайнвордом*, а это значит, что мы без всяких переделок можем использовать нашу программу и для составления чайнвордов. Но лучше начать новый проект под названием **Chainword** и сохранить в папке под соответствующим именем нашу программу.

Если у вас на примете есть подходящий словарик, то можете использовать его, а нет – так попробуйте из моих запасов - *морской.txt*.

Наша программа составит цепочку немалой длины (больше ста слов!), причём можно в**ы**резать из цепочки кусок так, чтобы первая буква первого слова совпала с последней буквой последнего слова (Рис. 42.2).

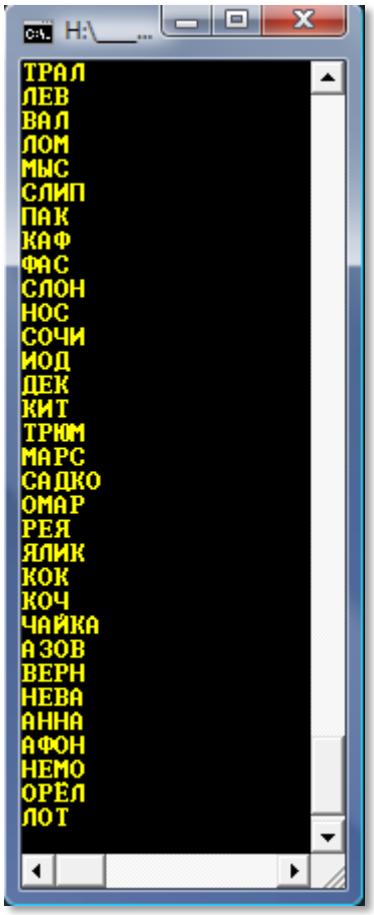


Рис. 42.2. Цепь замкнулась!

Мы легко соединим первое и последнее звенья цепочки и получим *замкнутый чайнворд*.

Не пытайтесь повторить этот трюк с нашими городами!

К сожалению, каждый раз мы будем получать одну и ту же цепочку, потому что наш алгоритм перебирает слова в том самом порядке, в котором они записаны в словаре. Но мы легко изменим этот порядок, если *перемешаем* слова:

```
'Перемешать слова в списке
Sub randomize
For i= 1 to nWords
  n1= Math.GetRandomNumber(nWords)
  n2= Math.GetRandomNumber(nWords)
  s= spisok[n1]
  spisok[n1]= spisok[n2]
```

```
  spisok[n2]= s
endfor
```

Осталось после загрузки файла вызвать процедуру *randomize*:

```
'считать файл:
fileName= "морской.txt"
readFile()
randomize()
```

Теперь после каждого запуска программы вы будете получать *новый* чайнворд (Рис. 42.3).

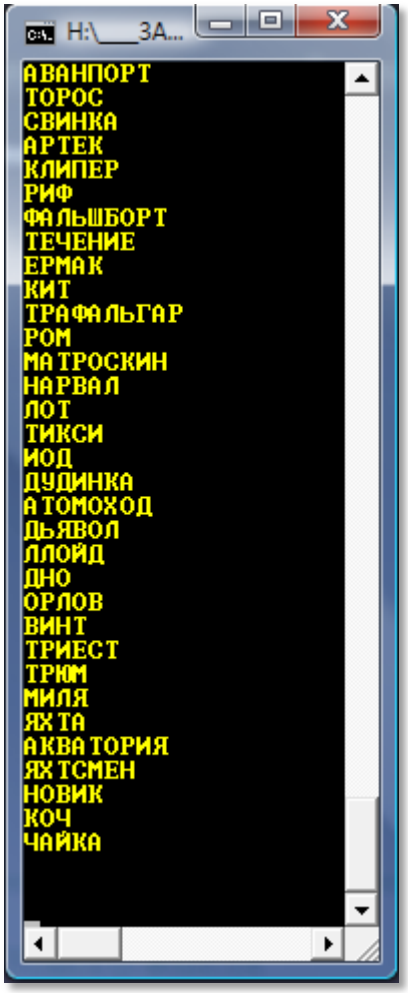


Рис. 42.3. Морской чайнворд

Вот так просто вы можете составлять сколько угодно чайнвордов – если, конечно, предварительно приготовите словарик с инте-

ресными толкованиями слов. Вообще говоря, составить чайнворд совсем нетрудно, поэтому красивое оформление сетки и неожиданные толкования – вот что может сделать вашу задачу по-настоящему занимательной!

Исходный код программы находится в папке **Chainword**.

Играем в города

А теперь вернёмся к началу урока и научим компьютер играть в *Города*. Его противником будет пользователь, который всегда называет первое слово.

Поскольку игра серьёзная, то и *переменных* нам понадобится немало. Большинство из них вам уже хорошо известны, а названия новых говорят сами за себя. Для укрепления знаний о *стеке* мы будем передавать параметры в процедуры через стек.

```
'ПРОГРАММА ДЛЯ ИГРЫ В ГОРОДА

'variables
spisok[0]=""    'массив-список названий городов
nWords=0        'число слов в списке
chr=""          'буква
len=0           'длина слова
txt=""          'вспомогательная переменная для считывания
файла
index=0         'текущее значение индекса
beg=0           'индекс начала слова

flgChain[0] = "False" 'если флаг равен True, значит, слово
стоит в цепочке
gorod=""                'очередной город
lenChain=0              'длина цепочки
_stack = "Stack"        'стек для передачи параметров в
процедуры
```

Перед началом игры нужно загрузить список с названиями городов. В файле *города3.txt* записаны больше тысячи российских городов. Отброшены те города, названия которых состоят из двух и более отдельных слов. Обычно такие слова не используют в словесных головоломках и играх. К сожалению, этот список имеет пробелы, поэтому вы можете дополнить его новыми названиями.

Здесь нужно обратить внимание на то, что название файла мы передаём в процедуру через стек:

```
'=========================================
'             ОСНОВНАЯ ПРОГРАММА
'=========================================
TextWindow.ForegroundColor="Red"
TextWindow.WriteLine("ИГРАЕМ В ГОРОДА")
TextWindow.WriteLine("")
TextWindow.Show()

'считываем файл:
fileName= "города3.txt"
Stack.PushValue(_stack,fileName)
readFile()
```

В самой процедуре **readFile** мы сначала извлекаем название из стека, а затем действуем, как обычно:

```
'ПОДПРОГРАММА ДЛЯ СЧИТЫВАНИЯ СЛОВ ИЗ ТЕКСТОВОГО ФАЙЛА
Sub readFile
  'имя файла со списком:
  txt= Stack.PopValue(_stack)
  txt= File.ReadContents(txt)
  . . .
EndSub
```

Не забываем *перемешать* слова в списке, чтобы компьютер играл всякий раз по-другому:

```
'Начало игры
startGame:
randomize()
'ни одно слово не стоит в цепочке:
for i=1 to nWords
```

```
  flgChain[i] = "False"
EndFor
'длина цепочки равна нулю:
lenChain=0
```

Как мы и договаривались, первый ход делает игрок. Для этого он просто вводит с клавиатуры название любого города:

```
'ХОД ИГРОКА
hodIgroka:
TextWindow.ForegroundColor="Yellow
TextWindow.WriteLine("")
TextWindow.Write("Напишите город > ")
gorod= Text.ConvertToUpperCase(TextWindow.Read())
```

При этом ему не нужно заботиться о регистре букв, потому что введённое им слово мы тут же переводим в *верхний* регистр, ведь все слова в нашем списке записаны ПРОПИСНЫМИ буквами.

Естественно, мы не можем слепо доверять игроку, поэтому мы обязательно должны проверить, есть ли названный город в нашем списке (вы можете дополнить список новыми городами, если вам этого недостаточно). Если он отсутствует, то игроку придётся повторить свой ход.

Если название города имеется в списке, но он уже назван (такая ситуация очень часто возникает в этой игре), то игрок должен назвать другой город.

И наконец, первая буква названия города должна совпадать с последней буквой предыдущего города – таковы суровые правила игры!

Обратите внимание, что для *первого* города сделано исключение в этой проверке:

```
'проверка:
flg="False"
For i=1 to nWords
  If gorod = spisok[i] Then
    flg="True"
```

```
    Goto test_exit
  EndIf
endfor
test_exit:
If  (flg="False") then
  TextWindow.ForegroundColor="Red"
  TextWindow.Write("Такого города нет!")
  goto hodIgroka
ElseIf (flgChain[i]= "True") then
  TextWindow.ForegroundColor="Red"
  TextWindow.Write("Такой город уже назван!")
  goto hodIgroka
ElseIf (lenChain > 0) and (Text.GetSubText(spisok[i],1,1) <>
endLetter) Then
  TextWindow.ForegroundColor="Red"
  TextWindow.Write("Неверная первая буква!")
  goto hodIgroka
EndIf
```

Итак, долго ли коротко, но игрок сделает правильный ход, что мы и должны зафиксировать в программе. Город игрока мы отмечаем в списке, чтобы его нельзя было назвать повторно. Увеличиваем длину цепочки и определяем последнюю букву слова *endLetter* для проверки следующего хода.

Обычно в игре действуют так. Если последняя буква слова *Ы* или *Ь* (наверное, можно добавить к ним *Й*), с которых русские слова не начинаются, то последней считают *предпоследнюю* букву. Для игры это хорошо, но при составлении чайнвордов такие послабления недопустимы.

После соблюдения всех необходимых процедур слово игрока печатается на экране ПРОПИСНЫМИ буквами:

```
'удачный ход:
flgChain[i]= "True"
lenChain= lenChain + 1
len= Text.GetLength(spisok[i])
endLetter= Text.GetSubText(spisok[i],len,1)
'если слово заканчивается на "плохие" буквы,
'то берем предыдущую:
While (len >= 2) And (endLetter= "Ы" Or endLetter= "Ь")
```

```
  len=len-1
  endLetter= Text.GetSubText(spisok[i],len,1)
endwhile

TextWindow.WriteLine("")
TextWindow.ForegroundColor="Yellow"
TextWindow.Write("Вы назвали город #" + lenChain + " ")
TextWindow.ForegroundColor="Green"
TextWindow.WriteLine(gorod)
```

Ответный ход за *компьютером*. Ему, конечно, ход сделать проще, поскольку весь список слов в его полном распоряжении.

Он просматривает список от начала до конца и проверяет каждое слово в процедуре *test*. Здесь мы опять передаём номер слова в списке через стек.

В начале процедуры переданное значение мы присваиваем «локальной» переменной *test_n*. Сами проверки ничем не отличаются от проверки слова игрока, за маленьким исключением: компьютер всегда выбирает слова из списка, поэтому одна проверка была бы лишней. Значение переменной *resolution* (также «локальной» для этой процедуры) возвращаем через стек:

```
'Проверка: можно ли поставить очередное
'слово в цепочку
Sub test
  'Номер слова в списке:
  test_n= Stack.PopValue(_stack)
  resolution= "True"
  'слово уже стоит в цепочке:
  If (flgChain[test_n] = "True") Then
    resolution= "False"
    Goto exit
  EndIf
  'первая буква проверяемого слова должна совпадать
  'с конечной буквой предыдущего слова:
  If (lenChain > 0) and (Text.GetSubText(spisok[test_n],1,1)
<> endLetter) Then
    resolution= "False"
  EndIf
  exit:
```

```
  Stack.PushValue(_stack, resolution)
EndSub
```

Если проверка завершилась неудачно, значит, в списке не осталось подходящих слов. Компьютер сообщает о своем поражении, после чего начинается новая игра:

```
'игра закончена:
endGame:
TextWindow.ForegroundColor="Red"
TextWindow.WriteLine("")
TextWindow.WriteLine("Признаю свое поражение!")
TextWindow.WriteLine("")
'начинаем новую игру:
Goto startGame
```

Но, скорее всего, этого не случится, и компьютер сделает свой ход.

```
'ХОД КОМПЬЮТЕРА
'просматриваем список слов с начала:
For i= 1 to nWords
  'подходит ли оно?
  Stack.PushValue(_stack,i)
  test()
  res= Stack.PopValue(_stack)
  'компьютер нашел слово:
  If res="True" then
    Goto c_yes
  EndIf
endfor
'компьютер не нашел продолжения:
Goto endGame
```

Последствия хода компьютера ничем не отличаются от последствий, вызванных ходом игрока:

```
'компьютер выполняет ход:
c_yes:
gorod= spisok[i]
flgChain[i]= "True"
lenChain= lenChain + 1
```

```
len= Text.GetLength(spisok[i])
endLetter= Text.GetSubText(spisok[i],len,1)
'если слово заканчивается на "плохие" буквы,
'то берем предыдущую:
While (len > 2) And (endLetter= "Ы" Or endLetter= "Ь")
  len=len-1
  endLetter= Text.GetSubText(spisok[i],len,1)
endwhile
TextWindow.Write("")
TextWindow.ForegroundColor="Yellow"
TextWindow.Write("Я называю город  #" + lenChain + " ")
TextWindow.ForegroundColor="Green"
TextWindow.WriteLine(gorod)
```

Вот только теперь ход следует передать игроку:

```
'теперь ход игрока:
goto hodIgroka
```

А теперь – за игру (Рис. 42.4)! Надо признать, компьютер играет великолепно, но есть *уловки*, которые приведут его к поражению. Подумайте и отыщите их. Кстати говоря, они вам пригодятся и при игре с настоящими соперниками, а не только с компьютерными.

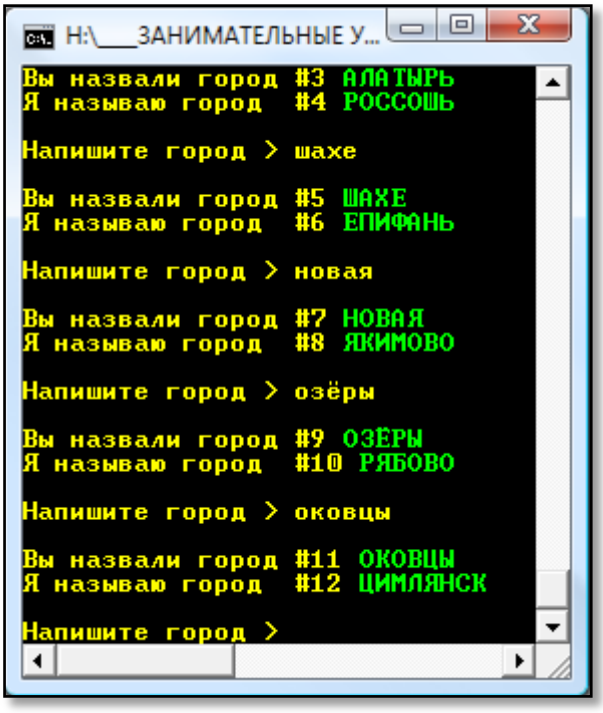


Рис. 42.4. Партия в самом разгаре.

Исходный код программы находится в папке **Города2**.

1. Вместо названий городов России можно взять европейские города или вообще любые города мира. Более того, совсем не обязательно играть только в города, это могут быть названия фруктов, овощей, птиц, названия химических элементов, иностранные слова – любые *тематические* наборы слов. Такие игры очень помогают обогатить свой словарный запас и могут пригодиться на любых уроках.

2. Компьютер признаёт свое поражение, если не может найти продолжения цепочки. Предусмотрите в программе возможность и для игрока признать своё поражение.

3. Перепишите программу так, чтобы игра велась в *графическом окне*, что и более красиво, и более удобно.

4. Поместите в *графическом окне* многострочное *текстовое поле*, в которое можно вывести список всех слов, чтобы игрок мог выбирать следующее слово из списка.

5. Дайте игроку возможность выбирать, будет ли начинать игру он или компьютер.

6. Некоторые действия игрока и компьютера практически совпадают, поэтому выделите их в отдельные процедуры.

7. Замените компьютер вторым игроком, чтобы играть в города с друзьями-товарищами.

Урок 43. Магические квадраты

Магический квадрат – это таблица, которая состоит из равного числа строк и столбцов. Во всех её клетках записано по одному числу. Если обозначить буквой n число строк (столбцов) в таблице (это число называется *порядком* магического квадрата), то в обычном магическом квадрате записаны последовательные числа от *1* до n^2. Например, в самом простом магическом квадрате порядка 3 (то есть размером 3 x 3 клетки) мы найдём все числа от 1 до $3^2 = 9$ (Рис. 43.1).

Магические квадраты иначе называют *волшебными*.

Таким образом, мы всегда знаем, какие числа следует записать в таблицу. Сделать это в *произвольном* порядке, конечно нетрудно, но ведь квадрат не зря называется *магическим*! В нём сумма чисел в каждой строке, столбце и в каждой из двух главных диагоналей должна быть *одинаковой*. Эта *сумма* называется *магической* и обозначается буквой *S*.

Магическая сумма называется также *константой* магического квадрата.

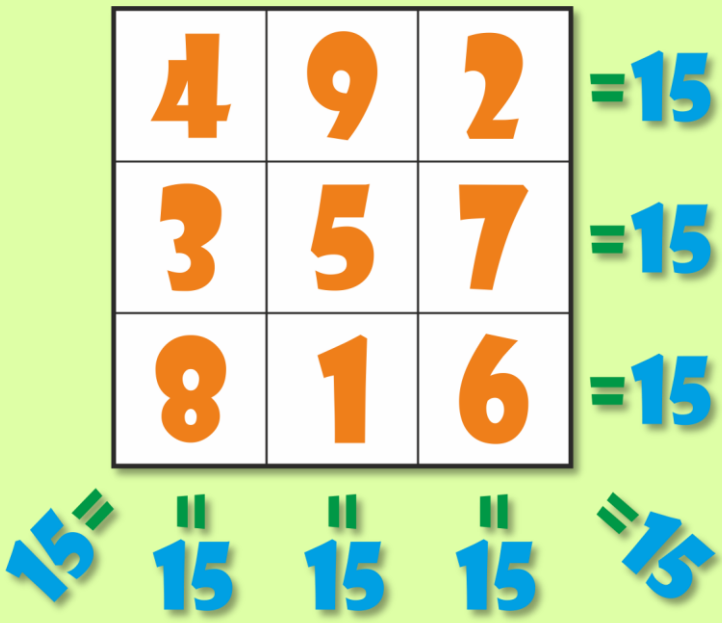


Рис. 43.1. Магический квадрат третьего порядка

Мы легко найдём магическую сумму любого квадрата, если подсчитаем сумму всех чисел в квадрате и разделим её на порядок квадрата. Последовательные натуральные числа, которые находятся в магическом квадрате, образуют *арифметическую прогрессию*, сумму членов которой мы умеем находить.

Первый член прогрессии равен *1*.
Последний член прогрессии равен n^2.
Число членов прогрессии равно n^2.

Сумму всех членов прогрессии можно вычислить по формуле:

$$sum = \frac{1 + n^2}{2} \cdot n^2 = \frac{n^2(1 + n^2)}{2}$$

А *магическую сумму* – по формуле:

$$S = \frac{n(1 + n^2)}{2}$$

История магических квадратов

Составление магических квадратов – одна из первых головоломок в истории человечества. Первый магический квадрат порядка 3 составили китайцы (Рис. 43.2) и назвали его *Ло-шу*. Легенда гласит, что люди впервые увидели его на панцире черепахи, которая выползла из реки Ло. Случилось это задолго до нашей эры, хотя дотошные историки утверждают, что квадрат Ло-шу появился не раньше четвёртого века до нашей эры.

Китайцы придавали математическим особенностям квадрата *Ло-шу* мистические свойства и считали его символом, объединяющим предметы, людей и вселенную. Чётные числа представляли женскую сущность Инь, а нечётные – мужскую – Ян (Рис. 43.3).

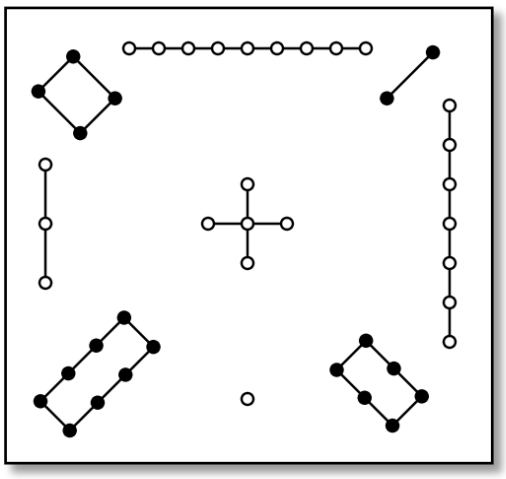


Рис. 43.2. Магический квадрат *Ло-шу* на панцире черепахи

Рис. 43.3. Знаменитый символ *Инь-Ян*

В центре *Ло-шу* находится пятёрка, которая символизирует Землю (Рис. 43.4). Вокруг неё располагаются элементы (стихии). Металл, представленный числами 4 и 9, вода – 1 и 6, огонь – 2 и 7, дерево 3 и 8. Легко заметить, что каждый элемент содержит мужское и женское начала *Инь* и *Ян*.

Рис. 43.4. Элементы природы в квадрате *Ло-шу*

Из Китая магические квадраты попали в Индию, а затем - через арабские страны – в Европу. В эпоху Ренессанса немецкий математик Генрих Корнелиус Агриппа (1486-1536) построил магические квадраты от третьего до девятого порядка и дал им астрономическое толкование.

Наименьшим магическим квадратом считается *Ло-шу*, поскольку квадрат, состоящий всего из одной клетки, трудно назвать магическим, а магических квадратов второго порядка вообще не существует, в чём вы легко сможете убедиться сами, если попробуете их составить.

Они представляли собой семь известных тогдашней науке «планет»: Сатурн, Юпитер, Марс, Солнце, Венеру, Меркурий и Луну.

Но создание первого «европейского» магического квадрата приписывают немецкому художнику, уроженцу города Нюрнберга, современнику Леонардо да Винчи Альбрехту Дюреру. На знаменитой гравюре *Меланхолия* (Рис. 43.5) в правом верхнем углу он поместил магический квадрат четвёртого порядка. Причём Дюрер составил его так ловко, что два средних числа в нижней строке образовали год создания произведения - **1514** (Рис. 43.6).

После *Ло-шу* это самый известный магический квадрат в мире!

Конечно, Дюрер украсил свою гравюру магическим квадратом не только для того, чтобы указать год. Дело в том, что в то время квадраты четвёртого порядка считались хорошим терапевтическим средством, и астрологи «прописывали» их для амулетов против меланхолии.

Необычным свойствам магических квадратов люди с давних времен находили применение в мистике и религии. Вырезанные на дереве, камне или металле магические квадраты служили амулетами (в этом качестве они до сих пор находят применение в восточных странах). Ещё в 16-17 веках люди верили, что выгравированный на серебряной пластине магический квадрат может защитить их от чумы.

Арабские астрологи более девяти веков назад использовали магические квадраты при составлении гороскопов.

Рис. 43.5. Дюрер. *Меланхолия*

Рис. 43.6. Магический квадрат крупным планом

А сколько всего магических квадратов?

Мало ли в Бразилии Педров? -
И не сосчитаешь!

Фраза тётушки Чарли
из комедии *Здравствуйте, я ваша тётя*

Мы уже знаем, что магические квадраты первого и третьего порядков существуют в единственном числе, а квадратов второго порядка вообще нет.

Мы считаем только *уникальные* квадраты, которые нельзя получить из других с помощью поворотов и отражений.

Французский математик Бернар де Бесси (Bernard Frénicle de Bessy, 1605 – 1675) подсчитал, что магических квадратов четвёртого порядка существует 880 штук.

Долгое время учёные оценивали число магических квадратов пятого порядка в 13 миллионов, пока в 1973 году американский программист Ричард Шрёппель (Richard Schroeppel) с помощью компьютера не нашёл их точное число - 275 305 224.

Сколько существует квадратов шестого порядка, до сих пор неизвестно, но их примерно 1.77×10^{19}. Число огромное, поэтому нет никаких надежд пересчитать их с помощью полного перебора, а вот формулы для подсчёта магических квадратов никто придумать не смог.

Как составить магический квадрат?

Существует очень много способов построения магических квадратов. Проще всего построить магические квадраты *нечётного порядка*. Мы воспользуемся очень простым методом, который предложил французский учёный XVII века А. де ла Лубер (De La Loubère). Он основан на *пяти* правилах, действие которых мы рассмотрим на самом простом магическом квадрате 3 x 3.

Правило 1. Поставить 1 в среднюю колонку первой строки (Рис. 43.7).

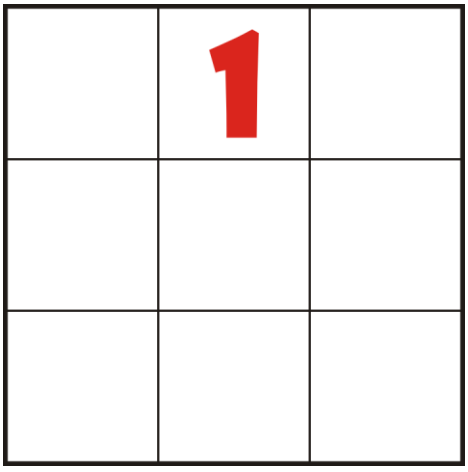


Рис. 43.7. *Первое* число

Правило 2. Следующее число поставить, если возможно в клетку, соседнюю с текущей по диагонали правее и выше (Рис. 43.8).

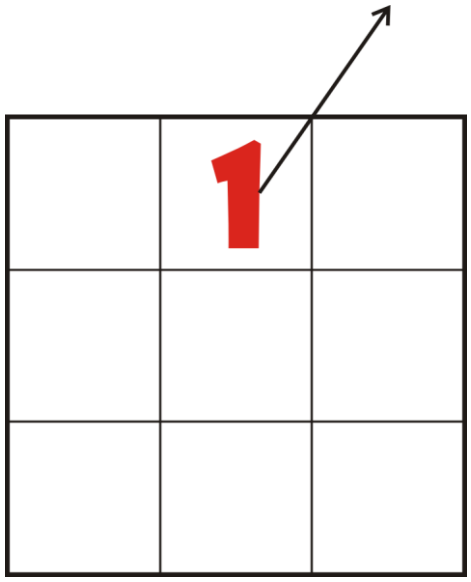


Рис. 43.8. Пытаемся поставить второе число

Правило 3. Если новая клетка выходит за пределы квадрата *сверху*, то число записывается в самую нижнюю строку и в следующую колонку (Рис. 43.9).

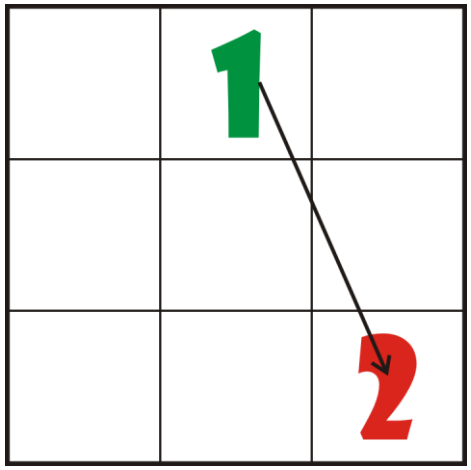


Рис. 43.9. Ставим *второе* число

Правило 4. Если клетка выходит за пределы квадрата *справа*, то число записывается в самую первую колонку и в предыдущую строку (Рис. 43.10).

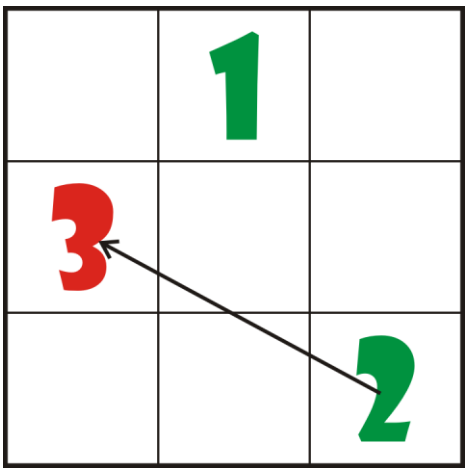


Рис. 43.10. Ставим *третье* число

Правило 5. Если в клетке уже стоит число, то очередное число записывается под текущей клеткой (Рис. 43.11).

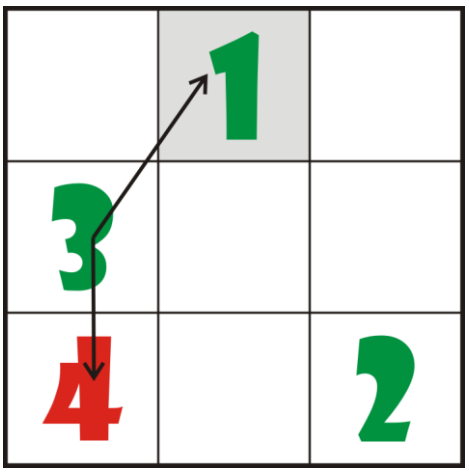


Рис. 43.11. Ставим *четвёртое* число

Далее переходим к *Правилу 2* (Рис. 43.12).

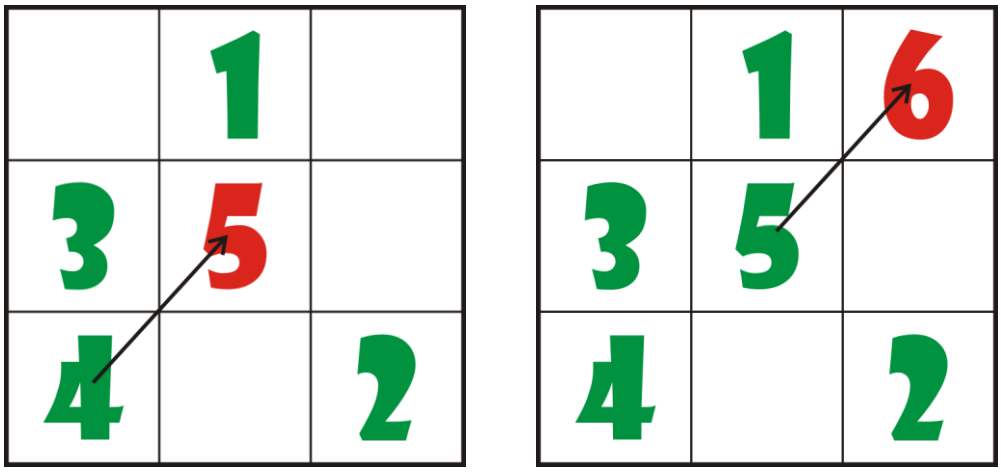


Рис. 43.12. Ставим *пятое* и *шестое* число

Снова выполняем *Правила 3, 4, 5*, пока не составим весь квадрат (Рис. 43.13).

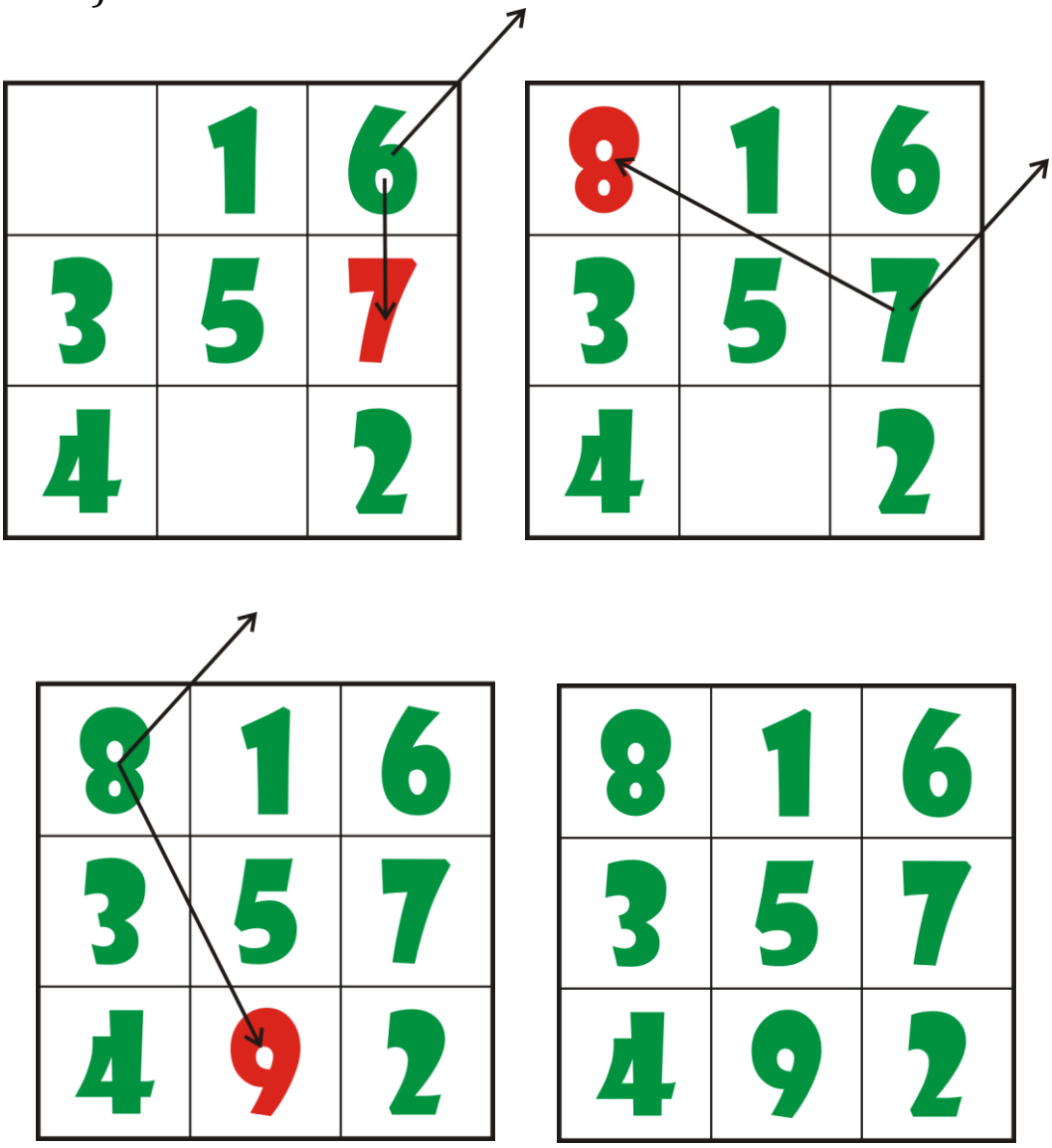


Рис. 43.13. Заполняем квадрат следующими числами

Не правда ли, правила очень простые и понятные, но всё равно довольно утомительно расставлять даже 9 чисел. Однако, зная алгоритм построения магических квадратов, мы легко можем поручить компьютеру всю рутинную работу, оставив себе только творческую, то есть написание программы.

Набор *переменных* для программы **Магические квадраты** совершенно очевиден:

```
' ПРОГРАММА ДЛЯ ГЕНЕРИРОВАНИЯ
' НЕЧЕТНЫХ МАГИЧЕСКИХ КВАДРАТОВ
```

```
'ПО МЕТОДУ ДЕ ЛА ЛУБЕРА

'var
n=0              'порядок квадрата
mq[0][0]= 0      'магический квадрат
number=0         'текущее число для записи в квадрат
col=0            'текущая колонка
row=0            'текущая строка
```

Метод де ла Лубера годится для нечётных квадратов *любого* размера, поэтому мы можем предоставить пользователю самостоятельно выбрать порядок квадрата. Однако, дав пользователю свободу, мы должны проверить результаты его деятельности («доверяй, но проверяй!»).

Физические размеры квадратов большого порядка превышают возможности *текстового окна* по ширине, поэтому подумайте, как организовать вывод готовых квадратов в *файл* или в *графическое окно*.

Проверки очень простые, но необходимые:

```
'========================================
'             ОСНОВНАЯ ПРОГРАММА
'========================================
start:
TextWindow.ForegroundColor="Yellow"
TextWindow.Write("Введите порядок квадрата 3..27 >  ")
n=TextWindow.ReadNumber()

'Проверяем заданное число -->

'Если нуль, работу с программой заканчиваем:
if (n <= 0) then
  Program.End()
EndIf

if (n < 3) Or (n > 27) then
  TextWindow.ForegroundColor="Red"
  TextWindow.WriteLine("Число должно быть от 3 до 27!")
  goto start
EndIf
```

```
if (Math.Remainder(n,2) <> 1) then
  TextWindow.ForegroundColor="Red"
  TextWindow.WriteLine("Число должно нечетным!")
  goto start
EndIf
```

Если пользователь попался сознательный, то в переменной *n* мы получим *порядок* магического квадрата.

Всегда обнуляйте массивы перед их использованием! В некоторых языках программирования это делается автоматически, но и там не следует полагаться на удачу, иначе в массиве окажутся совершенно случайные числа, которые погубят всю программу. Более того, при каждом запуске программы содержание массива будет другим, поэтому иногда вы будете получать правильные результаты, иногда – неправильные. Вот и попробуйте найти ошибку!

```
'Введено правильное число -->

'обнуляем массив:
For i= 1 To n
  For j= 1 To n
    mq[i][j]=0
  EndFor
EndFor
```

Начинаем действовать по правилам де ла Лубера и записываем первое число – единицу – в среднюю клетку первой строки квадрата (или массива, если угодно):

```
'первое число:
number=1

rule1:
'колонка для первого числа - средняя:
col= Math.Floor(n/2) + 1
'строка для первого числа - первая:
row=1
'заносим его в квадрат:
mq[row][col]= number
```

Теперь последовательно пристраиваем по клеткам остальные числа – от двойки до *n* * *n*:

```
'переходим к следующему числу:
nextNumber:
number= number+1
```

Запоминаем на всякий случай координаты актуальной клетки

```
tc=col
tr=row
```

и переходим в следующую клетку по диагонали:

```
col= col+1
row= row-1
```

Проверяем выполнение *третьего* правила:

```
rule3:
If (row < 1) Then
  row= n
EndIf
```

А затем *четвёртого*:

```
rule4:
If (col > n) then
  col=1
  Goto rule3
endIf
```

И *пятого*:

```
rule5:
If (mq[row][col] <> 0) Then
  col=tc
  row=tr+1
  Goto rule3
EndIf
```

Как мы узнаем, что в клетке квадрата уже находится число? – Очень просто: мы предусмотрительно записали во все клетки *нули*, а все числа в готовом квадрате *больше нуля*. Значит, по содержимому ячейки массива мы сразу же определим, пустая она или с числом! Обратите внимание, что здесь нам понадобятся те координаты клетки, которые мы запомнили перед поиском клетки для следующего числа.

Попробуйте иначе организовать проверку допустимости перехода в новую клетку!

Рано или поздно мы найдём подходящую клетку для числа и запишем его в соответствующую ячейку массива:

```
'заносим его в квадрат:
mq[row][col]= number
```

Если это число было последним, то программа свои обязанности выполнила, иначе она добровольно переходит к обеспечению клеткой следующего числа:

```
'если выставлены все числа, то
'квадрат составлен:
If (number= n*n) Then
  Goto complete
'иначе переходим к следующему числу:
Else
  Goto nextNumber
EndIf
```

И вот квадрат готов! Вычисляем магическую сумму и распечатывает квадрат в *текстовом окне*:

```
'построение квадрата закончено:
complete:
TextWindow.WriteLine("")
TextWindow.WriteLine("Магическая сумма = " + (n*n*n +n)/2)
writeMQ()

Goto start
```

Напечатать элементы массива очень просто, но важно учесть выравнивание чисел разной «длины», ведь в квадрате могут быть одно-, дву- и трёхзначные числа:

```
'Печатаем готовый квадрат
Sub writeMQ
  TextWindow.WriteLine("")
  TextWindow.ForegroundColor="Green"
  'печатаем магический квадрат:
  For i= 1 To n
    s=""
    For j= 1 To n
      If (n*n > 10) And (mq[i][j] < 10) Then
        s= s+ " "
      EndIf
      If (n*n > 100)And (mq[i][j] < 100) Then
        s= s+ " "
      EndIf
      s= Text.Append(s, mq[i][j]+" ")
    EndFor
    TextWindow.WriteLine(s)
  EndFor
  TextWindow.WriteLine("")
EndSub
```

Запускаем программу – а ведь недаром мы трудились: квадраты получаются быстро и на загляденье (Рис. 43.14).

Исходный код программы находится в папке **Магические квадраты.**

Магические ферзи

Магический квадрат седьмого порядка, составленный по методу де ла Лубера (Рис. 43.15), как показал американец Клиффорд Пиковер, непостижимым образом связан с другой знаменитой задачей – расстановкой ферзей на шахматном поле.

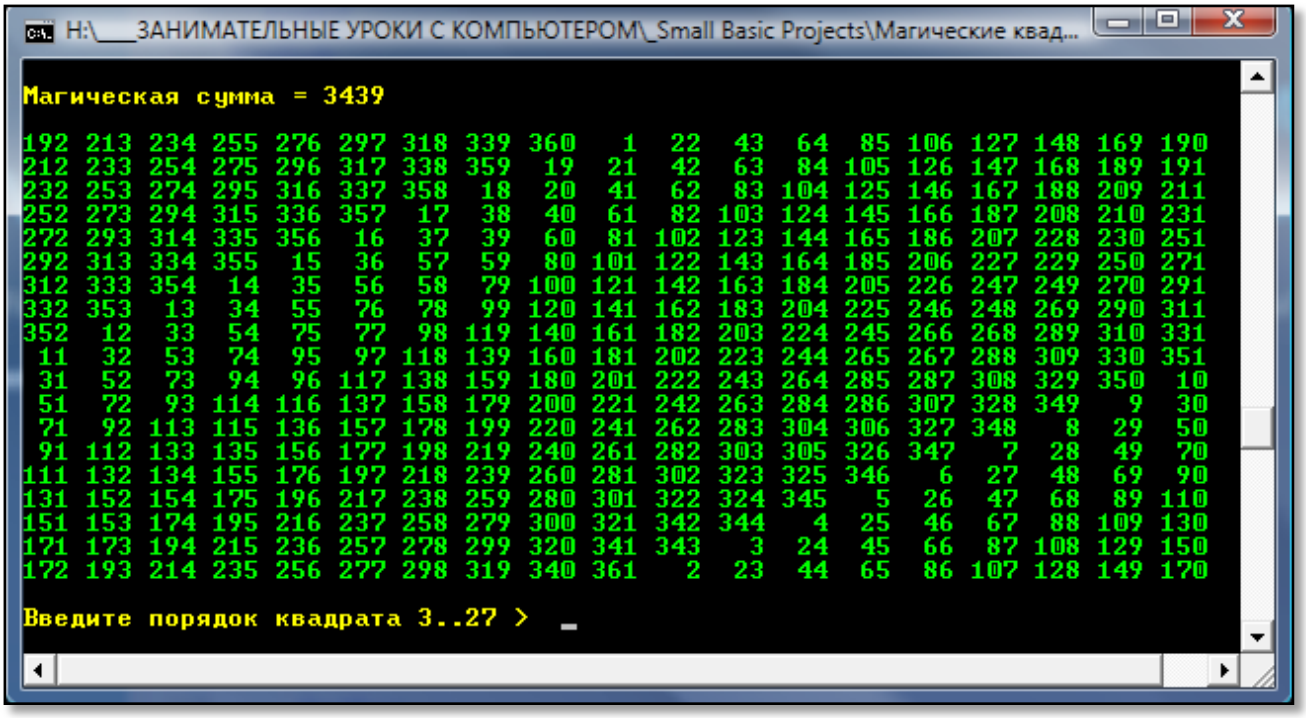


Рис. 43.14. Изрядный квадратище!

Поскольку метод де ла Лубера позволяет строить только нечётные квадраты, а шахматная доска имеет чётный порядок, то придётся взять квадрат, наиболее близкий к шахматной доске, то есть 7 x 7 клеток. Мы его легко построим с помощью нашей программы.

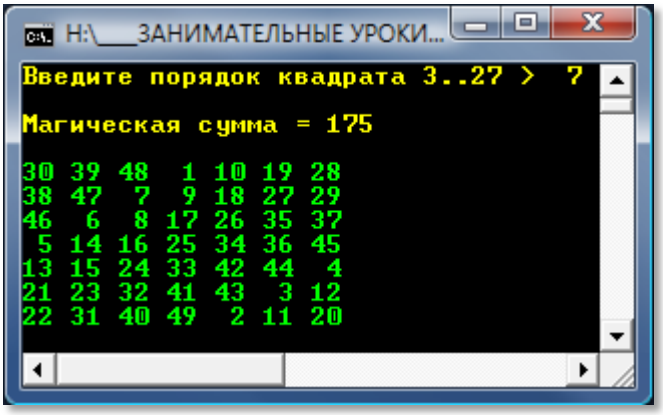


Рис. 43.15. Квадрат де ла Лубера

Следующий шаг: заменяем все числа в магическом квадрате остатками от их деления на семь. При этом нулевой остаток будем считать семёркой (Рис. 43.16).

30	39	48	1	10	19	28
38	47	7	9	18	27	29
46	6	8	17	26	35	37
5	14	16	25	34	36	45
13	15	24	33	42	44	4
21	23	32	41	43	3	12
22	31	40	49	2	11	20

2	4	6	1	3	5	7
3	5	7	2	4	6	1
4	6	1	3	5	7	2
5	7	2	4	6	1	3
6	1	3	5	7	2	4
7	2	4	6	1	3	5
1	3	5	7	2	4	6

Рис. 43.16. Преобразование квадрата де ла Лубера

Новый квадрат в каждой строке и паре нисходящих диагоналей содержит решение для задачи с ферзями. Например, из первой строки вытекает такое решение (Рис. 43.17).

2	4	6	1	3	5	7
3	5	7	2	4	6	1
4	6	1	3	5	7	2
5	7	2	4	6	1	3
6	1	3	5	7	2	4
7	2	4	6	1	3	5
1	3	5	7	2	4	6

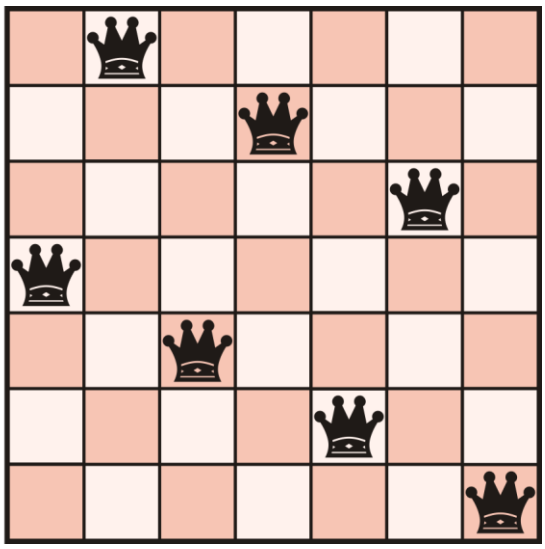


Рис. 43.17. Строки дают решение задачи с ферзями

Одна из главных диагоналей квадрата - нисходящая. Мы можем использовать её для решения задачи о ферзях, но мы рассмотрим другой пример - когда нисходящая диагональ является *побочной*, поэтому должна быть *продолжена* так, чтобы в ней оказалось ровно семь клеток (Рис. 43.18).

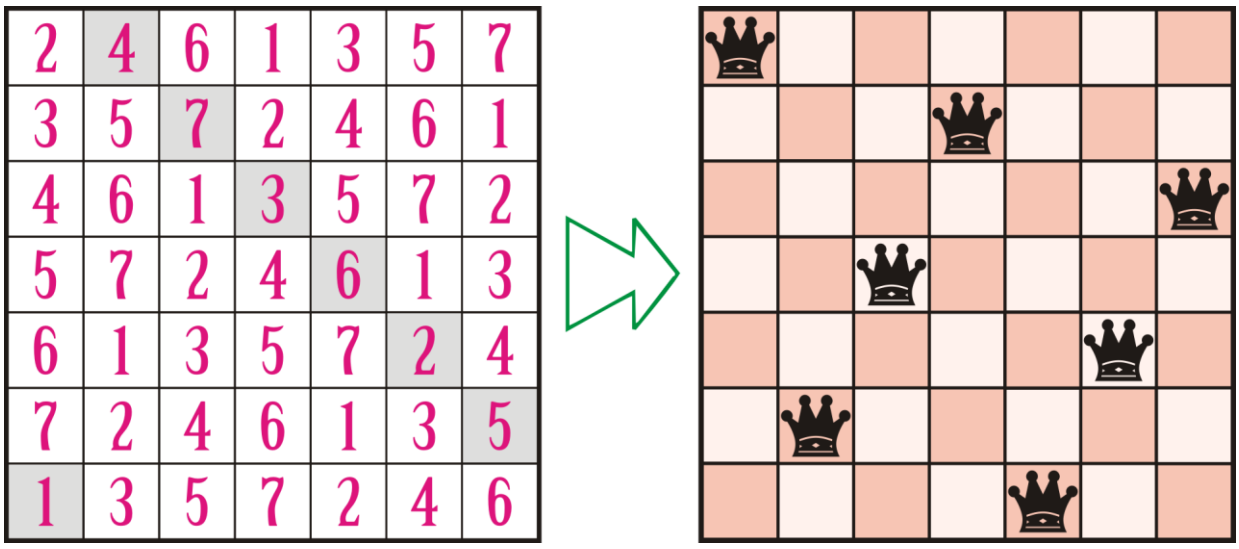


Рис. 43.18. Нисходящие диагонали дают решение задачи с ферзями

Магические квадраты чётного порядка

Теперь давайте составим магический квадрат четвёртого порядка. Один из таких квадратов и изображён на гравюре Дюрера.
Для удобства нарежьте из бумаги 16 квадратиков и напишите на них числа от 1 до 16. Это самая трудная и ответственная часть задания! Дальше будет легче, особенно если вам приходилось складывать картинки из пазлов.
Разложите бумажки квадратиком так, чтобы числа следовали друг за другом по ранжиру, то есть соблюдая субординацию (Рис. 43.19).

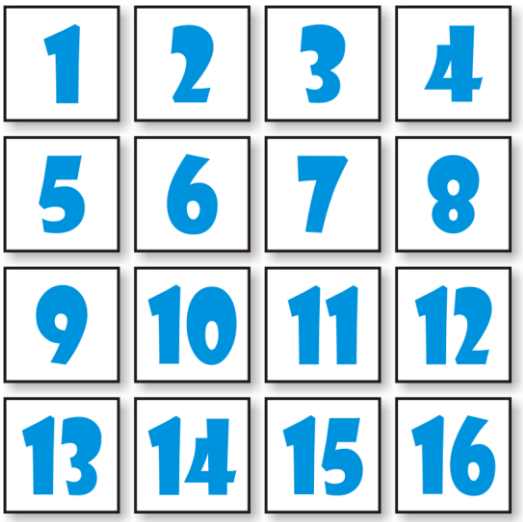


Рис. 43.19. Приняли исходное положение

Шаг 1. Поменяйте местами *вторую* и *третью* строки (Рис. 43.20).

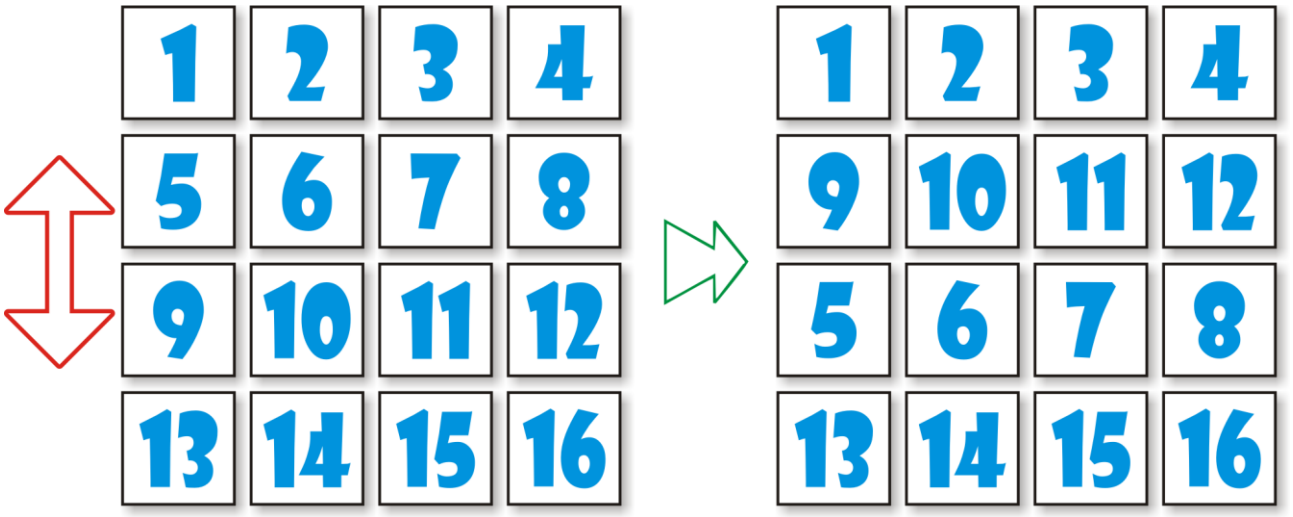


Рис. 43.20. Первый шаг

Шаг 2. Переложите числа во *второй* и *четвёртой* строках в обратном порядке (Рис. 43.21).

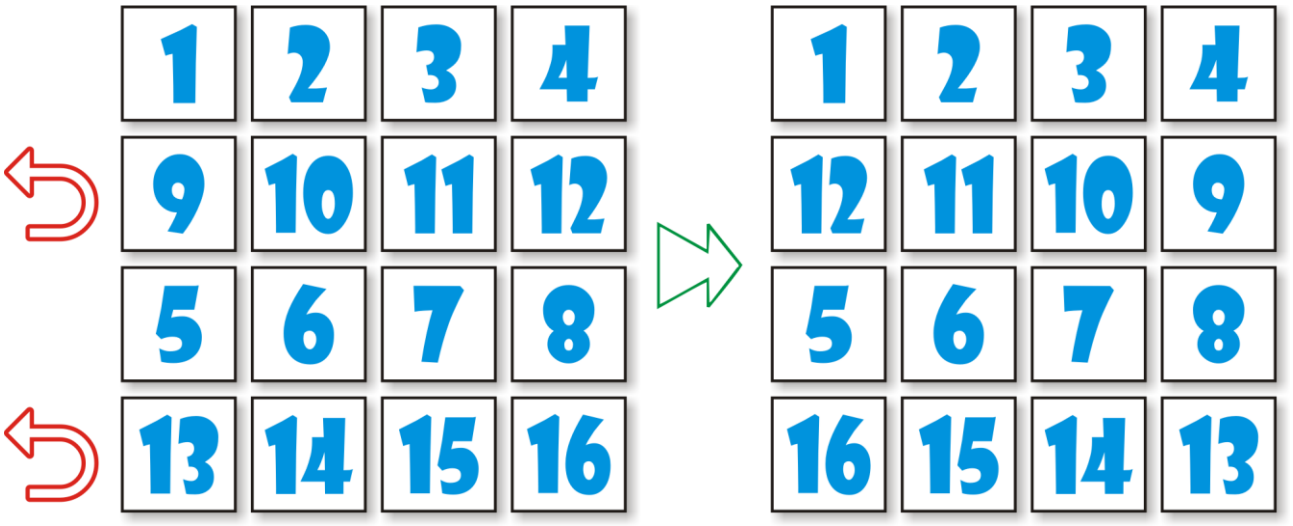


Рис. 43.21. Второй шаг

Шаг 3. Переложите числа во *втором* и *третьем* столбцах в обратном порядке (Рис. 43.22).

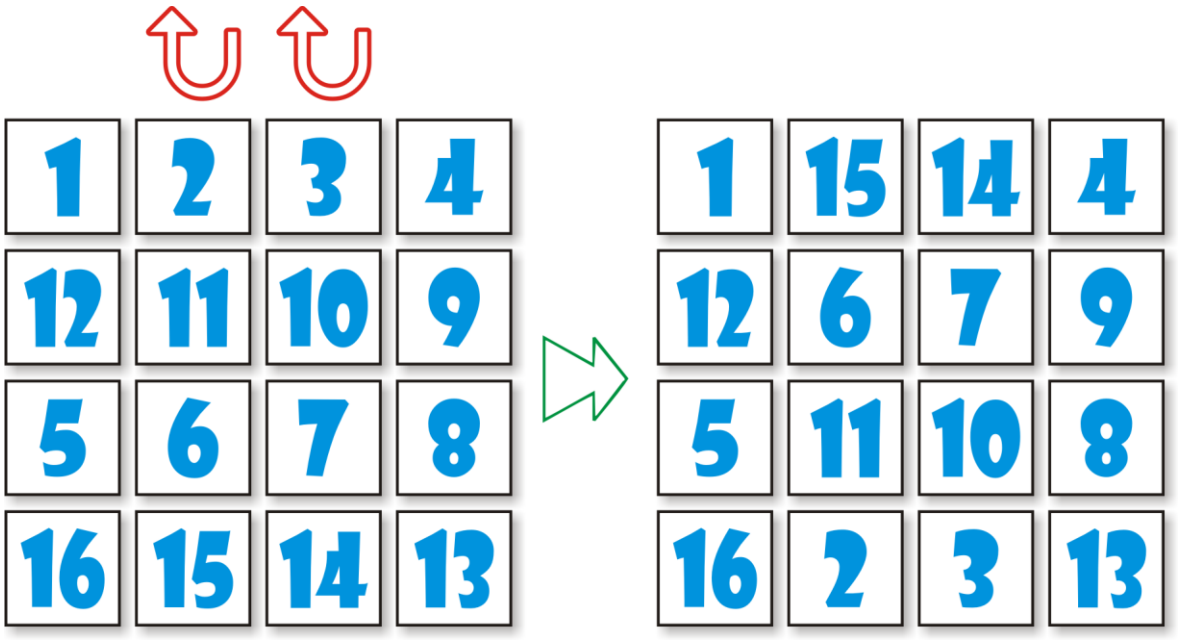


Рис. 43.22. Третий шаг

Шаг 4. Переложите числа в *третьей* и *четвёртой* строках в обратном порядке (Рис. 43.23).

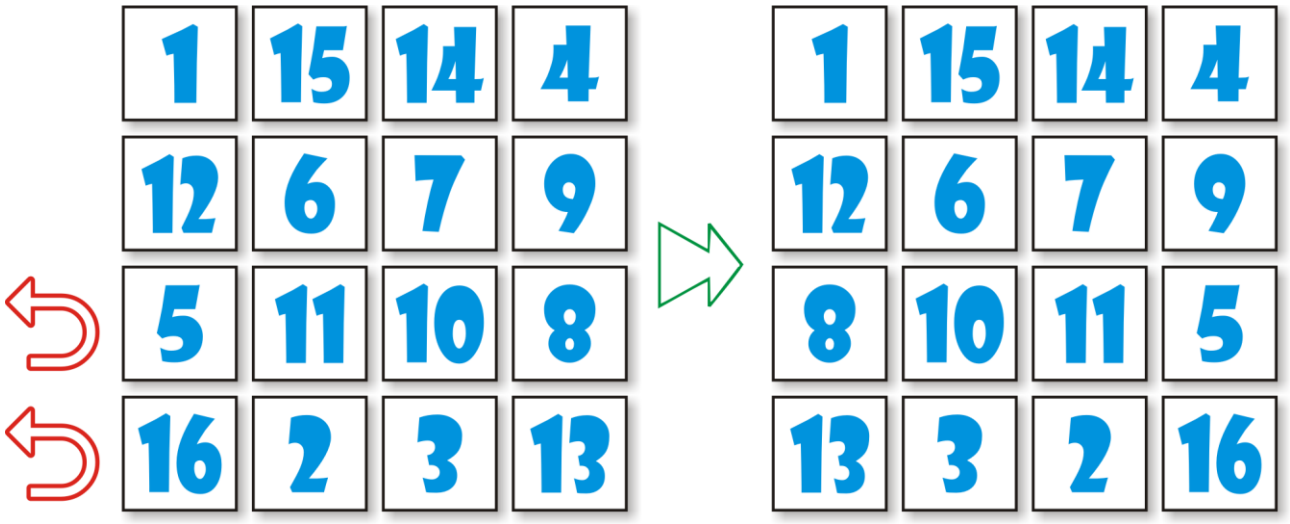


Рис. 43.23. Четвёртый шаг

Магический квадрат готов! И почти, как у Дюрера, - только год стоит не в последней, а в первой строке. Впрочем, вы можете перевернуть квадрат вверх ногами, чтобы дата оказалась внизу. Осталось переставить *первый* и *четвёртый* столбцы – и наш квадрат превращается в «дюреровский» (Рис. 43.24).

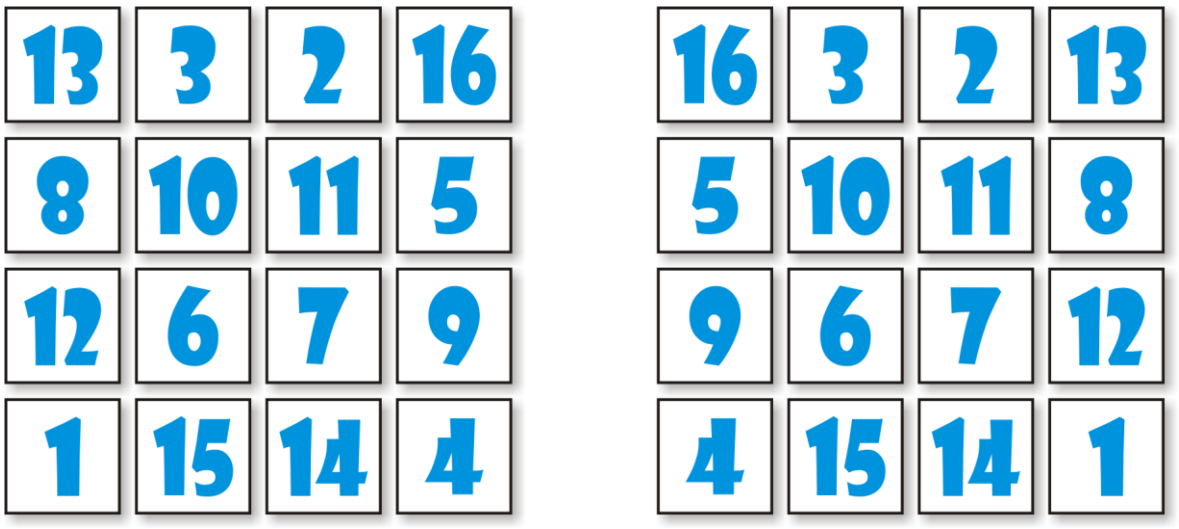


Рис. 43.24. Смастерили Дюреровский квадрат!

Мы не знаем, как именно построил Дюрер свой магический квадрат, но, возможно, и «нашим» способом.

Кстати говоря, в магическом квадрате, который мы построили, больше магических сумм, чем «требуется». Магическая сумма, которая в квадратах четвёртого порядка равна 34, повторяется не только в четырёх строках, четырёх столбцах и двух диагоналях, но и

- в пяти квадратиках 2 x 2 клетки (Рис. 43.25).

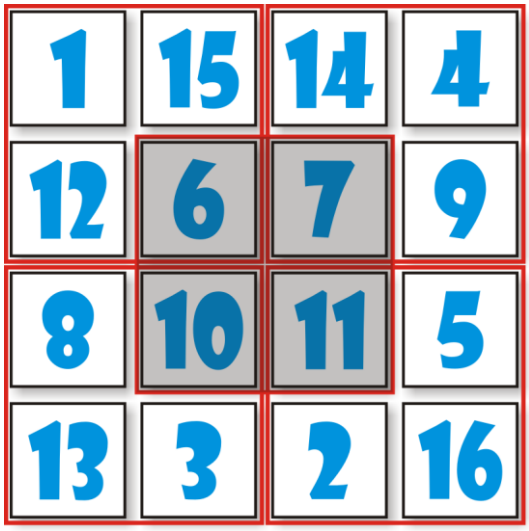


Рис. 43.25. Дополнительные магические квадратики 2 x 2

- в углах четырёх квадратов 3 x 3 клетки (Рис. 43.26).

Рис. 43.26. Дополнительные магические квадраты 3 х 3

- в углах двух прямоугольников 2 х 4 клетки (Рис. 43.27).

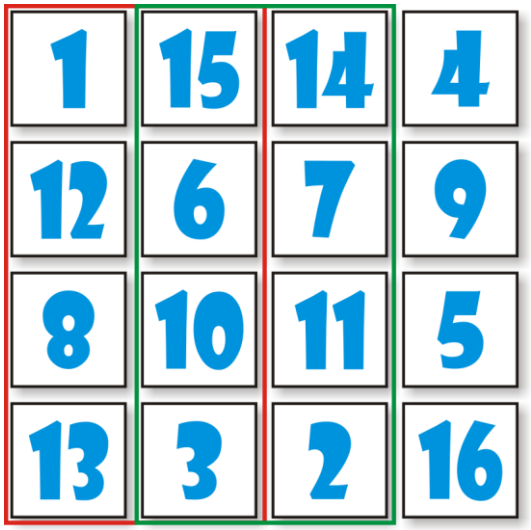


Рис. 43.27. Дополнительные магические прямоугольники 2 х 4

- в углах самого квадрата 4 х 4 клетки (Рис. 43.28).

Итого – *22* раза. Но, оказывается, и это не предел. До предела мы дойдём, если выполним ещё два шага.

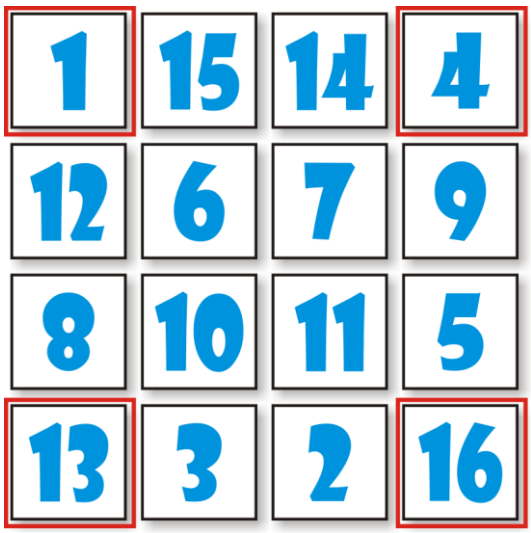


Рис. 43.28. Дополнительные магические прямоугольники 2 x 4

Шаг 5. Временно уберите *третью* и *четвёртую* строки. Переложите *вторую* строку на место *четвёртой*, а затем верните *третью* и *четвёртую* строки на свободные места. То есть *третья* строка займёт в квадрате место *второй*, а *четвёртая* – *третьей* (Рис. 43.29).

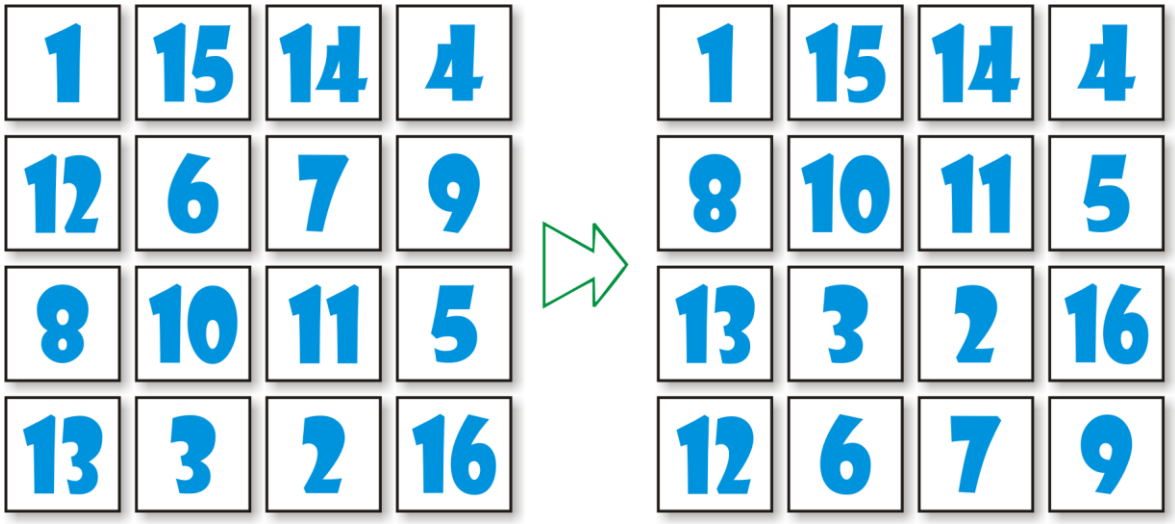


Рис. 43.29. Пятый шаг

Шаг 6. Поменяйте местами *третий* и *четвёртый* столбцы (Рис. 43.30).

Легко заметить, что все числа, кроме первых двух, заняли в квадрате *другие* места, и теперь магическая сумма повторяется уже *30*

раз. Конечно, в этом квадрате сохранились все 22 магические суммы прежнего квадрата, но к ним добавились еще 11:

- в углах девяти (а не пяти) квадратов 2 x 2 клетки;
- в углах шести (а не двух) прямоугольников 2 x 4 клетки.

Найдите самостоятельно *все* магические суммы!

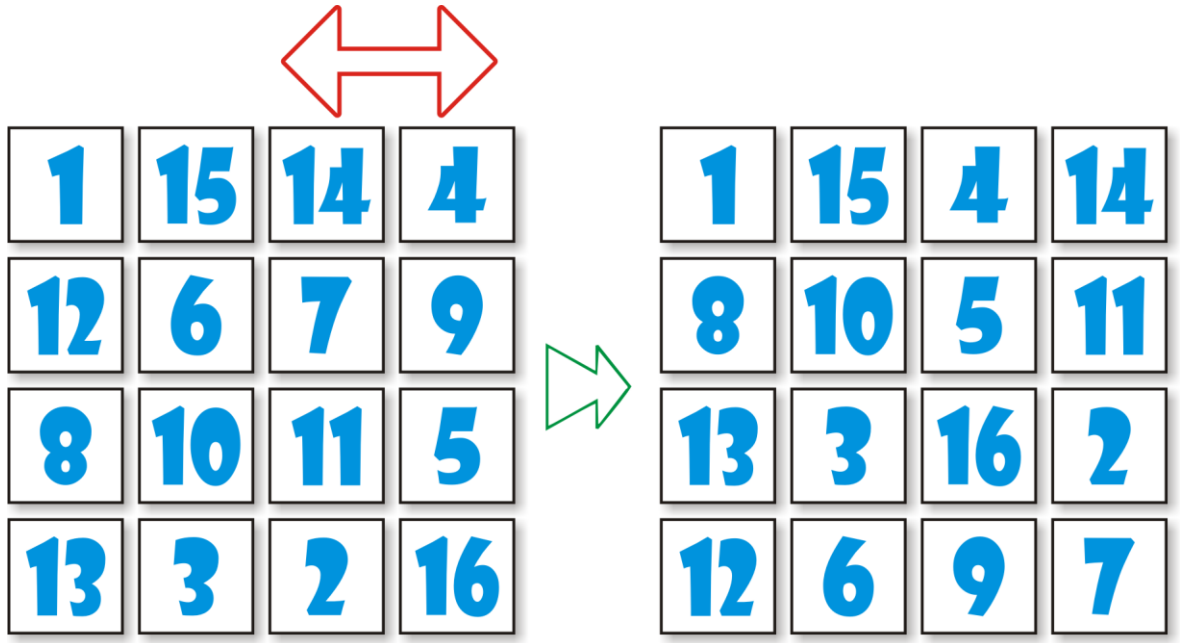


Рис. 43.30. Шестой шаг

1. С помощью программы *Магические квадраты* вы можете составить квадраты любого размера, но только по одному для каждого порядка. Увы, все подобные алгоритмы действуют именно так. Но мы всё-таки можем добавить некоторое разнообразие к нашим квадратам. Например, если прибавить или вычесть одно и то же число (или умножить на одно и то же число), то квадрат останется магическим, а его магическая сумма изменится. Научите программу составлять такие квадраты.

2. Можно также поворачивать квадрат на 90 градусов и отражать его относительно горизонтальной и вертикальной осей, как мы это делали с ферзями. Поскольку эти операции проводятся над уже готовыми квадратами, то это задание совсем несложное.

До новых встреч на занимательных уроках!

СПИСОК ЛИТЕРАТУРЫ

Литература

[PB10]

Рубанцев Валерий

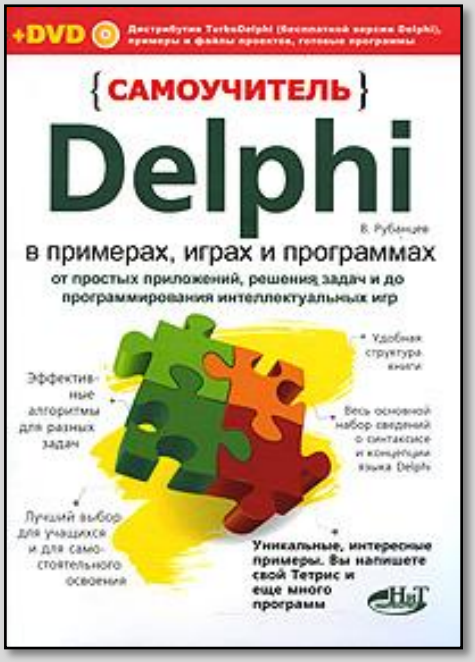
Хитори: круче, чем судоку

Эксмо-Пресс, 2010. – 288 с.

ISBN: 978-5-699-39575-0

Серия: Зарядка для ума

[PB11]



Рубанцев Валерий

Delphi в примерах, играх и программах. От простых приложений, решения задач и до программирования интеллектуальных игр

Наука и Техника, 2011. – 672 с.

ISBN: 978-5-94387-664-6

Серия: Самоучитель

Электронные книги издательства **RVGames**:

Цифр0886е книги

Рубанцев Валерий

2012 - 2014

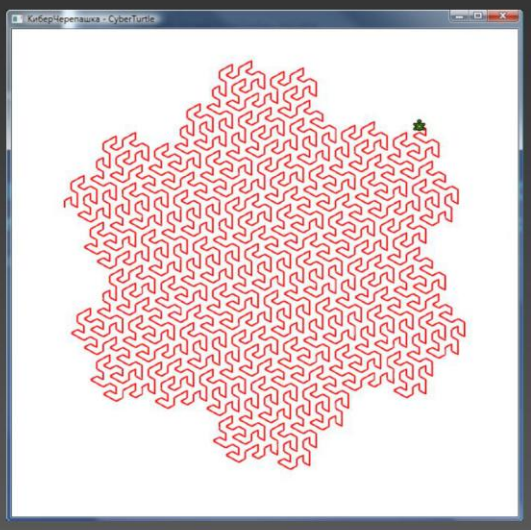
RVGAMES.DE 2014

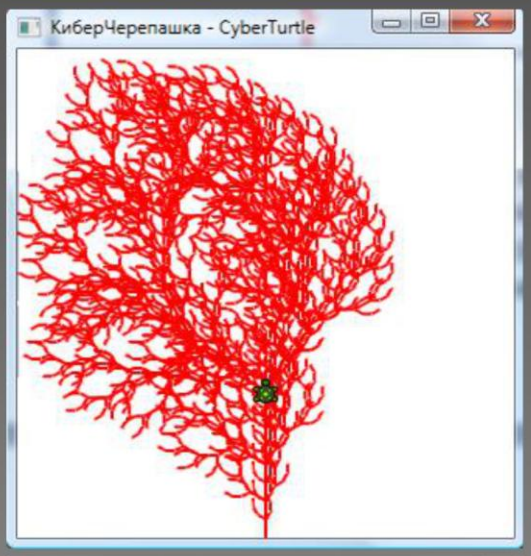
Цифровые книги

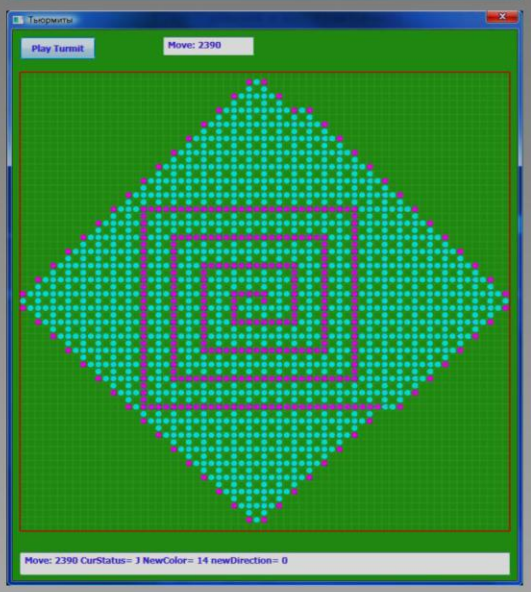
издательства *RVGames*

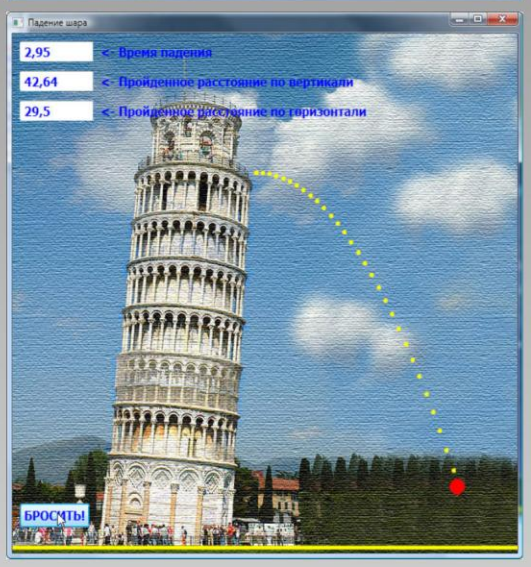
RVGAMES.DE 2014

Официальный сайт издательства: RVGames.de







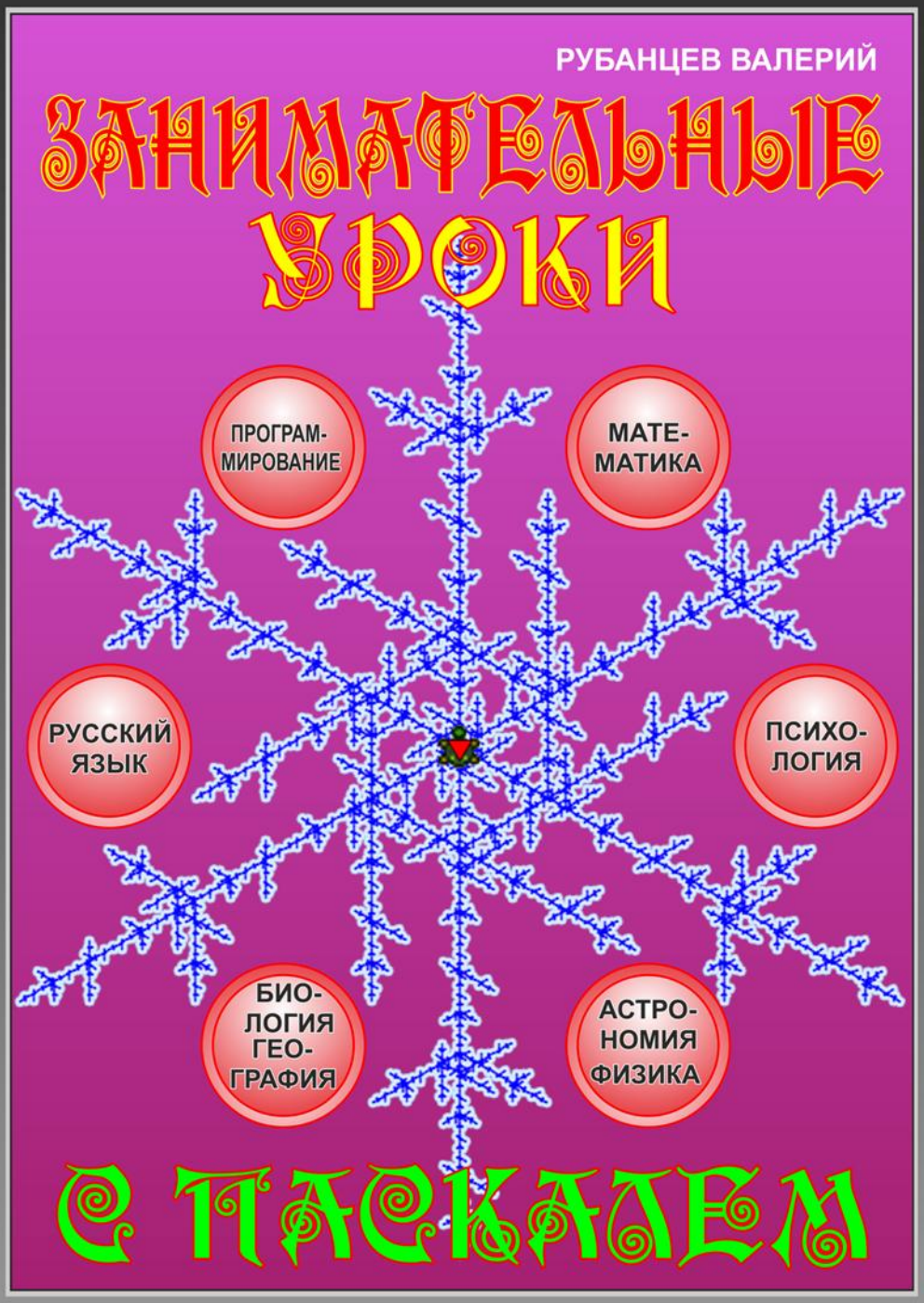


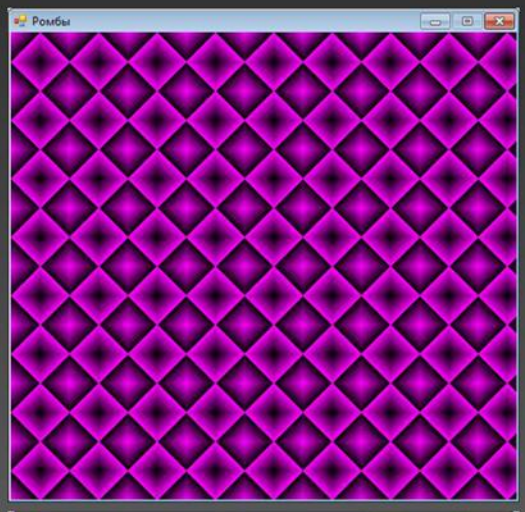
Занимательные уроки с компьютером, или Small Basic для начинающих

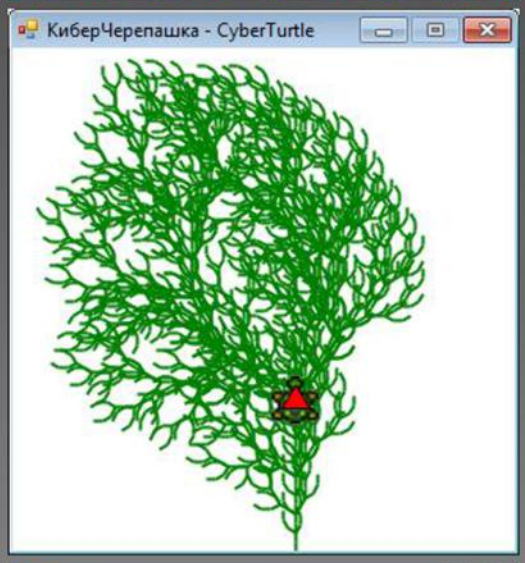
Small Basic - это простой язык для начального знакомства с современным программированием. В книге на многочисленных занимательных примерах из разных школьных дисциплин показана работа в Редакторе кода, основные конструкции языка Small Basic и этапы разработки приложений.

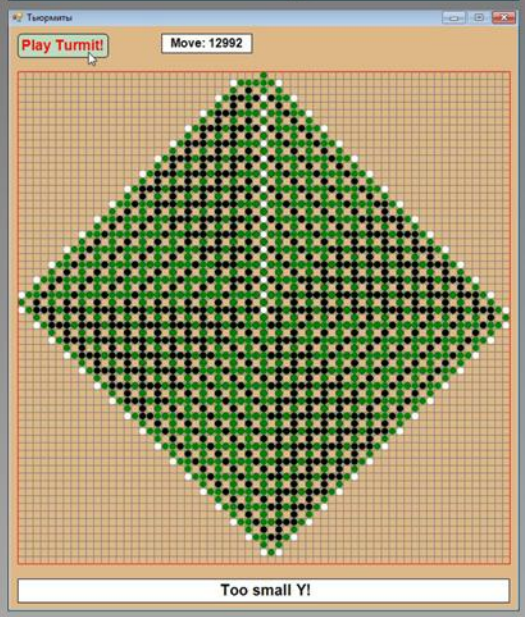
Издательство RVGames, 2013. – 580 с.

RVGAMES.DE 2013









Занимательные уроки с паскалем, или PascalABC.NET для начинающих

PascalABC.NET - простой язык для изучения современного программирования. Он удачно сочетает достоинства классического паскаля с мощью платформы .NET Framework. В книге на многочисленных занимательных примерах показана работа в Редакторе кода, основные конструкции языка паскаль и этапы разработки приложений.

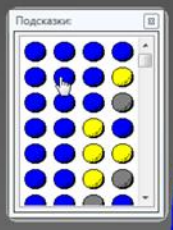
Издательство RVGames, 2013. – 698 с.

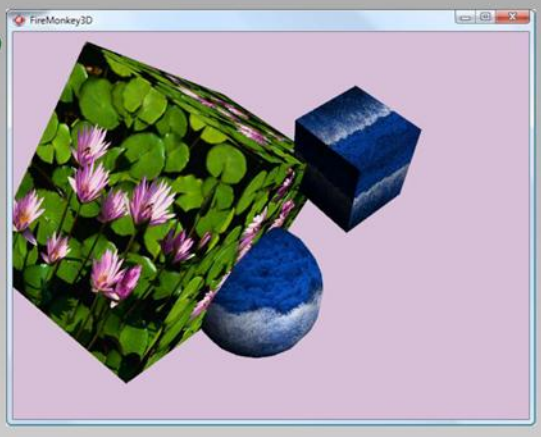
RVGAMES.DE 2013

БОЛЬШОЙ
самоучитель

Delphi XE3

Рубанцев Валерий



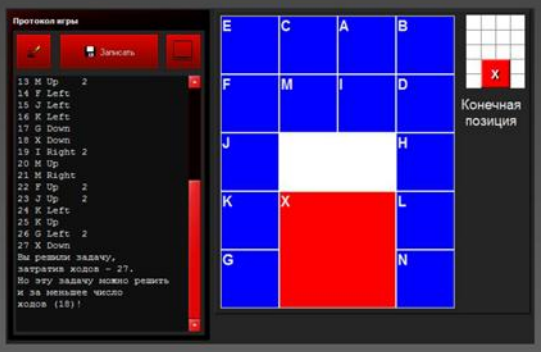


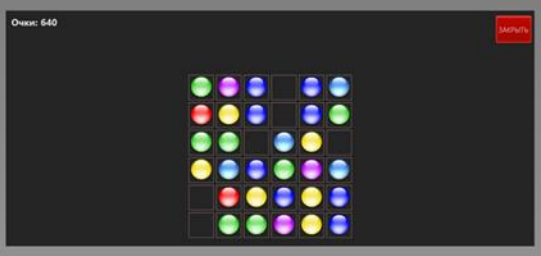
Большой самоучитель Delphi XE3

В книге подробно описаны основные конструкции языка Delphi и работа в Интегрированной среде разработки приложений. Особое внимание уделяется новым возможностям языка, появившимся в последней версии XE3 - платформе FireMonkey FM2, трёхмерной графике, применению стилей к элементам управления.
Теоретический материал поясняется многочисленными практическими примерами.

Издательство RVGames, 2013. – 1278 с.

RVGAMES.DE 2013





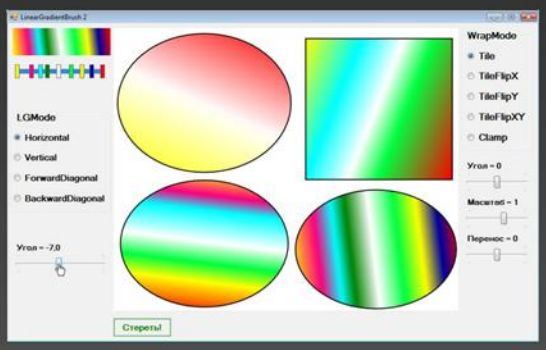
Delphi на пальцах

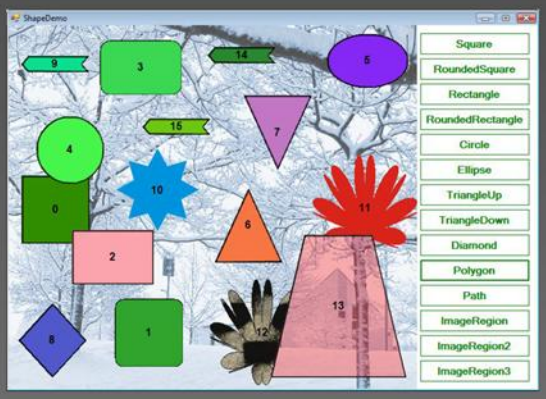
Осваиваем новые технологии Windows 8: Touch и Gesture

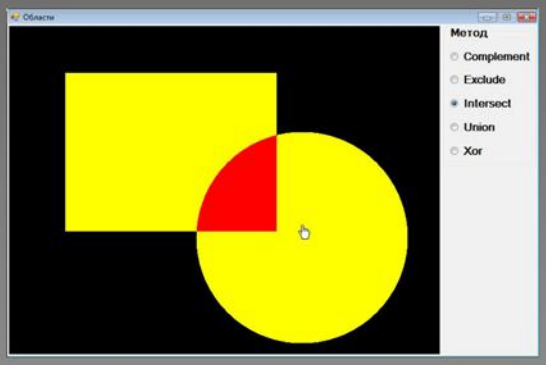
В книге подробно описывается разработка программ в среде Delphi XE3 с использованием новой технологии Windows 8 для компьютеров с сенсорным экраном, основанной на естественном интерфейсе пользователя - касаниях и жестах. Многочисленные практические примеры иллюстрируют и дополняют теоретический материал.

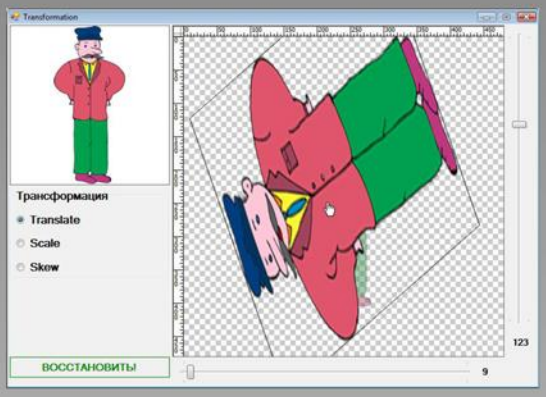
Издательство RVGames, 2013. – 288 с.

RVGAMES.DE 2013





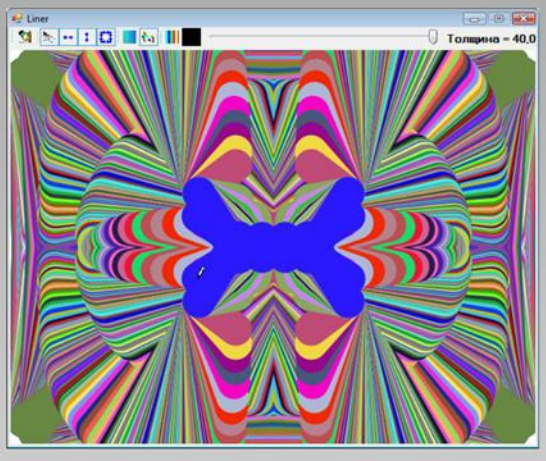




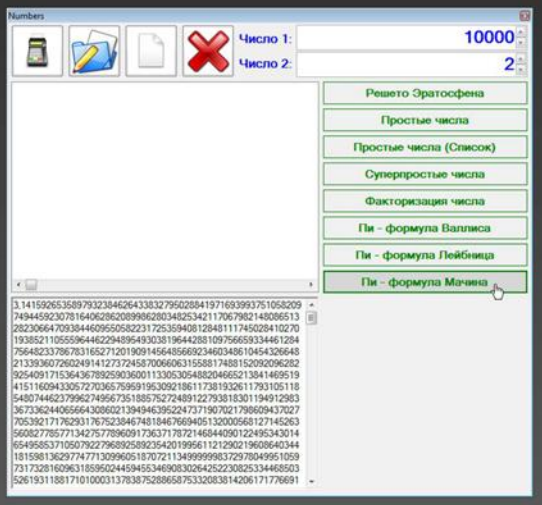
Занимательная графика на Си-шарпе
Подробный самоучитель

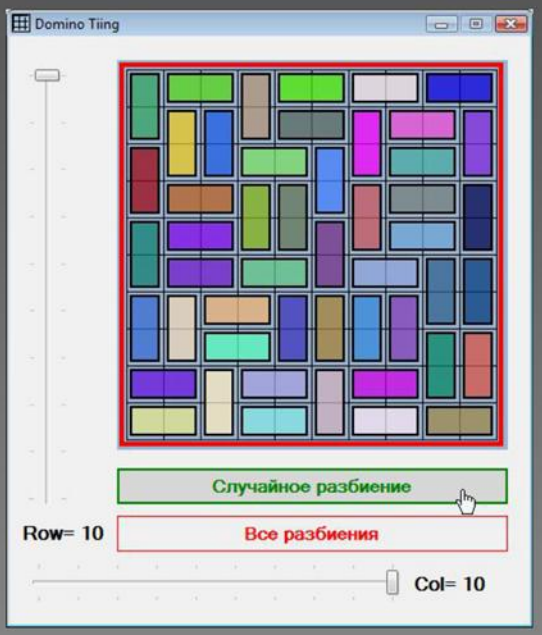
Книга содержит полную информацию обо всех наиболее важных классах и структурах графического интерфейса GDI+. Многочисленные примеры показывают применение графики в проектах на языке C#. В конце книги имеется справочник по интерфейсу GDI+.

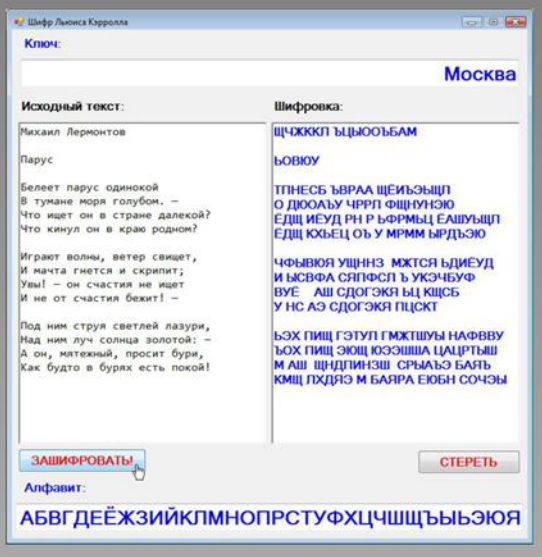
Издательство RVGames, 2013. – 759 с.

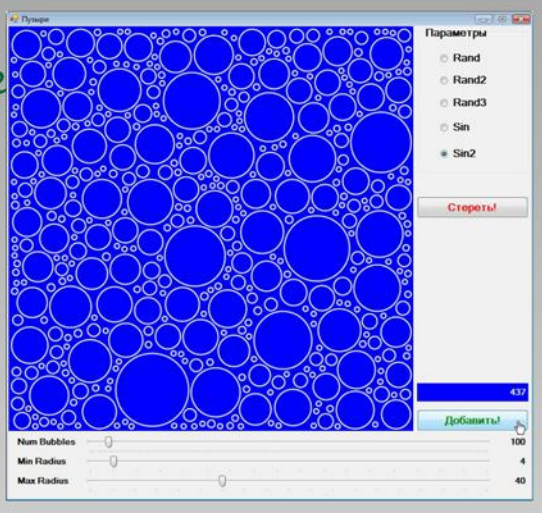


RVGAMES.DE 2013









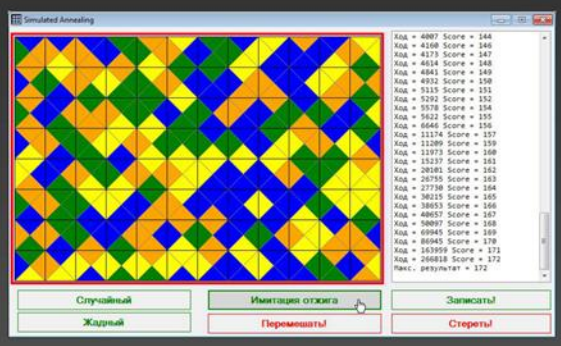
Тотальный тренинг по Си-шарпу

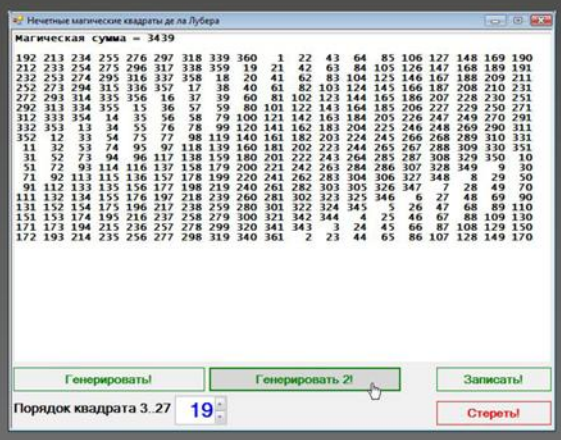
Большой практикум по программированию на языке C#

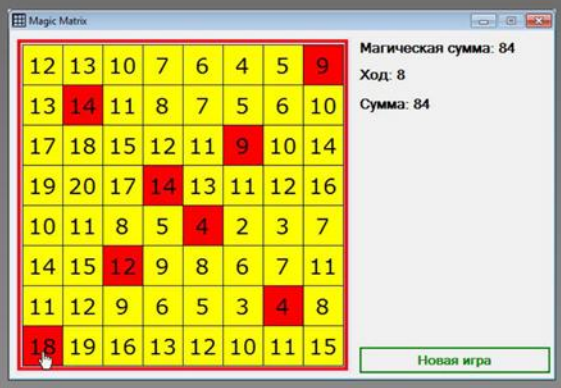
В книге подробно рассматривается решение задач из различных областей знания - русского языка, математики, криптографии, комбинаторики - на языке C#: разработка алгоритмов, выбор методов и структур данных, проектирование графического интерфейса пользователя.

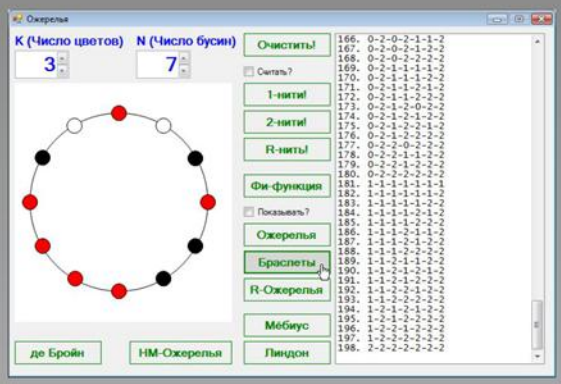
Издательство RVGames, 2013. – 615 с.

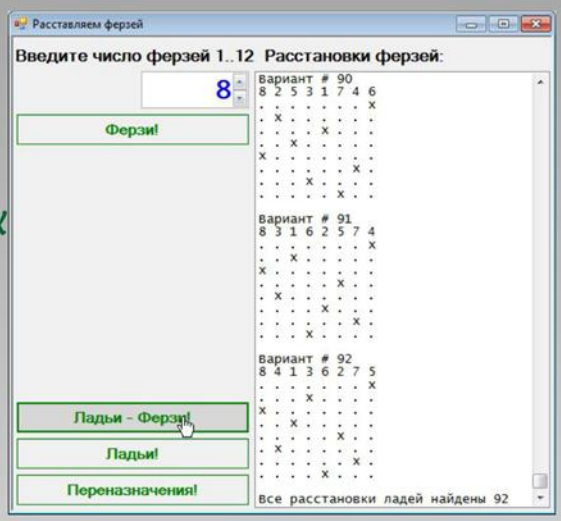
RVGAMES.DE 2013











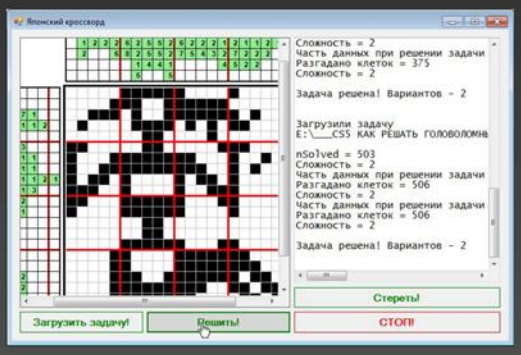
Как решать комбинаторные задачи на компьютере

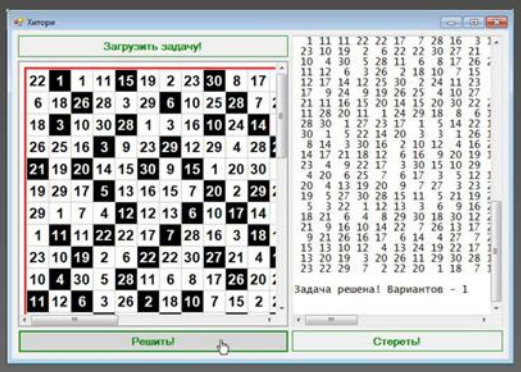
Комбинаторика для программистов на языке C#

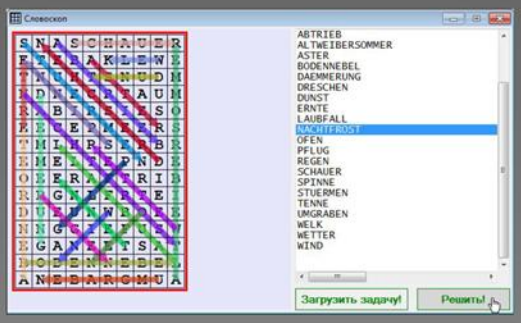
В книге подробно рассматривается решение комбинаторных проблем на языке C#. Наибольшее внимание уделяется историческим задачам: генерированию ожерелий, считалке Иосифа Флавия, созданию магических квадратов - словесных и числовых расстановке ферзей на шахматной доске.

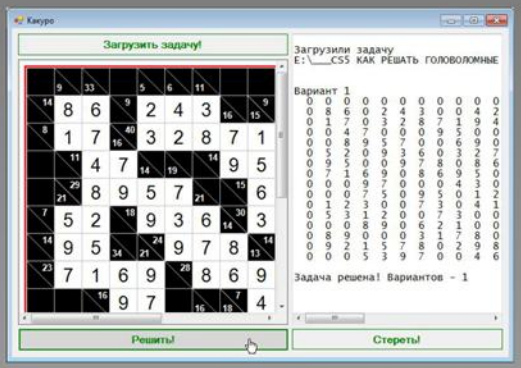
Издательство RVGames, 2013. – 369 с.

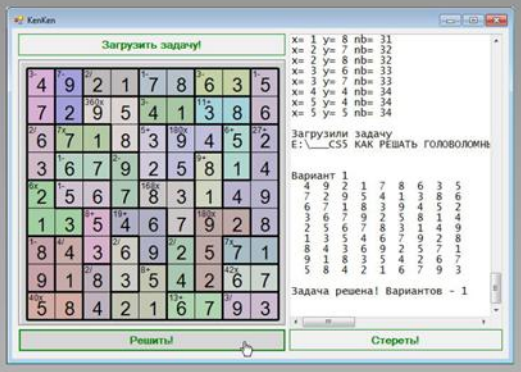
RVGAMES.DE 2013

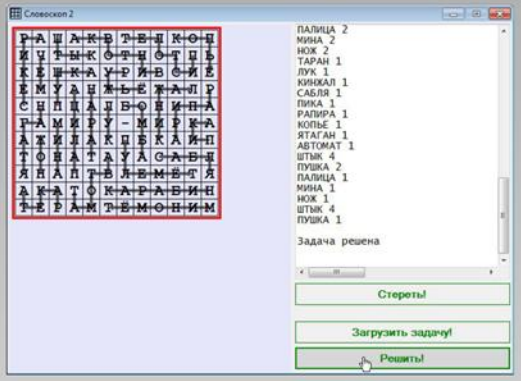












Как решать головоломные задачи на компьютере

Нетривиальные головоломки для программистов на языке C#

В книге подробно рассказывается, как решать современные головоломки на языке C#: судоку, хитори, какуро, японские рисунки, лабиринт-алфавит, домино-пасьянсы, словоскопы, буквенное лото, криптарифмы и другие задачи со словами и числами.

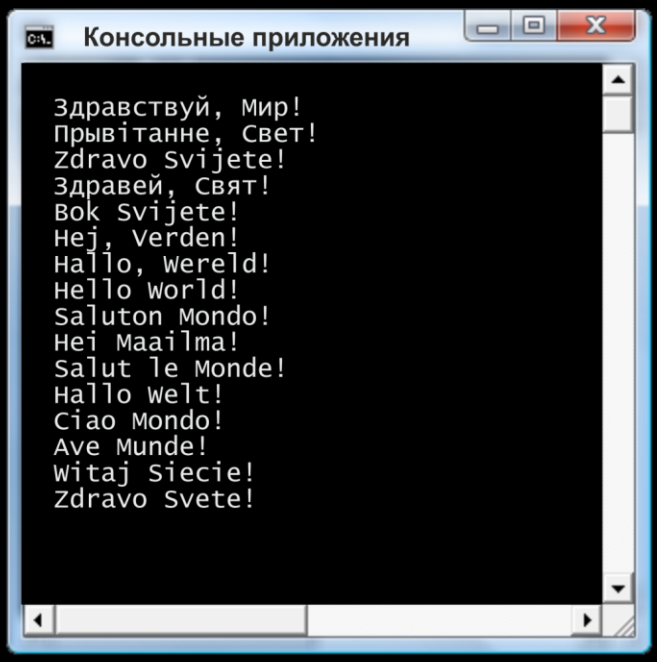
Издательство RVGames, 2013. – 470 с.

RVGAMES.DE 2013

Рубанцев Валерий

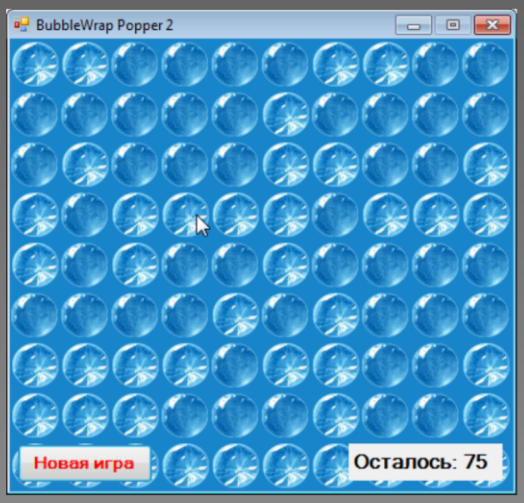
ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C# 5

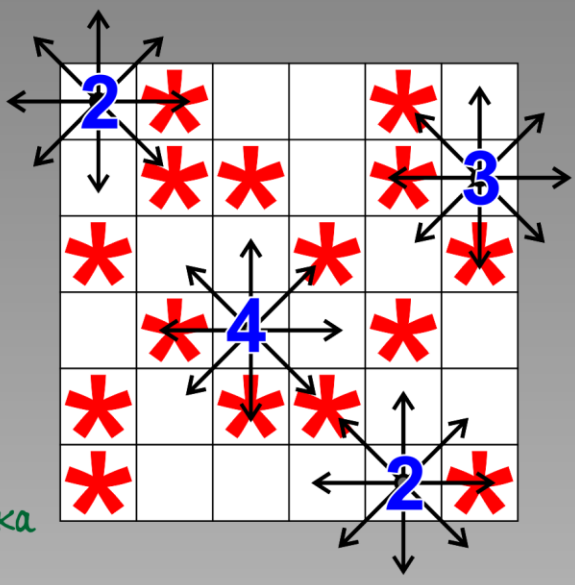
БАЗОВЫЙ УРОВЕНЬ



.NET 4.5

RVGAMES.DE 2013

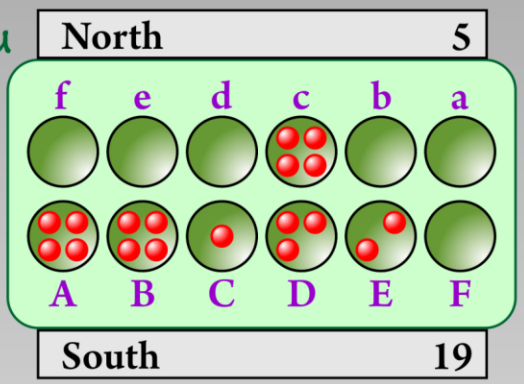




Программирование на языке C# 5.0 Базовый уровень

В книге подробно рассказывается об основах современного объектно-ориентированного языка программирования Си-шарп (C#): от грамматики языка до разработки конкретного класса - игры Awari. Теоретический материал дополняется многочисленными практическими проектами и заданиями для самостоятельного решения.

Издательство RVGames, 2013. – 855 с.

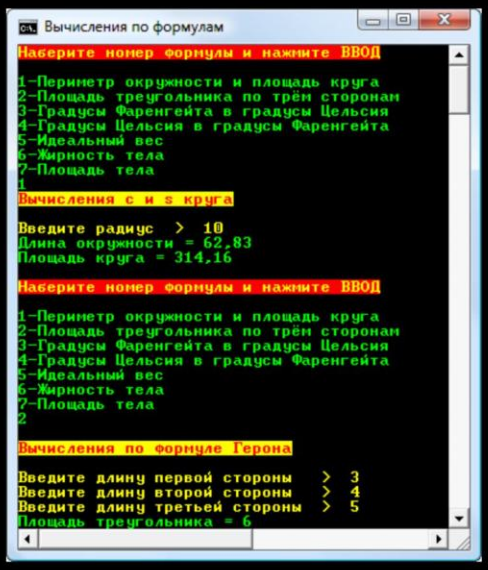


RVGAMES.DE 2013

Рубанцев Валерий

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C# 5

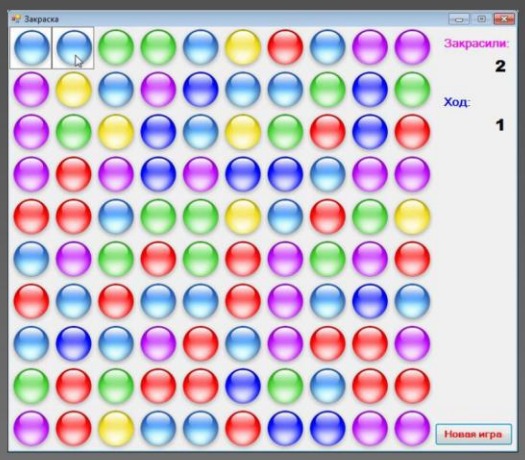
ПРАКТИКУМ ПО РЕШЕНИЮ ЗАДАЧ БАЗОВОГО УРОВНЯ

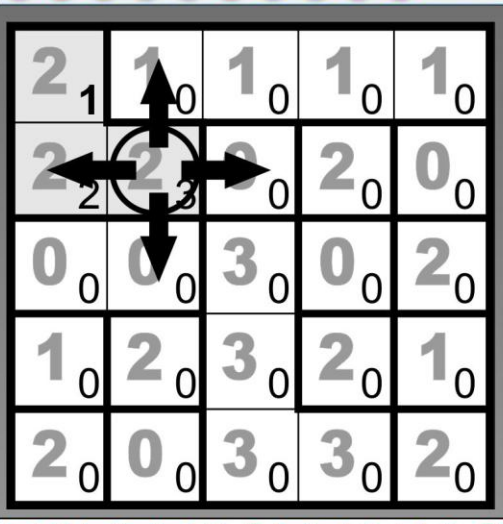


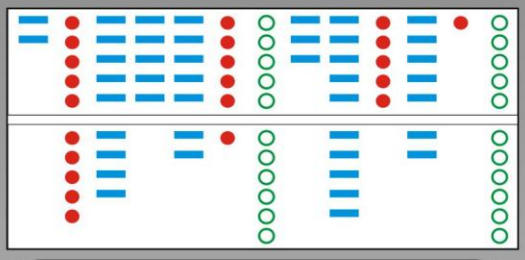
.NET 4.5

RVGAMES.DE 2013

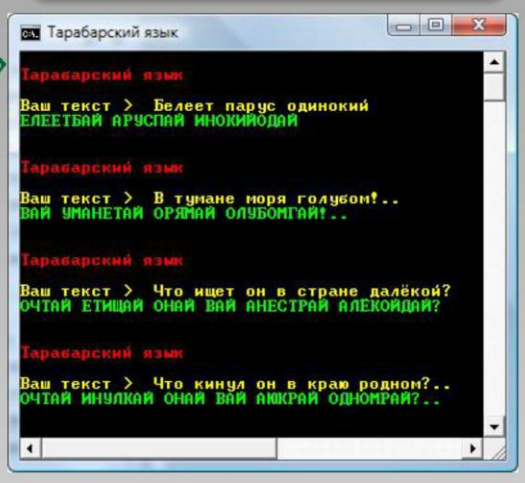
	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100







BOR · JOD = ARGON



Программирование на языке C# 5.0

Практикум по решению задач базового уровня

В книге на многочисленных примерах рассматривается решение реальных задач из различных областей знания - русского языка, математики, криптографии, комбинаторики - а также задач с сайта «Проект Эйлера» и из британского научного журнала «New Scientist». Много внимания уделяется разработке эффективных алгоритмов и выбору типов данных.

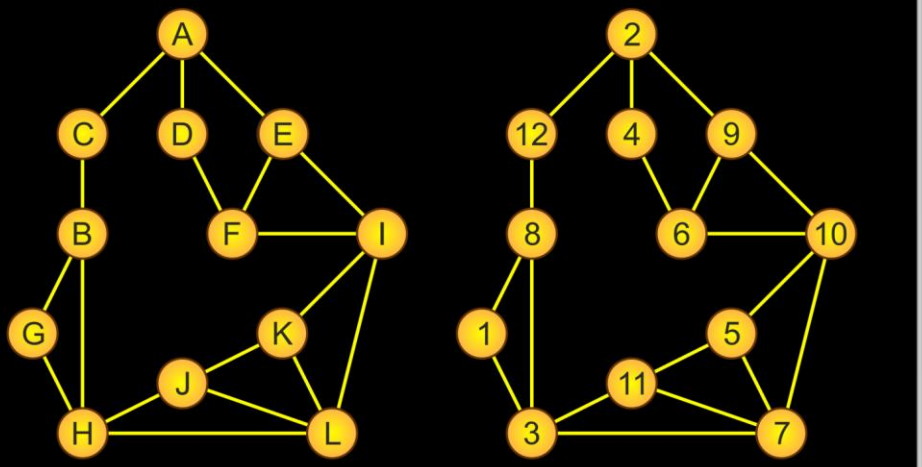
Издательство RVGames, 2013. – 405 с.

RVGAMES.DE 2013

Рубанцев Валерий

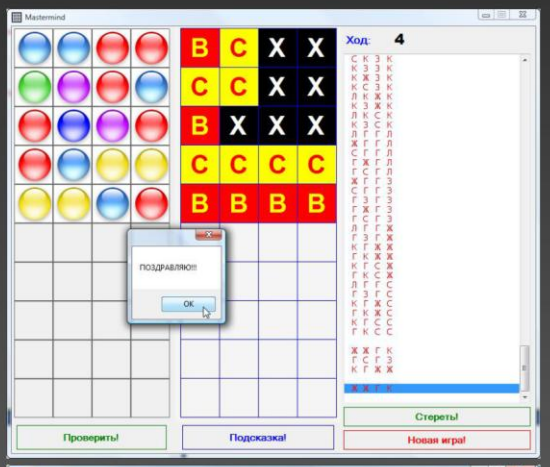
ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C# 5

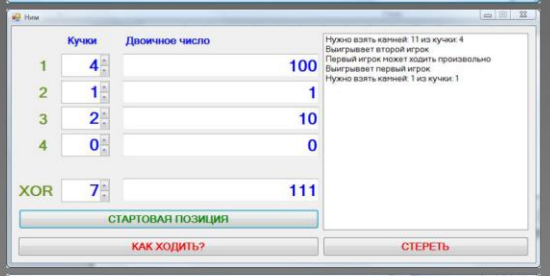
КОМБИНАТОРНЫЕ ИГРЫ И ГОЛОВОЛОМКИ

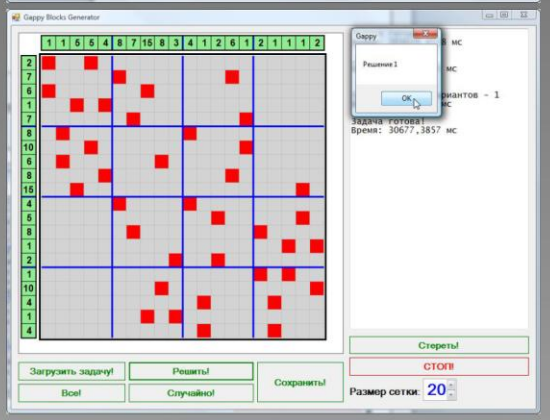


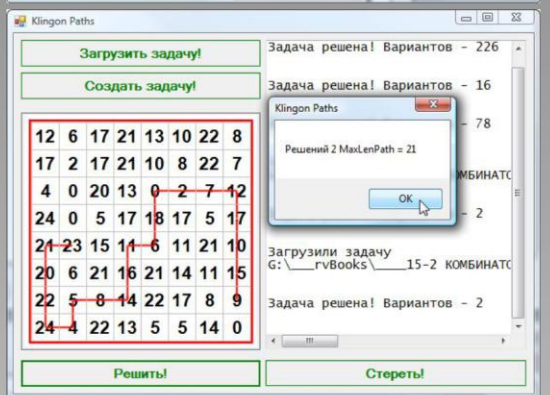
.NET 4.5

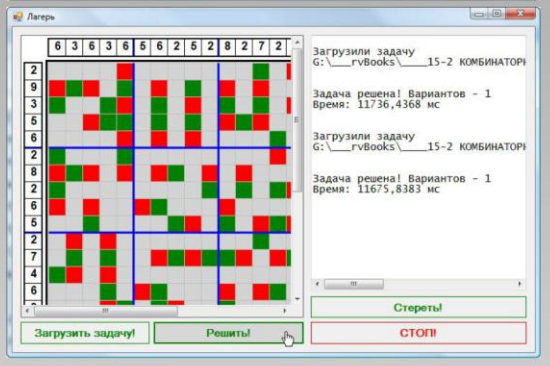
RVGAMES.DE 2013











Программирование на языке C# 5.0

Комбинаторные игры и головоломки

Второе, существенно дополненное издание книги «Как решать головоломные задачи на компьютере».

Новые главы:

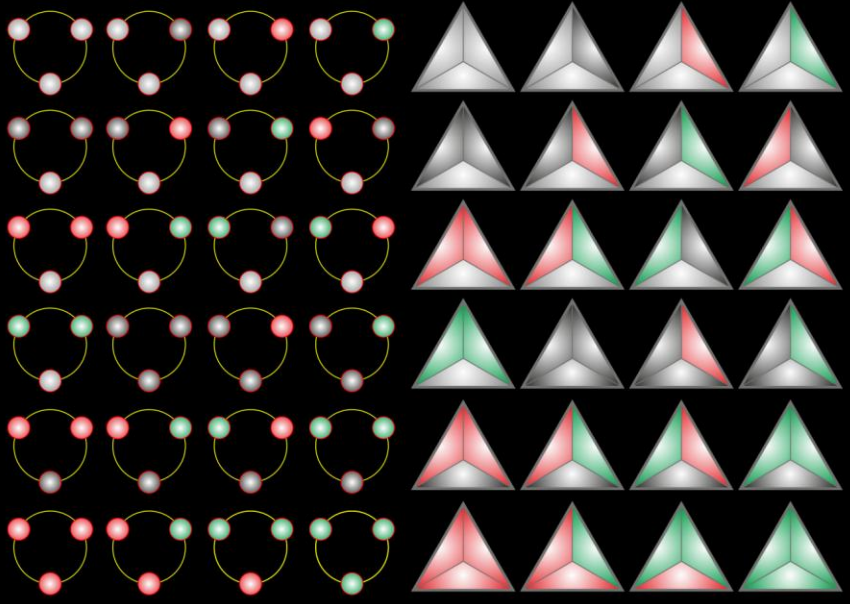
Издательство RVGames, 2013. – 830 с.

RVGAMES.DE 2013

Рубанцев Валерий

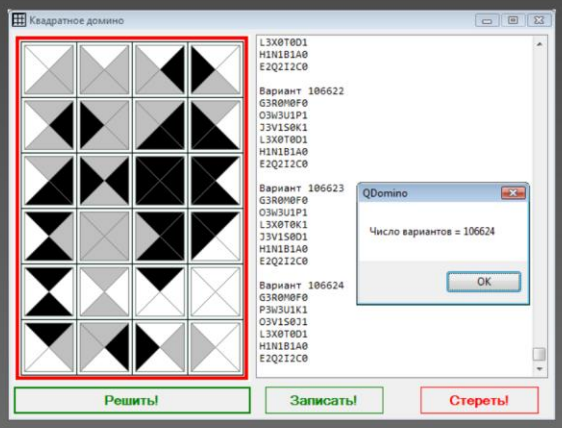
ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C# 5

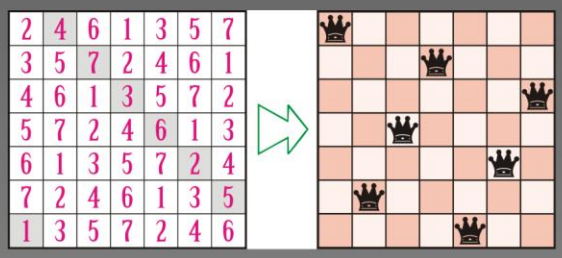
КОМБИНАТОРИКА, ЛОГИКА, ВЕРОЯТНОСТЬ

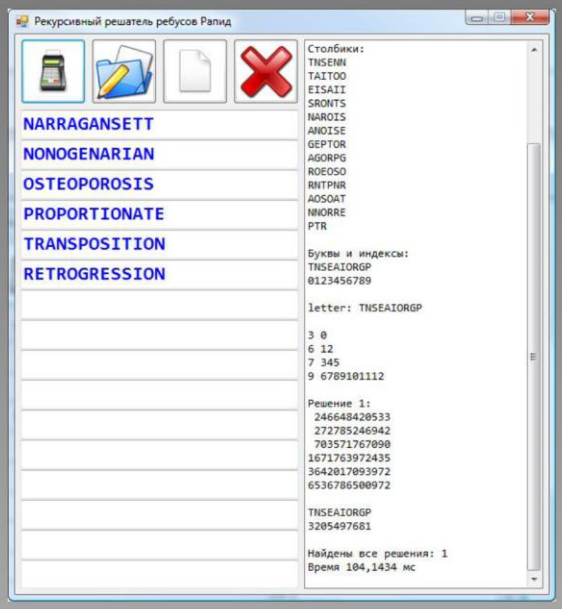


.NET 4.5

RVGAMES.DE 2013







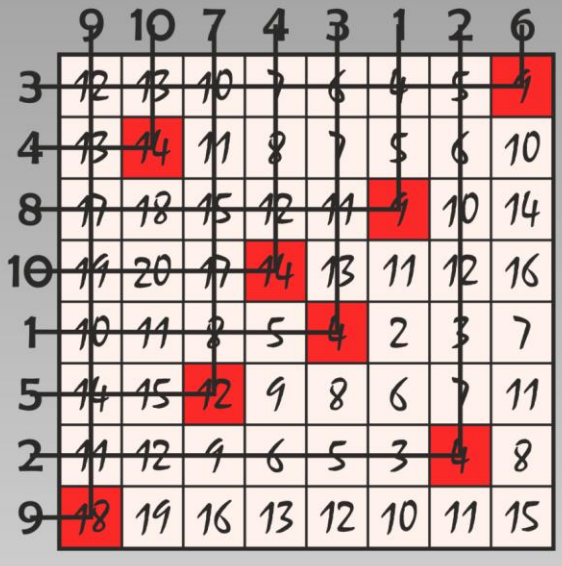
Программирование на языке C# 5.0

Комбинаторика, логика, вероятность

Второе, существенно дополненное издание книги «Как решать комбинаторные задачи на компьютере».

Новые главы:

Издательство RVGames, 2013. – 680 с.

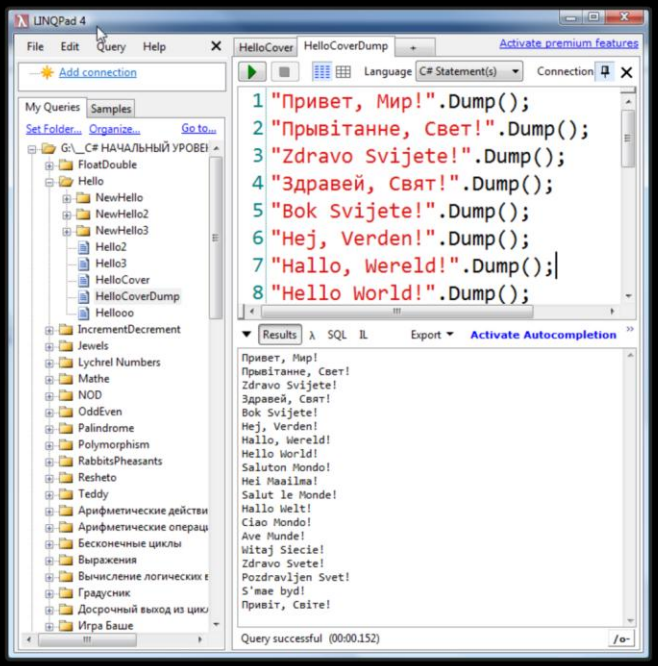


RVGAMES.DE 2013

Рубанцев Валерий

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C# 5

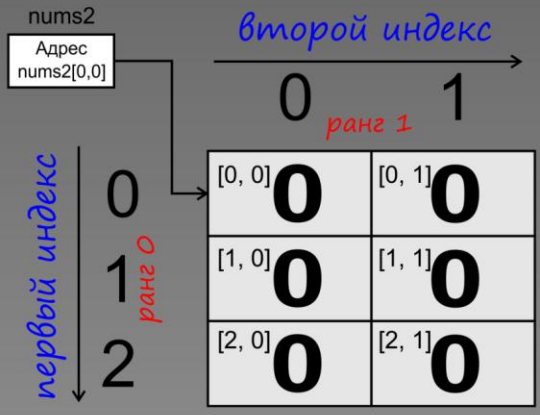
НАЧАЛЬНЫЙ УРОВЕНЬ

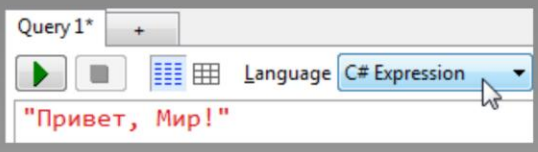


.NET 4.5

RVGAMES.DE 2014

int ← int int ← (int)long int ← (int)float int ← (int)double	long ← int long ← long long ← (long)float long ← (long)double
float ← int float ← long float ← float float ← (float)double	double ← int double ← long double ← float double ← double

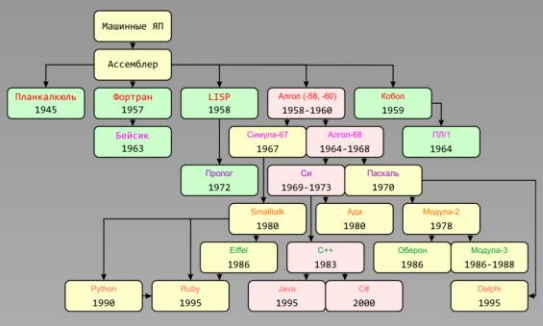


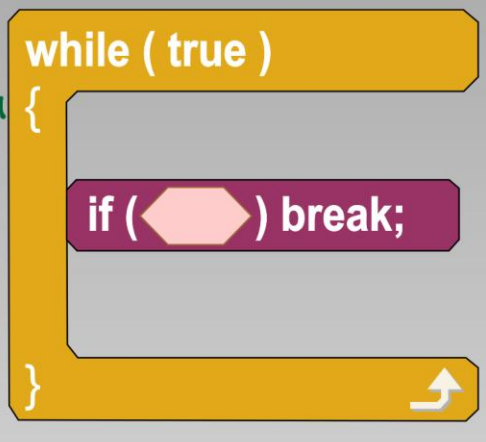


Программирование на языке C# 5.0
Начальный уровень

Подробный самоучитель современного объектно-ориентированного языка программирования Си-шарп. Книга содержит весь необходимый материал для разработки программ с текстовым интерфейсом. Теоретический материал сопровождается многочисленными практическими примерами и заданиями для самостоятельного решения.

Издательство RVGames, 2014. – 620 с.





RVGAMES.DE 2014

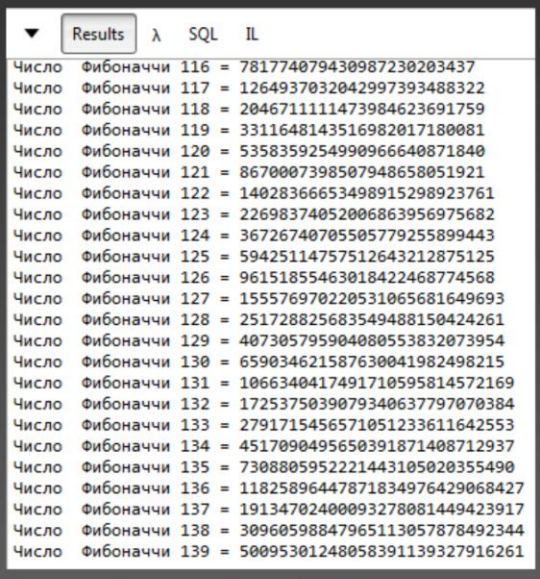
Рубанцев Валерий

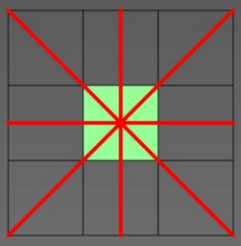
ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C# 5

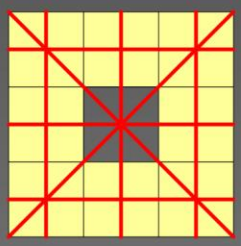
ПРАКТИКУМ ПО РЕШЕНИЮ ЗАДАЧ НАЧАЛЬНОГО УРОВНЯ

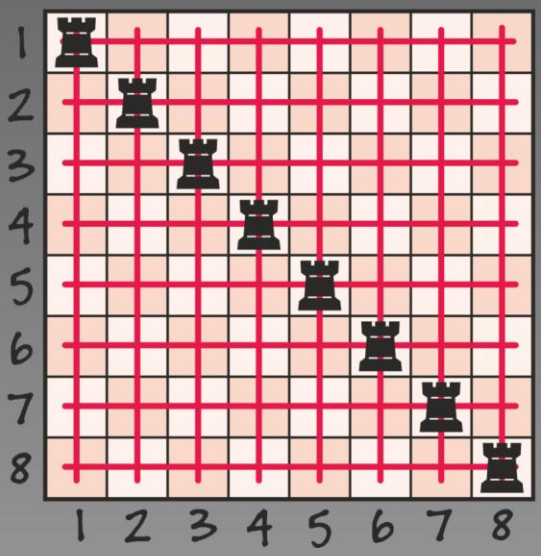
Числа, числа, числа
NOD, NOK и компания
Простые числа
Числовые ребусы
Степени и корни
Числовые ряды и другие задачи
Диофантовы уравнения
Линейное программирование
Компьютерные игры
Занимательная комбинаторика и
Теория вероятностей

1 .NET 4.5 RVGAMES.DE 2014 5





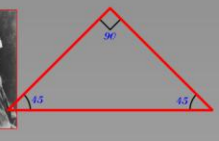


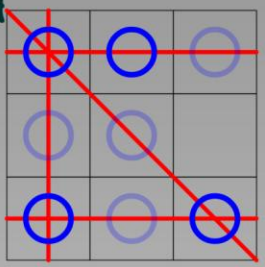


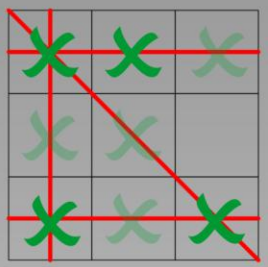
Программирование на языке C# 5.0

Практикум по решению задач начального уровня

2,7







Цель практикума - укрепление навыков программирования на языке Си-шарп.
В книге подробно рассматривается решение практических задач - в основном из области математики. Большое внимание уделяется разработке алгоритмов и выбору типов данных.

Издательство RVGames, 2014. – 420 с.

	(2)	(3)	4	(5)	6	(7)	8	9	10
(11)	12	(13)	14	15	16	(17)	18	(19)	20
21	22	(23)	24	25	26	27	28	(29)	30
(31)	32	33	34	35	36	(37)	38	39	40
(41)	42	(43)	44	45	46	(47)	48	49	50
51	52	(53)	54	55	56	57	58	(59)	60
(61)	62	63	64	65	66	(67)	68	69	70
(71)	72	(73)	74	75	76	77	78	(79)	80
81	82	(83)	84	85	86	87	88	(89)	90
91	92	93	94	95	96	(97)	98	99	100

RVGAMES.DE 2014

Список литературы, использованной в книге *Занимательные уроки с компьютером*

1. *Анатолий Маркуша*. **Вам - взлёт**! Издательство детской литературы МП РСФСР, 1982. – 238 с.
2. *Коснёвски Ч.* **Занимательная математика и персональный компьютер**. - Мир, 1987. – 192 с.
3. *Соболь И.М.* **Метод Монте-Карло**. - Наука, 1978. – 64 с.
4. *Мозговой М.В.* **Занимательное программирование: Самоучитель**. - Питер, 2005. – 208 с.
5. *Студенецкий Н.* **Мастерская головоломок**. – Детская литература, 1964. – 256 с.
6. *Кроновер Р.* **Фракталы и хаос в динамических системах**. – М.: Постмаркет, 2000. – 354 с.
7. *Гарднер Мартин.* **От мозаик Пенроуза к надёжным шифрам**. – М.: Мир, 1993. – 417 с.
8. *Эткинс П.* **Порядок и беспорядок в природе**. – М.: Мир, 1987. – 224 с.
9. *Гарднер Мартин*. **Математические досуги**. - М.:Мир, 1972. – 495 с.