ЛЕГКОЕ ПРОГРАММИРОВАНИЕ

SCRATCH ДЛЯ ДЕТЕЙ

САМОУЧИТЕЛЬ ПО ПРОГРАММИРОВАНИЮ

МАЖЕД МАРЖИ

МИФ ДЕТСТВО

no starch press

Majed Marji

Learn to Program with Scratch

A Visual Introduction to Programming
with Games, Art, Science, and Math

no starch
press

2014

Мажед Маржи

Scratch для детей

Самоучитель
по программированию

Перевод с английского
Марии Гескиной и Светланы Таскаевой

Издательство «Манн, Иванов и Фербер»
Москва, 2017

УДК 087.5:004.43
ББК 76.1, 62:32.973.412
М25

Издано с разрешения No Starch Press, Inc. a California Corporation

На русском языке публикуется впервые

Научный редактор Дарья Абрамова

Возрастная маркировка в соответствии с Федеральным законом № 436-ФЗ: 0+

Маржи, Мажед

М25 Scratch для детей. Самоучитель по программированию / Мажед Маржи; пер. с англ. М. Гескиной и С. Таскаевой — М. : Манн, Иванов и Фербер, 2017. — 288 с.

ISBN 978-5-00100-336-6

Scratch — это простой, понятный и невероятно веселый язык программирования для детей. В нем нет кодов, которые нужно знать назубок и писать без ошибок. Все, что требуется, — это умение читать и считать. Как из конструктора Lego, при помощи Scratch можно собирать программы из разноцветных «кирпичиков» — блоков. В программу можно вносить любые изменения в любой момент и сразу видеть, как она работает. Подробные объяснения, разобранные по шагам примеры и множество упражнений помогут освоить Scratch без труда. Книга подойдет детям от 8 лет (и их родителям!), а также всем, кто хочет научиться программировать с нуля.

УДК 087.5:004.43
ББК 76.1, 62:32.973.412

Все права защищены.
Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Title of English-language original:
Learn to Program with Scratch,
published by No Starch Press.

ISBN 978-5-00100-336-6

© Majed Marji, 2014
© Перевод на русский язык, издание на русском языке, оформление. ООО «Манн, Иванов и Фербер», 2017

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Scratch (читается как «скрэтч») — визуальный язык программирования, богатая обучающая среда для людей всех возрастов. Он позволяет создавать интерактивные мультимедийные проекты: мультфильмы, книжные обзоры, научные эксперименты, игры и симуляторы. В визуальной среде Scratch вы можете изучить те области знаний, которые были вам недоступны. В нем есть полный набор мультимедийных инструментов, с помощью которых легко создавать чудесные приложения. Причем это гораздо проще, чем в других языках программирования!

Scratch хорошо помогает в развитии навыка решения задач. А он важен во всех областях жизни, не только в программировании. В этой среде вы сразу получаете обратную связь и легко и быстро можете проверить свою логику. Визуальная структура позволяет очень просто отслеживать все шаги программ и развивать свое мышление. В целом благодаря Scratch легко понять основы компьютерной науки. Появляется мотивация для учебы и тяга к знаниям; вы учитесь в процессе, самостоятельно, исследуя разные области и совершая открытия. Начать легко, и все здесь зависит от вашей изобретательности и воображения.

Есть много книг, обещающих обучить вас программированию в среде Scratch. Большинство из них адресовано самым юным читателям, и в них есть всего несколько простых примеров, показывающих пользовательский интерфейс Scratch. Они скорее о самом Scratch, а не о программировании. Цель же нашей книги — обучить основам программирования, и Scratch здесь — только средство. А еще я хочу показать возможности Scratch как мощного инструмента преподавания и обучения.

Для кого эта книга

Эта книга — для всех, кто хочет освоить компьютерную науку. Она обучает основам программирования и может быть использована как задачник для учеников средней и старшей школы или самоучитель. В вузах с ее помощью можно обучить азам программирования студентов разных специальностей. Кроме того, она станет хорошей основой для вводного курса.

Преподаватели, которые захотят обучать языку Scratch, и сами лучше поймут принципы программирования благодаря этой книге. Они получат навыки, которые помогут грамотно объяснить философию Scratch студентам, опираясь на их потребности.

Чтобы заниматься по этой книге, не требуется опыта программирования. И она доступна всем, кто знает математику на уровне средней школы. Если какое-то задание покажется слишком сложным, его можно пропустить без ущерба для процесса обучения.

Читателям

Прелесть программирования — в возможности творить. Представьте себе: у вас появилась идея, вы пару часов провели за клавиатурой и создали новый программный продукт! Однако, как и любой другой навык, программирование требует тренировки. Во время обучения вы наверняка наделаете ошибок. Но не сдавайтесь. Подумайте, как можно воплотить вашу идею другими способами, опробуйте разные техники, пока не добьетесь результата. И возьмитесь за что-нибудь новое.

Особенности книги

В этой книге представлен практический подход к обучению программированию и основам компьютерных наук. Надеюсь, с его помощью любой читатель сможет развить свое воображение и научиться программировать.

Я предлагаю создавать проекты. Я буду подробно объяснять отдельные принципы, а потом мы вместе разработаем программы, иллюстрирующие их. Акцент здесь делается на решении задач, а не конкретных функциях Scratch.

Примеры в этой книге показывают широкий диапазон тем, которые вы сможете освоить с помощью Scratch. Они были тщательно отобраны и объясняют основы программирования и возможности глубже изучить другие темы в рамках этой среды.

Упражнения и задачи в конце каждой главы — отличный способ проверить ваши навыки программирования. В них также предлагается добавить новые ходы и объединить уже изученные принципы в более широкую задачу. Займитесь этими упражнениями и старайтесь придумывать новые задания. Умение решать собственные задачи показывает, что вы уже глубоко понимаете принципы программирования.

Структура текста

В первых трех главах этой книги я представлю Scratch как мощный инструмент для рисования геометрических фигур и создания мультимедийных приложений. Вы быстро и легко освоите эти его функции. А остальная часть книги посвящена элементам программирования, поддерживаемым в Scratch.

Глава 1: введение в программную среду Scratch, описание командных блоков и процесса создания простых программ.

Глава 2: обзор разделов «Движение» и «Рисование», команды движения, введение в графический редактор Scratch.

Глава 3: команды разделов «Звуки» и «Внешность».

Глава 4: основы процедур в Scratch, их структура, модульное программирование. Зачем так рано? Чтобы сразу научиться программировать грамотно.

Глава 5: использование переменных для записи информации. Также вы узнаете, как задавать вопросы пользователю и получать ответы. Это поможет в дальнейшем создавать различные интерактивные приложения.

Глава 6: принятие решений и контроль работы программ в Scratch.

Глава 7: использование доступных в Scratch функций повтора на конкретных примерах.

Глава 8: строковые данные и подборка полезных процедур, включающих использование строк.

Глава 9: списки в Scratch как области хранения информации и их использование для создания мощных программ.

Во всех главах также есть готовые проекты, которые могут быть использованы как основа для создания похожих приложений с различными параметрами. Изучив эту книгу, вы сможете самостоятельно писать программы!

Условные обозначения

В книге использованы разные текстовые выделения для обозначения элементов Scratch.

Элементы интерфейса Scratch: **когда щелкнут по**  .

Спрайты и переменные: Мяч.

Имена файлов, которые будут полезны для изучения соответствующего раздела, вынесены на поля:

RotationCenter.sb2

Упражнения оформлены следующим образом:

> **УПРАЖНЕНИЕ**
>
> В этом разделе я предложу вам испытать свои силы и сделать что-то самостоятельно.

Онлайн-ресурсы

На сайте http://nostarch.com/learnscratch/ можно найти дополнительные ресурсы по этой книге. Загрузив и распаковав архив, вы увидите следующие материалы.

Дополнительные программы (Bonus Applications). В этой папке размещены дополнительные приложения Scratch, которые можно изучить самостоятельно. В файле Bonus Applications.pdf вы найдете подробные объяснения.

Скрипты глав (Chapter Scripts). В этой папке есть все скрипты, описанные в книге.

Дополнительные ресурсы (Extra Resources). Три PDF-файла с более подробной информацией по отдельным темам (графический редактор, математические функции, рисование геометрических фигур).

Решения (Solutions). Решения для всех задач и упражнений из книги*.

---

* Для того чтобы вам легче было освоить Scratch, имена всех спрайтов, названия процедур, переменных и флагов, которые упоминаются в примерах и скриптах, мы приводим на русском языке. Scratch позволяет вводить данные на любом языке. Все спрайты, процедуры и параметры можно переименовать, и работа скриптов при этом не изменится. Для удобства оригинальные названия параметров и их перевод собраны в «Кратком англо-русском словаре Scratch» в конце книги. *Прим. ред.*

1

ПЕРВЫЕ ШАГИ

Вам когда-нибудь хотелось создать свою компьютерную игру, снять мультфильм, сделать учебное пособие или научную симуляцию? Scratch — графический язык программирования, который позволит вам легко создавать подобные программные приложения и многое другое.

В этой вводной главе вы:

- познакомитесь со средой программирования Scratch;
- узнаете о разных типах блоков команд;
- создадите в Scratch свою первую игру.

Создав программу в Scratch, вы можете сохранить ее на своем компьютере или загрузить на сайт Scratch, где другие пользователи смогут ее прокомментировать или использовать при создании новых проектов. Круто? Тогда за дело!

Что такое Scratch?

Программа — просто набор инструкций, которые сообщают компьютеру, что ему делать. Они пишутся при помощи *языка программирования*. Scratch — один из них, и он уникален. Большинство языков программирования имеют *текстовую основу*: нам приходится давать компьютеру команды, похожие на шифровки, на английском языке.

Например, чтобы на экране появилась строка «Привет!», нужно написать:

```
print('Привет!')                          (на языке Python)
std::cout << "Привет!" << std::endl;      (на языке C++)
System.out.print("Привет!");               (на языке Java)
```

Выучить эти языки и понять их синтаксис непросто для начинающего. Scratch же — *визуальный* язык программирования. Он был разработан в медиалаборатории Массачусетского технологического института, чтобы сделать программирование более доступным, а обучение ему — более занимательным.

На Scratch вам не надо писать никаких сложных команд. Вы будете создавать программы, соединяя графические блоки. Непонятно? Взгляните на простую программу на рис. 1.1, и я всё объясню.

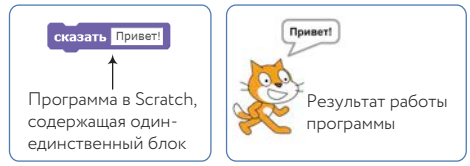


Рис. 1.1. Когда вы запускаете этот блок в Scratch, у кота в облачке появляется текст «Привет!»

Кот, которого вы видите на рис. 1.1, называется *спрайтом*. Спрайты понимают и выполняют наборы команд, которые вы им даете. Сиреневый блок слева говорит коту, что нужно, чтобы в облачке появился текст «Привет!». У многих программ, которые вы создадите, читая эту книгу, будет несколько спрайтов, и вы сможете использовать блоки, чтобы заставить спрайты двигаться, поворачиваться, говорить, играть музыку, решать задачи и т. п.

В Scratch вы можете программировать, соединяя разные блоки, промаркированные цветами, как если бы вы собирали что-то из кубиков Lego. Соединенные между собой блоки, или «стеки» блоков, которые вы создаете, называются *скриптами*. Например, на рис. 1.2 показано, как скрипт четыре раза меняет цвет спрайта.

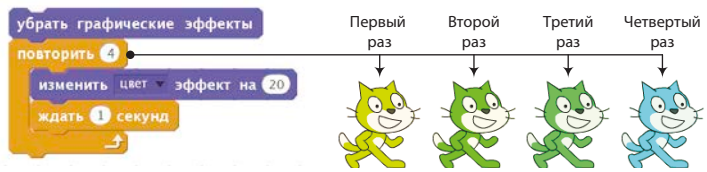


Рис. 1.2. Использование скрипта для изменения цвета спрайта

Этот скрипт ждет одну секунду, прежде чем поменять цвет, а те четыре кота, которых вы здесь видите, показывают новые цвета, в которые окрашивается спрайт после каждого изменения.

УПРАЖНЕНИЕ 1.1

Изучите блоки на рис. 1.2, посмотрите на их форму и попробуйте разобраться, какие шаги должны были содержаться в скрипте, чтобы кот стал голубого цвета. (Подсказка: первый сиреневый блок возвращает коту его исходную окраску.) Как вы думаете, что произойдет, если мы уберем из скрипта блок **ждать**?

В этой книге мы рассматриваем среду Scratch 2, которая была запущена в мае 2013 года. Эта версия позволяет вам создавать проекты непосредственно в браузере. Вам не нужно устанавливать на компьютер никаких программ, а весь материал в этой книге рассчитан на онлайн-интерфейс (при этом вы можете работать и с установленной программой на компьютере).

Теперь пора отправиться в ваше первое путешествие по программированию и научиться использовать этот язык.

Среда программирования Scratch

Чтобы запустить Scratch, войдите на сайт http://scratch.mit.edu/ и кликните по ссылке Попробуйте. Интерфейс редактора проектов Scratch показан на рис. 1.3.

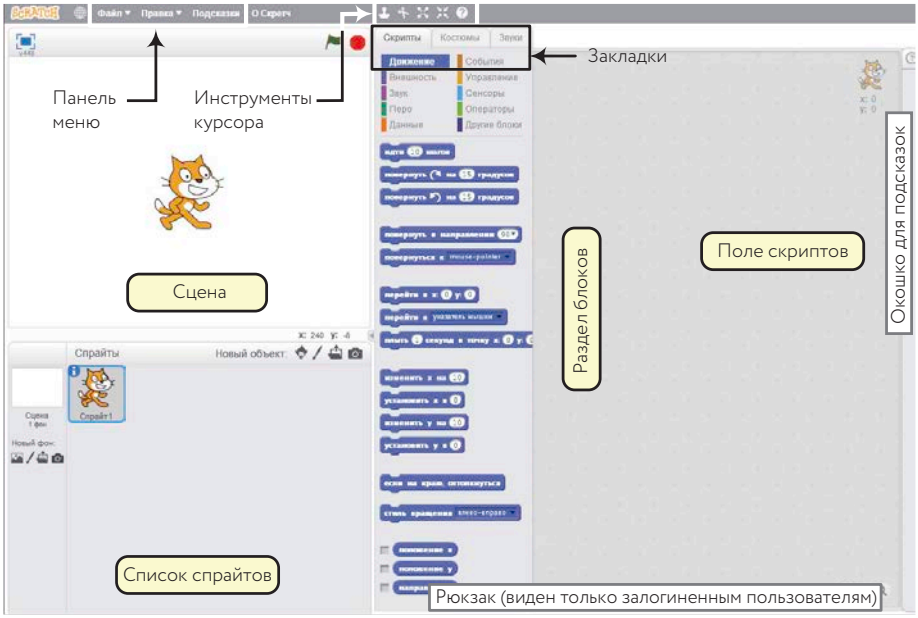


Рис. 1.3. Пользовательский интерфейс Scratch, где вы будете создавать свои программы

Вы должны видеть одно окно, состоящее как минимум из следующих трех панелей: **Сцены** (наверху слева), **Списка спрайтов** (слева внизу) и закладки **Скрипты** (справа), которая, в свою очередь, состоит из колонки **блоков** и поля **скриптов**. Правая панель также содержит две дополнительные закладки, **Костюмы** и **Звуки**, о которых мы поговорим чуть позже. Если вы создали аккаунт и вошли под ним на сайт Scratch, вы также должны увидеть **Рюкзак** (внизу справа), у которого есть кнопки, позволяющие делиться вашим проектом и использовать спрайты и скрипты из уже существующих проектов.

Познакомимся поближе с основными панелями.

Сцена

Сцена — место, где ваши спрайты двигаются, рисуют и взаимодействуют. Ширина ее 480 шагов, а высота — 360 шагов, как показано на рис. 1.4. Центр **Сцены** — точка отсчета осей координат *x* и *y*.

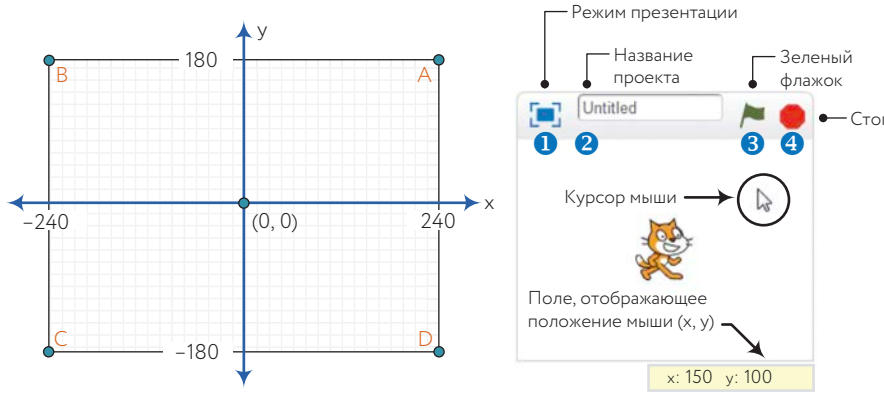


Рис. 1.4. Сцена — плоскость координат с отметкой (0, 0) посередине

Вы можете определить координаты любой точки на **Сцене**, наведя на нее курсор и посмотрев цифры в поле, отображающем положение мыши (*x*, *y*), которое расположено непосредственно под **Сценой**.

На небольшой панели управления над **Сценой** есть несколько пунктов. Иконка режима презентации ❶ скрывает все скрипты и инструменты программирования и разворачивает поле **Сцены** практически на весь экран. Поле редактирования ❷ показывает название текущего проекта. Зеленый флажок ❸ и значок **Стоп** ❹ позволяют вам запустить и остановить программу.

УПРАЖНЕНИЕ 1.2

Поводите курсором мыши по **Сцене** и посмотрите на поле, отображающее положение мыши. Что происходит, когда вы уводите курсор за пределы **Сцены**? А теперь перейдите в режим презентации и посмотрите, как изменится экран. Кликните по иконке слева вверху или нажмите **Esc** на клавиатуре, чтобы выйти из режима презентации.

Список спрайтов

Список спрайтов показывает имена и иконки всех спрайтов вашего проекта. Новые проекты начинаются с белого поля **Сцены** и одного спрайта-кота, как показано на рис. 1.5.

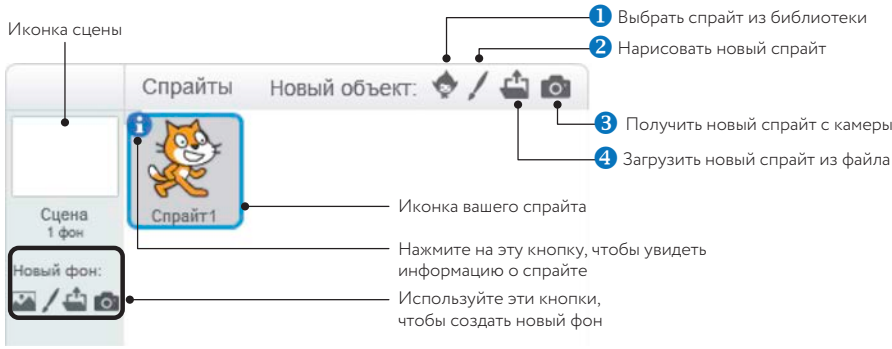


Рис. 1.5. Список спрайтов для нового проекта

Кнопки над списком спрайтов позволяют вам добавить в ваш проект новые спрайты из четырех источников: библиотеки спрайтов Scratch ❶, встроенного редактора Paint ❷ (вы можете там самостоятельно нарисовать костюм), с камеры, подключенной к вашему компьютеру ❸, или просто с вашего компьютера ❹.

УПРАЖНЕНИЕ 1.3

Добавьте в свой проект новые спрайты, используя кнопки, расположенные над списком спрайтов. Поменяйте расположение спрайтов в списке, перетаскивая с места на место их иконки.

У каждого спрайта в вашем проекте свои скрипты, костюмы и звуки.

Вы можете выбрать любой спрайт, чтобы посмотреть на его атрибуты. Либо кликните по его иконке в списке, либо дважды кликните по самому спрайту на **Сцене**.

Текущая иконка всегда выделена и обведена синей рамочкой. Выбрав спрайт, вы получаете доступ к его скриптам, костюмам и звукам.

Для этого нужно кликнуть по одной из трех закладок наверху над полем скриптов. Позже мы познакомимся с содержанием каждой из них. А пока кликните правой кнопкой мыши (или нажмите Ctrl+клик, если у вас Mac) по иконке спрайта-кота, чтобы увидеть выпадающее меню, как на рис. 1.6.

Функция дублирования ❶ копирует спрайт и дает копии другое имя. Вы можете удалить спрайт из своего проекта при помощи кнопки ❷, а также экспортировать спрайты в файл *.sprite2* на вашем компьютере, используя функцию «Сохранить локальный файл» ❸. (Чтобы импортировать или экспортировать спрайт в другой проект, кликните на **Загрузить спрайт из файла**, как показано на рис. 1.5.) Функция **спрятаться/показаться** ❹ позволяет делать спрайт на **Сцене** видимым или невидимым.

Рис. 1.6. При клике правой кнопкой мыши по иконке спрайта появляется вот такое удобное меню

Помимо иконок спрайтов, список спрайтов показывает слева иконку **Сцены** (см. рис. 1.6). У нее есть свой набор скриптов, изображений и звуков. Фоновое изображение, которое вы видите на **Сцене**, мы так и называем: *фон*. Когда вы начинаете новый проект, фон по умолчанию белый. Но вы можете добавить новые изображения при помощи любой из четырех кнопок, расположенных под иконкой **Сцены**. Кликните по иконке **Сцены** в списке спрайтов, чтобы просмотреть привязанные к ней скрипты, фоны и звуки.

Панель блоков

Блоки в Scratch разделены на 10 категорий: **Движение**, **Внешность**, **Звук**, **Перо**, **Данные**, **События**, **Управление**, **Сенсоры**, **Операторы** и **Другие блоки**. Каждой присвоен свой цвет, чтобы вам было проще находить родственные блоки. В Scratch 2 более 100 блоков, но некоторые появляются только при определенных условиях.

Например, блоки из раздела **Данные** (о них пойдет речь в главах 5 и 9) появляются только после того, как будут созданы переменная или список. Посмотрим на разные компоненты панели блоков на рис. 1.7.

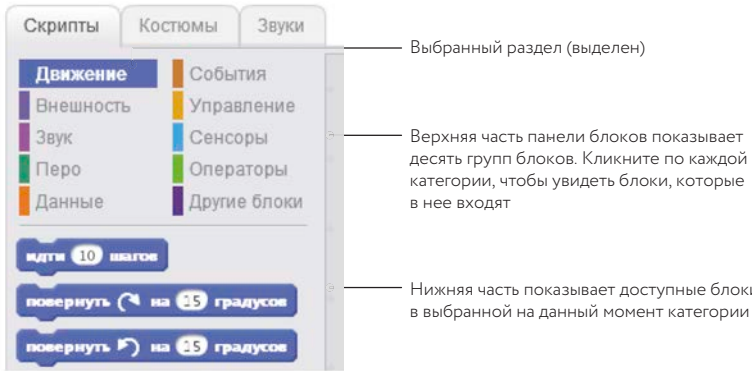


Рис. 1.7. Так выглядит панель блоков

Попробуйте кликнуть по блоку, чтобы увидеть, что он делает. Если кликнете по блоку **идти 10 шагов** из раздела **Движение**, то спрайт пройдет по сцене 10 шагов. Кликните еще раз — и спрайт пройдет еще 10 шагов. Кликните по блоку **говорить Привет! в течение 2 секунд** (из раздела **Внешность**), чтобы у спрайта в текстовом облаке на 2 секунды возникла надпись «Привет!». Вы также можете вызвать экран справки, выбрав **Помощь по блоку** (иконка со знаком вопроса) из панели инструментов и кликнув на блок, который вызывает у вас вопросы.

Некоторым блокам необходимы один или более вводов дополнительных данных (их также называют *аргументами*), которые указывают, что делать. Цифра 10 в блоке **идти 10 шагов** — пример такого аргумента. На рис. 1.8 вы увидите, какими способами разные блоки меняют дополнительные данные.

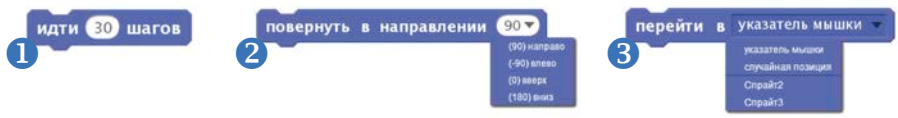


Рис. 1.8. Изменение дополнительных данных для разных типов блоков

Вы можете изменить число шагов в блоке **идти 10 шагов**, кликнув в белом окошке с цифрой 10 и введя новое число ❶, скажем 30, как на рис. 1.8. Некоторые блоки, например **повернуть в направлении 90**, оснащены выпадающим меню с дополнительными данными ❷. Чтобы увидеть доступные варианты и выбрать один из них, нужно кликнуть по стрелочке вниз. У этой команды есть белое редактируемое поле, так что здесь тоже можно ввести любую цифру.

Другие блоки, скажем **перейти в** ❸, потребуют, чтобы вы выбрали один из пунктов выпадающего меню.

УПРАЖНЕНИЕ 1.4

Откройте раздел **Внешность**, измените дополнительные данные блоков и кликните по блокам, чтобы посмотреть, что они делают. Поэкспериментируйте с блоком **установить цвет эффект в значение**. Попробуйте цифры 10, 20, 30 и так далее, пока кот не вернется к своему изначальному цвету. Попробуйте варианты из выпадающего меню с разными цифрами. Вы можете кликнуть по блоку **убрать графические эффекты** (также из раздела **Внешность**), чтобы удалить все изменения.

Поле скриптов

Чтобы ваш спрайт делал интересные вещи, вам нужно запрограммировать его, перетянув блоки из панели блоков в поле скриптов и соединив их между собой. Когда вы тащите блок по полю скриптов, белое выделение показывает вам, куда его можно прикрепить, чтобы получилась работающая связка с другим блоком (рис. 1.9). Блоки в Scratch соединяются между собой только определенным способом. Благодаря этому удается избежать опечаток, которые возникают, когда люди используют текстовые языки программирования.

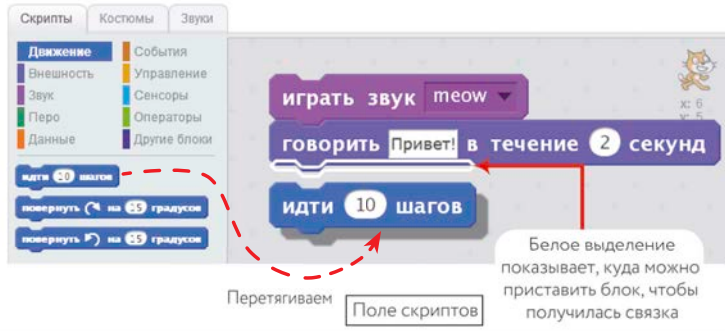


Рис. 1.9. Перетащите блоки на поле скриптов и присоедините их друг к другу, чтобы получились скрипты

Вам не нужно писать скрипт до конца, чтобы запустить его: вы можете проверять его в процессе работы. Клик по любой части скрипта запускает его от начала до конца.

Вы легко можете отсоединить несколько блоков и протестировать каждый из них отдельно. Эта функция вам очень пригодится, когда вы будете разбираться с длинными скриптами. Чтобы переместить целую группу блоков, возьмите верхний из нее. Чтобы отсоединить блок в середине стопки и все блоки под ним, выберите его и перетащите. Попробуйте это сделать самостоятельно. Эта функция также позволяет работать над проектом поэтапно. Вы можете объединять небольшие группы блоков,

тестировать их, удостоверяться, что они работают как нужно, а затем объединять их в более крупные скрипты.

УПРАЖНЕНИЕ 1.5

Создайте новый проект Scratch и задайте коту-спрайту скрипт, показанный ниже. (Блок **всегда** находится в разделе **Управление**, а остальные блоки — в разделе **Движение**.)

О большинстве этих блоков вы узнаете больше из главы 2. А пока кликните по своему скрипту, чтобы запустить его (Scratch выделит работающий скрипт светящейся желтой рамкой, как показано в правой части картинки). Вы даже можете поменять дополнительные данные в блоках и добавить новые блоки, пока скрипт работает! Например, измените цифру в блоке **идти** и посмотрите, как изменятся движения кота. Кликните по скрипту еще раз, чтобы остановить его.

А еще вы можете скопировать «стек» блоков у одного спрайта для другого. Просто перетащите его из поля скриптов исходного спрайта на иконку другого в списке спрайтов.

УПРАЖНЕНИЕ 1.6

Добавьте в ваш проект еще один спрайт. Перетащите скрипт спрайта-кота на иконку нового спрайта. Нужно, чтобы стрелка курсора оказалась на иконке нового спрайта. Проверьте колонку скриптов нового спрайта, чтобы удостовериться, что там есть копия скрипта.

Закладка Костюмы

Вы можете изменить внешний вид спрайта, сменив его костюм, который представляет собой картинку. Закладка **Костюмы** содержит все необходимое, чтобы вы могли организовать костюмы спрайта; это что-то вроде платяного шкафа. В нем может быть много костюмов, но в каждый конкретный момент спрайт может надеть только один.

Попробуем прямо сейчас поменять костюм спрайта-кота. Кликните по его иконке и выберите закладку **Костюмы**. Как показано на рис. 1.10,

у кота есть два костюма: костюм1 и костюм2. Выделенный (костюм1 в данном случае) — тот, что используется в настоящий момент.

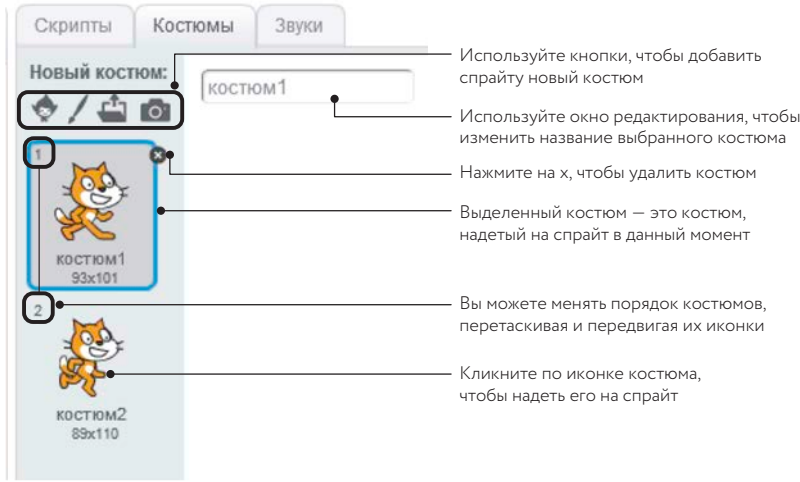


Рис. 1.10. Вы можете организовать все костюмы для спрайта в одноименной закладке

Если кликнуть правой кнопкой мыши на иконке костюма, вы увидите выпадающее меню с тремя вариантами: **дублировать**, **удалить** и **сохранить локальный файл**. Первый создает новый костюм, точно такой же, как тот, что вы дублировали. Кнопка **Удалить** удаляет выбранный костюм. Последний вариант позволяет сохранить костюм в файл. Костюмы можно импортировать и использовать в других проектах при помощи кнопки **Загрузить костюм из файла** (третья кнопка на рис. 1.10). А теперь попробуйте сами все эти функции.

УПРАЖНЕНИЕ 1.7

Кликните по первой кнопке над изображением кота на рис. 1.10, чтобы выбрать новый костюм из библиотеки Scratch. Затем выберите любое понравившееся вам изображение из появившегося окошка. Используйте советы с рис. 1.10, чтобы лучше понять функции костюмов.

Закладка Звук

Спрайты также могут издавать звуки, которые оживляют вашу программу: например, когда они счастливы или расстроены. Если в вашей игре есть спрайт, который выглядит как ракета, вы можете заставить его издавать разные звуки, когда она попадает в цель или промахивается.

Кнопки в закладке **Звук** помогут вам организовать различные звуки, которые издают ваши спрайты. Как вы видите на рис. 1.11, Scratch также

дает вам инструмент, который позволяет редактировать звуковые файлы. Мы не будем подробно рассматривать работу с этим инструментом, но я очень советую вам поэкспериментировать с ним самостоятельно.

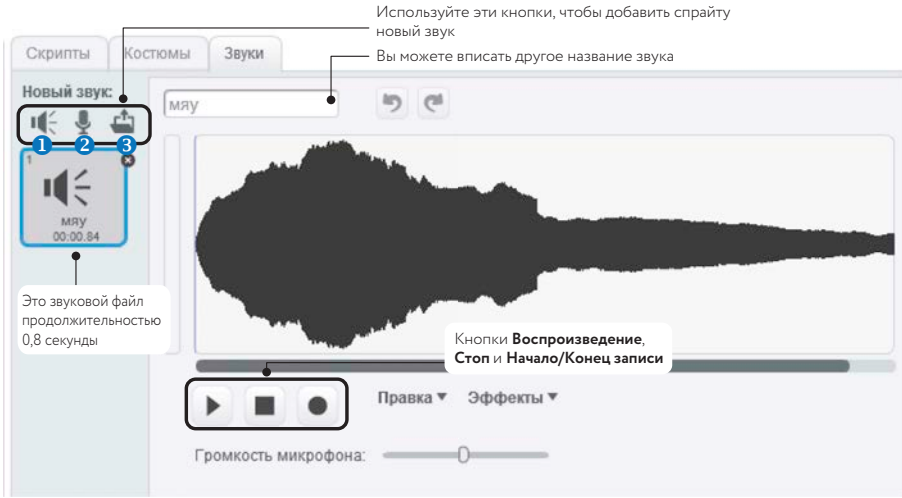


Рис. 1.11. Закладка **Звуки** *позволяет организовать звуки спрайта*

Вам в основном понадобятся только три кнопки наверху закладки **Звуки**. Они позволяют вам выбрать звук из библиотеки Scratch ❶, записать новый звук ❷ (если у вас есть микрофон) или импортировать уже существующий звуковой файл с вашего компьютера ❸. Scratch читает только файлы в форматах MP3 и WAV.

УПРАЖНЕНИЕ 1.8

Выберите закладку **Звуки** и кликните по кнопке **Выбрать звук** из библиотеки. Послушайте различные звуки, которые доступны в Scratch, в поисках идей для ваших будущих проектов.

Закладка Фоны

Когда вы выбираете иконку **Сцены** в списке спрайтов, название средней закладки меняется с **Костюмы** на **Фоны**. Используйте ее, чтобы организовать фоновые изображения **Сцены**, которые вы можете менять при помощи своих скриптов.

Например, если вы разрабатываете игру, вы можете показать один фон с инструкциями в начале, а потом переключиться на другой, когда пользователь начинает игру. Закладка **Фоны** идентична закладке **Костюмы**.

УПРАЖНЕНИЕ 1.9

Кликните по кнопке **Выбрать фон** из библиотеки под иконкой **Сцены** в списке спрайтов. Выберите в появившемся окне фон с сеткой координат и кликните **OK**. Scratch добавит сетку в ваш проект и сделает ее фоном по умолчанию. (Это двумерная координатная плоскость, которая полезна при работе с блоками команд движения.) Повторите эти шаги и выберите любой понравившийся вам фон.

Информация спрайта

Вы можете открыть поле информации о спрайте, кликнув по значку в левом верхнем углу иконки, как показано на рис. 1.12. В этом поле вы видите название спрайта, его текущую позицию (*x*,*y*), направление и стиль вращения, статус видимости. Также здесь указано, можно ли его перетащить в режиме презентации.

Коротко обсудим эти функции.

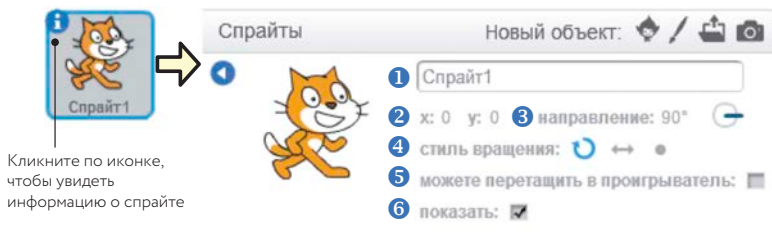


Рис. 1.12. Поле информации о спрайте

Окно редактирования ❶ вверху этого поля позволяет вам менять имя спрайта. При работе с этой книгой вам не раз придется им воспользоваться. Значения *x* и *y* ❷ показывают текущую позицию спрайта на **Сцене**.

Перетащите спрайт на **Сцену** и посмотрите, что произойдет с этими цифрами. Направление спрайта ❸ показывает, в каком направлении он будет двигаться в ответ на команды. Потяните за синюю линию, выходящую из центра кружка, чтобы повернуть спрайт.

Три кнопки стиля вращения ❹ (**Вращение**, **Прыжок вправо-влево** и **Нет вращения**) контролируют, как выглядит костюм, когда спрайт меняет направление движения. Чтобы понять смысл этих кнопок, создайте скрипт, как показано на рис. 1.13, а затем нажмите на каждую из этих кнопок, пока скрипт работает. Блок **ждать** вы найдете в разделе **Управление**.

Чекбокс **Можете перетащить в проигрыватель** ❺ показывает, может ли спрайт быть перетащен при помощи мыши с места на место в режиме презентации. Переключитесь в режим презентации, предварительно поставив галочку в этом чекбоксе, и попробуйте передвинуть спрайт по **Сцене**, чтобы увидеть действие этой функции.

Чекбокс **Показать** ❻ позволяет вам показать/скрыть спрайт на время, пока создается программа. Попробуйте и посмотрите, что получится. В этой книге вы увидите несколько примеров скрытых спрайтов, которые выполняют важную работу за кадром.

Рис. 1.13. Скрипт для демонстрации стилей вращения

Панель инструментов

Взглянем на панель инструментов Scratch (рис. 1.14). Начнем с кнопок. (Панель инструментов будет выглядеть чуть иначе, если вы вошли на сайт под логином; об этом можно прочесть в приложении А.) Используйте кнопки **Дублировать** и **Удалить**, чтобы скопировать или удалить спрайты, костюмы, звуки, блоки или скрипты. Кнопка **Увеличить** делает спрайт больше, а кнопка **Уменьшить** — меньше. Кликните по той кнопке, которую хотите использовать, а потом по спрайту (или скрипту), чтобы применить функцию. Чтобы вернуть курсору форму стрелки, кликните по пустому месту на экране. Вы можете использовать языковое меню, чтобы изменить язык интерфейса.

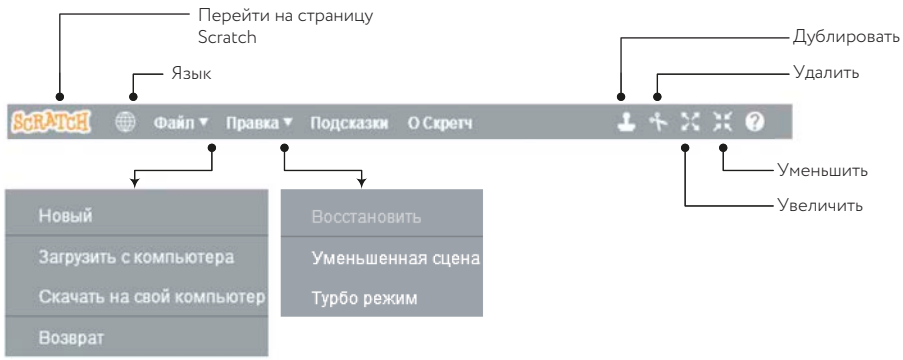


Рис. 1.14. Панель инструментов Scratch

При помощи меню **Файл** вы можете создавать новые проекты, загружать (**Открыть**) уже существующие с вашего компьютера, скачать

(**Сохранить**) текущий проект на ваш компьютер или отменить (**Возврат**) все ваши изменения в текущем проекте.

Файлы проектов в среде Scratch 2 имеют расширение *.sb2*. Поэтому их легко отличить от проектов, созданных в предыдущей версии (*.sb*).

В меню **Редактировать** кнопка **Восстановить** вернет последний блок, спрайт, костюм или звук, который вы удалили. Функция **Макет малой сцены** уменьшает **Сцену** и оставляет больше места для поля скриптов. Выбрав режим **Турбо**, вы увеличите скорость некоторых блоков. Например, выполнение блока **идти** 1000 раз в нормальном режиме займет около 70 секунд и около 0,2 секунды — в режиме турбо. Теперь, когда вы познакомились с содержимым панели инструментов Scratch, поговорим о встроенном графическом редакторе.

Графический редактор

Вы можете использовать графический редактор (рис. 1.15) для создания или редактирования костюмов и фонов (подойдет и любая другая программа, которая вам по душе). Если вы хотите узнать больше о графическом редакторе Scratch, загляните в файл *ScratchPaintEditor.pdf* (его можно найти среди онлайн-материалов на сайте *http://nostarch.com/learnscratch/*).

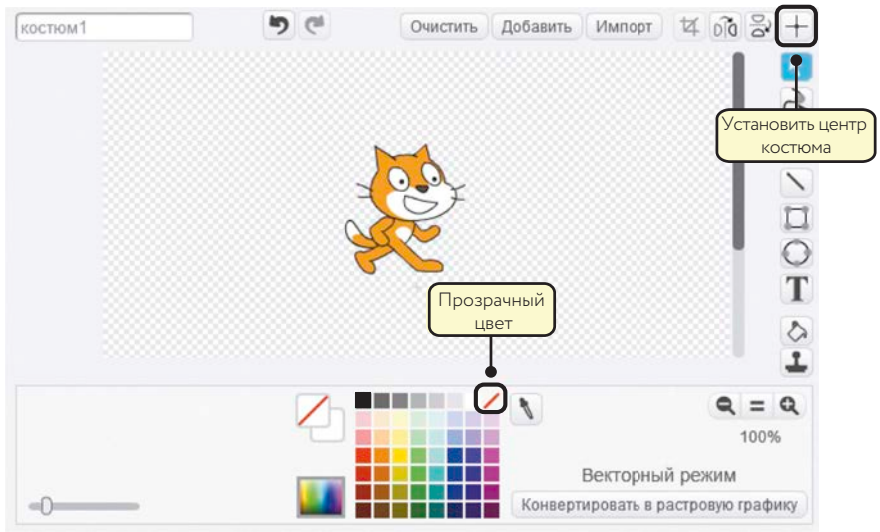


Рис. 1.15. Графический редактор Scratch

Есть две важные функции, о которых вам нужно узнать уже сейчас: обозначение центра изображения и прозрачные цвета. Я объясню, как они работают, чуть ниже.

Обозначение центра изображения

Когда вы даете спрайту команду повернуться (направо или налево), он поворачивается относительно какой-то опорной точки — центра его костюма. Кнопка **Установить центр** (в верхнем правом углу графического редактора) дает вам возможность выбрать эту точку. Нажав ее, вы увидите на поле для рисования две оси, как показано на рис. 1.16. Центральная точка находится в месте пересечения осей. Чтобы подвинуть центр костюма, просто перетащите ее на новую позицию. Чтобы скрыть оси, нажмите кнопку еще раз.

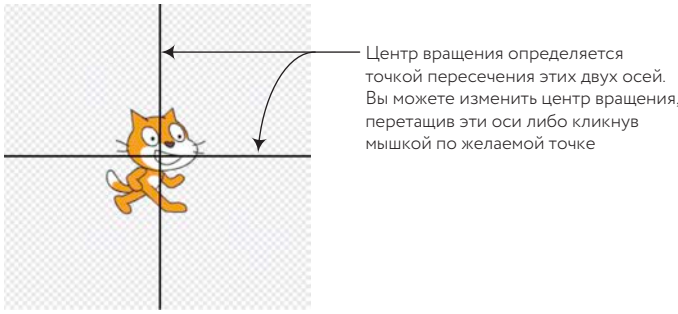
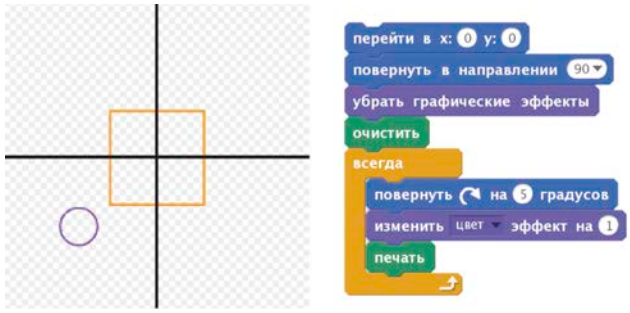


Рис. 1.16. Изменение центра костюма после нажатия кнопки **Установить центр**

УПРАЖНЕНИЕ 1.10

RotationCenter.sb2

Откройте файл *RotationCenter.sb2* и запустите его. Это программа с одним спрайтом в костюме и скриптом, показанным ниже. Центр костюма установлен в центре квадрата. Запустите скрипт и обратите внимание на узор. Затем отредактируйте костюм, установив его центр в середине кружка, и снова запустите скрипт, чтобы посмотреть, как изменится картинка.



Установка прозрачных цветов

Когда два изображения накладываются друг на друга, то, которое оказывается сверху, перекрывает часть нижнего. Точно так же спрайты закрывают часть **Сцены**. Если вы хотите видеть, как выглядит **Сцена** за каким-то изображением, нужно воспользоваться графическим редактором, чтобы сделать хотя бы часть изображения *прозрачным*, как пингвин справа на рис. 1.17.

Кликните на цветовой палитре по квадратику с красной диагональной линией и рисуйте «прозрачным» цветом, чтобы сделать что-то невидимым. Представьте себе, что это знак «Цвета нет», что-то вроде знака «Не курить» с красной линией, перечеркивающей сигарету.

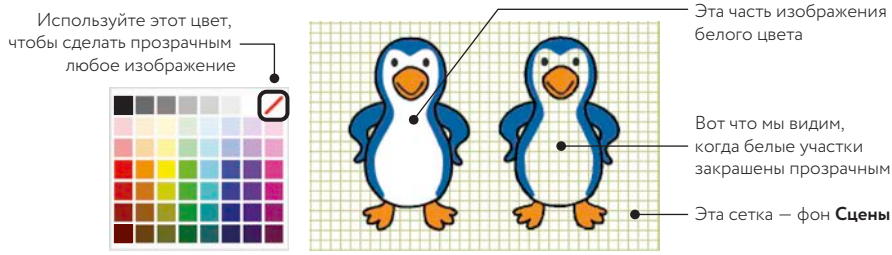


Рис. 1.17. Вы можете сделать прозрачной любую часть изображения, залив ее «прозрачным» цветом

Теперь, когда вы разобрались с интерфейсом Scratch, мы с толком используем эти знания и сделаем кое-что любопытное. Закатайте рукава и приготовьтесь: мы создаем игру!

Ваша первая игра в Scratch

Pong.sb2
Pong_NoCode.sb2

В этом разделе вы создадите свою компьютерную игру, в которой пользователю нужно будет передвигать ракетку, чтобы не дать прыгающему мячику удариться об пол. Она основана на классической аркадной игре Pong. Пользовательский интерфейс показан на рис. 1.18.

Как показано на рисунке, мяч начинает движение в верхней части **Сцены** и перемещается вниз под случайным углом, отскакивая от краев **Сцены**. Игрок перемещает ракетку по горизонтали (при помощи мыши), чтобы отбить мяч обратно наверх. Если мяч коснется нижней границы **Сцены**, игра окончена. Мы будем создавать игру пошагово, и первым делом нужно открыть новый проект. Выберите **Файл — Новый**, чтобы начать новый проект Scratch. Затем удалите спрайт-кота, кликнув по нему правой кнопкой мыши и выбрав **Удалить** из выпадающего меню.

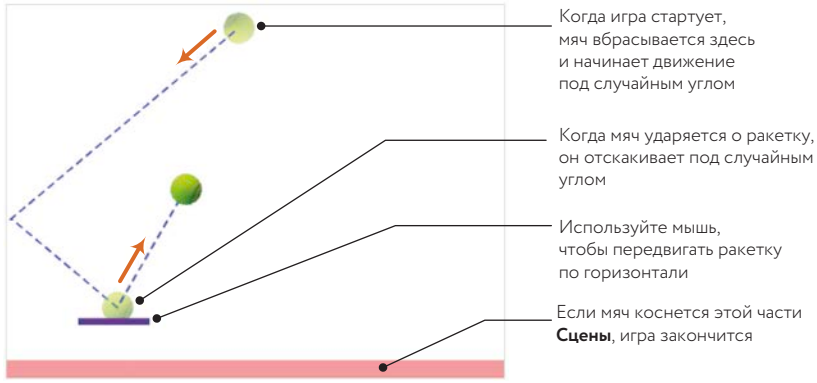


Рис. 1.18. Наша игра на экране

Шаг 1: подготовка фона

Чтобы определить, когда мяч попадает мимо ракетки, мы обозначим нижнюю границу **Сцены** определенным цветом и используем блок **дотронуться до цвета ?** (из раздела **Сенсоры**), чтобы определять, когда мяч будет касаться этого цвета. Сейчас у нас белый фон, и мы можем нарисовать у нижней границы **Сцены** тонкий цветной прямоугольник, как показано на рис. 1.19.

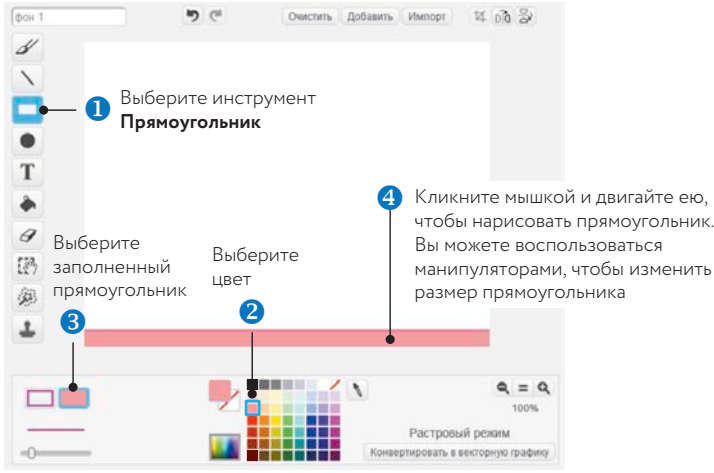


Рис. 1.19. Пошаговый процесс рисования прямоугольника внизу фона **Сцены**

Кликните по иконке **Сцены**, чтобы выбрать ее, а затем идите в закладку **Фон**. Повторите шаги, показанные на рис. 1.19, чтобы нарисовать тонкий прямоугольник внизу фона **Сцены**.

Шаг 2: добавляем ракетку и мяч

Нажмите кнопку **Нарисовать новый спрайт** наверху списка спрайтов, чтобы добавить в свой проект спрайт Ракетка (Paddle). Поскольку ракетка — просто узкий короткий прямоугольник, повторите шаг 1, чтобы нарисовать нужную фигуру, как на рис. 1.18. Раскрасьте ее в любой цвет, какой вам нравится, и установите центр приблизительно посередине прямоугольника. Затем дайте спрайту имя, которое объясняло бы, что он собой представляет. Я назвал его Ракеткой. А также кликните по изображению ракетки на **Сцене** и передвиньте его так, чтобы координата *y* была около –120.

Теперь у нас есть ракетка, но пока не хватает мяча, который скакал бы вокруг. Нажмите **Выбрать спрайт из библиотеки** наверху в списке спрайтов, чтобы импортировать спрайт.

В появившемся окне выберите категорию **Предметы**, затем изображение теннисного мяча и добавьте его в ваш проект. Переименуйте спрайт в Мяч.

Прежде чем вы начнете работать над скриптами для игры, выберите **Файл — Скачать на свой компьютер**, чтобы сохранить то, что вы уже сделали. В появившемся диалоговом окне выберите папку, куда вы хотите сохранить свою работу, назовите файл *Pong.sb2* и нажмите **Сохранить**. Если вы вошли на сайт Scratch под своим логином, вы можете сохранить свою работу в *облаке* (на сервере Scratch). В любом случае не забывайте сохранять файлы достаточно часто.

Теперь, когда у нас есть спрайты Ракетка и Мяч, **Сцена** должна выглядеть приблизительно как на рис. 1.18. Если на этом этапе у вас возникнут трудности, вы можете открыть файл *Pong_NoCode.sb2*, в котором есть всё, что мы только что сделали. Следующим шагом будет добавление скриптов, необходимых, чтобы игра заработала. Не переживайте из-за отдельных непонятных блоков, мы их обсудим позже. А пока сосредоточимся на том, чтобы научиться складывать из отдельных частей готовый проект.

Шаг 3: начать игру и заставить спрайты двигаться

Вам решать, как игроки могут начать новый раунд. Например, нажав кнопку, кликнув по спрайту на **Сцене** или даже хлопнув в ладоши или помахав рукой (если у вас есть веб-камера).

Зеленый флажок (расположенный над **Сценой**) — еще одна популярная функция, ею-то мы и воспользуемся.

Идея проста. Любой скрипт, начинающийся с блока-триггера **когда щелкнут по**  , начинает работать, как только вы нажмете эту кнопку. Флажок становится ярко-зеленым и остается таким до тех пор, пока скрипт не завершит работу. Чтобы увидеть это в действии, создайте скрипт для спрайта Ракетка, как показано на рис. 1.20.

Рис. 1.20. Скрипт для спрайта Ракетка

После того как игрок кликает по зеленому флажку ❶, блок **перейти в x: y:** ❷ устанавливает вертикальную позицию ракетки на –120, просто на тот случай, если вы до этого мышкой убрали ее оттуда. Ракетка должна парить прямо над розовым прямоугольником внизу **Сцены**, так что если ваш прямоугольник толще, измените координаты ракетки на те, которые вам подходят лучше.

Затем в скрипте используется блок **всегда** ❸, чтобы постоянно отслеживать позицию мышки. Мы будем двигать ракетку туда-сюда за счет того, что укажем в качестве ее позиции по оси *x* позицию мыши ❹. Запустите скрипт (кликнув по зеленому флажку) и попробуйте двигать мышь горизонтально. Ракетка должна следовать за ней.

Кликните по значку **Стоп** рядом с зеленым флажком, чтобы остановить скрипт.

Скрипт для спрайта-мяча немного длиннее, чем предыдущий, так что я разобью его на простые части. Мяч должен начинать двигаться, как только мы кликаем по зеленому флажку, поэтому первым делом добавьте спрайту-мячу скрипт с рис. 1.21.

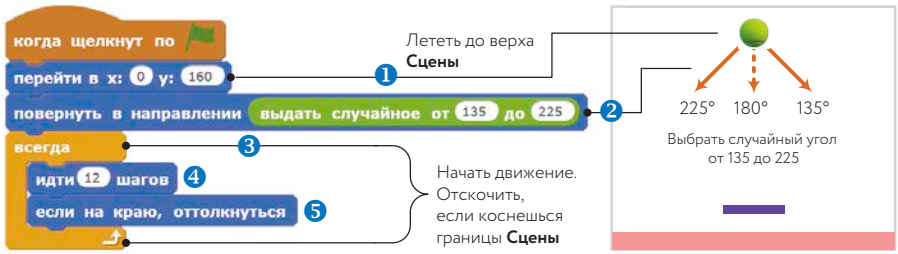


Рис. 1.21. Первая часть скрипта для мяча

Сначала мы перемещаем мяч наверх **Сцены** ❶ и заставляем его лететь вниз под случайным углом, используя блок **выбрать случайный угол** ❷ (из раздела **Операторы**). Затем скрипт использует блок **всегда** ❸, чтобы мяч перемещался по **Сцене** и отпрыгивал ❹ от ее краев. Используйте

зеленый флажок, чтобы проверить, что у вас получилось. Мяч должен передвигаться зигзагами, а ракетка — следовать за мышкой.

УПРАЖНЕНИЕ 1.11

Замените цифру 12 внутри блока **идти 12 шагов** на другие значения, запустите скрипт и посмотрите, что получится. Это покажет вам, как сделать игру для пользователя сложнее или легче. Когда закончите — нажмите кнопку **Стоп**.

Теперь пора добавить самое веселое — блоки, которые заставят мяч отскакивать от ракетки. Мы можем изменить блок **всегда**, который только что создали, так, что мяч будет лететь вверх, отскочив от ракетки, как показано на рис. 1.22.

Рис. 1.22. Добавление блока, чтобы мяч отскакивал вверх

Когда мяч и ракетка соприкасаются, мы даем мячу команду повернуть в случайном направлении от –30 до 30. Когда блок **всегда** зайдет на второй круг, он начнет выполнять блок **идти**, который теперь будет заставлять мяч лететь вверх.

Кликните снова по зеленому флажку, чтобы протестировать эту часть игры. Когда вы удостоверитесь, что мяч отскакивает от ракетки, как и предполагалось, нажмите **Стоп**.

Теперь единственный недостающий фрагмент — код, который останавливает игру, если мяч касается нижней границы **Сцены**. Добавьте спрайту-мячу скрипт, как показано на рис. 1.23, либо непосредственно перед, либо сразу после блока **если** (рис. 1.22). Блок **касается цвета ?** вы найдете в разделе **Сенсоры**, блок **стоп** — в разделе **Управление**.

Когда вы кликнете мышью по цветному квадрату внутри блока **касается цвета ?**, стрелка курсора изменится на руку. Когда вы переместите курсор и кликнете по розовому прямоугольнику внизу **Сцены**, этот квадрат внутри блока станет такого же цвета. Блок **стоп все** делает именно

то, о чем говорит его название: останавливает все работающие скрипты. Ни Мяч, ни Ракетка не будут исключением.

Рис. 1.23. Блоки для окончания игры

Теперь наша простейшая игра Pong полностью готова к работе. Кликните по зеленому флажку и сыграйте несколько раз, чтобы протестировать ее. Надеюсь, после того как вы увидели, что можете самостоятельно создать целую игру, сделав так мало, вы согласитесь, что Scratch — что-то невероятное!

Шаг 4: приправим блюдо звуком

Играть, конечно, гораздо веселее, когда есть звук. Добавим один завершающий штрих: звук, который будет раздаваться каждый раз, когда мы отбиваем мяч. Дважды кликните по мячу на **Сцене**, чтобы выбрать его, а затем зайдите в закладку **Звуки**. Кликните по кнопке **Выбрать звук из библиотеки**, чтобы добавить звук спрайту-мячу. В появившемся диалоговом окне выберите категорию **Эффекты**, затем **pop** и нажмите **OK**, чтобы добавить звук в закладку **Звуки**. Потом вернитесь в закладку **Скрипты** и вставьте блок **играть звук** (из раздела **Звуки**), как показано на рис. 1.24.

Рис. 1.24. Проигрывать звук, когда мяч касается ракетки

Запустите игру еще раз. На этот раз вы будете слышать короткий звук каждый раз, когда мяч касается ракетки.

Поздравляю! Ваша игра готова (если, конечно, вы не хотите добавить еще каких-нибудь функций). Вы только что написали свою первую программу в Scratch.

Если вы хотите еще поэкспериментировать, попробуйте продублировать спрайт Мяч, чтобы у вас было два (или более) мяча в игре, и посмотрите, что из этого получится!

В следующем разделе я познакомлю вас с разными типами блоков в Scratch. Вы узнаете многое о том, как работают эти блоки, а пока мы вкратце пройдемся по ним.

Блоки Scratch: обзор

Из этого раздела вы узнаете о разных блоках, с которыми работает Scratch, выучите их названия и области применения. Наша цель — разобраться в терминологии, которая встретится вам в следующих главах. Вы можете возвращаться к этому разделу позже, чтобы освежить что-то в памяти.

Как показано на рис. 1.25, Scratch использует четыре типа блоков: командные, функции, триггеры и контрольные. У *командных* и *контрольных блоков* (их еще называют *стеками*) снизу есть небольшой выступ и/или небольшая выемка наверху. Такие блоки можно соединять между собой в стопки. У *блоков-триггеров,* которые также называют *шляпами*, закругленный верх, потому что они помещаются наверх стопки блоков. Триггеры соединяют события в скрипты. Они ждут события — например нажатия кнопки или клика мышкой — и запускают расположенные под ними блоки, когда оно происходит. Например, все скрипты, начинающиеся с блока **когда щелкнут по**  , запускаются, если пользователь кликает по зеленому флажку.

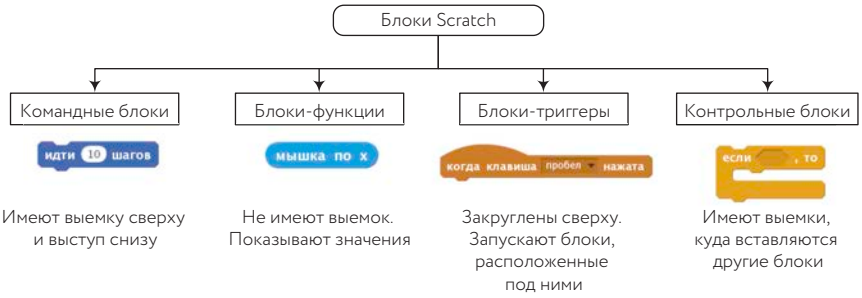


Рис. 1.25. Четыре типа блоков, используемых в Scratch

У *блоков-функций* (их мы называем *репортерами*) нет выемок или выступов. Они не могут самостоятельно сформировать слой скрипта, а используются как вставки в другие блоки. Форма этих блоков обозначает тип данных, которые в них содержатся. Так, например, блоки с закругленными краями содержат цифры или дополнительные условия, а блоки с заостренными краями показывают, истинно что-то или ложно. Это проиллюстрировано на рис. 1.26.

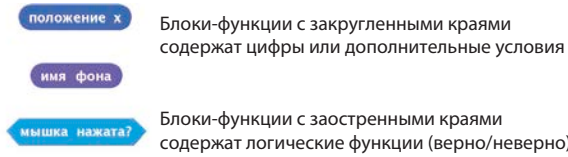


Рис. 1.26. Форма блока-функции говорит о типе данных, которые он содержит

Рядом с некоторыми блоками-функциями находятся чекбоксы. Если вы поставите там галочку, на **Сцене** появится *монитор*, который будет показывать актуальное значение репортера. Выберите спрайт и поставьте галочку в чекбоксе блока **положение x** (из раздела **Движение**). Затем передвигайте спрайт по всей **Сцене** и следите за этим монитором. Его показания должны меняться, когда вы двигаете спрайт туда-сюда.

Арифметические операторы и функции

Посмотрим на арифметические операторы и функции, поддерживаемые средой Scratch. Теперь, если вы потеряете свой калькулятор, не стоит волноваться! Вы сможете смастерить себе калькулятор в Scratch при помощи блоков из раздела **Операторы**, с которыми вы познакомитесь в этом разделе.

Арифметические операторы

Scratch поддерживает четыре основных арифметических действия: сложение (+), вычитание (–), умножение (*) и деление (/). Блоки, которые используются для осуществления этих операций, называются *операторами*. Они показаны на рис. 1.27. Поскольку они производят числа, их можно использовать как вставки в любой блок, принимающий цифры.

Рис. 1.27. Арифметические операторы в Scratch

Scratch также поддерживает оператор модуля (**модуль**), который показывает остаток при делении одного числа на другое. Например, **10 модуль 3** дает 1, потому что остаток при делении 10 на 3 — это 1.

Оператор модуля часто используется для проверки делимости целого числа на другое (меньшее). Модуль 0 говорит о том, что большее число делится на меньшее. У вас не появилось идеи таким образом проверить, четное число или нечетное? Еще один полезный оператор, поддерживаемый Scratch, — **округлить**, который округляет десятичные дроби до ближайшего целого числа. Например, **округлить (3.1*)** = 3, **округлить (3.5)** = 4 и **округлить (3.6)** = 4.

* В качестве десятичного разделителя в Scratch используется точка. *Прим. ред.*

Случайные числа

Когда вы будете активнее программировать, вам, возможно, в какой-то момент понадобится сгенерировать случайное число, особенно если вы станете создавать игры и симуляции. Scratch именно в этих целях предлагает вам блок **выдать случайное**. Он выдает случайные числа. Его два редактируемых белых поля позволяют задать пределы этих чисел, а Scratch будет лишь выбирать числа между указанными двумя числами (включительно). Таблица 1.1 показывает несколько примеров использования этого блока.

Таблица 1.1. Примеры использования блока **выдать случайное**

Пример	Возможный результат
	{0, 1}
	{0, 1, 2, 3, … , 10}
	{–2, –1, 0, 1, 2}
	{0, 10, 20, 30, … , 100}
	{0, 0.1, 0.15, 0.267, 0.3894, … , 1.0}
	{0, 0.01, 0.12, 0.34, 0.58, … , 1.0}

Результаты работы блока **выдать случайное от 0 до 1** *и блока* **выдать случайное от 0 до 1.0** *будут разными. В первом случае вы получите или 0, или 1, а во втором — десятичную дробь в промежутке от 0 до 1. Если хотя бы одно из чисел, введенных в блок* **выдать случайное**, *содержит десятичную дробь, результат тоже будет десятичной дробью.*

Математические функции

Scratch также поддерживает многие математические функции. Блок **квадратный корень от** объединяет 14 математических функций, которые можно выбрать из выпадающего меню, в том числе квадратный корень, тригонометрические, логарифмические и экспоненциальные функции. Более подробную информацию о них вы найдете в файле *MathematicalFunctions.pdf*.

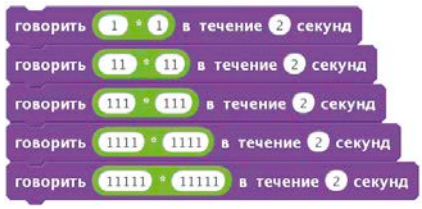
Итоги

В этой главе мы подробно познакомились с языком Scratch и его средой программирования. Вы узнали о различных элементах пользовательского интерфейса и даже создали игру! Мы также познакомились с математическими операторами и функциями Scratch.

Теперь вы получили всю необходимую базовую информацию, чтобы создать в Scratch мощные скрипты. Но это только первый шаг на пути к написанию потрясающих программ. В следующих главах мы копнем глубже и посмотрим, как при помощи Scratch развить свои навыки программирования.

Задания

1. Запишите результат каждого блока этого скрипта. Прослеживается ли какая-то закономерность?



2. Есть ли закономерность в произведении 9 × 9, 99 × 99, 999 × 999,... и т. д.?

   Используйте команду **сказать**, чтобы узнать результат этих действий и проверить свои ответы.

3. Заполните эту таблицу, вписав значение каждого выражения.

Выражение	Значение
3 + (2 × 5)	
(10 / 2) – 3	
7 + (8 × 2) – 4	

Выражение	Значение
(2 + 3) × 4	
5 + (2 × (7 – 4))	
(11 – 5) × (2 + 1) / 2	
5 × (5 + 4) – 2 × (1 + 3)	
(6 + 12) модуль 4	
3 × (13 модуль 3)	
5 + (17 модуль 5) – 3	

Теперь воспользуйтесь командой **сказать** и соответствующими блоками из раздела **Операторы**, чтобы проверить свои ответы.

4. Оцените следующие выражения Scratch при помощи бумаги и карандаша. Пусть x = 2 и y = 4:
   а) 6 * x;
   б) 2 * x + 4 * y;
   в) x * x;
   г) y + 4 / x * x;
   д) y * y / 2 * x + 2.

5. Используйте команду **сказать** с соответствующими блоками из раздела **Операторы**, чтобы посчитать следующее:
   а) квадратный корень из 32;
   б) синус 30°;
   в) косинус 60°;
   г) результат округления 99,459.

6. Создайте блок-функцию, который подсчитывает среднее арифметическое следующих трех чисел: 90, 95 и 98. Выведите результат на экран при помощи блока **сказать**.

7. Создайте блок-функцию, который переводит 60° по Фаренгейту в градусы Цельсия. (Подсказка: $C = (5/9) \times (F - 32)$.)

8. Создайте блок-функцию, который подсчитывает площадь трапеции высотой в 4/6 м и основаниями длиной в 5/9 и 22/9 м. (Подсказка: $A = 0{,}5 \times (b_1 + b_2) \times h$, где h — это высота, а b_1 и b_2 — длины оснований.)

9. Создайте блок-функцию, который подсчитывает силу, необходимую для ускорения машины весом в 2000 кг до 3 м/с. (Подсказка: сила = масса × ускорение.)

10. Стоимость электроэнергии — 3 рубля за кВт·ч. Создайте блок-функцию, который подсчитывает расходы на использование кондиционера

мощностью 1500 Вт на протяжении 2 часов. (Подсказка: энергия = мощность × время.)

11. При помощи простого математического трюка вы можете использовать оператор **округлить**, чтобы округлить число до определенного разряда десятичной дроби. Например, вы можете округлить число *5,3567* до десятых (то есть первой цифры справа от запятой), используя следующие шаги:
    а) *5,3567* × 10 = *53,567* (умножить число на 10);
    б) округлить (*53,567*) = *54* (округлить ответ из шага а);
    в) *54*/10 = *5,4* (разделить число из шага б на 10).

    Какие изменения вам нужно внести в перечисленные шаги, чтобы округлить число до сотых (второй цифры после запятой)? Создайте блок-функцию, который округляет *5,3567* до десятых (или сотых), и выведите результат на экран с помощью блока **сказать**.

2

ДВИЖЕНИЕ И РИСОВАНИЕ

Вы уже ориентируетесь в интерфейсе. Теперь можно использовать больше инструментов программирования на Scratch. Вот что вам предстоит сделать в этой главе:

- познакомиться с командами разделов **Движение** и **Перо**;
- анимировать спрайты и передвигать их по **Сцене**;
- рисовать геометрические узоры и создавать игры;
- узнать, почему клонирование спрайтов — такой ценный инструмент.

Самое время надеть шляпу художника и нырнуть с головой в мир компьютерной графики!

Использование команд движения

Если вы хотите делать игры или другие анимированные программы, чтобы перемещать спрайты по **Сцене**, нужно использовать блоки из раздела **Движение**. Более того, вам нужно будет давать спрайтам команду переместиться в конкретную точку на **Сцене** или повернуть в определенном направлении.

Абсолютное движение

На рис. 1.4 мы видели на **Сцене** прямоугольную систему координат 480 × 360 с центром в точке (0, 0). В Scratch есть четыре команды абсолютного

движения (**перейти в**, **плыть в**, **установить x в** и **установить y в**), которые дают вам возможность сообщить спрайту, куда ему переместиться.

Если вы хотите узнать больше об этих или любых других блоках, используйте окошко **Подсказка**, *которое находится справа от панели* **Скрипты**. *Если вы не видите его, кликните по знаку вопроса в верхнем правом углу редактора проектов.*

Чтобы продемонстрировать, как работают эти команды, представим, что вы хотите, чтобы спрайт Ракета на рис. 2.1 попал в имеющую форму звезды цель, расположенную в точке (200, 150). Самый очевидный способ — использовать блок **перейти**, как показано на рисунке справа. Ось *x* показывает, куда спрайту нужно переместиться по горизонтали, а ось *y* — по вертикали.

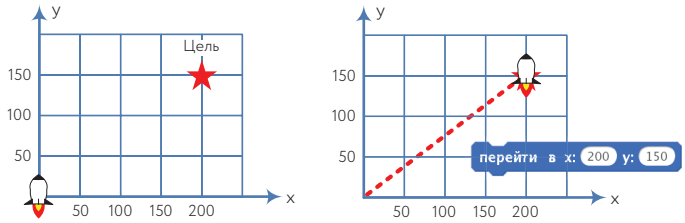


Рис. 2.1. При помощи блока **перейти** *вы можете переместить спрайт в любую точку на* **Сцене**

Ракета не повернется носом к цели, но будет двигаться по невидимой прямой, соединяющей ее исходное местоположение, точку (0, 0), с точкой (200, 150). Вы можете заставить ракету замедлить движение, если используете команду **плыть к**. Она практически идентична **перейти**, но позволяет устанавливать время, которое понадобится ракете, чтобы достичь цели.

Другой способ попасть в цель — независимо друг от друга изменить координаты *x* и *y* спрайта-ракеты при помощи блоков **установить x в** и **установить y в**, как показано на рис. 2.2. Помните, как вы использовали блок **установить x в**, когда делали игру Pong (см. рис. 1.20)?

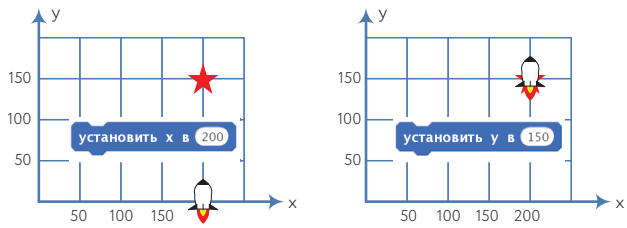
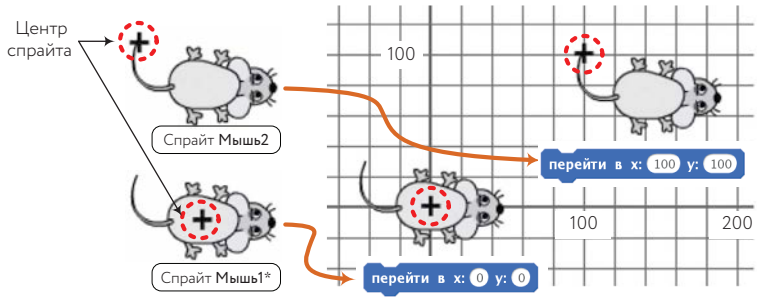


Рис. 2.2. Вы можете независимо изменить координаты x и y спрайта

Вы всегда можете видеть текущие координаты спрайта в правом верхнем углу поля скриптов. Если вы хотите, чтобы эта информация отображалась на **Сцене**, используйте блоки-репортеры **положение x** и **положение y**. Поставьте галочки в чекбоксах около них, чтобы увидеть их значения на **Сцене**.

Команды движения работают относительно центра спрайта, который вы можете установить в графическом редакторе. Например, если отправить спрайт в точку (100, 100), он переместится так, что его центр окажется в точке (100, 100), как показано на рис. 2.3. Поэтому, если вы рисуете или импортируете костюм для спрайта, который планируете перемещать, обратите внимание на его центр!

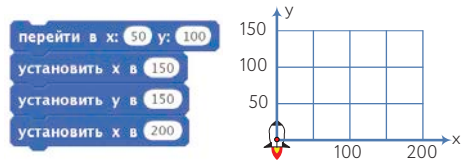


* В библиотеке спрайтов Mouse1.

Рис. 2.3. Команды движения относительно центра спрайта

УПРАЖНЕНИЕ 2.1

Список координат спрайта-ракеты после выполнения каждой из команд приведенного ниже скрипта.



Относительное движение

А теперь посмотрите внимательно на сетку координат на рис. 2.4, где показаны другие спрайты Ракета и Цель. На этот раз вы не видите координат, поэтому точное расположение спрайтов вам неизвестно. Если бы вам нужно было объяснить ракете, как попасть в цель, вы могли бы сказать: «Сделай три шага, потом поверни направо и сделай еще два шага».

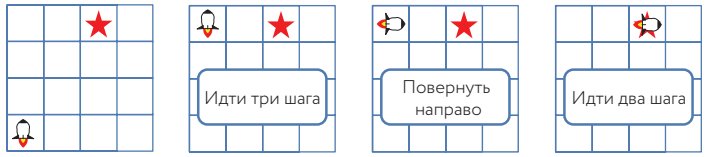


Рис. 2.4. Вы можете перемещать спрайт по **Сцене**, *используя команды относительного движения*

Идти и **повернуть** — *команды относительного движения*. Например, первая команда «идти» сверху заставляет ракету двигаться вверх, а вторая отправляет ее направо. Движение зависит от текущего *направления* спрайта. На рис. 2.5 изображены направления движения в Scratch.

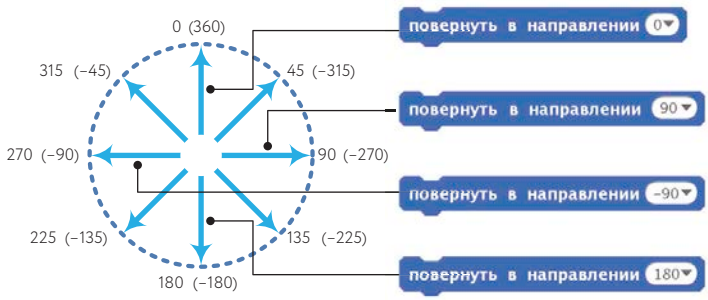


Рис. 2.5. В Scratch 0 — вверх, 90 — вправо, 180 — вниз и –90 — влево

Вы можете повернуть спрайт к конкретной цели (курс) с помощью команды **повернуть в направлении**. Чтобы выбрать направление, кликните по указывающей вниз стрелке и выберите нужный вариант из выпадающего меню. Чтобы выбрать другое направление, вставьте нужное значение в белое редактируемое поле. Можно даже использовать отрицательные значения! (Например, если написать 45 или –315, спрайт в обоих случаях повернет на северо-восток.)

Актуальное направление спрайта отражено в поле информации о нем. Также можете поставить галочку в чекбоксе напротив блока **направление** *(в разделе* **Движение***), чтобы увидеть направление на* **Сцене**.

Теперь, когда вы знаете, как в Scratch работает направление, посмотрим на команды относительного движения (**идти**, **изменить x на**, **изменить y на** и **повернуть**). Начнем с команд **идти** и **повернуть**, которые работают с учетом актуального положения спрайта, как показано на рис. 2.6.

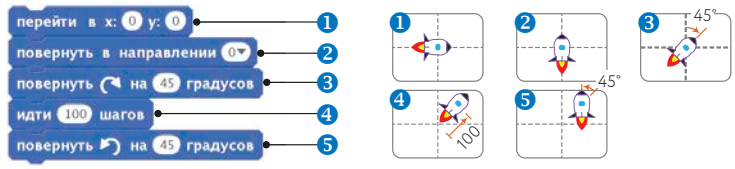


Рис. 2.6. Простой скрипт, который демонстрирует использование команд **идти** *и* **повернуть**

Во-первых, блок **перейти** ❶ двигает Ракету так, что ее центр оказывается совмещенным с центром **Сцены**. Второй командный блок ❷ направляет спрайт вверх, а третий ❸ поворачивает спрайт на 45° по часовой стрелке. Затем спрайт перемещается на 100 шагов ❹ по своему текущему направлению, прежде чем повернуть на 45° по часовой стрелке ❺, чтобы он остановился, будучи направленным вверх.

НАПРАВЛЕНИЯ И КОСТЮМЫ

Команда **повернуть в направлении** совершенно не в курсе костюма спрайта. Возьмем для примера два спрайта с рисунка слева. При помощи графического редактора мы нарисовали костюм птицы ориентированным направо, а костюм насекомого — вверх. Как вы думаете, что будет, если использовать команду **повернуть в направлении 90** (повернуться направо) для каждого из спрайтов? Можно предположить, что насекомое станет смотреть в правую сторону, но на самом деле ни один из спрайтов не пошевелится. Хотя 90° обозначается как «право», на деле это направление относится к *исходной ориентации костюма* в графическом редакторе. Так что если в нем насекомое ориентировано наверх, оно по-прежнему будет смотреть вверх после того, как вы скажете ему повернуться на 90°. Если вы хотите, чтобы ваш спрайт отреагировал на команду, как показано на рис. 2.5, нужно в графическом редакторе нарисовать костюм спрайта, ориентированный вправо (как костюм-птица на рисунке сверху).

Иногда вам, возможно, понадобится лишь переместить спрайт из его текущей позиции по горизонтали или вертикали. И здесь приходят на помощь блоки **изменить x на** и **изменить y на**. Скрипт на рис. 2.7 демонстрирует использование этих блоков.

После того как спрайт-ракета передвинулся в центр **Сцены**, первая команда **изменить x на 50** ❶ добавляет 50 к его координате *x* и он отправляется еще на 50 шагов направо. Следующая команда ❷, **изменить y на 50**, меняет на 50 координату *y*, в результате спрайт перемещается еще на 50 шагов вверх. Другие команды работают так же. Попробуйте проследить движение спрайта на рис. 2.7, чтобы найти его окончательный пункт назначения.

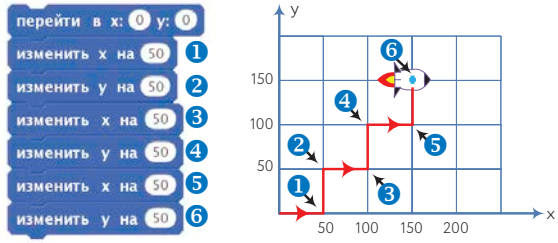
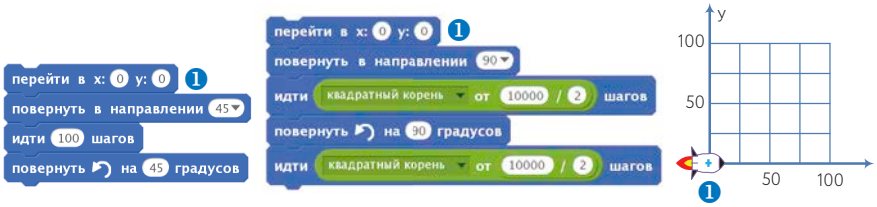


Рис. 2.7. Проведите ваш спрайт извилистой тропой при помощи **изменить x на** *и* **изменить y на**

УПРАЖНЕНИЕ 2.2

Найдите конечную позицию ракеты (*x*, *y*) после выполнения ею каждого из скриптов, показанных ниже. Какую математическую теорему можно использовать, чтобы доказать, что эти два скрипта эквивалентны?



Другие команды движения

TennisBallChaser .sb2

Осталось познакомиться всего с четырьмя командами движения: **повернуть к**; второй тип блока **перейти**; **если на краю, оттолкнуться** и **стиль вращения**. Вы уже знаете о стилях вращения и видели в действии команду **если на краю, оттолкнуться** в главе 1 (см. рис. 1.13). Чтобы посмотреть, как работают остальные две команды, создадим простую программу, в которой кот будет гоняться за теннисным мячиком, как показано на рис. 2.8.

Рис. 2.8. Программируем кота, чтобы он бегал за мячиком

* В библиотеке спрайтов Cat2 и Ball.

Как вы видите, в программе два спрайта (мы назвали их Кот и Мяч*) и два скрипта. Когда вы кликнете по зеленому флажку, спрайт Мяч будет следовать за курсором мыши. Спрайт Кот постоянно направлен в сторону мяча и движется за ним при помощи команды **плыть**. Попробуйте самостоятельно создать эту программу, чтобы посмотреть, как она работает. Блок **всегда** вы найдете в разделе **Управление**, а блоки **мышка по x** и **мышка по y** — в разделе **Сенсоры**. Готовая программа находится в файле *TennisBallChaser.sb2*. Ниже мы подробно разберем раздел **Перо** и научимся заставлять спрайты оставлять видимые следы своего передвижения.

Команды раздела Перо и программа Easy Draw

EasyDraw.sb2

Команды движения, которые вы использовали раньше, позволяют вам перемещать спрайт в любую точку на **Сцене**. Разве не было бы здорово увидеть путь, по которому перемещается ваш спрайт? Тут может помочь перо в Scratch.

У каждого спрайта есть невидимое перо, которое может быть либо поднято, либо опущено. Если оно опущено, спрайт по мере передвижения будет рисовать. В остальных случаях он передвигается, не оставляя следов. Команды из раздела **Перо** позволяют вам контролировать размеры пера, его цвет и тень.

УПРАЖНЕНИЕ 2.3

Откройте в Scratch окошко **Подсказки**, кликните по иконке с домиком, а затем по перу, чтобы получить краткое описание каждой команды **Пера**. Скрипты внизу показывают большую часть этих команд. Воссоздайте эти скрипты, запустите и опишите эффект от каждого из них. Не забудьте опустить перо спрайта вниз, прежде чем запустить скрипты. (Блок **повторить** вы найдете в разделе **Управление**.)

Давайте подробнее изучим некоторые команды пера и создадим простую программу для рисования картинок, передвигая и переворачивая спрайт на **Сцене** клавишами со стрелками. Одно нажатие на клавишу со стрелкой вверх (↑) переместит спрайт вперед на 10 шагов. Нажатие

клавиши со стрелкой вниз (↓) — на 10 шагов назад. Каждое нажатие клавиши со стрелкой вправо (→) повернет спрайт вправо на 10°, а нажатие клавиши со стрелкой влево (←) — на 10° влево.

Так, например, чтобы повернуть спрайт на 90°, как показано на рис. 2.9, вам нужно нажать клавиши со стрелкой направо или налево девять раз.

Сначала создайте новый проект в Scratch. Замените костюм кота на то, что точно показывает, когда спрайт указывает вверх, вниз, направо или налево. Хорошо подойдут костюмы beetle или cat2 (из раздела **Животные**), но вы можете выбрать любой понравившийся вам костюм. В закладке **Костюмы** нажмите кнопку **Выбрать костюм из библиотеки** и выберите подходящий костюм.

Рис. 2.9. Программа Easy Draw в действии

Теперь добавьте вашему спрайту все скрипты, показанные на рис. 2.10. Вы можете создать четыре блока **когда клавиша нажата** из блока **когда клавиша пробел нажата** в разделе **События**. Кликните по черной стрелочке в блоке и выберите клавишу со стрелкой, которая вам нужна.

Рис. 2.10. Скрипты для программы Easy Draw

Когда вы кликаете по зеленому флажку, спрайт начинает двигаться к центру **Сцены** ❶ и поворачивается вверх ❷. Затем устанавливаются цвет ❸ и размер ❹ пера, скрипт опускает перо вниз ❺, чтобы подготовиться к рисованию. После этого программа стирает со **Сцены** все старые рисунки ❻.

Чтобы очистить сцену и начать новый рисунок, нужно кликнуть по зеленому флажку. Используйте клавиши со стрелками, чтобы нарисовать любую форму, какую захотите. Как вы думаете, какую форму создаст последовательность ↑→↑→↑→...?

УПРАЖНЕНИЕ 2.4

Добавьте еще одну функцию, чтобы перо становилось шире, когда нажата клавиша *W*, и тоньше, когда нажата *N*. Подумайте о других способах усовершенствовать программу и попробуйте их внедрить.

Сила повторения

Пока наши программы были простыми, но когда вы начнете писать более длинные скрипты, вам будет нужно повторять одни и те же блоки несколько раз подряд. Дублирование скриптов может сделать программу длиннее и запутаннее, с ней станет труднее экспериментировать. Если вам, например, нужно изменить одну цифру, придется вносить изменения в каждую копию блока. Команда **повторить** из раздела **Управление** поможет вам избежать этой проблемы.

DrawSquare.sb2

Предположим, вы хотите нарисовать квадрат, как показано на рис. 2.11 (слева). Вы можете скомандовать спрайту следовать этим повторяющимся инструкциям.

1. Пройди некоторое расстояние и повернись на 90° против часовой стрелки.
2. Пройди некоторое расстояние и повернись на 90° против часовой стрелки.
3. Пройди некоторое расстояние и повернись на 90° против часовой стрелки.
4. Пройди некоторое расстояние и повернись на 90° против часовой стрелки.

Рис. 2.11. Квадрат (слева) и скрипт для рисования его (справа), использующий последовательность команд **идти** *и* **повернуть**

На рис. 2.11 вы также видите скрипт, выполняющий эти инструкции. Обратите внимание, что в нем четыре раза повторяются команды **идти 100 шагов** и **повернуть на 90 градусов**. Мы можем избежать использования одних и тех же блоков при помощи блока **повторить**, который запускает команды внутри себя столько раз, сколько вы ему скажете, как показано на рис. 2.12. Используя блок **повторить**, вы также можете сделать свои инструкции гораздо более легкими для понимания.

Рис. 2.12. Используем блок ***повторить****, чтобы нарисовать квадрат*

Квадрат, который вы нарисовали при помощи скрипта на рис. 2.11, зависит от того, куда ориентирован ваш спрайт в тот момент, когда вы запускаете скрипт. Эта концепция проиллюстрирована на рис. 2.13. После того как вы нарисовали квадрат, спрайт возвращается в свою исходную позицию.

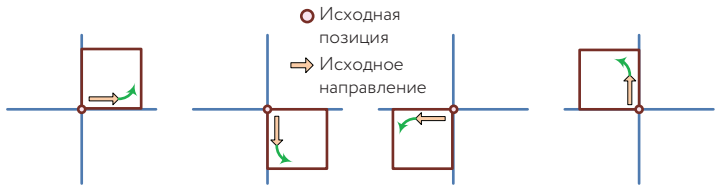


Рис. 2.13. Исходное направление спрайта меняет расположение квадрата

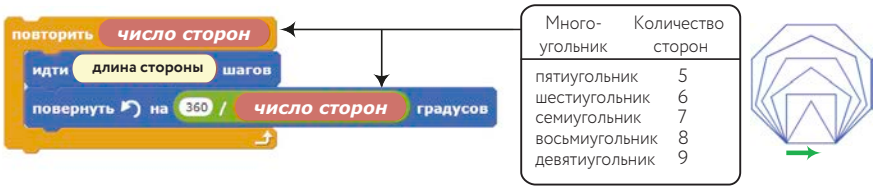
УПРАЖНЕНИЕ 2.5

Polygon.sb2

Вы можете модифицировать скрипт с рис. 2.12 так, чтобы он рисовал другие правильные многоугольники. Модифицированный скрипт имеет показанную ниже форму. Вместо «количества сторон» и «длины стороны» подставьте любые целые числа, чтобы обозначить желаемый многоугольник и контролировать его размеры. Рисунок показывает также шесть многоугольников с одинаковой длиной стороны, которые были нарисованы при помощи этого скрипта. Спрайт стартовал из своей исходной позиции в направлении, указанном зеленой стрелкой.

Откройте файл *Polygon.sb2* и запустите его, используя разные цифры для «количества сторон».

Что происходит, когда это число становится большим? Это должно дать вам представление о том, как рисовать круги.



Вращающиеся квадраты

RotatedSquares.sb2

Вы можете создавать потрясающие картины, повторяя один и тот же узор в определенной последовательности.

Вот, например, скрипт на рис. 2.14 создает симпатичный узор, вращая и рисуя квадрат 12 раз подряд (блоки, опускающие и поднимающие перо, здесь не показаны).

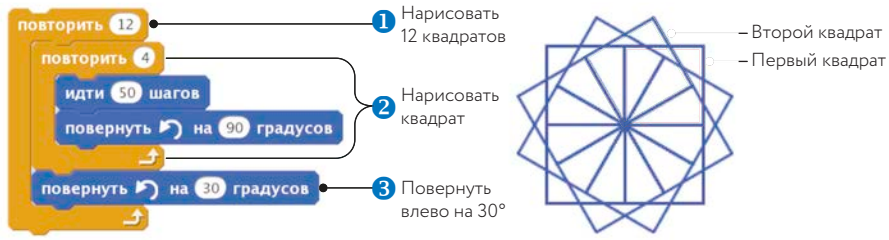


Рис. 2.14. Рисуем вращающийся квадрат

Внешний блок **повторить** ❶ работает 12 раз. За каждый повтор он рисует квадрат ❷, а потом делает поворот на 30° влево ❸, чтобы приготовиться рисовать следующий.

УПРАЖНЕНИЕ 2.6

Обратите внимание, что (12 повторов) × (30° на каждый повтор) = 360°. Как вы думаете, что произойдет, если вы измените в программе числа на 4 повтора по 90°? А как насчет 5 и 72°? Поэкспериментируйте с количеством повторов и значением угла поворота и посмотрите, что получится.

Исследуем печать

Windmill.sb2

Из предыдущего раздела вы узнали, как использовать блоки **повернуть** и **повторить**, чтобы превращать простые формы в сложные узоры. А что если вы захотите вращать более сложные формы? Вместо того чтобы рисовать простейшие формы при помощи команд **идти** и **повернуть**, вы можете создать новый костюм в графическом редакторе и использовать блок **печать**, чтобы нарисовать на **Сцене** несколько копий костюма. Чтобы проиллюстрировать эту технику, напишем программу, которая будет рисовать ветряную мельницу (рис. 2.15).

Мы нарисовали форму флага в графическом редакторе (см. рис. 2.15, слева) и использовали его в качестве костюма нашего спрайта. Мы установили центр костюма в основании флага так, чтобы вращать его вокруг этой точки.

Рис. 2.15. Команда **печать** *позволит вам легко создавать сложные геометрические узоры*

Скрипт для рисования ветряной мельницы показан в центре рис. 2.15.

Блок **повторить** выполняется восемь раз. Каждый раз на **Сцене** отпечатывается копия костюма, прежде чем спрайт повернется на 45° влево. Чтобы этот скрипт работал, вам нужно использовать блок **стиль вращения** с установленным значением **кругом**. Тогда флаг будет переворачиваться при вращении.

DrawingGeometricShapes.pdf в пакете дополнительных ресурсов (которые вы можете скачать с http://nostarch.com/learnscratch/) дает детальное представление о рисовании геометрических фигур, таких как прямоугольники, параллелограммы, ромбы, трапеции и многоугольники, а также научит вас создавать красивые рисунки из многоугольников.

УПРАЖНЕНИЕ 2.7

Блок **изменить цвет эффект на** (из раздела **Внешность**) позволит вам использовать такие графические эффекты, как цвет, завихрение или рыбий глаз. Откройте файл *Windmill.sb2* и добавьте эту команду в блок **повторить**. Поэкспериментируйте с другими графическими эффектами, чтобы получить классные узоры. Чтобы блок работал, цвет флага в графическом редакторе может быть любым, кроме черного.

Проекты Scratch

В этом разделе мы создадим две короткие программы, которые должны углубить ваше понимание уже знакомых разделов **Движение** и **Перо**. Фоны и спрайты есть в файлах проектов, поэтому мы можем сосредоточиться на скриптах, которые нужны для того, чтобы эти программы работали. Дополнительные материалы вы сможете найти в файле *BonusApplications.pdf* (*http://nostarch.com/learnscratch/*).

Некоторые из этих скриптов содержат блоки-команды, с которыми вы еще не сталкивались. Но не переживайте, если вы чего-то не поймете. В следующих главах я всё объясню.

Собери деньги

Money_NoCode.sb2

Наша первая программа — простая игра, в которой пользователь должен перемещать спрайт при помощи стрелок на клавиатуре, чтобы собрать максимальное количество мешков с золотом. Как показано на рис. 2.16, мешки появляются в случайных точках координатной сетки. Если игрок не успевает схватить мешок за три секунды, тот перемещается в другое место.

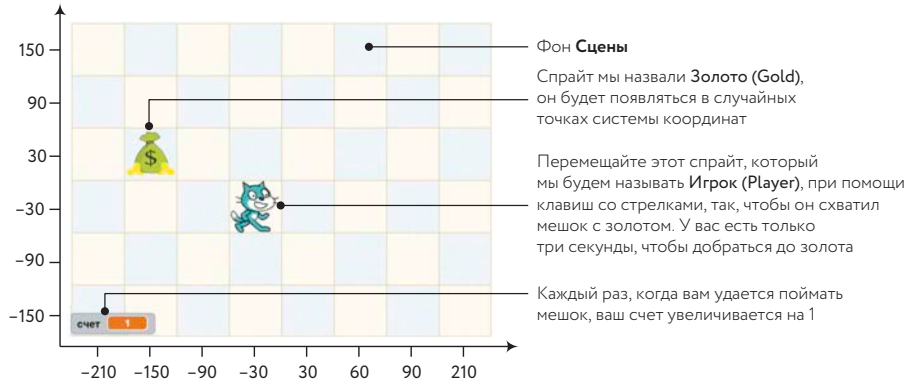


Рис. 2.16. Помогите коту схватить как можно больше мешков с золотом!

Откройте файл *Money_NoCode.sb2*. Здесь нет скриптов, но вы сейчас их создадите, а в этом файле есть еще кое-что полезное.

> ! *Оси координат, показанные на рис. 2.16, были добавлены, чтобы помочь вам понять цифры, которые используются в скриптах. Вернитесь к этому рисунку, когда нужно будет освежить в памяти картину передвижения спрайта.*

Начнем с написания скриптов для спрайта Игрок, как показано на рис. 2.17.

Когда игрок кликает по зеленому флажку, спрайт перемещается в точку (–30, –30) ❶ и поворачивается вправо ❷. Остальные четыре скрипта отвечают на использование клавиш со стрелками. Когда нажимается клавиша со стрелкой, соответствующий скрипт меняет направление спрайта ❸, проигрывает короткий звук (при помощи блока **играть звук** ❹ из раздела **Звуки**) и передвигает спрайт на 60 шагов ❺. Спрайт отскакивает от краев **Сцены** ❻, если это необходимо. Поскольку 60 шагов соответствуют одному квадрату в системе координат на рис. 2.16, каждый раз, когда вы нажимаете клавишу со стрелкой, спрайт перемещается на один квадрат.

Рис. 2.17. Скрипты для спрайта Игрок

Вы обратили внимание на то, что четыре скрипта, управляемых клавишами со стрелками, на рис. 2.17 практически одинаковые? В главе 4 вы узнаете, как избежать такого дублирования кода.

Протестируйте эту часть игры. У вас должно получаться передвигать игрока по всей **Сцене** при помощи клавиш со стрелками. Если все работает, мы перейдем к спрайту Золото, скрипт которого показан на рис. 2.18.

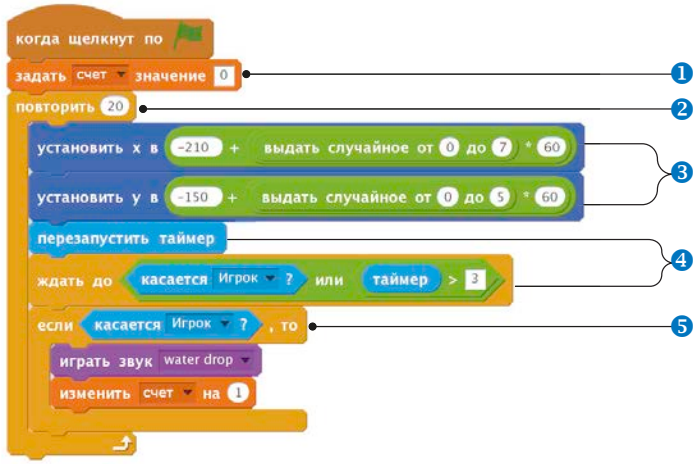


Рис. 2.18. Скрипт спрайта Золото

Как и спрайт Игрок, этот спрайт тоже начинает двигаться с момента нажатия на зеленый флажок. Он передвигает по **Сцене** мешок с золотом. Он также при помощи переменной под названием счет (которую

я создал для вас в разделе **Данные**) отслеживает, сколько мешков было собрано.

Переменные позволяют сохранять информацию, чтобы использовать ее позже в наших программах. Вы узнаете все о переменных в главе 5.

Поскольку игра только началась и пока нет никаких мешков, мы установим счет 0 ❶. Затем мы запустим цикл, который будет повторяться 20 раз ❷, чтобы показать игроку в общей сложности 20 мешков (или можете выбрать любое число). После каждого цикла мешок с золотом будет появляться в другом случайном месте ❸, давать игроку немного времени, чтобы схватить его ❹, и увеличивать счет, если игроку это удалось ❺. Нам нужно, чтобы мешок появлялся в одном из 48 квадратов сцены в случайном порядке. Как вы видите на рис. 2.16, координата *x* мешка может быть такой: –210, –150, –90, ..., 210. Эти цифры расположены на расстоянии 60 шагов друг от друга, так что вы сможете вычислить координату *x* начиная с –210:

$$x = -210 + (0 \times 60)$$
$$x = -210 + (1 \times 60)$$
$$x = -210 + (2 \times 60)$$
$$x = -210 + (3 \times 60)$$

Похожее выражение применимо и к координате *y*.

Мы можем устанавливать координату *x* мешка, сгенерировав случайное число от 0 до 7, умножая его на 60 и прибавляя к результату –210. Рис. 2.19 показывает детально алгоритм создания блока **установить x в** в нашем скрипте. Блок **установить y в** устроен похожим образом.

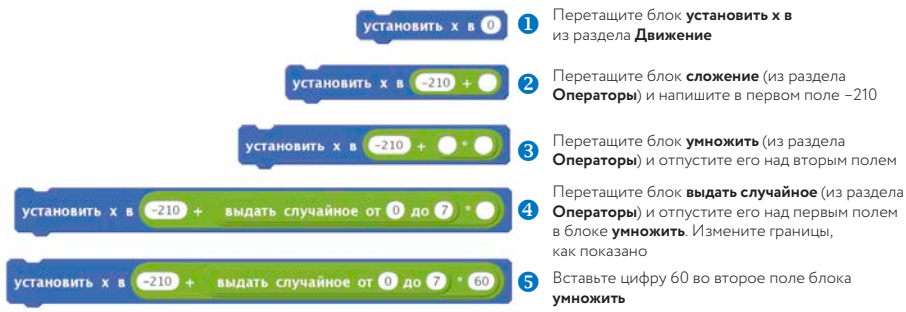


Рис. 2.19. Создаем блок **установить x в**, *как на рис. 2.18*

Появившись в случайном месте, мешок с золотом даст игроку три секунды на то, чтобы схватить его. (Вы можете изменить это время, сделав игру сложнее или проще.) Чтобы отслеживать время, скрипт первым делом обнуляет встроенный таймер. Затем он ждет, пока либо игрок схватит мешок, дотронувшись до него, либо выйдет время. Когда одно из этих условий выполняется, блок **ждать до** позволит скрипту перейти к блоку **если**. Подробное описание того, как создать блок **ждать до**, показано на рис. 2.20.

Рис. 2.20. Создание блока **ждать до** *в скрипте с рис. 2.18*

Блоки внутри блока **если** *будут выполнены, только если условие, указанное вами, верно. В главе 6 подробно объяснена работа этого блока, но пока вы достаточно знаете, чтобы использовать его и с его помощью добавлять в программу новые функции.*

Если игрок дотронется до мешка, команда внутри блока **если/то** запустится. В этом случае блок **играть звук** издаст звук water drop, а блок **изменить счет на 1** (из раздела **Данные**) добавит к счету одно очко.

Игра готова. Кликните по зеленому флажку, чтобы протестировать ее!

ТАЙМЕР SCRATCH

В среде Scratch есть таймер, который фиксирует, сколько времени прошло с момента начала работы. Когда вы открываете Scratch в своем браузере, таймер ставится на 0 и дальше считает время с точностью до десятых секунды, пока интерфейс открыт. Блок **таймер** (из раздела **Сенсоры**) содержит текущие показания таймера. Чекбокс рядом с блоком позволяет показать/скрыть таймер на **Сцене**. Блок **перезапустить таймер** обнуляет показания таймера, и отсчет времени начинается снова. Таймер продолжает работать, даже если проект остановлен.

Поймай яблоки

CatchApples_
NoCode.sb2

Теперь рассмотрим игру «Поймай яблоки» (Catch Apples), показанную на рис. 2.21. В ней яблоки появляются в случайных горизонтальных позициях вверху **Сцены** в случайные моменты времени и падают вниз. Игрок должен передвигать тележку и ловить яблоки, прежде чем они коснутся земли. За каждое пойманное яблоко он получает 1 балл.

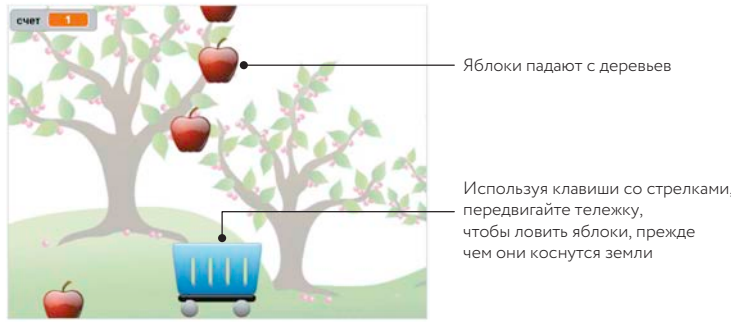


Рис. 2.21. Игра «Поймай яблоки»

Кажется, что в такой игре нужно много спрайтов с практически одинаковыми скриптами. Ведь яблок падает много. Однако в случае со средой Scratch 2 это не так. Благодаря функции *клонирования* вы можете легко создать много копий одного спрайта. В игре будет один спрайт-яблоко (Apple) и столько его клонов, сколько нам захочется.

Откройте файл *CatchApples_NoCode.sb2*, в котором содержится заготовка для нашей игры без скриптов. Чтобы было интереснее, в эту заготовку также была включена переменная под названием **счет** (она создана в разделе **Данные**), которую мы будем использовать, чтобы вести учет пойманных яблок. Но для начала вы сделаете скрипт для спрайта-тележки (Cart), как показано на рис. 2.22.

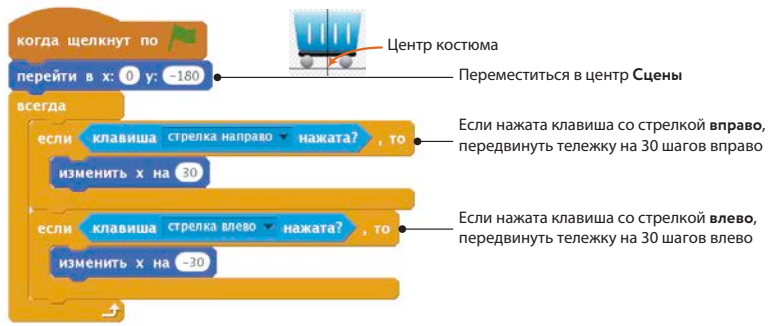


Рис. 2.22. Скрипт для спрайта-тележки

Когда зеленый флажок нажат, мы размещаем тележку внизу **Сцены** по центру. Затем скрипт постоянно проверяет состояние клавиш со стрелками и соответственно передвигает тележку. Методом проб и ошибок я выбрал число 30, а вы можете его изменить по собственному усмотрению.

Займемся клонированием. Для начала добавьте спрайту-яблоку скрипт с рис. 2.23. Он тоже начинает работать, когда нажат зеленый флажок.

Рис. 2.23. Первый скрипт спрайта-яблока

Пока мы ни одного яблока не поймали, поэтому скрипт устанавливает переменную счет на 0 ❶. Затем он делает спрайт видимым при помощи блока **показаться** из раздела **Внешность** ❷. После чего он запускает блок **повторить**, который повторяется 30 раз, чтобы упало 30 яблок ❸.

При каждом повторе спрайт-яблоко приходит в случайную горизонтальную позицию в верхней части **Сцены** ❹. Затем он отдает блоку **создать клон** (из раздела **Управление**) команду клонировать себя самого ❺, ждет непродолжительный случайный отрезок времени ❻ и начинает следующий раунд блока **повторить**. После завершения 30 раундов блока **повторить** скрипт скрывает спрайт-яблоко при помощи блока **спрятаться** ❼ из раздела **Внешность**.

Если вы теперь запустите игру, кликнув по зеленому флажку, 30 яблок начнут случайным образом появляться вверху **Сцены** и оставаться там — потому что мы не объяснили им, что им нужно делать. И вот тут как раз появляется следующий скрипт для спрайта-яблока (рис. 2.24).

Благодаря блоку **когда я начинаю как клон** ❶ (из раздела **Управление**) каждый клон будет выполнять скрипт, показанный на этом рисунке. Каждое яблоко двигается вниз на 10 шагов ❷ и проверяет, было ли оно поймано или пропущено тележкой. Если клон определяет, что коснулся тележки ❸, это означает, что его поймали. Тогда он увеличивает счет и удаляет себя (потому что для него больше нет работы). Если клон падает ниже тележки ❹, игрок промахнулся. В этом случае клон проигрывает другой

звук, прежде чем удалить себя. Если клон не был ни пойман, ни пропущен, он еще падает и блок **всегда** запускается по второму кругу.

Рис. 2.24. Второй скрипт для Яблока

Теперь, когда наши яблоки знают, как им нужно падать, игра готова! Протестируйте ее, кликнув по зеленому флажку. Если вы хотите поэкспериментировать, можете изменить время ожидания между клонированием разных яблок и скорость движения тележки. Нет ли у вас идей, как изменить уровень сложности игры?

И еще о клонированных спрайтах

Любой спрайт может создать копию себя или другого спрайта при помощи блока **создать клон**. (**Сцена** тоже может клонировать спрайты при помощи этого же блока.) Клонированный спрайт наследует состояние оригинала на момент клонирования: позицию и направление, костюм, статус видимости, цвет и размер пера, графические эффекты и т. д. (рис. 2.25).

Рис. 2.25. Клон наследует характеристики оригинала

Клоны также наследуют скрипты спрайта-оригинала, как показано на рис. 2.26. Здесь спрайт-оригинал создает два клона. Когда вы нажимаете **пробел**, все три спрайта (оригинал и два клона) поворачиваются на 15° вправо, потому что они выполняют скрипт **когда клавиша пробел нажата**.

Рис. 2.26. Клоны наследуют скрипты оригинала

Всегда будьте особенно внимательны, когда используете блок **создать клон** в скрипте, который не начинается с зеленого флажка. Иначе у вас может получиться больше спрайтов, чем вы планировали. Посмотрите внимательно на программу, показанную на рис. 2.27. Когда вы в первый раз нажмете **пробел**, появится клон, и после этого в программе будет два спрайта (оригинал и клон).

Рис. 2.27. Клонирование в ответ на нажатие клавиши

Если вы нажмете **пробел** во второй раз, в программе будет четыре спрайта. Почему? В ответ на нажатие клавиши спрайт-оригинал создаст клон, но первый клон тоже среагирует и создаст еще один (клон клона). Нажмите **пробел** в третий раз — и у вас будет восемь спрайтов. Количество клонов растет в геометрической прогрессии!

Вы можете исправить это, клонируя только те спрайты, скрипты которых начинаются с блока **когда щелкнут по**  . Они выполняются только спрайтом-оригиналом.

Итоги

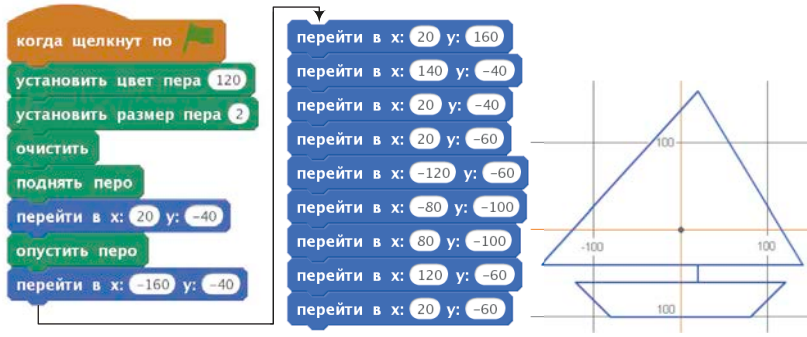
Из этой главы вы узнали, как перемещать спрайты на конкретные позиции на **Сцене** при помощи команд абсолютного движения. Затем вы использовали команды относительного движения, чтобы перемещать спрайты относительно их собственного местоположения и направления.

После этого вы создали несколько отличных рисунков при помощи команд **Пера**. Рисуя различные фигуры, вы открыли для себя возможности блока **повторить**, который позволяет создавать более короткие и более эффективные скрипты. Вы также узнали о команде **печать** и стали использовать ее вместе с блоком **повторить**, чтобы легко создавать сложные узоры.

В конце этой главы вы создали две игры и узнали о функции клонирования. В следующей главе вы будете пользоваться разделами **Внешность** и **Звук**, чтобы создавать еще более увлекательные программы.

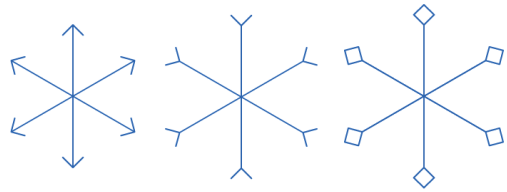
Задания

1. Объясните, как работает такой скрипт. Запишите координаты *x* и *y* для всех углов фигуры.



2. Напишите скрипт, чтобы соединить между собой точки в каждой из этих фигур и определить их окончательную форму:
   а) (30, 20), (80, 20), (80, 30), (90, 30), (90, 80), (80, 80), (80, 90), (30, 90), (30, 80), (20, 80), (20, 30), (30, 30), (30, 20);
   б) (–10, 10), (–30, 10), (–30, 70), (–70, 70), (–70, 30), (–60, 30), (–60, 60), (–40, 60), (–40, 10), (–90, 10), (–90, 90), (–10, 90), (–10, 10).

3. Напишите скрипт, чтобы нарисовать каждый из узоров, приведенных ниже.



4. Рассмотрите следующий скрипт и его результат. Воспроизведите скрипт, добавьте команды для установки пера, запустите его и объясните, как он работает.

5. Рассмотрите следующий скрипт и его результат. Воспроизведите скрипт, добавьте команды для установки пера, запустите его и объясните, как он работает.

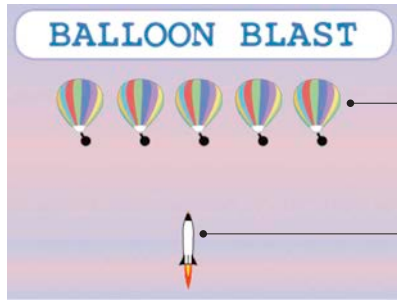
6. Рассмотрите следующий скрипт и его результат. Воспроизведите скрипт, добавьте команды для установки пера, запустите его и объясните, как он работает.

7. Создайте скрипт, показанный ниже, добавьте необходимые команды пера и запустите скрипт. Объясните, как скрипт работает.

8. Напишите программу, которая давала бы результат, показанный ниже.

BalloonBlast_NoCode.sb2

9. В этом задании вам нужно написать скрипты, необходимые, чтобы доделать игру Balloon Blast, показанную ниже.

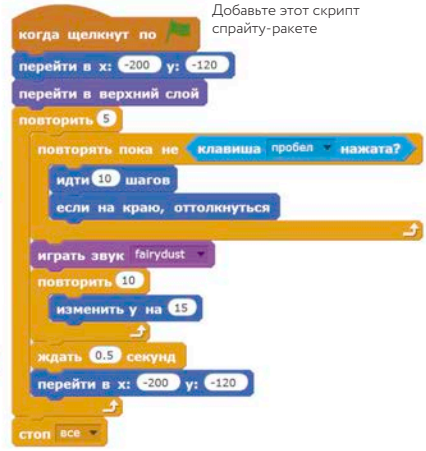


В этой игре два спрайта, Воздушный шар и Ракета (Balloon и Rocket). Когда вы кликаете по зеленому флажку, спрайт Шар создает пять клонов, вы видите их на рисунке вверху. Спрайт Ракета двигается вправо и влево сам по себе, отталкиваясь от краев **Сцены**. Надо нажать пробел в нужный момент, чтобы запустить ракету и полопать шары.

Откройте файл *BalloonBlast_NoCode.sb2*. В этом файле вы найдете код для создания пяти клонов, когда игра начнется. Ваша задача — доделать игру, добавив следующие два скрипта.

Добавьте этот скрипт спрайту-шару

Добавьте этот скрипт спрайту-ракете



3

ВНЕШНОСТЬ И ЗВУКИ

Вы уже научились перемещать спрайты по **Сцене** при помощи команд раздела **Движение** и использовать **Перо** для создания узоров. Сейчас вы познакомитесь с командами разделов **Внешность** и **Звуки**. Вот что вы будете делать:

- научитесь создавать анимацию и эффекты изображений;
- узнаете, как в Scratch работают слои;
- научитесь проигрывать звуковые файлы и сочинять музыку;
- научитесь самостоятельно создавать готовые анимированные сцены.

Команды раздела **Внешность** позволят вам создавать анимацию и применять графические эффекты, такие как завихрение, рыбий глаз, призрак и пр., в работе с костюмами и фонами. Команды из раздела **Звуки** пригодятся вам, если вы захотите добавить в свои программы звуки, голоса или музыку. Займемся анимацией!

Раздел Внешность

Вы можете рисовать прямо на **Сцене** при помощи команд **Пера**, но костюмы дают еще один мощный и иногда намного более простой способ добавить графику в ваши программы. Команды раздела **Внешность** позволят вам управлять костюмами, создавая анимацию, добавлять облачка с мыслями, применять графические эффекты и менять степень видимости спрайта. В этом разделе мы познакомимся с ними.

Анимирование костюма

Animation.sb2

Вы уже знаете, как отправить спрайт из одной точки на **Сцене** в другую. Но перемещающиеся по **Сцене** статичные спрайты смотрятся не слишком живо. Если вы будете использовать разные костюмы и переключаться между ними достаточно быстро, будет казаться, что спрайт действительно движется! Откройте файл *Animation.sb2,* чтобы протестировать анимацию, которую мы видим на рис. 3.1.

Рис. 3.1. Вы можете создать иллюзию анимации, меняя один костюм на другой

В этой программе один спрайт с семью костюмами и одним скриптом. У него есть семь костюмов в закладке **Костюмы** (**Frame1** ... **Frame7**), а скрипт — в закладке **Скрипты**. Когда вы запускаете программу, кликнув по зеленому флажку, кажется, что человечек идет по **Сцене**. Ключ к его движению — команда **следующий костюм**, которая говорит спрайту надеть следующий костюм из списка. Если спрайт надел последний костюм из списка, дальше он перейдет к первому.

После клика по зеленому флажку скрипт начинается с цикла **всегда** с блоком **ждать** в конце, чтобы сделать задержку на 0,1 секунды после каждой смены костюма. Если эту задержку убрать, будет казаться, что человечек не идет, а бежит. Поэкспериментируйте с разными значениями для блоков **идти** и **ждать** и посмотрите, как это скажется на анимации.

Такого человечка можно нарисовать и при помощи команд пера, но тогда нужен длинный скрипт. Зато после того, как вы нарисуете костюмы, программирование анимации не потребует больших усилий. Помните, что вы можете создавать и изображения — как при помощи ваших любимых графических редакторов, так и графического редактора Scratch.

ClickOnFace.sb2

Если вы хотите, чтобы пользователи взаимодействовали со спрайтом, вы можете изменять его костюм по клику мышки, как в программе Click on Face.

В этом приложении один спрайт — Лицо (Face), у которого есть пять костюмов, как показано на рис. 3.2. Программа использует блок **когда спрайт нажат** (из раздела **События**), чтобы дать спрайту знать, когда пора менять костюмы.

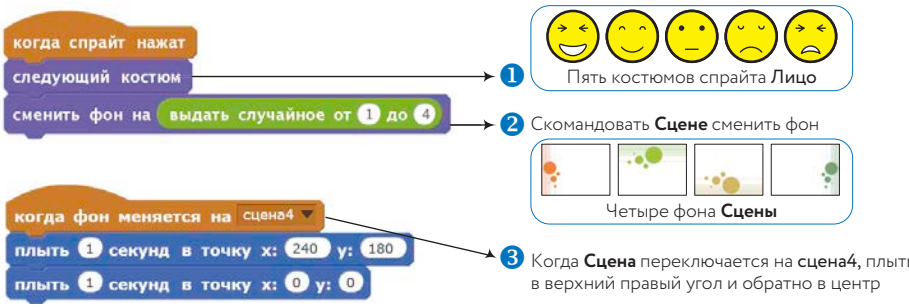


Рис. 3.2. Лицо смайлика и фон меняются каждый раз, когда по спрайту кликают

Каждый раз, когда вы кликнете по смайлику, он будет меняться на следующий в списке. Скрипт также использует блок **сменить фон на**, чтобы скомандовать **Сцене** случайным образом сменить фон на любой из четырех имеющихся. Когда **Сцена** переключается на сцена4, спрайт Лицо фиксирует это событие (при помощи блока-триггера **когда фон меняется на** из раздела **События**). В этом случае Лицо совершает путешествие в верхний правый угол **Сцены** и возвращается в центр.

УПРАЖНЕНИЕ 3.1

TrafficLight.sb2

Файл *TrafficLight.sb2* содержит один спрайт, у которого три костюма — Красный, Желтый и Зеленый (Red, Orange и Green), а также незавершенный скрипт, который показан ниже. Допишите программу, добавив необходимые блоки **ждать**, чтобы создать реалистичные анимированные светофоры.

Вы можете использовать команду **сменить фон на,** *чтобы менять сцены в мультфильме, уровни игры и т. д. Любой спрайт в вашем проекте может использовать блок* **когда фон меняется на**, *чтобы определять момент, когда* **Сцена** *меняет костюм, и действовать соответственно. Подробнее об этом вы прочтете в окошке* **Подсказки** *интерфейса Scratch.*

Спрайты, которые думают и говорят

Вы можете использовать команды **думать** или **сказать**, чтобы ваш спрайт начал говорить или думать как герой комикса, как показано на рис. 3.3 (слева).

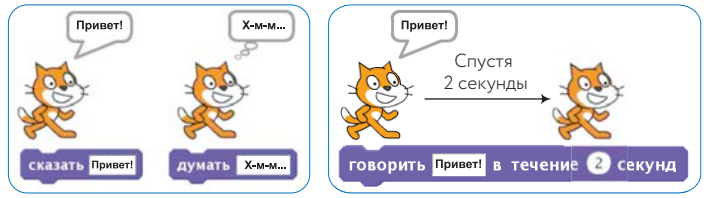


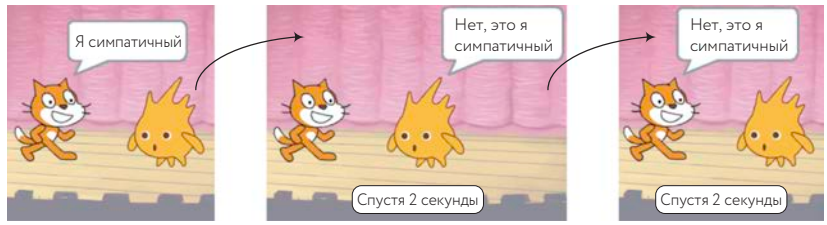
Рис. 3.3. Используйте команды ***думать*** *или* ***сказать****, чтобы показать текст речи или мыслей в облачке*

Любая фраза, которую вы используете в этих командах, появится над спрайтом и будет отображаться постоянно. Если вы хотите стереть текст, используйте пустые блоки **думать** или **сказать**. Вы также можете показывать текст на протяжении определенного времени, если используете команду **говорить в течение секунд** (или **думать в течение секунд**), как показано на рис. 3.3 (справа).

УПРАЖНЕНИЕ 3.2

Argue.sb2

Чтобы увидеть команды **думать** или **сказать** в действии, откройте файл *Argue.sb2* и запустите его. Эта программа симулирует бесконечный спор между двумя персонажами, как показано на картинке ниже. Внимательно посмотрите на их скрипты, чтобы понять, как используются установки таймера, чтобы синхронизировать действия двух персонажей.



Эффекты изображений

GraphicEffects.sb2

Команда **установить эффект** позволяет вам применять различные графические эффекты в работе с костюмами и фонами. В среде Scratch эти эффекты называются рыбий глаз, завихрение, мозаика и т. д. На рисунке 3.4 показано, какого результата можно достичь с их помощью.

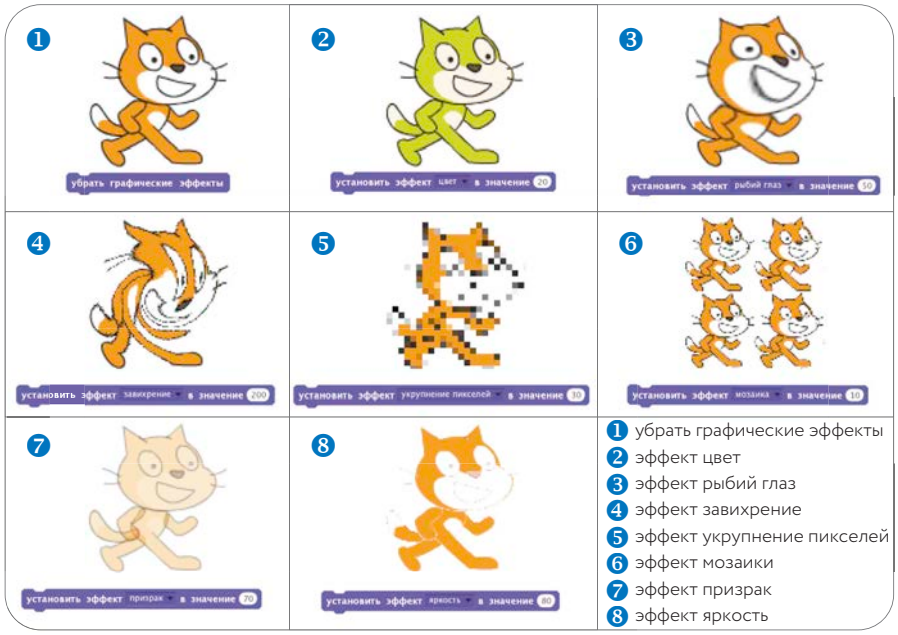


Рис. 3.4. Этот рисунок показывает, что произойдет с котом, если применить к нему графические эффекты Scratch

Кликните по стрелке вниз в блоке **установить эффект**, чтобы выбрать из выпадающего меню эффект, который вы хотите применить. Вы можете использовать команду **изменить эффект на**, чтобы постепенно перейти на новый эффект, а не резко сменить его. Например, если эффект **призрак** установлен на 40, то, изменив его на 60, мы получим 100, что заставит спрайт исчезнуть полностью (как настоящий призрак). Если вы хотите вернуть изображению исходный вид, воспользуйтесь блоком **убрать графические эффекты**.

Вы можете применить несколько эффектов одновременно, использовав последовательно несколько графических команд.

Размер и видимость

SneezingCat.sb2

Иногда нужно изменить размер спрайта или контролировать момент его появления в программе. Например, чтобы более близкие объекты были крупнее или в начале игры выводился спрайт с инструкциями.

Если вы хотите уменьшить или увеличить спрайт, вам помогут команды **установить размер %** и **изменить размер на**. Первая устанавливает размер спрайта в процентном соотношении с исходным, а вторая

меняет на определенную величину, соотносящуюся с исходным размером. Когда вам нужно, чтобы спрайт появился или исчез, используйте блоки **показаться** или **спрятаться** соответственно.

Чтобы посмотреть, как работают эти команды, откройте файл *SneezingCat.sb2*. В этой программе кот будет чихать как мультяшный персонаж, меняясь в размере, как показано на рис. 3.5.

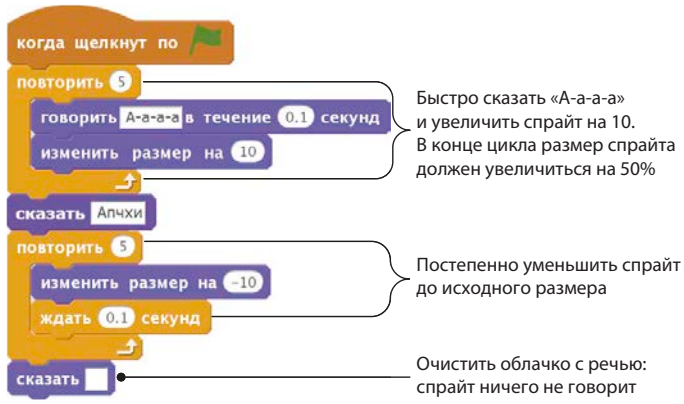


Рис. 3.5. Этот скрипт заставляет кота чихнуть

Размер спрайта увеличивается, когда он собирается чихнуть, а потом возвращается к исходному значению. Запустите программу и посмотрите, что получится, чтобы понять, как работают эти команды.

УПРАЖНЕНИЕ 3.3

Добавьте в конце скрипта с рис. 3.5 блок, чтобы кот исчез после своего эффектного чиха. Добавьте еще один блок, чтобы спрайт появился в начале скрипта.

Слои

Две последние команды раздела **Внешность** отражают порядок, в котором спрайты рисуются на **Сцене**. Он определяет, какой спрайт будет виден, когда они наслаиваются друг на друга. Предположим, вы хотите создать сцену с девушкой, стоящей за огромной скалой. Тут есть две возможности, как показано на рис. 3.6 (слева).

Если вы хотите, чтобы девушка была за скалой, вам нужно вывести скалу в первый слой рисунка или же отправить девушку во второй. Scratch предлагает две команды, с помощью которых можно перераспределять слои: **перейти в верхний слой** и **перейти назад на слоев** (также показаны на рисунке). Первая команда сообщает системе, что спрайт

всегда нужно рисовать поверх других изображений, а вторая отправляет его на столько слоев назад, на сколько вы укажете.

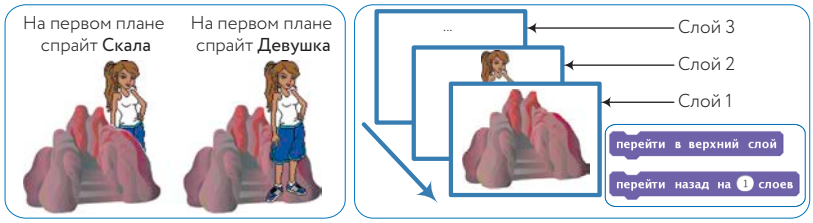


Рис. 3.6. Спрайт на первом плане полностью видим и может закрывать части других спрайтов, на которые наслаивается

УПРАЖНЕНИЕ 3.4

В программе *Layers.sb2* на **Сцене** движутся четыре объекта. Вы можете вывести объект на первый план, кликнув по первой букве его цвета. Запустите программу, чтобы получше изучить действие команды **перейти в верхний слой**. Layers.sb2

Мы поговорили про анимацию и использование команд раздела **Внешность**. Но есть еще одно средство, с помощью которого можно оживить ваши программы. Дальше мы поговорим о разделе **Звуки** и его наборе команд.

Раздел Звуки

Игры и другие программы используют звуки и фоновую музыку, чтобы оживить действие. Сейчас вы узнаете, как использовать в Scratch блоки для управления звуками, например добавить в программу аудиофайлы и проигрывать их в качестве фона.

Затем мы взглянем на блоки-команды для игры на барабанах и других музыкальных инструментах. Наконец, вы узнаете, как управлять громкостью и скоростью (темпом) игры инструментов.

Как проигрывать аудиофайлы

Вы можете сохранить аудиофайл на компьютере во множестве форматов, но Scratch распознает только два: WAV и MP3. Есть три блока-команды, которые позволяют использовать эти файлы в своих программах: **играть звук**, **играть звук до конца** и **остановить все звуки**. Две первые проигрывают заданный звук. Команда **играть звук** позволяет следующей команде начать действовать раньше, чем звук отзвучит. Команда **играть звук до конца** не перейдет к следующей, пока звук не закончится. Команда **остановить все звуки** выключает все играющие звуки.

Вы можете добавить фоновую музыку, проигрывая по кругу аудиофайл. Самый простой способ — использовать блок **играть звук до конца**, позволив файлу проиграть до конца, а затем запустить его заново, как показано на рис. 3.7 (слева).

Рис. 3.7. Два пути создания фоновой музыки: повторить звук после того, как он закончит звучать (слева), или вновь начать проигрывать после того, как он отзвучит какое-то время (справа)

Так можно создать очень короткую, но заметную паузу перед каждым новым запуском. Возможно использовать команду **играть звук** вместе с командой **ждать**, чтобы лучше контролировать продолжительность звучания, как показано на рис. 3.7 (справа). Экспериментируя со временем ожидания, вы сможете сократить паузу и сделать более плавный переход между окончанием мелодии и началом ее повтора.

Игра на барабанах и другие звуки

BeatsDemo.sb2

Если вы разрабатываете игры, возможно, вы захотите, чтобы короткий звуковой эффект раздавался, когда игрок попадает в цель, проходит уровень и т. п. Такие звуки легко создавать при помощи команды **барабану играть тактов**, которая проигрывает на ваш выбор один из 18 звуков барабанов определенное количество раз. Вы также можете добавить паузы при помощи команды **подождать тактов**. Программа *BeatsDemo.sb2* (рис. 3.8) демонстрирует эффект от изменения количества тактов.

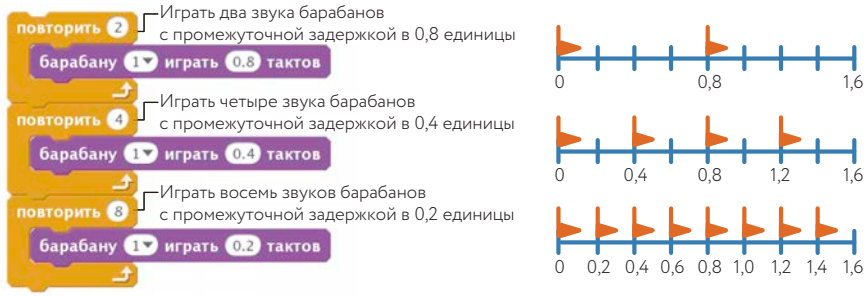


Рис. 3.8. Иллюстрация использования понятия тактов в Scratch

Скрипт содержит три блока **повторить** со счетом на два, четыре и восемь соответственно. Каждый блок проигрывает один и тот же звук барабанов, используя разное количество тактов. Если вы представите себе временну́ю ось, поделенную на интервалы в 0,2 единицы, то первый цикл будет проигрывать звуки барабана, разнесенные между собой на 0,8 единицы времени, второй — на 0,4 единицы, третий — на 0,2 единицы. Каждый цикл продолжается одно и то же количество времени; мы просто за один и тот же промежуток времени бьем в барабан разное количество раз.

Я говорю о «единице времени» вместо того, чтобы говорить о секундах, потому что продолжительность цикла зависит от темпа, который вы можете менять при помощи команды **установить темп**. При использовании темпа по умолчанию (60 ударов в минуту (beats per minute, bpm)) каждый цикл в приведенном нами примере будет продолжаться 1,6 секунды. Если вы установите темп в 120 bpm, каждый цикл будет продолжаться 0,8 секунды, при темпе в 30 bpm это будут 3,2 секунды и т. д.

Сочиняем музыку

FrereJacques.sb2

Scratch также содержит две команды, которые позволят вам проигрывать ноты и сочинять свою музыку. Команда **играть ноту тактов** проигрывает выбранную вами ноту от 0 до 127 в течение того количества тактов, которое вы укажете. Блок **выбрать инструмент** сообщает Scratch, на каком инструменте должна прозвучать нота. Давайте используем эти команды, чтобы записать песенку. Скрипт, показанный на рис. 3.9, проигрывает французскую детскую песенку «Братец Яков» (Frère Jacques).

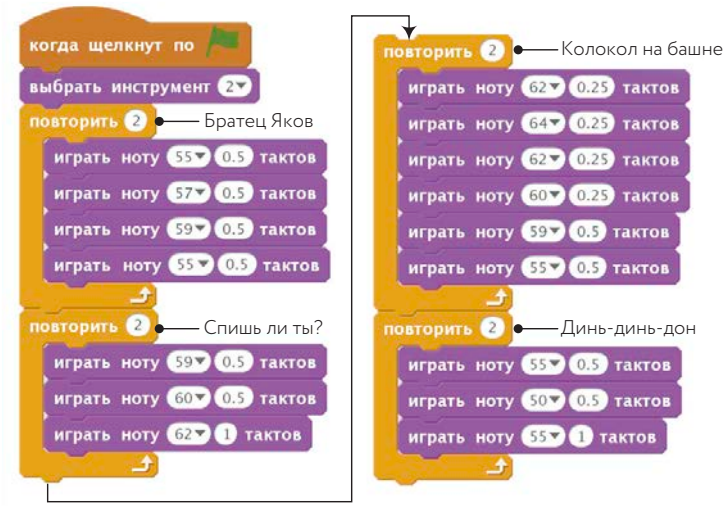


Рис. 3.9. Скрипт, играющий песенку «Братец Яков»

Откройте программу, которая называется *FrereJacques.sb2*, и поэкспериментируйте с разными значениями для команды **выбрать инструмент**, чтобы сменить инструмент, играющий песню.

Контроль громкости звука

Предположим, вы хотите, чтобы в вашей программе звук сходил на нет в ответ на какое-то событие. Если вы запускаете в космос ракету, вы, должно быть, захотите, чтобы она гудела громко в момент старта и потом постепенно затихала по мере того, как улетает все дальше.

В среде Scratch есть набор команд для управления громкостью аудиофайлов, звука барабанов и нот. Команда **установить громкость %** устанавливает громкость спрайта в процентном соотношении с максимальной громкостью колонок. Однако она действует только на тот спрайт, который ею пользуется (или на **Сцену**), так что если вы хотите, чтобы одновременно играли звуки разной громкости, придется использовать несколько спрайтов. Блок **изменить громкость на** уменьшает или увеличивает громкость на ту величину, которую вы введете. Отрицательные значения в этом поле делают звуки тише, а положительные — громче. Вы даже можете показать громкость спрайта на **Сцене**, поставив галочку в чекбоксе рядом с блоком **громкость**. Эти блоки пригодятся вам, если вы захотите, чтобы звук менялся в зависимости от того, как близко спрайт подходит к цели (как, например, в игре с поисками сокровищ), или сделать одну часть песни громче, а другую тише. Можно воспользоваться этими блоками, чтобы изобразить оркестр, играя на разных инструментах (с разными уровнями громкости) одновременно.

УПРАЖНЕНИЕ 3.5

VolumeDemo.sb2

Файл *VolumeDemo.sb2* показывает идущего по лесу кота. Программа использует команду **изменить громкость на**, чтобы звук шагов пропадал по мере того, как он углубляется в лес. Предложите свои идеи того, как эту симуляцию можно сделать более реалистичной, и попробуйте воплотить их в жизнь.

Устанавливаем темп

Последние три блока раздела **Звуки** связаны с темпом, или скоростью, с которой проигрываются барабаны и ноты. Темп измеряется в ударах в минуту (bpm). Чем он выше, тем быстрее воспроизводятся звуки.

Scratch позволяет выбирать конкретный темп при помощи команды **установить темп bpm**. Вы можете приказать спрайту ускориться или замедлиться при помощи команды **изменить темп на**. Если вы хотите увидеть темп спрайта отображенным на **Сцене**, вам нужно поставить галочку в чекбоксе рядом с блоком **темп**.

Проекты Scratch

Команды разделов **Внешность** и **Звук** помогут вам добавить в ваши программы много классных эффектов. В этом разделе мы соберем воедино всё, чему научились в этой главе, и создадим несколько анимированных сцен с танцующим человеком и фейерверками. Это поможет внимательнее изучить некоторые из новых команд и даст возможность попрактиковаться в создании проектов в Scratch.

УПРАЖНЕНИЕ 3.6

Откройте файл *TempoDemo.sb2* и запустите его, чтобы увидеть, как работают команды **установить темп bpm** и **изменить темп на**. TempoDemo.sb2

Танцы на сцене

В этом разделе вы анимируете танцора на **Сцене**. Картинка показана на рис. 3.10, а готовый скрипт сохранен в файле *DanceOnStage.sb2*. Сейчас мы повторим весь алгоритм его создания — следите внимательно за тем, как это делается! DanceOnStage.sb2

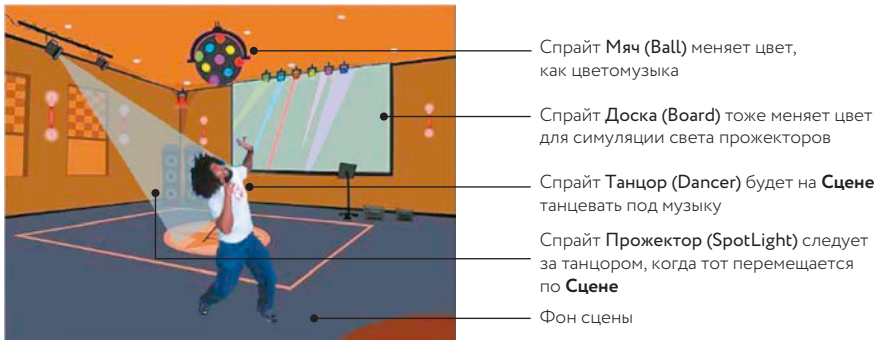


Рис. 3.10. Программа Dance Party в действии

Первым делом создайте новый проект. Если интерфейс Scratch уже открыт, достаточно запустить его — проект создастся автоматически. Другой вариант — выбрать **Новый** из меню **Файл**. В обоих случаях у вас будет новый проект, с котом-спрайтом по умолчанию. Фон, который вы будете использовать для этой программы, — party room из категории **В помещении**. Импортируйте его и удалите исходный белый фон, который вам не понадобится. Теперь **Сцена** должна выглядеть как на рис. 3.11.

Посмотрите на рис. 3.10 и 3.11 внимательно и обратите внимание на то, как спрайты Мяч и Доска похожи на фрагменты фона. Как вы сейчас увидите, они на самом деле были сделаны из картинки фона

и расположены на **Сцене** так, чтобы закрывать части изображения, из которых вырезаны. Такой способ создания спрайтов позволяет нам менять их цвет и сделать **Сцену** более реалистичной.

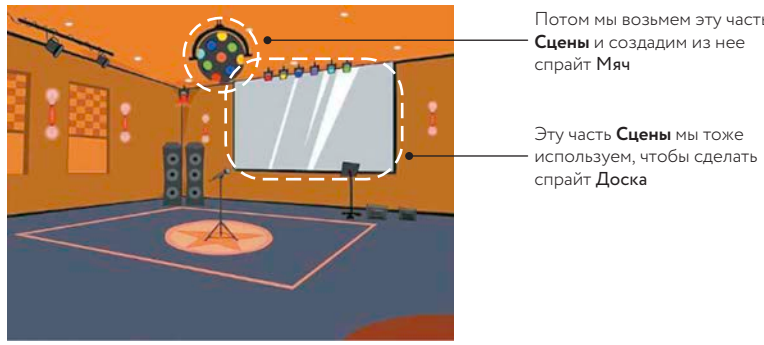


Рис. 3.11. Мы превратим некоторые части фона в спрайты

Теперь нам нужна фоновая музыка. Используем файл medieval1 из категории **Музыкальная петля**. Импортируйте этот файл на **Сцену** и затем удалите исходный звук pop. Добавьте на **Сцену** скрипт с рис. 3.12. В нем используется команда **играть звук** вместе со временем ожидания, которое позволяет аудиофайлу плавно запускаться по кругу. Время ожидания 9,5 секунды было выбрано экспериментальным путем.

Рис. 3.12. Сцена проигрывает нашу фоновую музыку

Кликните по зеленому флажку, чтобы протестировать то, что уже сделано. Вы услышите постоянно повторяющийся аудиоклип. Остановите скрипт, когда будете готовы, и мы добавим танцора.

Замените кота на танцора. Импортируйте костюмы dan-a и dan-b из категории **Люди**, удалите два костюма кота и поменяйте название спрайта «Кот» на «Танцора».

Скрипт для танцора показан на рис. 3.13. Он проходит 20 шагов вправо, меняет костюм, идет 20 шагов влево и снова меняет костюм. Эти шаги постоянно повторяются: возникает иллюзия, что он танцует. Скрипт также для разнообразия слегка меняет показатель эффекта **рыбий глаз**. Кликните по зеленому флажку, чтобы протестировать свои

дополнения к программе. Вы должны услышать фоновую музыку и увидеть, как Танцор движется по **Сцене** вправо-влево.

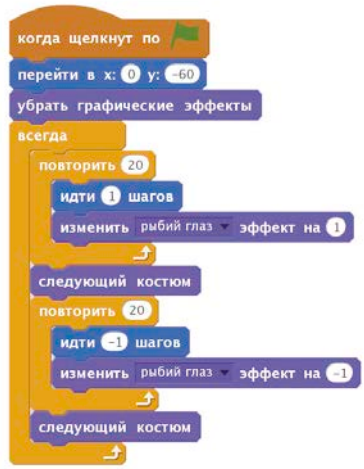


Рис. 3.13. Этот скрипт говорит спрайту Танцор, *как танцевать*

Теперь на вечеринке есть Танцор. Пора добавить ярких огней при помощи спрайтов Шар, Доска и Прожектор. Чтобы создать Шар, кликните по иконке **Сцены**, а затем откройте закладку **Фоны**. Кликните правой кнопкой мыши по иконке фона party room и выберите из выпадающего меню вариант **сохранить локальный файл**. Появится окно, которое позволит сохранить фон на вашем устройстве. Запомните, куда вы сохранили изображение: скоро вам нужно будет импортировать его обратно.

Нажмите кнопку **Загрузить с компьютера** (над списком спрайтов) и выберите изображение, которое вы только что сохранили. Появится новый спрайт, костюм которого идентичен фоновому изображению. Назовите его «Шаром» и измените его костюм в графическом редакторе, чтобы удалить всё, за исключением пестрого шара, как на рис. 3.14 (слева). Не забудьте раскрасить пространство вокруг шара прозрачным цветом. Теперь разместите шар на сцене точно в том месте фонового рисунка, откуда вы его взяли, так, чтобы он казался частью изображения (см. рис. 3.11).

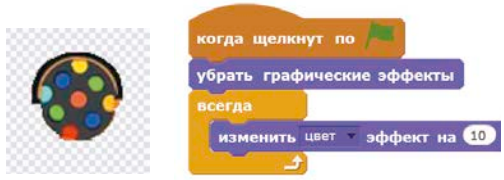


Рис. 3.14. Костюм спрайта Шар, *каким мы видим его в графическом редакторе, и его скрипт*

Рис. 3.14 также показывает скрипт, который вам нужно добавить спрайту-шару. Он постоянно меняет эффект цвета, создавая иллюзию, будто небольшие кружочки действительно меняют цвет.

Создайте спрайт Доска тем же способом, каким вы создали шар.

Рис. 3.15 показывает вид спрайта в графическом редакторе (слева) и скрипт, который вам понадобится, чтобы анимировать его (справа). Я добавил несколько цветов этому костюму (сравните с рис. 3.11), чтобы сделать команду **изменить цвет эффект** эффективной.

Поскольку спрайты Доска и Танцор накладываются друг на друга, скрипт отправляет доску на два слоя назад, чтобы танцор всегда оставался на первом плане. Вы можете сделать то же, выбрав спрайт Танцор и кликнув по блоку **перейти в верхний слой** из раздела **Внешность**.

Рис. 3.15. Спрайт Доска *и его скрипт*

Последний спрайт в этой программе — Прожектор. Рис. 3.16 показывает, как он выглядит в графическом редакторе. Вы также видите скрипт, который вам нужно создать. Центр изображения находится на кончике конусообразной формы, изображающей световой луч.

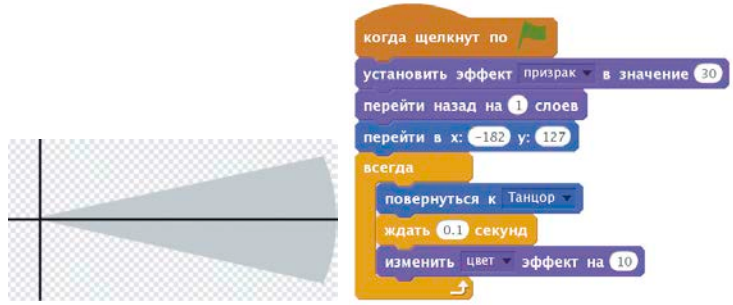


Рис. 3.16. Спрайт Прожектор *и его скрипт*

Первым делом скрипт устанавливает **эффект призрак** со значением 30, чтобы он был прозрачным и не заслонял фон. Затем он отправляет спрайт на один слой назад, и луч света оказывается за Танцором. Теперь спрайт расположен так, что кажется, будто луч света исходит

из прожектора (см. рис. 3.10). Вам нужно выбрать координаты *x* и *y* на основе вашего рисунка. После этого скрипт приказывает световому лучу следовать за танцором (при помощи команды **повернуться к**) и всегда менять свой цвет.

После того как вы добавите прожектор, программа будет готова. Кликните по зеленому флажку, чтобы посмотреть, что происходит на вашей вечеринке! Помимо музыки и танцев, вы увидите, как спрайты Шар, Доска и Прожектор меняют цвета, как на настоящей дискотеке.

Ниже мы рассмотрим другую программу, где используются многие графические эффекты, изученные нами в этой главе.

Фейерверки

Fireworks_NoCode.sb2

Еще одна программа, где можно поработать с графическими блоками и другими функциями, которые мы обсуждали в этой главе, — анимированные фейерверки. В этом разделе вы сделаете простую анимацию с фейерверками, которые рассыпаются по небу разноцветными искрами. Ракетницы взрываются в случайные моменты, при этом появляются искры, которые летят вниз, как будто под воздействием силы притяжения, и постепенно тают в воздухе, как показано на рис. 3.17.

Рис. 3.17. Анимация «Фейерверки» в действии

Первым делом откройте файл *Fireworks_NoCode.sb2*, в котором находится заготовка для программы без скриптов. Как показано на рис. 3.17, в этой программе два спрайта: Город и Ракета (City и Rocket). Город — картинка с высотными зданиями, которую вы можете анимировать как вам захочется. Спрайт Ракета постоянно создает клоны, которые взрываются в темном небе. Это и будут фейерверки.

У Ракеты восемь костюмов, как показано на рис. 3.18. Первый — C1 — просто маленькая точка, которую мы запускаем в небо. Когда она достигает цели, которая выбирается случайным образом, она меняет костюм,

причем тот тоже выбирается случайным образом. Так симулируется взрыв. Затем мы используем подходящий графический эффект, чтобы взрыв выглядел реалистичнее.

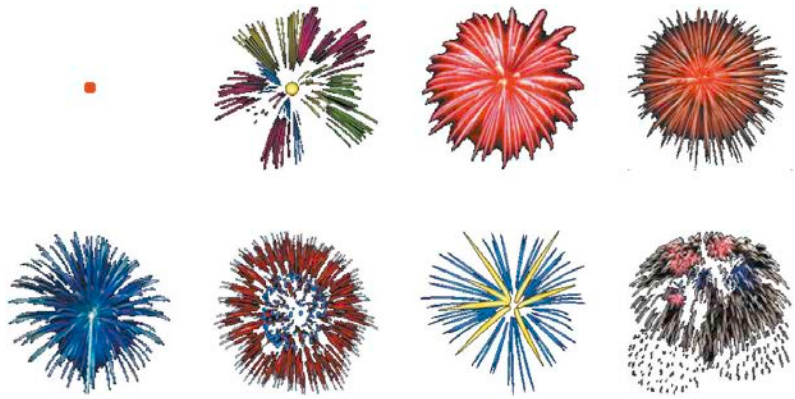


Рис. 3.18. Восемь костюмов спрайта Ракета

Помня об этом плане, добавьте ракете скрипт, показанный на рис. 3.19. Он запускается, когда пользователь кликает по зеленому флажку. Скрыв спрайт, скрипт начинает постоянно повторяющийся цикл (блок **всегда**) создания клонов самого себя в случайные моменты времени. Поскольку клоны наследуют степень видимости ракеты, все они сначала будут скрыты.

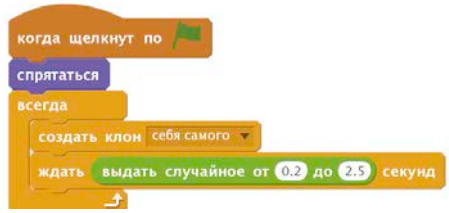


Рис. 3.19. Первый скрипт для спрайта Ракета

Теперь нам нужно дать клонированным ракетам команду, что им дальше делать. Этот скрипт показан на рис. 3.20.

Клонированная ракета начинает надевать свой первый костюм ❶ (маленькая красная точка). Затем она перемещается на случайную горизонтальную позицию в нижней части **Сцены** ❷, показывает себя ❸ и перемещается на случайную позицию в верхней части **Сцены** ❹ (где-то над зданиями). Эта часть скрипта симулирует запуск ракеты, и если вы запустите скрипт, то увидите красную точку, движущуюся от земли

в небо. Когда она достигает конечной точки на небе, она взрывается в соответствии с инструкциями во второй части скрипта.

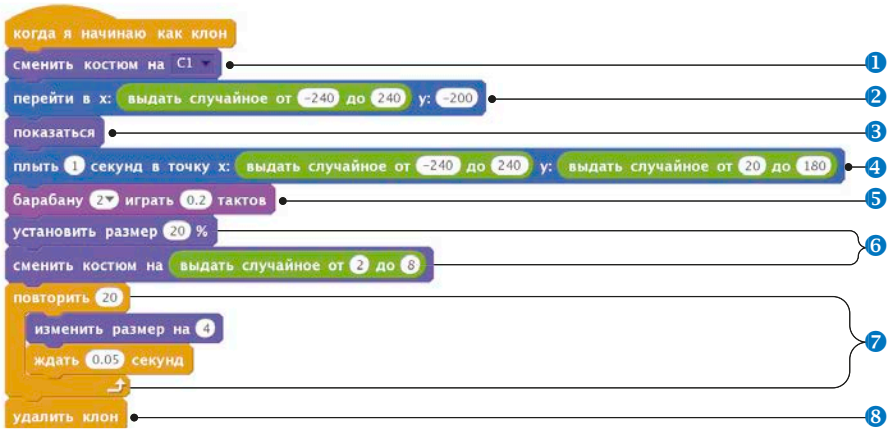


Рис. 3.20. Начало скрипта клонированных спрайтов

Сначала клон воспроизводит короткий звук барабана ❺ (он изображает звук взрыва). Взрыв фейерверка сначала совсем маленький, потом разрастается, так что сначала установлен размер клона 20% от оригинала и случайным образом выбирается один из костюмов ❻. Затем начинается цикл команды **повторить** ❼, чтобы фейерверки разрастались. На каждом повторе клон увеличивается на 4 единицы, а в конце цикла удаляется ❽.

Вот так и получается огненная феерия! Теперь вы можете запустить анимацию и посмотреть, что у вас вышло. При помощи всего лишь пары скриптов мы сделали довольно сложную анимацию.

Итоги

В этой главе мы ввели много новых блоков, которые могут быть использованы для того, чтобы добавить нашим программам ярких моментов. С помощью этих блоков мы можем ввести цвета, анимацию, графические эффекты, музыку и многое другое. Мы разобрались с блоками раздела **Внешность** и дали несколько примеров их использования. Вы анимировали спрайты, меняя их костюмы, узнали, как рисовать слои, и посмотрели, как они влияют на внешний вид перекрывающих друг друга спрайтов.

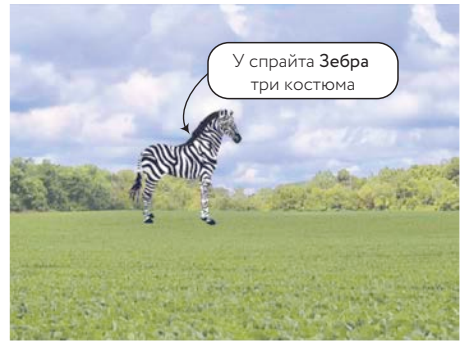
Затем мы обсудили команды раздела **Звуки** и объяснили, как проигрывать аудиофайлы, звуки барабанов и отдельные ноты. Вы создали готовую анимированную сцену танцев при помощи команд разделов **Внешность** и **Звуки**, а закончили мощным аккордом, сделав симуляцию фейерверков.

В следующей главе вы научитесь работать с разными спрайтами, используя технологию передачи и получения сообщений. Вы также узнаете, как разбивать большие программы на более компактные и легче управляемые части, которые называются процедурами. Это ключ к написанию более сложных программ.

Задания

Zebra.sb2

1. Откройте программу *Zebra.sb2*, которая показана на рисунке ниже. В ней один спрайт (Зебра), у которого три костюма. Напишите скрипт, с помощью которого зебра будет перемещаться по сцене и менять костюмы так, чтобы создавалась иллюзия бега.



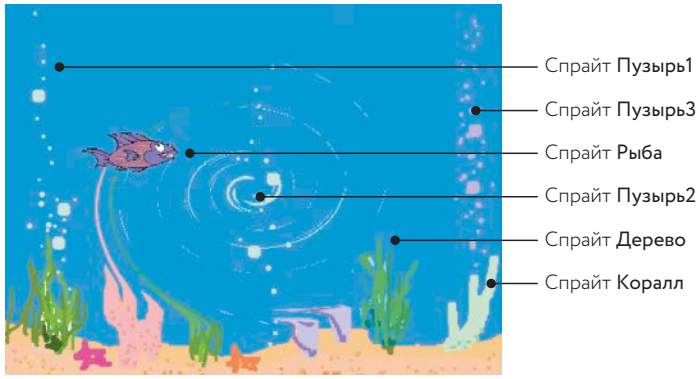
Wolf.sb2

2. Откройте программу *Wolf.sb2*, которая показана на рисунке ниже. Когда вы кликаете по зеленому флажку, Волк воспроизводит звук воя (WolfHowl), который продолжается около четырех секунд.

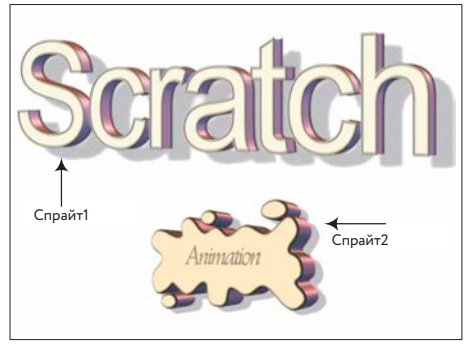
   Создайте скрипт, который меняет костюмы волка синхронно с воем. (Подсказка: вставьте команду **ждать** с подходящим временем задержки после каждой смены костюма.)

3. Откройте программу *ChangingHat.sb2*, которая показана ниже. В этой программе шляпа — спрайт с пятью костюмами. Напишите скрипт, который менял бы костюм Шляпы (Hat) по клику мышью. Затем создайте игру, в которой пользователь переодевает персонажей, кликая по разным частям их нарядов. ChangingHat.sb2

4. Откройте файл *Aquarium.sb2*. Эта программа содержит шесть спрайтов, как показано на рисунке ниже. Протестируйте различные графические эффекты, анимируя аквариум. Вот пара идей: Aquarium.sb2
   а) используйте на **Сцене** эффект **завихрение**. Начните с большого числа, скажем 1000, чтобы картинка казалась волнистой;
   б) измените костюмы спрайтов Пузырь1 и Пузырь2 (Bubble1 и Bubble2);
   в) перемещайте Рыбу (Fish) по сцене, меняя ее костюмы;
   г) примените эффект **призрак** к спрайту Дерево (Tree);
   д) используйте цветовые эффекты для спрайтов Коралл и Пузырь3 (Coral и Bubble3).

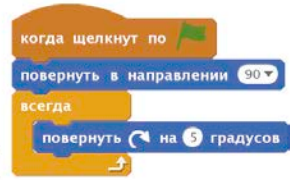


5. Откройте программу *Words.sb2* и анимируйте слова, работая с их размером и вращением. Создайте два скрипта, как показано на рисунке, и запустите программу, чтобы посмотреть на результат. Words.sb2



Скрипт для **Спрайта1**

Скрипт для **Спрайта2**



Joke.sb2

6. Откройте программу *Joke.sb2*, показанную на рисунке внизу. Доработайте скрипты для спрайтов Мальчик и Девочка (Boy и Girl) так, чтобы они рассказывали любую шутку на ваш выбор.

Скрипт для спрайта **Мальчик**

Скрипт для спрайта **Девочка**

7. Откройте файл *Nature.sb2*. Эта программа содержит три спрайта, как видно на картинке внизу. Анимируйте сцену, используя функции **Движения** и **Звука**. Вот несколько идей:

Nature.sb2

   а) у спрайта Птица (Bird) два костюма, которые создают эффект полета. Создайте скрипт, чтобы птица летала по всей **Сцене** и проигрывала звук bird в случайные моменты времени;
   б) у спрайта Утка (Duck) 12 костюмов, которые показывают, как она вытаскивает из воды рыбу и ест ее. Напишите скрипт, чтобы Утка перемещалась по **Сцене** и проигрывала звук duck в случайные моменты времени;
   в) у спрайта Тюлень (Seal) четыре костюма, которые показывают, как он играет с мячом. Создайте скрипт, чтобы Тюлень играл и в случайные моменты времени издавал звук sea lion.

4

ПРОЦЕДУРЫ

В этой главе объясняется, как применить к программированию принцип «разделяй и властвуй». Вместо того чтобы создавать программы целиком, вы сможете писать отдельные процедуры, которые потом сложите в единое целое. Использование процедур упростит написание, тестирование и исправление программ. Из этой главы вы узнаете, как:

- координировать поведение нескольких спрайтов с помощью сообщений;
- использовать передачу сообщений для внедрения процедур;
- использовать имеющуюся в Scratch 2 функцию **создай свой блок**;
- использовать техники структурированного программирования.

Большинство программ, написанных нами до настоящего момента, содержали только один спрайт. Но обычно необходимо, чтобы несколько спрайтов работали вместе. Например, в мультфильме может быть несколько персонажей, как и несколько фонов. Поэтому нам необходимо синхронизировать работу спрайтов.

В этой главе мы будем координировать работу нескольких спрайтов, используя механизм передачи сообщений. Мы также обсудим применение доступной в Scratch 2 функции пользовательских блоков для структурирования объемных программ, разделяя их на более мелкие, легче управляемые фрагменты, которые называются процедурами. Процедура — последовательность команд, выполняющая определенную функцию. Например, мы можем создавать процедуры, которые заставляют спрайты рисовать фигуры, выполнять сложные вычисления, запускать пользовательскую анимацию, проигрывать последовательности

нот, управлять играми и делать многое другое. Созданные однажды, эти процедуры могут служить кирпичиками, из которых складываются разнообразные полезные программы.

Отправка и получение сообщений

Как же работает система отправки сообщений в Scratch? Любой спрайт может передать сообщение (вы можете называть его как хотите), используя блоки **передать** или **передать и ждать** (из раздела **События**), как показано на рис. 4.1. Сообщение получают все спрайты, но реагируют на него только те, у которых есть соответствующий блок **когда я получу**. Они исполняют привязанные к этому блоку инструкции.

Рис. 4.1. Вы можете использовать блоки передачи и получения сообщений для координации работы нескольких спрайтов

Посмотрим на рис. 4.2. На нем показаны четыре спрайта: Морская звезда, Лягушка, Кот и Летучая мышь (Starfish, Frog, Cat1 и Bat1). Морская звезда передает сообщение о прыжке, и его получают все спрайты, включая ее саму. В ответ на него кот и лягушка запускают свои скрипты прыжка. Причем каждый прыгает по-своему, выполняя разный скрипт. Летучая мышь тоже получает сообщение о прыжке, но ничего не делает: она не получила команды, что делать, если получит такое сообщение. Кот на этом рисунке умеет ходить и прыгать, лягушка — только прыгать, а летучую мышь научили только летать.

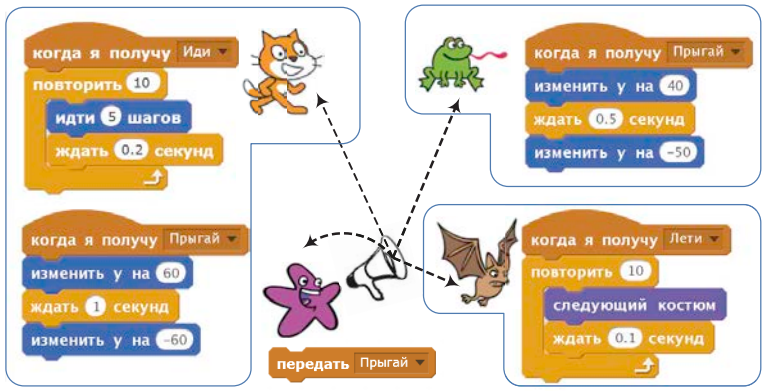


Рис. 4.2. Сообщение принимают все спрайты, даже тот, который его отправил

Команда **передать и ждать** работает так же, как команда **передать**, но прежде чем продолжить действовать, она ждет, пока получатели сообщения выполнят свои блоки **когда я получу**.

Отправка и получение сообщений

SquareApp.sb2

Чтобы показать, как работает передача и получение сообщений, создадим простое приложение, рисующее квадраты случайных цветов. Когда пользователь кликает по **Сцене** левой кнопкой мыши, та распознает это событие (используя функцию **когда спрайт нажат**) и передает сообщение, которое мы назовем **Квадрат** (можно выбрать любое другое название). Единственный спрайт в этой программе, получив сообщение, движется к текущей позиции мыши и рисует квадрат. Вот пошаговая инструкция по созданию программы.

1. Запустите Scratch и выберите **Новый** из меню **Файл**, чтобы создать новую программу. Вы можете заменить костюм кота любым другим.

2. Добавьте блок **когда я получу** (из раздела **События**) в поле скриптов спрайта. Кликните по стрелочке вниз на этом блоке и выберите из выпадающего меню **новое сообщение ...** . В появившемся диалоговом окне напишите **Квадрат** и нажмите **OK**. Название блока должно измениться на **когда я получу Квадрат**.

3. Завершите работу над скриптом, как показано на рис. 4.3. Спрайт сначала поднимает перо и перемещается на текущую позицию мыши, обозначенную блоками **мышка по x** и **мышка по y** (из раздела **Сенсоры**). Затем он выбирает случайный цвет пера, опускает его и рисует квадрат.

Теперь спрайт готов к обработке сообщения **Квадрат**, когда оно придет. Скрипт на рис. 4.3 можно назвать *обработчиком сообщений*.

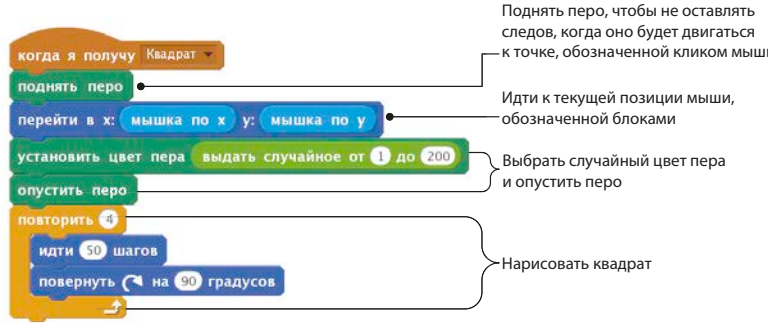


Рис. 4.3. Обработчик сообщения **Квадрат**

Теперь перейдем на **Сцену** и добавим код, чтобы передать сообщение **Квадрат** по клику мыши. Кликните по **Сцене** в списке спрайтов и добавьте два новых скрипта, как показано на рис. 4.4. Первый убирает все следы **Пера** на **Сцене**, как только вы кликнете по зеленому флажку. Второй выполняется при клике мышью по **Сцене**, посылая при помощи блока **передать** сообщение, получив которое спрайт начинает рисовать.

После клика по зеленому флажку очистить **Сцену**. Когда пользователь кликает мышью по **Сцене**, передать сообщение **Квадрат**

Рис. 4.4. Два скрипта для **Сцены** *в программе, рисующей квадраты*

Программа готова. Чтобы протестировать ее, кликните мышью по **Сцене**. В ответ она должна начать рисовать квадраты после каждого клика мышью.

Передача сообщений для координирования нескольких спрайтов

Flowers.sb2

Чтобы посмотреть, как несколько скриптов реагируют на одно и то же сообщение, создадим программу, рисующую на **Сцене** цветы по клику мышью. В программе *Flowers* пять спрайтов — от Цветок1 до Цветок5 (Flower1 ... Flower5), — которые отвечают за рисование пяти разных цветов.

У каждого спрайта свой костюм, как показано на рис. 4.5. Обратите внимание на то, что фон каждого костюма прозрачен, а также на то, где расположены центры их вращения (отмечены пересечением линий).

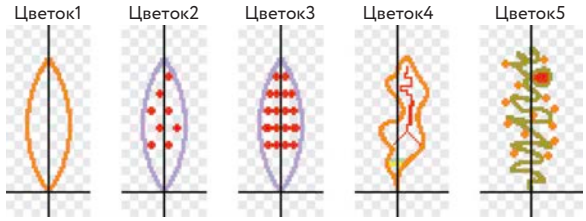


Рис. 4.5. Программа Flowers использует эти пять спрайтов-лепестков (как показано в графическом редакторе)

Когда спрайт получает сообщение с командой рисовать цветок, он начинает отпечатывать на **Сцене** поворачивающиеся копии своего костюма, как показано на рис. 4.6.

Рисунок также показывает образцы результатов работы скрипта для рисования цветов, который мы сейчас рассмотрим.

Рис. 4.6. Процесс рисования в программе Flowers (слева) и возможный вид цветов (справа)

Когда вы кликаете мышью по **Сцене**, та получает информацию об этом при помощи блока **когда спрайт нажат**. В ответ на этот сигнал она очищает фон и передает сообщение **Рисовать**. Все пять спрайтов реагируют на это сообщение, начиная выполнять свои скрипты, как показано на рис. 4.7.

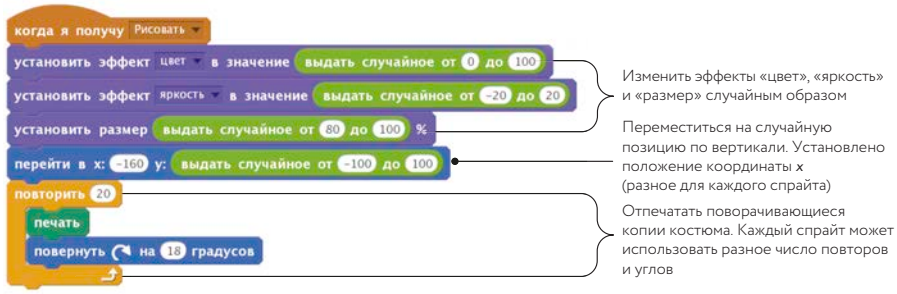


Рис. 4.7. Основной скрипт для каждого из пяти спрайтов

Первым делом скрипт присваивает случайные значения эффектам цвета и яркости, а также размеру, чтобы изменить внешний вид цветка. Затем он перемещается на случайную позицию на **Сцене** по вертикали и рисует цветок, отпечатывая поворачивающиеся копии своего костюма.

Откройте программу *Flowers.sb2* и запустите ее, чтобы посмотреть, как она работает. Несмотря на простоту, результат впечатляет. Предлагаю создать разные костюмы, чтобы получить разные типы цветов. Поменяйте центр костюмов, чтобы получить еще больше интересных вариантов.

Теперь, когда вы понимаете, как работают передача и получение сообщений, перейдем к структурному программированию как методу управления сложными или большими программами.

Создаем большие программы маленькими шажками

Написанные вами до настоящего момента скрипты были короткими и простыми. Скоро вы будете писать длинные и сложные скрипты, содержащие сотни блоков. А понять их структуру и управлять ими непросто. Подход, известный как *структурное программирование*, был разработан в середине 1960-х, чтобы упростить процесс написания компьютерных программ и поддержки их работоспособности. Вместо того чтобы писать одну большую программу, вы можете разделить ее на несколько фрагментов, каждый из которых решает часть общей задачи.

Представьте себе процесс выпекания пирога. Возможно, вы не задумываетесь об отдельных его шагах, но следуете четкому рецепту, который перечисляет необходимые действия. Он может включать в себя такие инструкции, как «смешайте 4 яйца, 60 г муки и 1 стакан воды»; «вылейте смесь в форму»; «поставьте форму в духовку»; «пеките 1 час при температуре 180°» и т. д. По сути, рецепт разбивает задачу выпекания пирога на определенные логические действия.

А когда вы создаете решение для задачи программирования, проще разбить задачу на более «обозримые» фрагменты. Такой подход позволит вам четко понимать программу и взаимоотношения между ее компонентами.

Взгляните на рис. 4.8. На нем показан длинный скрипт для рисования на **Сцене** простой формы. Вы видите, что он может быть разделен по признаку функций на небольшие логически обоснованные блоки. Первые шесть, например, приводят спрайт в исходное положение.

Первый блок **повторить** рисует квадрат, второй — треугольник и т. д. Структурное программирование позволяет сгруппировать родственные блоки под общим названием, чтобы сформировать процедуру.

Написав процедуры, мы можем вызывать их в определенной последовательности, чтобы решить нашу задачу. На рис. 4.8 видно, как отдельные процедуры могут быть объединены вместе, чтобы выполнять ту же функцию, что и исходный скрипт. Очевидно, что скрипт, использующий процедуры (справа), более универсален и понятен, чем оригинал (слева).

Процедуры также помогут вам избежать переписывания одного и того же кода дважды. Если набор команд выполняется в нескольких местах в рамках одной программы, вы можете написать процедуру, которая отвечает за них, и использовать его. Такая стратегия для избегания дублирования называется *повторным использованием кода*. Например, процедура **Нарисовать квадрат** была повторно использована на рис. 4.8.

Использование процедур дает вам возможность применить к решению сложных задач стратегию «разделяй и властвуй». Вы делите большую и сложную задачу на подзадачи и решаете их отдельно, тестируя каждую из них. После того как готово решение для всех подзадач, вы

складываете фрагменты так, чтобы получить решение исходной задачи. Это похоже на стратегию выпекания пирога: мы совершаем нужные шаги в правильном порядке, чтобы получить конечный продукт.

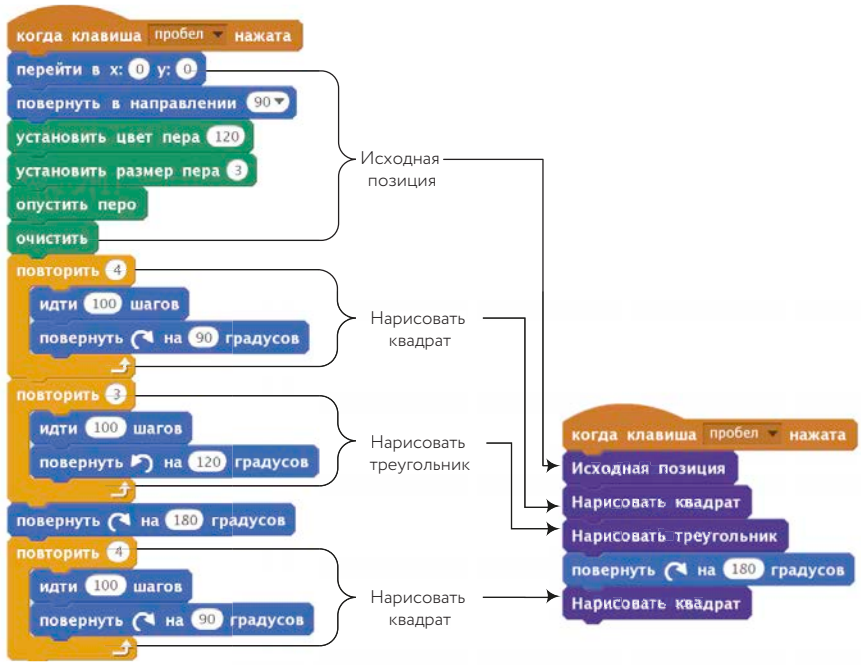


Рис. 4.8. Разбиваем большой скрипт на логические части, каждая из которых выполняет одну функцию

Здесь вы могли бы спросить: «А как же создаются эти процедуры?» До появления среды Scratch 2 вы не смогли бы создать блок **Исходная позиция**, показанный на рис. 4.8, а затем вызвать его из вашего скрипта. Единственным способом имитировать процедуры и сделать программу более структурированной тогда был механизм *передачи сообщений*. Это изменилось с появлением среды Scratch 2 и мощной функции пользовательских блоков.

В этом разделе мы покажем, как все делалось по старинке: именно такой способ решения вы увидите в скриптах, созданных в более старой версии Scratch. А функцию самостоятельной разработки блоков мы разберем в следующем разделе и будем постоянно ею пользоваться.

Поскольку спрайты получают сообщения, которые сами отправляют, мы можем внедрить идею процедур, дав спрайту команду передать сообщение себе и выполнить указанное в нем задание по сигналу блока-триггера **когда я получу**. Можно использовать команду **передать и ждать**, чтобы гарантировать, что наши процедуры будут вызываться

в правильной последовательности. Тогда наши программы будут более структурированными и построенными по модульному принципу. Ну что, я вас совсем запутал? Тогда посмотрим, как это работает.

Создаем процедуры при помощи передачи сообщений

Flowers2.sb2

Мы рассмотрим, как работают процедуры и как они могут улучшить ваш код, на примере уже знакомой нам программы *Flowers*, которую мы воссоздадим.

Откройте файл *Flowers2.sb2*, который содержит новую версию программы. Скрипт для **Сцены** остался таким же, как и раньше (она передает сообщение **Рисовать**, когда распознает клик мышью), но на этот раз наша программа использует только один спрайт вместо пяти. У него пять костюмов: leaf1, leaf2 и так до leaf5, — и он вызовет процедуру, чтобы нарисовать цветок для каждого костюма.

Поскольку спрайт у нас один, нам понадобится только одна копия кода для рисования (а не пять дублированных скриптов, как в первой версии). Программа будет меньше, а код — более понятным. Когда в этой программе спрайт получает сообщение **Рисовать**, он выполняет скрипт, показанный на рис. 4.9.

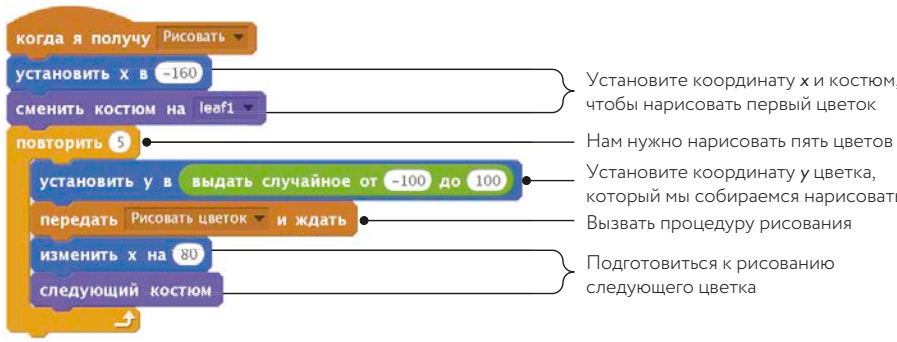


Рис. 4.9. Когда спрайт получает сообщение **Рисовать**, *он вызывает* **Рисовать цветок** *пять раз (подряд), чтобы отобразить пять цветов*

Скрипт устанавливает координату *x* и костюм, чтобы начать рисовать первый цветок, а затем входит в цикл, чтобы создать пять цветов. При каждом повторе цикл устанавливает для нового цветка координату *y* и вызывает команду **Рисовать цветок**, передавая самому себе сообщение. Этот вызов останавливает выполнение скрипта до того момента, когда будет выполнена команда **Рисовать цветок**. Когда это произойдет, скрипт **Рисовать** продолжит работу, поменяв координату *x* и костюм для рисования следующего цветка.

Вы можете назвать процедуру как хотите, но я советую выбрать имя, которое отражает ее назначение. Это поможет вам, когда вы вернетесь к программе, написанной много месяцев назад. Например, если вы хотите показать игрокам, сколько баллов они набрали, вы можете создать процедуру под названием «Показать счет». Если вы назовете такой скрипт «Мэри» или «Альфред», это вряд ли напомнит вам (или человеку, читающему вашу программу), какую функцию он выполняет.

Процедура **Рисовать цветок** показана на рис. 4.10. Она задает случайные значения для эффектов цвета и яркости, а также размера спрайта, прежде чем отпечатать поворачивающиеся копии текущего костюма скрипта и тем самым нарисовать цветок.

Первая созданная нами версия программы содержала 5 спрайтов и 5 повторяющихся скриптов. Вторая достигает того же эффекта при помощи одного спрайта, который запускает процедуру для рисования всех пяти цветов. Откройте файлы *Flowers.sb2* и *Flowers2.sb2* в двух окнах браузера и сравните их. Разве не проще отслеживать логику новой версии? Использование процедур позволяет вам создавать более компактные программы, которые легче понять и с которыми удобнее работать. По мере того как вы будете писать всё более сложные программы, решающие более комплексные задачи, это преимущество будет становиться всё существеннее.

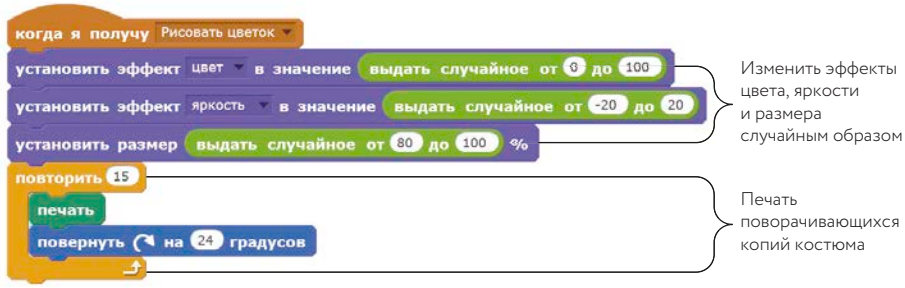


Рис. 4.10. Процедура **Рисовать цветок**

Создайте свой блок

В среде Scratch 2 можно создавать свои блоки. Новый блок появится в разделе **Другие блоки**, где вы сможете использовать его точно так же, как и любые другие блоки в Scratch.

Чтобы показать вам, как создавать и использовать такие блоки, мы модифицируем программу *Flowers2*, которую обсудили выше, так, чтобы использовать пользовательский блок для процедуры **Рисовать цветок**. Вот пошаговая инструкция по созданию этой новой версии.

1. Для начала откройте файл *Flowers2.sb2*, который мы уже рассматривали.

   Выберите **Файл — Скачать на свой компьютер** из меню **Файл** и сохраните файл как *Flowers3.sb2*. Можете выбрать и другое название.

2. Кликните по иконке спрайта Цветок, чтобы выбрать его. Затем выберите раздел **Другие блоки** и в нем функцию **Создать блок**. Вы увидите диалоговое окно, как на рис. 4.11 (слева). Введите «Рисовать цветок» в качестве названия блока и нажмите **OK**. Теперь в разделе **Другие блоки** должен появиться новый **Рисовать цветок**, а блок **определить «Рисовать цветок»** — в поле скриптов, как показано на рисунке справа.

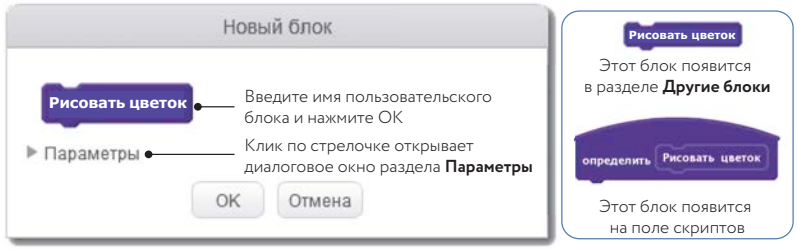


Рис. 4.11. Диалоговое окно **Новый блок** *и блоки, которые появятся после создания пользовательского блока* **Рисовать цветок**

3. Отсоедините скрипт, прикрепленный к блоку **когда я получу «Рисовать цветок»**, и присоедините его к блоку **определить «Рисовать цветок»**, как показано на рис. 4.12. Получается новая процедура, которую мы назвали **«Рисовать цветок»**, внедренная нами в виде пользовательского блока. Удалите блок **когда я получу «Рисовать цветок»**: он нам больше не нужен.

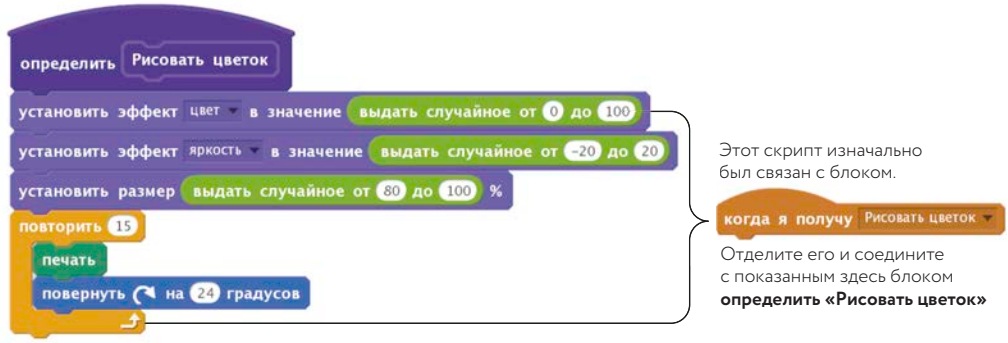


Рис. 4.12. Процедура **Рисовать цветок** *внедрена в виде пользовательского блока*

4. Теперь, когда мы создали процедуру **Рисовать цветок**, нам нужно вызывать ее из нашего обработчика сообщения **Рисовать**. Измените сообщение, как показано на рис. 4.13. Мы всего лишь заменили блок **передать «Рисовать цветок» и ждать** нашим новым пользовательским блоком **Рисовать цветок**.

Рис. 4.13. Вызов функции **Рисовать цветок** *из обработчика сообщения* **Рисовать**

Теперь программа готова, и вы можете протестировать ее. Кликните мышкой по **Сцене**, чтобы удостовериться, что программа работает как раньше. Изучите в разделе «Запуск без обновления экрана», как можно ускорить выполнение этой программы.

Теперь, когда вы в курсе основных принципов работы пользовательских блоков, вы можете пойти дальше и начать делать блоки, принимающие введенные данные.

ЗАПУСК БЕЗ ОБНОВЛЕНИЯ ЭКРАНА

Внедрение процедуры **Рисовать цветок** пользовательскими блоками подводит нас к еще одной функции, которая позволяет уменьшить время выполнения скрипта. Чтобы продемонстрировать ее, сделаем следующее.

1. Кликнем правой кнопкой мыши по блоку **Рисовать цветок** в разделе **Другие блоки** и выберем в выпадающем меню строку **редактировать**. Появится такое же диалоговое окно, как на рис. 4.11, только называться оно будет не **Новый блок**, а **Редактировать блок**.

2. Кликните по стрелке вниз в графе **Параметры**, поставьте галочку в окошке **Запуск без обновления экрана** и нажмите **OK** (см. рис. 4.15 на с. 98).

3. Теперь кликните мышью по **Сцене** и посмотрите, что произойдет. Вместо постепенного вращения и печати пяти цветов вы увидите, что они появятся *почти* одновременно. И вот почему.

Процедура **Рисовать цветок** содержит много блоков, которые меняют внешний вид спрайта, в том числе **установить цвет**, **установить яркость**, **установить размер** и **печать**.

После выполнения такого блока обычно Scratch делает небольшую паузу, чтобы *обновить* (нарисовать заново) экран. Вот почему раньше мы могли наблюдать процесс рисования.

Если вы выбираете опцию **Запуск без обновления экрана**, блоки будут запускаться без пауз на обновление экрана, что позволит процедуре выполняться гораздо быстрее. Экран обновится после того, как Scratch выполнит весь алгоритм. Это также помогает предотвратить мелькание, которое может возникнуть при повторяющемся перерисовывании.

Как присвоить пользовательским блокам новые параметры

Давайте для начала создадим блок под названием **Квадрат**, который рисует квадрат со стороной 100 пикселов, как показано на рис. 4.14.

Рис. 4.14. Процедура **Квадрат***, рисующая квадраты фиксированного размера*

Процедура **Квадрат** имеет ограниченные возможности, поскольку размеры квадрата, который она рисует, были однажды зафиксированы раз и навсегда. А что если вы хотите рисовать квадраты с разной длиной сторон, например 50, 75 или 200? Вы могли бы сделать несколько пользовательских блоков и назвать их **Квадрат50**, **Квадрат75** и **Квадрат200**, но обычно много блоков, делающих практически одну и ту же работу, не лучший вариант. Если вам нужно будет что-то изменить, придется отслеживать все копии и менять их тоже. Более удачное решение — сделать один блок **Квадрат**, который позволит пользователю указать желаемую длину стороны, а затем запустить его.

Этот способ использовался еще в главе 1. Например, Scratch дает нам один блок **идти**, который позволяет указать, сколько шагов нужно сделать спрайту, введя число в специальное поле. И не нужно обеспечивать отдельный блок для каждого возможного расстояния.

Итак, нужно добавить блоку **Квадрат** поле, куда пользователь сможет вводить длину стороны. Рис. 4.15 показывает, как нужно модифицировать блок.

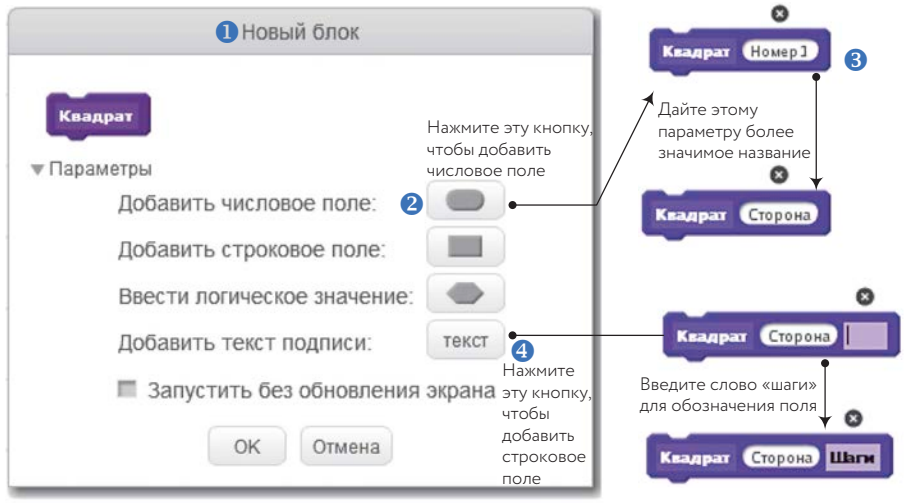


Рис. 4.15. Добавляем в блок **Квадрат** *числовое поле*

Для начала кликните правой кнопкой мыши по блоку **Квадрат** из раздела **Другие блоки** (или блоку **определить Квадрат** в поле скриптов) и выберите из выпадающего меню графу **редактировать**, чтобы вызвать диалоговое окно **Редактировать блок** ❶. Кликните по маленькой стрелочке рядом с графой **Параметры**, чтобы развернуть окно и увидеть доступные варианты.

Мы хотим, чтобы пользователь вводил желаемую длину стороны для квадрата. Эта длина — цифра, так что кликните по **Добавить числовое поле** ❷. В блоке появится числовое поле под названием Номер1.

Чтобы обозначить, что новое поле будет показывать длину стороны квадрата, измените его исходное название на что-то более осмысленное, скажем **длина** или **длина стороны** ❸. (Повторюсь, для Scratch неважно, каким названием вы пользуетесь, это существенно для вас! Выберите то, которое отражает суть параметра.) Для примера используем слово «сторона».

Технически это всё, что нам нужно сделать, чтобы добавить числовое поле в нашу процедуру. Нажав **OK**, мы получим блок **Квадрат** с числом в качестве дополнительной информации. Мы могли бы перетащить его в наши скрипты и указать нужную длину стороны в числовом поле, как у блока **Квадрат50**. Но как пользователь узнает, что означает номер,

присвоенный **Квадрату**? Может, площадь 50, диагональ 50 или длина стороны 50, а может, вообще что-то другое? Представьте себе, что блок **плыть** устроен так.

Как бы вы узнали, что первое цифровое поле — время (в секундах), а второе и третье — координаты *x* и *y* точки назначения? Создатели Scratch сделали блок **плыть** более простым для понимания и использования, добавив полям названия.

Сделаем то же и для блока **Квадрат**: добавим текст, который описывает значение (или функцию) числового поля. Кликните по **Добавить строковое поле** 4, как показано на рис. 4.15, чтобы добавить название параметру длины стороны квадрата.

Впишите в качестве описания поля слово **шагов** и нажмите **OK**.

Теперь, если вы посмотрите на определение процедуры **Квадрат** в поле скриптов, вы увидите, что к его шапке добавился маленький блок (под названием Сторона), как показано на рис. 4.16 (слева). Блок **идти** сохраняет фиксированный показатель 100, но теперь нам достаточно перетянуть блок Сторона из шапки и отпустить его над цифровым полем блока **идти**, чтобы заменить число, как показано на рис. 4.16 (справа).

Рис. 4.16. Модифицируем процедуру **Квадрат**, *чтобы использовать параметр* Сторона

Значок **сторона**, который появляется в шапке процедуры **Квадрат**, называется *параметром*. Вы можете представить себе, что это заглушка с названием.

Мы хотим, чтобы наш скрипт «Квадрат» был способен рисовать квадраты любого размера, поэтому, вместо того чтобы жестко прописывать в нашей процедуре фиксированное число, мы используем общий параметр Сторона. Пользователи будут указывать конкретное

значение. Проиллюстрируем это, изменив скрипт на рис. 4.14, и воспользуемся новой версией процедуры. Необходимые изменения показаны на рис. 4.17.

Рис. 4.17. Вызов процедуры **Квадрат** *с установленной стороной квадрата 100*

Здесь число 100 (так называемый *аргумент*) внедряется в процедуру **Квадрат**. Когда она выполняется, параметру Сторона присваивается значение 100, которое потом используется во всех ситуациях внутри процедуры.

Как вы могли заметить, возможность прописать в алгоритме различные аргументы — мощный инструмент, который добавляет гибкости вашим программам.

Мы можем усовершенствовать процедуру **Квадрат**, заставив ее принять цвет квадрата в качестве еще одного параметра, как показано на рис. 4.18. Здесь мы добавили второй входной параметр номерЦвета. Он обозначает желаемый цвет квадрата. Теперь процедура будет устанавливать цвет пера согласно указанному номеру, прежде чем начать цикл рисования. Измените блок **Квадрат**, чтобы внести усовершенствования, как показано на рисунке.

ПАРАМЕТРЫ ИЛИ АРГУМЕНТЫ?

Хотя многие программисты используют понятия «*параметр*» и «*аргумент*» как взаимозаменяемые, на самом деле это два разных термина. Чтобы понять, в чем различие, посмотрите на процедуру **Среднее** внизу, которая вычисляет среднее арифметическое двух чисел.

Получается, у этой процедуры два параметра: номер1 и номер2.

Параметр определяет дополнительную информацию для алгоритма. Вы запустите процедуру вместе с блоком, показанным слева, и укажете некие значения или выражения в свободных полях. Значения 100 и 50 в приведенном выше примере называются *аргументами* процедуры.

Само собой, количество аргументов в процедуре должно соответствовать количеству параметров в определении алгоритма. Когда вы запускаете процедуру **Среднее**, параметры номер1 и номер2 получают значение 100 и 50 соответственно, потому что позиции аргументов и параметров сочетаются.

Рис. 4.18. Эта версия процедуры **Квадрат** *использует желаемый цвет в качестве второго параметра*

УПРАЖНЕНИЕ 4.1

А как насчет толщины сторон квадрата? Измените процедуру **Квадрат**, добавив третий параметр. Назовем его Размер пера. Он будет показывать размер пера, используемый при рисовании квадрата.

В завершение раздела — несколько полезных советов по работе с пользовательскими блоками.

- Пользовательские блоки не могут применяться несколькими спрайтами. Если вы создаете блок, скажем, для Спрайта1, то только он может его использовать. А пользовательский блок для **Сцены** может использоваться только в шапке скрипта, который относится к **Сцене**.

- Давайте вашим параметрам названия со смыслом. Они должны указывать на то, для чего используется параметр.

- Чтобы удалить пользовательский блок, перетащите его блок **определить** (блок-шляпу) из поля скриптов и отпустите в зоне раздела **Блоки**. Вы можете удалить блок **определить**, только если ваш скрипт не содержит связанных с ним пользовательских

блоков. Сначала нужно удалить все случаи использования вашего пользовательского блока из скриптов; только потом вы сможете удалить его.

- Чтобы удалить параметр пользовательского блока, кликните по названию параметра в диалоговом окне **Редактировать блок**, а потом по значку ×, который появится над окошком параметра.

- Вы также можете вводить строковые и булевы параметры. Мы поговорим более подробно о типах данных в следующей главе, когда речь пойдет о переменных.

Сейчас вы, должно быть, задались вопросом: а может ли одна процедура обращаться к другой?

Скоро вы узнаете, как пользоваться обращениями к вложенным процедурам, чтобы расширить возможности уже существующих и сделать их еще более полезными.

Используем вложенные процедуры

Как мы уже отмечали выше, процедура должна выполнять одну хорошо сформулированную задачу. Для выполнения нескольких задач совершенно логично — а во многих случаях и желательно — сделать так, чтобы одна процедура обращалась к другой. Вложенные процедуры дают возможность быть более гибкими при структурировании и организации программ.

Чтобы посмотреть, как это работает, начнем с процедуры **Квадрат**, которая была нами написана в предыдущем разделе (см. рис. 4.17). Теперь мы создадим новую процедуру под названием **Квадраты**, которая будет рисовать четыре растягивающихся квадрата, как на рис. 4.19, путем четырехкратного обращения к процедуре **Квадрат**. Каждое обращение использует новый аргумент, а в результате получаются четыре квадрата с общим углом.

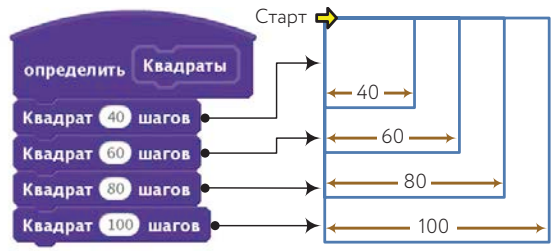


Рис. 4.19. Процедура **Квадраты** *и результат ее работы*

Мы можем теперь использовать квадраты, чтобы создать интересные картины. Рис. 4.20 показывает другую процедуру, которая называется **Вращающиеся квадраты**. Она несколько раз обращается к процедуре **Квадраты** и при каждом обращении поворачивает квадраты под определенным углом. RotatedSquares.sb2

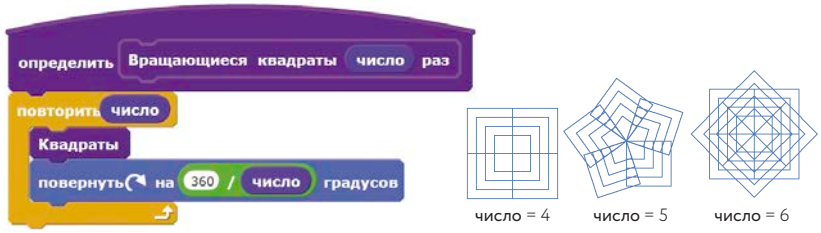


Рис. 4.20. Процедура **Вращающиеся квадраты** *и возможные результаты ее работы*

В этой процедуре параметр число используется дважды: первый раз для того, чтобы установить количество повторов, второй — чтобы вычислить угол поворота фигуры после обращения к процедуре **Квадраты**. Установив для этого параметра значение 5, мы получим узор из повторяющихся пять раз с правым поворотом под углом 72° (360° / 5) квадратов, как на рис. 4.20. Поэкспериментируйте с другими значениями, чтобы получить новые узоры.

Поработаем над другим примером, который демонстрирует возможности вложенных процедур. Мы начнем с процедуры **Квадрат** с рис. 4.16, а закончим шахматной доской.

Создайте новую процедуру (назовем ее **Ряд**), которая будет рисовать один ряд квадратов, как показано на рис. 4.21. Обратите внимание: количество квадратов, которые нужно нарисовать, задается в качестве параметра. Чтобы не усложнять, мы зафиксировали размер отдельного квадрата в 20 шагов, вместо того чтобы определять размер как второй параметр процедуры **Ряд**.

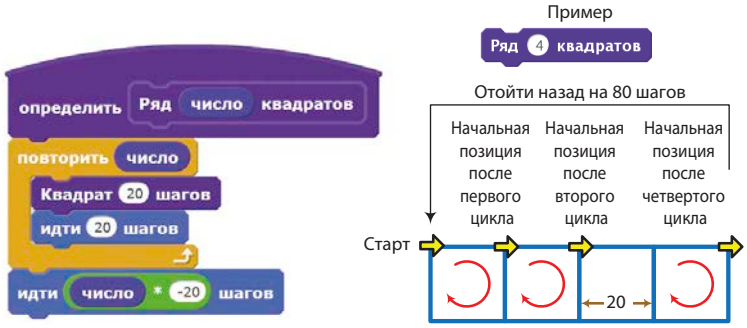


Рис. 4.21. Процедура **Ряд**

Рис. 4.21 также демонстрирует результат работы процедуры **Ряд** с аргументом 4, который заставляет ее четыре раза обращаться к блоку **Квадрат 20 шагов**.

Позиция спрайта после рисования каждого квадрата корректируется так, чтобы установить начальную точку для следующего квадрата. После того как нарисовано четыре квадрата, последняя команда возвращает спрайт в исходную позицию.

Checkers.sb2

Чтобы нарисовать еще один ряд квадратов под тем, который мы видим на рис. 4.21, нам нужно передвинуть спрайт вниз на 20 шагов и снова вызвать процедуру **Ряд**. Мы можем повторить ее, чтобы нарисовать нужное нам количество рядов. Именно это и делает процедура **Шашки** (**Checkers**), показанная на рис. 4.22.

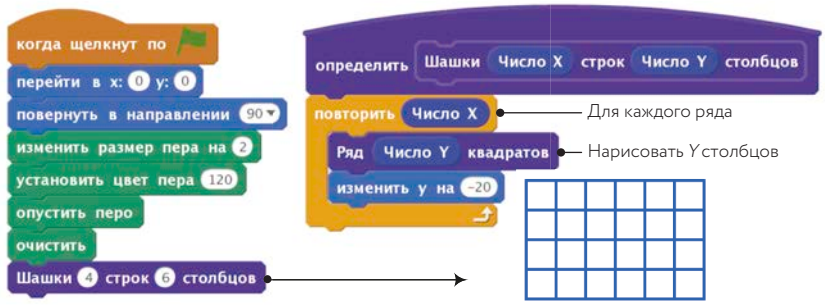


Рис. 4.22. Процедура **Шашки** *и результат ее работы*

Этой процедуре нужны два параметра: количество рядов и количество столбцов в желаемой клетке. После рисования каждого ряда процедура перемещает спрайт на 20 шагов вниз для подготовки к рисованию следующего ряда квадратов.

Примеры, приведенные в этом разделе, показывают, как процедуры могут помочь разделить программу на более мелкие и удобные в управлении фрагменты. После того как вы напишете и протестируете эти процедуры, вы будете использовать их как кирпичики при строительстве более сложных процедур, не особо волнуясь о процессах на нижнем уровне. Тогда вы сможете сосредоточиться на такой важной задаче, как сведение воедино всей программы.

УПРАЖНЕНИЕ 4.2

Как вы думаете, что будет, если изначально установить направление на 0° (вверх) вместо 90° (вправо)? Будет ли работать скрипт? Если нет, то как его исправить? Внесите изменения и запустите скрипт, чтобы проверить свои ответы.

Работа с процедурами

Теперь, когда вы знаете, почему так важно разбивать программу на более компактные части и разбираться с каждой из них отдельно, обсудим, как это разделение осуществляется. Все задачи разные, единого решения для всех нет и быть не может. Но именно поэтому разгадывать эту головоломку так интересно!

Мы сначала рассмотрим *нисходящий процесс* разделения большой программы на модульные фрагменты с четкой логической структурой. Потом мы обсудим другой способ создания сложных программ: *восходящий процесс* комбинирования уже существующих процедур. На рис. 4.23 показан общий план этих двух подходов.

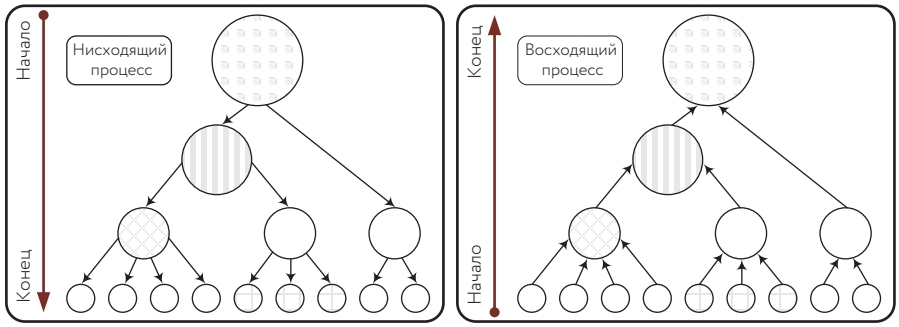


Рис. 4.23. Иллюстрация нисходящего (слева) и восходящего (справа) подходов

На обеих диаграммах задача, которую мы хотим решить, находится наверху, а отдельные шаги, из которых состоит наше решение, — у основания. Вы можете начать на любом уровне, так, как вам покажется разумнее.

Разбиваем программы на процедуры

Первый шаг в решении любой задачи программирования — полное понимание самой задачи. После этого вы можете спланировать общее решение и разделить его на основные части. Тут не бывает правильного или неправильного подхода. И по мере того как вы будете набираться опыта, вам будет все проще определять, какие задачи «основные». Если вы работаете от общего решения вниз, то это по меньшей мере гарантирует, что будет легче сохранять общую логику программы.

Чтобы продемонстрировать этот подход к решению задач, представим себе, как бы мы рисовали дом (рис. 4.24). House.sb2

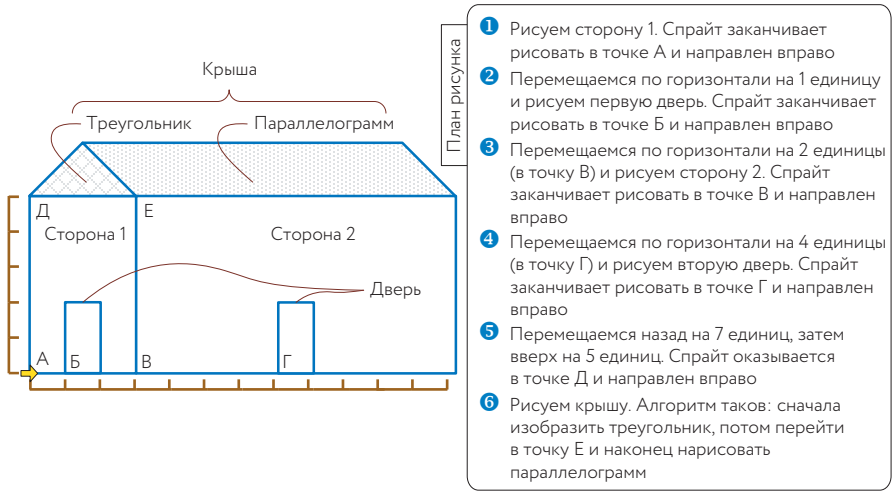


Рис. 4.24. Мы можем нарисовать дом, разделив задачу на несколько более мелких составляющих и управляя каждой из них отдельно

С одной стороны, работа над этой простой задачей позволяет сосредоточиться на стратегии поиска решения, не застревая в груде мелочей. С другой стороны, несмотря на очевидную простоту, задача имеет различные решения. Вот несколько вариантов.

- Мы можем представить себе дом состоящим из прямых линий. В таком случае основной задачей станет рисование каждой линии.

- Мы можем представить себе, что дом состоит из шести независимых фигур: сторона 1, сторона 2, две двери, треугольник и параллелограмм. Рисование каждой из этих фигур — основная задача.

- Поскольку две двери одинаковы, мы можем назвать одной из основных задач рисование двери и запустить ее дважды.

- Мы можем рассматривать треугольник и параллелограмм наверху дома как единое целое: крышу. В таком случае отдельной основной задачей будет нарисовать крышу.

- Мы можем рассматривать сторону 1 и дверь как одно целое: фасад дома. В этом случае отдельной задачей будет нарисовать фасад.

Есть много других вариантов, но этих достаточно, чтобы проиллюстрировать нашу идею. Нужно разделить общую задачу на небольшие,

обозримые подзадачи, с которыми проще работать и которые можно решать по одной. Если вы обнаруживаете схожие фрагменты, пробуйте найти для них единое решение и применяйте его ко всем таким случаям.

Наш план рисования дома, изображенный на рис. 4.24, тоже опирается на эту идею. Он подразумевает, что спрайт начинает в точке А и ориентирован направо. Надо создать скрипт, который подходил бы к шагам, прописанным в этом плане. Мы напишем процедуру **Сторона 1** (**Side1**), чтобы рисовать левую сторону дома, как показано в шаге 1. Мы также напишем еще три процедуры: **Дверь**, **Сторона 2** и **Крыша** (**Door, Side2** и **Roof**), чтобы нарисовать две двери, правую сторону дома и крышу (см. шаги 2, 3, 4 и 6), и соединим все эти процедуры с соответствующими командами **Движения**.

Наша процедура **Дом** изображена на рис. 4.25, где также показаны последовательно все этапы рисования и их привязка к процедуре. Ей требуется один параметр (масштаб), который определяет длину единицы (коэффициент масштабирования) при рисовании дома. Обратите внимание на то, что процедура **Дверь** используется дважды. А процедура **Крыша** отвечает за рисование всей крыши и может содержать различные подалгоритмы для рисования отдельных компонентов.

Рис. 4.25. Процедура **Дом**. *Обратите внимание на то, как основные задачи сопоставлены с планом рисования*

Отдельные процедуры для рисования дома показаны на рис. 4.26. Они изображают простые геометрические фигуры при помощи техник, которые вы узнали из главы 2.

Процедуры **Сторона 1**, **Дверь** и **Сторона 2** рисуют прямоугольники 3 × 5, 1 × 2 и 9 × 5 единиц соответственно. Процедура **Крыша** имеет два подалгоритма (**Треугольник** и **Параллелограмм**) для рисования двух частей крыши. Обратите внимание, что во всех этих случаях масштаб устанавливался показателем масштаб. Это позволяет нам рисовать дома разных размеров, вводя различные аргументы.

УПРАЖНЕНИЕ 4.3

Вы обратили внимание на то, что процедуры **Сторона 1**, **Дверь** и **Сторона 2** используют практически одинаковый код? Создайте новую процедуру под названием **Прямоугольник**, которая в качестве параметров использует длину, ширину и масштаб и рисует прямоугольник с указанными размерами.

Измените процедуры **Сторона 1**, **Дверь** и **Сторона 2** так, чтобы обратиться к новой процедуре **Прямоугольник**.

Рис. 4.26. Процедуры для рисования дома на рис. 4.24

Сборка программы из процедур

FlowerFlake.sb2

Другой способ работы с крупной задачей — сосредоточиться на мелких деталях. Если вы сначала решите мелкие части большой задачи (или найдете существующие решения), то сможете скомпоновать результаты, создав окончательное высокоуровневое решение.

Чтобы продемонстрировать этот метод, начнем с простой процедуры **Лист** (**Leaf**), которая рисует один лист, как показано на рис. 4.27.

Процедура содержит цикл **повторить**, который срабатывает дважды, чтобы нарисовать две половинки листа. Каждая рисуется как серия из 15 коротких фрагментов линий, расположенных относительно друг друга под углом 6°. Это похоже на метод, которым мы рисовали многоугольники в главе 2.

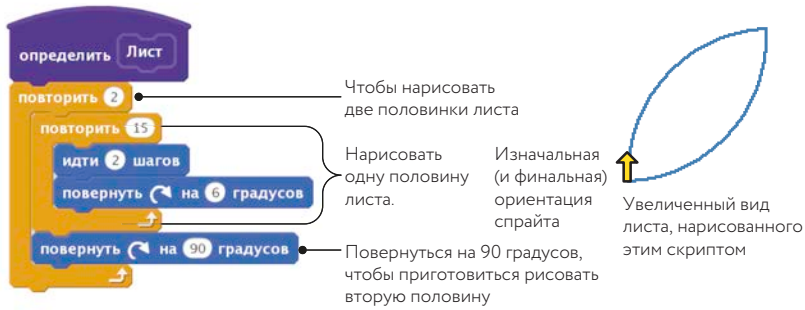


Рис. 4.27. Процедура ***Лист*** *и результаты ее работы*

Используя эту процедуру как отправную точку, мы можем рисовать более сложную форму, состоящую из пяти листьев. Наша новая процедура, которую мы назовем **Листья** (**Leaves**), и результаты ее работы показаны на рис. 4.28. Как вы видите, нам понадобилось только добавить процедуру **Лист** в цикл **повторить** с необходимым углом поворота между листьями.

Рис. 4.28. Процедура ***Листья*** *обращается к процедуре* ***Лист*** *пять раз с поворотом на 72° при каждом обращении*

Теперь мы можем использовать процедуры **Лист** и **Листья**, чтобы сделать что-то еще более сложное: ветку с листьями на ней. Наша процедура **Ветка** (**Branch**) и результат ее работы показаны на рис. 4.29. Спрайт перемещается вперед на 40 шагов, рисует один лист (обратившись к одноименной процедуре), перемещается еще на 50 шагов вперед, рисует пять листьев (обратившись к процедуре **Листья**) и наконец возвращается на исходную позицию.

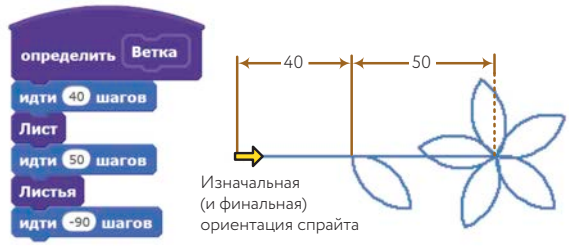


Рис. 4.29. Процедура **Ветка** *и результат ее работы*

Поднимемся еще на ступеньку вверх. Как насчет того, чтобы использовать процедуру **Ветка** для создания сложного рисунка цветка?

Новую процедуру мы назовем **Цветок** (**Flower**), а на результат ее работы можно посмотреть на рис. 4.30. Алгоритм шесть раз по кругу обращается к процедуре **Ветка**, при каждом повторе изображение поворачивается под углом 60°.

Мы можем продолжать бесконечно, но идея уже ясна. Мы начали с простой процедуры под названием **Лист** и использовали ее в новой (**Листья**), чтобы создать более сложный рисунок. Процедура **Ветка** опирается на эти два алгоритма при создании еще более сложного объекта. Процедура **Цветок** использует **Ветку**, чтобы создать еще более сложный рисунок.

Если бы мы хотели, мы могли бы создать алгоритм, рисующий целое дерево с цветами, а потом еще один, рисующий сад, полный деревьев.

Рис. 4.30. Процедура **Цветок** *и результат ее работы*

Независимо от сложности задачи мы всегда можем создать решение, объединив несколько небольших, более простых в управлении частей. Используя эту технику, мы начинаем с коротких процедур, которые решают простые задачи, а затем используем их для создания гораздо более сложных процедур.

Итоги

В этой главе мы познакомились с рядом основополагающих идей, которыми будем активно пользоваться на протяжении всей книги. Первым делом мы разобрались с передачей сообщений для коммуникации между спрайтами и синхронизации их работы. Затем мы ввели понятие структурного программирования и обсудили методы использования передачи сообщений при внедрении процедур. Мы показали, как работают пользовательские блоки в среде Scratch 2, и разобрались, как внедрять в процедуры аргументы, чтобы сделать алгоритмы более гибкими. Мы привели несколько примеров, показывающих, как крупная задача делится на более мелкие, удобные для работы части, и объясняющих, как использовать процедуры в качестве строительного материала при создании больших программ.

Наконец, мы рассмотрели восходящую технику решения задач, при которой мы складываем уже имеющиеся решения, чтобы получить результат для масштабной задачи.

В следующей главе вы узнаете о самом главном в любом языке программирования: о переменных. Это станет решающим шагом на пути к мастерству программиста.

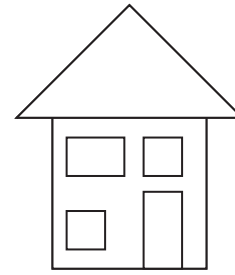
Задания

1. Разработайте разные процедуры для написания каждой буквы вашего имени. Дайте каждой название по той букве, которую она рисовала. Теперь напишите скрипт, который обращается к этим процедурам, чтобы вы могли написать свое имя на **Сцене**.

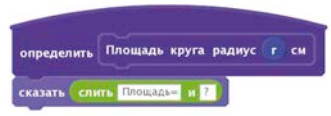
2. Создайте программу, показанную ниже, и объясните, как она работает.

3. Напишите процедуру, которая будет переводить градусы Цельсия в градусы Фаренгейта, как показано ниже. Сделайте так, чтобы процедура округляла числа до ближайшего целого. Протестируйте свой алгоритм при разных температурах. (Подсказка: °F = 9 / 5 × °C + 32.)

4. Напишите процедуру для создания дома, показанного справа. Сначала создайте маленькие процедуры для рисования мелких фрагментов (дверей, крыши, окон и т. д.). Затем соедините эти процедуры, чтобы нарисовать дом.

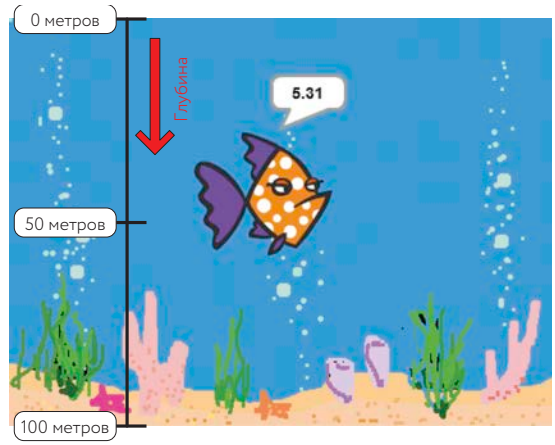


5. Напишите процедуру для вычисления площади круга ($A = \pi r^2$) при известном радиусе, как показано внизу. Используйте $\pi = 3{,}14$.



PressureUnderWater_NoSolution.sb2

6. В этом упражнении вам нужно имитировать давление, которое испытывает рыба в воде. Предположим, давление *P* (атмосфер), которое ощущает рыба, зависит от глубины *d* (в метрах от поверхности) согласно формуле: *P* = 0,1d + 1. Файл *PressureUnderWater_NoSolution.sb2* частично содержит реализацию этой задачи. Допишите скрипт так, чтобы рыба сообщала, какое давление она ощущает, когда плавает, как на рисунке ниже.



5

ПЕРЕМЕННЫЕ

В этой главе рассказывается, как создавать скрипты, которые умеют считывать и запоминать значения. Используя переменные, вы сможете писать приложения, которые взаимодействуют с пользователями и реагируют на введенную ими информацию. Вот темы, которые раскрываются в этой главе:

- какие типы данных поддерживает среда Scratch;
- как создавать значения и управлять ими;
- как получать информацию от пользователей и писать интерактивные программы.

Скрипты, которые вы писали в четырех предыдущих главах, помогли вам приобрести важные навыки программирования в Scratch, но в них отсутствовали многие ключевые элементы полномасштабного приложения. Более замысловатые программы способны запоминать значения и самостоятельно принимать решение, какие действия совершать при тех или иных условиях. Данная глава заполнит первый из двух этих пробелов, а следующая будет посвящена принятию решений.

В процессе выполнения скрипты обрабатывают различные типы данных и манипулируют ими. Данные могут быть введены в блоки-команды (например, число 10 вставлено в команду **идти 10 шагов**, а строка «Привет!» — в команду **сказать Привет!**); поступать из блоков-функций (например, **мышка по x, y** и **выдать случайное**); вводиться пользователем в ответ на команду **спросить и ждать**. В более сложных программах часто приходится сохранять и видоизменять данные для выполнения

определенных задач. Управление данными в Scratch осуществляется с помощью *переменных* и *списков*. Эта глава посвящена переменным, а о списках речь пойдет в главе 9.

Глава начинается с обзора типов данных, которые поддерживает Scratch. Потом вводятся переменные и рассказывается о том, как создавать и использовать их в программах. Овладев базовыми знаниями, вы научитесь применять команду **спросить и ждать**, чтобы получать информацию от пользователя.

Разновидности данных в Scratch

Чтобы создавать полезную информацию, компьютерные программы манипулируют разными типами данных: цифрами, текстом, изображениями и т. п. Это важная задача программирования, и вам необходимо познакомиться с типами данных и операциями в Scratch. Эта среда имеет встроенную поддержку трех типов данных, которые вы можете использовать в блоках: логические величины, числа и строки.

Логические величины могут принимать только одно из двух значений: «верно» или «неверно». Этот тип данных можно применять для проверки одного или нескольких условий: тогда на основании полученного результата ваша программа выберет тот или иной сценарий действий. Подробнее логические величины рассматриваются в следующей главе.

Числовая переменная может принимать как целочисленное, так и дробное значение. Scratch не видит разницы между целыми числами и дробными. Можно округлять десятичные дроби до ближайшего целого числа с помощью блока **округлить** из раздела **Операторы**. Также можно использовать функции **пол от** (или **предел от**), которые находятся в блоке **квадратный корень от** раздела **Операторы**, если нужно получить целое число из некоторой десятичной дроби. Например, **пол от 3.9** будет равен 3, а **предел от 3.1** — 4.

Строка — последовательность символов, которые могут быть буквами (как нижнего, так и верхнего регистра), цифрами (от 0 до 9) и другими знаками, которые набираются с помощью клавиатуры (+, -, &, @ и т. д.). Данные этого типа используются для сохранения названий, адресов, заглавий книг и т. п.

Что означает форма?

Обратили ли вы внимание на то, что блоки Scratch и окошки для ввода параметров имеют определенную геометрическую форму? Например, окошко для ввода в блоке **идти 10 шагов** — прямоугольник с закругленными уголками, а в блоке **сказать Привет!** это обычный прямоугольник. Форма окошка отражает тип данных, которые можно в него ввести. Попробуйте вбить ваше имя (или другой текст) в блок **идти 10 шагов**: вы увидите, что там можно ввести только цифры.

Форма блока-функции указывает на то, какой тип данных он выдает. Соответствие формы и типа данных показано на рис. 5.1.

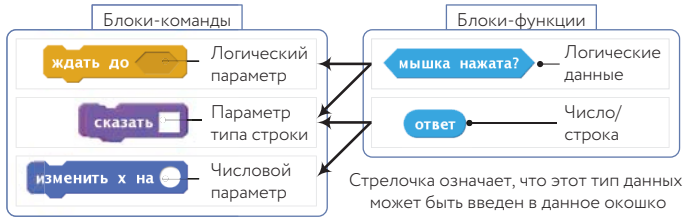


Рис. 5.1. Что означает форма различных блоков-команд и блоков-функций

Окошки параметров могут иметь форму трех геометрических фигур (шестиугольника, прямоугольника и прямоугольника со скругленными углами), а блоки-функции — только двух (шестиугольник и прямоугольник со скругленными углами). Каждой геометрической фигуре соответствует свой тип данных. А блок-функция в форме прямоугольника со скругленными углами может выдавать как число, так и строку.

Шестиугольные окошки и окошки в виде прямоугольника со скругленными углами соединяются только с блоками-функциями той же формы, а прямоугольное окошко может связываться с любым блоком-функцией. Scratch сам не даст вам перепутать типы, так что не нужно держать это правило в голове. Попробуйте перетащить шестиугольный блок в окошко, имеющее форму прямоугольника со скругленными углами: у вас ничего не получится, потому что эти типы друг с другом несовместимы.

Автоматическая конвертация типов данных

Как я упоминал выше, окошко для числовых параметров может соединяться только с блоком-функцией в форме прямоугольника со скругленными углами. Все блоки-функции — прямоугольники со скругленными углами, с которыми вы до сих пор работали, включая **положение x**, **положение y**, **направление**, **костюм**, **размер**, **громкость**, **темп** и пр., — выдают числа. Нет никаких сложностей с использованием чисел в окошках, для них и предназначенных (например, в блоке **идти 10 шагов**). Но некоторые блоки-функции в виде прямоугольников со скругленными углами (например, **ответ** из раздела **Сенсоры** или **слить** из раздела **Операторы**) могут работать как с числами, так и со строками. Возникает вопрос: что произойдет, если мы, например, вставим блок **ответ**, содержащий строку, в окошко, предназначенное для чисел? К счастью, при необходимости Scratch автоматически пытается конвертировать один тип данных в другой, как показано на рис. 5.2.

В данном случае пользователь вводит 125 в ответ на подсказку **Введите число**. Информация сохраняется в блоке-функции **ответ**. Когда она передается команде **сказать**, то автоматически конвертируется в строку. Когда этот же ответ передается для выполнения операции сложения (которая ожидает получить число), то происходит обратная конвертация в число 125. После выполнения операции сложения результат (25 + 125 = 150) конвертируется обратно в строку, а число 150 передается блоку **сказать**. Так Scratch автоматически пытается выполнить эту конверсию.

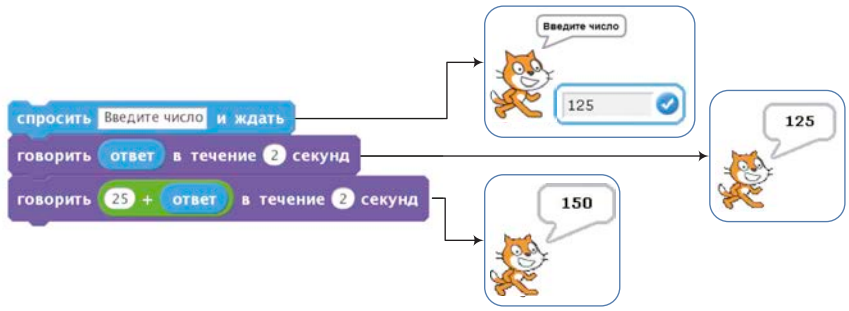


Рис. 5.2. Scratch на основе контекста автоматически конвертирует один тип данных в другой

Если вы знаете, с какими типами данных работает Scratch и какие операции выполняет, а также как он конвертирует один тип данных в другой, вам проще понять, почему среда работает именно таким образом. В следующем разделе я расскажу о переменных и о том, как использовать их в ваших программах, чтобы хранить данные и манипулировать ими.

Переменные

Предположим, нам захотелось создать компьютерную версию игры «Стукни крота» (Whac-a-Mole). Игровое поле — плоская поверхность с несколькими норами, из которых высовываются кроты. Пользователь должен стукнуть их молотком. В нашей версии спрайт появляется в произвольных точках **Сцены**, на короткое время становится видимым, а потом исчезает. Немного погодя он появляется в другом месте. Пользователю надо кликнуть по спрайту, как только тот появится. Если пользователю это удалось, он получает очко. Вопрос к вам как к программисту: как вести счет очкам, которые заработал игрок? Добро пожаловать в царство переменных!

В этом разделе я расскажу о переменных — одном из самых важных элементов любого языка программирования. Вы узнаете, как создавать

переменные в Scratch и использовать их для запоминания (или хранения) различных типов данных. Вы также научитесь использовать доступные блоки, чтобы задавать и менять значения переменных в своих программах.

Что такое переменная?

Переменная — поименованная область компьютерной памяти. Ее можно представить себе как ящик для хранения данных, числовых и текстовых, к которому при необходимости обращается программа. На рис. 5.3 показана переменная с именем сторона, чье текущее значение равно 50.

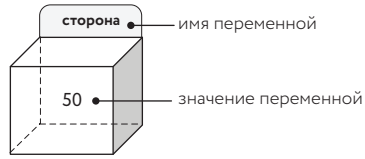


Рис. 5.3. Переменную можно сравнить с ящиком, которому присвоено имя и который содержит некоторое значение

Когда вы создаете переменную, ваша программа резервирует определенный объем памяти, чтобы сохранить ее значение, и присваивает выделенной памяти имя переменной. Создав переменную, вы можете использовать ее имя в своей программе, чтобы обращаться к ее значению. Если у нас, допустим, имеется ящик (переменная) по имени сторона, который содержит величину 50, мы можем сконструировать команду **идти (3 х сторона) шагов**. Выполняя эту команду, Scratch найдет в памяти компьютера ящик сторона, присвоит его содержимое (в данном случае 50) и использует это число вместо сторона в блоке **идти (3 х сторона) шагов**. В результате спрайт передвинется на 150 (50 умножить на 3) шагов.

Для игры «Стукни крота» мы должны найти способ запоминать количество очков, набранных пользователем. Чтобы сохранить его, надо зарезервировать место — ящик — в машинной памяти. Мы также должны присвоить этому ящику уникальный ярлык, например счет, с помощью которого мы всегда сможем найти его, если нам надо изменить его содержимое или узнать, что в нем хранится.

Когда начнется игра, мы скажем Scratch, чтобы он «установил счет на 0», и тогда Scratch, найдя ящик с ярлыком счет, поместит в него величину 0. Также мы велим Scratch «увеличить счет на 1» всякий раз, когда пользователь кликнет по спрайту. Реагируя на первый удачный клик, Scratch снова заглянет в ящик счет, найдет там 0 и прибавит к нему 1 — а потом поместит полученный результат (1) обратно в ящик. В следующий раз, когда пользователь кликнет по спрайту, Scratch снова выполнит

нашу команду, увеличит на 1 содержимое ящика и сохранит в нем полученную величину 2.

Вы скоро увидите, какие блоки использует Scratch для выполнения этих операций. Но пока обратите внимание на то, что величина счет меняется по мере работы программы. Вот почему мы называем переменную переменной: ее величина все время изменяется.

Переменные также полезны, когда нужно сохранить промежуточные результаты вычисления алгебраических выражений. Это похоже на счет в уме. Например, вас попросили посчитать, сколько будет 2 + 4 + 5 + 7: вы начнете с того, что сложите 2 и 4 и запомните результат (6). Затем прибавите к этому результату 5 и запомните новое число — 11. И наконец, прибавите к этому результату 8 и получите окончательный ответ — 18.

Чтобы проиллюстрировать, как можно использовать переменные для временного хранения, вообразим, что вы хотите написать программу, которая вычислила бы следующее алгебраическое выражение:

$$\frac{(1 / 5) + (5 / 7)}{(7 / 8) - (2 / 3)}$$

Вы можете провести расчет в одну команду, но, как видно ниже, если запихать всё в одну строку, то все выражение становится неудобочитаемым.

Другой способ написать программу — рассчитать числитель и знаменатель по отдельности, а затем использовать блок **сказать**, чтобы показать результаты деления одного на другой. Мы можем сделать это, создав две переменные числ (для числителя) и знам (для знаменателя) и задав их величины так, как показано на рис. 5.4.

*Рис. 5.4. Две переменные (*числ *и* знам*) — величины соответственно числителя и знаменателя данного алгебраического выражения*

Посмотрим, как наши переменные организованы в машинной памяти. В данном случае числ подобно ярлыку, который отсылает к ячейке памяти, где хранится результат расчетов (1/5 + 5/7). А знам — отсылка к месту

хранения результата вычисления (7/8 – 2/3). Когда выполняется команда **сказать**, Scratch забирает содержимое памяти, помеченное числ и знам, делит одно на другое и передает результат команде **сказать** для показа.

Мы можем разбить это выражение на еще более мелкие составляющие, вычисляя каждую дробь, прежде чем показать результат, как видно на рис. 5.5.

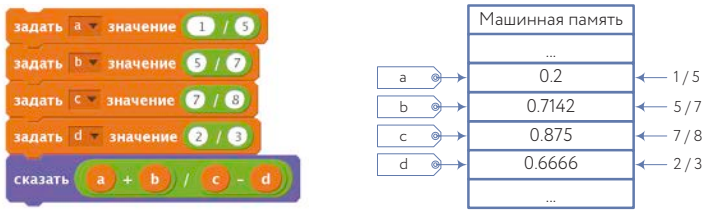


Рис. 5.5. Использование четырех переменных (a, b, c, d) *для вычисления четырех дробей выражения*

Мы используем четыре переменные (a, b, c, d) для вычисления четырех дробей в нашем математическом выражении. Рисунок также отображает выделение памяти; вы можете увидеть и переменные, и их содержимое.

Три эти программы дают один ответ, но исполнены они по-разному. В первой все «загнано» в одну строку, которую трудно прочесть. В третьей пример разбит на небольшие части, но его тоже нелегко прочесть. А во втором варианте выражение разбито разумно, а переменные использованы как для того, чтобы программу было легче понять, так и для того, чтобы выделить основные части выражения (числитель и знаменатель).

Этот простой пример показывает, что у одного примера может быть несколько решений. Иногда важнее скорость работы программы или ее размер, а иногда — легкость прочтения. Поскольку это пособие рассчитано на начальный уровень, скрипты в книге написаны с упором на легкость прочтения.

Теперь, когда мы разобрались с тем, что такое переменные и зачем они нужны, создадим несколько переменных и сделаем следующий шаг в работе с приложениями Scratch.

Создание и использование переменных

DiceSimulator_NoCode.sb2

Из этого раздела вы узнаете, как создавать и использовать переменные в простом приложении, которое имитирует бросок пары игральных кубиков и показывает сумму результатов, как показано на рис. 5.6.

Наш симулятор содержит три спрайта: Игрок, Кубик1 и Кубик2 (Player, Die1 и Die2). Управляет процессом первый из них. При нажатии иконки с зеленым флажком этот спрайт генерирует два случайных числа в диапазоне от 1 до 6 и сохраняет эти значения в двух переменных, которые названы слч1 и слч2. Затем эта информация передается

двум другим спрайтам (Кубик1 и Кубик2), чтобы они отобразили эти случайные величины. Кубик1 будет отражать величину слч1, а Кубик2 — величину слч2. После этого спрайт Игрок сложит слч1 и слч2 и отобразит полученную сумму, используя блок **сказать**.

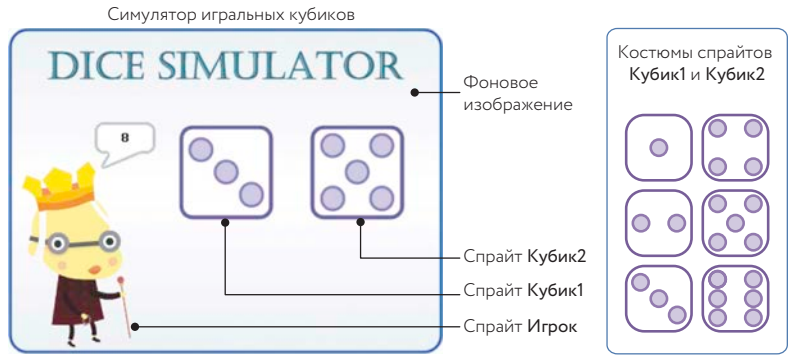


Рис. 5.6. Интерфейс пользователя для симулятора игральных кубиков

Создадим такое приложение. Откройте файл *DiceSimulator_NoCode.sb2*. Он содержит фоновое изображение для **Сцены**, а также три спрайта, используемых в приложении. Теперь по очереди создадим все необходимые скрипты.

Сначала кликните по иконке спрайта Игрок, чтобы выбрать его. Зайдите в раздел **Данные** и нажмите **Создать переменную**, как показано на рис. 5.7 (слева). В появившемся диалоговом окне напечатайте (как показано справа на рис. 5.7) имя переменной и выберите область определения для нее. *Область определения* переменной показывает, какие спрайты могут переписывать ее (или менять ее значение). Более подробно я расскажу об этом в следующем разделе. В нашем примере введите имя переменной слч1 и выберите **Для всех спрайтов** в качестве области определения. Когда закончите, нажмите **OK**.

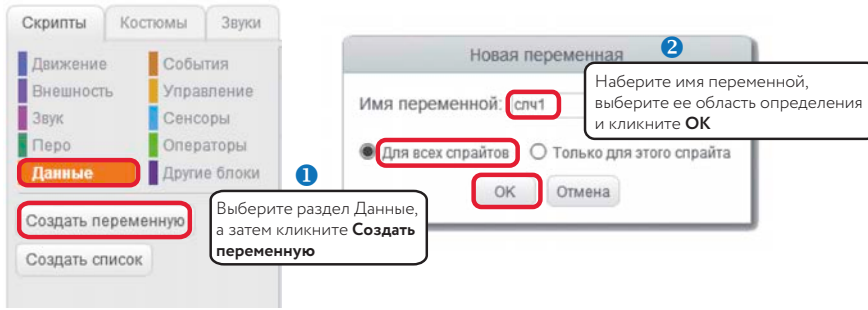


Рис. 5.7. Создание переменной, ее имени и указание ее области определения

После того как вы создали переменную, в разделе **Данные** возникнет несколько новых блоков, как показано на рис. 5.8.

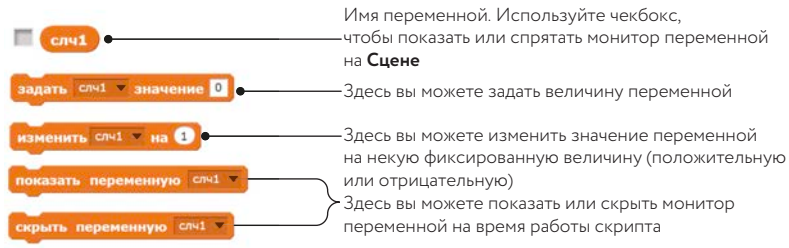


Рис. 5.8. Новые блоки, которые появляются после того, как вы создали переменную слч1

Вы можете использовать эти блоки, чтобы задать значение переменной, изменить ее на некоторую величину либо показать (или спрятать) ее монитор на **Сцене**. *Монитор*, как вы узнаете в разделе «Отображение мониторов переменной» на с. 131, отражает текущее значение переменной.

ИМЕНОВАНИЕ ПЕРЕМЕННЫХ

За много лет люди придумали множество способов именовать переменные. Один такой: начать имя с маленькой буквы, а каждое следующее слово писать с большой, например длинаСтороны, процентнаяСтавка.

Scratch позволяет давать переменным имена, которые начинаются с цифр и содержат пробелы (например, 123Сторона и длина стороны), но большинство языков программирования такого делать не разрешают. Так что я не рекомендую давать такие экзотические имена своим переменным. Лучше использовать описательные и осмысленные имена. Переменные с именем, состоящим из одной буквы, например w или z, должны быть в меньшинстве, разве что их значение очевидно. Но если имена слишком длинные, то скрипт может быть трудным для прочтения.

Имена переменных в Scratch чувствительны к регистру. Имена типа сторона, СТОРОНА, сторОНА будут не вариантами для одной переменной, а именами разных переменных. Чтобы избежать путаницы, лучше не использовать в одном скрипте имена переменных, различающиеся только регистром.

Повторите описанную выше процедуру и создайте еще одну переменную с именем слч2. Теперь в разделе **Данные** должен появиться второй блок переменной (с именем слч2), и направленные вниз стрелочки на блоках на рис. 5.8 помогут вам выбрать между слч1 и слч2. И теперь, когда мы создали две переменные, мы можем создать и скрипт для спрайта Игрок. Целиком он показан на рис. 5.9.

Рис. 5.9. Скрипт для спрайта Игрок

Первая команда присваивает переменной слч1 случайное значение в диапазоне от 1 до 6. Что это означает, если вернуться к сравнению с ящиком? Эта команда заставляет спрайт найти ящик с ярлыком слч1 и поместить в него случайным образом сгенерированное число. Следующая команда делает то же самое для переменной слч2. Далее скрипт передает сообщение **Катиться** двум другим спрайтам (Кубик1 и Кубик2), чтобы уведомить их о том, что они должны сменить свои костюмы соответственно слч1 и слч2. Как только спрайты Кубик1 и Кубик2 сделали свое дело, скрипт продолжает работу и отображает сумму чисел, выпавших на кубиках, с помощью блока **говорить**. Посмотрим на рис. 5.10, чтобы понять, как работает сообщение **Катиться** для спрайта Кубик1.

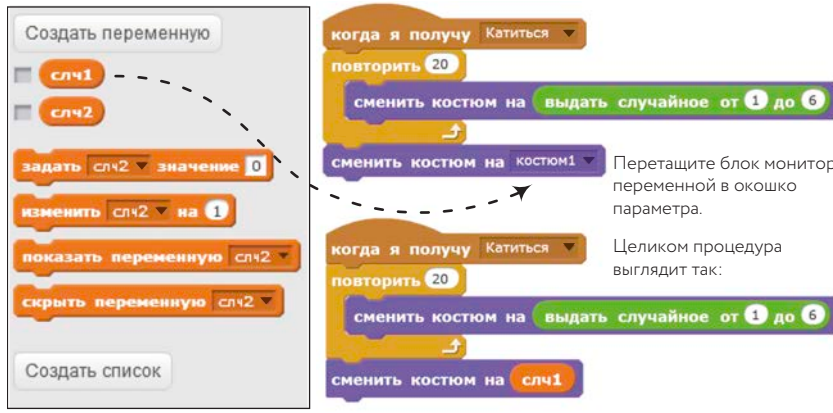


Рис. 5.10. Чтобы использовать переменную в блоке-команде, перетащите ее в окошко параметра нужного блока

Создав скрипт, который вы видите справа вверху, перетащите блок слч1 из раздела **Данные** в окошко параметра блока **сменить костюм**, чтобы получился законченный скрипт (справа внизу). В этом скрипте блок **повторить** 20 раз подряд случайным образом меняет костюм кубика, чтобы было похоже на то, что катится настоящий кубик (вы можете

выбрать любое другое число). После этого кубик «надевает» костюм числа, определяемого переменной слч1. Вы не забыли, что у каждого кубика шесть костюмов, соответствующих цифрам с 1 по 6? Это означает, что если слч1 равна 5, последняя команда **сменить костюм** отобразит костюм с пятью точками.

Теперь мы можем создать скрипт для спрайта Кубик2, который будет практически таким же, как для Кубика1. Поскольку Кубик2 меняет костюм согласно переменной слч2, остается дублировать скрипт спрайта Кубик1 для спрайта Кубик2 и заменить в нем слч1 на слч2.

Теперь наш симулятор игральных кубиков готов, протестируем его. Кликните по иконке с зеленым флажком, чтобы посмотреть на результат. Если приложение не работает, проверьте файл *DiceSimulator_NoCode.sb2*, который содержит корректную реализацию программы.

УПРАЖНЕНИЕ 5.1

Выберите спрайт Игрок и создайте новую переменную с именем сумма. Задайте область определения **Только для этого спрайта**. Модифицируйте последний блок скрипта Игрок, чтобы использовать эту новую переменную, вот так:

Теперь выберите спрайт Кубик1 (или Кубик2) и загляните в раздел **Данные**. Сможете объяснить, почему вы не видите там переменной сумма?

Область определения переменной

ScopeDemo.sb2

Еще один важный момент, связанный с переменной, — ее *область определения*. Она задает, какой спрайт может переписывать (или изменять) значение данной переменной.

Вы можете указать область определения, когда создаете переменную, выбрав один из двух вариантов, показанных на рис. 5.7. Если вы выберете **Только для этого спрайта**, то создадите переменную, которую будет менять только спрайт, которому она принадлежит. Другие смогут читать ее и использовать ее значение, но менять его не смогут. Пример, приведенный на рис. 5.11, иллюстрирует эту тонкость.

На этом рисунке вы видите переменную подсчет спрайта Кот. Ее область определения — **Только для этого спрайта**. Спрайт Пингвин (Penguin) может считывать переменную подсчет с помощью блока **x от Пингвин** из раздела **Сенсоры**. Когда вы выбираете Кота в качестве второго параметра этого блока, первый параметр позволяет выбрать любой атрибут спрайта, включая одну из его переменных.

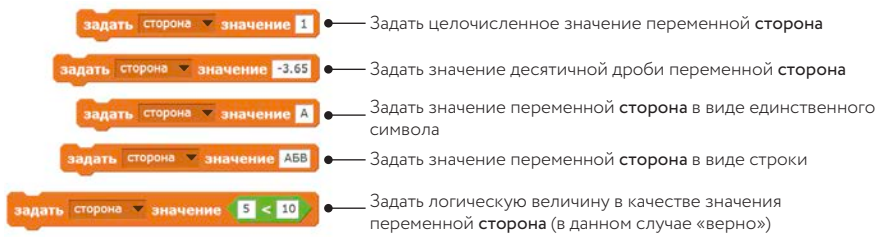
Рис. 5.11. Только спрайт Кот *может менять значение переменной* подсчет

Конечно, в Scratch нет блока, который позволит спрайту Пингвин менять переменную подсчет. Получается, Пингвин ничего не может с ней сделать и испортить скрипты спрайта Кот. Так что стоит использовать область определения **Только для этого спрайта**, если вы хотите, чтобы переменную обновлял один-единственный спрайт.

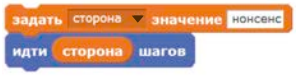
Переменные с областью определения **Только для этого спрайта** имеют *локальную область действия*, потому их можно назвать *локальными*. Разные спрайты могут использовать свои локальные переменные, даже если у тех идентичные имена, и никакого конфликта не возникнет. Например, если у вас есть два спрайта-машины для игры в гонки, каждый из них может иметь локальную переменную по имени скорость, которая определяет скорость движения машины на **Сцене**. Оба спрайта могут менять свою переменную скорость независимо друг от друга. Это значит, что если вы задаете значение скорость равным 10 для первой машины и равным 20 для второй, вторая машина будет двигаться быстрее первой.

ТИП ДАННЫХ ПЕРЕМЕННОЙ

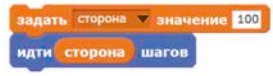
Здесь, наверное, вы удивитесь: «Но откуда Scratch знает, какой тип данных у этой переменной?» Ответ будет прост: «А он и не знает!» Когда вы создаете переменную, Scratch понятия не имеет, для чего она вам нужна: чтобы сохранять численное значение, строку или логическую величину. Любая переменная может иметь дело с любым типом данных. Например, все нижеследующие команды в Scratch допустимы.



Вам решать, какие значения сохранять в ваших переменных. Как я писал выше, Scratch попытается конвертировать одни типы данных в другие в зависимости от контекста. Чтобы посмотреть, что происходит, когда вы сохраняете неподходящий тип данных в своей переменной, проанализируем два примера.



Строка «нонсенс» конвертирована в число 0 и передана команде **идти**



Строка «100» конвертирована в число 100 и передана команде **идти**

Поскольку команда **идти** ожидает числовой параметр, Scratch автоматически попытается конвертировать величину, сохраненную в переменной сторона, в число, прежде чем передать информацию команде **идти**. В первом скрипте (слева) Scratch не может конвертировать строку «нонсенс» в число. Но вместо того чтобы выводить сообщение об ошибке, Scratch, не говоря ни слова, выставит ноль в качестве результата конвертации и передаст эту величину команде **идти**. В результате спрайт останется стоять на месте. А в случае второго скрипта (справа) Scratch не будет обращать внимания на пробелы в строке и передаст полученное число блоку **идти**, так что спрайт сделает 100 шагов вперед. Если бы результирующий блок ожидал получить строку, а не число, Scratch передал бы ее как есть, со всеми пробелами.

Переменные с областью определения **Для всех спрайтов** могут считывать и изменять любой спрайт в вашем приложении. Эти переменные, часто называемые *глобальными*, полезны для коммуникации между спрайтами и их синхронизации. Например, если в игре три кнопки, которые позволяют пользователю выбирать уровень, вы можете создать глобальную переменную с именем играУровень и позволять спрайту каждой кнопки устанавливать собственное значение этой переменной, если по ней кликнули. Тогда вам нетрудно будет узнать, какой уровень выбрал пользователь: вы просто посмотрите на переменную играУровень.

Вариант **Для всех спрайтов** также даст вам возможность воспользоваться чекбоксом **Облачная переменная** (см. рис. 5.7). Это позволит вам сохранять переменные на сервере Scratch (в облаке). Блоки для облачных переменных имеют небольшой квадратик спереди, который отличает их от обычных переменных. Выглядит это так.

Всякий, кто будет просматривать проект, который вы выложили для всеобщего обозрения на сайте Scratch, сможет прочесть облачные переменные проекта. Например, если вы выложили для совместного

использования игру, то можете использовать облачную переменную, чтобы зафиксировать самое большое количество очков, которое набрали пользователи. Облачная переменная счет должна обновляться практически мгновенно для всякого, кто начал играть в вашу игру. Поскольку эти переменные хранятся на серверах Scratch, они сохраняют свое значение, даже если вы выходите из браузера. Облачные переменные позволяют легко создавать обзоры и другие проекты, которые накапливают числовые данные.

Теперь, когда мы разобрались с областью определения переменной, пришла пора выяснить, как обновлять их, — и применить это знание для создания интересных программ.

Изменение переменных

Scratch имеет два блока-команды, которые позволяют изменять переменные. Команда **задать значение** прямо присваивает новое значение, независимо от текущего. Команда **изменить на** используется для того, чтобы изменить текущее значение переменной на определенную величину. Три скрипта на рис. 5.12 показывают, как применять эти команды, чтобы разными способами достичь одной цели.

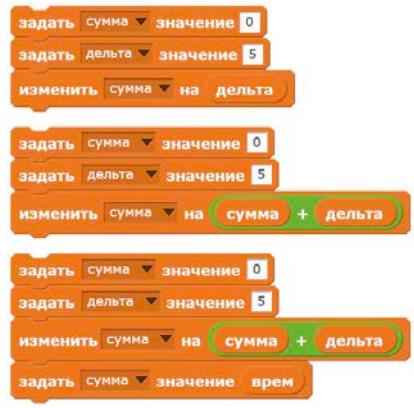


Рис. 5.12. Три способа изменить значение переменной

Все три скрипта на рисунке начинаются с того, что мы задаем значения двух переменных, сумма и дельта, равными 0 и 5 соответственно. Первый скрипт использует команду **изменить**, чтобы изменить величину сумма на величину дельта (с 0 на 5). Второй использует команду **задать**, чтобы прибавить текущее значение переменной к значению дельта (0 + 5) и сохранить полученный результат (5) в переменной сумма. Третий добивается того же результата с помощью переменной врем.

Он складывает значения переменных сумма и дельта, сохраняет результат в врем и копирует величину врем в сумма.

После исполнения любого из скриптов, показанных на рис. 5.12, величина переменной сумма будет равна 5. Все эти скрипты функционально эквивалентны друг другу. Метод, использованный во втором скрипте, — общепринятый в программировании, и я рекомендую вам получше к нему приглядеться. Теперь посмотрим, как работает команда **изменить**.

Паутина

Мы можем создать паутину, рисуя шестиугольники, которые постепенно увеличиваются в размере, как показано на рис. 5.13. Процедура **Треугольник** рисует равносторонний треугольник с переменной длиной стороны, а **Шестиугольник** вызывает **Треугольник** шесть раз, шесть раз поворачивая его на 60 градусов (360 градусов, деленные на 6) направо при каждом обращении. На рисунке показано, как шестиугольник собирается из шести треугольников.

SpiderWeb.sb2

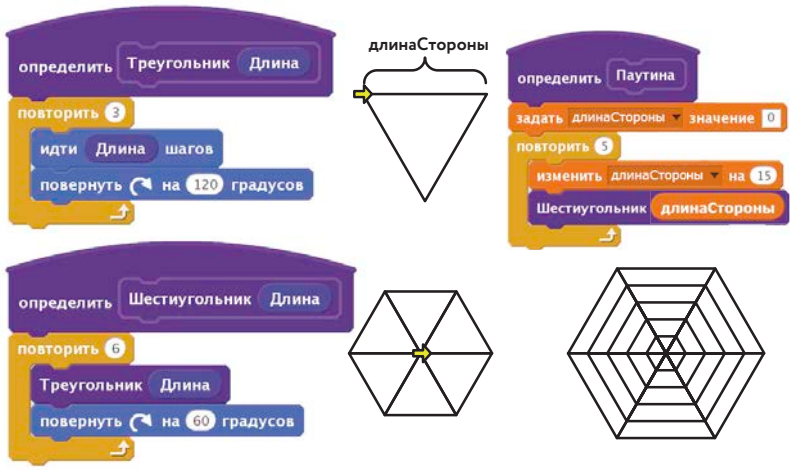


Рис. 5.13. Создание паутины путем рисования шестиугольников увеличивающегося размера

Процедура **Паутина** несколько раз обращается к **Шестиугольнику**, каждый раз меняя величину переменной длинаСтороны. В результате получаются концентрические (имеющие общий центр) шестиугольники. Обратите внимание, как команда **изменить** использована, чтобы задать величину длинаСтороны внутри цикла **повторить**. Воспроизведите алгоритм **Паутина**, запустите его и посмотрите, как он работает.

Вертушка

Pinwheel.sb2

Этот пример мало чем отличается от предыдущего, но в этот раз мы будем использовать переменную, чтобы управлять количеством повторений треугольника. Получившийся в результате алгоритм **Спицы** (**Pins**) показан на рис. 5.14. Процедура **Вертушка** (**Pinwheel**) на том же рисунке работает так же, как **Паутина**, но мы дополнительно меняем цвет пера при каждом повторении, чтобы получить интересный радужный эффект. Результаты работы процедуры **Вертушка** для разных величин переменной подсчет показаны в нижней части рисунка.

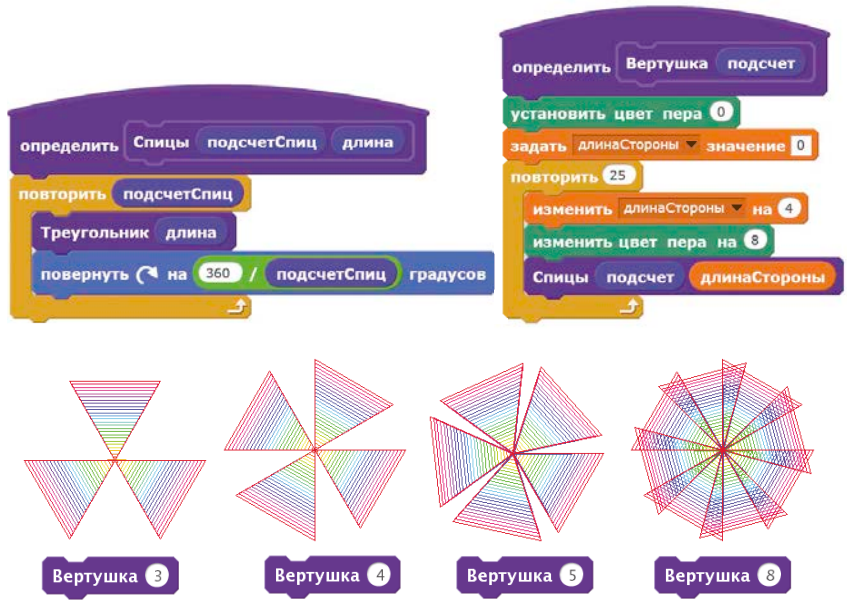


Рис. 5.14. Рисование вертушки путем вращения равностороннего треугольника

Теперь, когда мы разобрались с азами переменных, вы, возможно, задумались, что происходит с ними при дублировании спрайта. Использует ли дубликат переменные исходного спрайта или их копии? Имеют ли клоны доступ к глобальным переменным? В следующем разделе вы узнаете ответ.

УПРАЖНЕНИЕ 5.2

Поработав с процедурой **Вертушка**, спрячьте спрайт. Тогда вам будет проще следить, как возникает рисунок: спрайт не будет вам мешать.

Переменные клонов

Каждый спрайт имеет список свойств, включая его положение *x*, положение *y*, направление и т. д. Можно представить этот список как рюкзак, в котором хранятся текущие величины атрибутов спрайта, как показано на рис. 5.15. Когда вы создаете переменную, имеющую область определения **Только для этого спрайта**, она добавляется в рюкзак спрайта.

Когда вы клонируете спрайт, клон наследует копии атрибутов родителя, в том числе переменные. Унаследованное свойство вначале идентично свойству спрайта-родителя — каким оно было на момент создания клона. Но если после этого атрибуты и переменные клона меняются, эти перемены не оказывают никакого воздействия на спрайт-родителя. А изменения спрайта-родителя никак не сказываются на свойствах клонов.

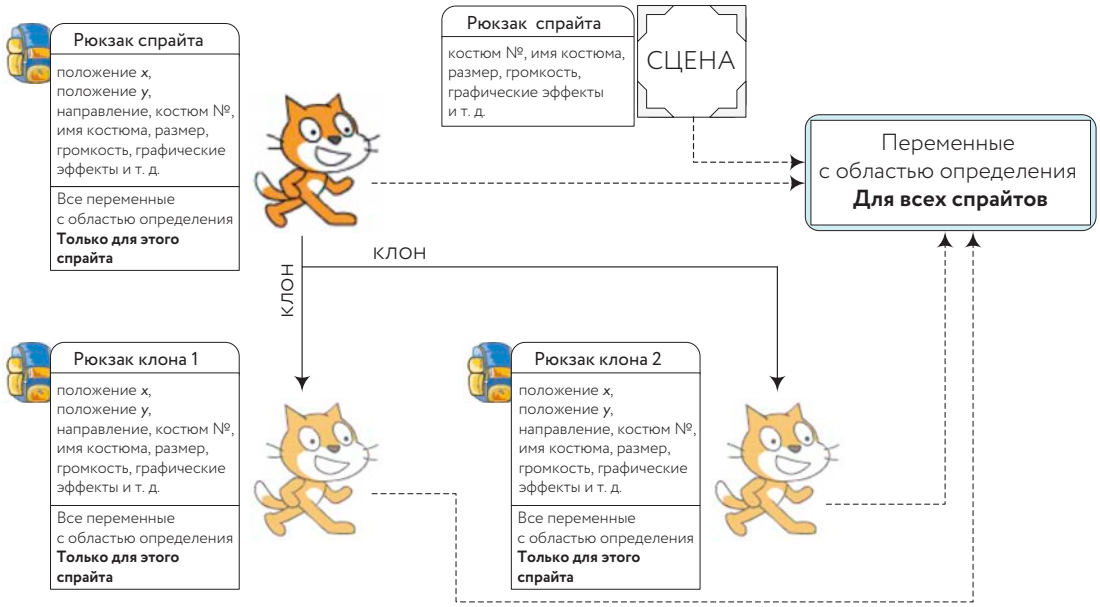


Рис. 5.15. Клоны наследуют копии переменных спрайта-родителя

Предположим, у спрайта-родителя есть переменная скорость, чья текущая величина равна 10. Когда вы клонируете этот спрайт, новый также будет иметь переменную скорость, равную 10. Если после этого скорость спрайта-родителя изменится и будет равна 20, величина переменной скорость у клона по-прежнему будет равна 10.

Вы можете использовать это свойство для того, чтобы отличать один клон от другого в вашем приложении. Изучим программу на рис. 5.16. CloneIDs.sb2

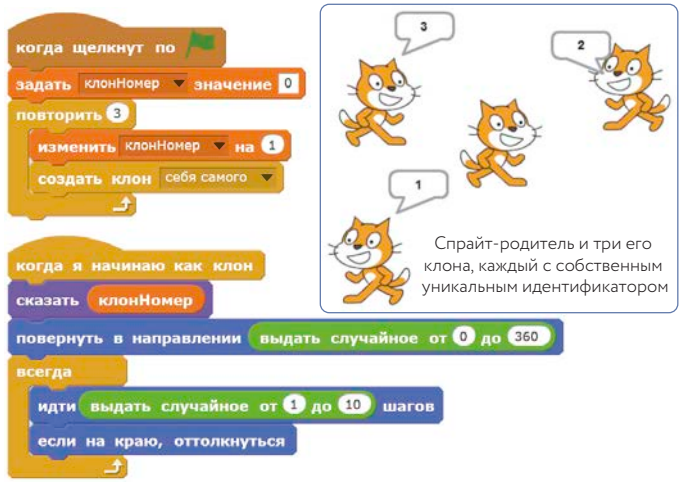


Рис. 5.16. Использование переменных для различения клонов

Спрайт-родитель в данном примере имеет переменную с именем клонНомер. Если кликнуть по зеленому флажку, в результате повтора возникнет три клона, при создании каждого из которых задается разное значение переменной клонНомер (в данном случае — 1, 2, 3). У каждого клона при его появлении на свет имеется своя копия переменной клонНомер со своим уникальным значением. Теперь вы можете использовать блок **если**, который мы подробно изучим в следующей главе, чтобы проверить идентификатор клона и заставить каждый клон выполнить определенные действия.

ClonesAnd GlobalVars.sb2

Теперь посмотрим, как клоны взаимодействуют с глобальными переменными. Вспомним рис. 5.15: переменные с областью определения **Для всех спрайтов** могут читаться и переписываться **Сценой** и всеми спрайтами, не исключая клонов. Например, программа, изображенная на рис. 5.17, использует эту особенность, чтобы проверить, все ли клоны спрайта-родителя исчезли.

Рис. 5.17. Использование глобальной переменной для уничтожения клонов

В данном скрипте спрайт-родитель задает величину глобальной переменной числоКлонов равной пяти и создает пять клонов. Затем ждет, пока числоКлонов не станет равна нулю, и объявляет, что игра закончилась. Клоны возникают через случайные промежутки времени и в случайно выбранных местах на **Сцене**, говорят «Привет!» в течение двух секунд, а затем исчезают. Когда все пять клонов исчезли, числоКлонов равна нулю, основной скрипт выходит из режима ожидания и спрайт-родитель объявляет: «Игра окончена!»

В следующем разделе вы узнаете о мониторах, которые позволяют вам видеть и даже менять текущие величины, сохраненные в переменных. Возможность просматривать и менять переменные прямо на **Сцене** позволяет создавать совершенно новые разновидности приложений.

Отображение мониторов переменных

Нередко возникает желание увидеть текущую величину переменной. Например, когда какой-то из ваших скриптов работает не так, как надо, вам хочется понаблюдать за значениями соответствующих переменных, чтобы понять, насколько корректно они меняются. Использование мониторов переменных поможет вам при отладке.

Переменную в Scratch можно увидеть на сцене с помощью монитора переменной. В зависимости от того, поставите вы или уберете галочку в чекбоксе рядом с именем переменной, вы сможете спрятать или показать монитор на **Сцене**, как показано на рис. 5.18. Вы также сможете контролировать видимость монитора изнутри вашего скрипта с помощью команд **показать переменную** и **скрыть переменную**.

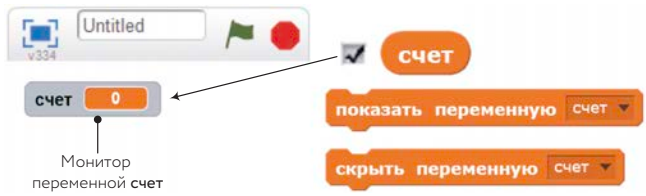


Рис. 5.18. Чтобы сделать видимым монитор переменной, поставьте галочку в чекбоксе рядом с ее именем

Монитор допустимо использовать в качестве индикаторной панели или пульта управления: он может показывать вам содержимое переменной, а может позволить вам его изменить. Дважды кликните по прямоугольнику монитора на **Сцене**, чтобы выбрать стандартный вид (по умолчанию), увеличенный или рычажок. Если вы хотите видеть рычажок, вы можете задать его диапазон, кликнув по нему правой кнопкой мыши и установив минимум и максимум в выпадающем окне, как показано на рис. 5.19.

Рис. 5.19. Установка максимального и минимального значений для монитора в режиме рычажка

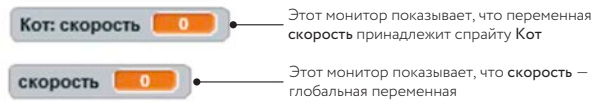
StageColor.sb2

Использование рычажка позволит менять величину переменной в процессе работы скрипта — удобный для пользователей способ взаимодействовать с приложением. Простой пример использования рычажка приведен на рис. 5.20.

Рис. 5.20. Изменение переменной цветСцены *с помощью рычажка*

В данном примере манипулирование рычажками меняет величину переменной цветСцены, которая является параметром команды **изменить цвет эффект на**. Поскольку этот скрипт принадлежит **Сцене**, передвижение рычажка должно приводить к изменению ее цвета.

Монитор переменной также указывает ее область определения. Если переменная принадлежит одному спрайту, на ее мониторе имя спрайта будет предшествовать имени переменной. Например, монитор **Кот: скорость 0** *показывает, что переменная* скорость *принадлежит спрайту* Кот. *Если бы переменная была глобальной, то на ее мониторе значилось бы только* **скорость 0**, *как показано на рисунке ниже.*



Использование мониторов переменных в приложениях

Теперь я покажу, как можно использовать мониторы, чтобы увеличить функциональность ваших приложений в среде Scratch.

Мониторы могут выполнять функции индикаторной панели и пульта управления. Это дает возможность работать с разными приложениями, включая игры, имитационное моделирование и интерактивные программы. В нижеследующих подразделах приводятся примеры использования мониторов.

Имитационное моделирование закона Ома

OhmsLaw.sb2

Первый пример — имитационное моделирование закона Ома. Когда электрическое напряжение (*V*) приложено к резистору, обеспечивающему сопротивление (*R*), через резистор проходит электрический ток (*I*). Согласно закону Ома, сила тока определяется следующим уравнением:

$$\textit{Сила тока } (I) = \frac{\textit{Напряжение } (V)}{\textit{Сопротивление } (R)}$$

Наше приложение позволит пользователю менять величины *V* и *R* с помощью рычажка-бегунка. Затем программа рассчитает и отобразит величину силы тока, *I*. Интерфейс пользователя для этого приложения показан на рис. 5.21.

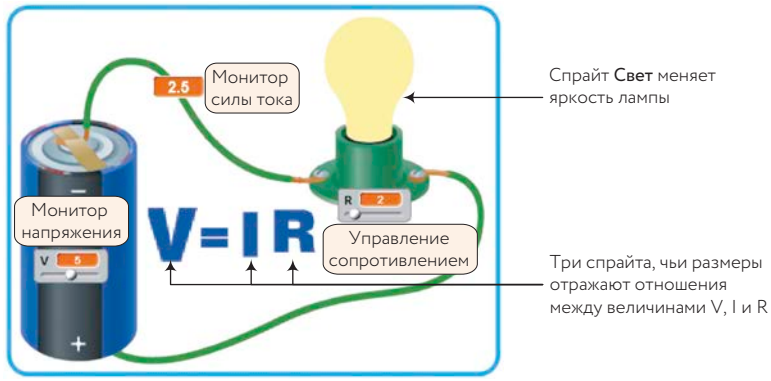


Рис. 5.21. Интерфейс пользователя для приложения, показывающего, как работает закон Ома

Бегунок, регулирующий напряжение батарейки (*V*), имеет диапазон [0, 10], а бегунок резистора (*R*) — [1, 10]. Когда пользователь меняет *V* или *R* с помощью бегунка, приложение рассчитывает силу тока (*I*)

в сети. Яркость лампы меняется сообразно силе тока, проходящего через нее: чем она выше, тем ярче горит лампа. Размер букв *V*, *I* и *R* на рисунке также меняется, отражая соотношение этих величин.

Всего в приложении пять спрайтов — Вольт, Ток, Сопротивление, Равно и Свет (Volt, Current, Resistance, Equal и Light) — и три переменные (V, I и R). Все прочее, что вы видите на рис. 5.21 (батарейка, провода, патрон лампы и т. п.), — часть фонового изображения **Сцены**. Основной скрипт, управляющий приложением и принадлежащий сцене, показан на рис. 5.22.

Рис. 5.22. Основной скрипт приложения, показывающего работу закона Ома

Скрипт запускает величины *V* и *R* и входит в бесконечный цикл. При каждом проходе цикла программа рассчитывает *I*, используя текущие значения *V* и *R*: их устанавливает пользователь с помощью бегунков. Затем сообщение передается другим спрайтам приложения, чтобы они обновили свой внешний вид по результатам перерасчета величин. Рис. 5.23 показывает реакцию спрайтов Вольт, Ток, Сопротивление и Свет (которые изображают буквы *V*, *I* и *R* и лампочку), когда они получают сообщение **Обновить**.

Рис. 5.23. Срабатывание скриптов при получении сообщения **Обновить**

Когда получено сообщение **Обновить**, спрайты Вольт, Ток и Сопротивление меняют размер (с изначальных 100% на 200%) соответственно

текущим величинам своих переменных. Спрайт Свет выполняет команду **изменить призрак эффект на**, чтобы изменить свой уровень прозрачности пропорционально величине *I*. Получается реалистичный визуальный эффект, имитация настоящей лампочки.

УПРАЖНЕНИЕ 5.3

Откройте и запустите имитационное моделирование закона Ома, изучите скрипты, чтобы понять, как работает программа. Как думаете, что произойдет, если вы добавите команду **изменить цвет эффект на 25** в конец скрипта для спрайта Свет? Вставьте эту команду, чтобы проверить свою догадку. Что еще можно сделать, чтобы усовершенствовать это приложение?

Демонстрация последовательного контура

SeriesCircuit.sb2

Наш второй пример — моделирование электрической цепи, в которой есть батарейка и три последовательно подключенных резистора. Пользователь может изменить напряжение батареи, а также сопротивление резистора, двигая бегунки. Ток, который течет через резисторы, и напряжение на каждом из них показаны с помощью больших мониторов. Интерфейс приложения можно увидеть на рис. 5.24. (Обратите внимание, что разноцветные полоски на резисторах не отражают текущее значение сопротивления.)

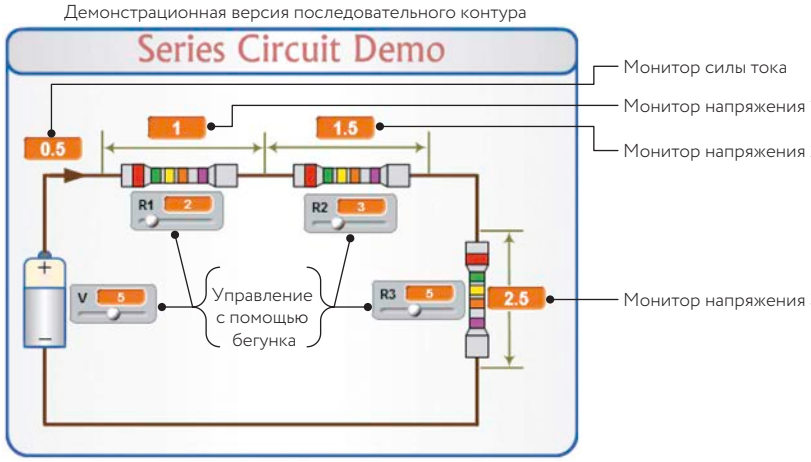


Рис. 5.24. Приложение, демонстрирующее работу последовательного контура

Уравнения, которые управляют работой контура, приведены ниже. Мы можем рассчитать ток в контуре, разделив напряжение батареи, *V*, на сумму сопротивлений трех резисторов. Теперь напряжение

на каждом резисторе рассчитывается путем умножения силы тока на сопротивление:

$$\text{Полное сопротивление: } R_{\text{общ}} = R_1 + R_2 + R_3$$
$$\text{Ток в цепи: } I + V = R_{\text{общ}}$$
$$\text{Напряжение в } R_1\text{: } V_1 = I \times R_1$$
$$\text{Напряжение в } R_2\text{: } V_2 = I \times R_2$$
$$\text{Напряжение в } R_3\text{: } V_3 = I \times R_3$$

У этого приложения нет спрайтов, но если кликнуть по зеленому флажку, начнет исполняться скрипт, показанный на рис. 5.35 и принадлежащий **Сцене**.

Этот скрипт решает за нас уравнения и отражает результаты на дисплеях мониторов на **Сцене**. Обратите внимание, что диапазон бегунка для резисторов R_2 и R_3 — от 0 до 10, а минимальная величина для R_1 — 1. Таким образом, $R_{\text{общ}}$ всегда будет больше 0, и программе не придется делить на ноль при расчете силы тока.

Рис. 5.25. Скрипт, который срабатывает после нажатия на кнопку с зеленым флажком

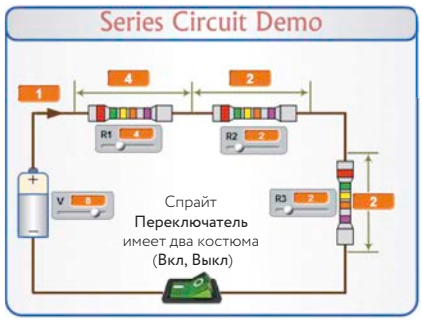
Бóльшая часть работы над этим приложением заключалась в разработке интерфейса (фона **Сцены**). После этого осталось только расположить мониторы и бегунки в нужных местах **Сцены**.

УПРАЖНЕНИЕ 5.4

SeriesCircuit WithSwitch.sb2

Откройте и запустите имитационное моделирование последовательного контура. Поиграйте со значениями R_1, R_2, R_3 и V. Посмотрите, как меняются величины V_1, V_2 и V_3, когда вы двигаете бегунки. Как связаны сумма напряжений (V_1 + V_2 + V_3) и напряжение батареи? Что это говорит вам о напряжении в последовательном контуре? Вы можете улучшить приложение, добавив изображение переключателя (switch), который открывает и закрывает цепь, как показано ниже. Когда переключатель открыт,

в цепи нет тока. Попытайтесь реализовать это дополнение, используя данные ниже подсказки.



Визуализация объема сферы и площади ее поверхности

Sphere.sb2

Третий пример — интерактивное приложение для расчета объема и площади поверхности сферы. Пользователь меняет диаметр сферы, кликая по кнопкам в интерфейсе пользователя, и приложение автоматически рассчитывает и отображает объем и площадь поверхности.

Чтобы сделать приложение более привлекательным, размер сферы, показанной на **Сцене**, также меняется пропорционально выбранному диаметру. Интерфейс пользователя для этого приложения приведен на рис. 5.26.

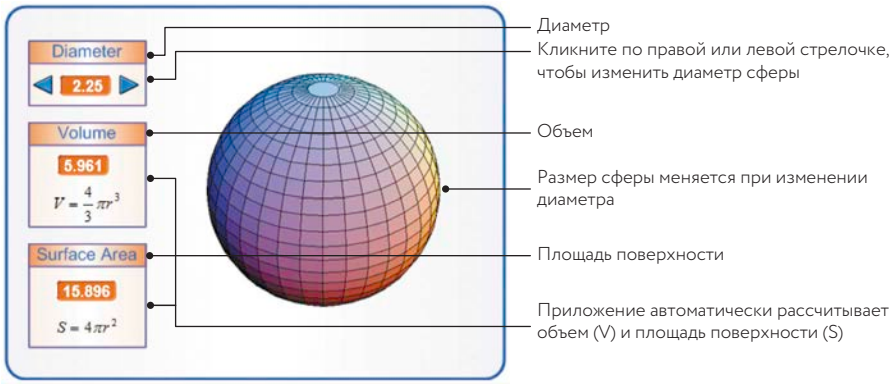


Рис. 5.26. Интерфейс пользователя в приложении, работающем с измерениями сферы

В этом приложении три спрайта: две кнопки-стрелочки Вверх и Вниз (Up и Down) и Сфера (Sphere). Скрипты, связанные с каждой кнопкой, передают сообщение о том, что по ним кликнули, как показано на рис. 5.27.

Рис. 5.27. Скрипты для спрайтов Вверх *и* Вниз

Спрайт Сфера имеет девять костюмов — сферы с диаметрами 1; 1,25; 1,5; 1,75; …; 3. Когда он получает сообщение от спрайтов Вверх и Вниз, он выполняет скрипт, показанный на рис. 5.28.

Рис. 5.28. Скрипты, запускаемые сообщениями Вверх *и* Вниз

Спрайт Сфера меняет костюм, а затем обращается к процедуре **Пересчитать**, чтобы обновить объем и площадь поверхности сферы. Эти скрипты используют значение текущего костюма, чтобы определить, достигла ли сфера минимального или максимального размера, обеспечивая ответ кнопкам Вверх и Вниз. В следующей главе я больше расскажу о блоке **если**, а пока рассмотрим процедуру **Пересчитать** для сферы, показанную на рис. 5.29.

Рис. 5.29. Процедура **Пересчитать**

Во-первых, значение переменной диаметр устанавливается по следующей формуле:

$$диаметр = 1 + 0{,}25 \times (номер\ костюма - 1).$$

Поскольку номер костюма может принимать значения в диапазоне от 1 до 9, соответствующие величины переменной диаметр будут 1; 1,25; 1,5; …; 2,75; 3, что нам и требовалось.

Скрипт высчитывает радиус *r*, разделив диаметр пополам. Затем он рассчитывает объем и площадь поверхности сферы, используя формулы, показанные на рис. 5.26. Рассчитанные величины будут автоматически отражены на соответствующих мониторах, расположенных на **Сцене**.

УПРАЖНЕНИЕ 5.5

Откройте приложение и запустите его. Изучите скрипты, чтобы понять, как работает приложение. Добавьте скрипт спрайту Сфера, чтобы он вращался и менял цвета во время работы приложения. Попробуйте сделать еще одно упражнение: измените изначальную программу так, чтобы у спрайта Сфера был только один костюм, а для изменения размера сферы используйте блок **изменить размер на**. Изображение увеличенного или уменьшенного размера будет выглядеть не так привлекательно, но в целом приложение должно работать так же, как раньше.

Рисуем розу с *N* лепестками

Создадим приложение, которое будет рисовать на **Сцене** розу со множеством лепестков. Процесс можно разбить на следующие шаги. N-LeavedRose.sb2

1. Начинаем в определенном месте **Сцены** — исходной точке.

2. Направляем спрайт в определенном направлении. Обычно греческая буква тета (θ) обозначает угол, так что мы назовем переменную для изменения направления спрайта Тета.

3. Двигаем спрайт на *r* шагов и ставим точку на **Сцене**, затем поднимаем перо и возвращаемся в исходную точку.

4. Меняем угол тета на некоторую величину (1 градус) и повторяем шаги 2–4.

Соотношение между расстоянием *r* и углом тета описывает следующее уравнение:

$$r = a \times \cos(n \times \ ),$$

где *a* — действительное число, а *n* — целое. Данное уравнение создает розу, чей размер и количество лепестков контролируются, соответственно, *a* и *n*. Это уравнение также включает в себя тригонометрическую функцию косинуса (**cos**), которую вы найдете в разделе **Операторы**,

в блоке функций (проверьте блок **квадратный корень**). Когда нам даны *a* и *n*, нам остается только выбрать различные величины тета, рассчитать соответствующие значения *r* и поставить точки на **Сцене**. Интерфейс пользователя для этого примера приведен на рис. 5.30.

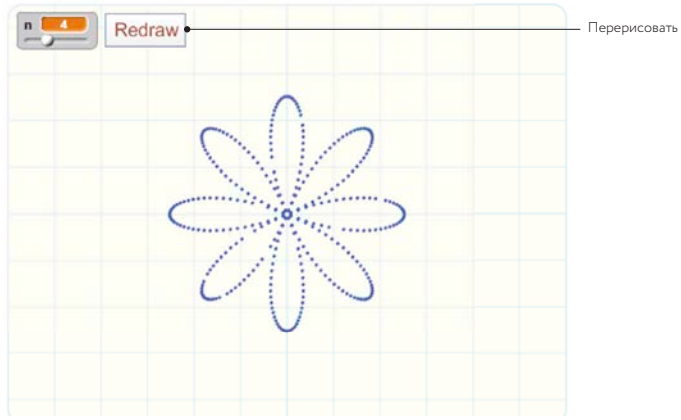


Рис. 5.30. Интерфейс пользователя в приложении для рисования розы с n лепестками

В этом приложении есть два спрайта: первому принадлежит костюм-кнопка Redraw (перерисовать), а второй (Художник) скрытый — именно он рисует розу. Пользователь контролирует количество лепестков, меняя число *n* с помощью бегунка и нажимая кнопку Redraw, чтобы нарисовать розу. Когда пользователь нажимает эту кнопку, спрайт кнопки передает сообщение **Перерисовать**. Когда спрайт Художник получает это сообщение, он выполняет скрипт, приведенный на рис. 5.31.

Рис. 5.31. Процедура **Перерисовать** *для изображения розы с n лепестками на* **Сцене**

Сначала скрипт задает цвет и размер пера и убирает предыдущие отметки, оставленные им на **Сцене**. Затем он устанавливает значение переменной *a* равным 100 и обращается к процедуре **Роза** (**Rose**), которая пройдет через 360 повторов, рисуя на **Сцене** цветок. При каждом повторе процедура будет указывать в направлении тета, делать *r* шагов и ставить в этом месте точку. Затем тета будет меняться на 1 градус — при подготовке к очередному повтору.

На рис. 5.32 изображено несколько роз, нарисованных при разных значениях *n*. Сможете вычислить, как соотносится значение *n* и количество лепестков у цветка?

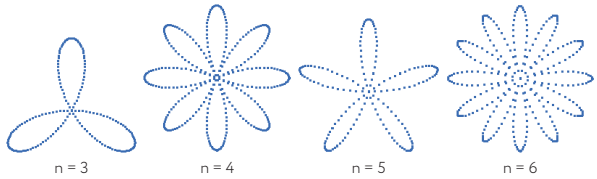


Рис. 5.32. Некоторые цветы, созданные с помощью программы для рисования розы

УПРАЖНЕНИЕ 5.6

Откройте приложение и запустите его. Измените значение *n*, чтобы увидеть, что еще можно создать с помощью этой процедуры. Добавьте к приложению еще один бегунок, чтобы пользователь мог менять значение *a* и при необходимости модифицировать скрипты. Вы также можете изменить алгоритм, сделав величину *a* параметром. (См. «Как присвоить пользовательским блокам новые параметры» на с. 97, чтобы освежить свои знания о том, как добавлять к процедурам параметры.)

Моделирование размещения семян на цветке подсолнуха

Sunflower.sb2

Биологи и математики много изучали расположение листьев на стеблях растений. Давайте и мы окунемся ненадолго в ботанику и исследуем геометрическую модель цветов со спиральным размещением семян. В частности, мы будем программировать два уравнения, которые моделируют размещение семян на цветке подсолнуха. Чтобы нарисовать *n*-ное семечко, мы должны выполнить следующие действия.

1. Повернуть спрайт в направлении $n \times 137{,}5°$.

2. Передвинуть его на расстояние $r = c\sqrt{n}$, где c — постоянный масштабный коэффициент (в нашем примере — 5).

3. Поставить точку на **Сцене**.

Мы повторим эти шаги для каждого семечка, которое хотим нарисовать. Для первого $n = 1$, для второго $n = 2$ и т. д. Если мы выберем при первом шаге угол не 137,5°, то семечки образуют другой узор.

Если вам любопытно, что это за уравнения, и вы хотите больше узнать об узорах, которые образуют семена подсолнечника, обратитесь к книге «Алгоритмическая красота растений» Пшемыслава Прущинкевича и Аристида Линденмайера (2004)*.

* На русский язык переведены только отдельные фрагменты, см. http://kildekode.ru/biotech/5673/L-systems-Modelirovanie-derev-ev.htm.

Наше приложение будет создавать узоры, схожие с теми, что описаны в этой книге, и некоторые из них представлены на рис. 5.33.

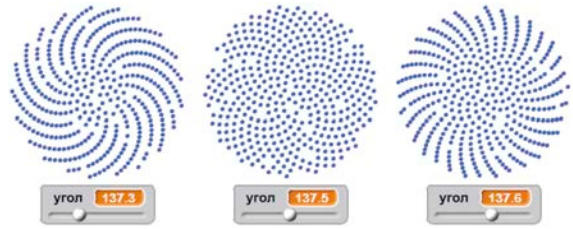


Рис. 5.33. Некоторые узоры расположения семян подсолнечника, созданные с использованием различных углов

Интерфейс в данном случае оснащен бегунком, с помощью которого можно менять угол от 137° до 138° с шагом 0,01°, а также кнопкой **Перерисовать**. Когда пользователь нажимает эту кнопку, она отсылает сообщение спрайту Художник, который исполняет скрипт, представленный на рис. 5.34.

Процедура **Подсолнух** повторяется, рисуя 420 семян, хотя вы можете изменить это число, если хотите. При каждом повторении процедура переходит к месту размещения n-ного семечка (рассчитывая угол семечка ❶ и двигаясь на $c\sqrt{n}$ шагов ❷) и ставит пером точку в этом месте. После этого алгоритм увеличивает значение n, которое представляет собой номер семечка, чтобы подготовиться к рисованию следующего.

Рис. 5.34. Скрипты для спрайта Художник

Скрипты, представленные в этом разделе, — лишь немногие из примеров удивительных приложений, которые можно создать с помощью переменных и мониторов. Возможность взаимодействовать с нашими приложениями с помощью бегунков — только начало работы с интерактивными приложениями. Далее вы научитесь создавать скрипты, которые напрямую требуют введения пользователями информации.

УПРАЖНЕНИЕ 5.7

Откройте приложение и запустите его. Измените величину угла, чтобы посмотреть, что еще можно сделать с процедурой **Подсолнух**. Изучите алгоритм, чтобы понять, как он работает, и подумайте, как можно его усовершенствовать.

Получаем данные от пользователя

GettingUserInput.sb2

Представьте, что вы хотите создать игру, которая бы обучала детей азам арифметики. Скорее всего, в ней будет спрайт, который выглядит как задачка на сложение и просит пользователя ввести ответ. Как бы вы прочли введенную пользователем информацию, чтобы понять, правилен ответ или нет?

В разделе **Сенсоры** имеется блок-команда **спросить и ждать**, который вы можете применить, чтобы прочесть введенную пользователем информацию. Он использует единственный параметр, который определяет показываемую пользователю строку, обычно в форме вопроса. Как вы видите на рис. 5.35, выполнение этого блока дает несколько различных результатов в зависимости от того, является спрайт видимым или невидимым. Результат, приведенный на рис. 5.35 справа, также возникает, когда команда **спросить и ждать** получена от скрипта, который принадлежит **Сцене**.

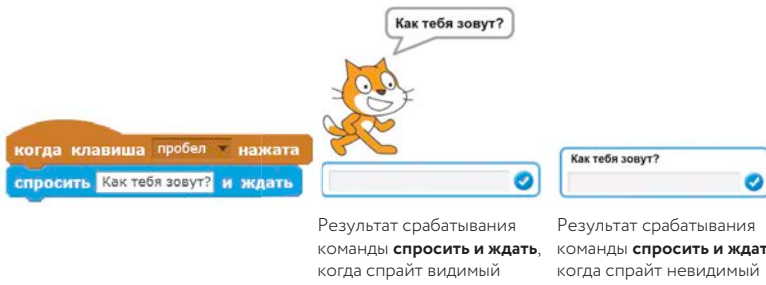


Рис. 5.35. Блок **спросить и ждать** *может давать разные результаты в зависимости от того, является ли исполняющий его спрайт видимым или невидимым*

После выполнения команды **спросить и ждать** вызывающий скрипт ждет, пока пользователь нажмет клавишу Enter или кликнет по галочке справа от поля ввода. Когда это происходит, Scratch сохраняет введенную пользователем информацию в блоке **ответ** и продолжает выполнение команды, следующей за блоком **спросить и ждать**. Чтобы увидеть этот блок-команду в работе, взгляните на приведенные ниже примеры, которые показывают, как ее использовать.

Считывание числа

AskAndWait.sb2

Скрипт на рис. 5.36 спрашивает пользователя о возрасте, ждет ответа и, получив его, сообщает пользователю, сколько ему будет через 10 лет.

Рис. 5.36. Скрипт, который принимает в качестве введенной пользователем информации его возраст

Рисунок показывает выходные данные программы в ответ на ввод пользователем числа 18 и нажатие Enter на клавиатуре. Обратите внимание, что программа использует блок **слить** (из раздела **Операторы**), чтобы объединить строки.

Считывание символов

AskAndWait2.sb2

Скрипт на рис. 5.37 запрашивает у пользователя его инициалы и затем создает и отображает приветствие исходя из ответа.

Программа использует две переменные (инициал1 и инициал2), чтобы сохранять введенные пользователем величины. Вы можете наблюдать результат работы программы, когда пользователь вводит в два предложенных окошка буквы M и C соответственно. Программа использует вложенные блоки **слить**, чтобы сконструировать приветствие. Вы можете использовать эту технику для того, чтобы создавать самые разные строки и отображать индивидуализированные сообщения в ваших приложениях.

Рис. 5.37. Скрипт, который использует две переменные, чтобы считывать и сохранять инициалы пользователя

Выполнение арифметических операций

Скрипт на рис. 5.38 просит пользователя ввести два числа, затем умножает одно на другое и отображает ответ в «пузырьке», используя команду **сказать**. Как и в предыдущем случае, скрипт использует две переменные (номер1 и номер2), чтобы сохранить величины, введенные пользователем.

AskAndWait3.sb2

Рис. 5.38. Вычисление величины на основе введенной пользователем информации

Рисунок показывает результат перемножения двух введенных пользователем чисел, 8 и 9. И снова обратите внимание, как я соединил блоки **слить**, чтобы сконструировать строку вывода.

Примеры, приведенные в этом разделе, демонстрируют несколько способов использования блока **спросить и ждать** для того, чтобы получить информацию от пользователя и решить множество задач. Например, вы можете написать программу, которая вычисляет корни квадратного уравнения вида $ax^2 + bx + c = 0$ для различных величин a, b и c, введенных пользователем. Затем вы можете использовать эту программу, чтобы проверить свой ответ. Надеюсь, теперь вы понимаете, как можно использовать этот ценный блок для решения любой математической задачи.

Итоги

Переменные — одна из основ программирования. Это имя области в машинной памяти, где мы можем сохранить единственную величину: число или строку.

Из этой главы вы узнали, какие основные типы данных поддерживает Scratch и какие операции разрешены с этими типами. Затем я рассказал, как создавать переменные и использовать их для сохранения данных.

Вы создали несколько приложений, которые использовали переменные для демонстрации тех или иных функций. Вы разобрались с мониторами переменных и использовали их для создания разных видов интерактивных программ. Наконец, вы научились использовать блок **спросить и ждать**, чтобы получать от пользователя информацию и обрабатывать ее в своей программе.

В следующей главе вы больше узнаете о логических величинах и о том, насколько они важны в принятии решений. Вы также познакомитесь с блоками **если** и **если/иначе** и будете использовать их для того, чтобы вывести ваши программы на новый интеллектуальный уровень. Так что закатывайте рукава и готовьтесь к еще одной захватывающей главе!

Задания

1. Создайте скрипт, который выполняет следующие действия.
   - Задать переменной скорость значение 60 (км/ч).
   - Задать переменной время значение 2,5 (часа).
   - Рассчитать пройденное расстояние и сохранить ответ в переменной расстояние.
   - Отобразить рассчитанное расстояние, показав пользователю соответствующее сообщение.

2. Каков будет результат выполнения каждого из приведенных ниже скриптов? Воспроизведите и выполните их, чтобы проверить ваш ответ.

3. Каковы значения *X* и *Y* в конце каждого цикла повторения скрипта справа? Воспроизведите скрипт и выполните его, чтобы проверить ваш ответ.

4. Пусть у нас будут две переменные — *x* и *y*. Создайте блоки-функции, эквивалентные следующим вычислениям.
   - Сложить *x* и 5 и сохранить полученный результат как *y*.
   - Умножить *x* на 3 и сохранить полученный результат как *y*.
   - Поделить *x* на 10 и сохранить полученный результат как *y*.
   - Вычесть 4 из *x* и сохранить полученный результат как *y*.
   - Возвести *x* в квадрат, к результату прибавить *y* и сохранить сумму снова как *x*.
   - Пусть *x* будет равен сумме удвоенного *y* и утроенного куба *y*.
   - Пусть *x* будет равен отрицательному квадрату *y*.
   - Пусть *x* будет равен результату деления суммы *x* и *y* на произведение *x* и *y*.

5. Напишите программу, которая попросит пользователя ввести существительное и глагол. Пусть затем программа создаст предложение типа *существительное + глагол*.

6. Напишите программу, которая просит пользователя ввести температуру в градусах Цельсия. Эта программа будет преобразовывать температуру в градусы по Фаренгейту и показывать результат пользователю в виде уместного сообщения. (Подсказка: °F = (1,8 × °C) + 32.)

7. Когда ток *I* проходит через резистор, обеспечивающий сопротивление (*R*), энергия *P*, рассеиваемая резистором, равна $I^2 \times R$. Напишите программу, которая, получив *I* и *R*, рассчитывает *P*.

8. Напишите программу, которая считывает длины двух катетов прямоугольного треугольника и рассчитывает длину гипотенузы.

9. Напишите программу, которая просит пользователя ввести длину (*L*), ширину (*W*) и высоту (*H*) ящика. Затем программа рассчитает и отобразит объем и площадь поверхности ящика. (Подсказка: объем = $L \times \times W \times H$; площадь поверхности = $2 \times [(L \times W) + (L \times H) + (H \times W)]$.)

10. Эквивалентное сопротивление *R* трех резисторов (R_1, R_2, R_3), соединенных параллельно, дано в этом уравнении:

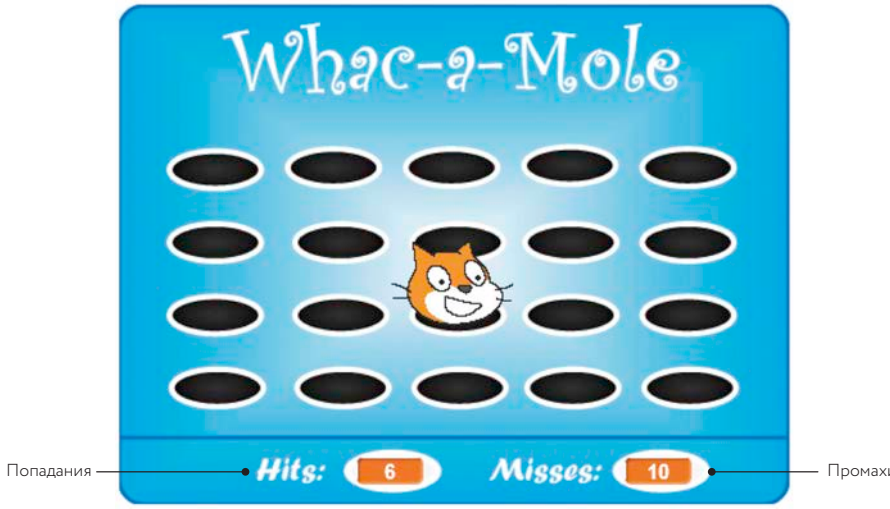
$$1/R = 1/R_1 + 1/R_2 + 1/R_3.$$

Напишите программу, которая считывает величины R_1, R_2 и R_3 и рассчитывает *R*.

Whac-a-Mole.sb2

11. Закончите игру «Стукни крота», о которой шла речь в начале главы. Файл *Whac-a-Mole.sb2* содержит частичную реализацию программы. После клика по зеленому флажку скрипт из файла начинает цикл, в ходе которого спрайт Кот появляется из случайных дыр. Добавьте два скрипта (один для Кота и другой для **Сцены**), чтобы поменять величины двух переменных (попадания и промахи). Попробуйте добавить звуковые эффекты, чтобы сделать игру еще веселее! Вы также можете добавить условие, которое завершает игру по окончании работы таймера или после того, как количество промахов достигнет определенной величины.

«Стукни крота»



6

ПРИНЯТИЕ РЕШЕНИЙ

Из этой главы вы узнаете об инструментах Scratch, которые нужны, чтобы писать программы, сравнивающие величины, оценивающие логические выражения и принимающие решения на основе результатов. Мы также рассмотрим несколько примеров полезных программ. Вот что вы узнаете из этой главы:

- каковы базовые техники решения задач;
- как использовать блоки **если** и **если/иначе** при выборе из нескольких альтернативных действий;
- как конструировать логические выражения для оценки заданных условий;
- как выглядит процесс управления операторами ветвления.

Программы, которые мы писали до сих пор, следовали простой модели. Они выполняют первую команду, переходят к следующей, и так до тех пор, пока не дойдут до конца. Блоки-команды выполняются один за другим, без пропусков или прыжков.

Но в программировании часто бывают ситуации, когда вам хочется изменить этот процесс. Если вы пишете программу для обучения детей основам арифметики, вы захотите, чтобы определенные блоки запускались в качестве вознаграждения за правильные ответы и совсем другие — если ответы были неправильными (например, чтобы сигнализировать о правильном ответе или предложить еще одну попытку). Ваш скрипт может решать, что делать дальше, сравнив ответ ученика

с правильным. Это основа всех задач на принятие решения. В этой главе мы разберем имеющиеся в среде Scratch команды на принятие решений и напишем несколько программ, которые будут использовать их для проверки вводимых данных и выполнения различных действий.

Для начала я познакомлю вас с операторами сравнения среды Scratch и покажу, как использовать их для сравнения чисел, букв и строковых последовательностей. Затем я введу блоки **если** и **если/иначе** и разъясню их ключевую роль в принятии решений. Вы также узнаете, как тестировать множественные условия, используя вложенные блоки **если** и **если/иначе** и написать управляемую с помощью меню программу, которая будет запускать их. После этого я расскажу о логических операторах как об альтернативном пути для тестирования множественных условий. В последнем разделе мы напишем несколько интересных программ, основанных на всех тех функциях, с которыми мы уже успели познакомиться.

Операторы сравнения

Вам приходится принимать решения каждый день, и большинство из них приводит к действию. Например, вы можете сказать себе: «Если эта машина стоит меньше 2000 долларов, я ее куплю». Затем вы интересуетесь ценой машины и решаете, будете вы ее покупать или нет.

В Scratch вы тоже можете принимать решения. При помощи *операторов сравнения* вы сравниваете значения двух переменных или выражений, чтобы определить разницу между ними. Операторы сравнения также называются *реляционными операторами*, потому что проверяют соотношение двух значений. В таблице 6.1 показаны три реляционных оператора, поддерживаемых Scratch.

Таблица 6.1. Реляционные операторы в Scratch

Оператор	Значение	Пример
	Больше чем	 Цена больше 2000?
	Меньше чем	 Цена меньше 2000?
	Равно	 Цена равна 2000?

БУЛЕВЫ, ИЛИ ЛОГИЧЕСКИЕ, ВЫРАЖЕНИЯ В РЕАЛЬНОМ МИРЕ

Само слово *булевы* используется в память о Джордже Буле — британском математике, жившем в XIX веке и разработавшем логическую систему, основанную на двух значениях: 1 и 0 (истина и ложь). Булева алгебра стала основой современной компьютерной науки.

Мы постоянно используем булевы выражения для принятия решений. Компьютеры тоже пользуются ими, чтобы определить ветку, которой последует программа. Роботизированная рука может быть запрограммирована проверять подвижные детали на конвейере и перемещать в Корзину 1 те части, у которых хорошееКачество = истина, а в Корзину 2 — для которых хорошееКачество = ложь. Системы домашней безопасности обычно программируются так, чтобы при введении неверного кода включалась сигнализация (верныйКод = ложь), а правильного — выключалась (верныйКод = истина). Удаленный сервер может предоставлять или отказывать в доступе, если прокатать свою кредитную карту в магазине, в зависимости от того, действует карта (истина) или нет (ложь). Компьютер в автомобиле автоматически выбросит подушки безопасности, если решит, что произошло столкновение (столкновение = истина). Мобильный телефон может показывать специальную иконку, когда батарея садится (батареяРазряжена = истина), и скрывать ее, когда заряд достаточный (батареяРазряжена = ложь).

Это несколько примеров того, как компьютеры инициируют различные действия, проверяя результаты булевых условий.

Обратите внимание, что все блоки в табл. 6.1 имеют шестиугольную форму. Как вы, возможно, помните из главы 5, это означает, что результат оценки этих блоков — *логическое значение*, которое либо верно, либо нет. Поэтому такие выражения также называют логическими. Например, выражение **цена < 2000** проверяет значение переменной цена. Если она меньше 2000, блок оценивается как истина; если нет — как ложь. Вы можете использовать это выражение, чтобы сформулировать условия принятия решения в формате «Если (**цена < 2000**), то покупаем машину».

Прежде чем мы взглянем на блок **если**, который позволяет провести такую проверку, рассмотрим простой пример, иллюстрирующий, как логические выражения оцениваются в Scratch.

Оцениваем булевы выражения

Предположим, у нас есть две переменные, x и y, далее: $x = 5$ и $y = 10$. В таблице 6.2 показаны несколько примеров использования блоков отношений Scratch.

Эти примеры показывают важные моменты, связанные с реляционными операторами. Во-первых, мы можем использовать их для сравнения как отдельных переменных (x, y), так и целых выражений (например, $2 \times x$ и $x + 6$). Во-вторых, результат сравнения всегда либо верен,

либо неверен (это всегда булево значение). В-третьих, блок **x = y** не означает «Задать значение x равным y». Он спрашивает: «x равен y?» Таким образом, когда утверждение **задать z значение (x = y)** выполняется, значение *x* по-прежнему равно 5.

Таблица 6.2. Образцы использования блоков отношений

Утверждение	Значение	z (результат)	Объяснение
	z = верно ли что (5 < 10)?	z = истина	Потому что 5 меньше 10
	z = верно ли что (5 > 10)?	z = ложь	Потому что 5 не больше 10
	z = верно ли что (5 = 10)?	z = ложь	Потому что 5 не равно 10
	z = верно ли что (10 > 2 × 5)?	z = ложь	Потому что 10 не больше 10
	z = верно ли что (5 = 5)?	z = истина	Потому что 5 равно 5
	z = верно ли что (10 < 5 + 6)?	z = истина	Потому что 10 меньше 11

Сравниваем буквы и строки

Представим себе игру, в которой пользователь пытается угадать секретный код, состоящий из одной буквы от А до Я. Игра считывает вариант, предложенный пользователем, сравнивает его с секретным кодом и инструктирует пользователя, как ему улучшить свой ответ на основе порядка букв в алфавите. Если секретная буква, скажем, Ё, а введен ответ Б, игра должна сказать что-то вроде «После Б», тем самым дав понять, что секретный код в алфавите стоит за буквой Б. Как сравнить правильную букву с ответом игрока, чтобы решить, какое сообщение нужно вывести на экран?

К счастью, реляционные операторы в Scratch умеют сравнивать буквы. Как показано на рис. 6.1, Scratch сравнивает буквы на основании порядка их расположения в алфавите. Поскольку А стоит в алфавите раньше Б, выражение **А < Б** верно. Тут важно помнить, что такого рода сравнения нечувствительны к регистру. Заглавная А — то же самое, что и прописная а. Поэтому выражение **А = а** тоже верно.

Зная это, вы можете проверить догадку игрока при помощи следующего набора условий:

ЕСЛИ (ответ = секретный код), то сказать «Правда»
ЕСЛИ (ответ > секретный код), то сказать «До <ответ>»
ЕСЛИ (ответ < секретный код), то сказать «После <ответ>»

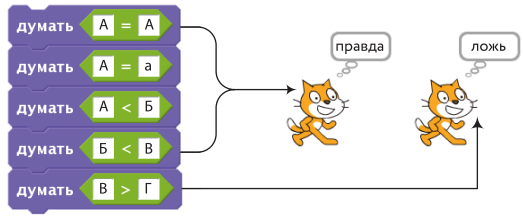


Рис. 6.1. Использование реляционных операторов для сравнения букв

Условие — это высказывание в форме «если условие верно, совершить такое-то действие». В следующем разделе я научу вас вводить условия в Scratch, а пока подробнее рассмотрим реляционные операторы на примере игры в угадывание секретного кода.

Что если код содержит более одной буквы? Например, игроку нужно угадать название животного. Сможем ли мы использовать реляционные операторы Scratch, чтобы провести сравнение? Ответ прост: да! Вы можете использовать реляционные операторы, чтобы сравнивать строки. Но как именно Scratch производит сравнение типа **слон > хомяк**? Примеры на рис. 6.2 демонстрируют результаты сравнения строк.

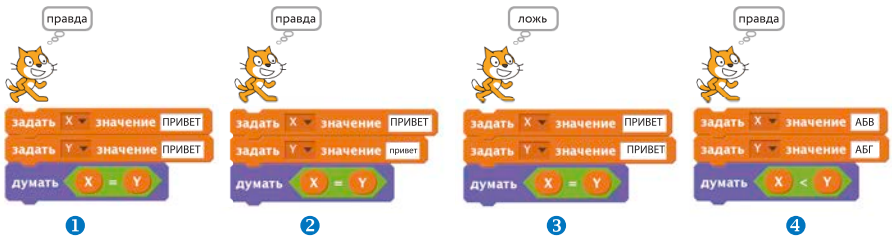


Рис. 6.2. Используя реляционные операторы для сравнения ❶ идентичных строк, ❷ строк, различающихся только регистром, ❸ одной строки с другой, которая содержит дополнительные пробелы, и ❹ строк, которые различаются в соответствии со словарным порядком букв

Внимательное изучение рис. 6.2 показывает следующее.

- Scratch сравнивает строки независимо от их регистра. Строки «ПРИВЕТ» и «привет» ❷, например, считаются одинаковыми.

- Scratch учитывает пробелы при сравнении. Строка «ПРИВЕТ», которая начинается и заканчивается пробелом, — не то же самое, что и строка «ПРИВЕТ» ❸.

- Сравнивая строки «АБВ» и «АБГ» ❹, Scratch сначала рассматривает первый символ в обеих. Поскольку он одинаковый, Scratch проверяет второй символ. Он тоже одинаковый, и Scratch переходит к проверке третьего. Поскольку буква *В* меньше буквы *Г* (*В* идет в алфавите раньше *Г*), Scratch считает, что первая строка меньше второй.

Зная это, вы не должны удивляться, что выражение **слон > хомяк** окажется неверным, хотя что в реальной жизни слоны гораздо больше хомяков. По принятым в Scratch правилам сравнения строк строка «слон» меньше строки «хомяк», потому что буква с (первая в слове «слон») идет в алфавите раньше, чем буква х (первая в слове «хомяк»).

Сравнение и сортировка строк на основе алфавитного порядка их букв используется во многих реальных ситуациях: приведение в порядок каталогов файлов, книг на полках, слов в словарях и т. д. Слово «слон» в словаре будет идти раньше слова «хомяк», и результат сравнения строк в Scratch основан именно на этом принципе.

Теперь, когда мы разобрались с тем, что такое реляционные операторы и как Scratch использует их для сравнения чисел и строк, пришло время узнать о блоках условных операторов.

Структуры решений

Раздел **Управление** в Scratch содержит два блока, позволяющих принимать решения и управлять действиями в ваших программах: **если** и **если/иначе**. Используя эти блоки, вы можете задать вопрос и совершить действие на основании ответа. В этом разделе мы подробно обсудим эти два блока, поговорим о флагах и научимся тестировать множественные условия при помощи блоков **если**. Затем я познакомлю вас с программами, управляемыми через меню, и объясню, как вложенные блоки **если** могут помочь в их работе.

Блок если

Блок **если** — структура принятия решений, которая дает вам возможность обозначить, должен ли какой-то набор команд выполняться согласно условиям теста. Структура блока **если** и схема его процессов показаны на рис. 6.3.

На рис. 6.3 форма бриллианта играет роль блока, дающего ответы да/нет (или истина/ложь) на вопрос. Если тестовое условие в шапке блока **если** верно, программа выполняет команды из блока, прежде чем перейти к команде, следующей за ним (на рисунке это **Команда М**). Если тестовое условие неверно, программа пропускает эти команды и сразу переходит к **Команде М**. Чтобы посмотреть на блок **если** в действии, создайте скрипт с рис. 6.4. Он запускает цикл **всегда**, перемещает спрайт

по **Сцене**, меняет его цвет и заставляет его отпрыгивать, когда он касается краев **Сцены**.

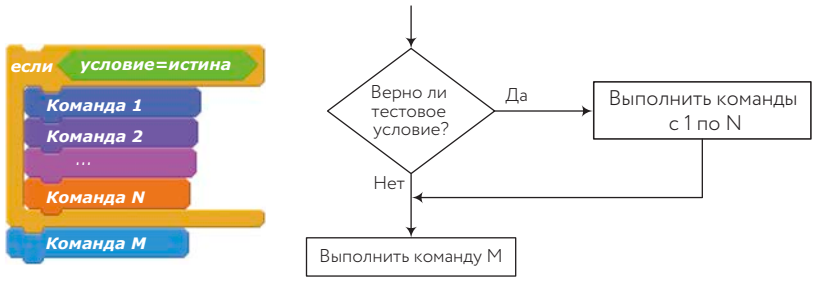


Рис. 6.3. Структура блока **если**

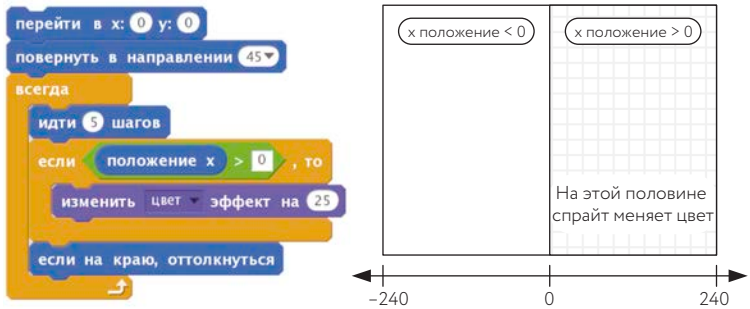


Рис. 6.4. Этот скрипт заставляет спрайт менять свой цвет, когда он движется по правой стороне **Сцены**

Цикл **всегда** содержит в вашем скрипте блок **если**, который проверяет положение по оси *x* после каждой команды **идти**. Если позиция по оси *x* больше нуля, спрайт должен сменить цвет. Когда вы запустите этот скрипт, вы заметите, что спрайт меняет цвет, только когда движется по правой половине **Сцены**. Это связано с тем, что блок **изменить цвет эффект на 25** выполняется только тогда, когда условие **положение x > 0** верно.

Используем переменные как флаги

Представим, что вы разрабатываете игру о приключениях в космосе, где цель в том, чтобы разрушить атакующий вас флот боевых кораблей. Игрок — капитан корабля, управляет им при помощи клавиш со стрелками и стреляет ракетами при помощи «пробела». Если в космический корабль игрока определенное количество раз попадает вражеская ракета, корабль теряет способность атаковать. С этого момента нажатие на клавишу «пробел» больше не должно приводить к запуску ракет, и капитану приходится принять оборонительную стратегию, чтобы

не допустить еще большего количества попаданий. Очевидно, что при нажатии клавиши «пробел» ваша программа должна проверять состояние корабля, чтобы принять решение, может ли игрок стрелять.

Обычно проверки такого рода осуществляются при помощи *флагов* — переменных, которые вы используете для выявления того, произошло ли интересующее вас событие. Вы можете взять любые два значения, чтобы описать статус события, но на практике обычно используют 0 (или ложь), чтобы показать, что событие не произошло, и 1 (или правда), чтобы показать, что оно произошло.

В вашей космической стрелялке вы можете использовать флаг под названием могуСтрелять, чтобы обозначить состояние корабля. Тогда 1 будет означать, что корабль может стрелять ракетами, а 0 — не может. На основе этого код для обработчика события — нажатия клавиши «пробел» — может быть таким, как показано на рис. 6.5. В начале игры вы устанавливаете 1 в качестве значения флага могуСтрелять, чтобы показать, что корабль способен стрелять. Когда в корабль попадет определенное количество вражеских выстрелов, вы установите флаг могуСтрелять на 0, чтобы показать, что боевая система вышла из строя. С этого момента нажатие на «пробел» не будет приводить к пуску ракет.

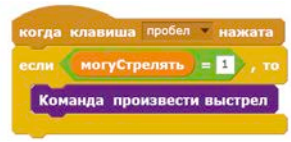
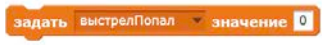


Рис. 6.5. Использование флага для выполнения условия

Вы можете называть свои флаги как хотите, но советую выбирать осмысленные имена. В таблице 6.3 приведены несколько примеров флагов, которые вы могли бы использовать в своей космической «стрелялке».

Таблица 6.3. Несколько примеров использования флагов

Пример	Значение и возможные последующие действия
задать играНачалась значение 0	Игра еще не началась. Игнорировать любые нажатия клавиш
задать играНачалась значение 1	Игра началась. Начать принимать данные от пользователя
задать играЗакончена значение 0	Игра еще не закончилась. Показать остающееся время
задать играЗакончена значение 1	Игра закончена. Скрыть дисплей с остающимся временем

Пример	Значение и возможные последующие действия
	Вражеский выстрел не попал в космический корабль. Аварийный сигнал выключен
	Вражеский выстрел попал в космический корабль. Аварийный сигнал включен

Теперь, когда вы знаете, как использовать блок **если** и флаги, поговорим о другом блоке условия, который позволит вам запускать один блок кода, когда определенное условие верно, и другой, когда оно неверно.

Блок если/иначе

Представьте себе, что вы создаете игру, которая должна обучать школьников младших классов основам математики. Игра предлагает задачу на сложение, а затем просит ввести ответ. Ученик должен получить балл за правильный ответ или потерять балл за неправильный. Вы можете выполнить эту задачу при помощи двух утверждений **если**:

Если ответ правильный, добавить один балл на счет
Если ответ неправильный, вычесть один балл со счета

Вы также можете упростить эту логику и сделать код более эффективным, скомбинировав два утверждения в одно **если/иначе**:

Если ответ правильный, добавить один балл на счет.
Иначе вычесть один балл со счета

Выбранное условие протестировано. Если оно верно, выполняются команды из части блока **если**. Если же неверно, выполняются команды из части **иначе**. Программа будет выполнять только одну группу команд блока. Такие альтернативные пути внутри программы называются ветвями. Структура блока **если/иначе** и соответствующая ему блок-схема показаны на рис. 6.6.

Вы можете использовать структуру **если/иначе**, когда вам нужно выбрать, где сегодня пообедать. Если у вас достаточно денег, вы пойдете в крутой ресторан; если же нет, вы остановите свой выбор на более простой еде. Назовем деньги в вашем кошельке доступнаяНаличность. Когда вы открываете кошелек, вы проверяете состояние доступнаяНаличность > $20. Если условие верно (у вас больше 20 долл.), вы идете в заведение с белыми скатертями; если же нет — в ближайшую закусочную.

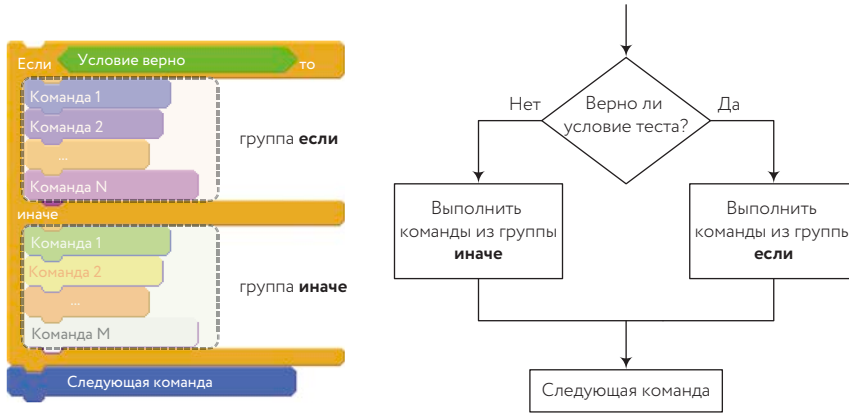


Рис. 6.6. Структура блока ***если/иначе***

На рис. 6.7 показан один простой скрипт, иллюстрирующий использование блока **если/иначе**. В этом примере используется оператор модуля (**модуль**), который выдает остаток от операции деления, чтобы определить, четное число было введено или нечетное. (Вы помните, что у четного числа при делении на два в остатке будет 0?)

Рис. 6.7. Этот скрипт определяет, ввел ли пользователь четное число или нечетное

Рис. 6.7 показывает два примера результата ввода пользователем чисел 6 и 9 соответственно в ответ на команду **спросить**. А вы можете объяснить, как работает этот скрипт?

Вложенные блоки если и если/иначе

Если вы хотите протестировать более одного условия, прежде чем совершить действие, вы можете вложить несколько блоков **если** (или **если/иначе**) друг в друга, чтобы выполнить проверку. Рассмотрим скрипт на рис. 6.8, который определяет, должен ли студент получать стипендию. Чтобы получить ее, студент должен иметь: средний балл (GPA) выше 3,8 и оценку по математике выше 92%.

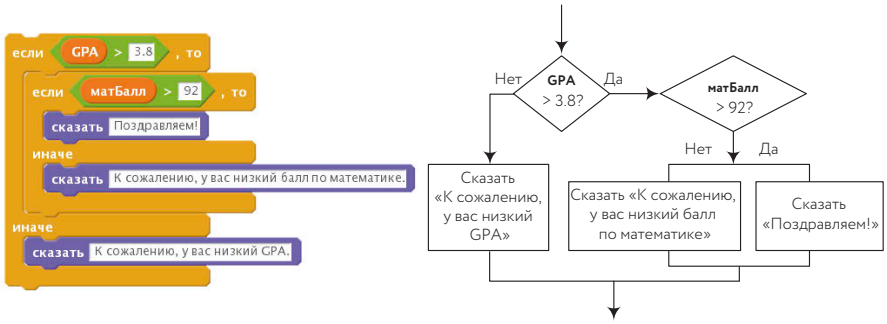


Рис. 6.8. Вы можете использовать вложенные блоки **если/иначе**, *чтобы протестировать несколько условий*

Первым делом тестируется выражение **GPA > 3.8**. Если оно неверно, нам не нужно проверять другое условие: студент не соответствует критериям для получения стипендии. А вот если **GPA > 3.8** верно, нужно проверить второе условие. Это делается при помощи вложенного блока **если/иначе**, который проверяет условие **матБалл > 92**. Если оно тоже верно, студент получает стипендию. Если нет — студент не получает стипендию и на экране появляется соответствующее сообщение.

Программы, управляемые с помощью меню

AreaCalculator.sb2

Рассмотрим типичный случай использования вложенных блоков **если**. В частности, вы узнаете, как писать программы, которые предлагают пользователю варианты на выбор и действуют в соответствии с принятым решением.

Некоторые программы, когда их запускают, отображают список (или меню) доступных вариантов и ждут, пока вы сделаете свой выбор. Иногда вы взаимодействуете с ними путем ввода цифры, соответствующей выбранному варианту. Такие команды можно использовать для определения выбора пользователя и совершения соответствующих действий с помощью последовательности из вложенных блоков **если/иначе**. Чтобы узнать, как работают вложенные блоки **если/иначе**, мы обсудим программу, показанную на рис. 6.9. Она рассчитывает площадь различных геометрических форм.

Пользовательский интерфейс этой программы содержит фоновое изображение **Сцены**, на котором показаны доступные варианты (цифры 1, 2 или 3) и спрайт Наставник (Tutor), который предлагает выбор, делает все вычисления и отображает результат. Основной скрипт, показанный на рис. 6.10, запускается нажатием на зеленый флажок.

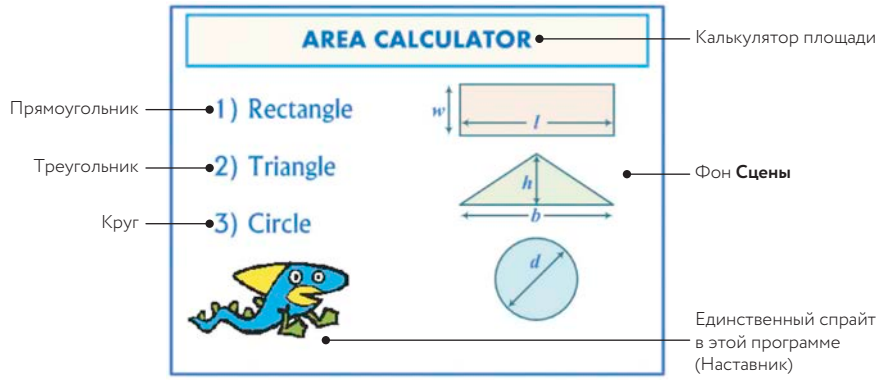


Рис. 6.9. Пользовательский интерфейс для программы — калькулятора площади

Рис. 6.10. Главный скрипт для спрайта Наставник

После того как пользователя попросят дать ответ, спрайт Наставник дожидается, пока тот введет цифру, и использует три блока **если/иначе**, чтобы эту информацию обработать. Если пользователь ввел корректный вариант (1, 2 или 3), скрипт обратится к соответствующей процедуре, чтобы вычислить площадь обозначенной фигуры. Иначе он запускает команду **сказать**, чтобы проинформировать пользователя о том, что его ответ некорректен.

На рис. 6.11 вы видите все три процедуры для вычисления площадей фигур.

Рис. 6.11. Процедуры для программы, вычисляющей площадь фигур

Каждый алгоритм просит пользователя ввести размеры фигуры, вычисляет площадь, а затем отображает результат. Например, **Прямоугольник** просит пользователя ввести длину и ширину прямоугольника и сохраняет ответы в качестве переменных длина и ширина соответственно. Затем вычисляет площадь, перемножив длину и ширину, и выводит ответ. Остальные две процедуры работают аналогично.

Логические операторы

В предыдущем разделе я рассказал, как использовать вложенные блоки **если** и **если/иначе**, чтобы проверить несколько условий. Но это можно сделать и при помощи логических операторов.

Используя логические операторы, вы можете комбинировать два или более выражения отношений, чтобы получить один результат формата правда/ложь. Например, логическое выражение **(x > 5) и (x < 10)** состоит из двух логических выражений ($x > 5$ и $x < 10$), которые соединены при помощи логического оператора **и**. Мы можем представить себе $x > 5$ и $x < 10$ как два компонента оператора **и**; результат верен, только когда верны оба компонента. В таблице 6.4 приведены три логических оператора, используемых в Scratch, с коротким объяснением их значения.

Таблица 6.4. Логические операторы

Оператор	Значение
	Результат верен, только если оба утверждения верны
	Результат верен, если одно из двух утверждений верно
	Результат верен, если утверждение неверно

Вы ознакомились с кратким обзором каждого из операторов. Теперь изучим более подробно, как работает каждый из них.

Оператор И

Оператор **и** использует в качестве параметров два выражения. Если оба верны, **и** оказывается истиной, если нет — ложью. В таблице 6.5 приводятся результаты для этого оператора по всем возможным комбинациям.

Таблица 6.5. Таблица истины для оператора **и**

X	Y	
истина	истина	истина
истина	ложь	ложь
ложь	истина	ложь
ложь	ложь	ложь

В качестве примера использования оператора **и** мы можем, скажем, разрабатывать игру, в которой пользователь получает 200 бонусных очков, если на первом уровне его счет достигает 100 баллов. Уровень игры отслеживается переменной под названием уровень, а счет — переменной под названием счет. Рис. 6.12 показывает, как эти условия могут быть проверены при помощи вложенных блоков **если** ❶ или оператора **и** ❷.

Рис. 6.12. Проверка множественных условий при помощи вложенных блоков **если** *и оператора* **и**

В обоих случаях бонусные очки добавлялись только в том случае, если оба условия выполнялись. Как вы видите, оператор **и** предлагает более компактный способ провести ту же проверку. Команды внутри блока **если** на рис. 6.12 ❷ будут выполнены, только если уровень равен 1, а счет — 100. Если одно из условий неверно, результат всей проверки оценивается как ложь и блок **изменить счет на 200** выполняться не будет.

Оператор ИЛИ

Оператор **или** также использует в качестве параметров два выражения. Если одно из них верно, **или** оказывается истиной. Оно будет ложью, если оба выражения неверны. В таблице 6.6 приводятся результаты для этого оператора по всем возможным комбинациям.

Таблица 6.6. Таблица истины для оператора **или**

X	Y	X или Y
истина	истина	истина
истина	ложь	истина
ложь	истина	истина
ложь	ложь	ложь

Чтобы продемонстрировать использование оператора **или**, представим себе, что в некой игре пользователи при попытке выйти на следующий уровень ограничены во времени. Они также начинают игру с определенным запасом энергии, которая расходуется по мере прохождения уровня.

Игра заканчивается, если игрок не может выйти на следующий уровень в доступное время или полностью израсходует всю энергию раньше, чем выйдет на следующий уровень. Остающееся время отслеживается при помощи переменной времяОсталось, а уровень энергии — переменной энергияОсталось.

Рис. 6.13 показывает, как эти условия могут быть проверены при помощи вложенных блоков **если/иначе** ❶ и оператора **или** ❷.

Рис. 6.13. Проверяем множественные условия при помощи вложенных блоков **если** *и оператора* **или**

Еще раз обратите внимание, что оператор **или** предлагает более компактный способ проверки множественных условий. Команды внутри блока **если** на рис. 6.13 ❷ будут выполнены, только если времяОсталось

или энергияОсталось равны 0. Если оба этих условия неверны, результат всей проверки оценивается как ложь и флаг играЗакончена не будет установлен на 1.

Оператор НЕ

Оператор **не** использует в качестве вводных данных только одно выражение. Результат верен, если выражение оказывается ложью, и неверен, если оно истинно (табл. 6.7).

Таблица 6.7. Таблица истины для оператора **не**

X	не X
истина	ложь
ложь	истина

Возвращаясь к нашей гипотетической игре, представим, что пользователь не может выйти на следующий уровень, пока его счет не превысит 100 очков. Это хороший момент для использования оператора **не** (рис. 6.14). Вы можете прочесть этот блок кода как «если счет не больше 100, выполнить команду внутри блока **если**».

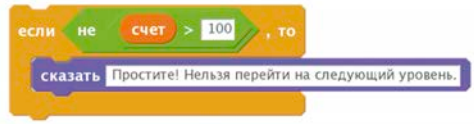


Рис. 6.14. Пример использования оператора **не**

Если значение переменной счет меньше или равно 100, то выражение оценивается как верное и команда **сказать** будет выполнена. Обратите внимание на то, что выражение **не (счет > 100)** эквивалентно выражению **(счет ⩽ 100)**.

Использование логических операторов для проверки областей числовых значений

Когда вам нужно оценить данные, введенные другим пользователем, или отфильтровать ошибки, вы можете использовать логические операторы, чтобы определить, какие числа находятся внутри (или за пределами) области числовых значений. Таблица 6.8 демонстрирует несколько примеров областей числовых значений.

Таблица 6.8. Области числовых значений

Выражение	Значение
(*x* > 10) и (*x* < 20)	Оценивается как верное, если значение *x* больше 10 и меньше 20
(*x* < 10) или (*x* > 20)	Оценивается как верное, если значение *x* меньше 10 или больше 20
(*x* < 10) и (*x* > 20)	Всегда неверно: *x* не может одновременно быть и меньше 10, и больше 20

В Scratch нет встроенных инструментов для операторов ⩾ (больше или равно) и ⩽ (меньше или равно), но вы можете использовать логические операторы. Представим себе, что вам нужно проверить в вашей программе условие **x ⩾ 10**. Решение показано на рис. 6.15 ❶. Закрашенный кружок означает, что число 10 входит во множество решений.

Один из способов проверки этого условия показан на рис. 6.15 ❷. Рисунок демонстрирует множество решений для **x < 10**, где незакрашенный кружок означает, что соответствующая точка не входит во множество решений. Как вы видите, дополнительное решение (*x* не меньше 10) эквивалентно **x ⩾ 10**. Еще один способ провести проверку неравенства показан на том же рисунке ❸. Очевидно, что если **x ⩾ 10**, то либо *x* больше 10, либо *x* равен 10.

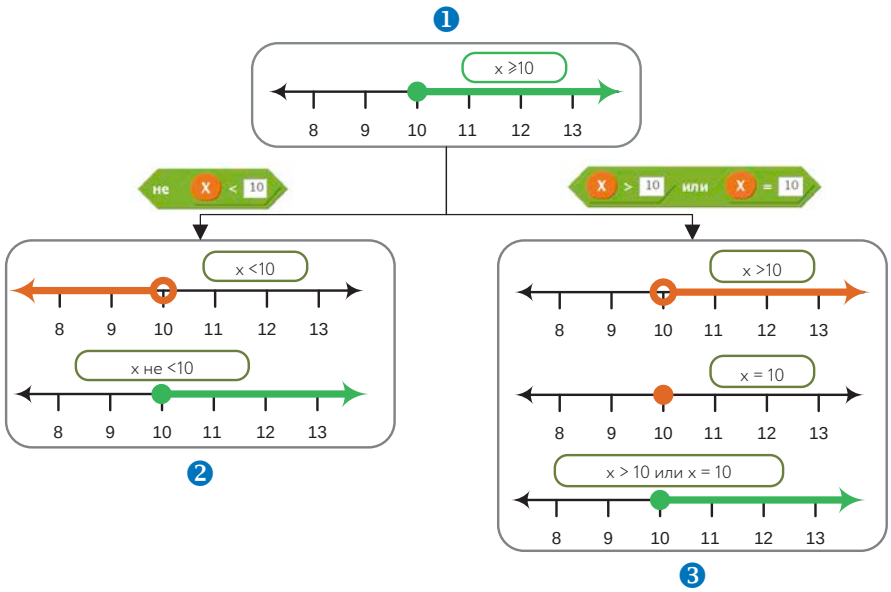


Рис. 6.15. Два способа реализации неравенства x ≥ 10

Примеры, предложенные в табл. 6.9, показывают, как использовать операторы отношения и логические операторы Scratch, чтобы выразить неравенства, содержащие операторы ≥ и ≤.

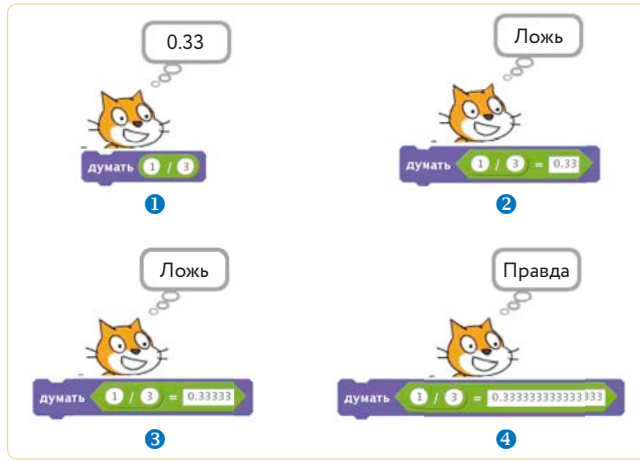
Таблица 6.9. Примеры проверки неравенств

Выражение	Реализация при использовании логических операторов
x ⩾ 10	
x ⩾ 10	
x ⩽ 10	
x ⩽ 10	
10 ⩽ x ⩽ 20	
10 ⩽ x ⩽ 20	

В этой главе мы уже рассмотрели несколько существующих в Scratch функций, в том числе сравнения, условные выражения и логические операторы. Теперь воспользуемся этим знанием и создадим парочку веселых и полезных программ.

СРАВНЕНИЕ ДЕСЯТИЧНЫХ ДРОБЕЙ

Особого внимания заслуживают ситуации, когда мы используем оператор равенства, чтобы сравнить десятичные дроби. Но способ размещения этих чисел в компьютерной памяти обусловливает тот факт, что иногда сравнение получается неточным. Рассмотрим блоки команд, показанные ниже.



Результатом деления 1 на 3 будет 0,3... с последовательностью из цифр 3, продолжающейся до бесконечности. Поскольку компьютер использует фиксированное количество места для хранения данного результата, он не может правильно сохраниться в памяти. Scratch говорит вам, что результатом деления будет 0,33 (верхняя картинка справа), но настоящий результат сохраняется внутри системы с гораздо большей точностью. И для первых двух сравнений на рисунке равенства оцениваются как неверные.

Вы можете избежать этой ошибки, применив один из предложенных ниже подходов.

- Используйте операторы **меньше чем (<)** и **больше чем (>)** вместо оператора равенства **(=)**, когда это возможно.
- Используйте блок **округлить**, чтобы округлить два числа, которые вам нужно сравнить, и потом проверяйте, равны ли округленные значения.
- Проверьте абсолютную разность между двумя значениями, которые вы сравниваете. Например, вместо того чтобы выяснять, равны ли *x* и *y*, мы можем проверить, укладывается ли абсолютная разность между ними в рамки допустимого отклонения, используя блок, аналогичный этому.

В вашем случае этот подход может оказаться достаточным.

Проекты Scratch

Новые команды, которые вы освоили в этой главе, позволят вам создать в Scratch много разнообразных полезных программ. Надеюсь, проекты, которые я предложу вам в этом разделе, станут для вас источником идей. Очень рекомендую опробовать эти программы, разобраться, как они работают, и подумать, как их можно усовершенствовать.

Угадай мои координаты

GuessMy Coordinates.sb2

Мы разработаем интерактивную игру, при помощи которой можно проверить знание декартовой системы координат. В ней есть только один спрайт — Звезда (Star), — который представляет собой случайную точку на **Сцене** (рис. 6.16).

Каждый раз, когда вы запускаете игру, спрайт перемещается в новую точку на **Сцене** и просит пользователя угадать ее координаты *x* и *y*. Игра проверяет ответы и выдает соответствующее сообщение.

Основной скрипт для спрайта-звезды показан на рис. 6.17.

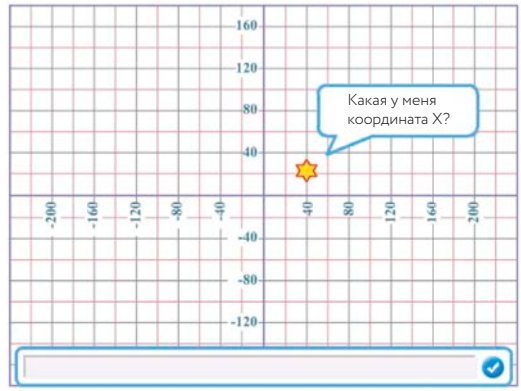


Рис. 6.16. Интерфейс игры «Угадай мои координаты»

Этот скрипт использует две переменные, X и Y, чтобы фиксировать случайные координаты спрайта. Ниже я объясню, как работает каждая из пронумерованных секций на рисунке.

1. Переменной X присваивается случайное значение из множества {–220, –200, –180, … , 220}. Случайным образом выбирается целое число между –11 и 11 и умножается на 20. Переменной Y присваивается случайное значение {–160, –140, –120, … , 160}. Выбранные значения X и Y обеспечивают нахождение нашей точки в одной из точек пересечения сетки, изображенной на рис. 6.16. Затем спрайт перемещается на позицию (*X*, *Y*).

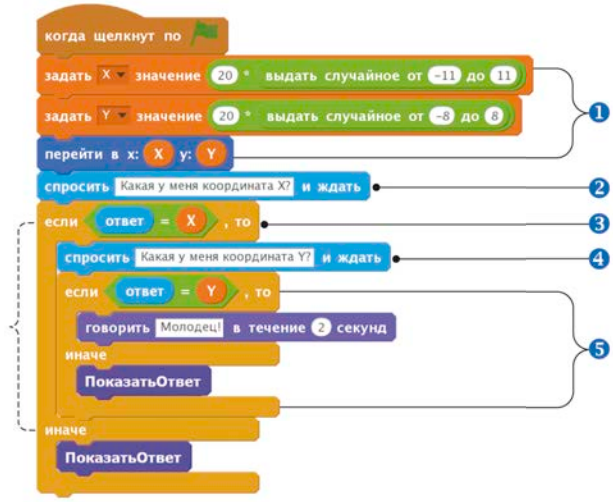


Рис. 6.17. Скрипт для игры «Угадай мои координаты»

2. Скрипт просит пользователя ввести значение координаты *x* спрайта и ждет ответа.

3. Если ответ верен, скрипт переходит к шагу 4. Если нет, он обращается к процедуре **ПоказатьОтвет**, чтобы вывести правильные координаты точки.

4. Когда пользователь вводит правильное значение координаты *x*, скрипт предлагает ему ввести значение *y* и ждет ответа.

5. Если пользователь отвечает правильно, скрипт отображает сообщение «Молодец!». Если нет, то он обращается к алгоритму **ПоказатьОтвет**, чтобы продемонстрировать координаты точки.

Процедура **ПоказатьОтвет** показана на рис. 6.18. Переменная точка сначала при помощи оператора **слить** образует текстовую строку в форме (*X*, *Y*). Затем алгоритм использует команду **сказать**, чтобы показать пользователю правильный ответ.

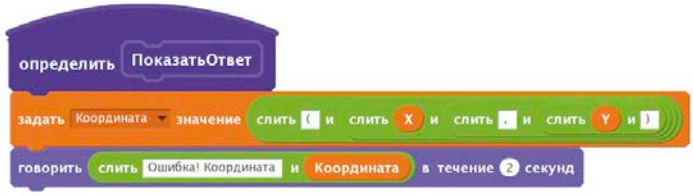


Рис. 6.18. Процедура **ПоказатьОтвет**

УПРАЖНЕНИЕ 6.1

Усовершенствуйте эту угадайку, добавив забавные детали. Например, вы можете сделать так, чтобы каждый раз, когда кто-то выигрывает, начинала играть музыка; на неправильный ответ раздавался смешной звук; игра запускалась автоматически (без клика по зеленому флажку каждый раз) или отслеживала количество правильных ответов и отображала счет.

Игра «Классификация треугольников»

Triangle Classification.sb2

Как показано на рис. 6.19, треугольники бывают разносторонними, равнобедренными и равносторонними. В этом разделе вы создадите игру, в которой пользователям нужно определять вид треугольника.

Рис. 6.19. Классификация треугольников на основании длины их сторон

Игра рисует на **Сцене** треугольник и просит пользователя отнести его к одному из трех типов. Интерфейс этой игры показан на рис. 6.20.

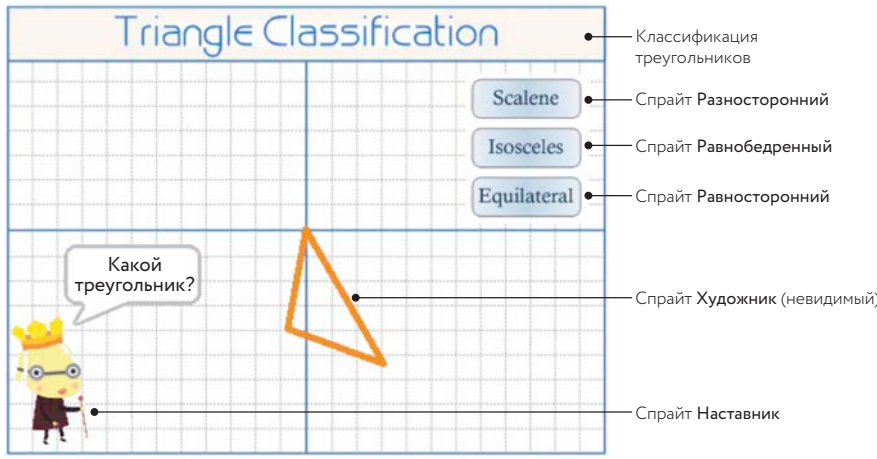


Рис. 6.20. Пользовательский интерфейс игры «Классификация треугольников»

Рис. 6.20 показывает, что в игре пять спрайтов. Три из них (Разносторонний, Равнобедренный и Равносторонний) представляют собой три кнопки, по которым пользователь кликает, чтобы выбрать ответ. Еще есть невидимый спрайт Художник, который рисует треугольник на **Сцене**.

Я сделал спрайт Художник *невидимым, убрав галочку из чекбокса* **Показать** *в окне информации о нем. Если вы предпочитаете управлять видимостью спрайта из скрипта, можете добавить блок* **спрятаться** *и скрыть спрайт в момент, когда начинается игра.*

Главный персонаж игры — спрайт Наставник. Он определяет, треугольник какого типа будет нарисован в этот раз, и проверяет ответ. Скрипты для спрайта Наставника показаны на рис. 6.21.

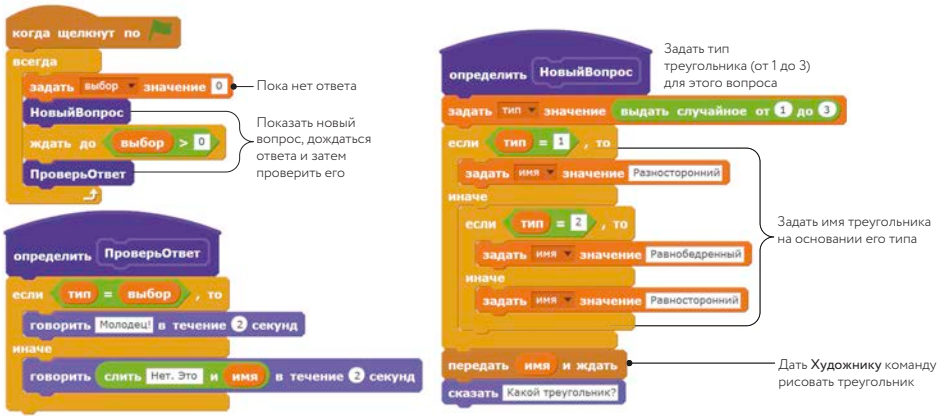


Рис. 6.21. Скрипты для спрайта Наставник. *Главный скрипт (наверху слева) обращается к процедурам* **НовыйВопрос** *(справа) и* **ПроверьОтвет** *(слева внизу)*

Когда нажат зеленый флажок и игра запущена, главный скрипт начинает свой бесконечный цикл. При каждом прохождении цикла скрипт устанавливает переменную выбор на 0 (чтобы обозначить, что игрок еще не дал ответа), рисует новый треугольник и ждет ответа. Переменная выбор должна измениться, когда пользователь кликнет по любой из трех кнопок с ответами. Когда пользователь нажимает кнопку, определяя тип треугольника, скрипт проверяет ответ и дает обратную связь. Рассмотрим каждый шаг подробнее.

Процедура **НовыйВопрос** начинает работу с того, что случайным образом устанавливает значение переменной тип (определяющей тип треугольника, который должен быть нарисован на **Сцене**) на 1, 2 или 3. Затем скрипт использует два блока **если/иначе**, чтобы задать значение переменной имя на основании значения переменной тип. Переменная имя выполняет несколько функций:

- она определяет, какое сообщение нужно отправить Художнику, чтобы тот знал, что ему рисовать (обратите внимание на то, как блок **передать и ждать** использует переменную имя);

- она используется в процедуре **ПроверьОтвет** для создания сообщения обратной связи с пользователем. Когда спрайт Художник заканчивает рисовать, процедура **НовыйВопрос** требует от пользователя ответа при помощи команды **сказать**.

Когда спрайт Художник получает сообщение, он начинает рисовать на **Сцене** соответствующий треугольник. Чтобы сделать игру более захватывающей, Художник использует случайные значения для установки размера треугольника, его положения на плоскости и цвета, как показано на рис. 6.22.

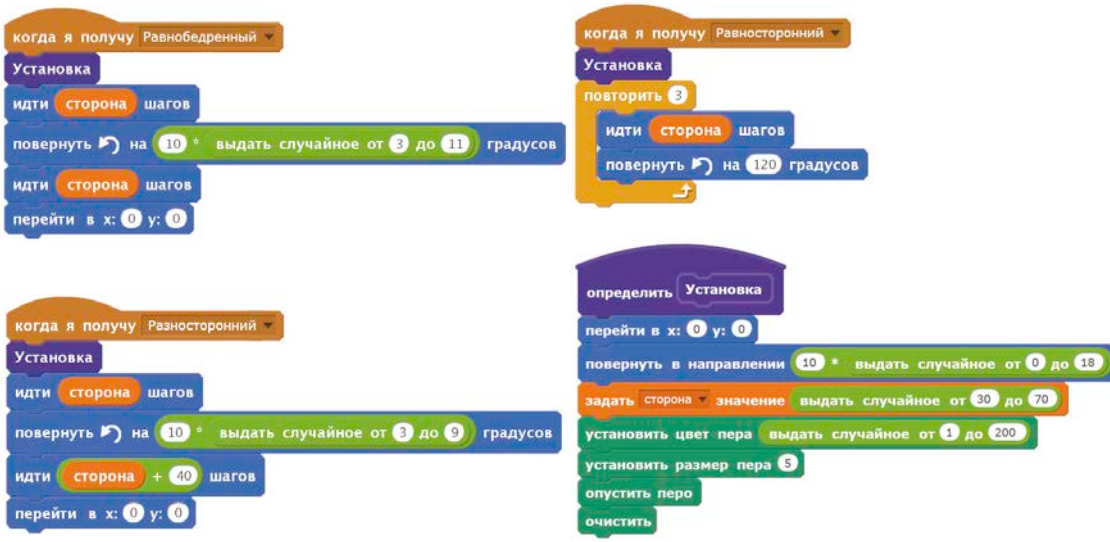


Рис. 6.22. Скрипты для спрайта Художник

Предложив пользователю определить тип нарисованного треугольника, главный скрипт переходит к блоку **ждать до** (из раздела **Управление**), чтобы сделать паузу до тех пор, пока выражение **выбор > 0** не станет верным. Три спрайта-кнопки будут менять значение переменной выбор, когда по ним будут кликать. Кнопка Разносторонний устанавливает значение переменной выбор на 1, кнопка Равнобедренный — на 2, а кнопка Равносторонний — на 3 (рис. 6.23).

Когда кнопка нажата, спрайт немного перемещается вниз и вправо, чтобы создать соответствующий визуальный эффект. Когда кнопку отпускают, спрайт возвращается в исходную позицию и устанавливает значение переменной выбор, обозначая, что пользователь нажал кнопку. Каждый спрайт устанавливает разное значение переменной. Блоки, двигающие кнопки в этих скриптах, можно убрать.

Скрипт для спрайта **Разносторонний** | Скрипт для спрайта **Равнобедренный** | Скрипт для спрайта **Равносторонний**

Рис. 6.23. Скрипты для трех спрайтов-кнопок

Когда пользователь выбрал тип треугольника, значение переменной выбор становится больше 0 и основной скрипт обращается к процедуре **ПроверьОтвет**. Этот алгоритм сравнивает значение переменной тип (которая указывает на тип нарисованного треугольника) со значением переменной выбор. Если значения этих двух переменных одинаковые, ответ пользователя верен. А если нет — ответ неверен и скрипт сообщит правильный вариант.

УПРАЖНЕНИЕ 6.2

Откройте игру и сыграйте в нее пару раз. Когда вы поймете, как она работает, попробуйте добавить дополнительные функции. Вот несколько идей.

- Сделайте так, чтобы программа вела счет. Она может добавлять по 1 очку за каждый правильный ответ и вычитать по 1 очку за каждый неправильный.
- Дайте пользователю возможность выйти из игры.
- Определите критерии для завершения игры. Например, вы можете сделать так, чтобы основной цикл повторялся не бесконечно, а 20 раз. Вы также можете заставить игру прекращаться после пяти неправильных ответов.
- Сделайте, чтобы, пока игра идет, происходило что-то интересное. Например, вы можете создать переменную под названием спецНомер и присвоить ей случайное значение в начале игры. Когда число правильных ответов совпадет с ее значением, игра присвоит дополнительные очки, заиграет музыка или даже спрайт расскажет анекдот.
- Оживите клавиши при помощи графических эффектов. Если вы добавите скрипт с рисунка внизу каждой кнопке, они будут менять цвет, когда на них наводится курсор.

Повторитель контура

LineFollower.sb2

А можем ли мы заставить спрайт следовать по определенному маршруту на **Сцене**, как показано на рис. 6.24, самостоятельно? Да. И сейчас мы напишем программу, которая будет это делать. Если вы внимательно посмотрите на спрайт на рисунке, то заметите, что мы нарисовали нос и два уха кота разными цветами. Вы также увидите увеличенное изображение головы кота.

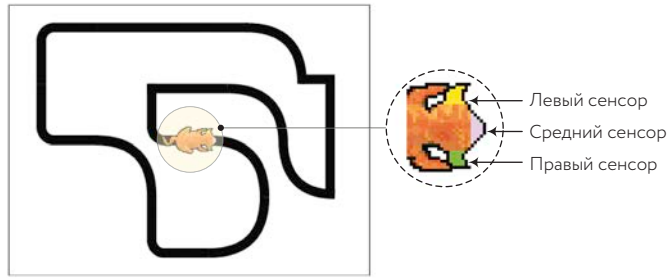


Рис. 6.24. Образец маршрута, по которому следует спрайт

План в том, чтобы использовать нос и уши кота как сенсоры цвета для определения черной линии под ними. Алгоритм для отслеживания линии использует следующие *эвристические правила* (основанные на логических заключениях и знаниях, которые получены методом проб и ошибок).

- Если нос кота (розовый цвет) касается линии, идти вперед.
- Если левое ухо кота (желтый цвет) касается линии, повернуть против часовой стрелки и идти вперед на сниженной скорости.
- Если правое ухо кота (зеленого цвета) касается линии, повернуть по часовой стрелке и идти вперед на сниженной скорости.

Естественно, точная скорость (движения) и углы поворотов могут быть разными для разных маршрутов. Их можно вычислить экспериментальным путем.

Скрипт, применяющий приведенный выше алгоритм и заставляющий спрайт следовать за линией, показан на рис. 6.25.

Скрипт на рис. 6.25 использует новый блок: **цвет касается** (из раздела **Сенсоры**). Он проверяет, соприкасается ли цвет спрайта (указан в первом цветовом квадратике) с другим цветом (во втором цветовом квадратике). Если цвет на спрайте касается другого цвета, блок возвращает значение **истина**; если нет — **ложь**. Цвет в цветовом квадратике можно выбрать, кликнув по любой точке в проекте Scratch, где есть нужный вам цвет.

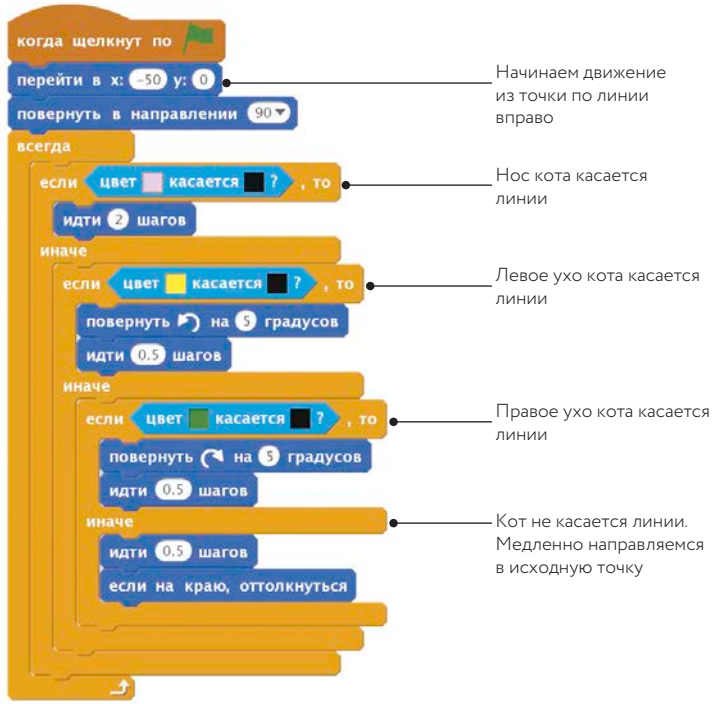


Рис. 6.25. Алгоритм следования за линией

УПРАЖНЕНИЕ 6.3

Откройте программу и запустите ее, чтобы посмотреть, как она работает. Поэкспериментируйте с заданными значениями, сделайте так, чтобы спрайт прошел маршрут максимально быстро. Один из пользователей написал нам, что прошел маршрут за 11 секунд. А вы сможете побить этот рекорд? Создайте другие маршруты и посмотрите, будет ли этот алгоритм по-прежнему работать.

Уравнение линии

EquationOf ALine.sb2

Уравнение линии, соединяющей две точки $P = (x_1, y_1)$ и $Q = (x_2, y_2)$, выглядит как $y = mx + b$, где $m = (y_2 - y_1) / (x_2 - x_1)$ — наклон линии, а b — отсекаемый отрезок на оси y. Вертикальная линия имеет уравнение $x = k$, а горизонтальная — $y = k$, где k — константа. В этом разделе мы создадим программу, которая находит решение уравнения линии, соединяющей две точки на координатной плоскости. Пользовательский интерфейс программы показан на рис. 6.26.

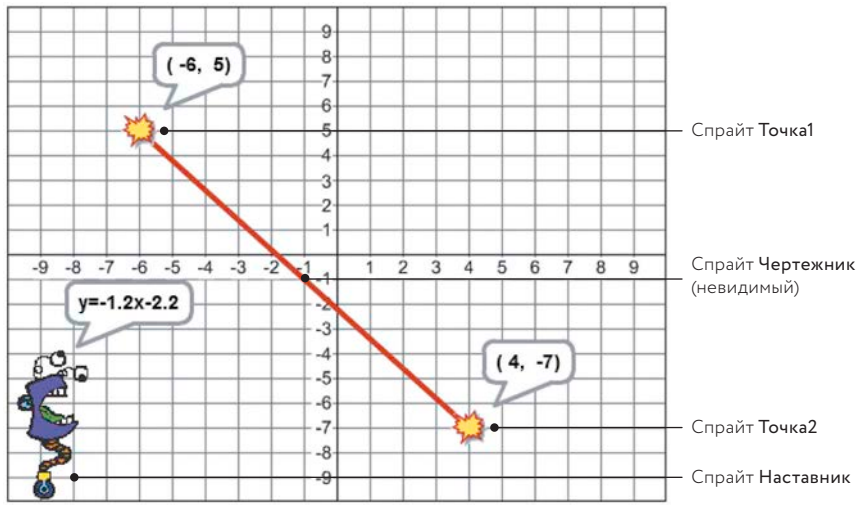


Рис. 6.26. Пользовательский интерфейс для программы, находящей решение уравнения

Пользователь перемещает два спрайта, представляющих точки на концах отрезка, по **Сцене**, и программа автоматически показывает уравнение получающейся линии. В программе задействовано четыре спрайта: Точка1 и Точка2 (Point1 и Point2) используются для обозначения точек на концах отрезка; Чертежник (Drawer) — скрытый спрайт, который рисует прямую линию между этими двумя точками, а Наставник (Tutor) отвечает за подсчеты и отображение уравнения линии.

Скрипты для спрайтов Точка1 и Точка2 очень похожи. Они содержат логику (не показанную здесь), которая сводит расположение спрайта к точкам пересечения линий системы координат. Когда пользователь передвигает спрайт Точка1, тот обновляет значение переменных, фиксирующих его координаты (назовем их X1 и Y1), и передает команду **Перерисовать**. А когда пользователь передвигает спрайт Точка2, тот обновляет значение переменных, фиксирующих его координаты (X2 и Y2), и передает такое же сообщение. Все четыре переменные (X1, X2, Y1 и Y2) могут принимать значения только целых чисел от –9 до 9. Подробнее

эти скрипты вы можете изучить в файле *EquationOfALine.sb2*. А сейчас посмотрим на скрипты спрайта Чертежник, показанные на рис. 6.27.

Рис. 6.27. Скрипты для спрайта Чертежник

Когда игра начинается, спрайт устанавливает размер и цвет своего пера и готовится чертить. Получив сообщение **Перерисовать**, он перемещается к спрайту Точка1, очищает **Сцену** и затем движется к спрайту Точка2. Результат — прямая линия, соединяющая Точку1 и Точку2.

Спрайт Наставник также выполняет свой скрипт, получив сообщение **Перерисовать**, как показано на рис. 6.28.

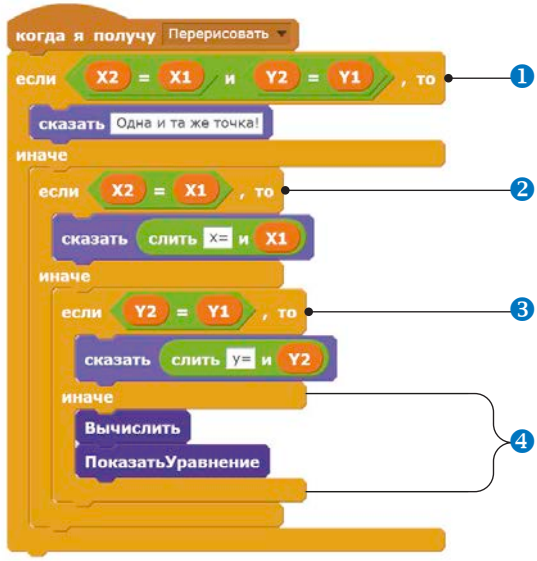


Рис. 6.28. Обработчик сообщения **Перерисовать** *для спрайта* Наставник

Скрипт осуществляет следующие проверки.

- Если координаты Точки1 и Точки2 одинаковые, линию прочертить невозможно. Скрипт говорит: «Одна и та же точка».

- Если точки разные, но значение их координаты *x* одинаковое, получится вертикальная линия. Скрипт отобразит уравнение **x = constant**.

- Если точки разные, но значение их координаты *y* одинаковое, получится горизонтальная линия. Скрипт отобразит уравнение **y = constant**.

- И наконец, если точки образуют прямую, уравнение которой имеет форму $y = mx + b$, сначала скрипт обращается к процедуре **Вычислить**, чтобы найти для этой линии угол наклона и отсекаемый отрезок на оси *y*. Затем он обращается к процедуре **ПоказатьУравнение**, чтобы придать уравнению должный формат и показать его пользователю.

Процедура **Вычислить** показана на рис. 6.29. Она вычисляет угол наклона (*m*) и отсекаемый отрезок на оси *y* (*b*), а потом округляет эти величины до сотых долей.

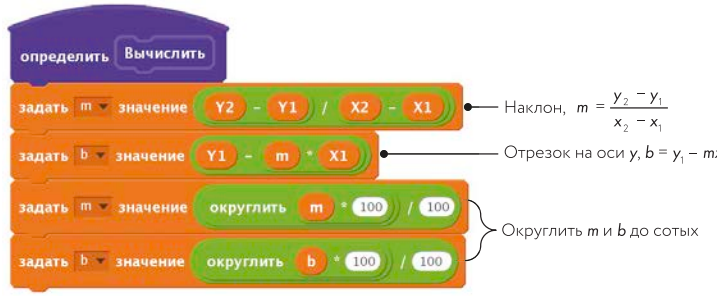


Рис. 6.29. Процедура **Вычислить**

Процедура **ПоказатьУравнение** изображена на рис. 6.30. Она использует две переменные (параметр1 и параметр2) и два субалгоритма, чтобы отобразить уравнение в нужном виде.

Процедура **ПоказатьУравнение** при форматировании уравнения принимает в расчет следующие особые случаи.

- Если наклон 1, значение переменной параметр1 будет *x* (вместо 1*x*).

- Если наклон –1, значение переменной параметр1 будет *–x* (вместо –1*x*).

- Переменная параметр2 формируется при помощи соответствующего знака (плюс или минус) координаты отрезка на оси *y*.

- Если координата отрезка на оси $y = 0$, уравнение будет иметь вид $y = mx$.

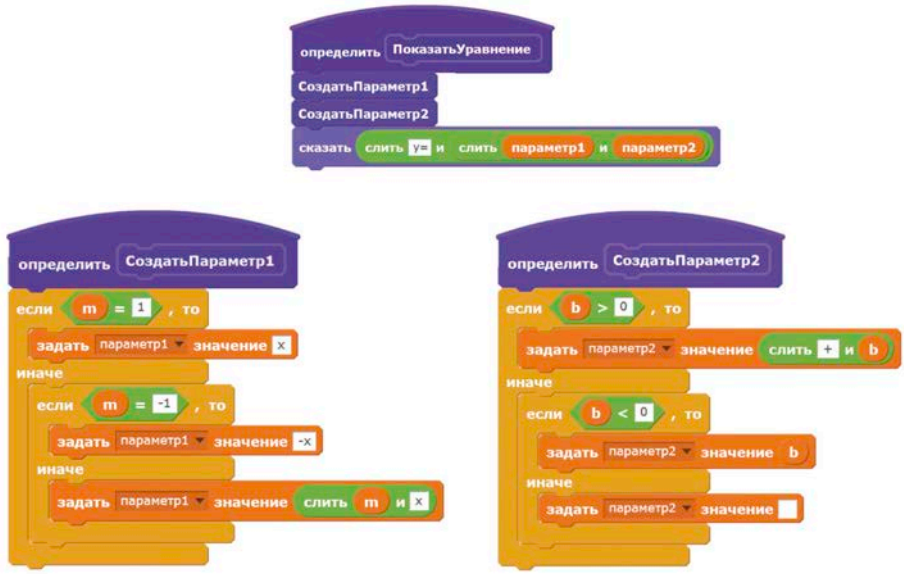


Рис. 6.30. Процедура **ПоказатьУравнение**

Другие программы

Теперь обсудим еще пару игр, которые вы найдете в разделе дополнительных материалов к книге (скачать их можно с http://nostarch.com/learnscratch/). Среди дополнительных материалов есть две классические игры, с которыми вам предстоит разобраться самостоятельно. Первая из них — «Угадай мой номер». Программа тайно выбирает целое число от 1 до 100 и предлагает игроку угадать его. Затем она сообщает игроку, больше или меньше предложенный им вариант правильного ответа, выводя надписи «слишком много» или «слишком мало». У игрока шесть попыток. Если он угадает — он победил, если ошибется — проиграл.

Guess MyNumber.sb2

Вторая игра — «Камень, ножницы, бумага» против компьютера. Пользователь выбирает, кликая по одной из трех кнопок, на которых изображены камень, ножницы и бумага. Компьютер делает случайную выборку. Победителя выбирают на основе следующих правил: бумага бьет (заворачивает) камень, камень бьет (ломает) ножницы, а ножницы бьют (режут) бумагу.

RockPaper.sb2

УПРАЖНЕНИЕ 6.4

Откройте программу и запустите ее. Перетащите две точки в разные места на **Сцене** и проверьте получившееся уравнение. Чтобы усложнить программу, добавьте скрипт, который убирает спрайт Наставник с пути, если он перекрывает координаты, отображаемые спрайтами Точка1 и Точка2.

Итоги

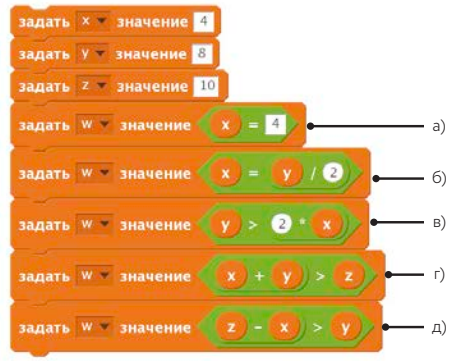
Из этой главы вы узнали об операторах сравнения в Scratch и использовали их для сопоставления чисел, букв и строк. Потом вы познакомились с блоками **если** и **если/иначе** и использовали их для принятия решений и управления действиями в нескольких программах. Вы также научились пользоваться вложенными блоками **если** и **если/иначе** для проверки множественных условий и применяли эту технику для разработки программы, управляемой меню. Вы также узнали о логических операторах как альтернативном и более точном способе проверки множественных условий.

И наконец, вы рассмотрели несколько готовых программ, демонстрирующих структуры принятия решений в действии.

В следующей главе мы глубже изучим раздел **Управление**. Я покажу вам различные структуры повторения, доступные в среде Scratch, а также научу вас их использовать для написания еще более сложных программ.

Задания

1. Каково значение *W* после выполнения каждой из команд этого скрипта?



2. Выразите каждое из следующих условий при помощи блока **если**:
   а) если *x* разделить на *y* будет 5, то задать *x* значение 100;
   б) если *x* умножить на *y* будет 5, то задать *x* значение 1;
   в) если *x* меньше *y*, то удвоить значение *x*;
   г) если *x* больше *y*, то увеличить значение *x* на 1.

3. Напишите программу, которая просит пользователя назвать пять чисел от 1 до 10. Затем она должна сосчитать количество чисел, которые больше 7.

4. Выразите каждое из следующих условий при помощи блока **если/иначе**:
   а) если *x* умножить на *y* будет 8, то задать *x* значение 1; иначе задать *x* значение 2;
   б) если *x* меньше *y*, то увеличить вдвое значение *x*, иначе увеличить *x* на 1;
   в) если *x* больше *y*, то увеличить обе переменные на 1, иначе уменьшить обе на 1.

5. Примените скрипт справа к каждому из следующих случаев, чтобы найти результаты:
   а) $x = -1$, $y = -1$, $z = -1$;
   б) $x = 1$, $y = 1$, $z = 0$;
   в) $x = 1$, $y = -1$, $z = 1$;
   г) $x = 1$, $y = -1$, $z = -1$.

6. Напишите программу, которая просит пользователей ввести три числа. Затем программа определит и напечатает самое большое из этих чисел.

7. Компания продает пять наименований продукции по розничным ценам, указанным в этой таблице. Напишите программу, которая просила бы пользователя ввести номер товара и сколько его была продано, а затем подсчитывала и показывала бы полную розничную стоимость.

Номер продукта	**Розничная цена, рублей**
1	29,5
2	49,9
3	54,9
4	78
5	88,5

8. Составьте логическое выражение, которое отображает каждое из этих условий:
   а) счет больше 90 и меньше 95;
   б) ответ либо *y*, либо «да»;
   в) ответ — четное число от 1 до 10;
   г) ответ — нечетное число от 1 до 10;
   д) ответ от 1 до 5, но не равен 4;
   е) ответ от 1 до 100 и делится на 3.

9. *Теорема неравенства треугольников* гласит, что сумма длин любых двух сторон треугольника больше его третьей стороны. Напишите программу, которая получает от пользователя три числа и определяет, могут ли они быть длинами сторон треугольника.

10. *Теорема Пифагора* утверждает, что если *a* и *b* — длины катетов прямоугольного треугольника, а *c* — длина гипотенузы, то $a^2 + b^2 = c^2$. Напишите программу, которая получает от пользователя три числа и определяет, могут ли они быть длинами сторон прямоугольного треугольника.

7

ПОВТОРЕНИЕ: ПОДРОБНЕЕ О ЦИКЛАХ

В этой главе мы подробнее разберем знакомые структуры повторения в Scratch. Поговорим о новых блоках, с помощью которых можно создавать циклы, вложенные циклы и рекурсию. К концу главы вы разберетесь в следующих темах:

- структуры повторения, обеспечивающие многократное выполнение команд;
- проверка правильности информации, введенной пользователем;
- циклы со счетчиком или циклы, управляемые событиями;
- процедуры, которые могут запускаться многократно.

Главное отличие компьютеров от живых людей в том, что машины любят постоянно выполнять повторяющиеся задачи. *Структуры повторения*, или *циклы*, — это команды, заставляющие компьютер многократно выполнять действие или последовательность действий. Самый простой тип — *явный цикл*, который повторяет последовательность команд определенное число раз. Его еще называют *циклом со счетчиком* или с *подсчетом*. Другие типы циклов повторяют последовательность действий, пока не будут соблюдены определенные условия. Это *циклы, управляемые условиями*. Существуют также *бесконечные циклы*. Из этой главы вы узнаете о разных структурах повторения, доступных в Scratch. Я подробно расскажу и о циклах со счетчиками, и о циклах, управляемых условиями,

а также познакомлю вас с блоком **стоп**, который можно использовать, чтобы завершить бесконечный цикл. Вы узнаете, как использовать циклы для проверки правильности информации, введенной пользователем. Кроме того, мы рассмотрим *вложенные циклы* (содержащие другие циклы) и разберем примеры их использования. Мы поговорим о еще одном способе реализации повторения: *рекурсии* — процедуре, которая вызывает сама себя. Вдобавок мы разработаем несколько интересных программ, в которых используются и циклы, и условные конструкции, а также посмотрим, как все это выглядит на практике.

Больше блоков-циклов в Scratch

Как вы уже знаете из главы 2, блоки-циклы дают возможность повторить в программе команду или набор программ. Scratch поддерживает три блока повторения (рис. 7.1).

Рис. 7.1. Блоки повторения в Scratch

Вы уже не раз пользовались двумя из этих блоков: **повторить** и **всегда**. В этом разделе вы познакомитесь с третьим — блоком **повторять пока не** — и разберетесь в технических моментах, связанных с циклами в целом. Каждый повтор цикла называется *итерацией*, а словом *«счет»* я описываю количество повторов цикла. Блок **повторить** — цикл со счетчиком: он повторяет свои команды определенное число раз. Это лучший вариант, когда мы знаем, сколько нужно повторений: например, если мы хотим нарисовать многоугольник с известным числом сторон. А блок **повторять пока не** — цикл, управляемый условием. Его команды выполняются, если верно проверяемое условие. Мы используем этот метод, когда не знаем заранее, сколько раз нужно повторить цикл, и хотим, чтобы алгоритм выполнялся до тех пор, пока не будет выполнено некое условие. Например, вы можете сказать: «Повторять команду **спросить** до тех пор, пока пользователь не введет положительное число». Или «Повторять пуск ракет до тех пор, пока уровень энергии игрока не опустится ниже определенного показателя». В следующем разделе мы подробнее рассмотрим циклы, управляемые условиями.

Блок повторять пока не

Представим себе, что вы разрабатываете игру, в которой перед пользователем ставится элементарная математическая задача. Если он отвечает неправильно, игра снова задает тот же вопрос, чтобы дать ему еще один

шанс. Иными словами, игра задает один и тот же вопрос *до тех пор,* пока пользователь не ответит правильно. Очевидно, блок **повторить** здесь не подходит: вы не знаете, сколько попыток понадобится игроку, чтобы ответить правильно. Может, повезет прямо с первой попытки, а может, придется попробовать 100 раз. В таких случаях поможет блок **повторять пока не**. Структура с ним показана на рис. 7.2.

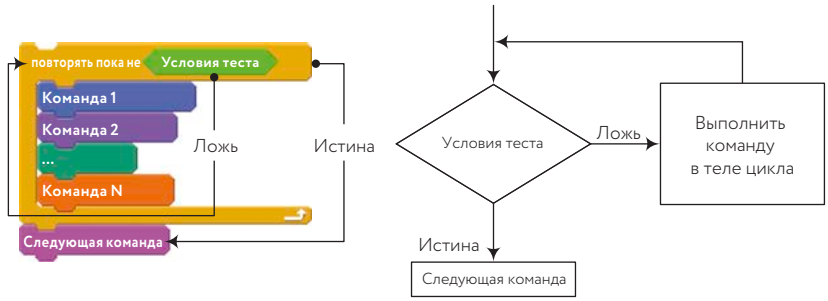


Рис. 7.2. Блок **повторять пока не** *дает возможность выполнять серию команд до тех пор, пока не будет соблюдено некое условие*

Блок содержит булево выражение, значение которого проверяется в начале цикла. Если оно верно, команды внутри цикла будут выполняться. Когда последняя команда цикла выполнена, он запускается заново и выражение проверяется снова. Если выражение неверно, команды внутри цикла будут выполнены еще раз. Цикл будет продолжаться до тех пор, *пока* тестовое выражение не станет верным. Тогда команды внутри цикла будут приостановлены и программа сразу перейдет к команде, следующей за циклом.

Если проверяемое условие окажется верным перед первым запуском цикла, его команды не будут выполняться вообще.

Блок **повторять пока не** остановится, только если команда (либо внутри цикла, либо из другой активной части программы) сделает условие верным. Если оно не может стать верным никогда, мы получим бесконечный цикл. Рис. 7.3 демонстрирует пример использования на практике блока **повторять пока не**. В этом примере до тех пор, пока спрайт Игрок (Player) находится на расстоянии более 100 шагов от спрайта Страж (Guard), последний будет двигаться в том направлении, в котором он движется (в данном случае по горизонтали), при касании отталкиваясь от правого и левого края **Сцены**. Если расстояние между двумя спрайтами становится меньше 100 шагов, блок **повторять пока не** прекратит работу и Страж начнет погоню за Игроком. Код для этой погони на рисунке не показан. Блок **расстояние до** вы найдете в разделе **Сенсоры**.

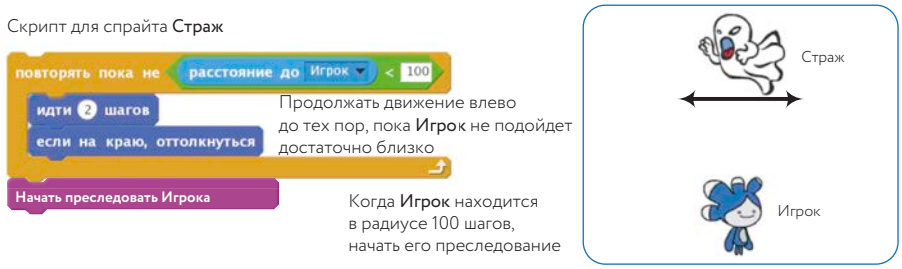


Рис. 7.3. Простой образец использования на практике блока **повторять пока не**

УПРАЖНЕНИЕ 7.1

Chase.sb2

Откройте программу *Chase.sb2* и запустите ее. При помощи клавиш со стрелками подведите Игрока ближе к Стражу, чтобы посмотреть погоню. Как бы вы изменили проверяемое условие, чтобы Страж реагировал не на приближение Игрока, а на его выход за пределы определенной области (например, от –50 до 50)? Внесите эти изменения в программу, чтобы проверить правильность вашего решения.

Создание блока всегда если

Бесконечные циклы полезны во многих программах. Например, в предыдущих главах мы использовали блок **всегда** для проигрывания фоновой музыки и анимации спрайтов путем постоянной смены их костюмов.

Блок **всегда** — безусловный бесконечный цикл. Он не имеет проверяемого условия, которое контролирует выполнение команд внутри него. Вы легко можете изменить это, вложив блок **если** внутрь блока **всегда**, чтобы создать *условный бесконечный цикл* (рис. 7.4).

Условие блока **если** тестируется в начале каждой итерации, и его команды выполняются только в случае, если оно верно. Блок **всегда** предполагает бесконечное выполнение команд без возможности добавления после него других блоков.

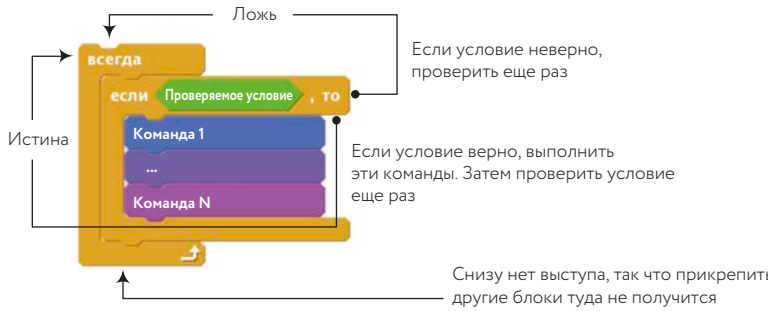


Рис. 7.4. Вы можете создать цикл **всегда/если**, *соединив блок* **всегда** *с блоком* **если**

Комбинированная структура **всегда/если** часто используется для управления движением спрайтов при помощи клавиш со стрелками, как показано на рис. 7.5.

Рис. 7.5. Эти скрипты позволяют вам перемещать спрайт при помощи клавиш со стрелками. Каждый скрипт реагирует на одну из четырех клавиш

Когда пользователь щелкнет по зеленому флажку, четыре клавиши со стрелками (влево, вправо, вверх и вниз) будут отслеживаться в четырех независимых бесконечных циклах. Когда нажимается одна из клавиш, соответствующий цикл меняет значение координаты *x* или *y* спрайта. ArrowKeys1.sb2

Создайте в Scratch эти скрипты (или откройте файл *ArrowKeys1.sb2*) и запустите программу. Когда вы нажимаете клавиши со стрелками вверх и вниз одновременно, спрайт перемещается по диагонали на северо-восток. Попробуйте другие комбинации клавиш со стрелками, чтобы посмотреть, как будет реагировать программа.

УПРАЖНЕНИЕ 7.2

Рассмотрим еще один способ управления движением спрайта при помощи клавиш со стрелками. Сравните этот метод с тем, что показан на рис. 7.5. Какой из них более чувствителен к нажатиям клавиш? Как альтернативный скрипт ведет себя, если вы нажмете сразу две клавиши (например, вниз и направо)? Попробуйте поместить четыре блока **если**, показанных на рис. 7.5, в один цикл **всегда** и нажмите одновременно две клавиши со стрелками. Как изменится поведение спрайта?

Стоп-команды

Представьте, что вы пишете команду для поиска первого целого числа меньше 1000, которое без остатка делилось бы на 3, 5 и 7. Вы можете создать скрипт, который будет одно за другим проверять числа 999, 998, 997 и т. д. При этом нужно остановить поиск после того, как вы найдете подходящее число (в данном случае 945).

Как сообщить Scratch, что цикл нужно прервать и остановить скрипт? Для остановки скриптов можно использовать команду **стоп** (из раздела **Управление**). Выпадающее меню предлагает три варианта (рис. 7.6).

Рис. 7.6. Использование команды **стоп** *в Scratch*

Первый вариант незамедлительно останавливает обратившийся к ней скрипт. Второй останавливает все работающие скрипты вашей программы. Он равносилен красной кнопке «Стоп» наверху **Сцены**.

Вы не можете прикрепить никакие другие команды после блока **стоп**, если вы использовали один из этих вариантов.

StopDemo.sb2

Третий вариант позволяет спрайту или **Сцене** остановить все свои скрипты, за исключением того, который обратился к команде **стоп**. К этому блоку вы можете добавить другие блоки снизу, чтобы они были выполнены после того, как спрайт приостановит работу других скриптов. Посмотрим на эту команду в действии на примере простой игры, показанной на рис. 7.7.

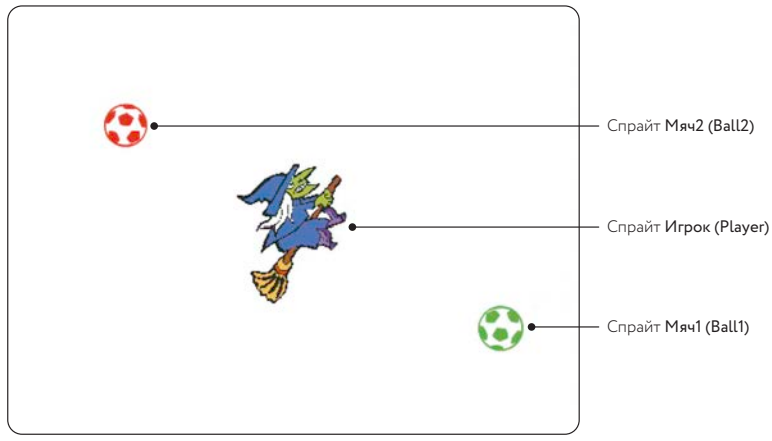


Рис. 7.7. В этой игре игрок перемещает ведьму по **Сцене**, *пытаясь увернуться от двух мячей*

Два мяча перемещаются по **Сцене** и преследуют ведьму. Игрок перемещает спрайт Ведьма при помощи клавиатуры и пытается избежать столкновений с двумя мячами. Если красный мяч в любой момент коснется Игрока, игра закончится. Если зеленый мяч коснется игрока, он перестанет преследовать его, но красный мяч будет двигаться быстрее — и убегать от него станет труднее.

Скрипты для перемещения спрайта Ведьма те же, что и на рис. 7.5, так что я не буду показывать их здесь. Скрипты для двух мячей показаны на рис. 7.8 — изучим их внимательнее.

Рис. 7.8. Скрипты зеленого мяча (слева) и красного мяча (справа)

Когда зеленый мяч касается Игрока, он увеличивает переменную скорость (обозначает скорость движения красного мяча) и запускает команду **стоп этот скрипт**, чтобы остановить свой скрипт. Все другие скрипты работают нормально. Здесь оптимальна команда **стоп этот скрипт**: мы хотим всего лишь однократно ускорить красный мяч. А когда игрока касается красный мяч, он запускает команду **стоп все**, которая останавливает все работающие скрипты программы.

УПРАЖНЕНИЕ 7.3

Загрузите игру и сыграйте в нее, чтобы понять, как она работает. Посмотрите, что происходит с желтой рамкой вокруг двух скриптов с рис. 7.8, когда зеленый и красный мячи касаются Игрока.

Вы можете использовать команду **стоп**, чтобы остановить процедуру или заставить ее с любого места в процессе выполнения вернуться к вызывающей команде. Следующий раздел демонстрирует этот принцип в действии.

Завершение вычислительного цикла

NumberSearch.sb2

Предположим, мы хотим найти первую степень числа 2, значение которой превышает 1000. Мы напишем процедуру, которая будет циклично проверять значение 2^1, 2^2, 2^3, 2^4 и т. д. Когда мы найдем нужное число, мы хотим, чтобы программа сообщила нам ответ и остановила процедуру. Рис. 7.9 демонстрирует два пути реализации этого подхода.

Рис. 7.9. Два способа нахождения первой степени 2, которая больше 1000

Процедура слева на рис. 7.9 задает переменной результат значение 2 (первая степень двойки, которую нужно проверить) и запускает бесконечный цикл в поисках ответа. Она проверяет значения переменной при каждой итерации цикла. Если результат больше 1000, процедура запускает команду **стоп этот скрипт**, чтобы остановить скрипт и вернуть управление команде, вызвавшей его. Иначе будет выполнена команда, следующая после блока **если** (она умножает предыдущее значение переменной результат на 2), и начнется новая итерация. В первой итерации блок **если** задает переменной значение 2, во второй — 4, в третьей — 8 и т. д. Это продолжается до тех пор, пока значение переменной не превысит 1000. В этот момент процедура остановится и вернется к вызывающей его команде, которая отображает значение переменной результат при помощи блока **сказать**.

Рисунок 7.9 (справа) демонстрирует другой способ применения процедуры.

Здесь мы использовали команду **повторять пока не**, которая прокручивает цикл до тех пор, пока значение переменной результат не станет больше 1000. Как и в первом варианте, цикл продолжает увеличивать вдвое значение переменной до тех пор, пока оно не достигнет 1000. Когда это произойдет, цикл остановится самостоятельно и процедура вернется к вызывающей ее команде. В этом случае нам не пришлось пользоваться командой **стоп**.

Команда **стоп** может пригодиться в ситуации, когда вам нужно проверить достоверность данных, введенных пользователем. Далее мы рассмотрим этот подход на практике.

Проверка данных, введенных пользователем

Когда вы пишете программу, которая получает данные от пользователя, всегда следует проверять правильность введенной информации до того, как начать ее обработку. Структуры повторения помогут вам в этом. Если введенные пользователем данные неверны, вы можете использовать цикл, чтобы вывести сообщения об ошибке и попросить пользователя ввести данные повторно.

Предположим, вы создаете игру с двумя уровнями и хотите дать игроку выбрать один из них. Единственными применимыми ответами в данном случае будут цифры 1 и 2. Если пользователь введет другую цифру, нужно предложить ему ввести приемлемый ответ. Один из способов такой проверки показан на рис. 7.10.

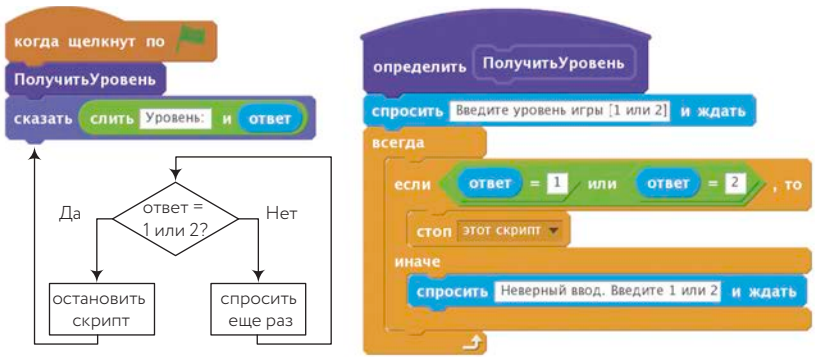


Рис. 7.10. Проверка данных, введенных пользователем при помощи блока **всегда**

Процедура **ПолучитьУровень** просит пользователя ответить еще раз и проверяет ответ внутри цикла **всегда**. Если он неприемлем, цикл просит пользователя ввести новый ответ. Если ответ корректен, процедура запустит команду **стоп этот скрипт**, чтобы остановить цикл и закончить процедуру. Тогда основной скрипт, ожидающий результатов процедуры **ПолучитьУровень**, переходит к выполнению команды **сказать**. На рис. 7.11 показано, как достичь того же результата при помощи команды **повторять пока не**.

Процедура на рис. 7.11 просит пользователя ввести ответ и ждет. Если тот вводит 1 или 2, условие в шапке блока **повторять пока не** оказывается верным, цикл останавливается, процедура завершается. А если пользователь вводит любой другой символ, условие цикла оценивается как ложь, выполняется команда **спросить** внутри цикла. Она снова ждет

ввода данных, а команда **повторять пока не** будет просить ответа до тех пор, пока пользователь не введет корректное значение. Этот вариант также не требует использования команды **стоп**.

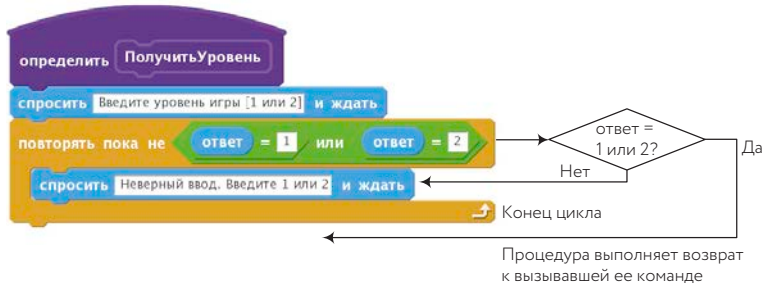


Рис. 7.11. Проверка данных, введенных пользователем, при помощи команды **повторять пока не**

Функции счета

Иногда нужно отследить, сколько итераций было совершено циклом. Например, если вы хотите дать пользователю только три попытки, чтобы ввести правильный пароль, нужно считать эти попытки и заблокировать ввод после третьей. Вы можете решать подобные задачи программирования при помощи переменной (ее принято называть *переменной цикла*), которая считает количество итераций.

Рассмотрим пару примеров, которые демонстрируют практическое применение функций счета.

Проверка пароля

PasswordCheck.sb2

Программа на рис. 7.12 просит пользователя ввести пароль, чтобы разблокировать ноутбук. У спрайта Ноутбук (Laptop) два костюма: картинка с надписью off говорит о том, что ноутбук заблокирован, а картинка с надписью on — что он разблокирован. Пользователь не получит доступа к ноутбуку, если неправильный пароль будет введен три раза.

При нажатии на зеленый флажок спрайт Ноутбук переключается на костюм off и запускает процедуру **ПолучитьПароль** для авторизации пользователя. Ожидается, что процедура вернет результат проверки пароля в основной скрипт при помощи флага **ПолучитьПароль**. После этого блок **если/иначе** проверяет флаг **ПолучитьПароль**, чтобы определить, нужно ли предоставить пользователю доступ к системе. Если **ПолучитьПароль** было присвоено значение 1, то есть пользователь ввел правильный пароль, то блок **если** выполняет команду **говорить,** которая отображает сообщение «Доступ разрешен» и меняет костюм спрайта Ноутбук на on. Иначе скрипт выводит сообщение «Доступ запрещен!», а спрайт остается в исходном костюме off.

Рис. 7.12. Этот скрипт дает пользователю три попытки для ввода правильного пароля

Процедура **ПолучитьПароль** задает для флага ПолучПароль значение 0, чтобы показать, что она пока не получила правильного ответа, и устанавливает для переменной ОшибкаСчета (цикловой переменной) значение 0. Затем она выполняет цикл **повторить** с максимальным количеством повторов 3. На каждой итерации цикла пользователю предлагается ввести пароль. Если пользователь вводит пароль правильно («Пароль123» для данного примера), флаг ПолучПароль меняет значение на 1, процедура автоматически останавливается путем запуска команды **стоп этот скрипт**, а затем возвращается к вызывающей ее команде.

Если пользователь еще не израсходовал все попытки, выводится сообщение об ошибке и дается еще один шанс. Если он ошибается трижды подряд, цикл **повторить** автоматически останавливается, процедура возвращается к вызывающей ее команде, а значение переменной ПолучПароль по-прежнему остается 0.

УПРАЖНЕНИЕ 7.4

Откройте программу и запустите ее. Что произойдет, если вы введете «пароЛЬ123» (вместо «Пароль123») в качестве пароля? Что это говорит вам о сравнении строк в Scratch? Попробуйте внедрить процедуру **ПолучитьПароль** при использовании блока **повторять пока не**.

Счет с постоянной величиной шага

CountingByConstAmount.sb2

Переменные внутри цикла можно изменять не только на 1. Скрипт на рис. 7.13 под номером ❶, например, заставляет спрайт считать от 5 до 55, прибавляя по 5. Скрипт под номером ❷ дает спрайту команду считать в обратном порядке от 99 до 0, вычитая по 11: 99, 88, 77, ... , 11, 0.

Рис. 7.13. Вы можете увеличивать или уменьшать цикловую переменную с шагом, отличным от 1

Чтобы увидеть практическое применение этой техники, предположим, что нам нужно найти сумму всех четных чисел от 2 до 20 включительно (2 + 4 + 6 + 8 + ... + 20). Именно это и делает скрипт на рис. 7.14.

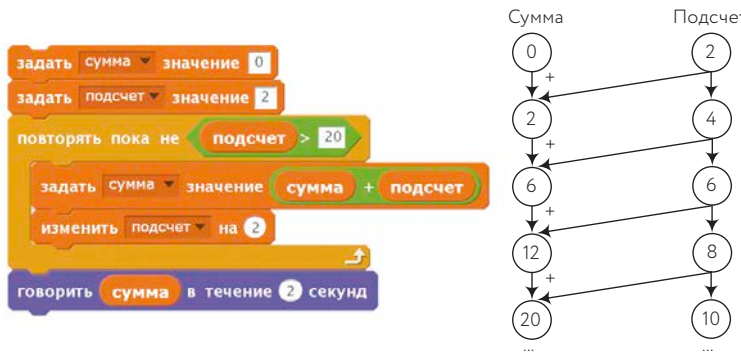


Рис. 7.14. Этот скрипт находит сумму всех четных чисел от 2 до 20

Этот скрипт сначала инициализирует переменную сумма, присваивая ей значение 0, а переменной подсчет — 2, а затем запускает условный цикл, который повторяется, пока значение переменной подсчет не превысит 20. При каждой итерации к текущей сумме прибавляется значение переменной подсчет, и та увеличивается на 2, чтобы получилось следующее четное целое число в последовательности. Предскажите результат работы этого скрипта, а затем запустите его, чтобы проверить свой ответ.

СЧЕТ С НЕЦЕЛЫМ ЧИСЛОМ ПОВТОРОВ

Non-Integer RepeatCount.sb2

Как вы думаете, что будет, если попросить Scratch повторить цикл 2,5 раза? Три примера, приведенных ниже, показывают, как Scratch решает задачи с нецелым числом повторов.

Конечно, на самом деле повторить что-то 2,5 раза невозможно, но Scratch позволяет вам вводить такие значения. Вместо того чтобы выдать сообщение об ошибке, он автоматически округляет десятичную дробь до ближайшего целого числа.

Снова о вложенных циклах

В главе 2 мы уже использовали вложенные циклы для рисования вращающихся квадратов. Один цикл (*внутренний*) отвечал за рисование квадратов, а второй (*внешний*) контролировал число поворотов.

В этом разделе вы научитесь использовать цикловые переменные совместно с вложенными циклами, чтобы создавать итерации в двух (и более) измерениях.

Эта техника очень важна в программировании. Она может использоваться для решения широкого спектра задач. Представим себе, что в соседнем ресторанчике в меню четыре сорта пиццы (P1, P2, P3 и P4) и три салата (S1, S2 и S3). Если бы вы решили поесть там, вы бы оказались перед выбором из 12 возможных комбинаций. Вы можете заказать пиццу P1 с любым из трех салатов, P2 с любым из трех салатов и т. д. Владелец хочет напечатать меню, в котором перечислены все возможные комбинации вместе с ценами и числом калорий. Посмотрим, как вложенные циклы могут быть использованы для решения этой задачи. (Подсчет цен и калорий я оставлю вам в качестве задания.)

Здесь нам нужно всего лишь два цикла: один (внешний) для всех сортов пиццы, а другой (внутренний) — для всех салатов. Внешний цикл начинается с P1, а внутренний подставляет S1, S2 и S3. Затем внешний цикл переходит к P2, а внутренний снова выбирает между S1, S2 или S3. Это продолжается до тех пор, пока внешний цикл не пройдет все четыре сорта пиццы. Применение этой модели показано на рис. 7.15.

NestedLoops1.sb2

Рис. 7.15. Визуализация вложенных циклов. Переменная P *контролирует внешний цикл, а* S *— внутренний*

В скрипте используются два цикла и две переменные. Переменная для внешнего цикла называется P, а для внутреннего — S. В первой итерации внешнего цикла (P = 1) значение переменной S задано на 1, а внутренний цикл повторяется три раза. Каждый раз он выполняет команду **сказать**, чтобы вывести актуальные значения P и S, а потом увеличивает S на 1. Таким образом, первая итерация внешнего цикла заставляет спрайт сказать «P1, S1», «P1, S2» и «P1, S3».

Когда внутренний цикл останавливается после трех итераций, P увеличивается на 1 и начинается вторая итерация внешнего цикла. Значение S задается равным 1, а внутренний цикл снова запускается. Спрайт говорит «P2, S1», «P2, S2» и «P2, S3». Процесс продолжается аналогично, заставляя спрайт сказать «P3, S1», «P3, S2» и «P3, S3», а потом наконец «P4, S1», «P4, S2» и «P4, S3», прежде чем скрипт завершится. Проследите за работой скрипта, чтобы понять, как он устроен.

Теперь, когда вы увидели, чем могут быть полезны вложенные циклы, применим эту технику для решения одной интересной математической задачи. Мы хотим написать программу, которая нашла бы три таких целых положительных числа n_1, n_2 и n_3, чтобы $n_1 + n_2 + n_3 = 25$ и $(n_1)^2 + (n_2)^2 + (n_3)^2 = 243$. Поскольку компьютеры хорошо решают повторяющиеся задачи, мы перепробуем все возможные комбинации чисел (эта техника называется *методом перебора*), дав компьютеру потрудиться.

Исходя из нашего первого уравнения первое число, n_1, может иметь любое значение от 1 до 23, так как нам нужно будет прибавить к нему два числа, чтобы получить 25. (Вы, должно быть, заметили, что n_1 не может быть больше 15, так как $16^2 = 256$, что больше 243. Но мы пока проигнорируем второе уравнение и установим верхнюю границу цикла на 23.) Второе число, n_2, может быть любым от 1 до $24 - n_1$. Например, если n_1 будет 10, максимально возможное значение n_2 — 14, потому что n_3 должно быть не меньше 1. Зная n_1 и n_2, мы можем получить n_3: $25 - (n_1 + n_2)$. Затем нам нужно проверить, действительно ли

сумма квадратов всех этих трех чисел составляет 243. Если это так, то задача выполнена. Если нет, придется попробовать другую комбинацию n_1 и n_2. Вы найдете законченный скрипт для нахождения n_1, n_2 и n_3 на рис. 7.16.

NestedLoops2.sb2

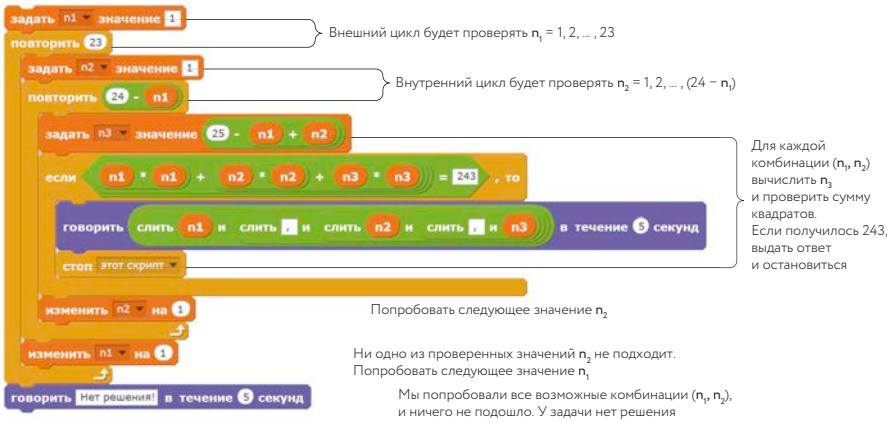


Рис. 7.16. Этот скрипт ищет три положительных числа, сумма которых равна 25 и сумма квадратов которых равна 243

Внешний цикл проверяет все значения n_1 от 1 до 23. Для каждого значения n_1 внутренний цикл проверяет все значения n_2 от 1 до $(24 - n_1)$. Для каждой комбинации n_1 и n_2 скрипт задает значение n_3 равным $25 - (n_1 + n_2)$, а затем проверяет, равна ли сумма квадратов этих чисел 243. Если да, скрипт выдает ответ и прекращает работу.

УПРАЖНЕНИЕ 7.5

Создайте скрипт, показанный на рис. 7.16, и запустите его, чтобы найти значения n_1, n_2 и n_3. Если вы внимательно посмотрите на скрипт, вы увидите, что он проверяет некоторые комбинации (n_1, n_2) более 1 раза. Например, числа (1, 2) были проверены в первой итерации внешнего цикла, а числа (2, 1) — во второй итерации. Эти два теста избыточны, нам достаточно одного из них. Это можно исправить, сделав так, чтобы внутренний цикл начинал с n_1, а не с 1. Внесите эти изменения в скрипт и запустите его, чтобы проверить, что он работает так, как предполагалось.

Рекурсия: процедуры, которые вызывают себя сами

Recursion.sb2

Структуры, с которыми мы сталкивались раньше, позволяют нам повторять команду или набор команд посредством итераций. Еще одна эффективная техника повторения — *рекурсия*. Она позволяет процедуре либо

снова вызвать саму себя напрямую, либо опосредованно через другую процедуру (например, A вызывает B, B вызывает C, затем C вызывает A). Полезность такого подхода может быть сразу не очевидна, но на деле рекурсия упрощает решение многих задач программирования. Посмотрим, как работает этот метод, на простом примере, показанном на рис. 7.17.

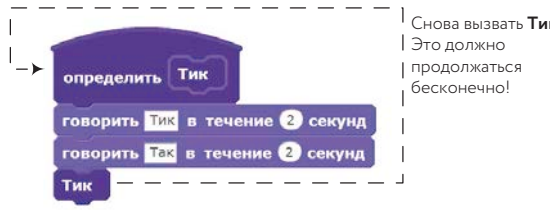


Рис. 7.17. Рекурсивная процедура

Процедура **Тик** выполняет две команды **говорить** (первая говорит «Тик», вторая — «Так»), затем вызывает себя снова. Спрайт продолжит говорить «Тик-Так» бесконечно, если его не остановит внешнее действие. Единственный способ остановить процедуру — кликнуть по красному значку «Стоп». То, что процедура сама себя вызывает подобным образом, дает нам возможность бесконечно повторять две команды **говорить**, не используя блоков-циклов. Форма рекурсии, использованная в этом примере, называется *концевой*, поскольку рекурсивный вызов вставлен в самом конце процедуры. В Scratch рекурсивные вызовы могут располагаться и раньше, но такой тип рекурсии мы рассматривать не будем.

Бесконечная рекурсия — в целом не лучшая идея. Стоит контролировать выполнение процедуры с помощью условных операторов. Например, процедура может включать блок **если**, который определяет, нужен ли рекурсивный вызов. На рис. 7.18 показана рекурсивная процедура, которая от некой первой цифры (ее задает переменная счет) считает вниз до 0.

Разберемся, как работает процедура **ОбратныйСчет**, если ее вызывают с аргументом 3. Когда процедура запускается, команда **говорить** выводит цифру 3, затем проверяет, верно ли, что переменная счет больше 0. Поскольку 3 больше 0, процедура вычтет 1 из значения переменной счет, чтобы вызвать себя саму с аргументом 2. При втором вызове процедура показывает число 2 и, поскольку 2 больше 0, еще раз вызывает себя с аргументом 1. Это будет продолжаться до тех пор, пока не будет совершен вызов **ОбратныйСчет(0)**. Когда цифра 0 появится в облачке разговора, процедура проверит, больше ли значение переменной счет, чем 0. Поскольку выражение в шапке блока **если** оценивается

как ложь, дальнейших рекурсивных вызовов не будет и процедура закончит работу. Попробуйте проследить путь возврата на рис. 7.18.

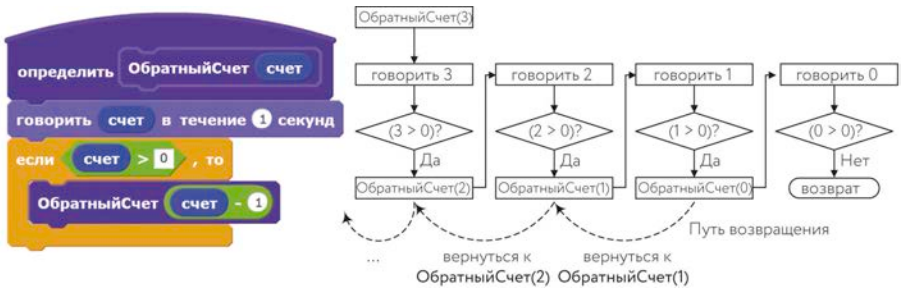


Рис. 7.18. Блок ***если*** *используется, чтобы определить, нужно ли совершить рекурсивный вызов*

Теперь, когда мы разобрались с основами концевой рекурсии, мы можем применить этот принцип в более интересных программах. Рассмотрим в качестве примера процедуру **Лезвие** (**Blade**), показанную на рис. 7.19.

RecursionBlade.sb2

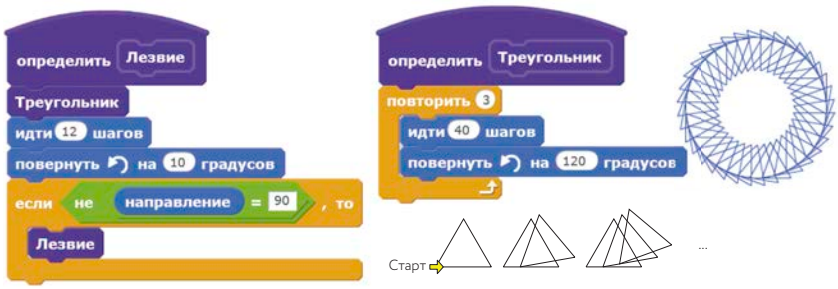


Рис. 7.19. Мы используем направление спрайта, чтобы остановить рекурсию

Предположим, спрайт, выполняющий эту процедуру, начинает работу в некой точке на **Сцене** и его исходное направление — 90°. Нарисовав равносторонний треугольник, спрайт проходит 12 шагов вперед и затем поворачивает на 10° против часовой стрелки. Затем процедура проверяет новое направление спрайта. Если он не смотрит в направлении 90°, процедура вызывает себя еще раз, чтобы нарисовать новый треугольник. Иначе рекурсивного вызова не произойдет и процедура завершит работу после того, как нарисует лезвие, как показано на рис. 7.19.

В простых случаях, вроде приведенных здесь примеров, проще использовать блок **повторить**, чем применять рекурсию. Но, как я уже говорил в начале этого раздела, есть много задач, которые проще выполнить при помощи рекурсии.

Проекты Scratch

Теперь вы в курсе, как использовать в скриптах Scratch повторы для решения задач. Пора применить полученные знания на практике. В этом разделе я покажу вам ряд проектов, чтобы помочь вам укрепить свои знания в программировании и подсказать идеи для ваших собственных проектов.

УПРАЖНЕНИЕ 7.6

Как вам кажется, что делает эта процедура? Внедрите ее и вызывайте с разными аргументами, чтобы проверить себя.

Аналоговые часы

AnalogClock.sb2

Блок **текущее** из раздела **Сенсоры** умеет сообщать текущий год, месяц, дату, день недели, час, минуты или секунды. В нашем первом проекте мы используем его, чтобы создать аналоговые часы (рис. 7.20). В этой программе четыре спрайта: Секунда, Минута и Час (Sec, Min и Hour), которые представляют собой три стрелки часов, и Время (Time) — маленькая белая точка, где отображается время в цифровом формате (см. облачко на рисунке).

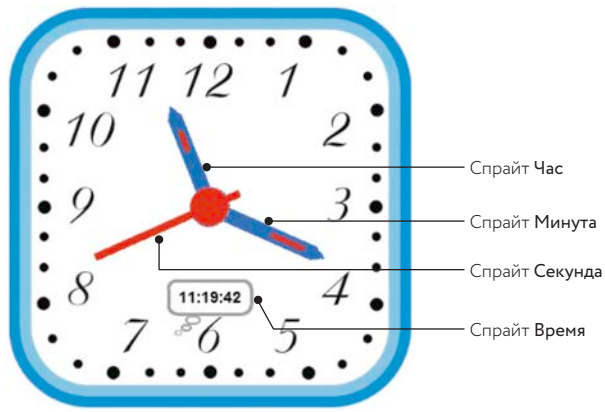


Рис. 7.20. Аналоговые часы

Часы начинают идти после клика по зеленому флажку. Все четыре спрайта начинают цикл **всегда**, в ходе которого они обновляют свой статус на основе текущего времени системы. Скрипты для спрайтов Секунда и Минута показаны на рис. 7.21.

Рис. 7.21. Скрипты для спрайтов Секунда *и* Минута

Блок **текущие** передает число минут или секунд в диапазоне от 0 до 59. Когда система выдает 0 секунд, спрайт Секунда должен повернуться вверх (в направлении 0°), на 15 секунд спрайт Секунда должен повернуться направо (в направлении 90°) и т. д. Каждую секунду стрелка должна повернуться на 6° (360 делить на 60) по часовой стрелке. Так же работает и стрелка Минута. Если вы понаблюдаете за тем, как идут часы, вы заметите, что стрелка Секунда двигается каждую секунду, а стрелка Минута — каждую минуту. Теперь взглянем на скрипт для спрайта Час, показанный на рис. 7.22.

Рис. 7.22. Скрипт для спрайта Час

Блок **текущие(час)** показывает текущий час, используя цифры от 0 до 23. Нам нужно, чтобы часовая стрелка указывала на 0° (вверх), если часов 0, на 30° в час, на 60° в два часа и т. д., как показано на рисунке. Если текущее время, скажем, 11:50, мы хотим, чтобы часовая стрелка указывала не точно на 11, а скорее куда-то ближе к 12. Это уточнение мы можем ввести, если будем принимать во внимание и минуты. Поскольку каждый час (или 60 минут) соответствует 30° на циферблате, на каждую минуту приходится 2°. Таким образом, нам нужно каждую минуту подправлять угол часовой стрелки так, чтобы он соответствовал числу минут, деленному на два, как показано в скрипте. Скрипт для спрайта Время ничем не примечателен, поэтому я здесь его не привожу. В нем используются вложенные блоки **слить** для создания на дисплее строки вида *час*:*мин*:*сек*, и эта строка отображается внутри облачка, как показано на рис. 7.20.

УПРАЖНЕНИЕ 7.7

Откройте программу и запустите ее. Измените скрипт для спрайта Минута так, чтобы он двигался плавно, а не совершал каждую минуту небольшой прыжок. (Подсказка: используйте идею, которую мы применяли, чтобы сделать плавным движение спрайта Час.) Также измените скрипт для спрайта Время так, чтобы отображалась строка в формате «3:25:00 pm» (12-часовой формат) вместо «15:25:00» (24-часовой формат). Подумайте, как еще можно преобразовать программу, и попробуйте внести эти изменения.

Стрельба по птицам

BirdShooter.sb2

Создадим простую игру с использованием всех блоков, с которыми мы познакомились в этой главе. Целью пользователя будет сбить всех птиц в небе, интерфейс игры показан на рис. 7.23.

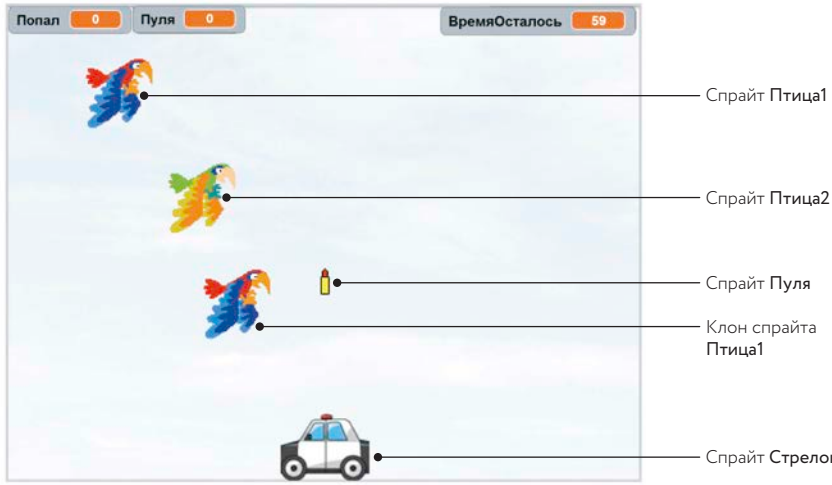


Рис. 7.23. Пользовательский интерфейс игры «Стрельба по птицам»

В игре задействовано пять спрайтов: Птица1, клон Птицы1, Птица2, Стрелок и Пуля (Bird1, Bird2, Shooter и Bullet). Игрок может перемещать Стрелка по горизонтали при помощи клавиш со стрелками вправо и влево. При нажатии клавиши «пробел» в небо вылетит пуля. Если она попадает в Птицу1 или ее клон, игрок получает очко. Птица2 — это птица из Красной книги, в нее стрелять нельзя: если пуля попадет в нее, игра закончится. У пользователя есть одна минута на то, чтобы подстрелить как можно больше птиц. Каждая птица использует два костюма. Переключение с одного на другой создает эффект хлопания крыльями. У **Сцены** два фона: **старт** и **конец**. Первый показан на рис. 7.23. Второй отличается от него

только надписью Game Over (Игра закончена) в центре. Скрипты **Сцены** показаны на рис. 7.24.

Рис. 7.24. Скрипты для **Сцены** *в игре*

Когда зеленый флажок нажат, **Сцена** меняет фон на **старт**, обнуляет таймер и запускает цикл, который будет обновлять и проверять оставшееся время игры (его отслеживает переменная ВремяОсталось). Когда эта переменная достигнет 0 или **Сцена** получит сообщение **ИграЗакончена**, она запустит обработчик события **ИграЗакончена**. Этот скрипт ждет некоторое время, чтобы позволить птицам спрятаться, меняет фон на **конец** и вызывает команду **стоп все**, чтобы завершить все работающие скрипты. Как вы вскоре увидите, сообщение **ИграЗакончена** отправляется спрайтом Пуля при попадании в Птицу2. А теперь посмотрим на скрипт спрайта Стрелок на рис. 7.25.

Рис. 7.25. Скрипт для спрайта Стрелок

Этот скрипт первым делом размещает Стрелка внизу **Сцены** по центру. Затем он запускает бесконечный цикл, который определяет, были ли нажаты клавиши со стрелками вправо и влево, и перемещает Стрелка в соответствующую сторону. Теперь перейдем к скриптам для спрайта Птица1, показанным на рис. 7.26.

Рис. 7.26. Скрипты спрайта Птица1

Когда игра начинается, спрайт Птица1 клонирует себя, перемещается к левому краю **Сцены** и вызывает процедуру **Старт**. Клон также начинает в левом краю **Сцены** (на другой высоте) и вызывает процедуру **Старт**. Она использует цикл **всегда**, чтобы перемещать птицу и ее клон по горизонтали через всю **Сцену**, слева направо на случайное число шагов. Когда птица достигает правого края **Сцены**, она перемещается обратно на левый край, как будто пролетела вокруг и появилась снова. Последний скрипт прячет обеих птиц, когда спрайту передается сообщение **ИграЗакончена**. Скрипты для Птицы2 очень похожи на скрипты для Птицы1, и мы не будем их тут разбирать. После клика по зеленому флажку Птица2 перемещается на правый край **Сцены** на высоте 40 и затем выполняет цикл, подобный процедуре **Старт** на рис. 7.26. Птица передвигается справа налево и как будто пролетает вокруг, достигая правого края **Сцены**. Птица2 также реагирует на сообщение **ИграЗакончена**, спрайт скрывается. Естественно, пользователь не может сбивать птиц, просто перемещая Стрелка, для этого нужен спрайт Пуля. Основной его скрипт показан на рис. 7.27.

Когда пользователь щелкнет по зеленому флажку, скрипт инициализирует переменные Выстрелил (число совершенных выстрелов) и Попал (сколько птиц сбито) со значением 0. Затем он поворачивает спрайт Пуля по направлению вверх и скрывает его. Потом он запускает бесконечный цикл, чтобы снова и снова перепроверять статус клавиши «пробел». Когда она нажата, скрипт увеличивает переменную Выстрелил на 1 и создает клон спрайта Пуля, чтобы переместить пулю вверх. Затем скрипт ждет какое-то время, чтобы игрок не смог слишком рано выстрелить снова. Теперь мы готовы запустить скрипт клонированной пули, показанный на рис. 7.28. Сначала Пуля перемещается в центр к Стрелку и становится видимой ❶. Затем Пуля перемещается вверх прыжками по 10 шагов при помощи команды **повторять пока не** ❷. Если значение координаты *y* Пули превысит 160, значит, она достигла верхнего края **Сцены**, не попав ни в одну

птицу. В этом случае блок **повторять пока не** заканчивает работу и клон ликвидируется ❺. Проверка попаданий проводится при каждом движении пули. Если пуля коснется Птицы1 (или ее клона) ❸, скрипт увеличит значение переменной Попал и проиграет звук. А если пуля коснется Птицы2 ❹, скрипт передаст сообщение **ИграЗакончена**, чтобы сигнализировать, что игра закончена. В обоих случаях клон удаляется: он свою работу выполнил. Теперь игра готова, но вы можете добавить много дополнительных функций. Вот несколько предложений.

Рис. 7.27. Основной скрипт спрайта Пуля

- Дайте пользователю ограниченное количество пуль и ведите счет на основании числа пропущенных выстрелов.
- Увеличьте число птиц и сделайте так, чтобы они летели с разной скоростью.
- Вознаграждайте игрока бóльшим количеством очков за попадание в более быстрых птиц.

УПРАЖНЕНИЕ 7.8

Запустите игру и попробуйте поиграть в нее, чтобы увидеть, как она работает. Внесите какие-нибудь из указанных выше изменений — или предложите свои идеи и внедрите их!

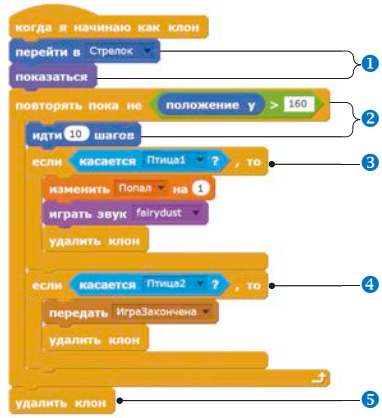


Рис. 7.28. Обработчик запуска клонированной пули

Симуляция свободного падения

FreeFall.sb2

В этом разделе я покажу вам программу, которая моделирует свободное падение предмета. Мы не будем учитывать такие факторы, как подъемная сила и сопротивление воздуха. Если предмет в состоянии покоя бросить с некоторой высоты, то расстояние d (м), которое он пролетит за время t (с), можно вычислить при помощи формулы $d = {}^1/_2 g t^2$, где $g = 9{,}8$ м/с^2 — ускорение свободного падения. Цель симуляции — показать положение падающего предмета в определенный момент времени: 0,5 с, 1 с, 1,5 с, 2 с и т. д., пока предмет не достигнет земли. Интерфейс этой симуляции показан на рис. 7.29.

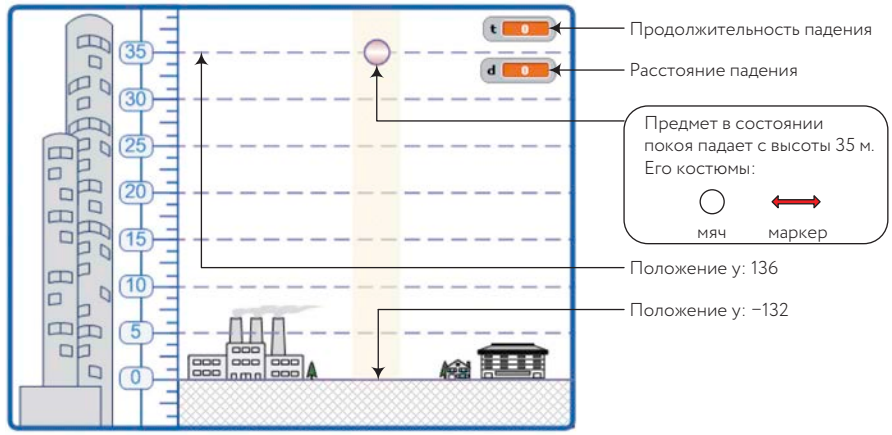


Рис. 7.29. Пользовательский интерфейс симуляции свободного падения

Предмет в состоянии покоя (мяч на рисунке) будет брошен с высоты 35 м. Простая подстановка данных в приведенную выше формулу показывает, что предмет достигнет земли через $t = \sqrt{(2 \times 35)/9{,}8} = 2{,}67$. В программе задействован один спрайт (Мяч), у которого два костюма. Когда приходит время показать положение падающего мяча, спрайт на мгновение меняет костюм на маркер, ставит печать и переключается обратно на костюм мяча. Симуляция начинается после клика по зеленому флажку. В ответ спрайт Мяч запускает скрипт, показанный на рис. 7.30. Во время инициализации ❶ спрайт перемещается на исходную позицию, меняет костюм на мяч, стирает из облака оставшееся с прошлого раза сообщение и очищает **Сцену** от прошлой печати. Затем он инициализирует переменные t и счетчик, присваивая им значение 0. Переменная t отражает продолжительность падения, а счетчик отслеживает число итераций цикла. Затем скрипт входит в бесконечный

цикл ❷, чтобы замерять параметры симуляции с разными интервалами. Переменная t выполняет эти расчеты и обновляет информацию о положении мяча каждые 0,05 с ❸, чтобы обеспечить плавность движения. Значение переменной t обновляется, и подсчитывается расстояние, которое пролетел мяч (d). Значение переменной счетчик также увеличивается на 1.

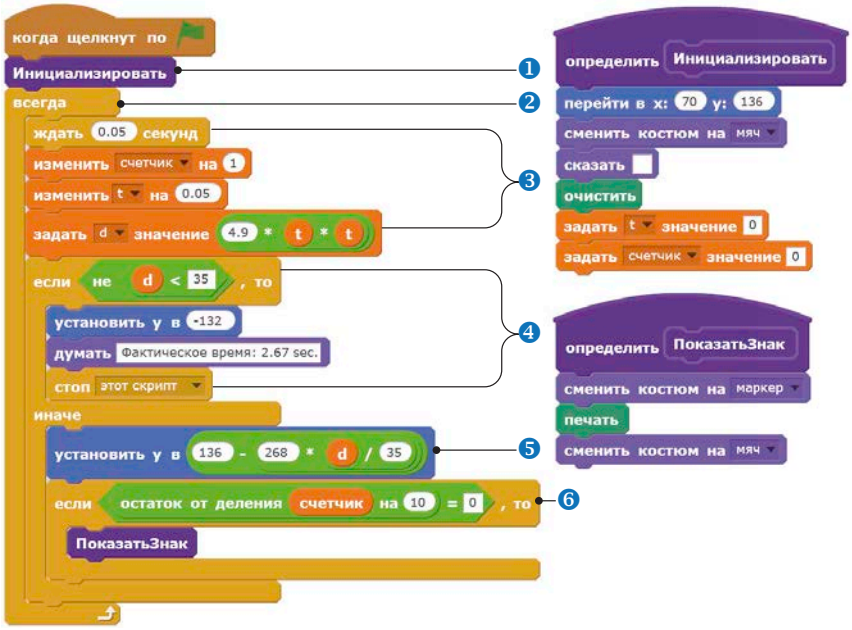


Рис. 7.30. Скрипт спрайта Мяч *для симуляции свободного падения*

Если мяч достигает земли ($d \geq 35$), скрипт устанавливает координату y мяча на уровне земли, отображает продолжительность пути и останавливает скрипт, чтобы завершить работу ❹. В других случаях скрипт устанавливает позицию мяча по вертикали в соответствии с пройденным расстоянием ❺. Поскольку высота 35 м соответствует 268 пикселам на **Сцене** (см. рис. 7.29), расстояние в d метров соответствует $268 \times (d/35)$. Окончательное значение координаты y вычисляется путем вычитания этого числа из изначального значения координаты y, то есть 136. Поскольку продолжительность итерации 0,05 с, чтобы получить 0,5 с, нужно 10 итераций. Когда число итераций достигает 10, 20, 30 и т. д., спрайт Мяч меняет костюм (и печати) на маркер, чтобы показать положение падающего мяча в эти моменты ❻. Рис. 7.31 демонстрирует результат работы симуляции. Обратите внимание: расстояние,

которое преодолевает мяч в падении, с каждым промежутком времени увеличивается. Из-за силы притяжения мяч летит все быстрее.

УПРАЖНЕНИЕ 7.9

Откройте программу и запустите ее, чтобы понять, как она работает. Попробуйте сделать из симуляции игру, в которой пользователи бросают мяч, чтобы попасть в объект, перемещающийся по земле. Можете добавить счет, изменить скорость мишени или даже перенести действие на другую планету (измените ускорение свободного падения).

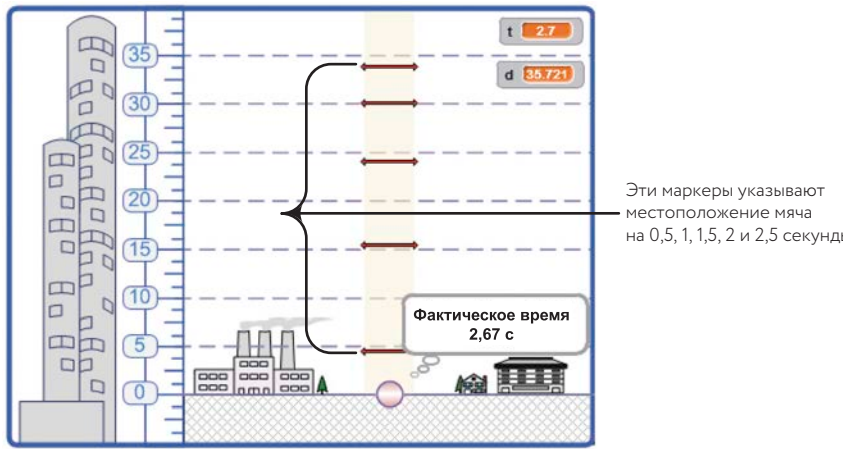


Рис. 7.31. Результат симуляции свободного падения

Симулятор полета ядра

Projectile.sb2

Рассмотрим ядро, которым выстрелила пушка, направленная под углом q к земле, и которое летит со скоростью (v_0). Вы можете проанализировать траекторию полета, разделив вектор скорости (v_0) на горизонтальный и вертикальный компоненты в разные моменты. Горизонтальный компонент остается неизменным, а на вертикальный воздействует сила притяжения. Когда движения, связанные с этими двумя компонентами, объединяются, они образуют путь в виде параболы. Рассмотрим уравнения для движения ракеты (не учитывая сопротивление воздуха). Начало координат в нашей системе — точка, в которой мяч начинает полет. Так что значение координаты x в любой момент времени (t) задается уравнением $x(t) = v_0xt$, а координаты y: $y(t) = v_0yt - (0{,}5)gt^2$, где $v_0x = v_0\cos(q)$ — компонент x, $v_0y = v_0\sin(q)$ — компонент y, а $g = 9{,}8$ м/с^2 — ускорение свободного падения. Используя эти уравнения, мы можем вычислить общее время полета, максимальную высоту и расстояние, на которое

мяч перемещается по горизонтали. Соответствующие уравнения вы видите на рис. 7.32.

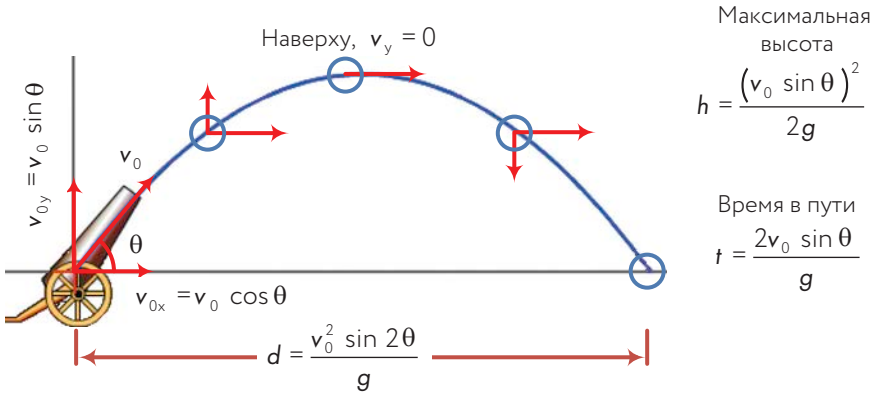


Рис. 7.32. Параболическая траектория ядра

Это всё, что нам нужно, чтобы создать реалистичную модель движения ядра. Напишем в Scratch программу, чтобы посмотреть на это физическое явление в действии, и углубим наши знания о траекториях движения. Пользовательский интерфейс симуляции показан на рис. 7.33.

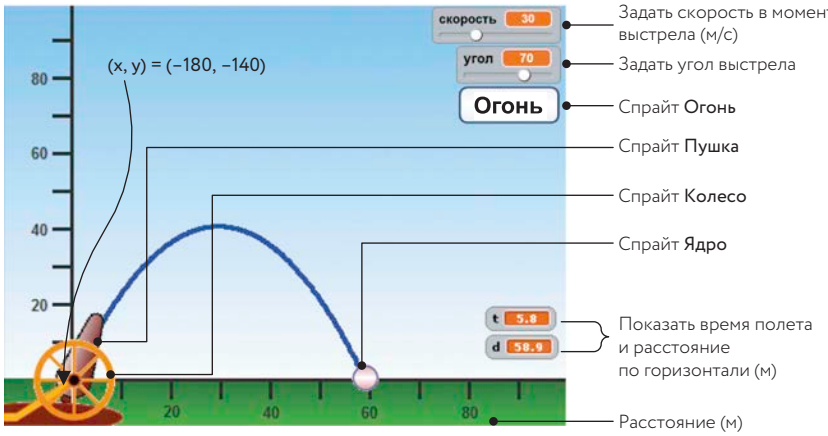


Рис. 7.33. Пользовательский интерфейс симулятора полета ядра

В программе задействовано четыре спрайта. Колесо (Wheel) создает для пушки ось вращения, а Пушка (Cannon), вращающаяся в соответствии с положением ползунка, наглядно демонстрирует угол выстрела. Спрайт Огонь (Fire) — кнопка, которую нажимает пользователь, чтобы

выстрелить, а спрайт Ядро (Ball) содержит основной скрипт для вычисления координат и рисования траектории. Пользователь определяет угол выстрела и стартовую скорость при помощи двух ползунков, затем жмет на кнопку Огонь. Ядро начинает движение из точки (–180, –140) на **Сцене** и рисует параболическую траекторию на основании специфических параметров. Два окошка в нижнем правом углу **Сцены** показывают время полета и пройденное по горизонтали расстояние. Симуляция начинает работать после клика по зеленому флажку. Скрипты для спрайта Пушка (здесь не приведены) наводят пушку в соответствии с положением ползунка **угол**. Пользователь также может выбрать угол, кликнув по пушке и перетащив ее. Когда пользователь нажимает на кнопку Огонь, она передает сообщение Огонь, которое получает и обрабатывает спрайт Ядро при помощи скрипта, который показан на рис. 7.34.

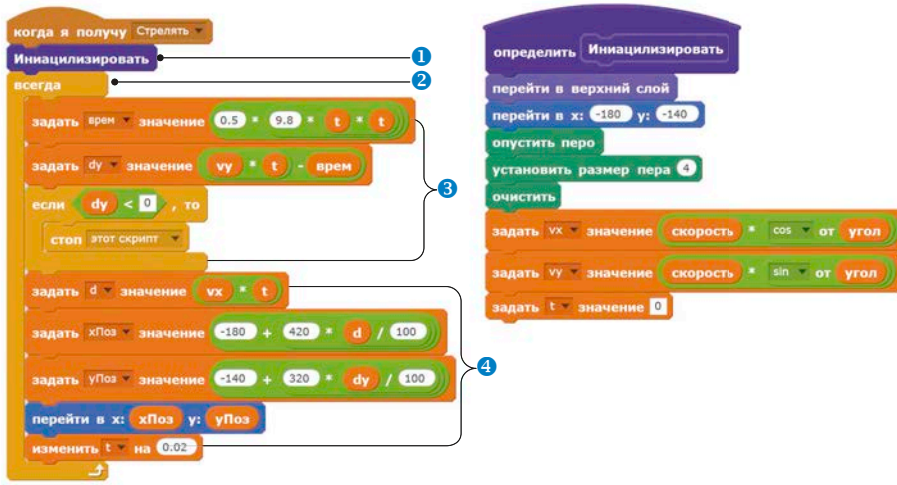


Рис. 7.34. Скрипт спрайта Ядро

При подготовке к выстрелу ❶ Ядро оказывается перед Пушкой и Колесом и занимает стартовую позицию. Оно опускает свое перо и стирает все следы пера со **Сцены**. Затем скрипт вычисляет горизонтальный (x) и вертикальный (y) компоненты начальной скорости (v_x и v_y соответственно) и инициализирует переменную времени (t), присвоив ей значение 0.

Затем скрипт запускает бесконечный цикл ❷, который вычисляет и актуализирует позицию ядра каждые 0,02 с. Сначала вычисляется местоположение спрайта по вертикали (d_y) ❸. Если значение отрицательное, ядро достигло земли. Тогда вызывается команда **стоп этот скрипт**, чтобы остановить программу. Если же d_y — не отрицательное,

вычисляется положение по горизонтали (d) ❹. Эти две величины (d_y и d) затем приводятся к общему знаменателю с фоном **Сцены**. В вертикальном направлении у нас 320 шагов (от –140 до 180), что соответствует 100 м, а по горизонтали 420 шагов (от –180 до 240), что соответствует 100 м. Это означает, что вертикальное расстояние в d_y метров — эквивалент $320 \times d_y$ / 100 шагов, а горизонтальное расстояние в d метров — $420 \times d$ / 100 шагов. Координаты ядра x и y обновляются, и оно перемещается по своей траектории на текущую позицию. Переменная времени (t) немного увеличивается (в данном случае на 0,02 с), а цикл повторяется, чтобы вычислить следующую позицию ядра. Если ядро было выпущено под углом 70° с начальной скоростью 30 м/с, как показано на рис. 7.33, то общее время его полета будет 5,75 с и оно улетит на 59 м. Проверка мониторов на рис. 7.33 показывает, что симуляция работает очень точно. Улучшить программу можно, еще чаще обновляя данные вычислений (например, каждые 0,01, а не 0,02 с), но это замедлит симуляцию. Этот параметр нужно доработать, чтобы достичь хорошего соотношения скорости и точности.

УПРАЖНЕНИЕ 7.10

Откройте программу и запустите ее, чтобы понять, как она работает. Попробуйте превратить симуляцию в игру. Например, вы могли бы показать предмет на случайной высоте с правого края **Сцены** и попросить пользователя попасть в него из пушки. Если пользователь не попадет в цель, игра может дать ему пару подсказок, чтобы подобрать правильный угол и скорость.

Другие программы

MatchThat Amount.sb2

Среди дополнительных ресурсов к этой книге (*http://nostarch.com/learnscratch/*) есть еще три игры, которые можно изучить самостоятельно, к каждому скрипту дано полное описание. Первая из них — обучающая игра, которую можно использовать для проверки навыков счета у школьников младших классов. Игра показывает сумму денег и предлагает пользователю собрать эту сумму, используя наименьшее количество монет.

Orbit.sb2

Molecules InMotion.sb2

Вторая программа — симуляция движения планет для упрощенной модели Солнечной системы, куда входят Солнце и одна планета. Третья — еще одна симуляция, демонстрирующая динамику движения одиночной молекулы газа, которая наталкивается на стенки контейнера. Откройте программы, запустите их, изучите пояснения и постарайтесь понять, как они устроены. Если вам захочется поразмяться, попробуйте изменить скрипты так, чтобы заставить программы делать что-то новенькое.

Итоги

В этой главе мы рассмотрели альтернативные способы повторения команд в Scratch. Сначала мы познакомились с блоками-циклами и разобрали связанную с ними техническую терминологию. Затем обсудили бесконечный цикл и цикл с ограниченным числом повторов, узнали разницу между циклами со счетчиками и циклами, управляемыми условиями. Мы познакомились со структурами **повторять пока не** и **всегда если**, привели несколько примеров их использования. Я также разъяснил стоп-команды Scratch и рассказал, как их использовать для остановки бесконечных циклов и процедур. После этого мы обсудили использование циклов для проверки данных, введенных пользователем. Затем вы узнали, как использовать счетчики, чтобы отслеживать количество итераций цикла, и как использовать счетчики со вложенными циклами, чтобы создавать итерации в двух и более измерениях. После этого мы рассмотрели рекурсию — процедуру, вызывающую саму себя, — как еще один способ обеспечить повторы. В последнем разделе мы создали несколько программ, где на практике связали воедино эти новые структуры. В следующей главе я дам больше информации по уже знакомым вам темам. Вы научитесь использовать счетчики и циклы для работы со строками и создавать интересные программы другого уровня: преобразователь двоичных чисел в десятичные, игру «Виселица» и программу для обучения дробям. Если вы хотите подробнее изучить темы, с которыми познакомились в этой главе, решите приведенные ниже задачи.

Задания

1. Создайте цикл для проверки введенных пользователем данных, который принимает только числа от 1 до 10.

2. Напишите скрипт, который спрашивал бы пользователя: «Вы уверены, что хотите выйти из программы [Д, Н]?» Затем скрипт проверяет данные, введенные пользователем, и в качестве ответа принимает только буквы Д и Н.

3. Напишите программу, которая вычисляет и отображает сумму всех целых чисел от 1 до 20.

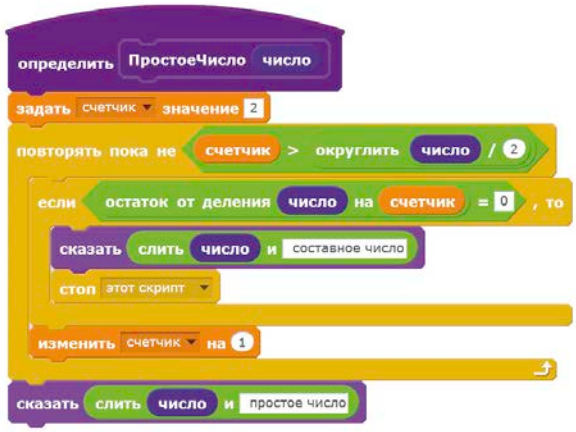
4. Напишите программу, которая вычисляет и отображает сумму всех нечетных целых чисел от 1 до 20.

5. Напишите программу, которая отображает первые 10 чисел в такой последовательности (используйте команду **говорить**): *5*, 9, 13, *17*, 21, ...

6. Что делает скрипт ниже? Внедрите его в программу и запустите ее, чтобы проверить ответ.

7. Если при делении одного целого числа (*x*) на другое (*y*) в остатке 0, мы говорим, что *y* — делитель для *x*. Например, 1, 2, 4 и 8 — делители для 8. Приведенный ниже скрипт находит и отображает все делители для заданных чисел (отличные от самого числа). Изучите его устройство и объясните, как он работает. Какими будут результаты, если ввести числа 125, 324 и 419?

8. Целое число называют простым, если оно делится только на 1 и на само себя. Например, 2, 3, 5, 7, 11 — простые числа, а 4, 6 и 8 — нет. Процедура ниже проверяет, является ли число простым или нет. Изучите процедуру и объясните, как она работает. Какими будут результаты ее работы, если ввести числа 127, 327 и 523?



9. Хотя процедура из задачи 8 проверяет все целые числа вплоть до половины от введенного значения, достаточно в качестве верхней границы поставить квадратный корень от введенного числа. Внесите это изменение в процедуру и проверьте, будет ли она выдавать те же ответы.

10. Ряд чисел 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... называется последовательностью Фибоначчи. Два первых ее числа — 0 и 1. Каждое последующее число — сумма двух предыдущих. Напишите программу, которая вычисляла бы число *n* последовательности Фибоначчи, где порядковый номер *n* задается пользователем.

11. Изучите следующую программу и результат ее работы. Воссоздайте программу и запустите ее, чтобы посмотреть, как она работает. Измените угол поворота (от 10°) и аргумент рекурсивного вызова (на сторона + 1 или сторона + 3 и т. д.), чтобы посмотреть, что еще вы можете нарисовать.

8

ОБРАБОТКА СТРОК

Строка — последовательность символов, которая воспринимается как единое целое. Вы можете писать программы, которые комбинируют, сравнивают, шифруют и иными способами манипулируют строками. Вот о чем вы узнаете из этой главы:

- как Scratch сохраняет строки;
- как использовать блоки, манипулирующие строками, в Scratch;
- каковы приемы обработки строк;
- как написать интересные программы, которые обрабатывают строки.

Начнем с подробного обзора строковых данных, а затем напишем процедуры для управления и манипулирования строками. Они будут удалять и заменять символы, вставлять и убирать подстроки, переставлять символы в случайном порядке. Потом мы применим эти процедуры и приемы, чтобы написать полезные и интересные приложения.

Повторение: тип данных — строка

Как я упоминал в главе 5, Scratch поддерживает три вида данных: логические величины, числа и строки. Проще всего сказать, что строка — упорядоченная последовательность символов: букв (верхнего и нижнего регистра), цифр и прочих символов, которые можно набрать с помощью

клавиатуры (+, -, &, @ и т. д.). Строки можно использовать, чтобы сохранять имена, адреса, номера телефонов, названия книг и многое другое.

В Scratch символы строки сохраняются последовательно. Например, если у вас есть переменная имя, выполняющая команду **задать имя значение Карен**, то символы будут сохраняться как показано на рис. 8.1.

Рис. 8.1. Строка сохраняется в виде последовательности символов

Вы можете получить доступ к отдельным символам строки с помощью оператора **буква в**. Например, блок **буква 1 в имени** выдаст «К», а блок **буква 5 в имени** — «н». В Scratch также есть оператор **длина**, который высчитывает количество символов в строке. Если вы используете эти два оператора с блоками **повторить**, то сможете пересчитывать символы, анализировать множество символов и делать еще много полезного.

Подсчет специальных символов в строке

VowelCount.sb2

Первый скрипт, приведенный на рис. 8.2, подсчитывает, сколько гласных в строке ввода. Скрипт просит пользователя ввести строку, а затем просчитывает ее и отображает число гласных в ней.

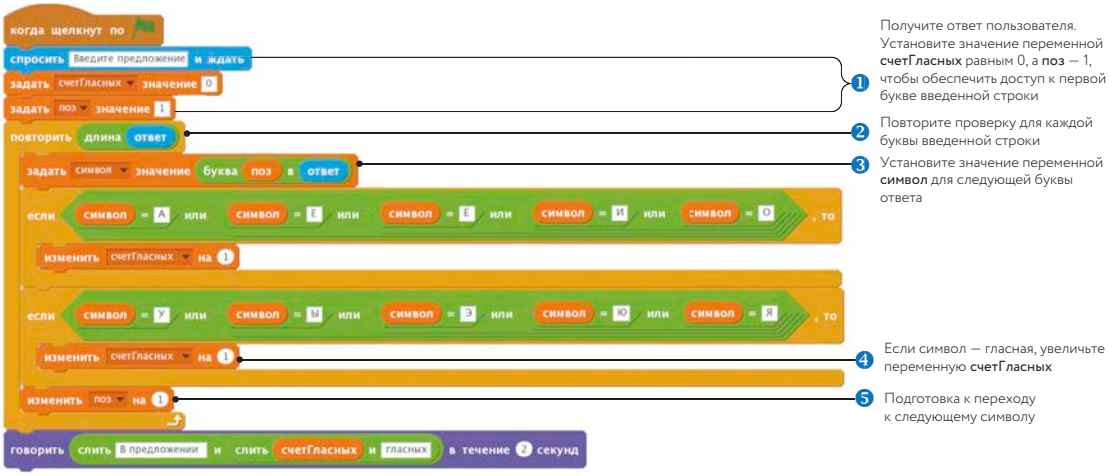


Рис. 8.2. Программа для подсчета гласных

Программа по очереди проверяет все буквы во введенной строке. Всякий раз, когда она находит гласную букву, она увеличивает значение

переменной счетГласных на 1. Скрипт использует переменную поз, чтобы отслеживать позицию проверяемого символа в слове. Посмотрим на этот скрипт внимательнее.

Сначала скрипт просит ввести предложение ❶. Scratch должен сохранить введенную пользователем строку автоматически, во встроенной переменной ответ. Затем он присваивает переменной счетГласных значение 0 (пока гласных не найдено), а переменной поз — 1, чтобы открыть доступ к первой букве введенной строки.

Цикл **повторить** ❷ проверяет каждую букву во введенной строке. Оператор **длина** сообщает количество символов в строке — а это и есть количество повторов цикла.

При каждом повторе цикл использует переменную символ, чтобы проверить один символ из введенной строки ❸. При первой итерации значение этой переменной установлено как первая буква **ответа**. Вторая итерация меняет значение на вторую букву и т. д., пока цикл не достигнет конца строки. Переменная поз используется для доступа к нужному символу.

Блок **если** проверяет, является ли символ гласной ❹. Если да, значение переменной счетГласных увеличивается на 1.

Проверив символ, цикл увеличивает значение поз на 1 ❺ и переходит к следующему символу. После того как все буквы в строке прошли проверку, цикл заканчивается и программа отображает количество обнаруженных гласных с помощью блока **говорить**.

Методики, использованные в этом примере, будут неоднократно применяться в данной главе. Загрузите скрипт *VowelCount.sb2*, прогоните его несколько раз и убедитесь, что хорошо понимаете, как он работает.

Сравнение символов строки

Palindrome.sb2

Второй пример проверяет, не является ли целое число, введенное пользователем, палиндромом. Палиндром — число (или текстовая строка), который одинаково читается вперед и назад. Например, палиндромы — числа 1234321 и 1122332211. Текстовые палиндромы — «Боб», «Анна», «наган». Чтобы проиллюстрировать нашу процедуру, предположим, что вводимое число — 12344321, как показано на рис. 8.3.

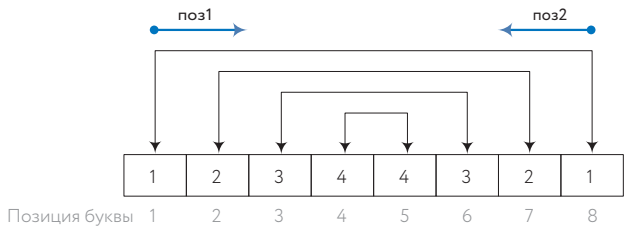


Рис. 8.3. Использование двух переменных для проверки того, является ли число палиндромом

Чтобы проверить, является ли число палиндромом, нужно сравнить попарно первую и восьмую цифры, вторую и седьмую, третью и шестую и т. д. Если сравнение дает результат «неверно» (то есть две цифры друг другу не равны), число — не палиндром. Программа, которая реализует процедуру тестирования на палиндромность, приведена на рис. 8.4.

Рис. 8.4. Эта программа проверяет, является ли введенное целое число палиндромом

Скрипт работает с цифрами, которые надо сравнить, с помощью двух переменных (поз1 и поз2 на рис. 8.3), которые двигаются в противоположных направлениях. Первая, поз1, начинает с первой цифры и двигается вперед, а вторая, поз2, начинает с последней цифры и двигается назад. Количество необходимых сравнений равняется максимум половине количества цифр во введенном номере. Введя число 12344321, мы должны провести сравнение четыре раза, поскольку во введенном числе 8 цифр (та же логика применяется и в случае, если введенное целое число имеет нечетное количество цифр: цифру в середине не с чем сравнивать). Как только программа определила, является ли число палиндромом или нет, она отображает сообщение с результатом.

УПРАЖНЕНИЕ 8.1

Palindrome.sb2

Загрузите файл *Palindrome.sb2* и запустите его, чтобы посмотреть, как он работает. Структура обработки десятичных чисел Scratch такова: если введенное число состоит из нечетного количества цифр, скрипт выполняет одно лишнее сравнение цифр, окружающих цифру в середине.

Попробуйте исправить программу, чтобы она выполняла корректное количество повторов, обрабатывая число с нечетным количеством цифр.

Ниже мы разберем самые распространенные операции со строками и исследуем стратегии для написания процедур Scratch, манипулирующих строками.

Примеры манипулирования строками

Оператор **буква в** позволяет вам только читать отдельные символы строки. Если вы хотите вставлять символы в строку или удалять их из нее, придется проделать эту работу самостоятельно.

В Scratch вы не можете изменить символы в строке, единственный способ изменить строку — создать новую. Например, если вы хотите заменить первую строчную букву на прописную в строке «джек», надо создать новую строку, в которой будет сначала буква «Д», а потом остальные — «жек». Идея в том, чтобы с помощью оператора **буква в** прочесть буквы изначальной строки и перенести эти буквы в новую строку в нужном виде с помощью оператора **слить**.

В этом разделе мы разработаем простые приложения, которые продемонстрируют распространенные методы манипулирования строками.

Оросячьяпей атыньлей

PigLatin.sb2

Что если наши спрайты умеют общаться на тайном языке? В этом разделе мы научим их языку-коду, который называется «поросячья латынь». Правила создания слов простые: чтобы «перевести» слово на поросячью латынь, перенесите первую букву слова в его конец и добавьте слог «ей». Тогда слово «разговор» будет «азговоррей», «забава» — «абавазей» и т. д. Теперь вы наверняка сможете перевести название этого раздела.

Стратегия, которую мы будем использовать, чтобы перевести слово на поросячью латынь, показана на рис. 8.5 для слова «абрикос».

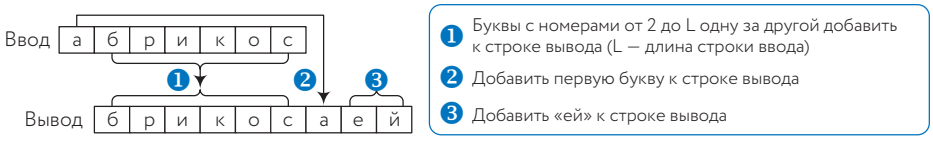


Рис. 8.5. Как перевести слово на поросячью латынь

Сначала мы добавим все буквы введенного слова (за исключением первой), одну за одной, к слову вывода ❶. Затем первую букву слова ввода перенесем в вывод ❷ и добавим «ей». Процедура **ПоросячьяЛатынь**, выполняющая эти шаги, приведена на рис. 8.6.

Эта процедура использует три переменные для создания закодированных слов. Переменная выводСлово дает нам строку вывода. Счетчик поз (от «позиция») информирует скрипт, какой символ изначальной строки добавлен к выводСлово. Наконец, переменная с хранит один символ из строки ввода. Процедура воспринимает слово, которое вы хотите перевести на поросячью латынь, в качестве параметра с именем «слово».

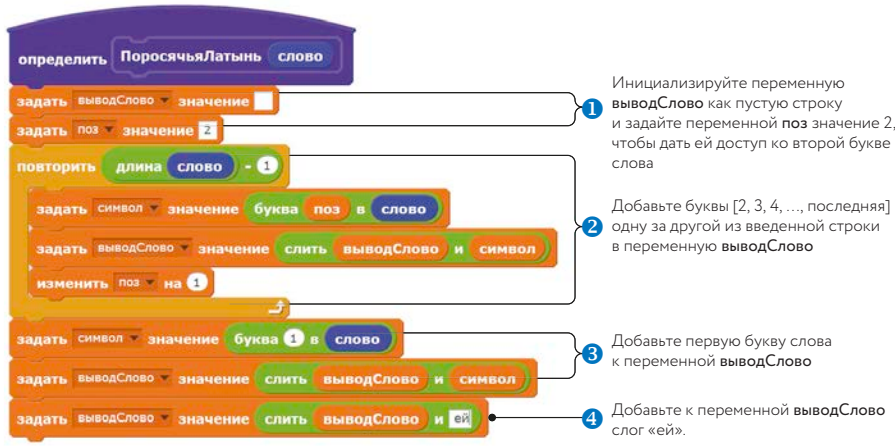


Рис. 8.6. Процедура **ПоросячьяЛатынь**

Первым делом процедура создает пустую строку для переменной выводСлово и задает переменной поз значение 2 ❶ (*пустая строка* — строка, в которой нет символов; ее длина равна 0). Затем она использует блок **повторить**, чтобы добавить все буквы, кроме последней, из введенной строки (слово) в строку вывода (выводСлово) ❷. Мы пропустили первую букву, так что теперь счет на единицу меньше длины введенной строки. На каждой итерации цикла к выводСлово прибавляется один символ слова ❸, а потом и слог «ей» ❹.

УПРАЖНЕНИЕ 8.2

PigLatin.sb2

Загрузите файл *PigLatin.sb2* и запустите его, чтобы проверить процедуру. Приложение запрашивает слово, а затем дает его перевод на поросячью латынь. Измените приложение так, чтобы оно переводило на поросячью латынь фразу, например «Хочешь сока?» (подсказка: обращайтесь к **ПоросячьяЛатынь** за каждым словом, чтобы собрать фразу). Еще одна трудная, но интересная задача: написать процедуру, которая переводит с поросячьей латыни.

Исправь ошибки

FixMySpelling.sb2

Мы создадим простую игру, которая выдает слова с ошибками и просит пользователя исправить их. Ошибки в словах игра будет создавать, вводя случайную букву на случайно выбранную позицию в слове. Конечно, для неверно написанного короткого слова может быть несколько правильных вариантов. Например, если изначальное слово — «лоза», а процедура дает нам «лкоза», то правильными вариантами можно счесть и «коза», и «лоза». Чтобы не усложнять игру, будем исходить из того, что правильный ответ один.

Для начала создадим общую процедуру, чтобы вставлять символы в конкретное место в строке. Эта процедура с именем **Вставка** использует три параметра: введенное слово (стрВв), строку (или символ) для вставки (стрДоб), а также позицию, на которой вы хотите увидеть эти новые символы (симвПоз). Она создает новую строку (стрВыв) с стрДоб, вставленным в стрВв в нужное место, как показано на рис. 8.7.

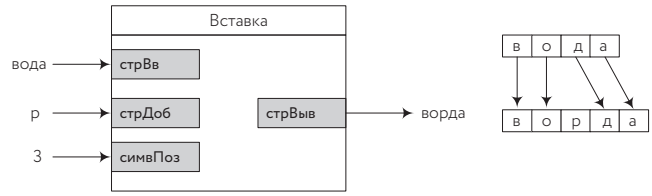


Рис. 8.7. Процедура **Вставка**

Добавим по очереди символы из стрВв в стрВыв. Когда мы дойдем до симвПоз, мы добавим один или несколько символов из стрДоб в стрВыв, прежде чем перенести букву с симвПоз из стрВв. Вся процедура целиком приведена на рис. 8.8.

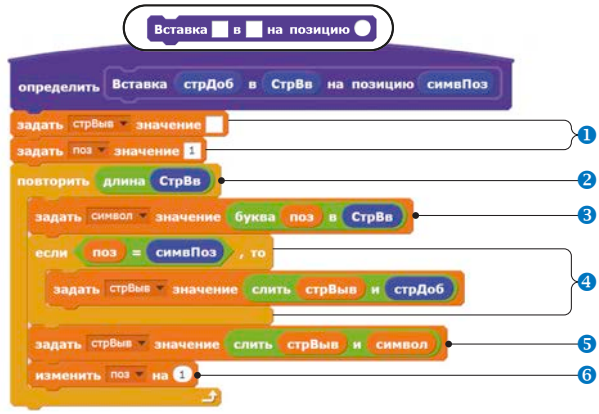


Рис. 8.8. Процедура **Вставка**

Сначала процедура задает переменную стрВыв как пустую и присваивает переменной поз значение 1, чтобы начать обработку первой буквы введенной строки ❶. Затем начинается цикл **повторить**, с помощью которого буквы из стрВв по одной переносятся в стрВыв ❷. При каждой итерации захватывается очередная буква из стрВв и переносится в переменную с ❸. Если позиция текущего символа совпадает со значением переменной симвПоз, процедура добавляет стрДоб к стрВыв ❹, а значение поз увеличивается на 1 для перехода к следующей букве стрВв ❺ ❻.

Теперь посмотрим на основной скрипт игры, показанный на рис. 8.9.

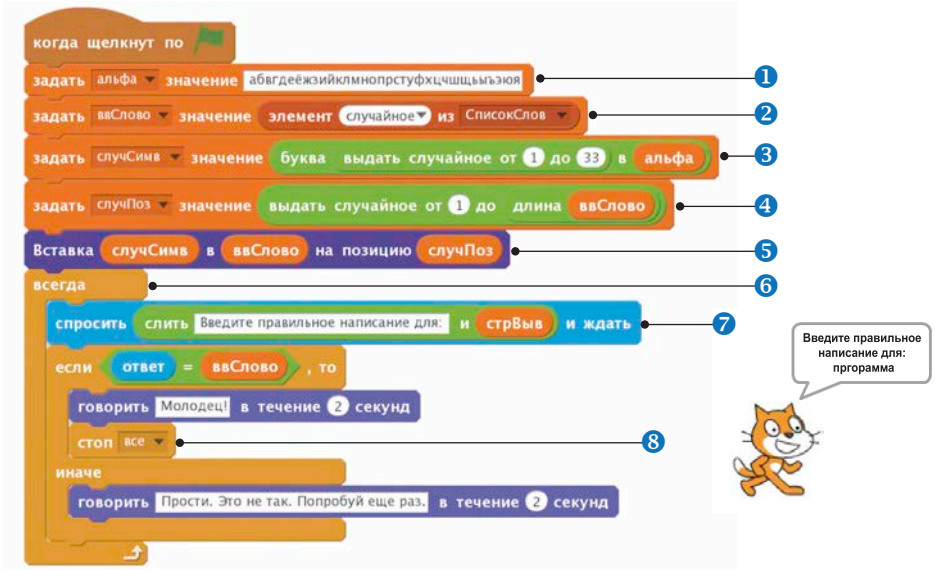


Рис. 8.9. Основной скрипт игры «Исправь ошибки»

В строке альфа содержатся все буквы алфавита. Из нее будет браться случайная буква для вставки в слово, правописание которого мы хотим «испортить» ❶. Скрипт выбирает случайное слово из заранее подготовленного списка и сохраняет его в переменной ввСлово ❷. Больше о списках вы узнаете из следующей главы, а пока представляйте себе список как банк, в котором хранятся слова. Затем скрипт выбирает случайную букву (случСимв) из строки альфа ❸ и случайную позицию (случПоз), чтобы поставить на нее букву в ввСлово ❹. Затем скрипт вызывает процедуру **Вставка**, чтобы создать неверное написание слова (стрВыв) ❺. После этого скрипт начинает цикл ожидания ответа от игрока ❻. Внутри цикла скрипт просит игрока ввести правильно написанное слово ❼ и использует блок **если/иначе**, чтобы проверить ответ ❽. Если ответ пользователя совпадает с изначальным словом (ввСлово), игра окончена; иначе пользователю придется попытать счастья еще раз.

УПРАЖНЕНИЕ 8.3

Загрузите файл *FixMySpelling.sb2* и проиграйте его несколько раз, чтобы разобраться, как он работает. Получится ли у вас изменить игру так, чтобы «испорченное» слово содержало две дополнительные буквы, а не одну? FixMySpelling.sb2

Расшифровка

Вот еще одна игра в слова, но немного посложнее. Мы возьмем обычное слово, перепутаем в нем буквы и попросим пользователя восстановить изначальное слово. Unscramble.sb2

Начнем с создания процедуры, которая будет переставлять символы строки в случайном порядке. Вызывающая программа задает введенную строку (стрВв), а процедура **Расставить в случайном порядке** модифицирует ее так, чтобы символы строки поменялись местами, как показано на рис. 8.10.

Шаг	Случайный выбор	Добавить к стр1	Удалить из стрВв
1	3	в	абгдеж
2	2	вб	агдеж
3	1	вба	гдеж
4	2	вбад	геж
…	…	…	…

Рис. 8.10. Процедура **Расставить в случайном порядке**

Мы выберем случайную букву из стрВв и добавим эту букву во *временную* строку стр1: она будет местом хранения переставленных букв. Удалим выбранную букву из стрВв, чтобы не использовать ее повторно, и повторим все действия, пока в стрВв не закончатся буквы. Процедура **Расставить в случайном порядке** выполняет эти шаги, как показано на рис. 8.11.

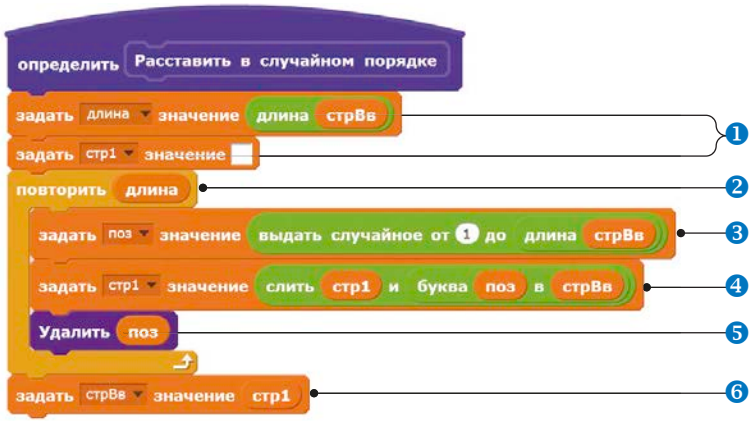


Рис. 8.11. Процедура **Расставить в случайном порядке**

Сначала процедура задает значение длина, равное длине введенной строки, стрВв, и временную строку стр1 пустой ❶. Затем она проходит цикл **повторить**, чтобы создать слово с «перемешанными» буквами ❷. Количество повторов равно длине введенной строки. При каждом повторе выбирается случайная позиция в стрВв ❸, и эта буква добавляется к стр1 ❹. Мы используем переменную длины на шаге ❸, поскольку стрВв и ее длина будут меняться во время исполнения цикла. После этого мы вызовем процедуру **Удалить**, чтобы с ее помощью удалить символ стрВв, который мы только что использовали ❺. Когда цикл повтора закончит перемешивать буквы, значением стр1 будет слово с перепутанным расположением букв ❻.

Процедура **Удалить**, которая позволяет добавить букву к новому слову только один раз, показана на рис. 8.12. Она убирает символ стрВв с позиции, которую вы задаете с помощью переменной симвПоз.

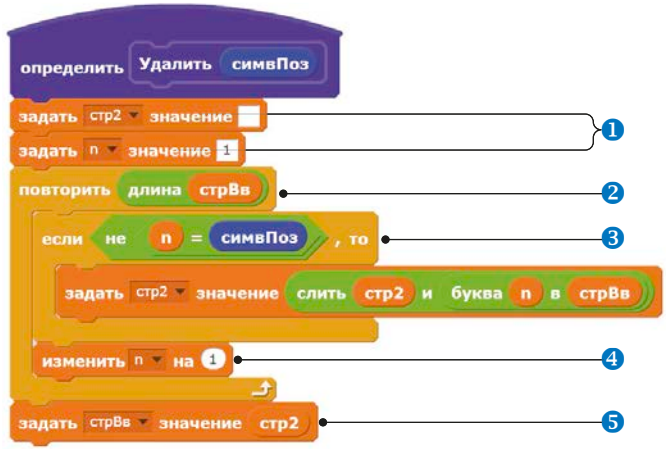


Рис. 8.12. Процедура **Удалить**

Эта процедура использует другую временную строку с именем стр2 для создания нужной нам новой строки. Вначале стр2 задается как пустая. Еще задается счетчик цикла n со значением 1, чтобы начать обработку первого символа стрВв ❶. Затем процедура начинает цикл, чтобы создать строку вывода ❷. Если мы не хотим удалять текущий символ, мы добавляем его в стр2 ❸. Значение счетчика цикла увеличивается на единицу, и можно перейти к следующей букве стрВв ❹. Когда процедура заканчивает работу, в стрВв вносится новое слово стр2 ❺.

Теперь рассмотрим основной скрипт игры, приведенный на рис. 8.13.

Скрипт случайным образом выбирает слово из списка и сохраняет его в переменной ввСлово ❶. Затем устанавливает значение стрВв равным ввСлово ❷ и вызывает **Расставить в случайном порядке**, чтобы поменять местами символы стрВв ❸. После этого скрипт начинает цикл, чтобы получить ответ пользователя ❹. В цикле скрипт просит игрока ввести правильно написанное слово ❺ и использует блок **если/иначе** для проверки ответа ❻. Эта часть ничем не отличается от того, что мы делали в игре «Исправь ошибки».

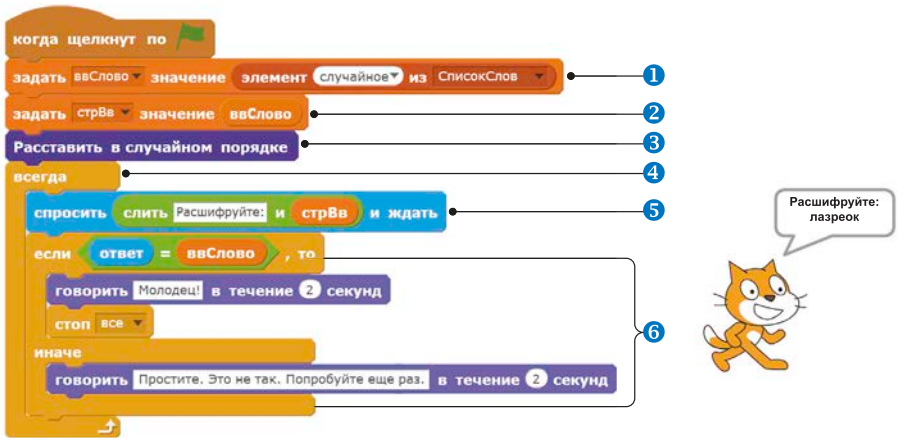


Рис. 8.13. Основной скрипт игры «Расшифровка»

Все эти примеры — небольшой набор операций, которые вы можете производить со строками, чтобы решать интересные задачки.

Проекты Scratch

Процедуры, которые мы только что рассмотрели, демонстрируют базовые принципы обработки строк. В этом разделе мы используем приобретенные знания для создания практических приложений. В процессе работы вы познакомитесь с новыми приемами программирования, которые сможете использовать в собственных проектах.

Охота

В этой игре в забавной и привлекательной форме реализуется идея относительного движения. Цель — оценить угол поворота и расстояние между двумя объектами, расположенными на **Сцене**. Интерфейс пользователя приведен на рис. 8.14.

Shoot.sb2

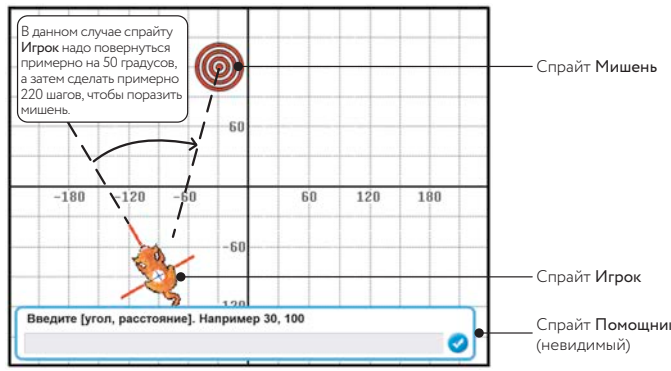


Рис. 8.14. Интерфейс пользователя для игры «Охота»

Когда игра начинается, спрайты Игрок и Мишень (Player и Target) случайным образом размещаются на **Сцене**. Затем игра просит пользователя оценить угол поворота и расстояние, на которое спрайту Игрок нужно переместиться, чтобы поразить Мишень. Игрок передвигается согласно этим цифрам. Если спрайт останавливается, не доходя до Мишени расстояние меньше заданного, пользователь выигрывает. В противном случае спрайт Игрок возвращается на исходную позицию и пользователь может попробовать еще раз. После щелчка по иконке с зеленым флажком спрайт Игрок выполняет скрипт, показанный на рис. 8.15.

Рис. 8.15. Скрипт спрайта Игрок, который начинает выполняться после клика по иконке с зеленым флажком

Скрипт передает сообщение **НоваяИгра**, после чего спрайт Помощник (Helper) присваивает новые местоположения спрайтам Игрок и Мишень ❶. Спрайт Помощник выполняет простую процедуру (она не показана),

которая обновляет пять следующих переменных на случайно выбранные числа, чтобы спрайты Игрок и Мишень были видимы (и находились на определенном расстоянии друг от друга) на **Сцене**.

ХИгрок и УИгрок	*x*- и *y*-координаты спрайта Игрок
ХМишень и УМишень	*x*- и *y*-координаты спрайта Мишень
изнУгол	Изначальное направление спрайта Игрок

Как только у скрипта появились новые позиции для Игрока и Мишени, он передает **СтартИгры**, чтобы передвинуть Мишень в новое местоположение ❷ (скрипт для спрайта Мишень здесь не показан). Затем скрипт вводит бесконечный цикл, чтобы дать пользователю несколько попыток поразить мишень ❸. Цикл будет прекращен командой **стоп все** (в процедуре **ПроверитьОтветы**), когда игрок попадет в мишень.

При каждой итерации цикла задается изначальное положение и направление спрайта Игрок, а **Сцена** очищается от всех линий, оставленных пером во время предыдущей попытки ❹. Затем скрипт передает **ПолучитьОтветы** ❺, на что спрайт Помощник предлагает пользователю ввести ответ, как показано на рис. 8.16. Затем Помощник делит ответ на две части (до запятой и после) и соответственно обновляет угол и расстояние.

Игрок делает ход в направлении, указанном пользователем, с опущенным пером ❻. Он чертит маршрут своего движения, и пользователь может использовать эту линию, чтобы при следующей попытке уточнить свою догадку.

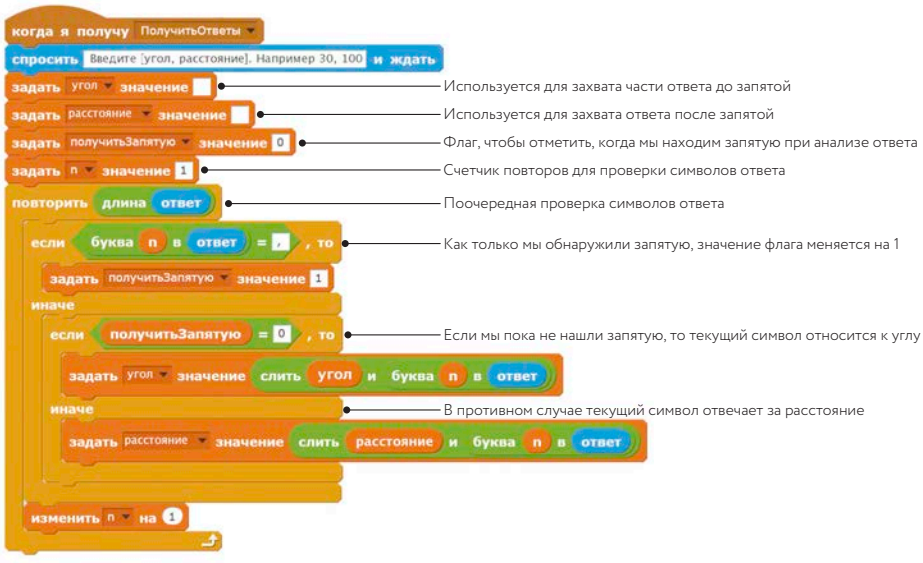


Рис. 8.16. Скрипт **ПолучитьОтветы**

Наконец спрайт Игрок выполняет процедуру **ПроверитьОтветы**, чтобы узнать, достаточно ли он близок к мишени. Игра завершается в том случае, если спрайт Игрок окажется совсем рядом с целью. На рис. 8.17 показано, как Игрок проверяет расстояние до мишени.

Спрайт Игрок использует блок **расстояние до**, чтобы проверить, насколько близко он подошел к спрайту Мишень. Если расстояние меньше 20 шагов, то в игре это считается попаданием и программа выдает «Ты выиграл!». Иначе считается, что пользователь промахнулся, и цикл **всегда** начинается снова, чтобы дать еще один шанс.

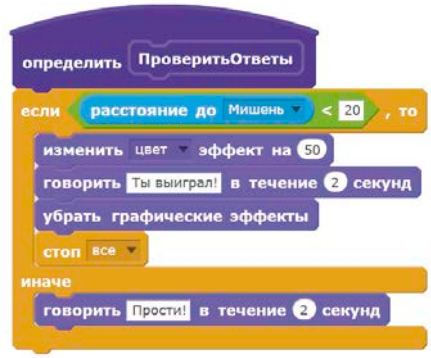


Рис. 8.17. Процедура **ПроверитьОтветы** *спрайта* Игрок

УПРАЖНЕНИЕ 8.4

Усовершенствуйте игру «Охота» так, чтобы она отслеживала количество попыток попасть в цель и вела счет для пользователя.

Преобразователь двоичных чисел в десятичные

BinaryToDecimal.sb2

Двоичные числа (в системе счисления с основанием 2) могут содержать только две цифры: 0 и 1. Именно двоичными числами оперирует большинство компьютеров. А люди предпочитают десятичные (в системе счисления с основанием 10). В этом разделе вы создадите приложение, которое переводит двоичные числа в их десятичные эквиваленты. Позже вы можете пользоваться этом преобразователем, чтобы проверить, насколько точно вы сами способны совершать такие операции в уме.

Для начала посмотрим, как переводить двоичные числа в десятичные. На рис. 8.18 приведен пример преобразования двоичного числа 100110011.

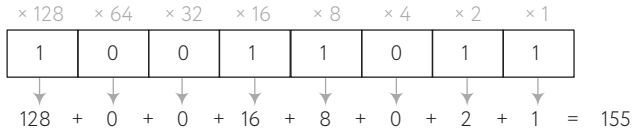


Рис. 8.18. Преобразование двоичного числа в десятичное

Нам нужно умножить каждую цифру двоичного числа на ее позиционную величину и сложить полученные результаты. *Позиционные величины* возрастают соответственно увеличению степени основания справа налево, причем самая правая позиция имеет степень 0. Поскольку двоичное основание — 2, самая правая цифра имеет позиционную величину $2^0 = 1$, так что цифра, оказавшаяся в этой позиции, умножается на 1. Вторая с конца цифра умножается на $2^1 = 2$, третья с конца — на $2^2 = 4$ и т. д.

На рис. 8.19 изображен интерфейс пользователя для преобразователя двоичных чисел в десятичные. Программа просит ввести восьмиразрядное двоичное число, отображает его на **Сцене** с помощью спрайта Разряд (Bit), который использует два костюма — Ноль и Единица (Off и On). Программа также рассчитывает соответствующее десятичное число, и спрайт Драйвер (Driver) в костюме, изображающем компьютер, показывает пользователю результат.

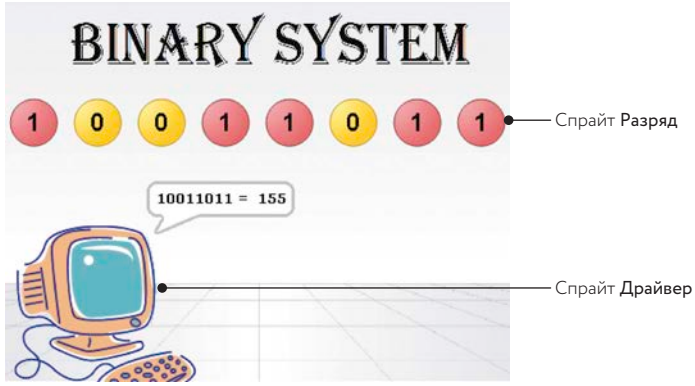


Рис. 8.19. Программа для преобразования двоичных чисел в десятичные

УПРАЖНЕНИЕ 8.5

Чтобы проверить себя, попрактикуйтесь с переводом двоичных чисел в десятичные на следующих числах: 1010100, 1101001, 1100001.

Программа начинает работать после щелчка по иконке с зеленым флажком. Это событие прерывается спрайтом Драйвер, который выполняет скрипт, описанный и показанный на рис. 8.20.

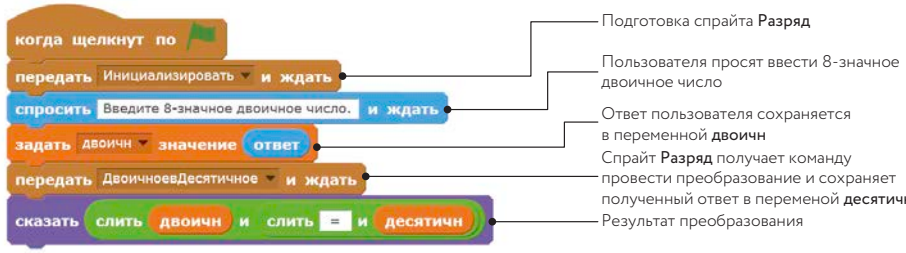


Рис. 8.20. Скрипт спрайта Драйвер

Этот скрипт подготавливает **Сцену** и просит пользователя ввести двоичное число, чтобы спрайт Разряд мог начать преобразование. Когда Разряд заканчивает работу, Драйвер показывает пользователю десятичное число, которое рассчитано и сохранено спрайтом Разряд в общей переменной двоичн.

Скрипт, который спрайт Разряд выполняет в ответ на сообщение **Инициализировать**, приведен на рис. 8.21.

Рис. 8.21. Скрипт **Инициализировать** *для спрайта* Разряд

Этот скрипт рисует на **Сцене** восемь знаков для восьми разрядов числа в виде восьми нолей. Как только в строке ввода пользователя появляется единица, скрипт печатает костюм цифры 1 в соответствующем разряде. Когда пользователь заканчивает ввод двоичного числа для преобразования в десятичное, спрайт Разряд получает сообщение **ДвоичноевДесятичное** и выполняет скрипт, приведенный на рис. 8.22.

Первым делом процедура преобразования инициализирует все переменные, которые собирается использовать ❶:

- длина — количество разрядов в двоичном числе пользователя;

- поз указывает на последнюю цифру введенного числа;
- вес вставляется как позиционная величина последней цифры двоичного числа;
- десятичн устанавливается равной 0, но в конце она будет содержать результат преобразования;
- хПоз устанавливается по координате *x* изображения последней цифры.

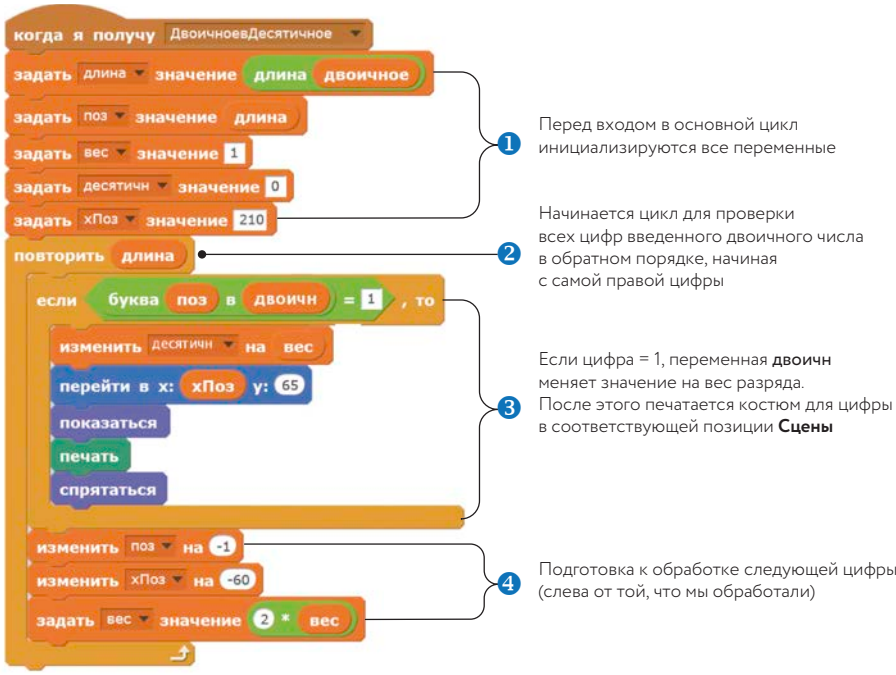


Рис. 8.22. Скрипт **ДвоичноевДесятичное** *спрайта* Разряд

В цикле **повторить** ❷ процедура проверяет каждую цифру, чтобы узнать, 1 это или 0. Если цикл находит единицу ❸, он добавляет текущую величину переменной вес к десятичн и печатает костюм цифры 1 поверх изображения с цифрой 0.

В конце цикла скрипт обновляет несколько переменных, прежде чем перейти к следующей итерации:

- поз обновляется, указывая на цифру слева от только что обработанной;

- хПоз меняется так, чтобы располагаться в центре изображения следующей цифры на случай, если нам нужно запечатать еще одно изображение;
- вес умножается на 2: данная переменная будет при последующих итерациях цикла принимать значения 1, 2, 4, 8, 16 и т. д.

УПРАЖНЕНИЕ 8.6

Сделайте так, чтобы спрайт Драйвер проверял правильность числа, вводимого пользователем, прежде чем передать сообщение **Двоичное-вДесятичное** спрайту Разряд. Вам нужно удостовериться в том, что число, введенное пользователем, является двоичным (никаких других цифр, кроме 1 и 0, в нем нет); длина введенного слова не превышает 8 цифр.

Виселица

Hangman.sb2

В этом разделе мы будем писать классическую игру «Виселица». На рис. 8.23 показано, как она выглядит.

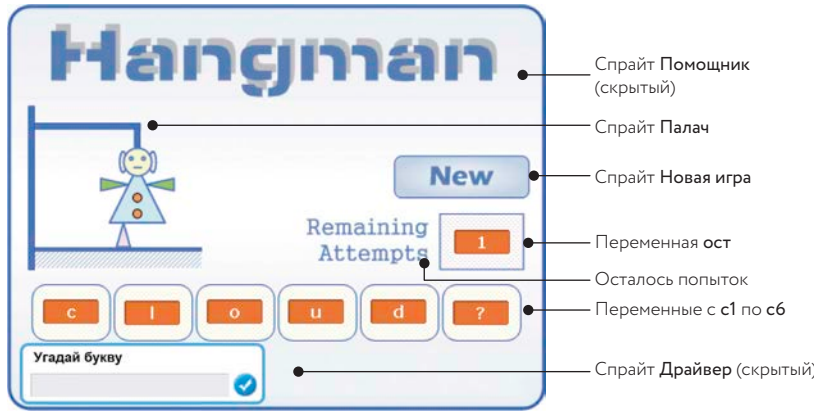


Рис. 8.23. Интерфейс пользователя для игры «Виселица»

Программа загадывает случайное слово из шести букв, но вместо букв показывает вопросительные знаки. У пользователя есть восемь попыток угадать буквы. Если он называет букву правильно, программа показывает ее в слове столько раз, сколько она в нем используется. Иначе она показывает очередную часть тела фигурки повешенного (голова, тело, левая рука и т. д.). Если за восемь попыток слово не угадано, программа дорисовывает повешенного и пользователь проигрывает. Если игрок угадал загаданное слово за восемь попыток или быстрее, он побеждает. У приложения есть четыре спрайта.

Драйвер (Driver). Этот спрайт скрывается, когда начинается игра. Он предлагает пользователю ввести буквы и обрабатывает ответы. Когда игра заканчивается, этот спрайт показывает один из двух следующих костюмов:

Палач (Hangman). Этот спрайт выводит меняющееся изображение повешенного. Всего у него девять костюмов, каждый последующий показывает на одну часть тела больше (рис. 8.24).

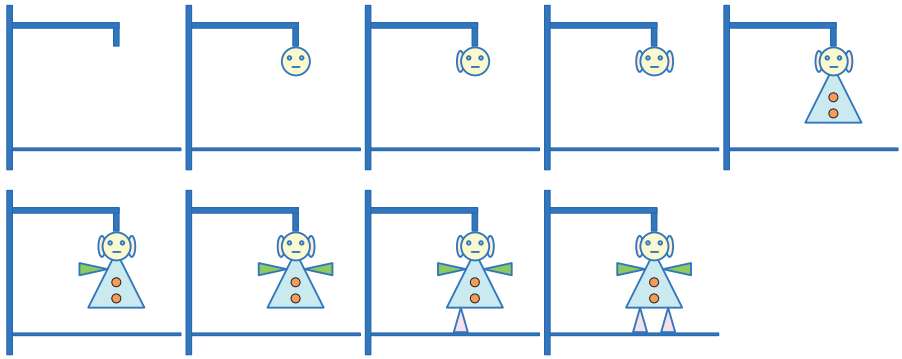


Рис. 8.24. Девять костюмов спрайта Палач

Новая игра (New). Этот спрайт отображает кнопку Новая игра на **Сцене**.

Помощник (Helper). Этот невидимый спрайт отображает буквы, которые игрок угадал, а также количество оставшихся попыток. Он использует семь переменных с мониторами в виде больших дисплеев, размещенных в нужных местах **Сцены**. Использование отдельного спрайта для обновления дисплея отделяет логику игры от интерфейса пользователя. Вы можете, например, изменить спрайт так, чтобы показывать больше букв на **Сцене**, не влияя на остальную часть приложения.

Когда игрок нажимает кнопку Новая игра, то есть активирует спрайт Новая игра, тот передает сообщение **НоваяИгра** спрайту Драйвер, чтобы известить его о начале нового раунда. Когда Драйвер получает это сообщение, он выполняет скрипт, приведенный на рис. 8.25.

Скрипт возвращает в исходное состояние интерфейс пользователя ❶ и начинает цикл ❷, обрабатывающий ввод информации о букве. Другая процедура, вызываемая спрайтом Драйвер, прекратит этот цикл с помощью блока **стоп все**, как только выполнится условие окончания игры.

Рис. 8.25. Скрипт **НоваяИгра** *спрайта* Драйвер

Во время каждой итерации цикла спрайт Драйвер просит пользователя угадать букву и ждет ее ввода ❸. Когда пользователь вводит букву, скрипт вызывает процедуру **ОбработатьОтвет**, которая обновит флаг (с именем естьБуква) ❹, чтобы указать, правильна ли догадка.

Когда **ОбработатьОтвет** возвращается в точку вызова, скрипт проверяет флаг естьБуква ❹ и, исходя из того, правильно игрок угадал или нет, действует дальше. Процедуры, которые после этого вызывает **НоваяИгра**, начиная со скриптов, показаны на рис. 8.26.

Рис. 8.26. Скрипты, которые запускаются процедурой **Инициализировать**

Драйвер прячется, инициализирует значение отобразитьСлово как шесть вопросительных знаков и задает значение остПопыток равным 8. Затем спрайт выбирает загаданноеСлово из заранее подготовленного списка шестибуквенных слов. После этого процедура передает **Обновить**, и спрайт Помощник приписывает правильные значения своим переменным (тем, чьи мониторы видны на **Сцене**). Последняя инструкция — передать сообщение **Сбросить** спрайту Палач. Когда Палач

его получает, он переключается на костюм старт, который изображает пустую виселицу.

Рассмотрим простой пример, который поможет нам понять, что делает процедура **ОбработатьОтвет** (см. рис. 8.27). Предположим, загаданное слово — «думать» и сейчас первый раунд игры (значение переменной отобразитьСлово — ??????). Если первая буква, которую предлагает пользователь, — «м», **ОбработатьОтвет** должна задать значение переменной естьБуква равным 1, чтобы указать на успешность догадки, задать значение переменной отобразитьСлово как «??м???», чтобы указать на положение отгаданной буквы в слове, задать значение переменной вопрКол (количество знаков вопроса в обновленной строке, отображаемой на дисплее) равным 5. Когда значение переменной вопрКол достигнет 0, это будет означать, что игрок отгадал все буквы в загаданном слове. **ОбработатьОтвет** принадлежит спрайту Драйвер, и весь скрипт целиком показан на рис. 8.27 (слева).

ОбработатьОтвет начинается со сброса флага естьБуква и вопрКол до 0. Эта процедура будет увеличивать переменную вопрКол на 1 для каждой неотгаданной буквы в слове. Временная переменная врем, используемая для конструирования отображаемой на дисплее строки после каждой попытки, инициализируется как пустая строка. Переменная поз используется как счетчик цикла.

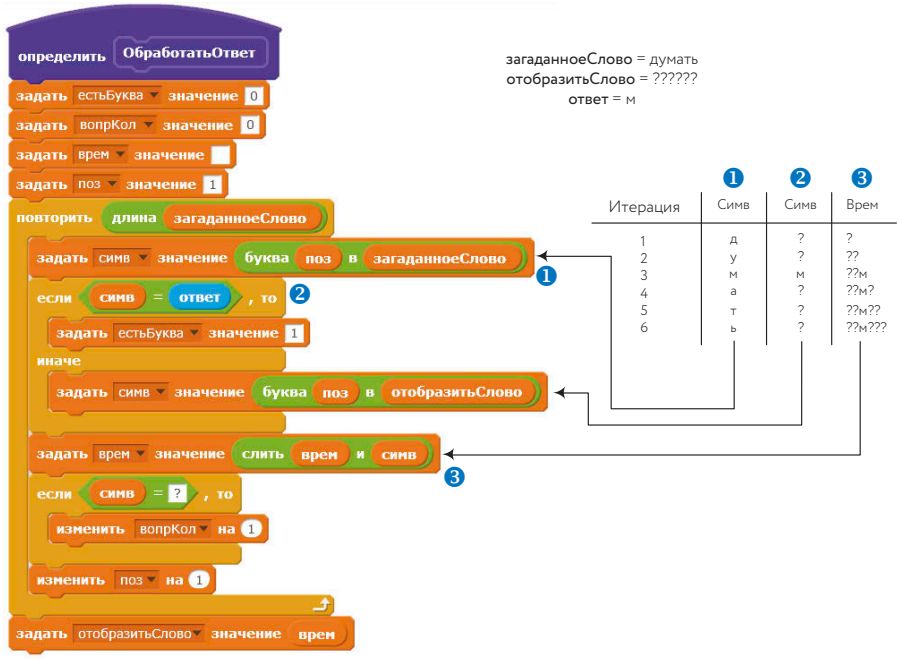


Рис. 8.27. Процедура **ОбработатьОтвет**

Цикл проверяет каждую букву переменной загаданноеСлово, используя поз в качестве индекса. Если проверяемая буква (сохраненная в переменной симв) совпадает с предложенной пользователем (она сохраняется во встроенной переменной Scratch ответ), флагу естьБуква присваивается значение 1. Иначе значение переменной симв устанавливается как буква в соответствующей позиции в переменной отобразитьСлово. В любом случае скрипт добавит переменную симв в конец переменной врем, как показано на рис. 8.27 (справа).

Когда цикл прекращается, переменная отобразитьСлово содержит шесть букв, которые должны быть отображены на **Сцене**, с учетом последних попыток пользователя. Цикл также отслеживает количество вопросительных знаков в выведенной на дисплее строке. Если их нет, то пользователь верно угадал слово.

Когда процедура **ОбработатьОтвет** возвращается в точку вызова, обработчик сообщения **НоваяИгра** проверяет переменную естьБуква, чтобы узнать, правильна ли догадка игрока. Если нет, он вызывает процедуру **ОбработатьНеверныйОтвет,** показанную на рис. 8.28.

Эта процедура передает сообщение **НевернаяДогадка**, информируя спрайт Палач, чтобы он показал свой следующий костюм, а затем уменьшает количество попыток на 1. Если у пользователя закончились попытки, скрипт показывает загаданное слово и заканчивает игру. Иначе он передает сообщение **Обновить**, чтобы показать, сколько попыток осталось.

Рис. 8.28. Процедура **ОбработатьНеверныйОтвет**

Если пользователь ввел букву, которая есть в слове, вместо процедуры **ОбработатьНеверныйОтвет** должна быть вызвана процедура **ОбработатьВерныйОтвет**, показанная на рис. 8.29.

Рис. 8.29. Процедура **ОбработатьВерныйОтвет**

Процедура **ОбработатьВерныйОтвет** передает сообщение **Обновить**, чтобы отобразить букву, правильно угаданную игроком. Затем процедура проверяет значение переменной вопрКол. Если оно равно 0, то пользователь угадал все буквы. Спрайт Драйвер показывает костюм победа и заканчивает игру.

УПРАЖНЕНИЕ 8.7

Программа «Виселица» не проверяет, что вводит пользователь, и может получиться так, что игрок ввел слово или не букву, а какой-то другой символ. Усовершенствуйте программу так, чтобы она отклоняла недопустимые операции при вводе.

Учим дроби

FractionTutor.sb2

Сейчас мы создадим образовательную программу, которая обучает вычислению дробей. Интерфейс приведен на рис. 8.30. Пользователь может выбрать математическую операцию (сложение, вычитание, умножение или деление) и щелкнуть по кнопке Новая задача, чтобы решить очередную математическую задачу. Когда пользователь вводит ответ и щелкает по кнопке Проверка, спрайт Учитель (в виде учительницы) проверяет ответ и выдает обратную связь в виде сообщения.

В этом приложении шесть спрайтов. Спрайт Операция (Operation) позволяет игроку выбирать между четырьмя математическими операциями. Спрайт Читать (Read) показывает кнопку ввода ответа. Спрайты Новая задача (New) и Проверка (Check) показывают соответствующие кнопки. Спрайт Учитель (Teacher) проверяет ответ игрока, а спрайт-невидимка по имени Цифра (Digit) печатает числа, которые соответствуют задаче, на **Сцене**.

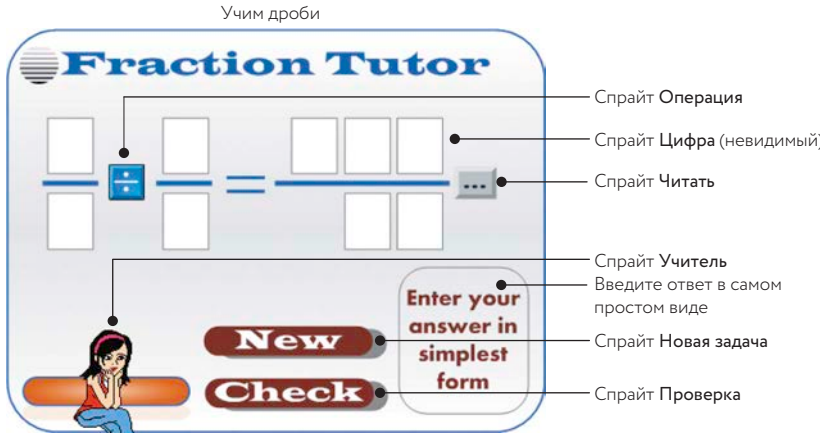


Рис. 8.30. Интерфейс пользователя для приложения «Учим дроби»

Когда пользователь щелкает по спрайту Новая задача (кнопка с тем же именем), запускается скрипт, показанный на рис. 8.31. Скрипт задает случайные значения от 1 до 9 числителю и знаменателю обоих операндов, представленных четырьмя переменными числ1, знам1, числ2 и знам2. Затем скрипт передает сообщение **НоваяЗадача** спрайту Цифра, чтобы тот показал эти числа на **Сцене**.

Рис. 8.31. Скрипт спрайта Новая задача

Спрайт Цифра имеет 12 костюмов (с именами от ц1 до ц12), как показано справа на рис. 8.32. Когда этот спрайт получает сообщение **НоваяЗадача**, он печатает костюмы, представляющие собой числители и знаменатели обоих операндов. На рис. 8.32 вы также можете увидеть саму процедуру вывода.

Процедура использует вложенные блоки **если/иначе**, чтобы определить соответствие между костюмом и цифрой, которая должна быть напечатана. Имена костюмов для цифр с 1 по 9 создаются с помощью оператора **слить**. После переключения на нужный костюм спрайт Цифра переходит на указанную позицию (x, y) и печатает изображение костюма в этом месте.

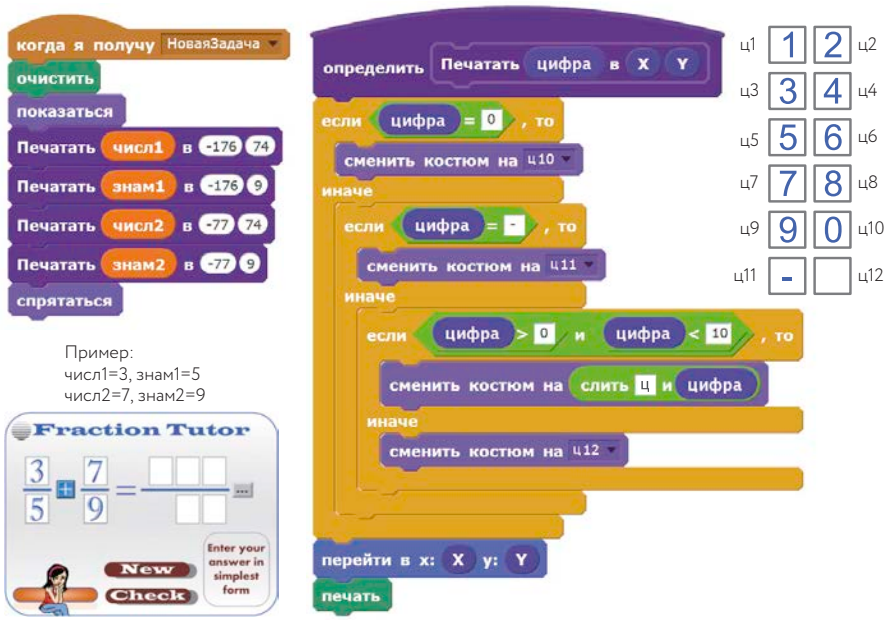


Рис. 8.32. Функционал спрайта Цифра

После вывода новой задачи пользователь может щелкнуть по кнопке Читать, чтобы ввести ответ. Скрипт, связанный с этой кнопкой, представлен на рис. 8.33. Часть скрипта, которая разрезает ответ игрока на две части (числитель и знаменатель), похожа на ту, что представлена на рис. 8.16, где в игре «Охота» надо разделить «ответ» на «угол» и «расстояние», и поэтому здесь не приводится. Если вы хотите увидеть всю процедуру целиком, загляните в файл *FractionTutor.sb2*.

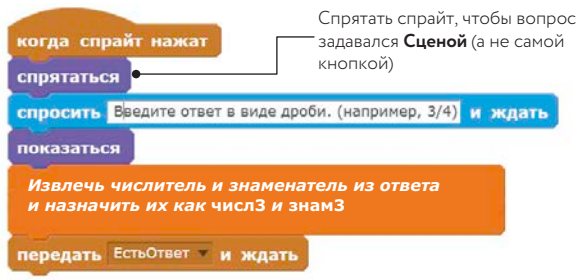
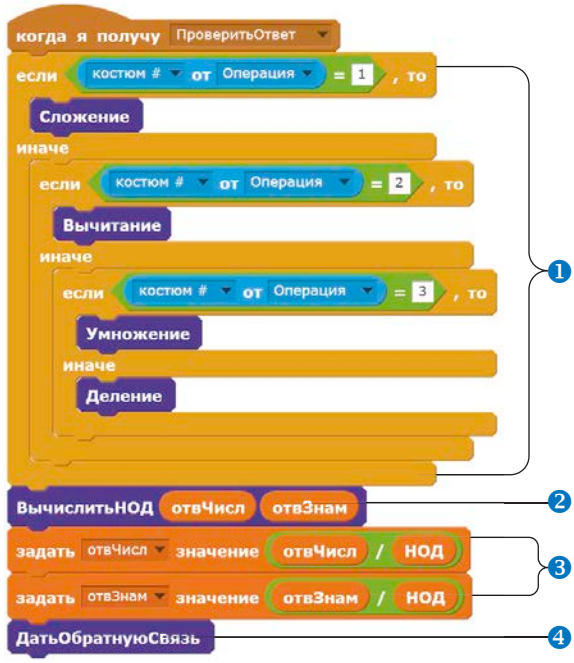


Рис. 8.33. Скрипт спрайта Читать

Сначала пользователя просят ввести ответ в виде дроби (например, 3/5 или –7/8). Затем скрипт извлекает из строки ответа числитель

и знаменатель (они разделены знаком «/») и приписывает их величины переменным числ3 и знам3 соответственно. Например, если пользователь вводит –23/15, значение числ3 будет –23, а знам3 — 15. После этого скрипт передает сообщение **ЕстьОтвет** спрайту Цифра, чтобы тот отобразил ответ пользователя на **Сцене**. Спрайт Цифра, получив это сообщение, печатает цифры переменных числ3 и знам3 в соответствующих местах **Сцены** точно так же, как он отображал числители и знаменатели операндов. Если вы хотите ознакомиться с этой процедурой подробнее, обратитесь к файлу *FractionTutor.sb2*.

Введя ответ, пользователь может щелкнуть по кнопке Проверка, чтобы узнать, правильно он ответил или нет. Скрипт спрайта Проверка передает сообщение **ПроверитьОтвет**, информируя остальные спрайты о запросе пользователя. Это сообщение захватывается и обрабатывается спрайтом Учитель, который будет выполнять скрипт, показанный на рис. 8.34.



1. Вычисление правильного ответа на основе текущей операции, которая определяется номером костюма спрайта **Операция**. Четыре математические операции (**Сложение**, **Вычитание**, **Умножение** и **Деление**) вычисляют числитель (**отвЧисл**) и знаменатель (**отвЗнам**) правильного ответа
2. Расчет наибольшего общего делителя (НОД) **отвЧисл** и **отвЗнам**, чтобы вывести ответ в самом простом виде
3. Уменьшение **отвЧисл** и **отвЗнам** путем деления их на наибольший общий делитель (НОД)
4. Обратная связь для пользователя

Рис. 8.34. Скрипт **ПроверитьОтвет**

Текущий костюм спрайта Операция информирует о том, какая именно математическая операция (**Сложение**, **Вычитание**, **Умножение** или **Деление**) должна быть выполнена ❶. Операции используют переменные числ1, знам1, числ2 и знам2 как ввод и задают значения величин

отвЧисл и отвЗнам, которые, соответственно, представляют правильный числитель и знаменатель ответа. Все четыре процедуры приведены на рис. 8.35.

Когда ответ вычислен, процедура **ПроверитьОтвет** должна представить его в самом простом виде. Например, дробь 2/4 должна быть упрощена до 1/2. Чтобы выполнить упрощение, скрипт сначала находит наибольший общий делитель числителя и знаменателя ❷. Рассмотрим эту процедуру.

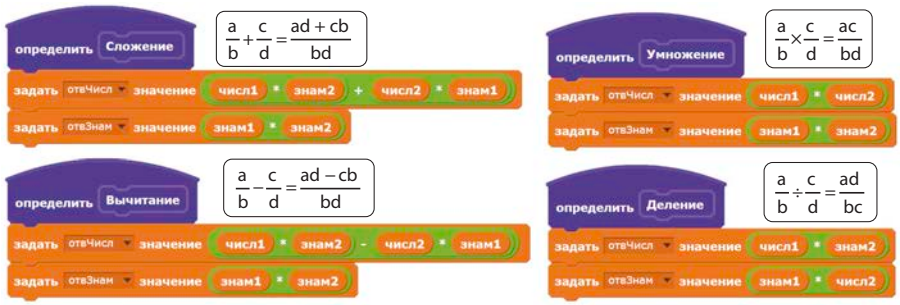


Рис. 8.35. Процедуры **Сложение**, **Вычитание**, **Умножение** *и* **Деление** *спрайта* Учитель

После вычисления НОД скрипт делит отвЧисл и отвЗнам на НОД ❸ и вызывает процедуру **ДатьОбратнуюСвязь** ❹, чтобы показать пользователю, правильный его ответ или нет.

Внимательнее рассмотрим эти процедуры. Начнем с четырех математических операций, которые показаны на рис. 8.35. Они рассчитывают результат математической операции следующим образом:

$$\frac{\text{числ1}}{\text{знам1}}\left[+,-,\times,\div\right]\frac{\text{числ2}}{\text{знам2}}=\frac{\text{отвЧисл}}{\text{отвЗнам}}$$

Потом они сохраняют результат в виде двух переменных (отвЧисл и отвЗнам), соответствующих числителю и знаменателю ответа.

Теперь перейдем к процедуре **ВычислитьНОД**, показанной на рис. 8.36.

Рассмотрим операцию **ВычислитьНОД** на примере. Предположим, числ1 равна –10, а числ2 — 6. Нам нужно вычислить наибольшее целое положительное число, на которые числ1 и числ2 делятся без остатка. Процедура начинается с того, что задается НОД — наименьшая абсолютная величина, или модуль обоих чисел. В нашем примере это 6. Затем цикл проверяет числа 6, 5, 4 и т. д., пока не найдет число, на которое и числ1, и числ2 делятся без остатка. Это-то нам и нужно. В нашем примере величина НОД будет равна 2.

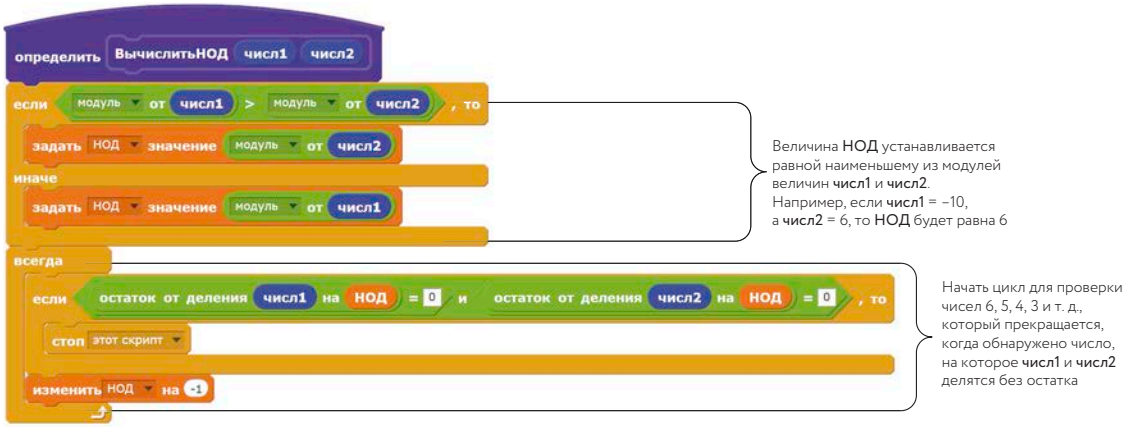
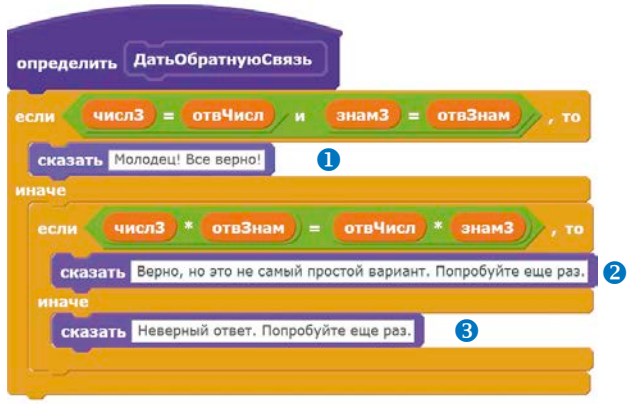


Рис. 8.36. Процедура **ВычислитьНОД** *спрайта* Учитель

Последняя процедура, которую мы рассмотрим, — **ДатьОбратнуюСвязь**. Она сравнивает ответ пользователя с правильным и отображает соответствующее высказывание, как показано на рис. 8.37. На рисунке также приведены примеры, которые показывают, как в разных случаях работает структура **если/иначе**.



	Примеры	
	Верный ответ $\frac{\text{отвЧисл}}{\text{отвЗнам}}$	Ответ пользователя $\frac{\text{числ3}}{\text{знам3}}$
1	$\frac{3}{4}$	$\frac{3}{4}$
2	$\frac{3}{4}$	$\frac{6}{8}$
3	$\frac{3}{4}$	$\frac{2}{3}$

Рис. 8.37. Процедура **ДатьОбратнуюСвязь** *спрайта* Учитель

УПРАЖНЕНИЕ 8.8

Усовершенствуйте программу обучения работе с дробями так, чтобы она вела счет правильным и неправильным ответам. Разработайте схему ведения счета и показа его пользователю.

Итоги

Обработка строк — важный навык программирования. Из этой главы вы узнали, как работать с отдельными символами строки: комбинировать, сравнивать, удалять и менять их местами.

Для начала мы подробно рассмотрели строковые данные и то, как они сохраняются: как последовательность символов. Затем мы написали несколько процедур, которые демонстрируют базовые приемы манипулирования строкой. Потом мы использовали эти приемы, чтобы написать несколько интересных и полезных приложений. Идеи, развиваемые в этих программах, могут применяться во многих других областях, и я очень надеюсь, что они помогут вам в работе над вашими проектами.

Из следующей главы вы узнаете о списках и о том, как они используются, чтобы хранить сразу несколько величин и манипулировать ими. Овладев этой структурой данных, вы получите все инструменты, необходимые для того, чтобы писать профессиональные программы в среде Scratch.

Задания

1. Напишите программу, которая просит пользователя ввести слово, а затем произносит его *n* раз, где *n* — количество букв во введенном слове.

2. Напишите программу, которая просит пользователя ввести слово, а затем определяет, сколько раз в нем встречается буква «а».

3. Напишите программу, которая, получив от пользователя одну букву (от «а» до «я»), выдает номер места этой буквы в алфавите (а = 1, б = 2, в = 3 и т. д.). Программа не должна видеть разницы между буквами верхнего и нижнего регистра (задайте переменную **альфа**, содержащую все буквы алфавита, как мы сделали на рис. 8.9, а затем примените цикл, чтобы определить позицию введенной буквы внутри этой переменной).

4. Напишите программу, которая просит пользователя ввести какую-нибудь букву алфавита, а затем показывает букву, которая ей предшествует (используйте тот же прием, что и в предыдущей задаче).

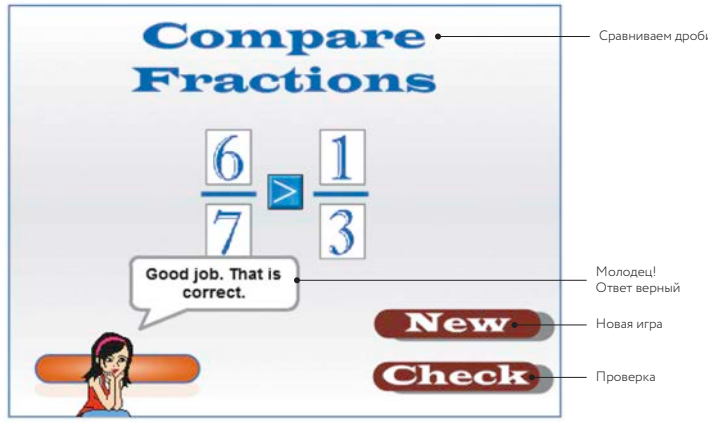
5. Напишите программу, которая, получив от пользователя положительное целое число, рассчитывает и отображает сумму всех его цифр. Например, если пользователь вводит 3582, то программа должна дать ответ 18 (3 + 5 + 8 + 2).

6. Напишите программу, которая, получив от пользователя слово, отображает его буквы в обратном порядке, используя блок **сказать**.

7. Напишите программу, которая, получив от пользователя число, вставляет пробелы между всеми его цифрами. Например, если введенное число — 1234, то выводимая строка будет 1 2 3 4 (создайте переменную вывода, сливая отдельные символы введенного числа с пробелами).

CompareFractions.sb2

8. Создайте игру, в которой пользователи будут сравнивать дробные числа. Интерфейс пользователя показан ниже. Когда нажата кнопка Новая игра, программа случайным образом выбирает две дроби для сравнения. Пользователь выбирает знаки «меньше» (<), «больше» (>) или «равно» (=), щелкая по кнопке оператора. Когда пользователь щелкает по кнопке **Проверить**, игра проверяет ответ и дает игроку обратную связь. Откройте файл *CompareFractions.sb2* и добавьте скрипты, которых не хватает в игре.



9

СПИСКИ

В программах, которые мы до сих пор писали, для хранения единичного фрагмента информации использовались обычные переменные. Но они неприменимы, когда надо сохранить несколько величин, например телефонные номера друзей, названия книг или информацию о ежедневной температуре за месяц.

Если вам хочется, чтобы ваша программа запомнила телефонные номера двадцати ваших друзей, понадобится 20 переменных! И уж создание такой громоздкой программы веселым делом не назовешь. В этой главе мы познакомимся с еще одним типом встроенных данных: *списками*. Это удобный способ группировать однородные величины. Вот темы, которые мы будем разбирать в этой главе:

- как создавать списки и управлять ими;
- как инициализировать отдельные элементы списка и получать к ним доступ;
- каковы базовые приемы сортировки и поиска.

Для начала расскажу, как создавать в Scratch списки, покажу команды, которые используются при работе с ними, и объясню, как добавлять в списки данные, введенные пользователем. Затем мы обсудим нумерованные списки и обычные операции, которые к ним применяются, например определение минимальной, максимальной и среднеарифметической величины их элементов. Потом мы изучим алгоритм сортировки элементов в списке. И наконец, мы создадим несколько программ, чтобы понять, как применять списки на практике.

Списки в Scratch

Список похож на контейнер, в который можно поместить несколько величин и достать при необходимости. Или комод, в каждом выдвижном ящике которого хранится один предмет. Создавая список, вы даете ему такое же имя, как переменной. До каждого его элемента вы можете добраться, используя его собственную позицию в списке. На рис. 9.1 показан список с именем днейСписок, в котором хранятся названия дней недели.

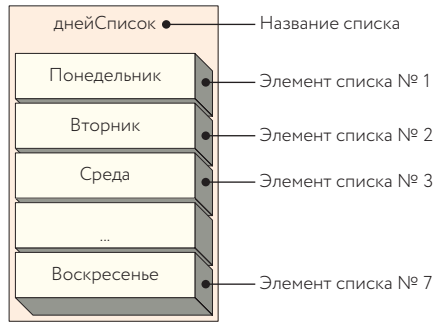


Рис. 9.1. Список, в котором содержатся названия дней недели

Можно ссылаться на элементы списка с помощью *числового показателя* (или позиции). В Scratch первый элемент имеет числовой показатель 1, второй — 2 и т. д. Например, если в списке дней недели вторник оказался вторым, то его числовой показатель будет 2. Так что можно ссылаться на третий элемент днейСписок, используя команду вида «элемент 3 списка днейСписок».

А теперь создадим списки в Scratch. Также мы рассмотрим команды, которые позволяют нам управлять списками в наших программах, и узнаем, как Scratch реагирует на ошибочные команды по управлению списками.

Создание списков

Создание списка мало чем отличается от создания переменной. Выберите раздел **Данные** и нажмите кнопку **Создать список**. Появится диалоговое окно, воспроизведенное справа на рис. 9.2. Введите имя списка (используйте днейСписок) и укажите его область определения. Если вы выберете вариант **Для всех спрайтов**, у вас получится *глобальный* список, к которому будут иметь доступ все спрайты вашего приложения, а вариант **Только для этого спрайта** создает *локальный* список, который принадлежит спрайту, используемому в данный момент. Локальные списки может читать (и менять) только спрайт-владелец.

Когда вы, нажав **OK**, подтвердите введенную информацию, Scratch создаст новый *пустой список* и покажет блоки, связанные со списками

(рис. 9.3). Вы видите примерно то же, когда создаете новую переменную. Пустой список — это список, в котором еще нет ни одного элемента.

Вы можете использовать новые команды, чтобы управлять содержимым списка во время работы скрипта: добавлять элементы, вставлять их на нужные позиции, удалять или менять их параметры.

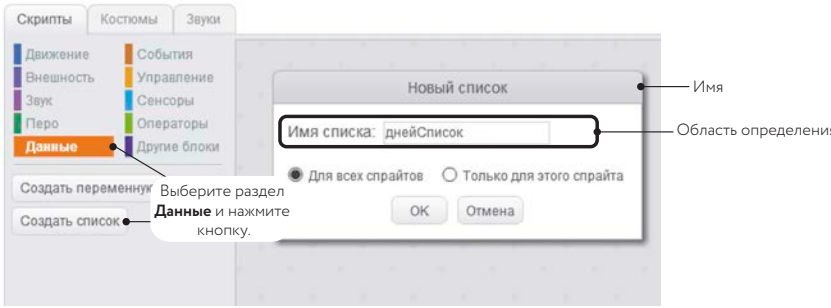


Рис. 9.2. Создание списка в Scratch мало чем отличается от создания переменной

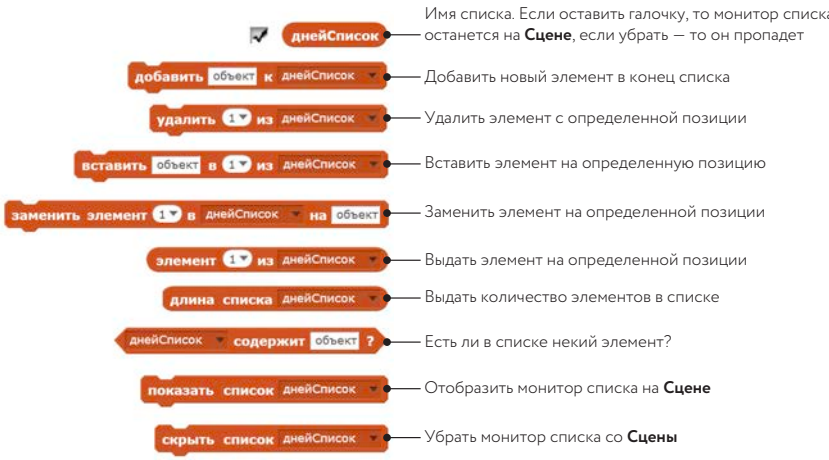


Рис. 9.3. Блоки-команды и блоки-функции, которыми вы можете пользоваться при работе со списками

После создания нового списка Scratch отображает его монитор на **Сцене**, как показано на рис. 9.4. Список изначально пустой, его длина равна 0. Этот блок монитора можно использовать для добавления элементов в ваш список при создании программы.

Если у вас есть данные, которые вы хотите сохранить в списке (как в случае с нашим днейСписок), вы можете ввести их с помощью монитора. На рис. 9.5 показано, как добавить названия дней в днейСписок через монитор.

Рис. 9.4. Монитор только что созданного списка

Щелкните по значку **плюс** в левом нижнем углу семь раз — и вы создадите семь записей. Затем в каждое из семи полей текстового ввода впечатайте название дня недели. Чтобы переходить от одного поля к другому, нажимайте на клавишу **Tab**. После одного нажатия этой клавиши следующая запись в списке будет обведена желтым. Если нажать **Tab** еще раз, будет подсвечен редактируемый текст выбранной записи, а желтая обводка исчезнет. Если вы щелкнете по знаку **плюс**, когда выбранный элемент обведен желтым, новый элемент списка будет добавлен после выделенного элемента; иначе он будет добавлен до текущего элемента. А теперь самостоятельно попробуйте передвигаться по списку.

УПРАЖНЕНИЕ 9.1

Введите в днейСписок названия дней недели, как показано на рис. 9.5.

Рис. 9.5. Заполнение списка днейСписок

Команды управления списками

На рис. 9.3 показаны все блоки, которые Scratch добавил после того, как мы создали днейСписок. В этом разделе мы подробнее рассмотрим эти блоки, чтобы четко понять, как они работают.

Добавить и удалить

Команда **добавить** размещает новый элемент в конце списка, а команда **удалить** удаляет элемент с указанной позиции. На рис. 9.6 эти команды показаны в действии.

Сначала скрипт выполняет команду **удалить** — и убирает второй элемент из списка, «Апельсин». Затем он вводит в конец списка «Лимон» с помощью команды **добавить**.

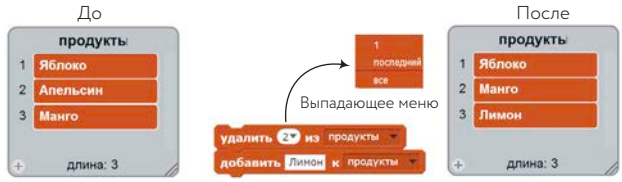


Рис. 9.6. Список до и после использования команд **добавить** *и* **удалить**

Команда **добавить** — простая, а команду **удалить** рассмотрим внимательнее. Можно ввести показатель элемента, который вы хотите удалить, прямо в окошко параметра, или щелкнуть по направленному вниз треугольнику. Выпадающее меню (см. рис. 9.6) предлагает три варианта: **1**, **последний** и **все**. Можно выбрать удаление из списка первого элемента («Яблоко»), последнего («Манго») или всех сразу.

Вставить и заменить

Предположим, вы захотели сохранить имена и номера телефонов своих друзей в виде алфавитного списка — как в телефонной книге мобильника. При составлении такого списка нужно вставлять контакты каждого друга в определенное место. Но потом вдруг один из ваших друзей меняет номер, и вам нужно отредактировать список. На помощь придут команды **вставить** и **заменить**. На рис. 9.7 показано использование этих команд на примере списка телефонов.

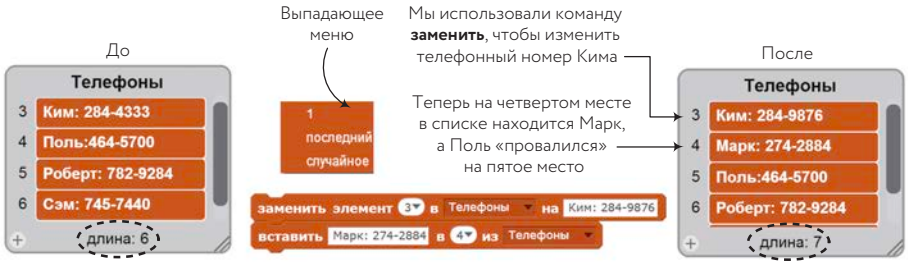


Рис. 9.7. Использование команд **вставить** *и* **заменить** *для обновления списка телефонных номеров*

Команда **заменить** переписывает текущую строку на позиции 3 — теперь у Кима новый номер. Команда **вставить** перемещает номер нового друга, Марка, на позицию 4. Следующие элементы сдвинулись вниз на одну позицию.

Если щелкнуть по номеру элемента, а точнее по направленной вниз стрелочке в обеих командах, **заменить** и **вставить**, появится выпадающее меню с тремя вариантами: **1**, **последний** и **случайное** (см. рис. 9.7). Если вы выберете вариант **случайное**, команда возьмет случайный элемент из списка. Дальше мы увидим, как можно с пользой применить эту функцию.

Доступ к элементам списка

Как было сказано выше, доступ к любому элементу в списке обеспечивается показателем данного элемента. Например, скрипт на рис. 9.8 показывает использование блока **элемент из** для доступа к элементам нашего списка **днейСписок**. Этот скрипт использует переменную поз, чтобы выполнить итерацию каждого элемента списка, показывая содержимое элемента с помощью команды **говорить**.

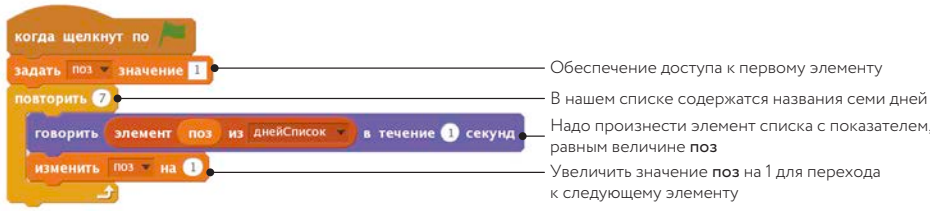


Рис. 9.8. Скрипт заставляет спрайт отображать названия семи дней из днейСписок

Скрипт присваивает переменной поз значение 1, чтобы можно было обратиться к первому элементу списка днейСписок, после чего входит в цикл. Количество повторений равно семи, что соответствует количеству элементов списка. При каждом повторе цикла будет называться элемент списка с показателем, равным величине поз, а затем поз будет увеличиваться на 1, чтобы перейти к следующему элементу. Иными словами, мы используем поз в качестве указателя на элемент в списке.

УПРАЖНЕНИЕ 9.2

Замените цифру 7 в блоке **повторить** на блок **длина списка днейСписок**. Именно так вы бы поступили, если бы не знали, сколько элементов в списке. Также в блоке **элемент** выберите вариант **случайное** в первом выпадающем меню. Скрипт будет отображать элементы списка в случайном порядке.

Блок содержит

Проверить наличие в списке определенной строки вы можете с помощью блока **содержит**. Этот логический блок дает ответ «верно» или «неверно» в зависимости от того, есть ли в списке данная строка. Скрипт, приведенный на рис. 9.9, иллюстрирует одно из применений этого блока. Поскольку днейСписок содержит строку «Пятница», команда **говорить** внутри блока **если** будет выполнена.

Блок **содержит** *не принимает во внимание регистр букв. Блок* **днейСписок содержит пятНИЦА**, *например, будет опознан как верный.*

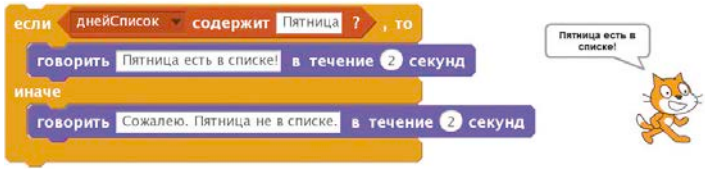


Рис. 9.9. Использование блока **содержит** *для проверки наличия строки в списке*

Проверка границ

Четыре блока для управления списками (**удалить**, **вставить**, **заменить** и **элемент**) требуют ввода параметра — показателя элемента, к которому вы хотите обратиться. Например, для удаления седьмого элемента списка днейСписок мы будем использовать команду **удалить 7 из** днейСписок. Но что, по-вашему, произойдет, если вы используете неверный показатель в одном из этих блоков? Например, как Scratch отреагирует, если вы попросите его удалить восьмой элемент из днейСписок, а в нем всего семь элементов?

Попытка обращения к элементу за пределами списка теоретически является ошибкой. Но, вместо того чтобы вывести сообщение об ошибке или грубо прервать программу, Scratch пытается что-то сделать с неверным блоком. Поэтому отсутствие сообщений об ошибках еще не говорит о том, что все правильно: в вашем коде вполне могут быть ошибки, и, если так, вам нужно их исправить. Scratch не будет жаловаться на неверные показатели в блоках, просто результат, скорее всего, получится не тот, на который вы рассчитывали. В таблице 9.1 показано, что произойдет, если вы попытаетесь обратиться к днейСписок с неверным показателем.

Примеры в табл. 9.1 показывают, что, хотя блоки Scratch пытаются извлечь какой-то смысл из неверно введенных команд, они не обязательно сделают то, что требуется. Если вы хотите, чтобы программа работала так, как вам нужно, надо обеспечить корректный ввод.

Таблица 9.1. Неожиданные результаты обращения к неверным показателям списка

Блок-команда или блок-функция	Результат
	Выдает пустую строку, поскольку **днейСписок** включает только семь элементов. То же происходит, если вы используете показатель меньше 1
	Scratch не обращает внимания на «.9» и выдает первый элемент **днейСписок**, то есть **Понедельник**. А если вы обратитесь к элементу 5.3, Scratch выдаст вам пятый элемент списка, **Пятница**
	Scratch игнорирует эту команду, поскольку она пытается создать в списке пустой промежуток. Список не меняется
	Эффект будет тот же, что и от применения команды **добавить**. Эта команда добавит **Новый день** в конец списка
	Эта команда будет проигнорирована (поскольку в **днейСписок** только семь элементов), и список не изменится

До этого момента в качестве примеров использовались простые списки, которые мы создавали вручную с помощью мониторов. Теперь вопрос: а что если нам неизвестно содержимое списка, когда мы создаем программу? Например, вам может понадобиться сделать список введенных пользователем чисел или заполнить список случайными величинами при каждом запуске программы. Разберемся с этой проблемой.

Динамические списки

Списки — мощный инструмент, поскольку они могут увеличиваться или уменьшаться по ходу работы программы. Предположим, вы пишете приложение — классный журнал, в который учителя вводят количество баллов за правильные ответы в контрольных. Причем оценки могут обрабатываться в дальнейшем: учителю может понадобиться найти максимальный балл класса, минимальный балл, среднее арифметическое, медиану и т. д. Но количество учеников в классах разное. Может, учителю надо будет ввести 20 оценок для Класса 1, 25 для Класса 2 и т. д. Как программе узнать, закончил ли учитель вводить баллы? Скоро вы узнаете ответ на этот вопрос.

Первым делом мы введем два способа наполнить списки данными пользователя. Затем рассмотрим нумерационные списки и некоторые

типичные операции с ними. Как только вы поймете общий подход, вы сможете приспособить эти приемы для своих приложений.

Ввод информации пользователя в списки

Есть два простых способа ввести в списки информацию пользователя. В первом случае программа начинает с вопроса, сколько записей будет в списке, а потом запускает цикл сбора информации. Ее скрипт показан на рис. 9.10.

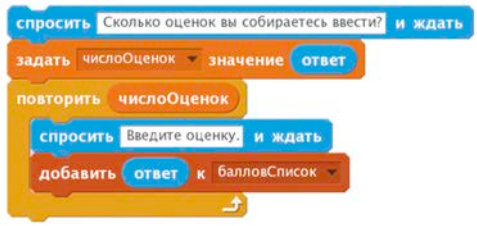
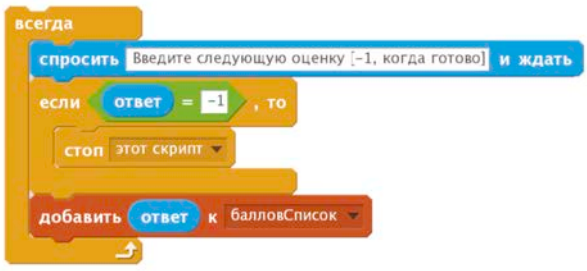


Рис. 9.10. Скрипт спрашивает у пользователя, сколько оценок тот собирается ввести

Получив ответ, сколько оценок ему ожидать, скрипт начинает цикл, число повторов которого равно введенному пользователем числу. При каждой итерации у пользователя запрашивается новая оценка, которая добавляется к списку балловСписок.

Второй динамический способ пополнения списка — ввести специальную величину (*сигнальную метку*), которая отмечает конец списка. Конечно, ее надо выбрать так, чтобы ее нельзя было принять за обычный элемент списка. Если ваш список будет состоять из имен или положительных чисел, то сигнальная метка «–1» — хорошая идея. Если же пользователь будет вводить отрицательные величины, то «–1» в качестве метки не годится. Сигнальная метка «–1» подходит для нашего списка балловСписок, и в скрипте, показанном на рис. 9.11, она применяется, чтобы отметить окончание ввода данных пользователем.



Так выглядит содержание списка балловСписок, если вводить 85, 100, 95, –1

Рис. 9.11. Использование сигнальной метки для контроля окончания ввода данных

При каждом повторе цикла скрипт просит пользователя ввести число и сравнивает его с сигнальной меткой. Скрипт выводит значение метки (в нашем случае –1) в подсказке, обращенной к пользователю. Если пользователь вводит –1, скрипт останавливается, поскольку знает, что ввод закончен. Иначе введенные величины продолжают добавляться к списку, а у пользователя запрашивается следующее число. На рис. 9.11 показано, как будет выглядеть баллов**Список**, если пользователь вводит три числа, а затем сигнальную метку.

Создание столбчатой диаграммы

BarChart.sb2

Чтобы понять, как данные от пользователя организуются в списки, напишем приложение, которое рисует столбчатую диаграмму (*гистограмму*), используя введенные пользователем числа. Для простоты будем работать с пятью цифрами в диапазоне от 1 до 40. Как только программа получила пять чисел, она изобразит пять столбиков, чья высота будет пропорциональна введенным числам. Интерфейс пользователя для нашего создателя диаграмм приведен на рис. 9.12.

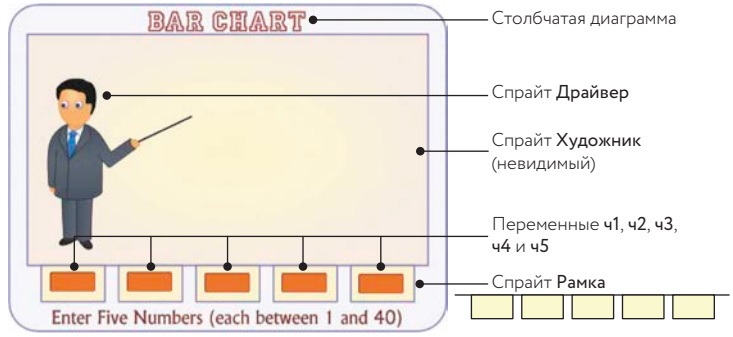


Рис. 9.12. Приложение, создающее столбчатую диаграмму

В этом приложении три спрайта. Драйвер управляет работой приложения; он содержит скрипт, который получает информацию от пользователя, наполняет список и отдает спрайту Художник команду начать рисовать диаграмму. Художник (Painter) — спрайт-невидимка, который рисует гистограмму. Спрайт Рамка (Frame) нужен только для красоты: он закрывает низ каждого столбика, чтобы все было ровно. Без него было бы видно, что нижняя часть столбиков округлая. Величины для пяти столбцов используют пять переменных с именами от ч1 до ч5, чьи мониторы расположены на **Сцене** справа. Когда вы щелкаете по зеленому флажку, чтобы запустить приложение, спрайт Драйвер инициирует скрипт, показанный на рис. 9.13.

Рис. 9.13. Основной скрипт спрайта Драйвер

Для начала на **Сцене** появляется спрайт Драйвер и полностью очищает экран ❶. Если на **Сцене** нарисована диаграмма, она будет стерта. Затем скрипт очищает список чиселСписок, чтобы можно было использовать его для сбора новых записей пользователя, и вызывает **ПоказатьВеличины** ❷: величины ч1–ч5 будут заданы так, что их мониторы окажутся пустыми.

Теперь, когда **Сцена** подготовлена, скрипт начинает цикл **повторить** ❸, который должен сработать пять раз. Внутри цикла Драйвер просит пользователя ввести число, которое он добавляет к чиселСписок. Получив от пользователя все пять чисел и сохранив их в списке, Драйвер прячется ❹, чтобы освободить место для столбчатой диаграммы. После этого он снова вызывает **ПоказатьВеличины**, чтобы обновить величины ч1–ч5 данными, которые ввел пользователь, и передает Художнику команду нарисовать все пять столбиков.

Прежде чем узнать, как Художник рисует столбики, посмотрим на процедуру **ПоказатьВеличины**, показанную на рис. 9.14.

ПоказатьВеличины задает переменным ч1–ч5 значения соответствующих записей в чиселСписок. Первый вызов процедуры производится сразу после очистки чиселСписок, поэтому все пять переменных содержат пустые строки. В результате производится очистка

Рис. 9.14. Процедура **ПоказатьВеличины**

всех пяти мониторов на **Сцене**, что нам и нужно. Когда числСписок содержит данные пользователя, вызов ПоказатьВеличины приводит к тому, что на мониторах снова появляются данные.

Посмотрим на процедуру **Рисовать**, которая запускается, когда спрайт Художник получает соответствующее сообщение. Ее скрипт показан на рис. 9.15.

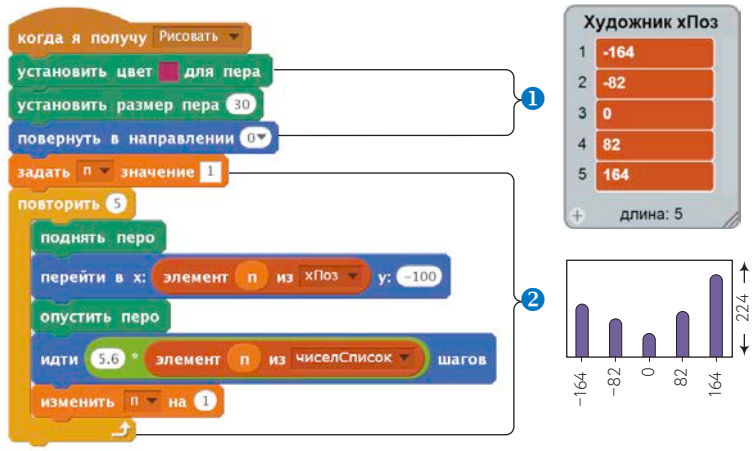


Рис. 9.15. Скрипт **Рисовать** *спрайта* Художник

Сначала спрайт определяет цвет пера. Затем он задает большой размер пера, чтобы рисовать широкие столбики. Готовясь рисовать пять вертикальных линий, спрайт указывает вверх ❶.

Скрипт начинает цикл, чтобы нарисовать пять столбиков ❷. Нам заранее известны *x*-позиции каждого из них, ведь мы создали список хПоз для хранения этих величин (тоже показан на рисунке). Во время каждого повтора цикла спрайт Художник переходит к *x*-позиции текущего столбца, опускает перо и двигается вверх, рисуя вертикальную линию.

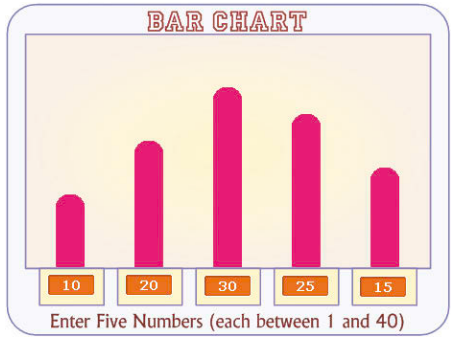


Рис. 9.16. Пример выходных данных приложения для создания столбчатых диаграмм

Длина этой линии пропорциональна величинам в числСписок. Максимальная высота нашей диаграммы на **Сцене** составит 224 пиксела, а поскольку самое большое возможное число — 40, то ввод этого числа должен давать нам столбик высотой в 224 пиксела. Чтобы рассчитать высоту в пикселах для любого числа в числСписок, надо умножить его

на 5,6 (224/40). На рис. 9.16 показан результат работы приложения на основе введенных пользователем чисел. Обратите внимание, что спрайт Рамка закрывает округленный низ линий широкого пера, так что основания столбиков выглядят плоскими.

УПРАЖНЕНИЕ 9.3

Запустите приложение несколько раз, чтобы разобраться, как оно работает. Измените скрипт так, чтобы все столбики диаграммы были разных цветов. Создайте еще один список по имени цвета, чтобы спрайт Художник мог хранить там числа цветов для пяти столбиков, а перед рисованием очередного столбика используйте следующую команду:

Нумерационные списки

Списки чисел используются во многих приложениях. Это могут быть списки баллов за контрольные, замеры температуры, цены на товары и т. д. В этом разделе мы познакомимся с некоторыми операциями, которые вы сможете выполнить с нумерационными списками. В частности, мы напишем процедуры для поиска максимальной и минимальной величин, а также расчета среднего арифметического.

Поиск минимума и максимума

FindMax.sb2

Представьте, что вы учитель и вам надо узнать самое большое количество баллов, набранных во время последнего экзамена в классе. Можно написать программу, которая сравнит все полученные баллы и найдет максимальную величину. Первый скрипт, приведенный на рис. 9.17, ищет самое большое число в списке баллы.

Процедура **НайтиМаксимум** начинается с того, что переменной максБалл присваивается величина, равная первому числу в списке. Затем начинается цикл сравнения остальных номеров списка с текущей величиной максБалл. Всякий раз, когда обнаруживается величина больше максБалл, эта переменная меняет значение на большее. Когда цикл прекращается, величина, сохраненная в переменной максБалл, будет самым большим числом в списке.

Поиск самого маленького числа в списке выполняется примерно по тому же алгоритму. Мы начинаем с предположения, что первый элемент списка и есть минимум, а затем используем цикл, чтобы сравнить с ним оставшиеся элементы. Когда мы находим число меньше, мы обновляем значение переменной, в которой сохраняем минимальную величину.

Рис. 9.17. Нахождение в списке самого большого числа

УПРАЖНЕНИЕ 9.4

Создайте процедуру **НайтиМинимум**, которая находила бы самое маленькое число в списке набранных баллов, используя знания, полученные при изучении этого раздела.

Расчет среднего арифметического

FindAverage.sb2

Мы напишем процедуру, которая высчитывает среднее арифметическое всех баллов, сохраненных в списке баллы. Оно высчитывается как сумма *N* чисел, деленная на *N*. Именно эту операцию и выполняет процедура, приведенная на рис. 9.18.

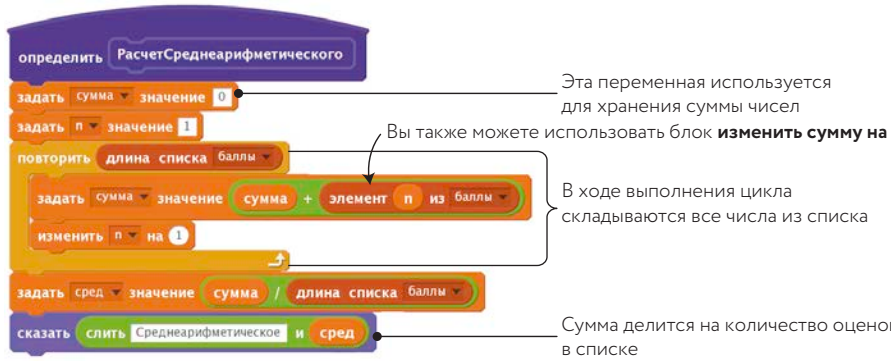


Рис. 9.18. Вычисление среднего арифметического для списка чисел

Процедура **РасчетСреднеарифметического** использует цикл **повторить**, чтобы пройти по всем баллам, сохраняемым в списке: она

складывает числа и сохраняет результат в переменной сумма (выставляется на 0 перед началом цикла). Когда цикл закончен, скрипт рассчитывает среднее арифметическое, деля сумма на длину списка, и сохраняет результат в переменной сред.

Обратите особое внимание на то, как мы накапливали переменную сумма *в цикле. Этот прием — накапливающая суммация — очень часто используется в программировании.*

В следующем разделе мы узнаем, как осуществлять поиск в списке и сортировку — это обычные задачи программирования. Я также познакомлю вас с несколькими простыми процедурами для выполнения этих операций.

УПРАЖНЕНИЕ 9.5

Соберите процедуры **РасчетСреднеарифметического**, **НайтиМинимум** и **НайтиМаксимум** в единую процедуру (с именем **ОбработкаСписка**), которая будет отображать среднее арифметическое, минимальную и максимальную величины для списка баллов одновременно.

Поиск и сортировка списков

Предположим, у вас есть неупорядоченный список контактов. Требуется *рассортировать* их в алфавитном порядке по именам. Если вам нужен телефон человека из списка, вы сможете *провести поиск* по его фамилии, чтобы найти контактную информацию. Цель этого раздела — представить основные методы программирования для поиска и сортировки списков.

Линейный поиск

SearchList.sb2

Использование блока Scratch **содержит** — простой способ проверить, есть ли в списке определенный элемент. Но если мы хотим узнать позицию искомого элемента в списке, придется искать вручную.

В этом разделе вы познакомитесь с методом *линейного поиска* (*последовательного перебора*), который позволяет вести поиск в списке. Этот прием нетруден для восприятия и легок в исполнении, и работает он с любым списком — отсортированным и беспорядочным. Но, поскольку при линейном поиске с заданной величиной надо сравнить все элементы в списке, операция может занять много времени.

Представим, что вы ищете некий элемент в списке фрукты. Если он в списке есть, вы хотите узнать его позицию. Процедура **ПоисквСписке**, показанная на рис. 9.19, проводит линейный поиск в списке фрукты.

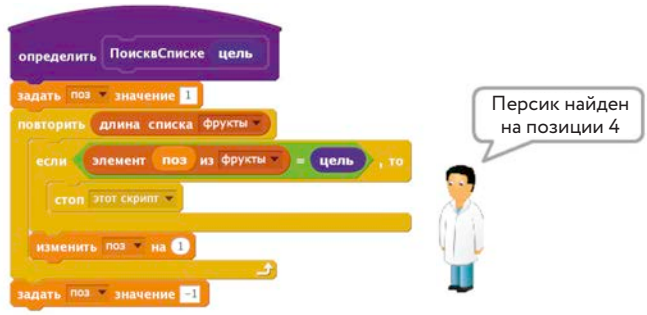


Рис. 9.19. Процедура **ПоисквСписке**

Процедура **ПоисквСписке** с первого элемента начинает сравнивать названия фруктов в списке, одно за другим, с тем, что мы ищем. Объект поиска задан параметром **цель**. Процедура останавливается либо если величина найдена, либо если достигнут конец списка. Если скрипт находит нужную величину, переменная поз будет содержать позицию элемента. Иначе процедура меняет значение поз на неверную величину (–1 в данном случае), чтобы указать на отсутствие элемента в списке. На рис. 9.20 приведен пример вызова процедуры и результата, который она выдает.

Рис. 9.20. Использование процедуры **ПоисквСписке**

Величина поз указывает на два момента: содержится нужный нам элемент в списке или нет; если он в списке есть, какова его позиция. В ходе работы этой процедуры скрипт присвоит поз значение 4, указывая, что «Персик» найден на четвертой позиции списка фрукты.

Частота появления события

ItemCount.sb2

Предположим, в вашей школе проводится опрос о качестве питания в кафетерии. Учащиеся ставят еде оценки от 1 до 5 (1 — «плохо», 5 — «отлично»). Все ответы заносятся в список, и вас просят написать

программу для обработки данных. Для начала школа хочет знать, сколько учащихся считают, что еда отвратительная (сколько из них поставили оценку 1). Как бы вы стали писать такую программу?

Вам явно нужна процедура, которая посчитает, сколько раз в списке появляется единица. Возьмем список, который содержит 100 случайных чисел. Процедура его наполнения данными показана на рис. 9.21. Она добавит 100 случайных чисел от 1 до 5 в список под названием Опрос.

Рис. 9.21. Процедура **НаполнениеСписка**

Теперь, когда у нас есть список оценок, мы можем посчитать, сколько раз в нем встречается заданная величина — 1. Для этого нужна процедура **ПосчитатьЭлемент**, показанная на рис. 9.22.

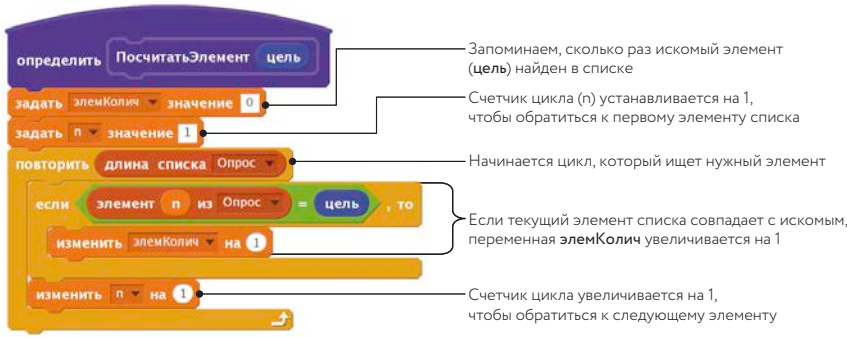


Рис. 9.22. Выясняем, сколько раз элемент появляется в списке

Параметр цель — искомый элемент, а переменная элемКолич отслеживает, сколько раз он был найден. Процедура начинается с того, что переменной элемКолич присваивается значение 0, а затем стартует цикл **повторить**, ищущий в списке нужную величину. При каждом прохождении цикла происходит проверка элемента списка под соответствующим номером. Если элемент равен цели, скрипт увеличивает значение переменной элемКолич на 1.

Чтобы узнать, насколько учащиеся недовольны едой в кафетерии, надо вызвать процедуру **ПосчитатьЭлемент** с аргументом 1, как показано на рис. 9.23.

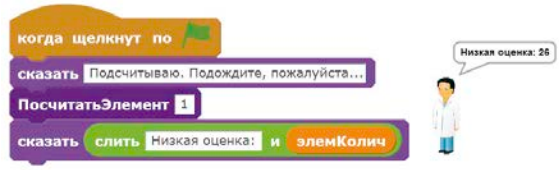


Рис. 9.23. Применение процедуры **ПосчитатьЭлемент**

УПРАЖНЕНИЕ 9.6

Вы нашли ответ на заданный директором вопрос, но тут ему стало интересно, сколько человек поставили кафетерию оценку «отлично». Еще директору захотелось узнать, сколько учащихся участвовало в опросе. Усовершенствуйте программу и запустите ее, чтобы предоставить эту информацию.

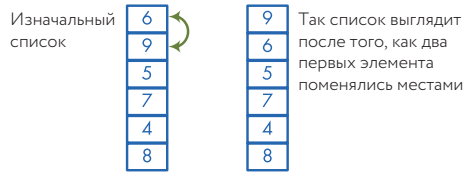
Пузырьковая сортировка

BubbleSort.sb2

Если у вас есть список имен, или набранных в игре очков, или каких-то еще данных, которые вы хотите расположить в определенном порядке — алфавитном, или от большего к меньшему, или как-то иначе, — вам надо провести сортировку списка. Здесь есть много подходов, но один из самых простых — *пузырьковая сортировка*. Она носит такое странное имя, потому что величины уподобляются пузырькам, всплывающим на поверхность воды: величины точно так же «всплывают» на нужные места в списке. В этом разделе мы познакомимся с пузырьковой сортировкой и напишем в Scratch программу, которая будет выполнять этот алгоритм.

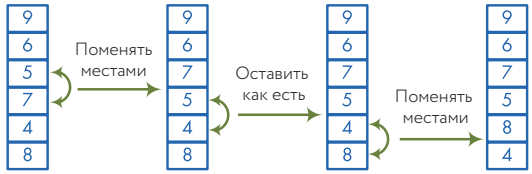
Предположим, у нас есть список чисел [6 9 5 7 4 8], которые мы хотим расставить в порядке убывания. Дальше пошагово показано, как работает алгоритм пузырьковой сортировки.

1. Для начала мы сравниваем два первых элемента в списке. Поскольку 9 больше 6, их надо поменять местами, как показано ниже.

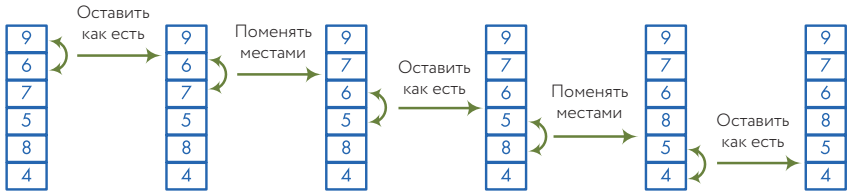


2. Теперь можно перейти к сравнению второго и третьего элементов, 6 и 5. Поскольку 6 больше 5, эти два числа и так стоят в нужном нам порядке и мы можем двигаться дальше, к следующей паре.

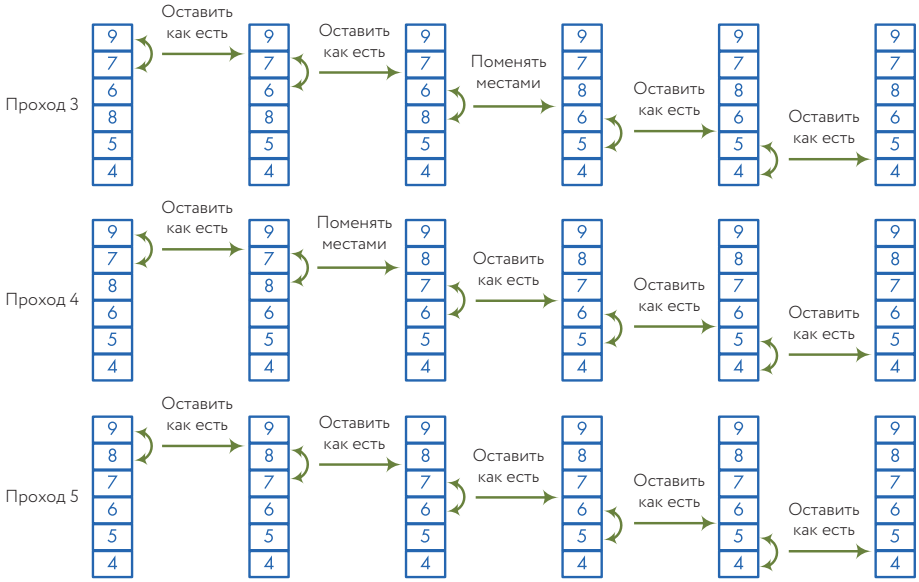
3. Дальше мы будем повторять действие снова и снова, сравнивая третий и четвертый, четвертый и пятый и, наконец, пятый и шестой элементы. Посмотрите, как выглядит список после трех сравнений.



4. Этот проход окончен, но наш список еще далек от идеала. Надо повторить операцию с начала. Мы еще раз сравним все пары элементов и поменяем их при необходимости местами.



5. Мы будем повторять процесс пузырьковой сортировки до тех пор, пока числа не прекратят меняться местами. Это и будет означать, что наш список отсортирован. Последние три прохода алгоритма показаны ниже.



Теперь реализуем пузырьковую сортировку в среде Scratch. Скрипт, показанный на рис. 9.24, включает два цикла. Внутренний проходит по списку, сравнивая и при необходимости меняя местами числа, и задает флагу сделано значение 0, если требуется еще один проход. Внешний цикл повторяется, пока значение флага сделано равно 0: это означает, что сортировка еще не завершена. Если внутренний цикл заканчивает проход без перестановки чисел, внешний цикл завершается. Это означает окончание процедуры.

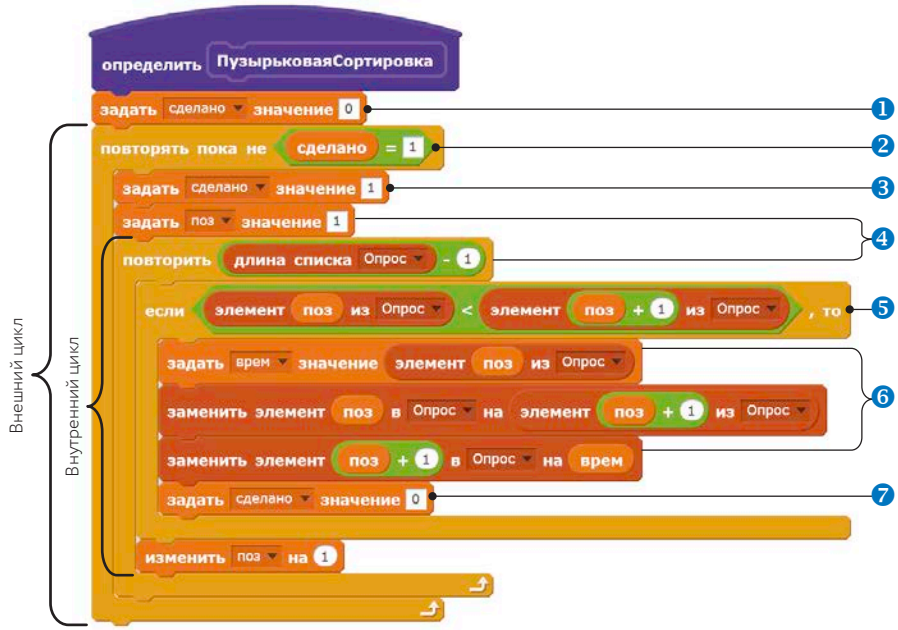


Рис. 9.24. Процедура **ПузырьковаяСортировка**

Внимательнее рассмотрим эту процедуру. Поскольку мы только начинаем сортировку, значение флага сделано равно 0 ❶. Внешний цикл использует блок **повторять пока не**, чтобы снова и снова прорабатывать список, пока он не будет отсортирован (пока значение сделано не станет равно 1) ❷. В начале каждого прохода цикл выставляет значение сделано на 1 ❸ (исходя из предположения, что никаких перестановок не будет). Кроме того, переменной поз присваивается значение 1, чтобы сортировка началась с первого же числа.

После этого внутренний цикл сравнивает каждую пару элементов списка. Требуется выполнить $N - 1$ сравнений ❹, где N — количество элементов в списке.

Если элемент с показателем поз+1 больше, чем элемент с показателем поз ❺, эти два числа надо поменять местами. Иначе процедура добавляет 1 к поз, чтобы перейти к сравнению следующей пары элементов.

Если все-таки надо поменять числа местами, процедура выполняет эту операцию с помощью временной переменной врем ❻.

По окончании прохода по списку внутренний цикл снова задает флагу сделано значение 0, если перестановка понадобилась, или оставляет 1, если обошлось без перестановок ❼. Внешний цикл продолжит работу, пока список не будет полностью отсортирован.

УПРАЖНЕНИЕ 9.7

Составьте список уже не чисел, а имен, и при помощи пузырьковой сортировки упорядочьте его. Работает ли сортировка как надо? Какие изменения нужно внести в процедуру, чтобы сортировка производилась в порядке возрастания?

Расчет медианы

Median.sb2

Теперь, когда мы знаем, как сортировать список, мы можем легко вычислить медиану любой последовательности чисел. Медиана — среднее число в последовательности. Если у нас нечетное количество элементов, то мы выбираем средний, а если четное, то медиана — среднее арифметическое двух средних чисел. Медиану отсортированного списка из N элементов можно описать следующим образом.

$$\text{медиана} = \begin{cases} \text{элемент с показателем } \dfrac{N+1}{2} & \text{если } N \text{ — нечетное число} \\ \text{среднее арифметическое элементов с показателями } \dfrac{N}{2} \text{ и } \dfrac{N}{2}+1 & \text{если } N \text{ — четное число} \end{cases}$$

Процедура, которая выполняет данное вычисление, представлена на рис. 9.25. Она исходит из того, что список уже отсортирован.

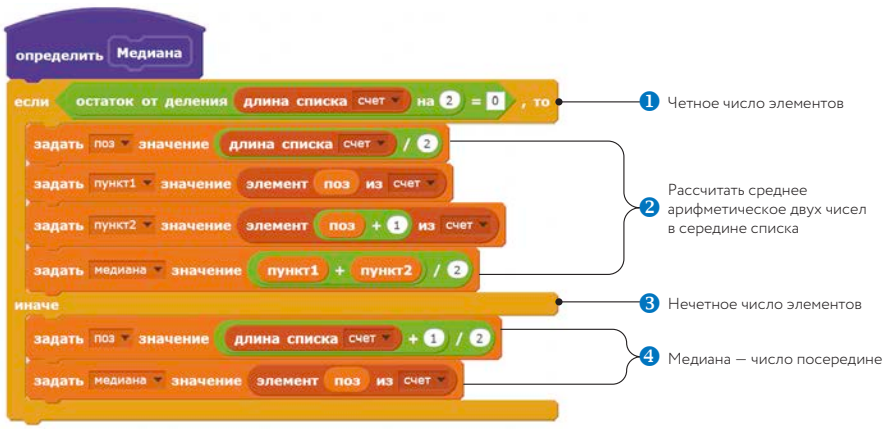


Рис. 9.25. Расчет медианы отсортированного списка чисел

Процедура использует блок **если/иначе**, чтобы обработать два случая: с нечетным и четным числом элементов в списке. Если число элементов в списке делится на 2 без остатка (в нем четное число элементов) ❶, переменная медиана рассчитывается как среднее арифметическое двух цифр в середине списка ❷. Иначе список состоит из нечетного числа элементов ❸ и значение переменной медиана устанавливается равным числу в середине списка ❹.

Мы прошли большой материал, пора применить новые знания для чего-нибудь интересного. Оставшаяся часть главы посвящена примерам того, как списки используются в более сложных приложениях.

Проекты Scratch

В этом разделе вы познакомитесь с проектами Scratch, в которых широко используются различные возможности списков. Я познакомлю вас с новыми идеями и приемами, которые вы сможете использовать в своей работе.

Поэт

Poet.sb2

Начнем с генератора стихов на английском. Наш поэт-машина будет наобум выбирать слова из пяти списков (артикль, прилагательное, существительное, глагол и предлог) и комбинировать их по определенному образцу. Чтобы у стихов была какая-то тема, все слова в списках как-то связаны с любовью и природой (конечно, на выходе все равно может получиться белиберда, но тем веселее!).

Идея такой программы в измененном виде позаимствована из книги Дэниела Уотта «Учимся с Logo» (Learning with Logo, McGraw-Hill, 1983). Полные списки слов, которые мы используем, находятся в файле Poet.sb2.*

* Обучающий язык программирования, созданный в 1967 году Дэниелом Боброу, Уолтером Фюрцайгом, Сеймуром Папертом и Синтией Соломон.

Стихотворение будет состоять из трех строк, которые устроены следующим образом.

- Строка 1: артикль, прилагательное, существительное.
- Строка 2: артикль, существительное, глагол, предлог, артикль, прилагательное, существительное.
- Строка 3: прилагательное, прилагательное, существительное.

Посмотрим на процедуру, которая компонует первую строку стихотворения (рис. 9.26).

Этот скрипт выбирает случайное слово из списка артикль и сохраняет его в переменной строка1. Затем скрипт добавляет пробел, случайно

выбранное слово из списка прилагательное, потом еще пробел и случайно выбранное слово из списка существительное. Наконец спрайт Поэт произносит строку целиком. Я не буду давать процедуры для двух оставшихся строк, поскольку они устроены почти так же. Вы можете ознакомиться с ними, открыв файл *Poet.sb2*.

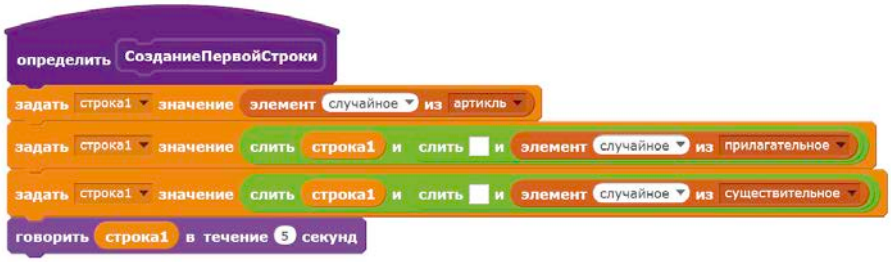


Рис. 9.26. «Создание» первой строки стихотворения

Вот два стихотворения, созданных поэтом-машиной.

each glamorous road
a fish moves behind each white home
calm blue pond

every icy drop
a heart stares under every scary gate
shy quiet queen*

* Каждая прекрасная дорога / Рыба двигается за каждым белым домом / Спокойный голубой пруд; Каждая ледяная капля / Сердце смотрит под каждые страшные ворота / Робкая спокойная королева

УПРАЖНЕНИЕ 9.8

Poet.sb2

Откройте файл *Poet.sb2* и запустите его несколько раз, чтобы оценить компьютерную поэзию. Затем измените программу так, чтобы в ней было три спрайта, каждый из которых отвечает за одну строку стихотворения, — чтобы можно было увидеть стихотворение на **Сцене** целиком.

Игра «Назови четырехугольник»

QuadClassify.sb2

Наш следующий проект — простая игра, в которой можно поработать с разными видами четырехугольников. На **Сцене** появляется одна из шести фигур (параллелограмм, ромб, прямоугольник, квадрат, трапеция, дельтоид), и программа просит игрока назвать форму, нажав на соответствующую кнопку, как показано на рис. 9.27.

В игре семь спрайтов: Кнопка1, Кнопка2 ... Кнопка6 и седьмой (Драйвер) — для основного скрипта. Как показано на рис. 9.27, у спрайта Драйвер шесть костюмов, которые соответствуют шести разновидностям четырехугольников. После щелчка по иконке с зеленым

флажком спрайт Драйвер выполняет скрипт запуска игры, приведенный на рис. 9.28.

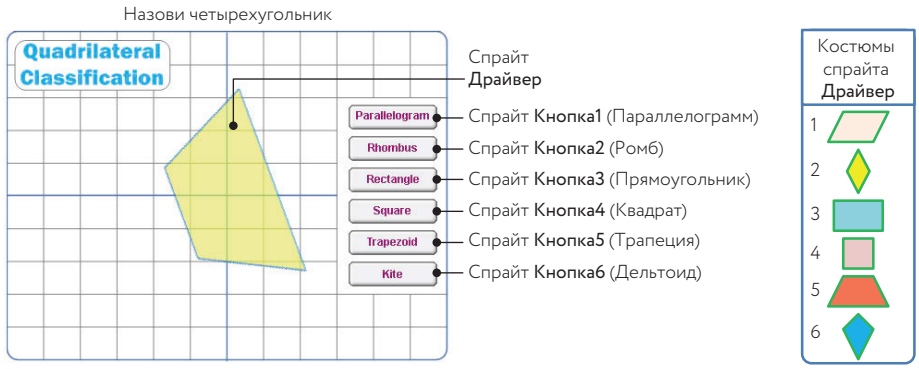


Рис. 9.27. Интерфейс пользователя для игры «Назови четырехугольник»

Первым делом спрайт Драйвер переходит в верхний слой ❶, чтобы его не закрывали никакие кнопки. В основном цикле игры ❷ скрипт при каждом проходе выводит случайно выбранный прямоугольник при помощи процедуры **ПокажиФигуру** ❸. Показав четырехугольник, скрипт присваивает глобальной переменной выбор значение 0: это указывает на то, что пользователь пока не дал ответа ❹.

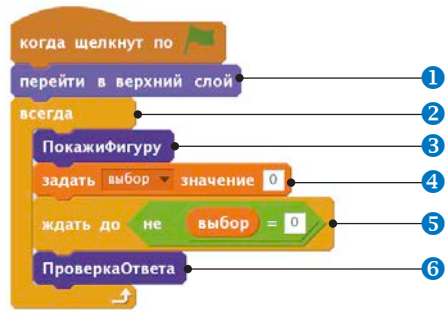


Рис. 9.28. Основной скрипт спрайта Драйвер

Затем скрипт ожидает ❺, пока значение переменной выбор не станет отличным от 0 (это происходит после того, как пользователь нажал на одну из кнопок). После этого скрипт вызывает процедуру **ПроверкаОтвета** ❻, чтобы сообщить пользователю, правильна его догадка или нет.

Теперь вы знаете, как работает основной скрипт. Рассмотрим процедуру **ПокажиФигуру**, отображенную на рис. 9.29.

Сначала процедура перемещает спрайт Драйвер в центр **Сцены** так, чтобы он указывал в случайном направлении ❶. Потом она приписывает

переменной фигура случайное значение от 1 до 6 и меняет костюм спрайта ❷, показывая игроку четырехугольник, который тот должен назвать.

Рис. 9.29. Процедура **ПокажиФигуру** *спрайта* Драйвер

Чтобы сетка-фон оставалась видимой, процедура **ПокажиФигуру** устанавливает уровень прозрачности ❸ равным случайной величине от 25 до 50. Чтобы создать иллюзию, будто каждый раз появляется новая фигура, процедура также задает эффект цвета равным случайной величине и соответственно меняет цвет костюма ❹, а также размер спрайта — 80, 90 или 150% от изначального ❺.

Теперь рассмотрим скрипты шести спрайтов-кнопок, приведенные на рис. 9.30. Они одинаковые, разнятся только значения, присваиваемые переменной выбор.

Рис. 9.30. Скрипты для спрайтов-кнопок

Эти однострочные скрипты устанавливают величину переменной выбор в зависимости от того, на какую кнопку нажимает пользователь. Потом процедура **ПроверкаОтвета**, приведенная на рис. 9.31, может сравнить ее с величиной переменной фигура, которая указывает на то, какой именно четырехугольник показан на **Сцене**.

Если значения переменных выбор и фигура равны, то ответ игрока верен. В противном случае он неверен, и тогда спрайт назовет правильный. Процедура **ПроверкаОтвета** использует переменную фигура в качестве индекса к списку четырехугНазв, который также приведен на рис. 9.31: так показывается верное название фигуры.

УПРАЖНЕНИЕ 9.9

QuadClassify.sb2

Откройте файл *QuadClassify.sb2* и запустите его несколько раз, чтобы разобраться, как он работает. Как видно, эта игра бесконечна. Усовершенствуйте программу, добавив к ней критерий окончания игры. Также добавьте счет: сколько раз пользователь дал правильные и неправильные ответы.

Рис. 9.31. Процедура **ПроверкаОтвета**

Волшебник-математик

MathWizard.sb2

На примере этого приложения мы познакомимся с двумя способами полезного применения списков: для хранения записей разного размера и в качестве указателя к другому списку. Под *записью* имеется в виду набор данных, относящихся к одному человеку, месту или предмету. В нашем случае каждая запись содержит ответ на загадку, а также инструкции к ней. И хотя у каждой загадки всего один ответ, количество инструкций может быть разным.

Волшебник-математик просит пользователя загадать число и выполнить с ним несколько математических действий (сначала умножить его на 2, затем вычесть 2, потом разделить ответ на 10 и т. п.). А потом, когда пользователь выполнил все расчеты, волшебник использует свою магию, чтобы отгадать получившееся число. Таблица 9.2 показывает, как работает игра.

Таблица 9.2. Как работает волшебник-математик

Инструкции волшебника	Число пользователя
Задумайте число	2
Прибавьте к нему 5	7
Умножьте полученное число на 3	21
Вычтите из полученного числа 3	18
Разделите полученное число на 3	6
Вычтите из полученного числа задуманное	4

Выдав последнюю инструкцию, программа скажет, что, если вы следовали всем инструкциям, у вас должно было получиться 4, хотя она и не знает, что загадали вы 2. Поиграйте, загадывая разные числа, и попробуйте понять, в чем тут фокус.

Интерфейс этого приложения показан на рис. 9.32.

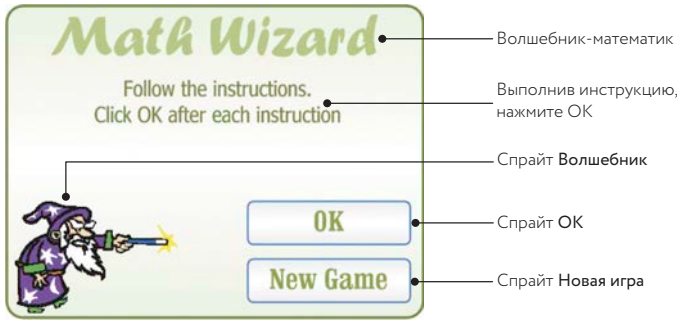


Рис. 9.32. Интерфейс пользователя для приложения «Волшебник-математик»

В приложении три спрайта: Волшебник (Wizard) — он дает игроку указания — и спрайты OK и Новая игра (New game), которые отвечают за одноименные кнопки. Также программа использует два списка, приведенных на рис. 9.33.

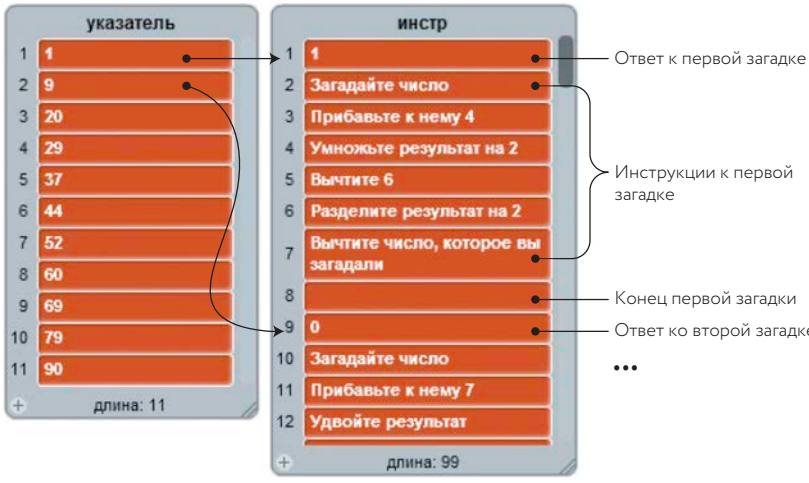


Рис. 9.33. Два списка спрайта Волшебник

Список инстр (справа) содержит 11 записей-загадок. Каждая включает ответ к загадке, инструкции и пустой элемент, маркирующий конец записи. Элементы списка слева (указатель) указывают на номер первого

элемента каждой загадки в списке инстр. Например, второй элемент в списке указатель — 9, и это значит, что запись, в которой содержится вторая загадка, начинается на девятой позиции списка инстр, как показано на рис. 9.33. В общих чертах опишем, как устроена игра.

1. Когда пользователь начинает новую игру, программе надо выбрать случайное число в диапазоне от 1 до 11 (сейчас в игре 11 загадок).

2. Посмотреть в списке указатель, с какой позиции начинается запись выбранной загадки. Например, если выбрана вторая загадка, то список указатель сообщает нам, что запись начинается с элемента 9 в списке инстр.

3. Обратиться к списку инстр по показателю, определенному на предыдущем шаге. Первый элемент понимается как ответ на загадку. Следующие — инструкции, которые будет произносить волшебник.

4. Теперь пусть волшебник поочередно произносит инструкции к загадке, пока не дойдет до пустого элемента, который означает последнюю инструкцию. Волшебник должен подождать, пока пользователь нажмет кнопку OK, чтобы произнести еще одну инструкцию.

5. Дать ответ к загадке.

Теперь, когда мы в общем и целом представляем себе, как должна работать эта игра, изучим скрипты для двух кнопок, приведенные на рис. 9.34.

Рис. 9.34. Скрипты для обоих спрайтов-кнопок

Кнопка Новая игра при нажатии передает сообщение **НоваяИгра**. Когда, в ходе выполнения инструкций, пользователь нажимает OK, спрайт задает переменной щелк значение 1 и тем самым информирует спрайт Волшебник, что пользователь выполнил все инструкции. Когда спрайт Волшебник получает сообщение **НоваяИгра**, он выполняет скрипт, показанный на рис. 9.35.

НоваяИгра начинается с того, что убирается облачко с текстом, оставшееся от предыдущей загадки (если она была), а значение переменной щелк устанавливается равным 0 ❶. Затем в переменной загадкиНомер сохраняется номер случайно выбранной загадки ❷. Потом программа считывает начальную позицию выбранной загадки из списка **указатель** и сохраняет его в переменной поз ❸. После этого скрипт использует переменную поз, чтобы получить ответ и сохранить его в переменной загадкиОтвет ❹. Теперь скрипт увеличивает значение переменной поз на 1, и она уже указывает на первую инструкцию к загадке. Начинается цикл **повторять пока не**, поочередно выводятся все инструкции к загадке ❺. Произнеся инструкцию, скрипт ждет, пока переменная щелк не изменит значение на 1, и после этого переходит к следующей инструкции ❻. Когда цикл доходит до пустого элемента, он прекращается, и скрипт произносит ответ ❼.

Рис. 9.35. Скрипт **НоваяИгра** *спрайта* Волшебник

УПРАЖНЕНИЕ 9.10

Если вы удалите одну из загадок или измените число инструкций для некоторых загадок, придется переделать список-указатель, чтобы привести его в соответствие со списком инстр. Напишите процедуру, которая автоматически наполняет список указатель исходя из текущего содержания списка инстр. Надо ориентироваться на пустые строки в списке инстр: они указывают на конец одной записи и начало следующей.

Тест по ботанике

FlowerAnatomy.sb2

В этом разделе на примере теста «Как устроен цветок» я покажу, как создавать в Scratch простые опросники. На рис. 9.36 приведен интерфейс пользователя в начале и после того, как программа проверила ответы. Пользователь должен вводить буквы, соответствующие помеченным частям цветка, а затем нажать кнопку Проверить для проверки ответов. Программа сравнивает ответы пользователя с правильными и дает обратную связь, показывая зеленую галочку или красный крестик рядом с каждым ответом.

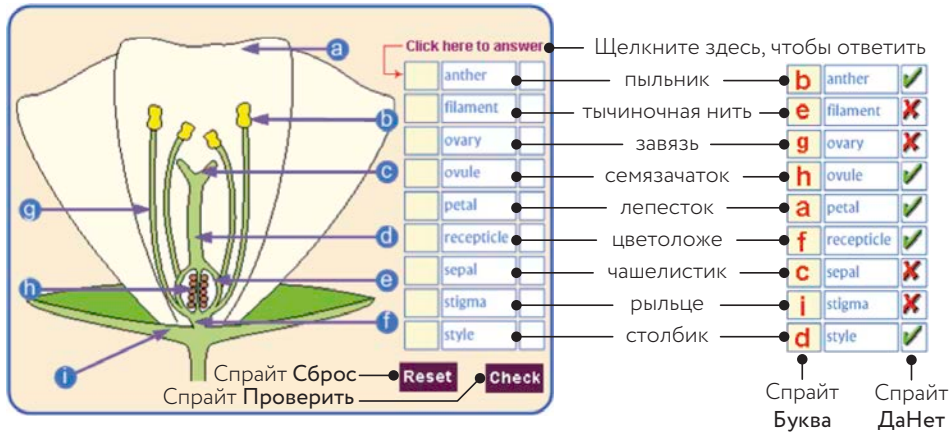


Рис. 9.36. Интерфейс пользователя теста на знание устройства цветка

У этого опроса три списка. Первый (правОтв) содержит буквы, которые соответствуют правильным ответам для девяти частей теста. Второй (отв) содержит введенные пользователем ответы, а третий (ячейкаУЦентр) — 11 вертикальных позиций, которыми пользуются спрайты Буква (Letter) и ДаНет (YesNo). Так они узнают, где им печатать свои костюмы. Когда пользователь щелкает мышкой по одному из квадратиков ответа, спрайт **Сцена** запрашивает ответ, обновляет элемент списка отв в соответствии с введенной информацией и печатает введенную букву поверх квадратика. Откройте файл *FlowerAnatomy.sb2* и изучите скрипты, которые считывают и отображают ответы пользователя.

Когда пользователь нажимает кнопку Проверить, спрайт ДаНет, у которого есть костюмы для зеленой галочки и красного крестика, выполняется скрипт, приведенный на рис. 9.37.

Скрипт поочередно сравнивает элементы списков правОтв и отв. Если величины равны, он печатает галочку, показывая пользователю, что тот дал правильный ответ. Иначе скрипт печатает красный крестик, показывая пользователю, что ответ неверен. В обоих случаях Проверить

сверяется со списком ячейкаУЦентр, чтобы получить правильные координаты для печати изображения (см. упражнение 9.11).

Рис. 9.37. Процедура **Проверить** *спрайта* ДаНет

Другие приложения

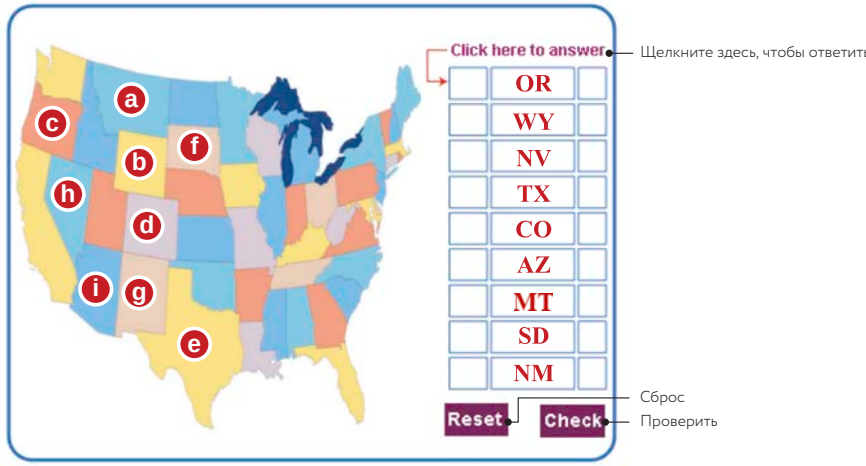
SayThatNumber.sb2

Дополнительные ресурсы, которые можно загрузить с сайта книги (*http://nostarch.com/learnscratch/*), включают еще три приложения, которые вы можете изучить самостоятельно, с подробными объяснениями. Первое — игра для двух человек, основанная на сортировке дробей. Каждому игроку сдаются пять случайно выбранных карт из колоды в 31 карту. Затем каждый игрок получает еще одну карту из оставшихся — и тогда он должен либо отказаться от нее, либо заменить ею одну из своих пяти. Выигрывает тот, кто первым выложит пять своих карт в возрастающем порядке.

УПРАЖНЕНИЕ 9.11

USMapQuiz.sb2

Откройте это приложение и протестируйте его. Придумайте, какие еще тесты по разным предметам можно создать, — и сделайте их. Внизу для примера показан интерфейс теста (файл *USMapQuiz.sb2*). Откройте этот файл и доделайте программу, чтобы тестом можно было пользоваться.



SortEmOut.sb2

Второе приложение — программа, которая записывает числа буквами. Она просит пользователя ввести число, а затем показывает, как оно пишется. Например, если пользователь вводит «3526», программа выдаст «три тысячи пятьсот двадцать шесть». Идея в том, чтобы разбить число справа налево, на группы из трех цифр, и выводить каждую группу с соответствующим словом-множителем («тысяча», «миллион» и т. д.).

Sieve.sb2

Третья программа демонстрирует работу алгоритма «решето Эратосфена»: он ищет все простые числа меньше 100.

Итоги

Списки очень полезны в программировании. Это удобный способ хранить множество элементов. В этой главе мы изучили создание списков в среде Scratch, команды, которые можно использовать при работе с ними, и практиковались в динамическом наполнении списков данными, вводимыми пользователем.

Мы также познакомились с нумерационными списками и узнали, как находить минимальную, максимальную и среднеарифметическую величину их элементов. После этого мы изучили простые алгоритмы поиска и сортировки списков. В конце главы мы поработали с несколькими программами, которые показывают, как можно использовать списки в приложениях.

Задания

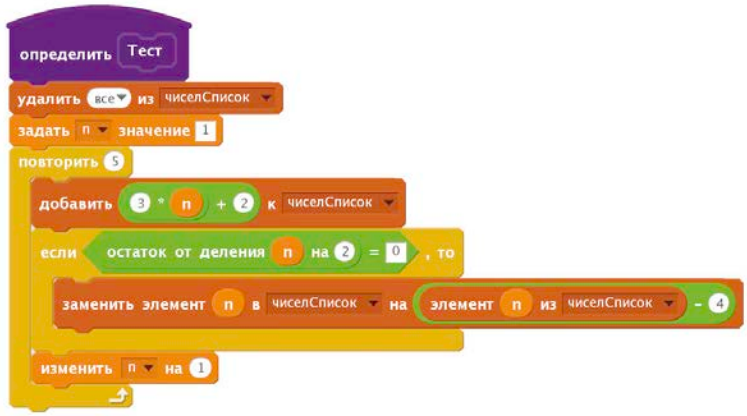
1. Создайте список, в котором содержались бы первые десять простых чисел. Напишите скрипт, который будет отображать эти числа с помощью блока **сказать**.

2. Создайте три списка для хранения личных записей. Пусть в первом хранятся имена, во втором — даты рождения, а в третьем — телефонные

номера. Напишите программу, которая будет запрашивать у пользователя имя человека, информацию о котором нужно найти. Если имя этого человека есть в первом списке, программа назовет его дату рождения и телефонный номер.

3. Создайте два списка для хранения названий продуктов, купленных в магазине, и цен на них. Напишите программу, которая просит пользователя ввести название продукта, а затем отображает его цену — если продукт есть в списке.

4. Что сохраняет список чиселСписок после исполнения скрипта, показанного ниже? Воссоздайте процедуру и запустите ее, чтобы проверить свой ответ.



5. Напишите программу, которая увеличивает вдвое каждый из элементов, сохраненных в нумерованном списке.

6. Напишите программу, которая просит пользователя ввести имена учащихся и их оценки и сохраняет эти данные в двух списках. Пусть программа останавливает сбор данных, когда вместо имени ученика пользователь вводит –1.

7. Напишите программу, которая просит пользователя ввести температурные минимумы и максимумы для всех 12 месяцев года. Пусть введенные величины сохраняются в двух списках.

8. Напишите программу, которая просит пользователя ввести 10 целых чисел. Пусть каждое число сохраняется в списке лишь при условии, что оно не дублирует уже имеющееся в списке число.

9. Напишите программу, которая обрабатывает список 20 результатов теста со 100 вопросами и высчитывает количество учащихся, которые набрали от 85 до 90 баллов.

КРАТКИЙ АНГЛО-РУССКИЙ СЛОВАРЬ SCRATCH

Спрайты

Apple	Яблоко	58
Ball	Мяч, ядро	48, 75, 188, 210
Bat	Летучая мышь	87
Bird	Птица	85, 202
Bit	Разряд	229
Board	Доска	75
Boy	Мальчик	84
Btn	Кнопка	267
Bubble	Пузырь	83
Bullet	Пуля	202
Cannon	Пушка	209
Cat	Кот	48, 49, 87
Cart	Тележка	58
Check	Проверка	237
Checkers	Шашки	104
City	Город	79
Coral	Коралл	83
Current	Ток	134
Dancer	Танцор	75
Die	Кубик	119
Digit	Цифра	237
Down	Вниз	137
Drawer	Чертежник	176
Driver	Драйвер	229, 233
Duck	Утка	85
Equal	Равно	134
Equilateral	Равносторонний	170
Face	Лицо	66
Fire	Огонь	209
Fish	Рыба	83
Flower	Цветок	89, 90, 93, 94, 110
Frog	Лягушка	87
Girl	Девушка, Девочка	84
Gold	Золото	54
Guard	Страж	185
Hangman	Палач	233
Helper	Помощник	226, 233
Hour	Час	200
House	Дом	105
Isosceles	Равнобедренный	170
Kite	Дельтоид	268
Laptop	Ноутбук	192
Letter	Буква	274

Light	Свет	134
Min	Минута	200
Mouse	Мышь	44
New	Здесь: новая игра, новая задача	233, 237
New game	Новая игра	271
Operation	Операция	237
Paddle	Ракетка	32
Painter	Художник	254
Penguin	Пингвин	123
Player	Игрок	54, 119, 185, 188, 226
Point	Точка	176
Read	Читать	237
Rectangle	Прямоугольник	268
Resistance	Сопротивление	134
Rhombus	Ромб	268
Rock	Скала	71
Rocket	Ракета	64, 79
Rose	Роза	141
Scalene	Разносторонний	170
Seal	Тюлень	85
Sec	Секунда	200
Shooter	Стрелок	202, 205
Sphere	Сфера	137
Spotlight	Прожектор	75
Sprite	Спрайт	16
Star	Звезда	167
Starfish	Морская звезда	87
Square	Квадрат	268
Switch	Переключатель	136
Target	Мишень, цель	226
Teacher	Учитель	237
Time	Время	200
Trapezoid	Трапеция	268
Tree	Дерево	83
Tutor	Наставник	159, 176
Up	Вверх	137
Volt	Вольт	134
Wizard	Волшебник	271
Wheel	Колесо	209
YesNo	ДаНет	274

Процедуры

Add	Сложение	240
Blade	Лезвие	199
Branch	Ветка	109
BubbleSort	Пузырьковая Сортировка	264
CheckAnswers	ПроверитьОтветы	171, 226, 228, 268, 270
Checkers	Шашки	104
Circle	Круг	160
Compute	Вычислить	177
CountDown	ОбратныйСчет	199
CreateTerm	СоздатьПараметр	179
Divide	Деление	240
Door	Дверь	107
FillList	Наполнение Списка	261
FindAnswer	НайтиОтвет	190
FindAverage	РасчетСредне-арифметического	258
FindGCD	ВычислитьНОД	240, 242
FindMax	НайтиМаксимум	258
Flower	Цветок	89, 110
GetItemCount	ПосчитатьЭлемент	262
GetPassword	ПолучитьПароль	193
GiveFeedback	ДатьОбратную Связь	240, 242
Hexagon	Шестиугольник	127
House	Дом	107
Initialize	Инициализировать	207, 210, 234
Insert	Вставка	221, 222
Leaf	Лист	108
Leaves	Листья	109
MakeLine1	СозданиеПервой Строки	267
Median	Медиана	265
Multiply	Умножение	240
NewQuestion	НовыйВопрос	171
Parallelogram	Параллелограмм	106
PigLatin	ПоросячьяЛатынь	220
Pins	Спицы	128
Pinwheel	Вертушка	128
Pressure at depth	Давление на глубине	112
ProcessAnswer	ОбработатьОтвет	234, 235
Process CorrectGuess	ОбработатьВерный Ответ	234
Process WrongGuess	Обработать НеверныйОтвет	234, 236

Переменные, параметры, флаги

PinCount	ПодсчетСпиц	128
Point	Координата	169
Pos	Поз	216, 218, 220, 223, 235, 264
PuzzleAnswer	ЗагадкиОтвет	273
PuzzleNum	ЗагадкиНомер	273
QmarkCount	ВопрКол	235
QuadName	ЧетырехугНазв	270
Radius	Радиус	161
Rand	Слч	119
RandChar	СлучСимв	222
RandPos	СлучПоз	222
RemAttempts	ОстПопыток	234, 236
Result	Результат	190
Row	Строка	104
Scale	Масштаб	107
Score	Счет, баллы	55, 59, 258
SecretWord	ЗагаданноеСлово	234, 235
Shape	Фигура	269
Side	Сторона	172
SideLenght	ДлинаСтороны	127
Speed	Скорость	189, 210
StageColor	ЦветСцены	132
Steps	Шаги	103
Str	Стр	223
StrAdd	СтрДоб	221
StrIn	СтрВв	221, 223, 225
StrOut	СтрВыв	221
Sum	Сумма	126, 194, 258
Survey	Опрос	261, 264
Target	Цель	260, 261
TimeLeft	ВремяОсталось	163, 203
Times	Раз	103
Temp	Врем	126, 210, 235, 264
Term	Параметр	179
Theta	Тета	140
Type	Тип	171
VowelCount	СчетГласных	216
Weight	Вес	231
WordList	СписокСлов	225, 234
xPlayer	хИгрок	226
xPos	хПоз	210, 256
yPlayer	уИгрок	226
yPos	уПоз	210

Сообщения

Access denied!	Доступ разрешен!	193
Access granted!	Доступ запрещен!	193
Actual time	Фактическое время	207
BinaryToDecimal	Двоичное вДесятичное	230
Check	Проверить	275
CheckAnswer	ПроверитьОтвет	240
Choooo	Апчхи	70
Correct, but not is simplest form. Try again	Верно, но это не самый простой вариант. Попробуйте еще раз	242
Counting. Please wait	Подсчитываю. Подождите, пожалуйста…	262
Down	Вниз	138
Draw	Рисовать	90, 255
Draw Flower	Рисовать цветок	93
Enter a sentence	Введите предложение	216
Enter an integer	Введите целое число	218
Enter an 8-bit binary number	Введите 8-значное двоичное число	230
Enter correct spelling for	Введите правильное написание для…	222
Enter the base	Введите длину основания	161
Enter the height	Введите высоту	161
Enter the length	Введите длину	161
Enter the width	Введите ширину	161
Enter your answer as a fraction	Введите ответ в виде дроби	239
Enter your choice	Введите свой выбор	160
Enter your first initial	Введите первую букву имени	145
Enter your last initial	Введите первую букву фамилии	145
Fire	Стрелять	210
Game over!	Игра окончена!	130
GetAnswers	ПолучитьОтветы	226, 227
Good job!	Отлично! Молодец!	168, 171, 222

GotAnswer	ЕстьОтвет	239
Guess a letter	Угадай букву	234
How old are you?	Сколько тебе лет?	144
Hmm… You have	Хм… Вы загадали…	273
I am the cute one	Я симпатичный	68
In ten years, you will be	Через десять лет тебе будет	144
Initialize	Инициализировать	230
Invalid choice	Ошибка выбора	160
Invalid! Try again	Ошибка! Попробуйте снова	193
Is a palindrome	Палиндром	218
Is not a palindrome	Не палиндром	218
NewGame	НоваяИгра	226, 234, 273
NewProblem	НоваяЗадача	238
No solution!	Нет решения!	197
No! This is	Нет! Это…	171
Password?	Пароль?	193
Peach	Персик	260
Peach is at position	Персик найден на позиции	260
Peach is not in the list	Персик не в списке	260
Poor ratings	Низкая оценка у…	262
Redraw	Перерисовать	140, 142, 177
Reset	Сбросить	234
Roll	Катиться	122
Same point	Одна и та же точка	177
Sorry! That is not it. Try again	Простите! Это не так. Попробуйте еще раз	222, 225
Sorry. That is incorrect. Try again	Неверный ответ. Попробуйте еще раз	242
Sorry. This point is	Ошибка. Координата…	169
Square	Квадрат	88
StartGame	СтартИгры	226
The average value is	Среднеарифметическое	258
The maximum value is	Максимальное число	258
Unscrumble	Расшифруйте	225
Up	Вверх	138
Update	Обновить	134, 234
What is the x-coordinate?	Какая у меня координата x?	168
What is the y-coordinate?	Какая у меня координата y?	168
What is this triangle?	Какой треугольник?	171
WrongGuess	НевернаяДогадка	236
You are	Тебе	144
You won!	Ты выиграл!	228
Your sentence has	В предложении…	216

Костюмы и фоны

Check	Галочка	275
Costume	Костюм	24
End	Конец	203
Green	Зеленый	67
Leaf	Лист	93
Lose	Поражение	237
Marker	Маркер	207
Off	Здесь: ноль	192, 229
On	Здесь: единица	192, 229
Orange (для светофора)	Желтый	67
Red	Красный	67
Stage	Сцена	67
Start	Старт	203
Win	Победа	237
X	Крестик	275

БЛАГОДАРНОСТИ

На обложке книги указан только один автор, но в ее создании участвовало немало людей. Хочу сказать спасибо многочисленным профессионалам из издательства No Starch Press, которые внесли существенный вклад в эту работу. Отдельная благодарность моему редактору Дженнифер Гриффит-Делгадо и литературному редактору Элисон Лоу за их огромный вклад. Их полезные рекомендации и профессионализм помогли нам сделать книгу еще лучше, а их стремление к идеалу видно на каждой странице. Также хочу поблагодарить Паулу Флеминг и Серену Янг за работу над книгой.

Моя сердечная признательность за бесценные отзывы и рекомендации моему техническому редактору Тайлеру Уоттсу. Его предложения очень обогатили эту книгу.

И наконец, спасибо моей жене Марине и нашим сыновьям Асаду и Караму, которые поддерживали меня во время работы над этим долгим проектом. Им пришлось со многим мириться, чтобы дать мне время и пространство, в которых я так нуждался. Возможно, теперь мне удастся наверстать упущенное!

ОБ АВТОРЕ

Мажед Маржи получил степень доктора технических наук в области проектирования электронных устройств в Университете Уэйна и степень магистра по управлению бизнесом в Давенпортском университете. У него более 15 лет опыта работы в автомобильной промышленности: он создавал программные решения для сбора информации в режиме реального времени, управления устройствами и испытательными стендами, анализа технических данных, встроенных контролеров, телематики, автомобилей с гибридным приводом и трансмиссий с высокими требованиями по технической безопасности.

Также доктор Маржи преподает на электротехническом факультете Университета Уэйна. Помимо прочего, он ведет курсы по технической кибернетике, микропроцессорам, управлению системами, а также алгоритмам и структурам данных.

Издание для досуга
Для среднего и старшего школьного возраста

Мажед **Маржи**

Scratch для детей

Самоучитель по программированию

Главный редактор *Артем Степанов*
Руководитель редакции *Анастасия Троян*
Продюсер проекта *Евгения Рыкалова*
Ответственный редактор *Татьяна Рапопорт*
Литературный редактор *Ольга Свитова*
Научный редактор *Дарья Абрамова*
Верстка обложки *Сергей Хозин*
Корректоры *Надежда Болотина, Лев Зелексон*

НАУЧИТ ПРОГРАММИРОВАТЬ С НУЛЯ

ОТ ВЕДУЩЕГО ИНЖЕНЕРА GENERAL MOTORS

Scratch — это простой, понятный и невероятно веселый язык программирования для детей. В нем нет кодов, которые нужно знать назубок и писать без ошибок. Все, что требуется, — это умение читать и считать. Как из конструктора Lego, при помощи Scratch можно собирать программы из разноцветных «кирпичиков» — блоков. В программу можно вносить любые изменения в любой момент и сразу видеть, как она работает. Подробные объяснения, разобранные по шагам примеры и множество упражнений помогут освоить Scratch без труда.

Эта книга подойдет детям от 8 лет (и их родителям!), а также всем, кто хочет научиться программировать с нуля.

Вы узнаете, как:

— использовать основные процедуры в Scratch: логические операторы, переменные для записи информации, функции и командные блоки;
— считывать, сохранять и обрабатывать данные;
— самостоятельно писать программы;
— создавать разнообразные игры, мультфильмы, презентации, слайд-шоу.

Автор книги Мажед Маржи — старший разработчик в General Motors и преподаватель в Университете Уэйна в Мичигане. Мажед написал эту книгу, чтобы показать возможности Scratch как мощного инструмента для изучения основ программирования.

МИФ Детство

Детские книги на сайте
mann-ivanov-ferber.ru

facebook.com/mifdetstvo
vk.com/mifdetstvo
instagram.com/mifdetstvo

no starch press

ISBN 978-5-00100-336-6