Знакомство с программированием на языке Processing

Практическое введение в создание интерактивной графики

Кейси Риас, Бен Фрай

ДМК издательство

Кейси Риас, Бен Фрай

Знакомство с программированием на языке Processing

Getting Started with Processing

Second edition

Casey Reas and Ben Fry

Знакомство с программированием на языке Processing

Кейси Риас и Бен Фрай

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2021

УДК 004.4Processing
ББК 32.972
Р49

Риас К., Фрай Б.
Р49 Знакомство с программированием на языке Processing / пер. с англ. В. С. Яценкова. – М.: ДМК Пресс, 2021. – 194 с.: ил.

ISBN 978-5-97060-950-7

Это руководство по языку Processing написано его создателями, Кейси Риасом и Беном Фраем. Книга удобно структурирована и ведет читателя от знакомства с языком и написания первой программы на нем до разработки интерактивной графики.

Главы книги последовательно раскрывают основные приемы программирования на Processing: определение и рисование простых фигур; хранение, изменение и повторное использование данных; управление выполнением программы с помощью мыши и клавиатуры; преобразование координат; загрузка и отображение мультимедийных файлов и др. Авторы используют метод «обучение через практику»: в каждой главе приводится ряд подробных примеров выполнения тех или иных задач (иллюстрация, описание, код). В приложениях представлен справочный материал.

Книга пригодится тем, кто хочет научиться создавать компьютерную графику и простые интерактивные программы. Благодаря простой и ясной манере изложения она подойдет как для новичков, так и для читателей с опытом программирования, которые планируют освоить интерактивную графику для создания игр, анимации и интерфейсов.

УДК 004.4Processing
ББК 32.972

ISBN 978-1-457-18708-7 (англ.)
ISBN 978-5-97060-950-7 (рус.)

Отзывы о книге

«Раньше создать компьютерную программу было столь же просто, как включить компьютер и набрать одну или две строки кода, чтобы на экране появилось слово “Привет”. Теперь для этого требуется руководство в 500 страниц и целая команда программистов. Или не все так плохо? Благодаря этой небольшой книге Бена и Кейси вы сможете рисовать линии, треугольники и круги при помощи компьютерной программы спустя несколько минут после включения компьютера. Они снова сделали компьютерные программы простыми и понятными для обычного человека – а это немалое достижение».

– Джон Маэда, президент школы дизайна Род-Айленда

«Это не только простое введение в основы программирования – это еще и весело! Книга похожа на рабочую тетрадь для взрослых. Вам захочется купить ее, даже если вы никогда не думали, что заинтересуетесь программированием, – потому что вы обязательно заинтересуетесь».

– Марк Аллен, основатель и директор Machine Project

«Это отличный учебник для тех, кто хочет окунуться в программирование графики. Методика “обучение через практику” делает его особенно подходящим для художников и дизайнеров, которых часто отталкивают более традиционные подходы, основанные на теории. Доступная стоимость книги и тот факт, что среда Processing имеет открытый исходный код, делают ее отличным выбором для студентов».

– Джиллиан Крэмптон Смит,
профессор дизайна Fondazione Venezia,
Венецианский университет IUAV

«Processing радикально изменил способ обучения программированию, и это один из основных факторов успеха Arduino».

– Массимо Банци, соучредитель Фонда Arduino

«Кейси Риас и Бен Фрай в своей книге раскрыли захватывающую силу программирования для творческих людей. Риас и Фрай ясны и прямолинейны, но как художники они не боятся быть немного эксцентричными и нестандартными. Это делает их способ обучения уникальным и эффективным».

– Холли Уиллис, директор академических программ,
Институт мультимедийного образования,
Школа кинематографических искусств, USC

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Maker Media очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Предисловие

Мы создали Processing, чтобы упростить программирование интерактивной графики. Мы были разочарованы тем, насколько сложно было написать этот тип программного обеспечения на языках программирования, которые мы обычно использовали (Java и C++), и были вдохновлены тем, насколько просто было писать интересные программы на языках нашего детства (Logo и BASIC). На нас больше всего повлиял созданный нашим научным руководителем Джоном Маэдой[1] язык Design By Numbers (DBN), который мы поддерживали и преподавали в то время.

Processing родился весной 2001 года как результат мозгового штурма на листке бумаги. Нашей целью было создание прототипа такого типа программного обеспечения, которое почти всегда было полноэкранным и интерактивным. Мы искали лучший способ легко проверить наши идеи в коде, а не просто говорить о них или тратить слишком много времени на их программирование на C++. Другой нашей целью было создать язык для обучения студентов-дизайнеров искусству программирования и в то же время дать большему количеству технических студентов простой способ работы с графикой. Эта комбинация отличается от того, как обычно преподают программирование. Мы решили, что нужно начинать с графики и взаимодействия, а не структур данных и вывода текстовой консоли.

Processing пережил долгое детство; это была альфа-версия с августа 2002 года по апрель 2005 года, а затем публичная бета-версия до ноября 2008 года. В течение этого времени его постоянно использовали в учебных классах тысячи людей по всему миру. При этом сам язык, среда разработки и учебные программы постоянно пересматривались. Многие из наших первоначальных решений относительно языка были подтверждены, а многие изменены или отвергнуты. Мы разработали систему программных расширений, называемых библиотеками, которые позволили людям применить Processing во многих неожиданных и удивительных проектах. (Сейчас существует более 100 библиотек.)

Осенью 2008 года мы запустили версию 1.0 программы. После семи лет работы запуск 1.0 означал долгожданную стабильность. Весной 2013 года мы выпустили версию 2.0, работающую заметно быстрее. В этой версии улучшена интеграция с OpenGL, шейдерами GLSL и ускорено воспроизведение видео с помощью GStreamer. Выпущенная в 2015 году версия 3.0 облегчает программирование в Processing благодаря новому интерфейсу и проверке ошибок.

Теперь, спустя четырнадцать лет после своего появления, Processing вышел за рамки своего первоначального предназначения, и мы узнали, как он может быть полезен в других контекстах. Соответственно, эта книга написа-

---

[1] Тот самый Джон Маэда – всемирно известный гуру дизайна, идеолог максимальной простоты, лаконичности и удобства в дизайне и технологиях. – *Прим. перев.*

на для новой аудитории – специалистов в других областях, которым нужно программировать время от времени, любителей и всех, кто хочет изучить, на что способен Processing, не теряясь в деталях огромного учебника. Мы надеемся, что вам понравится и вы решите продолжить занятия программированием. Эта книга – только начало.

Хотя мы (Кейси и Бен) ведем корабль Processing по водам последние два десятка лет, мы не можем переоценить тот факт, что Processing – это плод коллективных усилий сообщества. От написания библиотек, расширяющих возможности языка, до публикации кода в интернете и помощи другим в обучении сообщество пользователей Processing продвинуло его далеко за пределы первоначальной концепции. Без этих коллективных усилий Processing не был бы тем, чем он является сегодня.

Краткое содержание книги

В главах этой книги раскрыты следующие темы:

- глава 1: знакомство с языком Processing;
- глава 2: первая программа на языке Processing;
- глава 3: определение и рисование простых фигур;
- глава 4: хранение, изменение и повторное использование данных;
- глава 5: управление выполнением программы с помощью мыши и клавиатуры;
- глава 6: преобразование координат;
- глава 7: загрузка и отображение мультимедийных файлов, включая шрифты, растровые и векторные изображения;
- глава 8: перемещение и взаимосвязанное движение фигур;
- глава 9: создание новых модулей кода;
- глава 10: создание модулей кода, сочетающих переменные и функции;
- глава 11: упрощение работы со списками переменных;
- глава 12: загрузка и визуализация данных;
- глава 13: работа с 3D, экспорт PDF, компьютерное зрение и чтение данных с платы Arduino.

Кому адресована эта книга

Эта книга написана для людей, которые нуждаются в кратком и понятном введении в программирование, чтобы научиться создавать компьютерную графику и простые интерактивные программы. Она адресована читателям, которые хотят разобраться в тысячах примеров бесплатного кода на языке Processing и справочных материалов, доступных в интернете. «Знакомство с программированием на языке Processing» – это не учебник по программированию; это именно первое знакомство с ним. Эта книга предназначена для подростков, любителей технического творчества, их бабушек и дедушек и всех остальных.

Книга также подходит для читателей с опытом программирования, желающих изучить основы интерактивной компьютерной графики. В этой книге они найдут описание методов, которые можно применять для создания игр, анимации и интерфейсов.

Условные обозначения и соглашения, принятые в книге

В книге используются следующие типографские соглашения.

Курсив – используется для смыслового выделения важных положений, новых терминов, имен команд и утилит, а также слов и предложений на естественном языке.

`Моноширинный шрифт` применяется для листингов программ, а также в обычном тексте для обозначения имен переменных, функций, типов, объектов, баз данных, переменных среды, операторов, ключевых слов и других программных конструкций и элементов исходного кода.

`Моноширинный курсив` применяется для обозначения в исходном коде или в командах шаблонных меток-заполнителей, которые должны быть заменены соответствующими контексту реальными значениями.

Этот элемент обозначает совет, предложение или общее примечание.

Этот элемент указывает на предупреждение или предостережение.

Благодарности

За первое и второе издания этой книги мы благодарим Брайана Джепсона за его огромную энергию, поддержку и понимание. В первом издании Нэнси Котари, Рэйчел Монаган и Сумита Мукхерджи умело довели книгу до финиша. Том Сгоурос тщательно отредактировал книгу, а Дэвид Хамфри представил содержательный технический обзор.

Мы не можем представить эту книгу без книги Массимо Банци *Getting Started With Arduino*. (В России издана под названием «Arduino для начинающих волшебников».) Превосходная книга Массимо – наш образец для подражания.

Небольшая группа людей годами вкладывала время и энергию в Processing. Дэн Шиффман – наш партнер в Processing Foundation, организации, которая поддерживает программное обеспечение Processing. Большая часть основного кода для Processing 2.0 и 3.0 рождена острыми умами Андреса Колубри и Маниндры Мохараны. Скотт Мюррей, Джейми Косой и Джон Гакник создали замечательную веб-инфраструктуру для этого проекта. Джеймс Грэди развивает пользовательский интерфейс версии 3.0. Мы благодарим Флориана Дженетта за его годы разнообразной работы над проектом, включая

форумы, веб-сайт и дизайн. Эли Зананири и Андреас Шлегель разработали инфраструктуру для подключения и документирования сторонних библиотек и потратили бесчисленные часы на составление списков. Многие пользователи внесли значительный вклад в проект; точные данные доступны по адресу https://github.com/processing.

Релиз Processing 1.0 был поддержан Университетом Майами и Oblong Industries. Институт интерактивных медиаисследований Армстронга при Университете Майами профинансировал Оксфордский проект – серию семинаров по разработке программ на языке Processing. Эти семинары стали возможными благодаря упорному труду Иры Гринберг. Четырехдневные конференции в Оксфорде, штат Огайо, и Питтсбурге, штат Пенсильвания, позволили в ноябре 2008 года запустить Processing 1.0. Летом 2008 года компания Oblong Industries профинансировала Бену Фраю разработку проекта Processing; это было важно для выпуска стабильной версии.

Выпуску Processing 2.0 способствовал семинар по развитию, спонсируемый Программой интерактивных телекоммуникаций Нью-Йоркского университета. Работа над Processing 3.0 была щедро спонсирована программой Emergent Digital Practices в Университете Денвера. Мы благодарим Кристофера Колемена и Лале Мехран за неоценимую поддержку.

Эта книга выросла из лекций по языку Processing в Калифорнийском университете в Лос-Анджелесе. Чендлер МакВильямс сыграл важную роль в создании учебного курса. Кейси благодарит студентов факультета дизайна и медиаискусства Калифорнийского университета в Лос-Анджелесе за их энергию и энтузиазм. Его ассистенты были отличными соавторами в разработке учебного плана. Снимаю шляпу перед Тацуей Сайто, Джоном Хоуком, Тайлером Адамсом, Аароном Сигелом, Кейси Альт, Андресом Колубри, Майклом Контопулосом, Дэвидом Эллиотом, Христо Аллегрой, Питом Хоуксом и Лорен Маккарти.

Все это стало возможным благодаря тому, что Джон Маэда основал группу компьютерной эстетики (1996–2002) в MIT Media Lab.

Глава 1

Знакомство с языком Processing

Processing предназначен для написания программ, которые позволяют создать анимированное изображение и взаимодействовать с ним. Напишите одну строку кода – и на экране появится кружок. Добавьте еще несколько строк кода – и кружок будет следовать за курсором мыши. Еще одна строка кода – и кружок меняет цвет при нажатии на кнопку мыши. Мы называем эти короткие наброски кода *скетчами*[1]. Вы пишете одну строку, к ней добавляете другую и смотрите, что получилось. Затем добавляете новые строки и так далее. В результате программа создается по частям.

Курсы программирования традиционно сосредоточены на структуре и теории выбранного языка. Все, что касается визуального оформления, – интерфейс, анимация – считается десертом, которым можно наслаждаться только после того, как вы съели отварные овощи, обычно спустя несколько недель изучения алгоритмов и методов. На протяжении многих лет мы наблюдали, как наши друзья пытались пройти такие курсы и бросали учебу после первой же лекции или после долгой бесплодной ночи перед крайним сроком выполнения первого задания. Их стремление заставить компьютер выполнять полезную работу быстро угасало, потому что они не видели прямой связи между тем, чему их учили, и тем, что они хотели получить.

Processing предлагает изучать программирование посредством создания интерактивной графики. Существуют разные методики обучения программированию, но для начинающих программистов наилучшей мотивацией чаще всего служит возможность немедленно увидеть на экране эффектный результат. Именно способность языка Processing предоставлять визуальную обратную связь буквально с первой строки кода сделала его столь популярным во всем мире средством обучения программированию. Давайте кратко обсудим основные подходы к программированию на языке Processing.

---

[1] Sketch – набросок, эскиз. Все, кому доводилось программировать для Arduino, уже знакомы с этим термином. Более того, Arduino и Processing – родные братья, и у них одинаковая среда разработки кода. Поэтому мы будем использовать в этой книге привычный многим читателям термин «скетч», хотя в интерфейсе среды разработки Processing он переведен как «набросок». – *Прим. перев.*

1.1. Скетчи и прототипы

Скетчи – это особый способ мышления; с ними можно программировать весело и быстро. Основная цель – изучить множество идей за короткий промежуток времени. Приступая к разработке программы, мы обычно начинаем с набросков алгоритма на бумаге, а затем переносим свои идеи в код. Идеи анимации и взаимодействия с пользователем обычно набрасываются в виде раскадровки с примечаниями. После создания набросков программного обеспечения лучшие идеи отбираются и объединяются в *прототипы* (рис. 1.1). Это циклический процесс разработки, тестирования и улучшения, когда вы перемещаетесь между бумагой и экраном.

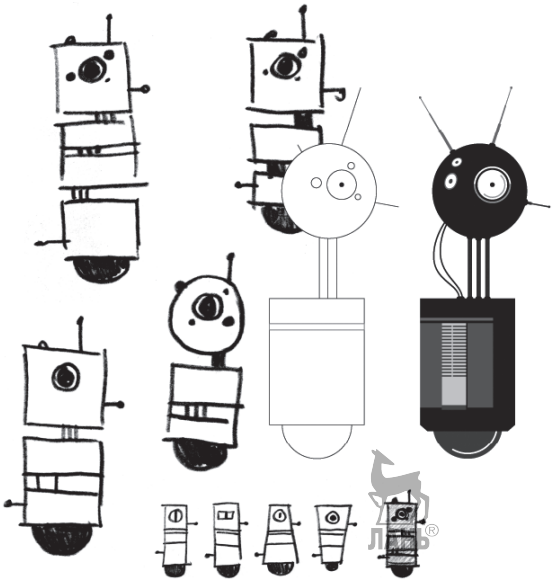


Рис. 1.1 ❖ По мере того как эскизы перемещаются из блокнота на экран, их облик становится все более детализованным

1.2. Гибкость

Подобно ящику со слесарными приспособлениями, Processing состоит из множества инструментов, которые можно использовать в разных комбинациях. В результате его можно применять как для быстрых набросков кода, так и для серьезного сложного программирования. Программа на языке Processing может состоять из одной строки или тысяч строк, поэтому у вас есть простор для роста и выбора вариантов. Более 100 специализированных библиотек расширяют возможности Processing в таких областях, как звук, компьютерное зрение и генерация цифровых объектов (рис. 1.2).

Рис. 1.2 ❖ Processing может оперировать разнообразными типами данных

1.3. Гиганты прошлого

Люди создают изображения с помощью компьютеров с 1960-х годов, и в истории компьютерной графики можно найти много интересного. Например, до того, как компьютеры смогли отображать изображения на ЭЛТ или ЖК-экранах, для рисования изображений применялись огромные плоттеры (рис. 1.3). В жизни мы все в том или ином смысле стоим на плечах гигантов, а среди гигантов обработки данных есть мыслители из области дизайна, компьютерной графики, искусства, архитектуры, статистики и смежных областей. Взгляните, например, на альбом Ивана Сазерленда *Sketchpad* (1963), *Dynabook* Алана Кея (1968) и работы многих художников, представленных в книге Рут Левитт «Художник и компьютер» (*Artist and Computer*, Harmony Books, 1976). Архивы ACM SIGGRAPH и Ars Electronica позволяют заглянуть в историю графики и программного обеспечения.

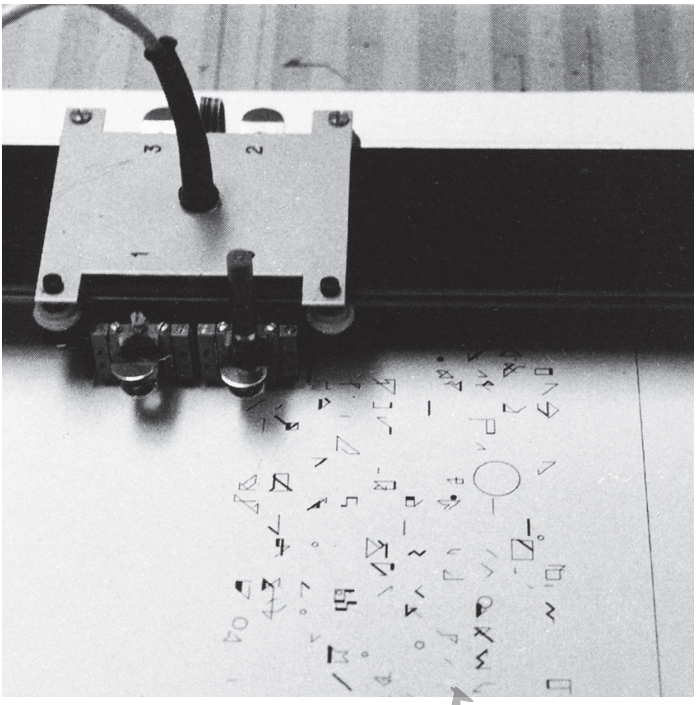


Рис. 1.3 ❖ Демонстрационный рисунок Манфреда Мора, представленный в Музее современного искусства в Париже, выполнен с использованием плоттера Benson и компьютера 11 мая 1971 г. (фото Райнера Мюрле любезно предоставлено галереей *bitforms*, Нью-Йорк)

1.4. Генеалогическое древо языков

Языки программирования, как и человеческие языки, образуют семейства родственных языков. Processing – это диалект языка программирования Java; синтаксис этих языков почти идентичен, но Processing добавляет специальные функции, связанные с графикой и визуальным взаимодействием (рис. 1.4). Графические элементы Processing связаны с PostScript (основа PDF) и OpenGL (спецификация трехмерной графики). Благодаря этим родственным связям изучение языка Processing – это первый шаг к программированию на других языках и с использованием различных программных инструментов.

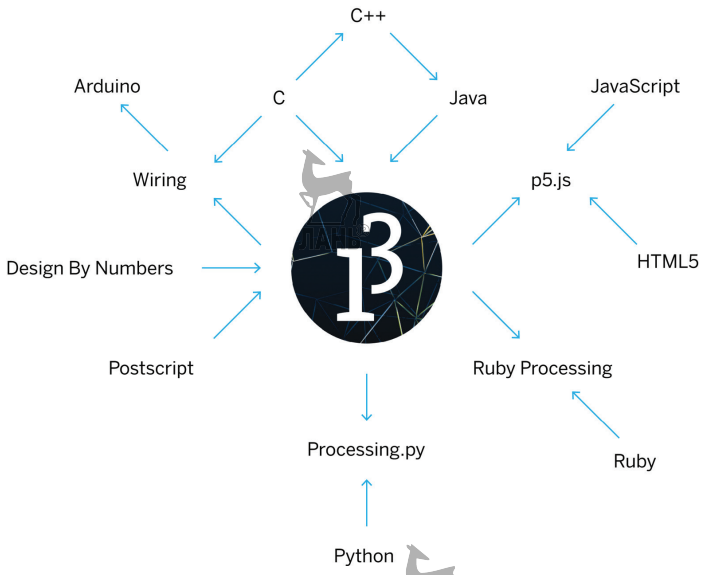


Рис. 1.4 ❖ У Processing есть большое семейство родственных языков и сред разработки

1.5. Присоединяйтесь к сообществу!

Тысячи людей по всему миру используют Processing каждый день. Как и они, вы можете бесплатно скачать Processing. У вас даже есть возможность изменить исходный код Processing в соответствии с вашими потребностями. Processing – это бесплатное программное обеспечение с открытым исходным кодом, и мы призываем вас участвовать в жизни сообщества, делясь своими проектами и знаниями в интернете на сайте Processing.org и на различных сайтах социальных сетей, где есть группы, посвященные языку Processing. Ссылки можно найти на сайте Processing.org.

Глава 2

Начинаем программировать

Чтобы извлечь максимальную пользу из этой книги, недостаточно просто ее прочитать. Вам нужно экспериментировать и практиковаться. Вы не можете научиться программировать, просто читая об этом, – вам нужно это делать. Для начала скачайте Processing и напишите свой первый скетч.

Первым делом зайдите на сайт http://processing.org/download и выберите версию для Mac, Windows или Linux, в зависимости от того, какой у вас компьютер[1]. Процедура установки среды разработки Processing чрезвычайно проста:

- версия для Windows распространяется в виде архивного файла *.zip*. Дважды щелкните по нему и перетащите расположенную внутри папку в любое место на жестком диске. Это может быть папка Program Files или просто рабочий стол, важно только, чтобы папка с файлами Processing была извлечена из скачанного архива. Затем дважды щелкните по файлу *processing.exe*, чтобы запустить Processing;
- версия для Mac OS X – это тоже файл *.zip*. Дважды щелкните по нему и перетащите значок *Processing* в папку *Applications* (Приложения). Если вы используете чужой компьютер и не можете изменить эту папку, просто перетащите приложение на рабочий стол. Затем дважды щелкните по иконке *Processing*, чтобы запустить Processing;
- версия для Linux представляет собой файл *.tar.gz*, который должен быть знаком большинству пользователей Linux. Загрузите файл в домашний каталог, затем откройте окно терминала и введите:

  ```
  tar xvfz processing-xxxx.tgz
  ```

  (Замените символы xxxx номером версии.) Будет создан каталог с именем `processing-3.5.4` или наподобие этого. Затем перейдите в этот каталог:

  ```
  cd processing-xxxx
  ```

  и запустите Processing:

  ```
  ./processing
  ```

[1] На момент работы над переводом была доступна версия Processing 3.5.4, поэтому все снимки экрана и пункты меню будут представлены для этой версии. – *Прим. перев.*

Если вы все сделали правильно, то увидите главное окно среды разработки Processing (рис. 2.1). Если же программа не запустилась или возникла какая-то другая проблема, прочитайте об устранении неполадок по адресу http://bit.ly/process-wiki/.

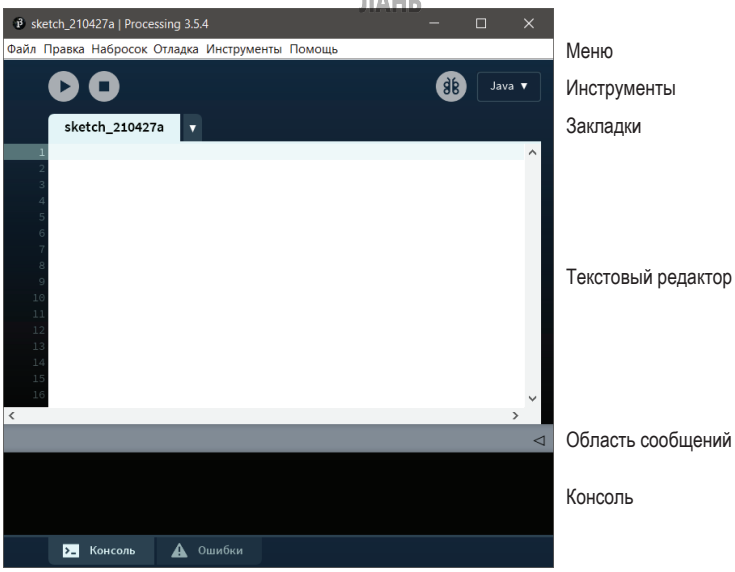


Рис. 2.1 ❖ Среда разработки Processing

2.1. Ваша первая программа

Итак, у вас запущена среда разработки Processing (или Processing development environment, PDE). В ней нет ничего особенного; большая область – это *текстовый редактор*, а в верхней части расположены две кнопки; это *панель инструментов*. Под редактором находится *область сообщений*, а еще ниже – *консоль*. Область сообщений используется для вывода однострочных сообщений, а консоль используется для многострочного текстового вывода.

Пример 2.1. Рисование эллипса

В редакторе введите следующий текст:

```
ellipse(50, 50, 80, 80);
```

Эта строка кода означает «нарисовать эллипс с центром, расположенным на 50 пикселей вправо и на 50 пикселей вниз относительно верхнего левого угла, с шириной и высотой 80 пикселей». Нажмите кнопку **Запустить** (кнопка с треугольником на панели инструментов).

Если вы ввели код правильно, на экране появится кружок. Если вы ввели его неправильно, область сообщений станет красной и появится сообщение об ошибке. В этом случае убедитесь, что вы точно скопировали пример кода: числа должны быть заключены в круглые скобки и разделены запятыми, а строка должна заканчиваться точкой с запятой.

Одна из самых сложных вещей, связанных с началом программирования, – это то, что вы должны очень точно соблюдать требования синтаксиса. Processing, как и другие инструменты программирования, не настолько умный, чтобы догадываться, что вы имеете в виду, и может быть очень придирчивым к расстановке знаков препинания. Вы быстро привыкнете к этому после небольшой тренировки.

Далее мы перейдем к более интересному скетчу.

Пример 2.2. Рисование кругов

Удалите из окна редактора текст предыдущего примера и введите новый:

```
void setup() {
  size(480, 120);
}

void draw() {
  if (mousePressed) {
    fill(0);
  } else {
    fill(255);
  }
  ellipse(mouseX, mouseY, 80, 80);
}
```

Эта программа создает рабочее окно скетча шириной 480 пикселей и высотой 120 пикселей, а затем начинает рисовать в нем белые круги с центром в том месте, куда указывает курсор мыши. При нажатии кнопки мыши цвет круга меняется на черный. Подробнее об этой программе мы расскажем позже. А пока запустите код, перемещайте указатель мыши над рабочим окном и периодически нажимайте левую кнопку мыши, чтобы увидеть, что будет происходить. Справа от кнопки **Запустить** на панели нструментов находится кнопка **Остановить** со значком в виде квадрата, на который вы можете на-

жать, чтобы остановить выполнение скетча. Вы можете также закрыть рабочее окно скетча, как вы это обычно делаете с любым другим приложением.

2.2. Режим отображения рабочего окна

Если вы не хотите использовать кнопки на панели инструментов, вы всегда можете обратиться к меню **Набросок** (Скетч), в котором есть сочетание клавиш **Ctrl**-**R** (или **Cmd**-**R** на Mac) для запуска скетча. Опция **Режим презентации** перед запуском скетча очищает экран, чтобы на нем осталось только рабочее окно скетча. Вы также можете перейти в **Режим презентации**, удерживая нажатой клавишу **Shift** при нажатии кнопки **Запустить**.

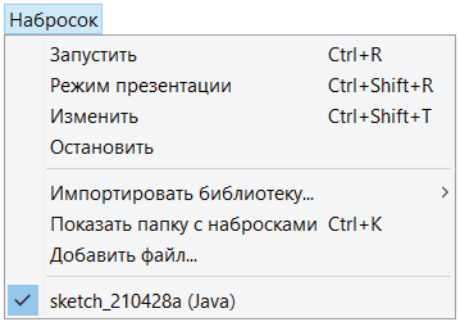


Рис. 2.2 ❖ Скетч можно выполнить при помощи опций **Запустить** и **Режим презентации**. В режиме презентации экран полностью очищается перед запуском скетча

2.3. Создание и сохранение нового скетча

Следующая важная команда – **Сохранить**. Вы можете найти ее в меню **Файл**. По умолчанию ваши программы сохраняются в папке Processing, внутри которой при сохранении скетча автоматически создается вложенная папка, название которой совпадает с названием скетча. Выберите пункт **Рабочая папка** в меню **Файл**, чтобы открыть список всех скетчей в вашей рабочей папке.

Всегда полезно почаще сохранять свои программы. Пробуя разные изменения кода, сохраняйте их под разными именами, чтобы всегда можно было вернуться к более ранней версии. Это особенно полезно, если... нет, *когда* что-то пойдет не так. Вы также можете увидеть, где находится скетч на вашем компьютере, с помощью команды **Показать папку с набросками** в меню **Наброски**.

Вы можете создать новый скетч, выбрав опцию **Создать** в меню **Файл**. Будет создан новый пустой скетч в отдельном окне.

2.4. Распространение программы

Написав программу, вы можете поделиться ей с другими пользователями. Опция **Экспорт приложения** в меню **Файл** объединит ваш код в одну папку, которой вы сможете поделиться с другими пользователями. Можно создать приложения для Mac OS X, Windows и/или Linux по вашему выбору. Это простой способ сделать автономные версии ваших проектов, которые можно запускать двойным щелчком по имени файла в полноэкранном режиме или в окне. Если вы не уверены, что на компьютерах пользователей, с которыми вы делитесь приложением, установлена среда выполнения Java SE, установите в окне **Настройка экспорта** галочку **Встроить Java для Windows**.

Папки приложения удаляются и создаются заново каждый раз, когда вы используете команду **Экспорт приложения**, поэтому обязательно переместите готовую папку в другое место, если вы не хотите, чтобы она была удалена при следующем экспорте.

2.5. Примеры и ссылки

Чтобы научиться программировать, нужно пропустить через свои руки большой объем кода: запускать, изменять, портить и исправлять его до тех пор, пока у вас не получится что-то новое. Именно поэтому доступные для скачивания файлы готовых скетчей Processing включают в себя десятки примеров, демонстрирующих различные опции этого языка.

Чтобы открыть пример, выберите опцию **Примеры** в меню **Файл** и дважды щелкните по имени примера. Примеры сгруппированы по категориям в зависимости от их функции, например **Форма**, **Движение** и **Изображение**. Найдите в списке интересную тему и попробуйте запустить скетч.

Все примеры из этой книги можно загрузить и запустить непосредственно из среды разработки Processing. Выберите опцию **Примеры** в меню **Файл**, затем нажмите кнопку **Добавить...**, чтобы открыть список пакетов с примерами, доступных для загрузки. Выберите пакет **Getting Started with Processing, 2nd Edition** и нажмите кнопку **Install** (Установить).

Взглянув на код в редакторе, вы увидите, что такие функции, как `ellipse()` и `fill()`, окрашены в другой цвет на фоне остального текста. Если вы видите незнакомую функцию, выделите окрашенный текст и нажмите опцию **Найти в документации** в меню **Помощь**. Вы также можете щелкнуть по тексту правой кнопкой мыши (или при нажатой клавише **Ctrl** на Mac) и выбрать опцию **Найти в документации** в появившемся контекстном меню. После этого в веб-браузере откроется страница с описанием функции. Кроме того, вы можете просмотреть полную документацию, выбрав опцию **Документация** в меню **Помощь**.

Справочник по языку Processing объясняет каждый элемент кода с описанием и примерами. Программы в справочнике очень короткие (обычно

четыре или пять строк), и их намного проще понять, чем более длинный код примеров. Мы рекомендуем держать справочник открытым во время чтения этой книги и программирования. Вы можете перемещаться в справочнике по темам или в алфавитном порядке; иногда быстрее всего выполнить текстовый поиск в окне браузера.

Справочник был написан для новичков; мы надеемся, что сделали его ясным и понятным. Мы благодарны множеству людей, которые все прошедшие годы замечали ошибки и сообщали о них. Если вы думаете, что можете улучшить статью в справочнике или обнаружили ошибку, сообщите нам об этом, перейдя по ссылке вверху каждой справочной страницы.

Глава 3

Рисование

Прежде всего рисование на экране компьютера похоже на работу с миллиметровой бумагой. Поначалу оно выглядит как кропотливая техническая процедура, но по мере усвоения новых приемов программирования вы переходите от рисования простых фигур к анимации и взаимодействию с пользователем. Однако прежде чем совершить этот переход, вам нужно изучить основы.

Экран компьютера представляет собой сетку светящихся элементов, называемых пикселями. У каждого пикселя есть позиция в сетке, определяемая числовыми координатами. В языке Processing, как и в большинстве других языков, координата `x` – это расстояние от левого края рабочего окна, а координата `y` – это расстояние от верхнего края. Мы записываем координаты пикселя так: `(x, y)`. Итак, если размер рабочего окна программы 200×200 пикселей, координаты верхнего левого угла равны `(0, 0)`, центр находится в точке с координатами `(100, 100)`, а нижний правый угол имеет координаты `(199, 199)`. Эти цифры могут сбить вас с толку; почему мы считаем от 0 до 199, вместо того чтобы считать от 1 до 200? Дело в том, что в программировании принято начинать отсчет от 0, потому что это проще для вычислений, о которых мы поговорим позже.

3.1. Рабочее окно

Рабочее окно программы создается, и изображения внутри него рисуются с помощью элементов кода, называемых функциями. *Функции* – это основные строительные блоки программы на языке Processing. Поведение функции определяется ее параметрами. Например, почти каждая программа Processing содержит функцию `size()` для установки ширины и высоты окна отображения. (Если в вашей программе отсутствует функция `size()`, размер рабочего окна устанавливается равным 100×100 пикселей.)

Пример 3.1. Рисование рабочего окна

Функция `size()` имеет два параметра: первый устанавливает *ширину* окна, а второй – его *высоту*. Чтобы отобразить окно шириной 800 пикселей и высотой 600 пикселей, наберите в редакторе единственную строку:

```
size(800, 600);
```

Запустите эту строку кода, чтобы увидеть результат. Задайте разные значения, чтобы увидеть, что получится. Попробуйте использовать очень маленькие числа и числа, превышающие размер дисплея вашего компьютера.

Пример 3.2. Рисование точки

Чтобы задать цвет отдельного пикселя в рабочем окне, мы используем функцию `point()`. У нее есть два параметра, которые определяют позицию пикселя: координата `x`, за которой следует координата `y`. Чтобы нарисовать маленькое окошко и точку в центре экрана с координатами (240, 60), введите:

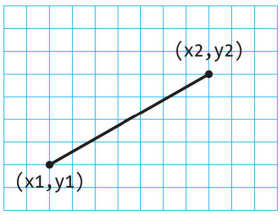

```
size(480, 120);
point(240, 60);
```

Попробуйте написать программу, которая ставит точки в каждом углу рабочего окна и одну точку в центре. Попробуйте расположить точки рядом, чтобы они образовали горизонтальные, вертикальные и диагональные линии.

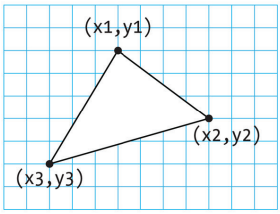
3.2. Основные фигуры

Processing располагает набором функций для рисования основных фигур (рис. 3.1). Простые фигуры, такие как линии, можно комбинировать для рисования более сложных форм, таких как лист или даже лицо.

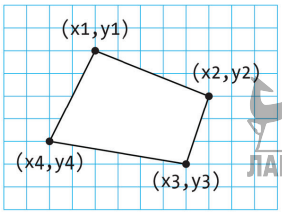
Чтобы нарисовать одиночную линию, нам нужно четыре параметра: два для начальной точки и два для конечной.



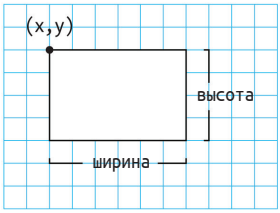
line(x1, y1, x2, y2)



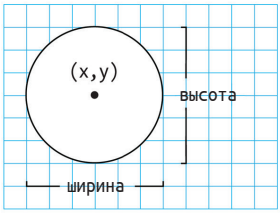
triangle(x1, y1, x2, y2, x3, y3)



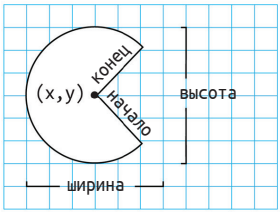
quad(x1, y1, x2, y2, x3, y3, x4, y4)



rect(x, y, ширина, высота)



ellipse(x, y, ширина, высота)

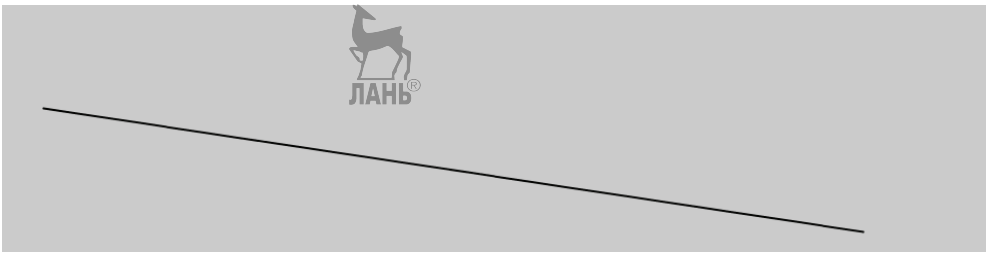


arc(x, y, ширина, высота, начало, конец)

Рис. 3.1 ❖ Основные фигуры и их координаты

Пример 3.3. Рисование линии

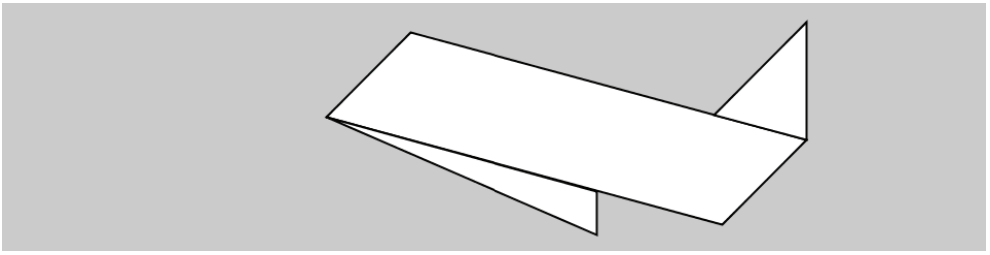
Чтобы нарисовать линию между точками с координатами `(20, 50)` и `(420, 110)`, введите и запустите следующий код:



```
size(480, 120);
line(20, 50, 420, 110);
```

Пример 3.4. Рисование основных фигур

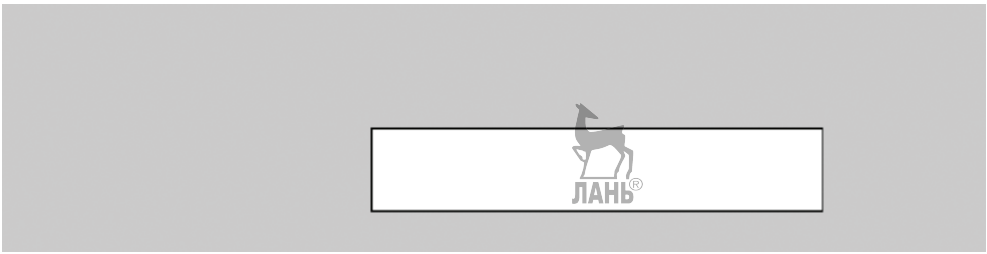
Если развить это правило, треугольнику нужно шесть параметров, а четырехугольнику – восемь (по одной паре на каждую точку):



```
size(480, 120);
quad(158, 55, 199, 14, 392, 66, 351, 107);
triangle(347, 54, 392, 9, 392, 66);
triangle(158, 55, 290, 91, 290, 112);
```

Пример 3.5. Рисование прямоугольника

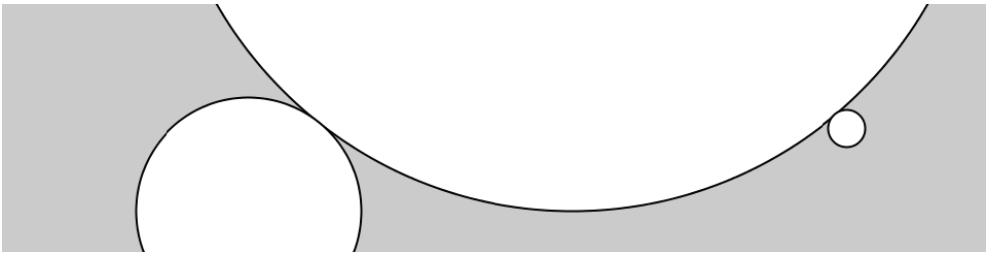
Прямоугольники и эллипсы определяются четырьмя параметрами: первый и второй задают координаты `x` и `y` *точки привязки*, третий задает ширину, четвертый – высоту. Чтобы создать прямоугольник, привязанный к точке `(180, 60)`, с шириной `220` пикселей и высотой `40`, используйте функцию `rect()` следующим образом:



```
size(480, 120);
rect(180, 60, 220, 40);
```

Пример 3.6. Рисование эллипса

Координаты *x* и *y* прямоугольника – это левый верхний угол, а для эллипса – центр формы. В этом примере обратите внимание, что координата *y* для первого эллипса находится за пределами окна. Объекты могут частично (или полностью) выйти за пределы рабочего окна, и это не приведет к ошибке выполнения программы:

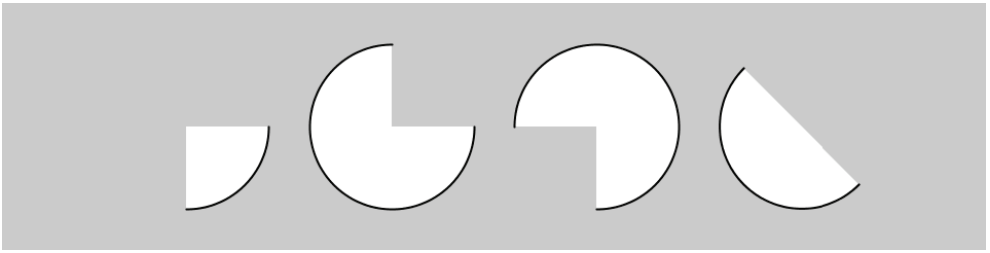


```
size(480, 120);
ellipse(278, -100, 400, 400);
ellipse(120, 100, 110, 110);
ellipse(412, 60, 18, 18);
```

Processing не имеет отдельных функций для рисования квадратов и кругов. Чтобы получить эти фигуры, используйте одно и то же значение для параметров `width` (ширина) и `height` (высота) в функциях `ellipse()` и `rect()`.

Пример 3.7. Рисование сегмента

Функция `arc()` рисует *сегмент* (часть круга):



```
size(480, 120);
arc(90, 60, 80, 80, 0, HALF_PI);
arc(190, 60, 80, 80, 0, PI+HALF_PI);
arc(290, 60, 80, 80, PI, TWO_PI+HALF_PI);
arc(390, 60, 80, 80, QUARTER_PI, PI+QUARTER_PI);
```

Первый и второй параметры задают расположение центра, третий и четвертый – ширину и высоту. Пятый параметр определяет угол начала дуги, а шестой – угол окончания. Углы задаются в радианах, а не в градусах. *Радиан* – это единица измерения угла, основанная на значении числа π (3,14159). На рис. 3.2 показано, как соотносятся градусы и радианы. Как вы можете видеть в этом примере, четыре значения в радианах используются так часто, что для них в Processing добавили специальные имена. Значения `PI`, `QUARTER_PI`, `HALF_PI` и `TWO_PI` можно использовать как замену значений в радианах для углов величиной 180°, 45°, 90° и 360° соответственно.

Пример 3.8. Использование градусов

Если вы предпочитаете использовать значения в градусах, вы можете преобразовать их в радианы с помощью функции `radians()`. Эта функция получает угол в градусах и возвращает соответствующее значение в радианах. Следующий пример аналогичен примеру 3.7, но в нем используется функция `radians()` для определения начального и конечного значений в градусах:

```
size(480, 120);
arc(90, 60, 80, 80, 0, radians(90));
arc(190, 60, 80, 80, 0, radians(270));
arc(290, 60, 80, 80, radians(180), radians(450));
arc(390, 60, 80, 80, radians(45), radians(225));
```

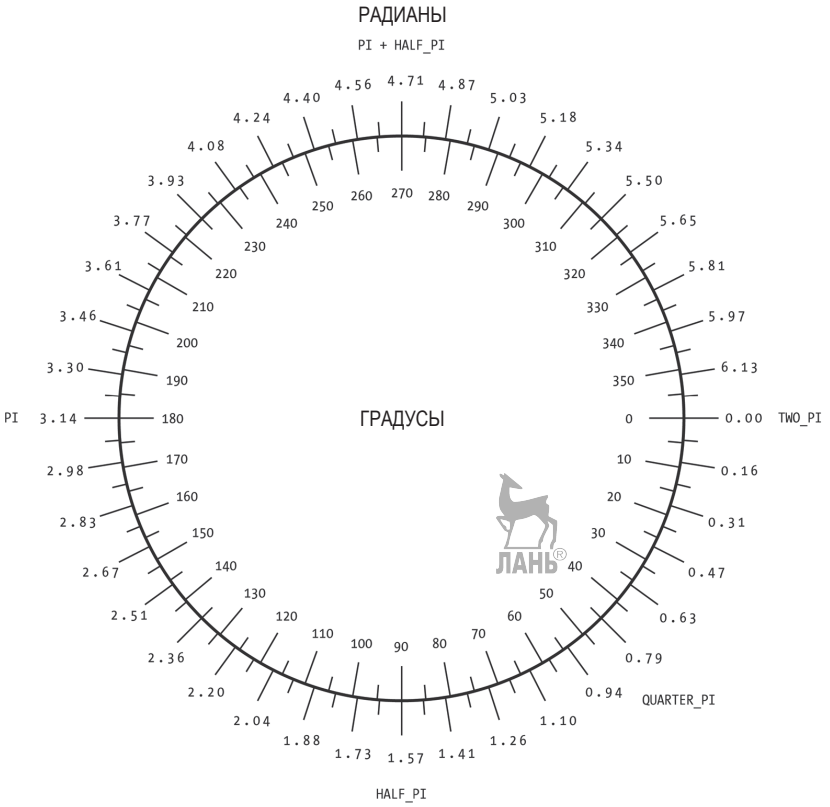
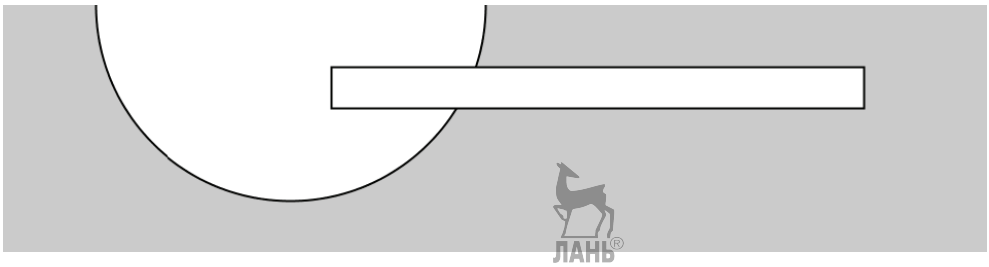


Рис. 3.2 ❖ Радианы и градусы – это два способа обозначить величину угла. При перемещении по кругу значения угла в градусах меняются от 0 до 360, в то время как значения в радианах меняются от 0 до примерно 6,28

3.3. Порядок рисования

Когда вы запускаете программу на языке Processing, компьютер начинает с первой строки и движется сверху вниз, пока не достигает последней строки, а затем останавливается. Если вы хотите, чтобы определенная фигура была нарисована поверх других фигур, функция рисования этой фигуры должна следовать в коде после других функций.

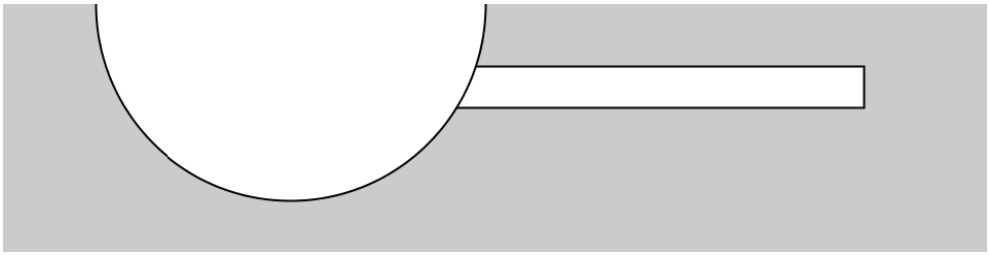
Пример 3.9. Управление порядком рисования



```
size(480, 120);
ellipse(140, 0, 190, 190);
// Прямоугольник будет нарисован поверх круга,
// потому что он размещен вторым в коде
rect(160, 30, 260, 20);
```

Пример 3.10. Обратный порядок рисования

Измените порядок следования функций `rect()` и `ellipse()`, чтобы круг оказался поверх прямоугольника:



```
size(480, 120);
rect(160, 30, 260, 20);
// Круг будет нарисован поверх прямоугольника,
// потому что он размещен вторым в коде
ellipse(140, 0, 190, 190);
```

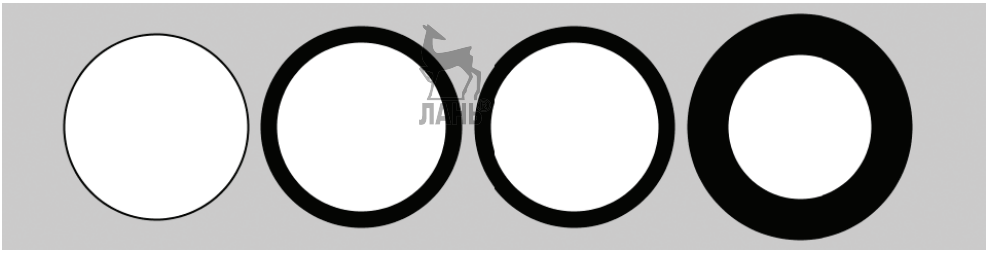
Вы можете представлять, будто наносите мазки кистью или делаете коллаж. Последний элемент, который вы добавляете, отображается сверху.

3.4. Свойства фигуры

Самыми основными и полезными свойствами фигуры являются толщина обводки, способ рисования концов линий (законцовка) и отображение углов фигур.

Пример 3.11. Толщина обводки

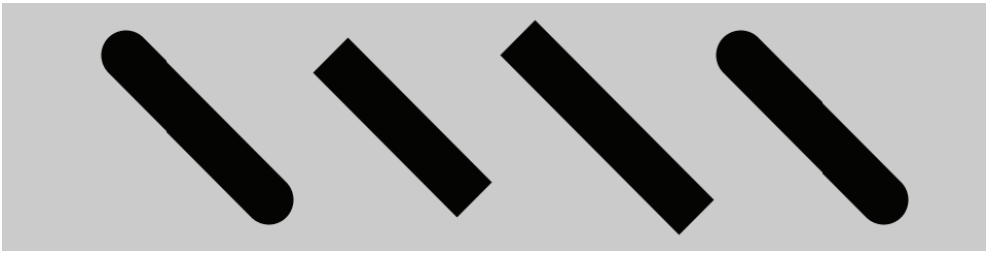
Толщина обводки по умолчанию составляет один пиксель, но этот параметр можно изменить с помощью функции `strokeWeight()`. Единственный параметр `strokeWeight()` устанавливает толщину нарисованных линий:



```
size(480, 120);
ellipse(75, 60, 90, 90);
strokeWeight(8); // Толщина обводки 8 пикселей
ellipse(175, 60, 90, 90);
ellipse(279, 60, 90, 90);
strokeWeight(20); // Толщина обводки 20 пикселей
ellipse(389, 60, 90, 90);
```

Пример 3.12. Разные законцовки линий

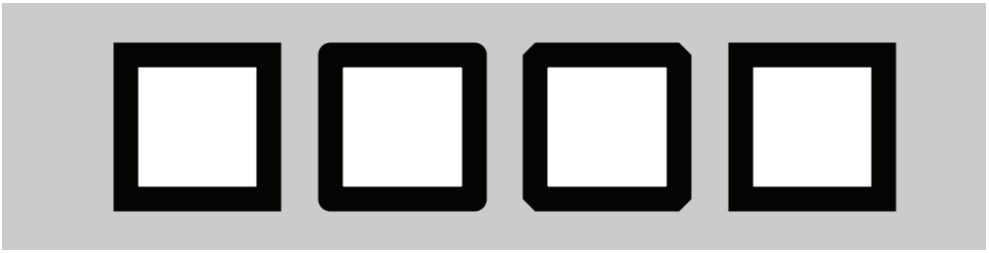
Функция `strokeCap()` изменяет способ рисования линий в их конечных точках. По умолчанию у них закругленные концы:



```
size(480, 120);
strokeWeight(24);
line(60, 25, 130, 95);
strokeCap(SQUARE); // Квадратные законцовки
line(160, 25, 230, 95);
strokeCap(PROJECT); // Удлиненные законцовки
line(260, 25, 330, 95);
strokeCap(ROUND); // Скругленные законцовки
line(360, 25, 430, 95);
```

Пример 3.13. Разные соединения линий

Функция `strokeJoin()` изменяет способ соединения линий (внешний вид углов). По умолчанию они имеют заостренные (резкие) углы:



```
size(480, 120);
strokeWeight(12);
rect(60, 25, 70, 70);
strokeJoin(ROUND); // Скругленные углы
rect(160, 25, 70, 70);
strokeJoin(BEVEL); // Скошенные углы
rect(260, 25, 70, 70);
strokeJoin(MITER); // Резкие углы
rect(360, 25, 70, 70);
```

Когда вы задали какой-то из перечисленных выше параметров, он будет влиять на все фигуры, нарисованные после этого. Например, в примере 3.11 обратите внимание на то, что второй и третий круги имеют одинаковую толщину обводки даже несмотря на то, что параметр толщины установлен только один раз, прежде чем нарисовать два круга.

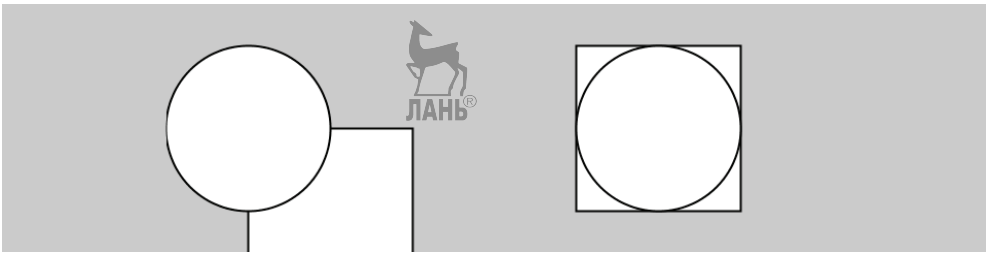
3.5. Режимы рисования

Группа функций, в названии которых есть слово `Mode` (режим), изменяет способ отображения геометрических фигур в рабочем окне Processing. В этой главе мы рассмотрим функции `ellipseMode()` и `rectMode()`, которые помогают нам рисовать эллипсы и прямоугольники соответственно; позже в книге мы рассмотрим функции `imageMode()` и `shapeMode()`.

Пример 3.14. Отсчет координат от угла

По умолчанию функция `ellipse()` использует свои первые два параметра как координаты `x` и `y` геометрического центра фигуры, а третий и четвертый параметры как ширину и высоту. После выполнения команды `ellipseMode(CORNER)` первые два параметра `ellipse()` определяют положение верхнего левого угла прямоугольника, в который вписан эллипс. Это делает поведе-

ние функции `ellipse()` похожим на поведение функции `rect()`, как показано в данном примере:



```
size(480, 120);
rect(120, 60, 80, 80);
ellipse(120, 60, 80, 80);
ellipseMode(CORNER);
rect(280, 20, 80, 80);
ellipse(280, 20, 80, 80);
```

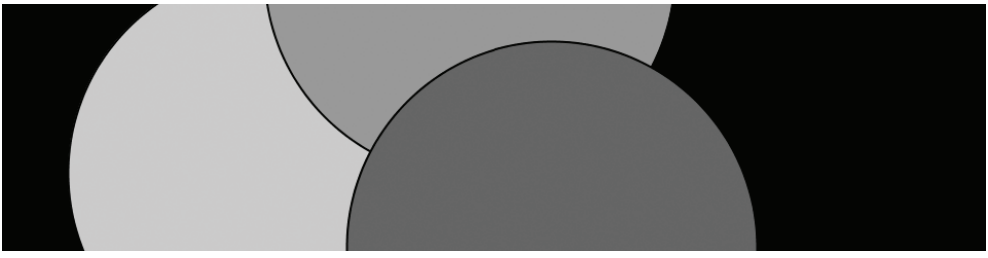
Вы встретите функции с расширением `Mode` в примерах по всей книге. В справочной документации Processing показаны и другие варианты их использования.

3.6. Использование цвета

Все фигуры до сих пор были закрашены белым цветом с черными контурами, а фон рабочего окна был светло-серым. Чтобы изменить эти параметры, используйте функции фона `background()`, заливки `fill()` и обводки `stroke()`. Значения параметров находятся в диапазоне от `0` до `255`, где `255` – белый, `128` – средний серый, а `0` – черный.

Пример 3.15. Рисование в градациях серого

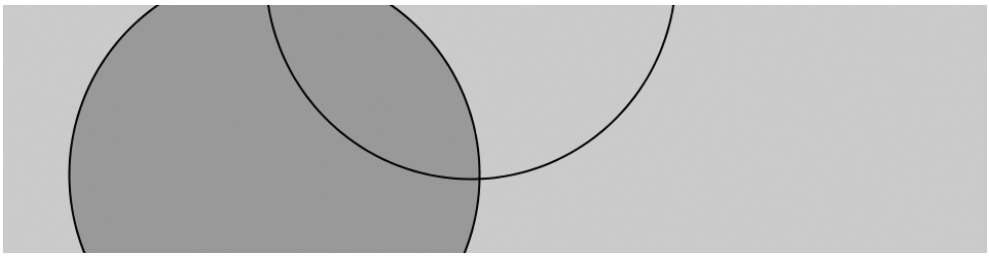
В этом примере показаны три разные градации (уровня) серого на черном фоне:



```
size(480, 120);
background(0);               // Черный
fill(204);                   // Светло-серый
ellipse(132, 82, 200, 200);  // Светло-серый круг
fill(153);                   // Средне-серый
ellipse(228, -16, 200, 200); // Средне-серый круг
fill(102);                   // Темно-серый
ellipse(268, 118, 200, 200); // Темно-серый круг
```

Пример 3.16. Управление заливкой и обводкой

Вы можете отключить обводку, чтобы не было контура, используя функцию `noStroke()`, а также можете отключить заливку фигуры с помощью функции `noFill()`:

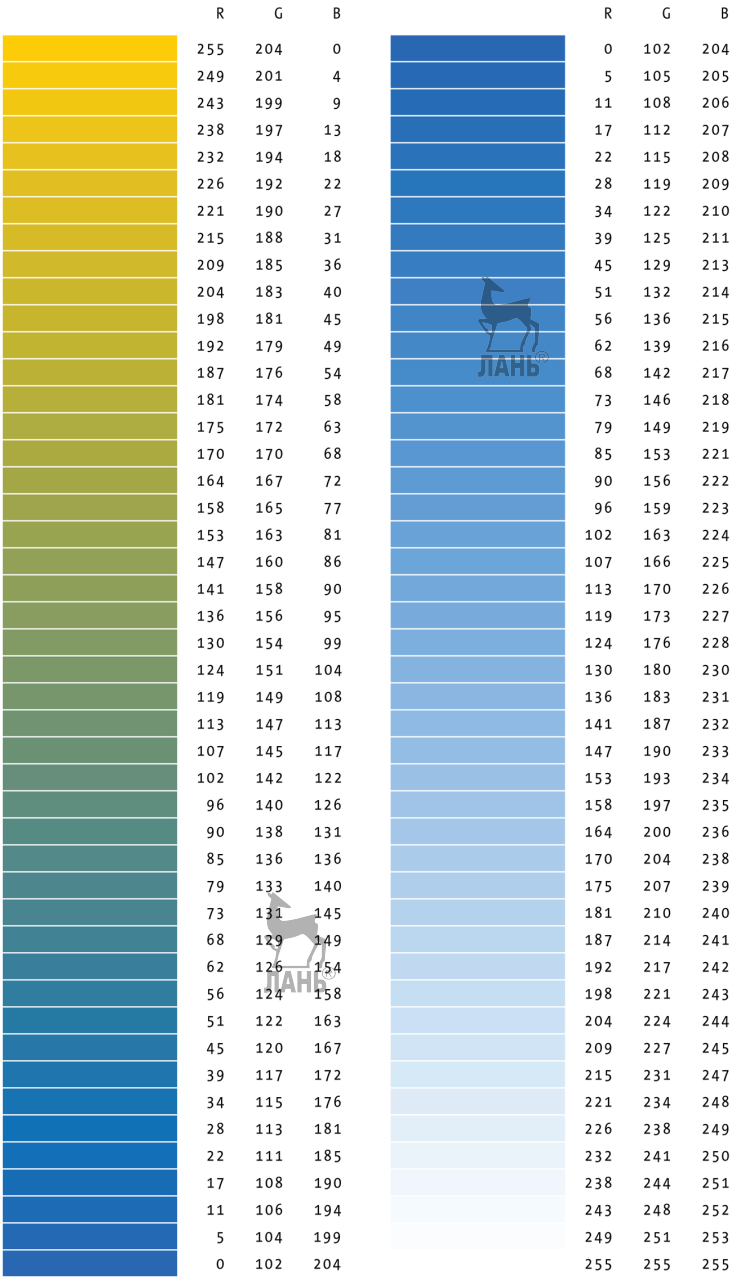


```
size(480, 120);
fill(153);                   // Среднесерый
ellipse(132, 82, 200, 200);  // Серый круг
noFill();                    // Отключение заливки
ellipse(228, -16, 200, 200); // Контур окружности
noStroke();                  // Отключение обводки
ellipse(268, 118, 200, 200); // Ничего не видно!
```

Будьте осторожны, чтобы не отключить заливку и обводку одновременно, как мы это делали в предыдущем примере, потому что тогда фигура не отобразится в рабочем окне.

Пример 3.17. Рисование в цвете

Чтобы выйти за рамки значений оттенков серого, вы должны использовать три параметра – уровни красного, синего и зеленого компонентов цвета. На рис. 3.3 показано соответствие между значениями от `0` до `255` и разными уровнями интенсивности цветовых компонентов.

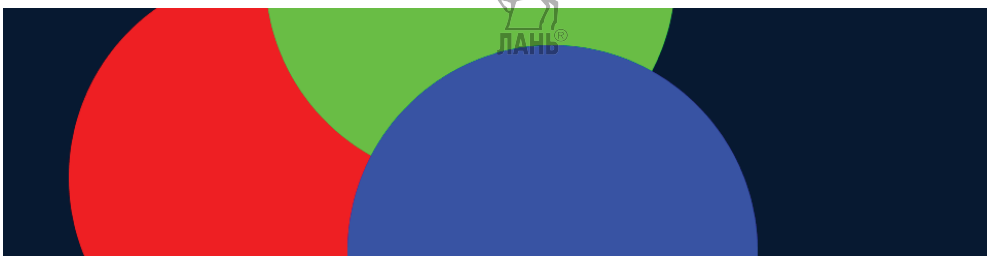


R	G	B
255	204	0
249	201	4
243	199	9
238	197	13
232	194	18
226	192	22
221	190	27
215	188	31
209	185	36
204	183	40
198	181	45
192	179	49
187	176	54
181	174	58
175	172	63
170	170	68
164	167	72
158	165	77
153	163	81
147	160	86
141	158	90
136	156	95
130	154	99
124	151	104
119	149	108
113	147	113
107	145	117
102	142	122
96	140	126
90	138	131
85	136	136
79	133	140
73	131	145
68	129	149
62	126	154
56	124	158
51	122	163
45	120	167
39	117	172
34	115	176
28	113	181
22	111	185
17	108	190
11	106	194
5	104	199
0	102	204

R	G	B
0	102	204
5	105	205
11	108	206
17	112	207
22	115	208
28	119	209
34	122	210
39	125	211
45	129	213
51	132	214
56	136	215
62	139	216
68	142	217
73	146	218
79	149	219
85	153	221
90	156	222
96	159	223
102	163	224
107	166	225
113	170	226
119	173	227
124	176	228
130	180	230
136	183	231
141	187	232
147	190	233
153	193	234
158	197	235
164	200	236
170	204	238
175	207	239
181	210	240
187	214	241
192	217	242
198	221	243
204	224	244
209	227	245
215	231	247
221	234	248
226	238	249
232	241	250
238	244	251
243	248	252
249	251	253
255	255	255

Рис. 3.3 ❖ Цвета создаются путем определения значений RGB (красный, зеленый, синий)

Запустите следующий код, демонстрирующий использование цвета:



```
size(480, 120);
noStroke();
background(0, 26, 51);         // Темно-синий фон
fill(255, 0, 0);               // Красный цвет
ellipse(132, 82, 200, 200);    // Красный круг
fill(0, 255, 0);               // Зеленый цвет
ellipse(228, -16, 200, 200);   // Зеленый круг
fill(0, 0, 255);               // Синий цвет
ellipse(268, 118, 200, 200);   // Синий круг
```

Это так называемая *цветовая схема RGB*, которая описывает, как компьютер определяет цвета на экране. Три числа обозначают значения красного, зеленого и синего компонентов, и они находятся в диапазоне от `0` до `255`, как и значения серого. Использовать числа для выбора промежуточного цвета RGB довольно трудно, поэтому для выбора цвета используйте опцию меню **Инструменты** | **Выбрать цвет**, которая показывает цветовую палитру, аналогичную палитре в других программах (рис. 3.4). Выберите цвет, а затем используйте значения R, G и B в качестве параметров для функции `background()`, `fill()` или `stroke()`.

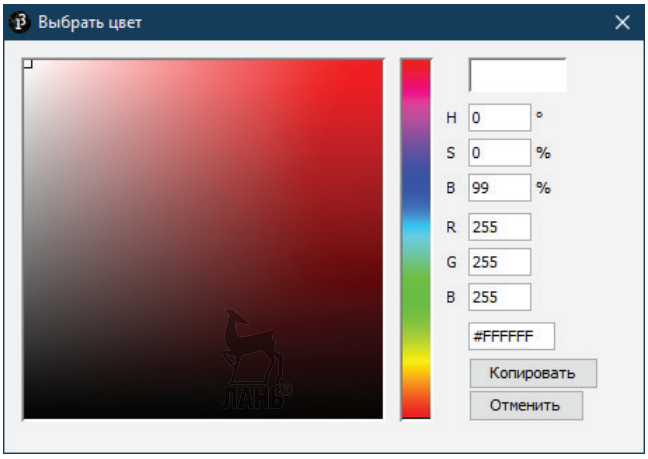


Рис. 3.4 ❖ Палитра выбора цвета в Processing

Пример 3.18. Использование прозрачности

Добавив необязательный четвертый параметр в `fill()` или `stroke()`, вы можете управлять *прозрачностью* графического объекта. Этот четвертый параметр известен как *альфа-канал*, и в нем также применяются значения в диапазоне от `0` до `255` для установки степени прозрачности. Значение `0` определяет цвет как полностью прозрачный (он не отображается), значение `255` означает полностью непрозрачный цвет, а значения между этими крайними параметрами вызывают смешивание цветов на экране:

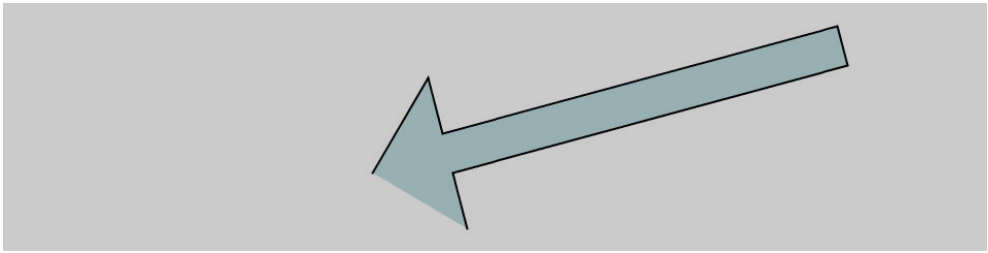

```
size(480, 120);
noStroke();
background(204, 226, 225);   // Светло-голубой фон
fill(255, 0, 0, 160);        // Красный цвет
ellipse(132, 82, 200, 200);  // Красный круг
fill(0, 255, 0, 160);        // Зеленый цвет
ellipse(228, -16, 200, 200); // Зеленый круг
fill(0, 0, 255, 160);        // Синий цвет
ellipse(268, 118, 200, 200); // Синий круг
```

3.7. Пользовательские фигуры

Ваш выбор не ограничен использованием основных геометрических фигур – вы также можете определять новые фигуры, соединяя последовательность точек.

Пример 3.19. Рисование стрелки

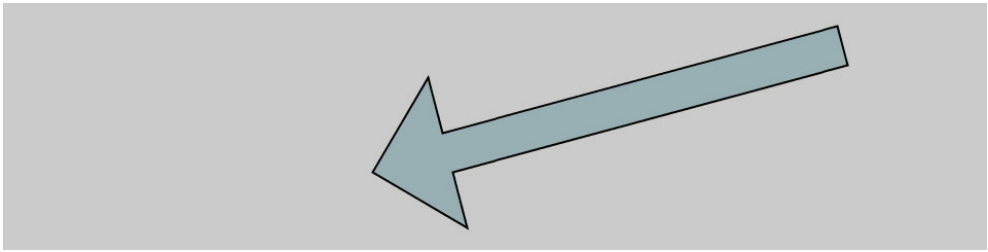
Функция `beginShape()` обозначает начало описания новой фигуры. Функция `vertex()` применяется для определения пары координат `x` и `y` каждой угловой точки фигуры. Наконец, функция `endShape()` сообщает, что описание фигуры завершено:



```
size(480, 120);
beginShape();
fill(153, 176, 180);
vertex(180, 82);
vertex(207, 36);
vertex(214, 63);
vertex(407, 11);
vertex(412, 30);
vertex(219, 82);
vertex(226, 109);
endShape();
```


Пример 3.20. Устранение разрыва

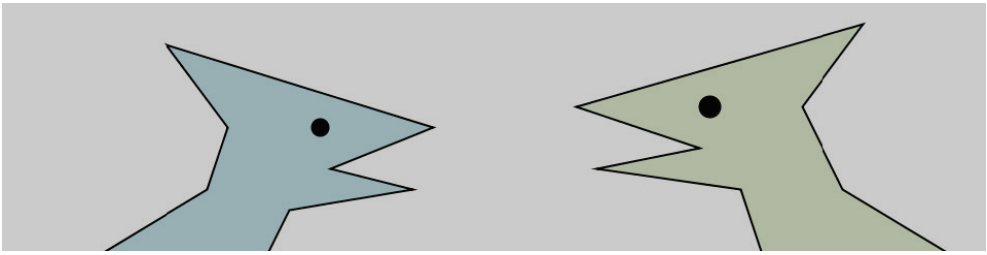
Когда вы запустите скетч из примера 3.19, то увидите, что первая и последняя точки не связаны. Для этого добавьте слово `CLOSE` в качестве параметра в функцию `endShape()`, например:



```
size(480, 120);
beginShape();
fill(153, 176, 180);
vertex(180, 82);
vertex(207, 36);
vertex(214, 63);
vertex(407, 11);
vertex(412, 30);
vertex(219, 82);
vertex(226, 109);
endShape(CLOSE);
```

Пример 3.21. Небольшое развлечение

Мощь определения фигур с помощью `vertex()` заключается в возможности создавать формы со сложными контурами. Processing может рисовать тысячи и тысячи линий одновременно, чтобы заполнить экран фантастическими существами, возникающими в вашем воображении. Ниже приводится скромный, но более сложный, чем раньше, пример:



```
size(480, 120);

// Левое существо
fill(153, 176, 180);
beginShape();
vertex(50, 120);
vertex(100, 90);
vertex(110, 60);
vertex(80, 20);
vertex(210, 60);
vertex(160, 80);
vertex(200, 90);
vertex(140, 100);
vertex(130, 120);
endShape();
fill(0);
ellipse(155, 60, 8, 8);

// Правое существо
fill(176, 186, 163);
beginShape();
vertex(370, 120);
vertex(360, 90);
vertex(290, 80);
vertex(340, 70);
vertex(280, 50);
vertex(420, 10);
vertex(390, 50);
vertex(410, 90);
vertex(460, 120);
endShape();
fill(0);
ellipse(345, 50, 10, 10);
```

3.8. Комментарии

В примерах в этой главе двойная косая черта (//) в конце строки используется для добавления комментариев к коду. *Комментарии* – это части кода, которые игнорируются при запуске программы. Они полезны для того, чтобы делать себе заметки, объясняющие, что происходит в коде. Если предполагается, что другие люди будут читать ваш код, комментарии особенно важны, чтобы помочь им понять ход ваших мыслей.

Комментарии также полезны для сохранения в коде различных вариантов одной функции, например когда вы пытаетесь выбрать правильный цвет. Допустим, я пытаюсь найти подходящий оттенок красного цвета для эллипса:

```
size(200, 200);
fill(165, 57, 57);
ellipse(100, 100, 80, 80);
```

А теперь предположим, что я хочу попробовать другой оттенок красного, но не хочу терять старый. Я могу скопировать и вставить строку, внести в нее изменения, а затем «закомментировать» старую строку:

```
size(200, 200);
//fill(165, 57, 57);
fill(144, 39, 39);
ellipse(100, 100, 80, 80);
```

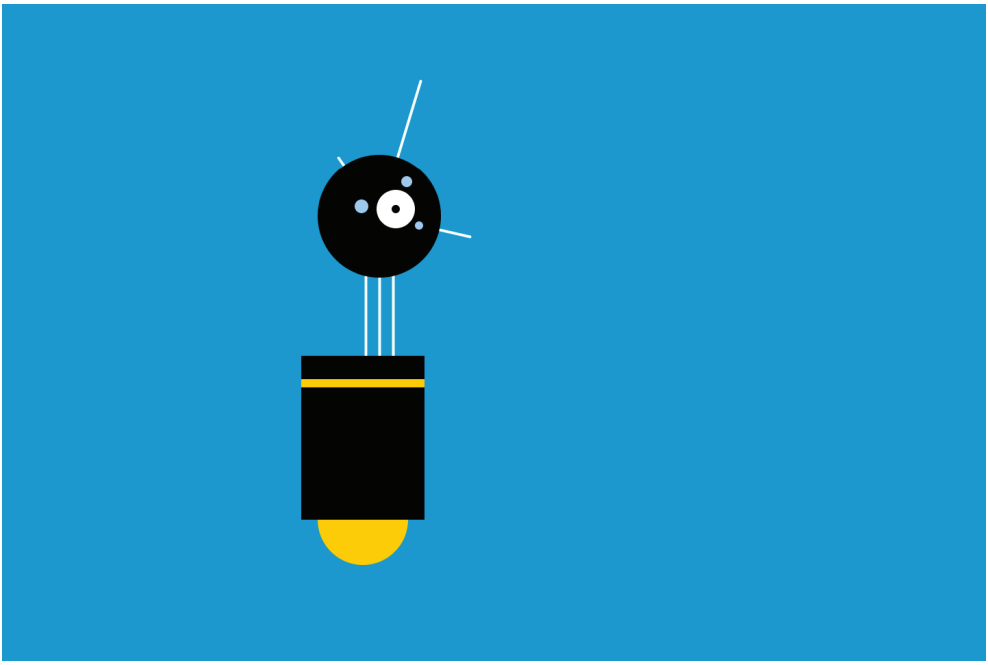
Размещение символов // в начале строки временно отключает ее. Потом я могу удалить символы // и поместить их перед другой строкой, если захочу вернуться к первому оттенку:

```
size(200, 200);
fill(165, 57, 57);
//fill(144, 39, 39);
ellipse(100, 100, 80, 80);
```

Работая со скетчами на языке Processing, вы будете создавать десятки вариантов разных идей; использование комментариев для заметок или отключения строк поможет вам сохранить все эти варианты в одном файле кода.

Для быстрого добавления или удаления комментария вы можете использовать комбинацию клавиш **Ctrl+/** (**Cmd+/** на Mac). Вы также можете закомментировать несколько строк одновременно, используя альтернативную нотацию комментариев, представленную в разделе «Комментарии» примечания 1.

3.9. Робот 1: рисование



Это P5, робот, обитающий в мире языка Processing. В книге вы найдете 10 различных программ, способных нарисовать и оживить его; каждая программа иллюстрирует разные приемы программирования. Дизайн P5 был создан по мотивам советского Спутника I (1957 г.), робота Шеки из Стэнфордского исследовательского института (1966–1972 гг.), дрона-истребителя из «Дюны» Дэвида Линча (1984 г.) и HAL 9000 из «2001: Космическая одиссея» (1968 г.), а также многих других популярных роботов.

Первая программа робота использует функции рисования, представленные в этой главе. Параметры функций `fill()` и `stroke()` устанавливают значения серого. Функции `line()`, `ellipse()` и `rect()` определяют фигуры, составляющие шею, антенны, тело и голову робота. Чтобы лучше ознакомиться с функциями, запустите программу и измените их параметры, чтобы изменить конструкцию робота:

```
size(720, 480);
strokeWeight(2);
background(0, 153, 204);     // Голубой фон
ellipseMode(RADIUS);

// Шея
stroke(255);                 // Белый цвет линий
line(266, 257, 266, 162);    // Левая
line(276, 257, 276, 162);    // Средняя
line(286, 257, 286, 162);    // Правая
```

```
// Антенны
line(276, 155, 246, 112);   // Короткая
line(276, 155, 306, 56);    // Длинная
line(276, 155, 342, 170);   // Средняя

// Туловище
noStroke();                 // Отключение обводки
fill(255, 204, 0);          // Оранжевая заливка
ellipse(264, 377, 33, 33);  // Антигравитационная опора
fill(0);                    // Черная заливка
rect(219, 257, 90, 120);    // Корпус
fill(255, 204, 0);          // Желтая заливка
rect(219, 274, 90, 6);      // Желтая полоска

// Голова
fill(0);                    // Черная заливка
ellipse(276, 155, 45, 45);  // Голова
fill(255);                  // Белая заливка
ellipse(288, 150, 14, 14);  // Большой глаз
fill(0);                    // Черная заливка
ellipse(288, 150, 3, 3);    // Зрачок
fill(153, 204, 255);        // Светло-голубая заливка
ellipse(263, 148, 5, 5);    // Маленький глаз 1
ellipse(296, 130, 4, 4);    // Маленький глаз 2
ellipse(305, 162, 3, 3);    // Маленький глаз 3
```

Глава 4

Переменные

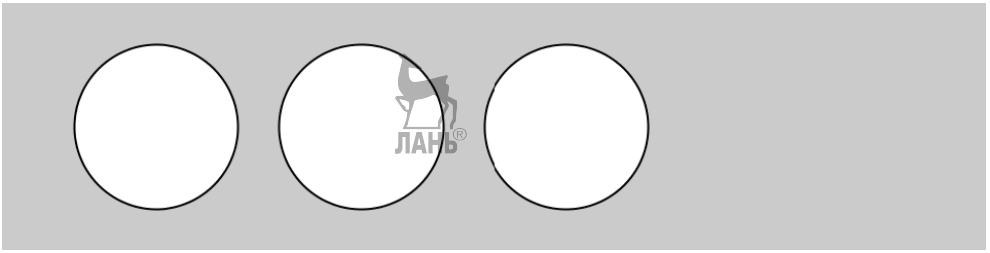
Переменная хранит значение в памяти, чтобы его можно было использовать позже в программе. Во время работы программы переменной можно воспользоваться несколько раз, а также легко изменить ее значение.

4.1. Первые переменные

Одна из причин, по которой мы используем переменные, заключается в том, чтобы не повторять одни и те же данные в коде. Если вы вводите одно и то же число более одного раза, возможно, лучше использовать вместо него переменную, чтобы ваш код был более обобщенным и его было легче обновлять.

Пример 4.1. Повторное использование одних и тех же значений

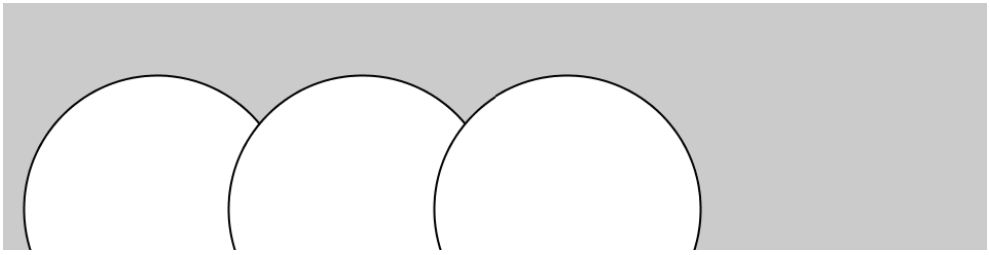
В этом примере координата y и диаметр одинаковые для трех эллипсов, поэтому в качестве соответствующих аргументов функций используются переменные:



```
size(480, 120);
int y = 60;
int d = 80;
ellipse(75, y, d, d);  // Слева
ellipse(175, y, d, d); // Середина
ellipse(275, y, d, d); // Справа
```

Пример 4.2. Изменение значений

Простое изменение переменных y и d изменяет все три эллипса:



```
size(480, 120);
int y = 100;
int d = 130;
ellipse(75, y, d, d); // Left
ellipse(175, y, d, d); // Middle
ellipse(275, y, d, d); // Right
```

Если бы не было переменных, вам пришлось бы изменить значение координаты y, используемое в коде, три раза, а значение диаметра – шесть раз. Сравнивая пример 4.1 и пример 4.2, обратите внимание на то, что три нижние строки совпадают и только две средние строки с переменными отличаются. Переменные позволяют отделить строки кода, которые изменяются, от строк, которые не изменяются, что упрощает изменение программ. Например, если вы поместите переменные, которые управляют цветами и размерами фигур, в одном месте, то сможете быстро испробовать различные визуальные параметры, сосредоточившись всего на нескольких строках кода.

4.2. Создание переменных

Создавая свои собственные переменные, вы определяете имя, тип данных и значение. *Имя* – это то, как вы решили назвать переменную. Выберите имя, которое говорит о том, что хранит переменная, но при этом логичное и не слишком длинное. Например, имя переменной «radius» будет понятнее, чем «r», когда вы позже вернетесь к своему коду.

Диапазон значений, которые могут быть сохранены в переменной, определяется ее *типом данных*. Например, целочисленный тип данных может хранить числа без десятичных разрядов (целые числа). В коде целое число (integer) сокращается до `int`. Существует много разных типов данных: целые числа, числа с плавающей запятой (десятичные), символы, слова, изображения, шрифты и т. д.

Прежде всего переменную нужно *объявить*, чтобы выделить место в памяти компьютера для хранения информации. При объявлении переменной вам

также необходимо указать ее тип данных (например, int), который говорит, какая информация сохраняется. После того как тип данных и имя объявлены, переменной можно присвоить значение:

```
int x;  // Объявляем переменную x типа int
x = 12; // Присваиваем значение переменной x
```

Следующий код делает то же самое, но короче:

```
int x = 12; // Объявляем x как переменную типа int и присваиваем значение
```

Наименование типа данных включается в строку кода, в которой объявляется переменная, и больше в коде не упоминается. Каждый раз, когда перед именем переменной указан тип данных, компьютер думает, что вы пытаетесь объявить новую переменную. У вас не может быть двух переменных с одинаковыми именами в одной и той же части программы (приложение 4), поэтому программа содержит ошибку:

```
int x;      // Объявляем x как переменную типа int
int x = 12; // ОШИБКА! Не может быть двух переменных с именем x
```

Сохраняемые данные должны строго соответствовать заявленному типу переменной. Например, переменная типа int может хранить целое число, но не может хранить число с десятичной запятой, называемое в коде float (число с плавающей запятой). Термин «с плавающей запятой» относится к методу, который применяется для хранения числа с десятичной запятой в памяти компьютера. (Сейчас эти детали не имеют значения.)

Число с плавающей запятой нельзя присвоить типу int, поскольку информация будет потеряна. Например, значение 12.2 отличается от ближайшего к нему эквивалента типа int, значения 12. При выполнении программы эта операция вызовет ошибку:

```
int x = 12.2; // ОШИБКА! Значение с плавающей запятой не может поместиться в int
```

Однако переменная с плавающей запятой может хранить целое число. Например, целочисленное значение 12 может быть преобразовано в эквивалент 12.0 с плавающей запятой, поскольку информация не теряется. Следующий код сработает без ошибок:

```
float x = 12; // Автоматическая конвертация 12 в 12.0
```

Типы данных обсуждаются более подробно в приложении 2.

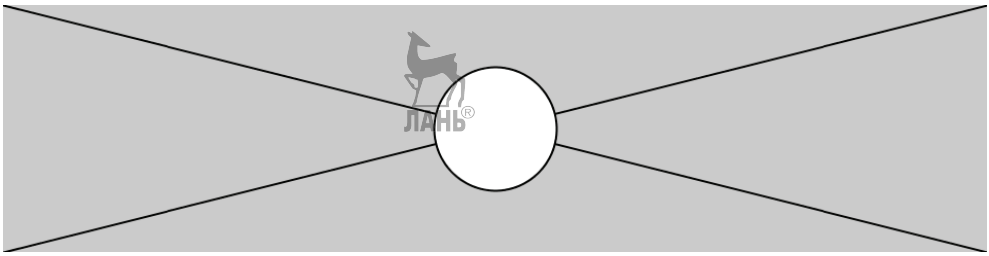
4.3. Специальные переменные Processing

Processing имеет ряд *специальных переменных* для хранения информации о программе во время ее выполнения. Например, ширина и высота окна хранятся в переменных с именами width и height. Эти значения устанавливаются функцией size(). Их можно использовать для корректного рисования

элементов относительно границ окна, даже если впоследствии изменить строку `size()`.

Пример 4.3. Рисование фигур относительно границ окна

В этом примере измените параметры функции `size()`, чтобы увидеть, как работает рисование фигур относительно границ окна:



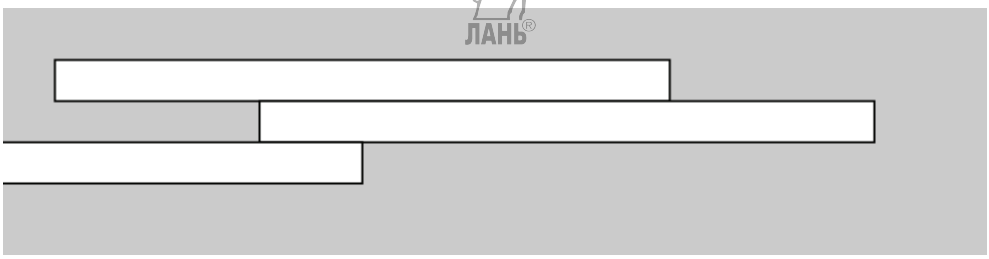
```
size(480, 120);
line(0, 0, width, height); // Линия между точками (0,0) и (480, 120)
line(width, 0, 0, height); // Линия между точками (480, 0) и (0, 120)
ellipse(width/2, height/2, 60, 60);
```

Другие специальные переменные отслеживают состояние значений мыши и клавиатуры и многое другое. Они обсуждаются в главе 5.

4.4. Немного математики

Люди часто думают, что математика и программирование – одно и то же. Хотя для разработки некоторых программ действительно требуется хорошее знание математики, в большинстве случаев программирование ограничивается основными арифметическими действиями.

Пример 4.4. Основные арифметические операции



```
size(480, 120);
int x = 25;
int h = 20;
int y = 25;
rect(x, y, 300, h);       // Верхний
x = x + 100;
rect(x, y + h, 300, h);   // Средний
x = x - 250;
rect(x, y + h*2, 300, h); // Нижний
```

В коде такие символы, как +, - и *, называются *операторами*. Оказавшись между двумя значениями, они создают *выражение*. Например, `5+9` и `1024-512` – это выражения. Операторы для основных математических операций перечислены в таблице ниже:

+	Сложение
–	Вычитание
*	Умножение
/	Деление
=	Присваивание

Processing имеет набор правил, определяющих приоритет одних операций над другими, то есть какие вычисления выполняются первыми, вторыми, третьими и т. д. Эти правила определяют порядок выполнения кода. Знания об этом имеют большое значение для понимания того, как работает такая короткая строка кода:

```
int x = 4 + 4 * 5; // Присваиваем значение 24 переменной x
```

Выражение `4*5` вычисляется первым, потому что умножение имеет наивысший приоритет. Вторым шагом к произведению `4*5` прибавляется `4`, и получается значение `24`. Наконец, поскольку *оператор присваивания* (знак равенства) имеет самый низкий приоритет, значение 24 присваивается переменной `x`. В это выражение можно добавить скобки, но результат не изменится:

```
int x = 4 + (4 * 5); // Присваиваем значение 24 переменной x
```

Если вы хотите, чтобы сложение было выполнено первым, просто переместите скобки. Поскольку любой оператор в скобках имеет более высокий приоритет, чем умножение, порядок вычислений изменится, и это повлияет на результат:

```
int x = (4 + 4) * 5; // Присваиваем значение 40 переменной x
```

Студенты в англоязычных странах обычно заучивают аббревиатуру PEMDAS от слов Parentheses, Exponents, Multiplication, Division, Addition, Subtraction, что означает круглые скобки, возведение в степень, умножение, деление, сложение, вычитание, где круглые скобки имеют наивысший приоритет, а вычитание – самый низкий. Полный перечень приоритетов операций приведен в приложении 3.

Некоторые вычисления так часто используются в программировании, что для них были разработаны специальные обозначения; всегда приятно сэкономить несколько нажатий клавиш. Например, вы можете прибавить к переменной или вычесть из нее значение с помощью одного оператора:

```
x + = 10; // То же самое, что x = x + 10
y - = 15; // То же самое, что y = y - 15
```

Также очень часто приходится прибавлять или вычитать 1 из переменной, поэтому для таких операций тоже придумали специальные операторы ++ и --:

```
x ++; // То же самое, что x = x + 1
y--; // То же самое, что y = y - 1
```

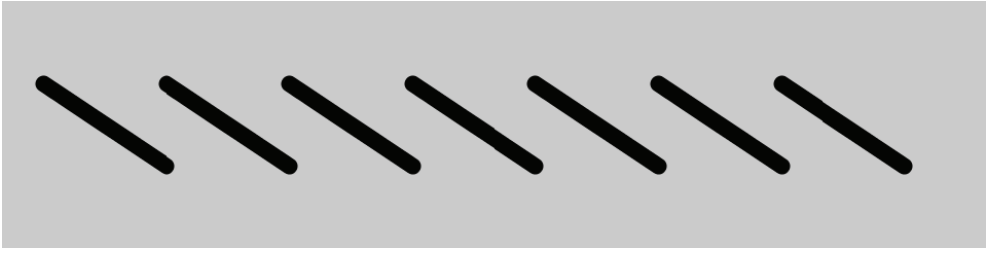
Более подробный список обозначений можно найти в справочнике по языку Processing.

4.5. Повторение

По мере написания новых программ вы заметите, что часто сталкиваетесь с ситуацией, когда строки кода повторяются, но с небольшими вариациями. Структура кода, называемая *циклом* `for`, позволяет выполнить строку кода более одного раза, чтобы обойтись в программе минимальным количеством строк. Это делает ваши программы более модульными, и их легче изменять.

Пример 4.5. Повторение однотипного действия

В этом примере есть шаблонное действие, повторение которого можно упростить с помощью цикла `for`:



```
size(480, 120);
strokeWeight(8);
line(20, 40, 80, 80);
line(80, 40, 140, 80);
line(140, 40, 200, 80);
line(200, 40, 260, 80);
line(260, 40, 320, 80);
```

```
line(320, 40, 380, 80);
line(380, 40, 440, 80);
```

Пример 4.6. Использование цикла for

Тот же результат, что и в предыдущем примере, можно получить с помощью цикла for и с меньшим объемом кода:

```
size(480, 120);
strokeWeight(8);
for (int i = 20; i < 400; i += 60) {
  line(i, 40, i + 60, 80);
}
```

Цикл for во многом отличается от кода, который мы написали до сих пор. Обратите внимание на фигурные скобки { и }. Код между фигурными скобками называется *блоком*. Это код, который будет повторяться в каждой итерации цикла for.

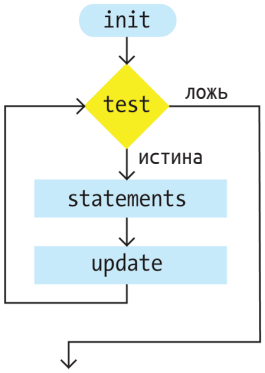
Внутри круглых скобок находятся три оператора, разделенных точкой с запятой. От них зависит, сколько раз будет запускаться код внутри блока. Эти операторы по порядку называются *инициализацией* (init), *проверкой* (test) и *обновлением* (update) цикла:

```
for (init; test; update) {
  операторы
}
```

Оператор инициализации устанавливает начальное значение, обычно объявляя новую переменную для использования в цикле for. В предыдущем примере была объявлена целочисленная переменная с именем i, которой присвоено значение 20. Имя переменной i традиционно используется в циклах, но на самом деле в нем нет ничего особенного. Оператор проверки оценивает значение этой переменной (в нашем примере он проверяет, осталось ли значение i меньше 400), а оператор обновления изменяет значение переменной (прибавляя 60 перед повторением цикла). На рис. 4.1 показан порядок, в котором выполняются операторы управления циклом, и то, как они управляют операторами кода внутри блока.

Оператор проверки требует дополнительных пояснений. Это всегда *выражение отношения*, сравнивающее два значения с помощью *оператора отношения*. В этом примере выражением отношения является i<400, а оператором является символ < (меньше). Наиболее распространенные операторы отношения:

>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
==	Равно
!=	Не равно



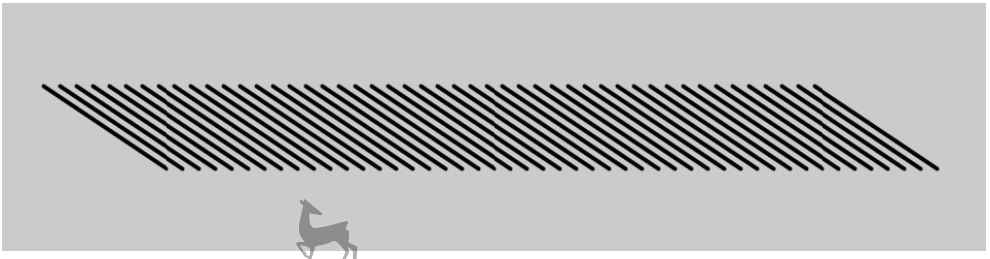

```
for (init; test; update) {
  statements
}
```

Рис. 4.1 ❖ Блок-схема цикла for

Выражение отношения всегда оценивается как *истинное* (`true`) или *ложное* (`false`). Например, выражение `5>3` истинное. Мы можем задать вопрос: «Пять больше трех?» Поскольку ответ «да», мы говорим, что выражение истинное. Выражению `5 < 3` соответствует вопрос: «Пять меньше трех?» Поскольку ответ «нет», мы говорим, что выражение ложное. Когда проверочное выражение истинно, выполняется код внутри блока, а когда ложно, код внутри блока не выполняется, и цикл `for` завершается.

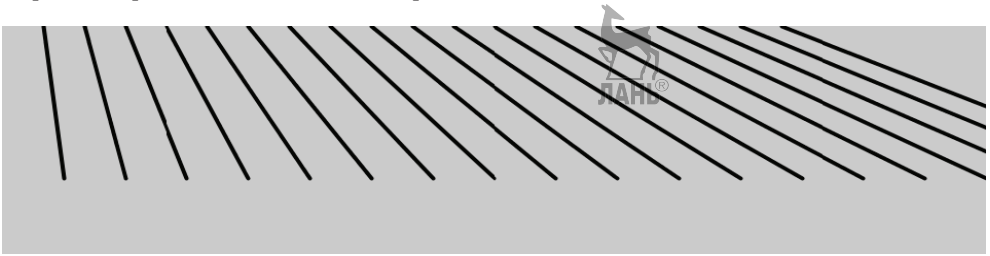
Пример 4.7. Преимущество использования цикла for

Одно из главных преимуществ использования цикла `for` – это возможность быстро вносить изменения в код. Поскольку код внутри блока обычно запускается несколько раз, изменение блока отражается на результате каждого запуска кода внутри цикла. Немного изменив пример 4.6, мы можем создать ряд различных повторяющихся элементов:



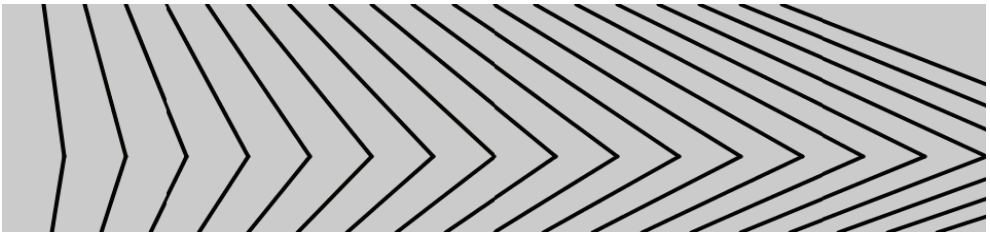
```
size(480, 120);
strokeWeight(2);
for (int i = 20; i < 400; i += 8) {
  line(i, 40, i + 60, 80);
}
```

Пример 4.8. Линии с разным наклоном



```
size(480, 120);
strokeWeight(2);
for (int i = 20; i < 400; i += 20) {
  line(i, 0, i + i/2, 80);
}
```

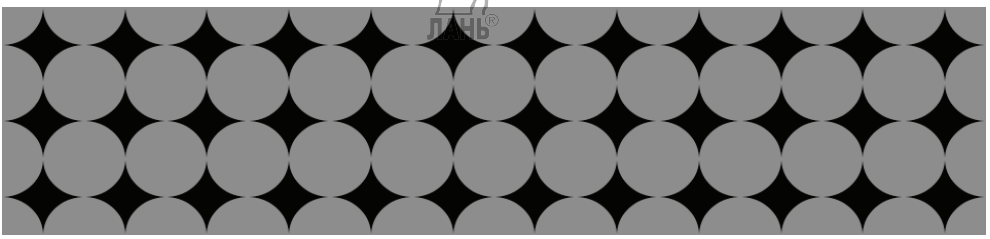
Пример 4.9. Ломаные линии



```
size(480, 120);
strokeWeight(2);
for (int i = 20; i < 400; i += 20) {
  line(i, 0, i + i/2, 80);
  line(i + i/2, 80, i*1.2, 120);
}
```

Пример 4.10. Вложенные циклы

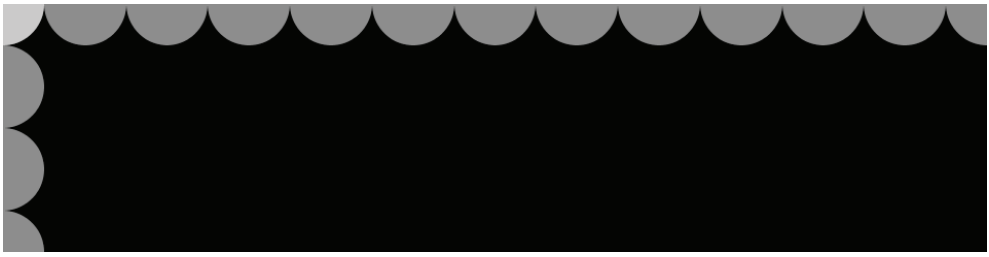
Когда один цикл `for` вложен в другой, количество повторений умножается. Сначала давайте рассмотрим небольшой пример, а затем разберем его код в примере 4.11:



```
size(480, 120);
background(0);
noStroke();
for (int y = 0; y <= height; y += 40) {
  for (int x = 0; x <= width; x += 40) {
    fill(255, 140);
    ellipse(x, y, 40, 40);
  }
}
```


Пример 4.11. Строки и столбцы

В этом примере циклы for являются смежными, а не вложенными один в другой. По результату видно, что один цикл for рисует столбец из 4 кругов, а другой рисует строку из 13 кругов:

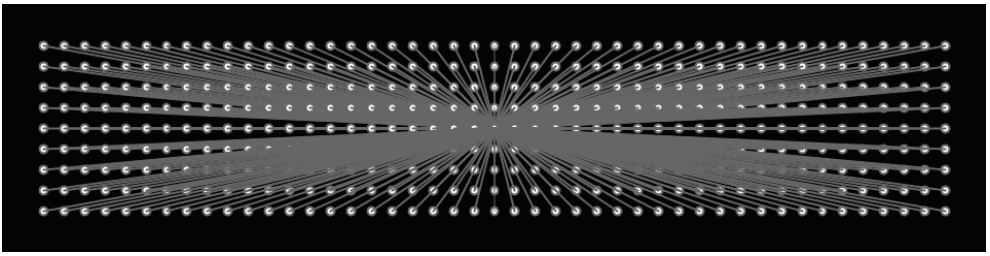


```
size(480, 120);
background(0);
noStroke();
for (int y = 0; y < height+45; y += 40) {
  fill(255, 140);
  ellipse(0, y, 40, 40);
}
for (int x = 0; x < width+45; x += 40) {
  fill(255, 140);
  ellipse(x, 0, 40, 40);
}
```

Когда один из этих циклов for помещается внутрь другого, как в примере 4.10, 4 повторения первого цикла объединяются с 13 повторениями второго, чтобы выполнить код внутри вложенного блока 52 раза (4 × 13 = 52).

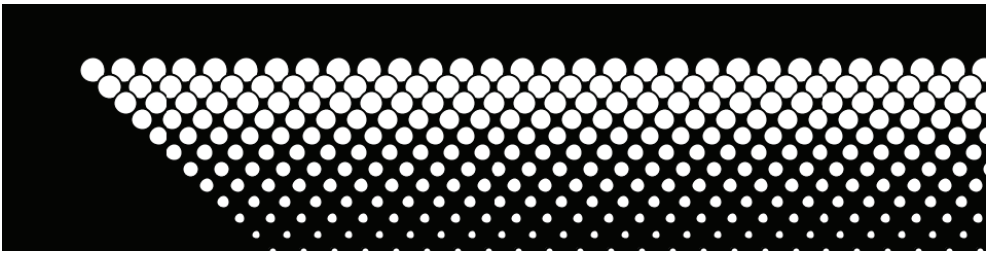
Пример 4.10 – хорошая основа для изучения многих типов повторяющихся визуальных шаблонов. В следующих примерах показано несколько способов усложнения рисунка, но это лишь малая часть того, что можно сделать. В примере 4.12 код рисует линию от каждой точки сетки к центру экрана. В примере 4.13 эллипсы сжимаются с каждой новой строкой и перемещаются вправо путем прибавления координаты y к координате x.

Пример 4.12. Точки и линии



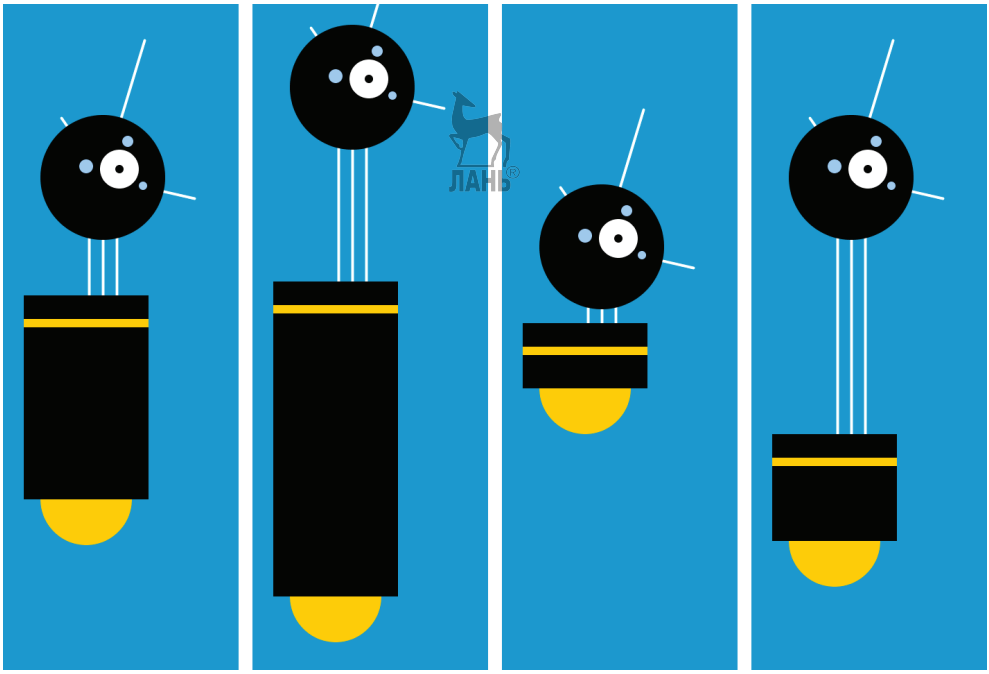
```
size(480, 120);
background(0);
fill(255);
stroke(102);
for (int y = 20; y <= height-20; y += 10) {
  for (int x = 20; x <= width-20; x += 10) {
    ellipse(x, y, 4, 4);
    // Рисование линии к центру рабочего окна
    line(x, y, 240, 60);
  }
}
```


Пример 4.13. Полутоновые точки



```
size(480, 120);
background(0);
for (int y = 32; y <= height; y += 8) {
  for (int x = 12; x <= width; x += 15) {
    ellipse(x + y, y, 16 - y/10.0, 16 - y/10.0);
  }
}
```

4.6. Робот 2: переменные



Использование переменных делает этот код сложнее по сравнению с кодом робота 1 (см. раздел 3.9), но теперь его намного легче изменить, поскольку числа, зависящие друг от друга, находятся в одном месте. Например, шею можно нарисовать на основе переменной `bodyHeight`. Группа переменных в верхней части кода определяет параметры робота, которые мы будем менять: местоположение, длину туловища и высоту шеи. Вы можете увидеть некоторые из возможных вариаций на рисунке, а в таблице ниже приведены значения переменных, которые им соответствуют:

`y = 390`
`bodyHeight = 180`
`NeckHeight = 40`
	`y = 460`
	`bodyHeight = 260`
	`NeckHeight = 95`
		`y = 310`
		`bodyHeight = 80`
		`NeckHeight = 10`
			`y = 420`
			`bodyHeight = 110`
			`NeckHeight = 140`

При изменении собственного кода для использования переменных вместо чисел тщательно спланируйте изменения, а затем внесите их небольшими шагами. Например, при разработке этой программы каждая переменная создавалась по очереди, чтобы минимизировать сложность перехода. После добавления переменной и запуска кода, чтобы убедиться, что он работает правильно, добавлялась следующая переменная:

```
int x = 60;           // Координата x
int y = 390;          // Координата y
int bodyHeight = 180; // Высота туловища
int neckHeight = 40;  // Длина шеи
int radius = 45;
int ny = y - bodyHeight - neckHeight - radius; // Координата y шеи

size(170, 480);
strokeWeight(2);
background(0, 153, 204);
ellipseMode(RADIUS);

// Шея
stroke(255);
line(x+2, y-bodyHeight, x+2, ny);
line(x+12, y-bodyHeight, x+12, ny);
line(x+22, y-bodyHeight, x+22, ny);

// Антенны
line(x+12, ny, x-18, ny-43);
line(x+12, ny, x+42, ny-99);
line(x+12, ny, x+78, ny+15);

// Туловище
noStroke();
fill(255, 204, 0);
ellipse(x, y-33, 33, 33);
fill(0);
rect(x-45, y-bodyHeight, 90, bodyHeight-33);
fill(255, 204, 0);
rect(x-45, y-bodyHeight+17, 90, 6);

// Голова
fill(0);
ellipse(x+12, ny, radius, radius);
fill(255);
ellipse(x+24, ny-6, 14, 14);
fill(0);
ellipse(x+24, ny-6, 3, 3);
fill(153, 204, 255);
ellipse(x, ny-8, 5, 5);
ellipse(x+30, ny-26, 4, 4);
ellipse(x+41, ny+6, 3, 3);
```

Глава 5

Отклик на внешние события

Код, реагирующий на ввод с помощью мыши, клавиатуры и других устройств, должен работать непрерывно. Чтобы это произошло, поместите строки кода в функцию языка Processing с именем `draw()`.

5.1. Порядок выполнения программы

Код в блоке `draw()` выполняется сверху вниз, а затем повторяется, пока вы не выйдете из программы, нажав кнопку **Остановить** или закрыв рабочее окно. Каждое прохождение через `draw()` называется *кадром*. (Частота кадров по умолчанию составляет 60 кадров в секунду, но ее можно изменить.)

Пример 5.1. Функция `draw()`

Чтобы увидеть, как работает функция `draw()`, запустите этот пример:

```
void draw () {
  // Выводит количество кадров в консоль
  println ("Я рисую");
  println (frameCount);
}
```

Вы увидите в окне консоли следующий текст:

```
Я рисую
1
Я рисую
2
Я рисую
3
...
```

В этом примере функция `println()` выводит в консоль текст «Я рисую», за которым следует текущее значение счетчика кадров, которое хранится в спе-

циальной переменной языка Processing под названием `frameCount` (1, 2, 3, ...). Текст появится в консоли, в черной области внизу окна редактора среды разработки.

Пример 5.2. Функция `setup()`

В дополнение к циклической функции `draw()` в языке Processing есть функция с именем `setup()`, которая запускается только один раз при запуске программы:

```
void setup() {
  println("Я стартую");
}

void draw() {
  println("Я работаю");
}
```

Когда этот код запускается, в консоль записывается следующее:

```
Я стартую
Я работаю
Я работаю
Я работаю
...
```

Текст «Я работаю» продолжает записываться в консоль до тех пор, пока программа не будет остановлена.

В типичной программе код внутри блока `setup()` используется для определения начальных значений. Первая строка – это всегда функция `size()`, за которой часто следует код для установки начального цвета заливки и обводки или, возможно, для загрузки изображений и шрифтов. (Если вы не включите в код функцию `size()`, размер окна отображения будет 100×100 пикселей.)

Теперь вы знаете, как использовать `setup()` и `draw()`, но это еще не все. Существует еще одно место для размещения кода – вы также можете размещать переменные вне `setup()` и `draw()`. Если вы создаете переменную внутри `setup()`, вы не можете использовать ее внутри `draw()`, поэтому вам нужно поместить эти переменные в другое место. Такие переменные называются *глобальными*, потому что их можно использовать где угодно («глобально») в программе. Этот момент станет понятнее, если мы перечислим порядок, в котором выполняется код:

1. Создаются переменные, объявленные вне `setup()` и `draw()`.
2. Код внутри `setup()` запускается один раз.
3. Код внутри `draw()` выполняется постоянно.

Пример 5.3. Глобальная переменная

Следующий пример объединяет все, о чем мы только что говорили, воедино:

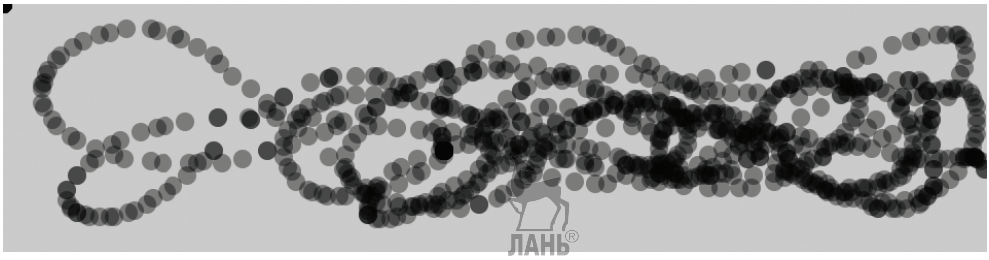
```
int x = 280;
int y = -100;
int diameter = 380;

void setup() {
  size(480, 120);
  fill(102);
}

void draw() {
  background(204);
  ellipse(x, y, diameter, diameter);
}
```

5.2. Отслеживание действий пользователя

Теперь, когда у нас есть код, работающий непрерывно, мы можем отслеживать положение указателя мыши и использовать его координаты для перемещения элементов на экране.

Пример 5.4. Отслеживание указателя мыши

Переменная `mouseX` хранит координату `x`, а переменная `mouseY` хранит координату `y`:



```
void setup() {
  size(480, 120);
  fill(0, 102);
  noStroke();
}

void draw() {
  ellipse(mouseX, mouseY, 9, 9);
}
```

В этом примере каждый раз, когда запускается код в блоке `draw()`, в рабочем окне рисуется новый кружок. Изображение для этого примера было создано путем перемещения указателя мыши. Поскольку заливка круга настроена как частично прозрачная, более плотные черные области показывают, где мышь проводила больше времени, а где двигалась быстро. Чем дальше кружки расположены друг от друга, тем быстрее двигался указатель мыши.

Пример 5.5. Точка следует за указателем мыши

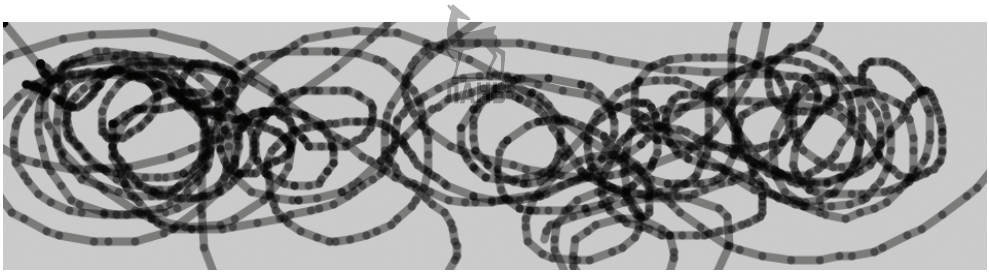
В этом примере, как и в предыдущем, каждый раз, когда запускается код в блоке `draw()`, в рабочем окне рисуется новый кружок. Чтобы обновить экран и отобразить только самый новый кружок, поместите функцию `background()` в начало блока `draw()` до того, как фигура будет нарисована:

```
void setup() {
  size(480, 120);
  fill(0, 102);
  noStroke();
}

void draw() {
  background(204);
  ellipse(mouseX, mouseY, 9, 9);
}
```

Функция `background()` очищает все окно, поэтому всегда помещайте ее перед другими функциями внутри `draw()`; в противном случае фигуры, нарисованные до нее, будут стерты.

Пример 5.6. Непрерывное рисование

Переменные `pmouseX` и `pmouseY` хранят положение курсора мыши в предыдущем кадре. Как и `mouseX` и `mouseY`, эти специальные переменные обновляются каждый раз при запуске блока `draw()`. Вы можете использовать сочетание этих функций для рисования непрерывных линий, соединяя текущее и самое последнее местоположения:



```
void setup() {
  size(480, 120);
  strokeWeight(4);
  stroke(0, 102);
}

void draw() {
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

Пример 5.7. Изменяемая толщина линии

Переменные `pmouseX` и `pmouseY` также можно использовать для расчета скорости движения указателя мыши. Это делается путем измерения расстояния между текущим и предыдущим местоположениями мыши. Если мышь движется медленно, расстояние невелико, но если мышь начинает двигаться быстрее, расстояние увеличивается. Функция `dist()` упрощает этот расчет, как показано в следующем примере. Здесь скорость мыши используется для того, чтобы задать толщину нарисованной линии:

```
void setup() {
  size(480, 120);
  stroke(0, 102);
}

void draw() {
  float weight = dist(mouseX, mouseY, pmouseX, pmouseY);
  strokeWeight(weight);
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

Пример 5.8. Реакция с отставанием

В примере 5.7 значения мыши преобразуются непосредственно в позиции на экране. Но иногда бывает нужно, чтобы значения координат объекта слегка отставали от координат указателя мыши для создания более плавного движения. Этот прием называется *отставанием*. При отставании применяются два значения: текущее значение и значение, к которому нужно двигаться (рис. 5.1). На каждом шаге программы текущее значение немного приближается к целевому значению:

```
float x;
float easing = 0.01;

void setup() {
  size(220, 120);
}

void draw() {
  float targetX = mouseX;
  x += (targetX - x) * easing;
  ellipse(x, 40, 12, 12);
  println(targetX + " : " + x);
}
```

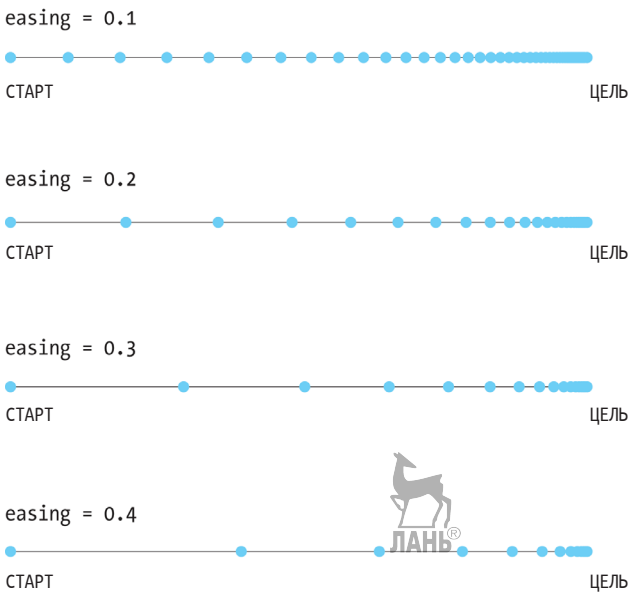


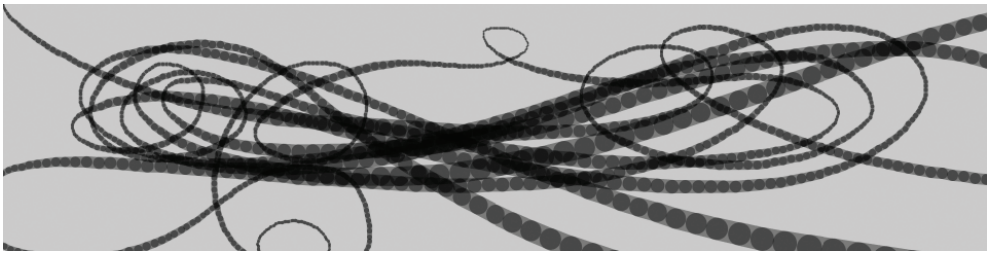
Рис. 5.1 ❖ От значения переменной `easing` зависит количество шагов, необходимых для перехода с одного места на другое

Значение переменной x всегда стремится к targetX. Скорость, с которой оно догоняет targetX, задается в переменной easing числом от 0 до 1. Чем меньше значение переменной easing, тем больше отставание. При значении 1 отставание отсутствует. Когда вы запускаете код из примера 5.8, фактические значения отображаются в консоли с помощью функции println(). Обратите внимание, что при перемещении мыши числа заметно различаются, но когда мышь перестает двигаться, значение x приближается к targetX.

Вся работа в этом примере выполняется в строке, которая начинается с символов x +=. Здесь вычисляется разница между целевым и текущим значениями, затем умножается на переменную easing и прибавляется к x, чтобы приблизиться к целевому значению.

Пример 5.9. Плавные линии с отставанием

В этом примере прием отставания применяется к примеру 5.7. Вы можете убедиться, что линии стали более плавными:



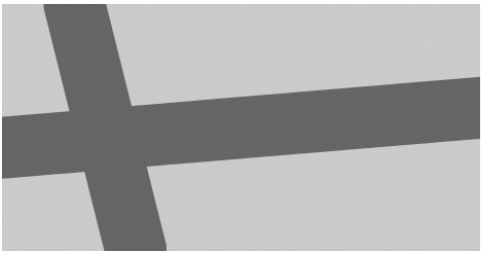
```
float x;
float y;
float px;
float py;
float easing = 0.05;


void setup() {
  size(480, 120);
  stroke(0, 102);
}

void draw() {
  float targetX = mouseX;
  x += (targetX - x) * easing;
  float targetY = mouseY;
  y += (targetY - y) * easing;
  float weight = dist(x, y, px, py);
  strokeWeight(weight);
  line(x, y, px, py);
  py = y;
  px = x;
}
```

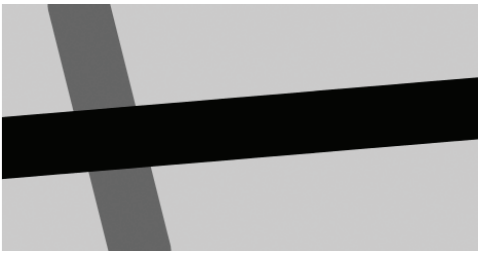
5.3. Нажатие

Помимо местоположения указателя мыши, Processing также отслеживает, нажата ли кнопка мыши. Значение переменной `mousePressed` зависит от того, нажата кнопка мыши или нет. Переменная `mousePressed` относится к типу данных, называемому логическим, что означает, что у нее есть только два возможных значения: `true` и `false`. Значение `mousePressed` истинно при нажатии кнопки.

Пример 5.10. Отслеживание нажатия на кнопку мыши

Используя переменную `mousePressed` вместе с оператором `if`, можно определить, когда строка кода будет выполняться, а когда – нет. Запустите этот пример, прежде чем мы продолжим объяснения:





```
void setup() {
  size(240, 120);
  strokeWeight(30);
}

void draw() {
  background(204);
  stroke(102);
  line(40, 0, 70, height);
  if (mousePressed == true) {
    stroke(0);
  }
  line(0, 70, width, 50);
}
```

В этой программе код внутри блока `if` выполняется только при нажатой кнопке мыши. Когда кнопка не нажата, этот код игнорируется. Подобно циклу `for`, описанному в разделе 4.5, оператор `if` также выполняет проверку, которая дает истинный или ложный результат:

```
if (test) {
  блок кода
}
```

Когда результат теста истинный, выполняется блок кода внутри оператора `if`; когда результат ложный, этот блок не выполняется. Компьютер определяет, является тест истинным или ложным, оценивая выражение `test` в круглых скобках. (Если вы хотите освежить свою память, обсуждение выражений отношения приведено в примере 4.6.)

Оператор `==` означает проверку взаимного равенства значений слева и справа. Двойной символ `==` оператора *эквивалентности* отличается от оператора *присваивания*, одиночного символа `=`. Оператор `==` спрашивает «Равны ли эти значения между собой?», а оператор `=` присваивает значение переменной.

Даже опытные программисты часто ошибаются: пишут в коде одиночный символ `=`, когда следовало написать `==`. Среда разработки Processing не всегда предупреждает вас о такой ошибке, поэтому будьте внимательны.

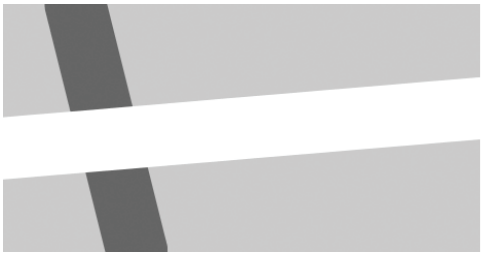
В качестве альтернативы проверку в `draw()` можно записать так:

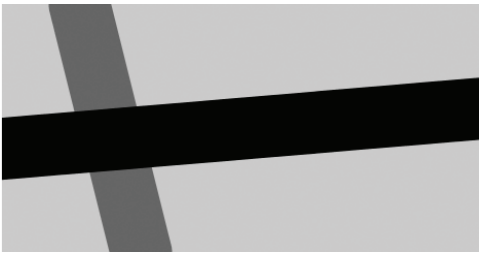

```
if (mousePressed) {
```

Логические переменные, включая `mousePressed`, не нуждаются в явном сравнении при помощи оператора `==`, потому что они могут быть только истинными или ложными.

Пример 5.11. Реакция на отсутствие нажатия мыши

Единственный блок `if` дает вам выбор: запустить код или игнорировать его. Вы можете расширить блок `if` блоком `else`, что позволит вашей программе выбирать между двумя вариантами. Код внутри блока `else` запускается, когда результат проверки блока `if` ложный. Например, цвет линии может быть белым, когда кнопка мыши не нажата, и может измениться на черный, когда кнопка нажата:



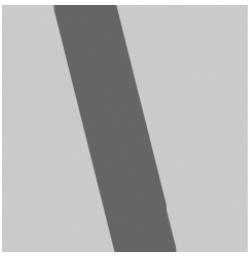


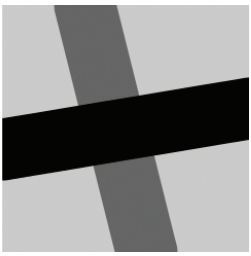
```
void setup() {
  size(240, 120);
  strokeWeight(30);
}
```

```
void draw() {
  background(204);
  stroke(102);
  line(40, 0, 70, height);
  if (mousePressed) {
    stroke(0);
  } else {
    stroke(255);
  }
  line(0, 70, width, 50);
}
```

Пример 5.12. Распознавание кнопок мыши

Processing также отслеживает, какая кнопка нажата, если на вашей мыши больше одной кнопки. Переменная `mouseButton` может принимать одно из трех значений: `LEFT`, `CENTER` или `RIGHT`. Чтобы проверить, какая кнопка была нажата, необходимо использовать оператор `==`, как показано ниже:





```
void setup() {
  size(120, 120);
  strokeWeight(30);
}

void draw() {
  background(204);
  stroke(102);
  line(40, 0, 70, height);
  if (mousePressed) {
    if (mouseButton == LEFT) {
      stroke(255);
    } else {
      stroke(0);
    }
    line(0, 70, width, 50);
  }
}
```

В программе может быть намного больше структур `if` и `else` (рис. 5.2), чем в предыдущих коротких примерах. Их можно объединить в длинную серию,

где каждый условный оператор проверяет что-то свое, и блоки `if` могут быть вложены в другие блоки `if` для принятия более сложных решений.

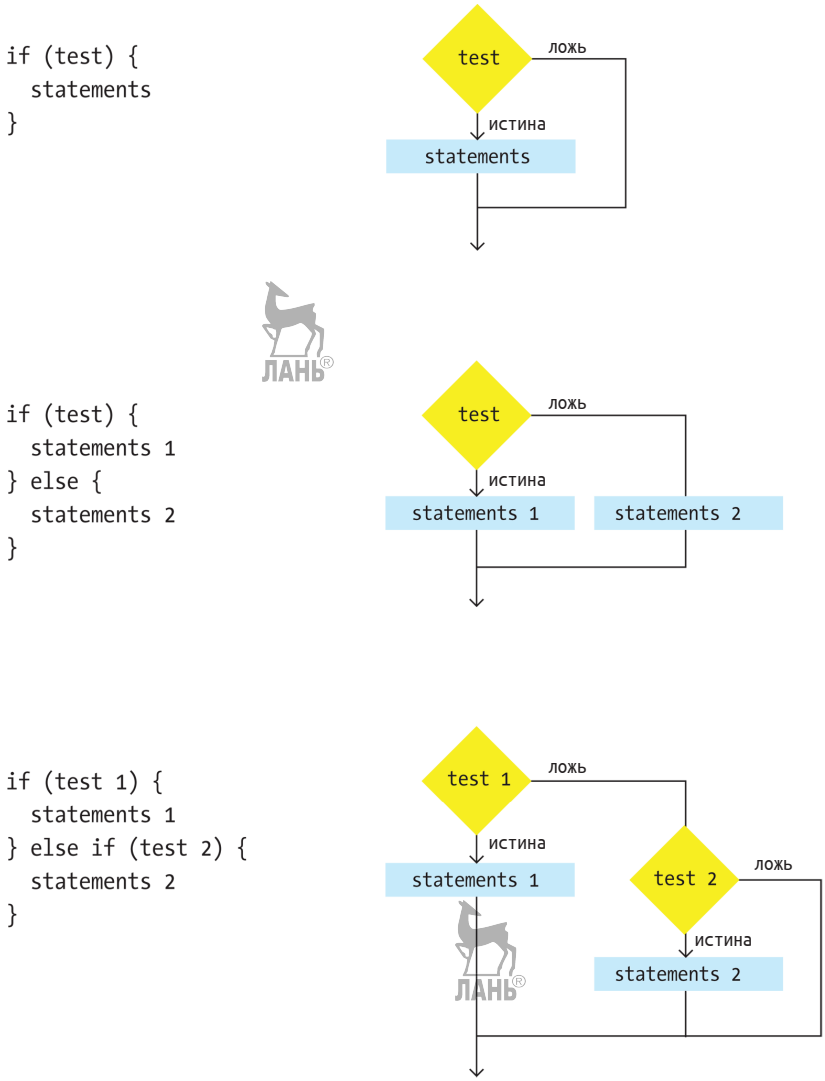


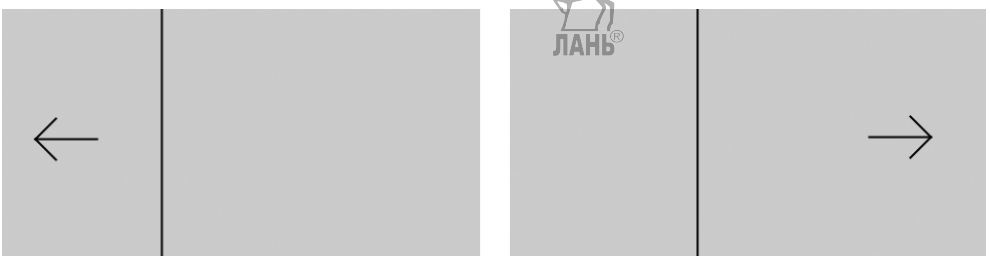
Рис. 5.2 ❖ Структура `if` и `else` принимает решения о том, какие блоки кода запускать

5.4. Расположение указателя мыши

Условный оператор `if` можно использовать со значениями `mouseX` и `mouseY` для определения расположения указателя мыши в окне.

Пример 5.13. Движение объекта к курсору

В этом примере код проверяет, находится ли курсор слева или справа от линии, а затем перемещает линию в сторону курсора:

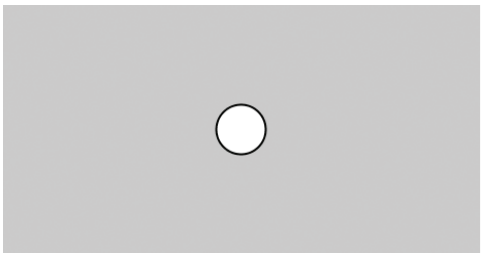


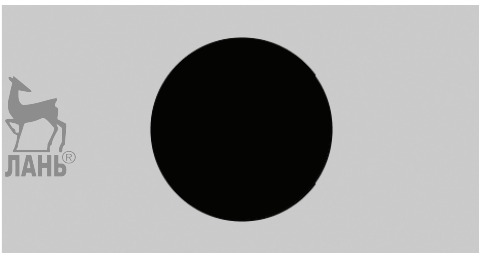
```
float x;
int offset = 10;

void setup() {
  size(240, 120);
  x = width/2;
}

void draw() {
  background(204);
  if (mouseX > x) {
    x += 0.5;
    offset = -10;
  }
  if (mouseX < x) {
    x -= 0.5;
    offset = 10;
  }
  // Рисование стрелки влево или вправо в зависимости от переменной offset
  line(x, 0, x, height);
  line(mouseX, mouseY, mouseX + offset, mouseY - 10);
  line(mouseX, mouseY, mouseX + offset, mouseY + 10);
  line(mouseX, mouseY, mouseX + offset*3, mouseY);
}
```

Чтобы создавать программы с графическим пользовательским интерфейсом (кнопки, флажки, полосы прокрутки и т. д.), нам нужно уметь писать код, который знает, когда курсор находится внутри замкнутой области экрана. В следующих двух примерах показано, как проверить, находится ли курсор внутри круга и прямоугольника. Код написан по модульному принципу с переменными, поэтому его можно использовать для проверки нахождения курсора внутри любого круга и прямоугольника, изменяя значения соответствующих переменных.

Пример 5.14. Границы круга

Для проверки того, находится ли курсор внутри круга, мы используем функцию `dist()`, чтобы получить расстояние от центра круга до курсора, затем проверяем, меньше ли это расстояние, чем радиус круга (рис. 5.3). Если это так, то курсор находится внутри. В этом примере, когда курсор находится в области круга, его размер увеличивается:





```
int x = 120;
int y = 60;
int radius = 12;

void setup() {
  size(240, 120);
  ellipseMode(RADIUS);
}

void draw() {
  background(204);
  float d = dist(mouseX, mouseY, x, y);
  if (d < radius) {
    radius++;
    fill(0);
  } else {
    fill(255);
  }
  ellipse(x, y, radius, radius);
}
```

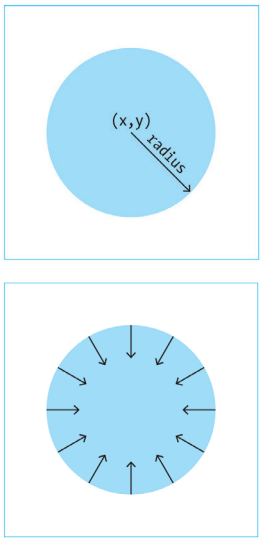
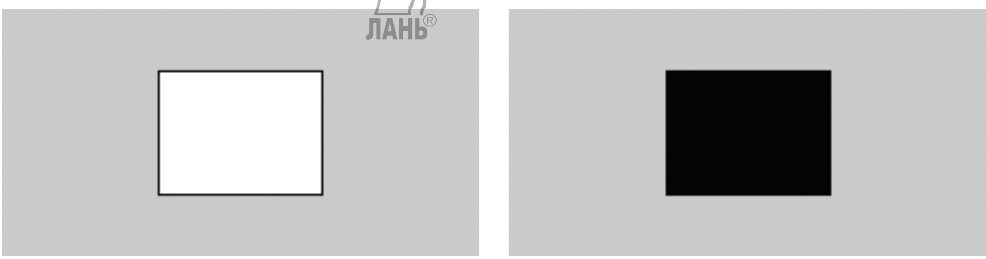


Рис. 5.3 ❖ Тест на попадание курсора в круг.
Когда расстояние между курсором и центром круга меньше радиуса, курсор находится внутри круга

Пример 5.15. Границы прямоугольника

Чтобы проверить, находится ли курсор внутри прямоугольника, мы используем другой подход – выполняем четыре отдельных теста, чтобы проверить, находится ли курсор на внутренней стороне относительно каждого края прямоугольника, и если все четыре результата истинные, мы знаем, что курсор находится внутри. Принцип проверки показан на рис. 5.4. Каждый шаг по отдельности прост, но программа выглядит сложнее, если собрать все шаги вместе:



```
int x = 80;
int y = 30;
int w = 80;
int h = 60;

void setup() {
  size(240, 120);
}
```

```
void draw() {
  background(204);
  if ((mouseX > x) && (mouseX < x+w) &&
      (mouseY > y) && (mouseY < y+h)) {
    fill(0);
  } else {
    fill(255);
  }
  rect(x, y, w, h);
}
```

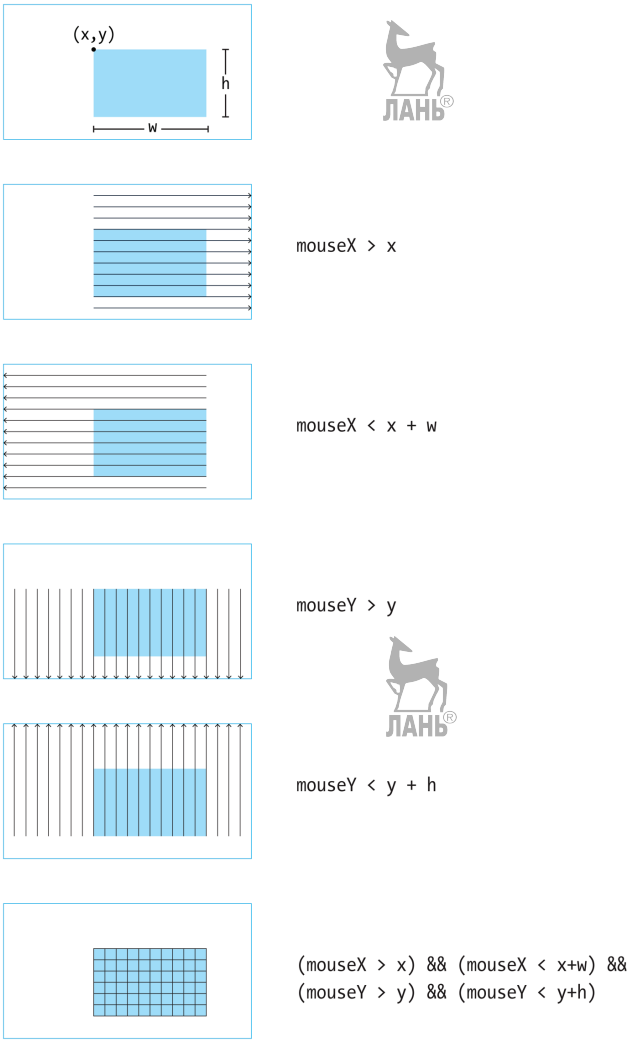


Рис. 5.4 ❖ Проверки на попадание курсора в прямоугольник. Когда все четыре проверки выполнены и истинны, курсор находится внутри прямоугольника

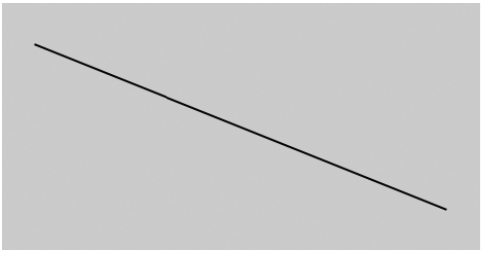
Проверка в операторе `if` немного сложнее, чем мы видели. Четыре отдельных теста (например, `mouseX > x`) комбинируются при помощи логического оператора И (AND), который записывается символами `&&`, чтобы гарантировать, что каждое условное выражение в последовательности истинно. Если хотя бы одно из них ложно, результат проверки будет ложным, и цвет заливки не изменится на черный. Более подробно про оператор `&&` вы можете прочитать в справочнике по языку Processing.

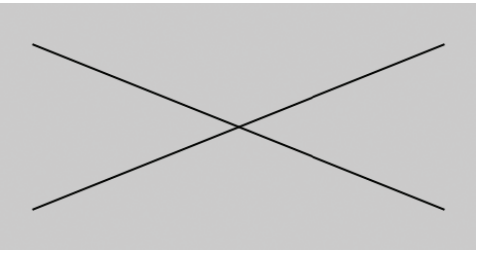
5.5. События клавиатуры

Processing отслеживает, нажата ли какая-либо клавиша на клавиатуре, а также запоминает последнюю нажатую клавишу. Как и переменная `mousePressed`, переменная `keyPressed` имеет значение `true`, когда нажата какая-либо клавиша, и `false`, когда никакие клавиши не нажаты.

Пример 5.16. Нажатие клавиши

В этом примере вторая линия отображается только при нажатии клавиши:





```
void setup() {
  size(240, 120);
}

void draw() {
  background(204);
  line(20, 20, 220, 100);
  if (keyPressed) {
    line(220, 20, 20, 100);
  }
}
```

В переменной `key` хранится последняя нажатая клавиша. Тип данных для этого значения – `char`, что является сокращением от слова «`character`» (символ), но обычно произносится как «ча». Переменная `char` может хранить любой отдельный символ, включая буквы алфавита, цифры и спецсимволы. В отличие от строкового значения (см. пример 7.8), которое выделяется двой-

ными кавычками, тип данных `char` определяется одинарными кавычками. Вот как объявляется и назначается переменная типа `char`:

```
char c = 'A'; // Объявляет переменную c и присваивает ей 'A'
```

Эти попытки присвоения вызовут ошибку:

```
char c = "A"; // Ошибка! Невозможно присвоить значение String переменной типа char
char h = A;   // Ошибка! Отсутствуют одинарные кавычки вокруг 'A'
```

В отличие от логической переменной `keyPressed`, которая принимает значение `false` при каждом отпускании клавиши, переменная `key` сохраняет свое значение до нажатия следующей клавиши. В следующем примере значение `key` используется для вывода символа на экран. Каждый раз, когда нажимается новая клавиша, значение обновляется, и рисуется новый символ. Некоторые клавиши, такие как **Shift** и **Alt**, не имеют отображаемого символа, поэтому при их нажатии ничего не отображается.

Пример 5.17. Рисование букв

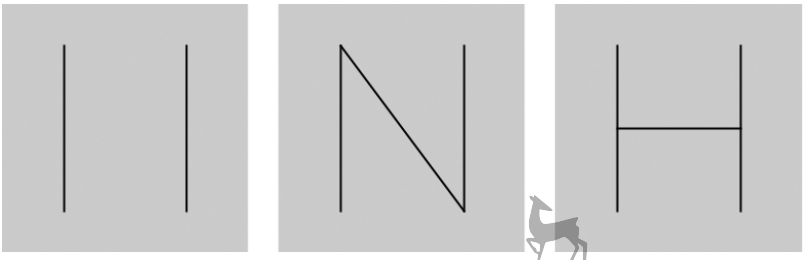
В этом примере представлена функция `textSize()` для установки размера букв, функция `textAlign()` для позиционирования текста по его координате `x` и функция `text()` для рисования буквы. Эти функции более подробно описаны в разделе 7.2.


```
void setup() {
  size(120, 120);
  textSize(64);
  textAlign(CENTER);
}

void draw() {
  background(0);
  text(key, 60, 80);
}
```

Используя структуру `if`, мы можем проверить, нажата ли конкретная клавиша, и выполнить нужное ответное действие.

Пример 5.18. Проверка нажатия заданных клавиш

В этом примере мы проверяем, нажаты ли клавиши **H** или **N**. Мы используем оператор эквивалентности ==, чтобы проверить, совпадает ли значение клавиши с искомыми символами:



```
void setup() {
  size(120, 120);
}

void draw() {
  background(204);
  if (keyPressed) {
    if ((key == 'h') || (key == 'H')) {
      line(30, 60, 90, 60);
    }
    if ((key == 'n') || (key == 'N')) {
      line(30, 20, 90, 100);
    }
  }
  line(30, 20, 30, 100);
  line(90, 20, 90, 100);
}
```

Когда мы следим за нажатием клавиш **H** или **N**, нам нужно проверять как строчные, так и прописные буквы на тот случай, если кто-то нажал клавишу **Shift** или **Caps Lock**. Мы объединяем эти два теста с помощью оператора логического ИЛИ, который представляет собой две вертикальные черты ||. Если мы переведем второй оператор `if` в этом примере на простой разговорный язык, он будет звучать так: «Если нажата клавиша “h” ИЛИ нажата клавиша “H”». В отличие от логического И (`&&`), достаточно только одному из условий быть истинным, чтобы результат проверки был истинным.

Некоторые клавиши труднее обнаружить, потому что они не привязаны к определенной букве. Такие клавиши, как **Shift**, **Alt** и клавиши со стрелками, возвращают код, не связанный с буквой, поэтому требуется дополнительный шаг, чтобы выяснить, нажаты ли они. Во-первых, нам нужно проверить, является ли нажатая клавиша закодированной, а затем мы проверяем код с помощью переменной `keyCode`, чтобы узнать, что это за клавиша. Наиболее

часто используемые значения keyCode – это ALT, CONTROL и SHIFT, а также клавиши со стрелками UP, DOWN, LEFT и RIGHT.

Пример 5.19. Перемещение с помощью клавиш со стрелками

В следующем примере показано, как проверить нажатие клавиш со стрелками влево или вправо для перемещения прямоугольника:

```
int x = 215;

void setup() {
  size(480, 120);
}

void draw() {
  if (keyPressed && (key == CODED)) { // Это не буквенная клавиша
    if (keyCode == LEFT) { // Это левая стрелка
      x--;
    } else if (keyCode == RIGHT) { // Это правая стрелка
      x++;
    }
  }
  rect(x, 45, 50, 50);
}
```


5.6. Сопоставление диапазонов

Числа, которые генерируются с помощью мыши и клавиатуры, часто бывает нужно изменить, чтобы их можно было использовать в программе. Например, если ширина экрана составляет 1920 пикселей, а значения mouseX используются для установки цвета фона, значения в диапазоне от 0 до 1920, которые может принимать mouseX, желательно преобразовать в значения, лежащие в диапазоне от 0 до 255, чтобы лучше контролировать цвет. Это преобразование может быть выполнено с помощью уравнения или с помощью функции map().

Пример 5.20. Сопоставление значений с диапазоном

В этом примере расположение двух линий зависит от переменной mouseX. Серая линия синхронизируется с положением курсора, а черная линия остается рядом с серой в районе центра экрана и отодвигается дальше от нее в области левого и правого краев:

```
void setup() {
  size(240, 120);
  strokeWeight(12);
}

void draw() {
  background(204);
  stroke(102);
  line(mouseX, 0, mouseX, height); // Серая линия
  stroke(0);
  float mx = mouseX/2 + 60;
  line(mx, 0, mx, height); // Черная линия
}
```

Функция `map()` – это более общий способ внести в значения изменения подобного рода. Она преобразует переменную из одного диапазона чисел в другой. Первый параметр – это переменная, которая должна быть преобразована, второй и третий параметры – это исходные наименьшее и наибольшее значения этой переменной, а четвертый и пятый параметры – это желаемые наименьшее и наибольшее значения. Функция `map()` выполняет необходимые математические вычисления.

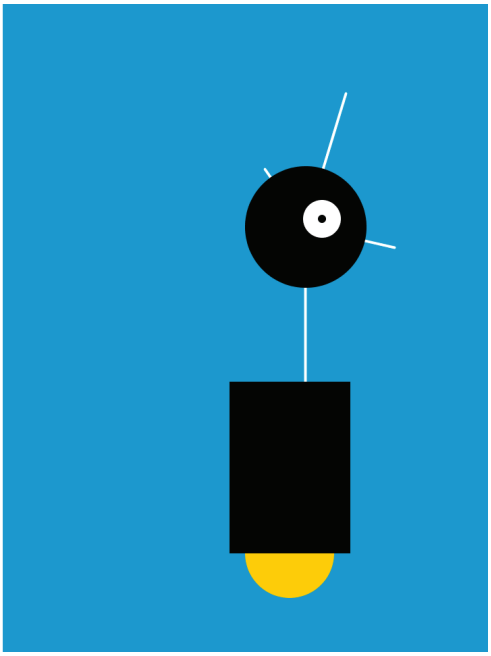
Пример 5.21. Сопоставление значений при помощи функции `map()`

Этот пример делает то же самое, что пример 5.20, но с помощью функции `map()`:

```
void setup() {
  size(240, 120);
  strokeWeight(12);
}

void draw() {
  background(204);
  stroke(102);
  line(mouseX, 0, mouseX, height); // Серая линия
  stroke(0);
  float mx = map(mouseX, 0, width, 60, 180);
  line(mx, 0, mx, height); // Черная линия
}
```

Функция `map()` упрощает чтение кода, поскольку наименьшее и наибольшее значения четко записаны как параметры. В этом примере значения `mouseX` в диапазоне между 0 и шириной окна преобразуются в число от 60 (когда `mouseX` равно 0) до 180 (когда `mouseX` равно ширине). Вы встретите полезную функцию `map()` во многих примерах в этой книге.

5.7. Робот 3: отклик на воздействие



Эта программа использует переменные, представленные в разделе 4.6, и позволяет изменять их значение во время работы программы, чтобы фигуры реагировали на действия мышью. Код внутри блока `draw()` многократно выполняется каждую секунду. В каждом кадре переменные, определенные в программе, изменяются в соответствии со значениями переменных `mouseX` и `mousePressed`.

Значение `mouseX` управляет положением робота с помощью рассмотренного ранее метода отставания, чтобы реакция робота была не столь мгновенной и казалась более естественной. Когда нажата кнопка мыши, значения `NeckHeight` и `bodyHeight` изменяются, чтобы робот стал коротким:

```
float x = 60;           // Координата X
float y = 440;          // Координата Y
int radius = 45;        // Радиус головы
int bodyHeight = 160;   // Высота тела
int neckHeight = 70;    // Длина шеи

float easing = 0.04;

void setup() {
  size(360, 480);
  ellipseMode(RADIUS);
}
```

```
void draw() {
  strokeWeight(2);

  int targetX = mouseX;
  x += (targetX - x) * easing;

  if (mousePressed) {
    neckHeight = 16;
    bodyHeight = 90;
  } else {
    neckHeight = 70;
    bodyHeight = 160;
  }

  float neckY = y - bodyHeight - neckHeight - radius;

  background(0, 153, 204);

  // Шея
  stroke(255);
  line(x+12, y-bodyHeight, x+12, neckY);

  // Антенны
  line(x+12, neckY, x-18, neckY-43);
  line(x+12, neckY, x+42, neckY-99);
  line(x+12, neckY, x+78, neckY+15);

  // Тело
  noStroke();
  fill(255, 204, 0);
  ellipse(x, y-33, 33, 33);
  fill(0);
  rect(x-45, y-bodyHeight, 90, bodyHeight-33);

  // Голова
  fill(0);
  ellipse(x+12, neckY, radius, radius);
  fill(255);
  ellipse(x+24, neckY-6, 14, 14);
  fill(0);
  ellipse(x+24, neckY-6, 3, 3);
}
```

Глава 6

Перемещение, вращение и масштабирование

Еще один способ позиционирования и перемещения объектов на экране – изменение системы координат экрана. Например, вы можете переместить фигуру на 50 пикселей вправо или переместить положение координаты (0, 0) на 50 пикселей вправо – видимый результат будет одинаковым.

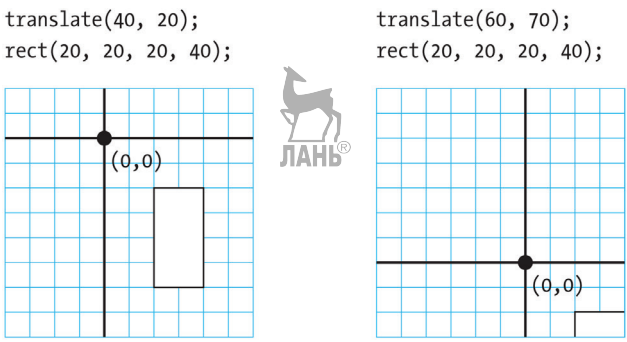


Рис. 6.1 ❖ Перемещение системы координат

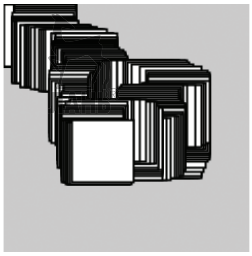
Изменяя систему координат по умолчанию, мы можем создавать различные преобразования, включая *перемещение*, *вращение* и *масштабирование*.

6.1. Перемещение

Работа с преобразованиями объектов и координат может быть сложной, но функция `translate()` является наиболее простой, поэтому мы начнем с нее. Как показано на рис. 6.1, эта функция может сдвигать систему координат влево, вправо, вверх и вниз.

Пример 6.1. Перемещение объекта на экране

В этом примере обратите внимание, что прямоугольник нарисован с координатой (0,0), но он перемещается по экрану, потому что на него влияет функция `translate()`:

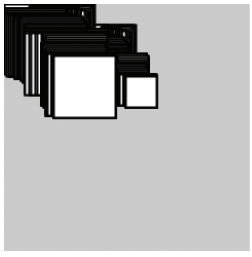




```
void setup() {
  size(120, 120);
}

void draw() {
  translate(mouseX, mouseY);
  rect(0, 0, 30, 30);
}
```

Функция `translate()` устанавливает координату (0,0) экрана в положение указателя мыши (`mouseX` и `mouseY`). При каждом выполнении блока `draw()` функция `rect()` рисует прямоугольник в новой точке отсчета, полученной из текущего местоположения курсора.

Пример 6.2. Множественные перемещения

После того как преобразование выполнено, оно применяется ко всем последующим функциям рисования. Обратите внимание, что происходит, когда добавляется вторая функция перемещения для управления вторым прямоугольником:



```
void setup() {
  size(120, 120);
}
```

```
void draw() {
  translate(mouseX, mouseY);
  rect(0, 0, 30, 30);
}
```

Значения параметров функций `translate()` складываются. Меньший прямоугольник был смещен на величину `mouseX+35` и `mouseY+10`. Координаты `x` и `y` для обоих прямоугольников равны (0,0), но функции `translate()` перемещают их в другие позиции на экране.

Однако, несмотря на то что преобразования суммируются в блоке `draw()`, они сбрасываются при каждом новом выполнении `draw()`.

6.2. Вращение

Функция `rotate()` вращает систему координат. У нее есть один параметр – угол поворота (в радианах). Вращение всегда происходит относительно точки (0,0); это называется вращением вокруг начала координат. На рис. 3.2 в примере 3.7 показаны значения углов в радианах. На рис. 6.2 показана разница между вращением с положительным и отрицательным аргументами функции.

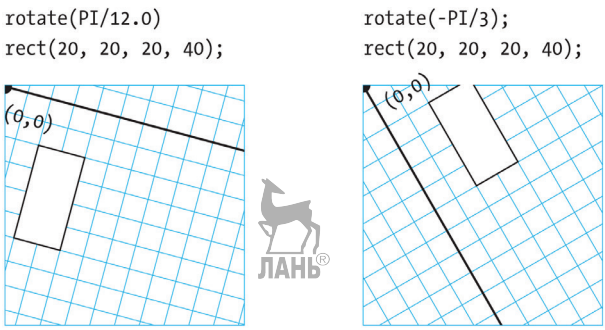
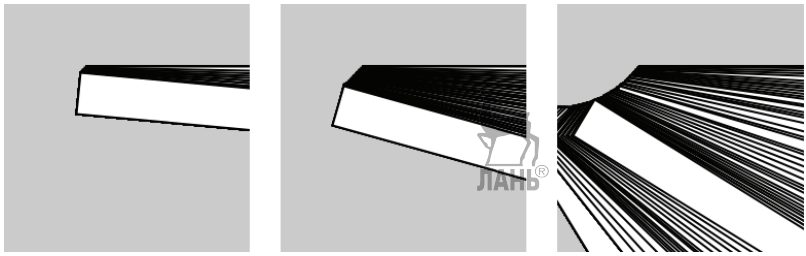


Рис. 6.2 ❖ Вращение координат

Пример 6.3. Вращение на переменный угол

Чтобы повернуть фигуру, сначала определите угол поворота с помощью `rotate()`, затем нарисуйте фигуру. В этом скетче величина поворота (`mouseX/100.0`) будет находиться в диапазоне от 0 до 1,2, потому что `mouseX` будет находиться в диапазоне от 0 до 120 – это ширина рабочего окна, указанная с помощью функции `size()`. Обратите внимание, что вы должны делить на `100,0`, а не на `100`, из-за того, как в языке Processing работают разные типы чисел (см. раздел 4.2).



```
void setup() {
  size(120, 120);
}

void draw() {
  rotate(mouseX / 100.0);
  rect(40, 30, 160, 20);
}
```

Пример 6.4. Вращение вокруг собственного центра

Чтобы повернуть фигуру вокруг собственного центра, она должна быть нарисована с координатой (0,0) посередине. В этом примере, поскольку форма имеет ширину 160 и высоту 20, как определено аргументами функции `rect()`, она рисуется с опорными координатами (–80, –10), чтобы точка начала координат (0,0) оказалась в центре фигуры:

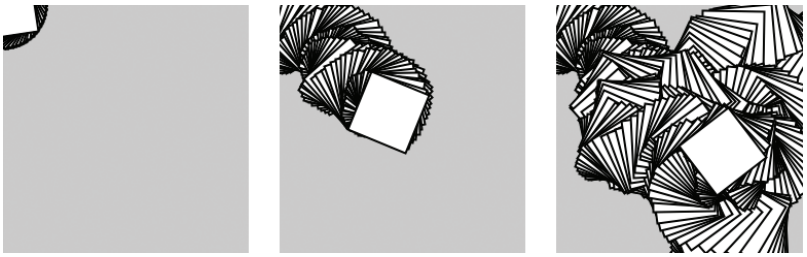
```
void setup() {
  size(120, 120);
}

void draw() {
  rotate(mouseX / 100.0);
  rect(-80, -10, 160, 20);
}
```

Предыдущая пара примеров показала, как вращать графические объекты вокруг координаты (0,0), но как насчет других возможностей? Вы можете использовать функции `translate()` и `rotate()` вместе. Когда эти функции объединены, порядок, в котором они выполняются, влияет на результат. Если

система координат сначала смещается, а затем вращается, результат будет отличаться от ситуации, когда система координат сначала вращается, а затем смещается.

Пример 6.5. Смещение, затем вращение

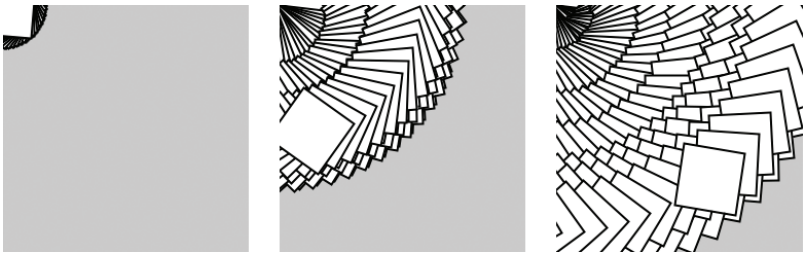
Чтобы повернуть фигуру вокруг ее центральной точки в месте, расположенном на экране вдали от начала координат, сначала используйте `translate()`, чтобы переместиться в то место, где вы хотите видеть фигуру, затем вызовите функцию `rotate()` и нарисуйте фигуру с центром в начале координат (0,0):



```
float angle = 0;
void setup() {
  size(120, 120);
}

void draw() {
  translate(mouseX, mouseY);
  rotate(angle);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}
```

Пример 6.6. Вращение, затем смещение

Следующий пример идентичен примеру 6.5, за исключением того, что `translate()` и `rotate()` поменялись местами. Теперь фигура вращается вокруг левого верхнего угла рабочего окна на заданном расстоянии от угла, установленном с помощью `translate()`:

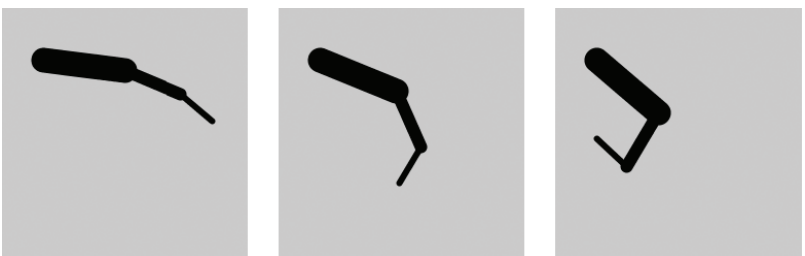


```
float angle = 0.0;

void setup() {
  size(120, 120);
}

void draw() {
  rotate(angle);
  translate(mouseX, mouseY);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}
```

Другой вариант – использовать функции `rectMode()`, `ellipseMode()`, `imageMode()` и `shapeMode()`, которые упрощают рисование фигур относительно их центра. Вы можете прочитать об этих функциях в справочнике по языку Processing.

Пример 6.7. Рычаг с шарнирными сочленениями

В этом примере мы использовали серию функций `translate()` и `rotate()`, чтобы нарисовать шарнирно-сочлененный рычаг, который изгибается вперед и назад. Каждая функция `translate()` дополнительно перемещает положение линий, а каждая функция `rotate()` добавляет угол поворота, чтобы сильнее согнуть рычаг:



```
float angle = 0.0;
float angleDirection = 1;
float speed = 0.005;

void setup() {
  size(120, 120);
}

void draw() {
  background(204);
  translate(20, 25); // Переход к начальной позиции
  rotate(angle);
  strokeWeight(12);
  line(0, 0, 40, 0);
  translate(40, 0);  // Переход к следующему сочленению
  rotate(angle * 2.0);
```

```
  strokeWeight(6);
  line(0, 0, 30, 0);
  translate(30, 0);  // Переход к следующему сочленению
  rotate(angle * 2.5);
  strokeWeight(3);
  line(0, 0, 20, 0);
  angle += speed * angleDirection;
  if ((angle > QUARTER_PI) || (angle < 0)) {
    angleDirection = -angleDirection;
  }
}
```

Значение переменной `angle` увеличивается от `0` до `QUARTER_PI` (одна четверть значения π), затем уменьшается, пока не станет меньше нуля, после чего цикл повторяется. Значение переменной `angleDirection` всегда равно 1 или –1, чтобы значение угла соответственно увеличивалось или уменьшалось.

6.3. Масштабирование

Функция `scale()` растягивает или сжимает координаты на экране, поэтому все, что нарисовано в рабочем окне, увеличивается или уменьшается в размерах. Например, вы можете использовать функцию `scale(1.5)`, чтобы увеличить все фигуры на 150 % от исходного размера, или `scale(3)`, чтобы увеличить их в три раза. Использование `scale(1)` не даст никакого эффекта, потому что у всех фигур останется размер 100 % от оригинала. Чтобы уменьшить размер фигур вдвое, используйте `scale(0.5)`.

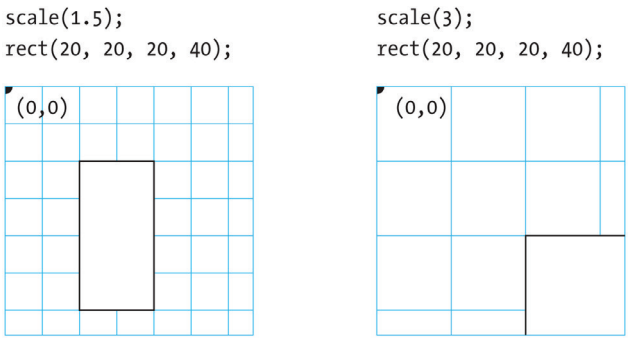
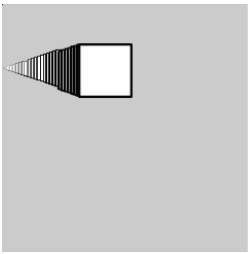
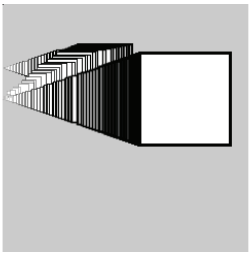
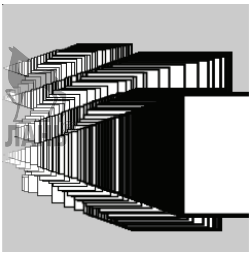


Рис. 6.3 ❖ Масштабирование координат

Пример 6.8. Масштабирование изображения

Как и `rotate()`, функция `scale()` выполняет преобразование относительно начала координат. Следовательно, как и в случае с `rotate()`, чтобы масштабировать фигуру относительно ее центра, переместите центр координат в нужное

местоположение, масштабируйте координатную сетку, а затем нарисуйте фигуру с центром в начале координат (0,0):

```
void setup() {
  size(120, 120);
}

void draw() {
  translate(mouseX, mouseY);
  scale(mouseX / 60.0);
  rect(-15, -15, 30, 30);
}
```

Пример 6.9. Сохранение постоянства штрихов

Глядя на толстые линии в примере 6.8, вы можете заметить, что функция `scale()` влияет на толщину штриха. Чтобы поддерживать постоянную толщину штриха при масштабировании фигуры, разделите желаемую толщину штриха на значение переменной `scalar`:


```
void setup() {
  size(120, 120);
}

void draw() {
  translate(mouseX, mouseY);
  float scalar = mouseX / 60.0;
  scale(scalar);
  strokeWeight(1.0 / scalar);
  rect(-15, -15, 30, 30);
}
```

6.4. Сохранение и восстановление системы координат

Чтобы эффекты преобразования не влияли на последующие команды, их нужно изолировать. Используйте для этого функции `pushMatrix()` и `popMatrix()`. После вызова функции `pushMatrix()` она сохраняет копию текущей системы координат, а затем координаты можно восстановить из копии при помощи функции `popMatrix()`. Это полезно, когда преобразования необходимы для одной фигуры, но не нужны для другой.

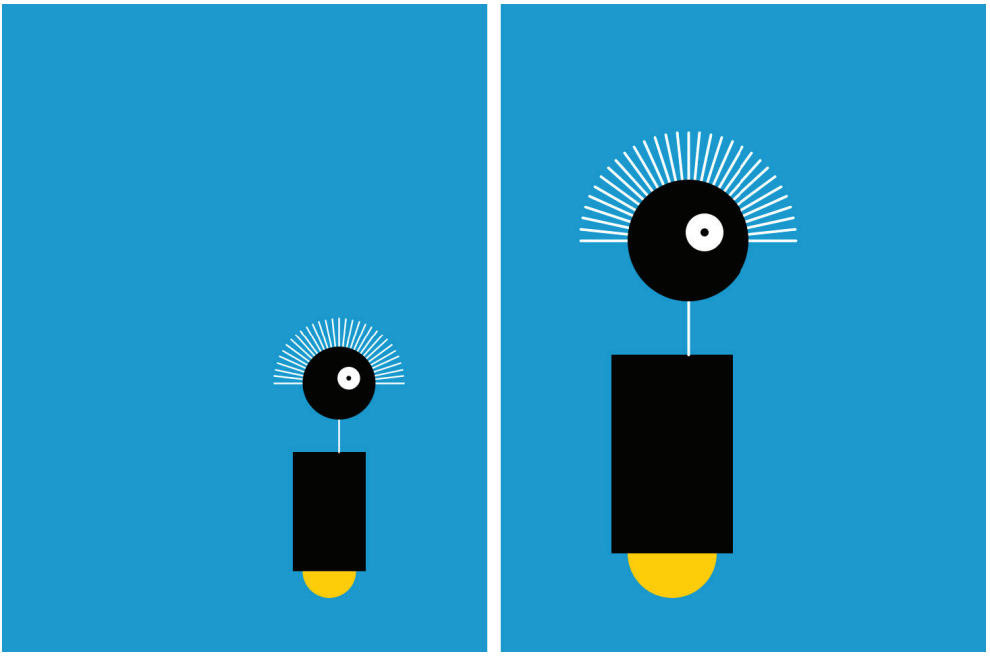
Пример 6.10. Изолированные преобразования

В этом примере меньший прямоугольник всегда отображается в одной и той же позиции, потому что эффект от `translate(mouseX, mouseY)` отменяется функцией `popMatrix()`:

```
void setup() {
  size(120, 120);
}

void draw() {
  pushMatrix();
  translate(mouseX, mouseY);
  rect(0, 0, 30, 30);
  popMatrix();
  translate(35, 10);
  rect(0, 0, 15, 15);
}
```

Функции `pushMatrix()` и `popMatrix()` всегда используются парами. Для каждого вызова `pushMatrix()` вам необходимо иметь соответствующий вызов `popMatrix()`.

6.5. Робот 4: смещение, вращение, масштабирование



В этом модифицированном скетче для рисования робота используются функции `translate()`, `rotate()` и `scale()`. По сравнению со скетчем из раздела 5.7 функция ~~`translate()`~~ облегчает чтение кода. Обратите внимание, что в этом скетче значение `x` больше не нужно добавлять к аргументу каждой функции рисования, потому что `translate()` перемещает систему координат целиком.

Точно так же функция `scale()` используется для установки размеров всего робота. Когда мышь не нажата, изображение отображается в масштабе 60 %, а при нажатии оно приобретает размер 100 % по отношению к исходным координатам.

Функция `rotate()` используется внутри цикла, чтобы нарисовать линию, немного повернуть ее, затем нарисовать вторую линию, потом повернуть еще немного и так далее, пока цикл не нарисовал 30 линий на половине от полного круга, чтобы на голове робота образовалась стильная прическа:

```
float x = 60;           // Координата X
float y = 440;          // Координата Y
int radius = 45;        // Радиус головы
int bodyHeight = 180;   // Высота тела
int neckHeight = 40;    // Длина шеи

float easing = 0.04;
```

```
void setup() {
  size(360, 480);
  ellipseMode(RADIUS);
}

void draw() {
  strokeWeight(2);

  float neckY = -1 * (bodyHeight + neckHeight + radius);

  background(0, 153, 204);

  translate(mouseX, y);  // Перенос в точку (mouseX, y)

  if (mousePressed) {
    scale(1.0);
  } else {
    scale(0.6);  // 60 %, если кнопка мыши нажата
  }

  // Туловище
  noStroke();
  fill(255, 204, 0);
  ellipse(0, -33, 33, 33);
  fill(0);
  rect(-45, -bodyHeight, 90, bodyHeight-33);

  // Шея
  stroke(255);
  line(12, -bodyHeight, 12, neckY);

  // Волосы
  pushMatrix();
  translate(12, neckY);
  float angle = -PI/30.0;
  for (int i = 0; i <= 30; i++) {
    line(80, 0, 0, 0);
    rotate(angle);
  }
  popMatrix();

  // Голова
  noStroke();
  fill(0);
  ellipse(12, neckY, radius, radius);
  fill(255);
  ellipse(24, neckY-6, 14, 14);
  fill(0);
  ellipse(24, neckY-6, 3, 3);
}
```

Глава 7

Медиафайлы

Processing способен на большее, чем рисование простых линий и фигур. Пришло время узнать, как загружать в наши программы растровые изображения, векторные файлы и шрифты, чтобы расширить возможности создания визуальных эффектов за счет использования фотографий, подробных диаграмм и различных шрифтов.

Processing использует для хранения таких файлов папку с именем `data`, поэтому вам никогда не придется думать об их местонахождении при перемещении и экспорте скетчей. Мы разместили в интернете мультимедийные файлы, которые вы можете использовать в примерах из этой главы по адресу http://www.processing.org/learning/books/media.zip.

Загрузите этот архивный файл, распакуйте его на рабочий стол (или в другое удобное место) и запомните его местонахождение.

Чтобы распаковать в Mac OS X, просто дважды щелкните файл, и операционная система создаст папку с именем `media`. В Windows дважды щелкните файл `media.zip`, чтобы открыть новое окно. Перетащите папку `media` из этого окна на рабочий стол.

Создайте новый скетч и выберите опцию **Добавить файл** в меню **Набросок**. Найдите файл `lunar.jpg` в папке `media`, которую вы только что распаковали, и выберите его. Если все прошло хорошо, в области сообщений появится строка **Файл добавлен в скетч**.

Чтобы проверить наличие этого файла, выберите опцию **Показать папку с набросками** в меню **Набросок**. Вы должны увидеть папку с именем `data`, которая содержит копию файла `lunar.jpg`. Когда вы добавляете файл в скетч, автоматически создается папка `data`. Вместо использования команды меню **Добавить файл** вы можете сделать то же самое, перетащив файлы в область редактора кода среды разработки Processing. Файлы будут точно так же скопированы в папку `data` (папка будет создана, если таковой не существует).

Вы также можете создать папку `data` вне главной папки Processing и скопировать туда файлы самостоятельно. Вы не получите сообщение о том, что файлы были добавлены, но это полезный метод, когда вы работаете с большим количеством файлов.

В Windows и Mac OS X расширения файлов по умолчанию скрыты. Рекомендуется изменить этот параметр, чтобы вы всегда видели полное имя своих файлов. В Mac OS X выберите **Настройки** в меню приложения **Finder**, а затем убедитесь, что на вкладке **Дополнительно** установлен флажок **Показать все расширения файлов**. В Windows найдите **Параметры папки** и установите там такой же флажок.

7.1. Изображения

Прежде чем рисовать изображение на экране, необходимо выполнить три шага.

1. Добавьте изображение в папку `data` скетча (инструкции приведены выше).
2. Создайте переменную `PImage` для хранения изображения.
3. Загрузите изображение в переменную с помощью `loadImage()`.

Пример 7.1. Загрузка изображения

После выполнения всех трех шагов вы можете нарисовать изображение на экране с помощью функции `image()`. Первый параметр функции `image()` указывает изображение для рисования; второй и третий задают координаты `x` и `y`:

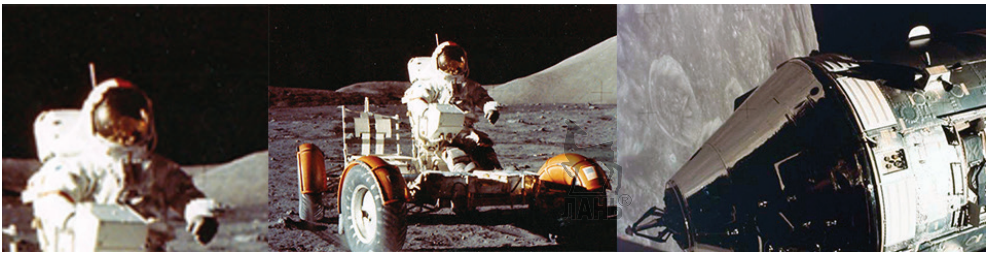
```
PImage img;

void setup() {
  size(480, 120);
  img = loadImage("lunar.jpg");
}

void draw() {
  image(img, 0, 0);
}
```

Необязательные четвертый и пятый параметры устанавливают ширину и высоту для рисования изображения. Если четвертый и пятый параметры не используются, изображение будет отрисовано в том размере, в котором оно было создано.

В следующих примерах показано, как работать с более чем одним изображением в одной программе и как изменить размер изображения.

Пример 7.2. Загрузка нескольких изображений

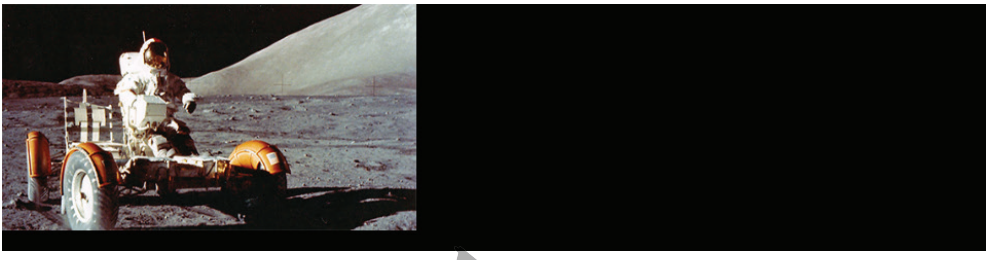
В этом примере вам нужно будет добавить файл `Capsule.jpg` (находящийся в скачанной вами папке `media`) в свой скетч одним из описанных ранее способов:



```
PImage img1;
PImage img2;

void setup() {
  size(480, 120);
  img1 = loadImage("lunar.jpg");
  img2 = loadImage("capsule.jpg");
}

void draw() {
  image(img1, -120, 0);
  image(img1, 130, 0, 240, 120);
  image(img2, 300, 0, 240, 120);
}
```

Пример 7.3. Наведение курсора на изображение

Когда значения `mouseX` и `mouseY` используются как четвертый и пятый аргументы функции `image()`, размер изображения изменяется при перемещении мыши:



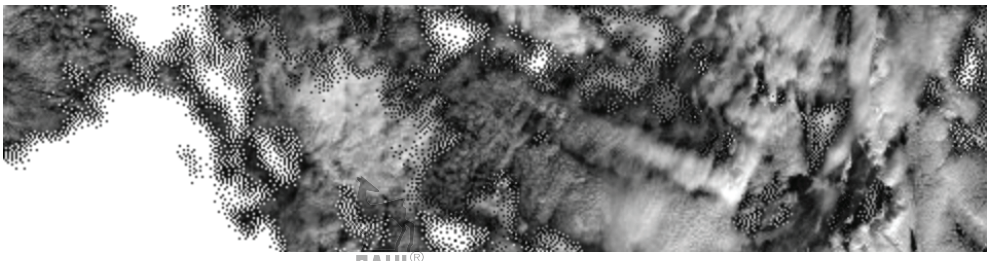
```
PImage img;
  void setup() {
  size(480, 120);
img = loadImage("lunar.jpg");
}

void draw() {
  background(0);
  image(img, 0, 0, mouseX * 2, mouseY * 2);
}
```

Когда отображаемый размер изображения больше или меньше его фактического размера, оно может искажаться. Будьте аккуратны и готовьте изображения того размера, в котором они будут использоваться. Когда размер отображения изменяется с помощью функции `image()`, исходное изображение на жестком диске не изменяется.

Processing может загружать и отображать растровые изображения в форматах JPEG, PNG и GIF. (Векторные фигуры в формате SVG могут отображаться другим способом, как описано в разделе 7.3 далее в этой главе.) Вы можете конвертировать изображения в форматы JPEG, PNG и GIF с помощью таких программ, как GIMP и Photoshop. Большинство цифровых фотоаппаратов сохраняют изображения в формате JPEG, которые намного больше, чем область рисования большинства скетчей Processing, поэтому изменение размера таких изображений до их добавления в папку `data` повысит эффективность ваших скетчей.

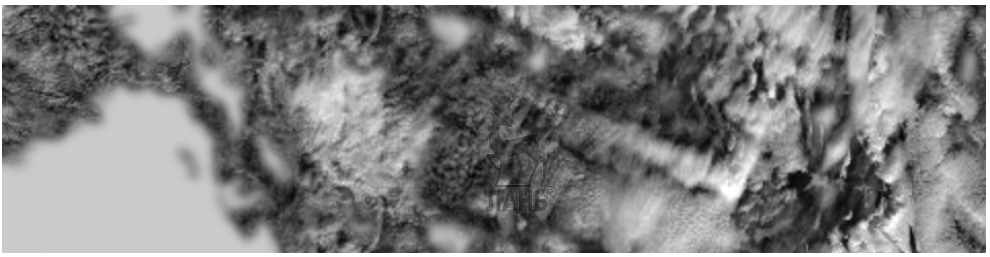
Изображения в формате GIF и PNG поддерживают прозрачность, что означает, что пиксели могут быть невидимыми или частично видимыми (вспомните обсуждение значений `color()` и `alpha` в примере 3.17. Изображения GIF имеют 1-битную прозрачность. Это означает, что пиксели либо полностью непрозрачны, либо полностью прозрачны. Изображения PNG имеют 8-битную прозрачность, то есть каждый пиксель может иметь переменный уровень непрозрачности. В следующих примерах показано различие этих форматов с использованием файлов `clouds.gif` и `clouds.png`, находящихся в загруженной вами папке media. Не забудьте загрузить их в свой скетч, перед тем как пробовать каждый пример.

Пример 7.4. Прозрачность в изображениях GIF



```
PImage img;

void setup() {
  size(480, 120);
  img = loadImage("clouds.gif");
}

void draw() {
  background(255);
  image(img, 0, 0);
  image(img, 0, mouseY * -1);
}
```

Пример 7.5. Прозрачность в изображениях PNG



```
PImage img;

void setup() {
  size(480, 120);
  img = loadImage("clouds.png");
}

void draw() {
  background(204);
  image(img, 0, 0);
  image(img, 0, mouseY * -1);
}
```

Не забывайте добавлять расширения файлов `.gif`, `.jpg` или `.png` при загрузке изображения. Кроме того, убедитесь, что имя изображения в скетче полностью соответствует названию файла, включая регистр букв. При необходимости еще раз прочитайте в этой главе о том, как сделать расширения файлов видимыми в Mac OS X и Windows.

7.2. Шрифты

Скетч Processing может отображать текст с использованием шрифтов TrueType (.ttf) и OpenType (.otf), а также пользовательского формата растрового изображения, называемого VLW. В этом разделе мы будем загружать шрифт TrueType из папки `data`. Это шрифт `SourceCodePro-Regular.ttf`, размещенный в папке `media`, которую вы скачали ранее.

Шрифты с открытыми лицензиями для использования в скетчах Processing можно найти на следующих веб-сайтах:

- Google Fonts (www.google.com/fonts);
- библиотека открытых шрифтов (https://fontlibrary.org/);
- лига переносимых шрифтов (https://www.theleagueofmoveabletype.com/).

Теперь можно загрузить шрифт и добавить к отображаемой информации слова. Это похоже на работу с изображениями, но есть один дополнительный шаг.

1. Добавьте шрифт в папку `data` скетча (инструкции приведены ранее в этой главе).
2. Создайте переменную `PFont` для хранения шрифта.
3. Создайте шрифт и присвойте его переменной с помощью `createFont()`. Эта процедура читает файл шрифта и создает его версию определенного размера, которая может быть использована скетчем Processing.
4. Используйте функцию `textFont()`, чтобы установить текущий шрифт.

Пример 7.6. Использование шрифтов

Теперь вы можете нарисовать эти буквы на экране с помощью функции `text()`, а также изменить размер с помощью `textSize()`:

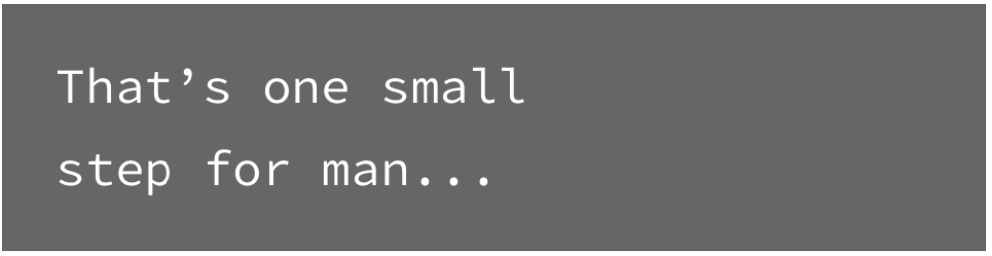

```
PFont font;

void setup() {
  size(480, 120);
  font = createFont("SourceCodePro-Regular.ttf", 32);
  textFont(font);
}

void draw() {
  background(102);
  textSize(32);
  text("That's one small step for man...", 25, 60);
  textSize(16);
  text("That's one small step for man...", 27, 90);
}
```

Первый параметр функции `text()` – это символы, отображаемые на экране. (Обратите внимание, что символы заключены в кавычки.) Второй и третий параметры задают горизонтальное и вертикальное положения. Положение отмеряют относительно *базовой линии* текста (рис. 7.1).

Рис. 7.1 ❖ Типографские координаты

Пример 7.7. Рисование текста в рамке

Вы также можете настроить рисование текста внутри поля, добавив четвертый и пятый параметры, которые определяют ширину и высоту поля:




```
PFont font;

void setup() {
  size(480, 120);
  font = createFont("SourceCodePro-Regular.ttf", 24);
  textFont(font);
}

void draw() {
  background(102);
  text("That's one small step for man...", 26, 24, 240, 100);
}
```

Пример 7.8. Сохранение текста в строке

В предыдущем примере длинное предложение внутри функции `text()` начало затруднять чтение кода. Мы можем сохранить текстовую строку в переменной, чтобы сделать код более модульным. Тип данных `String` (строка) используется для хранения текстовых данных. Вот новая версия предыдущего примера, в которой используется `String`:

```
PFont font;
String quote = "That's one small step for man...";
```

```
void setup() {
  size(480, 120);
  font = createFont("SourceCodePro-Regular.ttf", 24);
  textFont(font);
}

void draw() {
  background(102);
  text(quote, 26, 24, 240, 100);
}
```


В языке Processing есть набор дополнительных функций, которые влияют на отображение букв на экране. Они описаны с примерами в разделе **Typhography** (Типографика) справочного руководства Processing.

7.3. Векторные фигуры

Если вы создаете векторные фигуры в такой программе, как Inkscape или Illustrator, то можете напрямую загрузить их в Processing. Это полезно для создания сложных фигур, которые вы не хотите строить с помощью функций рисования Processing. Как и в случае с растровыми изображениями, вам необходимо добавить их в свой скетч, прежде чем они будут загружены.

Чтобы загрузить и отобразить файл SVG, нужно выполнить три шага.

1. Добавьте файл SVG в папку данных скетча.
2. Создайте переменную `PShape` для хранения векторного файла.
3. Загрузите векторный файл в переменную с помощью `loadShape()`.

Пример 7.9. Рисование готовыми фигурами

После выполнения этих шагов вы можете нарисовать изображение на экране с помощью функции `shape()`:

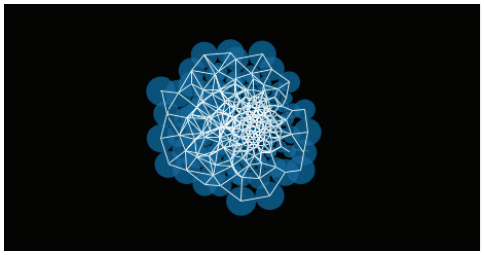

```
PShape network;

void setup() {
  size(480, 120);
  network = loadShape("network.svg");
}
```

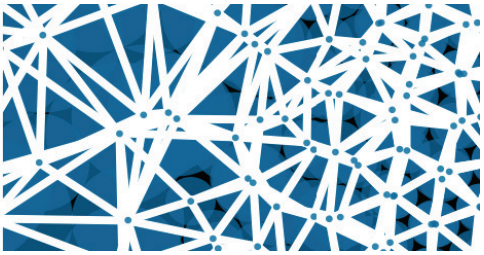
```
void draw() {
  background(0);
  shape(network, 30, 10);
  shape(network, 180, 10, 280, 280);
}
```

Параметры для `shape()` аналогичны `image()`. Первый параметр сообщает `shape()`, какой SVG-файл нужно отобразить, а следующая пара устанавливает позицию на экране. Необязательные четвертый и пятый параметры устанавливают ширину и высоту.

Пример 7.10. Масштабирование фигур

В отличие от растровых изображений, векторные фигуры можно масштабировать до любого размера без потери разрешения. В этом примере фигура масштабируется на основе переменной `mouseX`, а функция `shapeMode()` используется для рисования фигуры из ее центра, а не из позиции по умолчанию – верхнего левого угла:



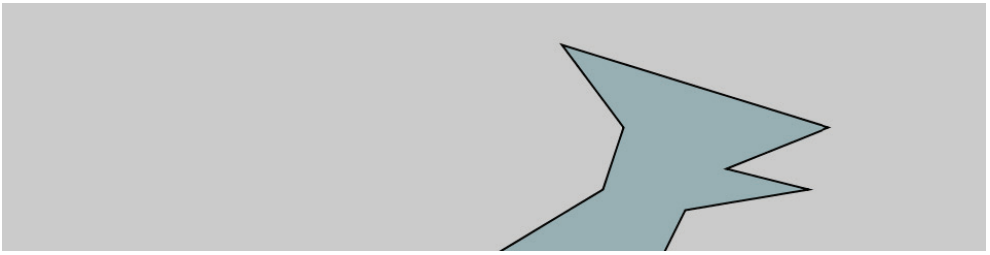


```
PShape network;

void setup() {
  size(240, 120);
  shapeMode(CENTER);
  network = loadShape("network.svg");
}

void draw() {
  background(0);
  float diameter = map(mouseX, 0, width, 10, 800);
  shape(network, 120, 60, diameter, diameter);
}
```

Processing поддерживает не все возможности формата SVG. См. раздел PShape в справочнике по языку Processing для получения дополнительных сведений.

Пример 7.11. Создание новой векторной фигуры

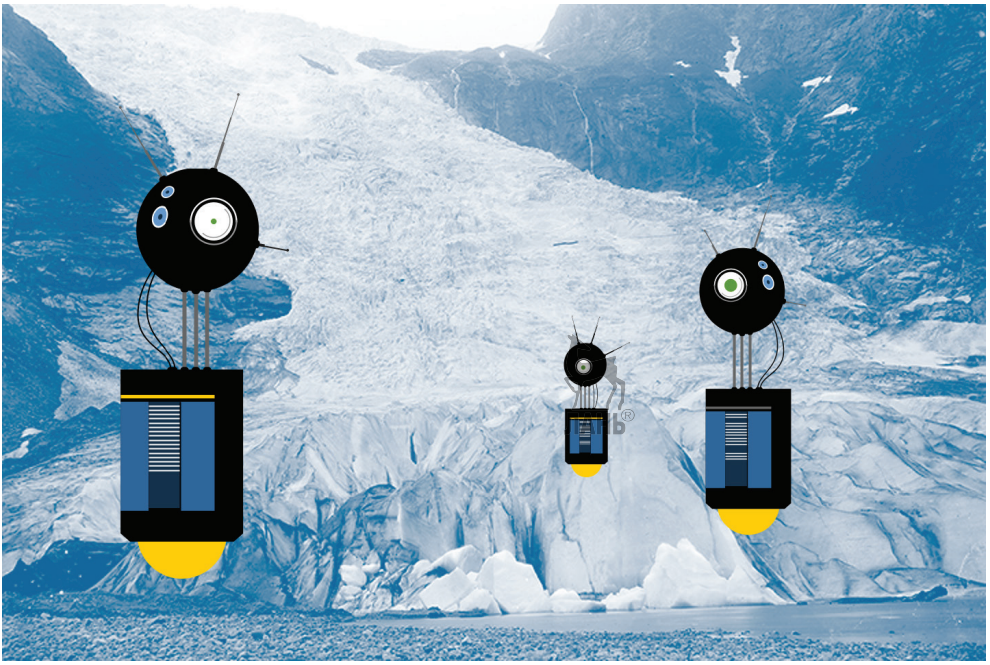
Помимо загрузки фигур через папку `data`, новые фигуры можно создавать с помощью кода с использованием функции `createShape()`. В следующем примере одно из фантастических существ из примера 3.21 определяется в функции `setup()`. Как только это сделано, фигуру можно будет использовать в любом месте программы с помощью функции `shape()`:



```
PShape dino;

void setup() {
  size(480, 120);
  dino = createShape();
  dino.beginShape();
  dino.fill(153, 176, 180);
  dino.vertex(50, 120);
  dino.vertex(100, 90);
  dino.vertex(110, 60);
  dino.vertex(80, 20);
  dino.vertex(210, 60);
  dino.vertex(160, 80);
  dino.vertex(200, 90);
  dino.vertex(140, 100);
  dino.vertex(130, 120);
  dino.endShape();
}

void draw() {
  background(204);
  translate(mouseX - 120, 0);
  shape(dino, 0, 0);
}
```

Создание пользовательской формы `PShape` с помощью `createShape()` может сделать скетчи более эффективными, когда одна и та же фигура рисуется много раз.

7.4. Робот 5: медиафайлы



В отличие от роботов, созданных из линий и прямоугольников, нарисованных средствами языка Processing в предыдущих главах, эти роботы были созданы с помощью программы векторного рисования. Чтобы нарисовать сложные изображения, часто бывает удобнее воспользоваться таким инструментом, как Inkscape или Illustrator, чем определять отдельные фигуры с координатами в коде.

И тот, и другой методы создания фигур обладают своими достоинствами и недостатками, поэтому выбор метода – это всегда компромисс. Когда фигуры определены в скетче, появляется больше возможностей изменять их во время работы программы. Если фигуры определены в другом месте, а затем загружены в Processing, изменения ограничиваются положением, углом и размером. При загрузке каждого робота из файла SVG, как показывает этот пример, варианты изменений, представленные в разделе 4.6, невозможны.

Изображения могут быть загружены в программу для добавления визуальных эффектов, созданных в других программах или снятых камерой. На этом фоновом изображении наши роботы исследуют формы жизни в Норвегии на заре XX века.

Файлы SVG и PNG, используемые в этом примере, можно загрузить по адресу http://www.processing.org/learning/books/media.zip:

```
PShape bot1;
PShape bot2;
PShape bot3;
PImage landscape;
```

```
float easing = 0.05;
float offset = 0;

void setup() {
  size(720, 480);
  bot1 = loadShape("robot1.svg");
  bot2 = loadShape("robot2.svg");
  bot3 = loadShape("robot3.svg");
  landscape = loadImage("alpine.png");
}

void draw() {
  // Размещение фонового изображения на заднем плане, это изображение
  // должно иметь высоту и ширину, идентичные указанным в программе
  background(landscape);

  // Задаем смещение влево/вправо и применяем отставание, чтобы
  // сделать перемещения более плавными
  float targetOffset = map(mouseY, 0, height, -40, 40);
  offset += (targetOffset - offset) * easing;

  // Рисуем левого робота
  shape(bot1, 85 + offset, 65);

  // Рисуем правого робота меньшего размера и задаем для него меньшее смещение
  float smallerOffset = offset * 0.7;
  shape(bot2, 510 + smallerOffset, 140, 78, 248);

  // Рисуем самого маленького робота и задаем для него наименьшее смещение
  smallerOffset *= -0.5;
  shape(bot3, 410 + smallerOffset, 225, 39, 124);
}
```

Глава 8

Движение

Как и в мультипликации, анимация на экране создается путем рисования первого изображения, потом рисования второго изображения, которое немного отличается от первого, затем еще одного и так далее. Иллюзия плавного движения создается за счет *инерции зрения*. Когда набор похожих изображений отображается с достаточно высокой скоростью, наш мозг воспринимает эти последовательные картинки как движущееся изображение.

8.1. Кадры

Чтобы создать плавное движение, Processing пытается запустить код внутри `draw()` с частотой 60 кадров в секунду. Кадр – это один проход через `draw()`, а частота кадров – это количество кадров, отрисовываемых каждую секунду. Следовательно, если программа формирует движущееся изображение с частотой 60 кадров в секунду, это означает, что программа выполняет весь код внутри `draw()` 60 раз в секунду.

Пример 8.1. Проверка частоты кадров

Чтобы проверить частоту кадров, запустите эту программу и посмотрите, как значения выводятся в консоль (переменная `frameRate` отслеживает скорость программы):

```
void draw () {
  println (frameRate);
}
```

Пример 8.2. Установка частоты кадров

Функция `frameRate()` изменяет скорость выполнения программы. Чтобы увидеть результат, раскомментируйте разные версии `frameRate()` в этом примере:

```
void setup() {
  frameRate(30);    // Тридцать кадров в секунду
  //frameRate(12);  // Двенадцать кадров в секунду
```

```
  //frameRate(2);   // Два кадра в секунду
  //frameRate(0.5); // Один кадр каждые две секунды
}

void draw() {
  println(frameRate);
}
```

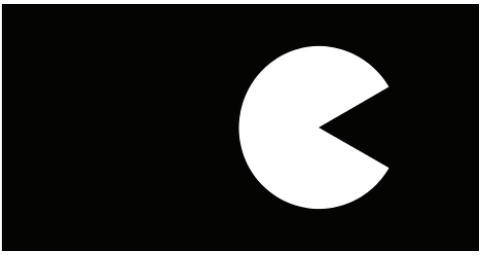
Processing пытается запускать код с частотой 60 кадров в секунду, но если для выполнения кода `draw()` требуется больше 1/60 секунды, то частота кадров будет уменьшаться. Функция `frameRate()` указывает только максимальную частоту кадров, а фактическая частота кадров для любой программы зависит от компьютера, на котором выполняется код.

8.2. Скорость и направление

Чтобы создать плавно движущиеся изображения, мы используем тип данных, называемый `float`. Переменные этого типа хранят числа с десятичными знаками после запятой, что обеспечивает большее разрешение при работе с движением. Например, при использовании `int` самое медленное перемещение – это один пиксель за раз (1, 2, 3, 4, ...), но с `float` вы можете двигаться так медленно, как хотите (`1.00`, `1.01`, `1.02`, `1.03`, ...).

Пример 8.3. Перемещение фигуры

В следующем примере фигура перемещается слева направо за счет обновления переменной `x`:



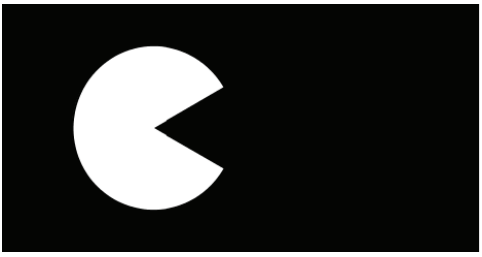
```
int radius = 40;
float x = -radius;
float speed = 0.5;

void setup() {
  size(240, 120);
  ellipseMode(RADIUS);
}

void draw() {
  background(0);
```

```
  x += speed; // Увеличение значения x
  arc(x, 60, radius, radius, 0.52, 5.76);
}
```

Когда вы запустите этот код, вы заметите, что фигура перемещается за правую часть экрана, когда значение переменной x становится больше ширины окна. Значение x продолжает увеличиваться, но фигура больше не видна.

Пример 8.4. Замкнутое движение

Существует много вариантов движения фигуры по экрану, которые вы можете выбрать по своему усмотрению. Во-первых, мы доработаем код, чтобы продемонстрировать, как переместить фигуру обратно к левому краю экрана, после того как она исчезнет с правого. В этом случае представьте себе экран в виде цилиндра с фигурой, которая движется по его поверхности и возвращается в исходную точку:



```
int radius = 40;
float x = -radius;
float speed = 0.5;

void setup() {
  size(240, 120);
  ellipseMode(RADIUS);
}

void draw() {
  background(0);
  x += speed;              // Увеличение значения x
  if (x > width+radius) { // Если фигура покинула экран,
    x = -radius;           // возвращаем ее к левому краю
  }
  arc(x, 60, radius, radius, 0.52, 5.76);
}
```

При каждом запуске блока `draw()` код проверяет, не вышло ли значение x за пределы ширины экрана (плюс радиус фигуры). Если это так, мы присваиваем переменной x отрицательное значение, чтобы при ее увеличении фигура появлялась на экране слева. На рис. 8.1 показано, как это работает.

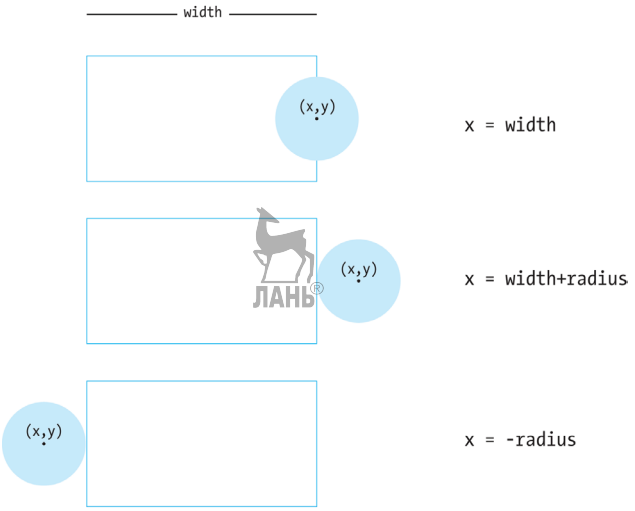
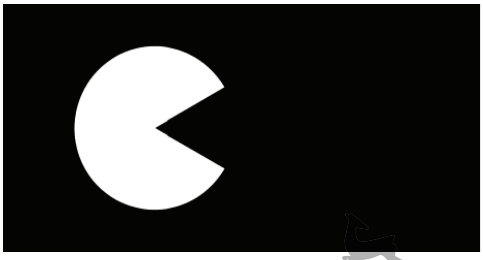
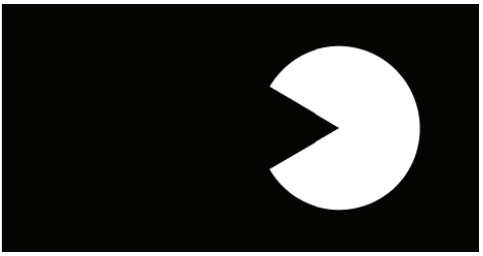


Рис. 8.1 ❖ Проверка на выход за край окна

Пример 8.5. Отскок от стены

В этом примере мы переделаем код примера 8.3, чтобы изменить направление движения фигуры при касании края экрана, а не зацикливать движение по кругу. Чтобы это произошло, мы добавляем новую переменную для хранения направления движения. Значение 1 означает движение вправо, а -1 – влево:





```
int radius = 40;
float x = 110;
float speed = 0.5;
int direction = 1;

void setup() {
  size(240, 120);
  ellipseMode(RADIUS);
}

void draw() {
  background(0);
```

```
  x += speed * direction;
  if ((x > width-radius) || (x < radius)) {
    direction = -direction; // Смена направления
  }
  if (direction == 1) {
    arc(x, 60, radius, radius, 0.52, 5.76); // Движение вправо
  } else {
    arc(x, 60, radius, radius, 3.67, 8.9); // Движение влево
  }
}
```

Когда фигура достигает края, этот код меняет направление движения, изменяя знак переменной `direction`. Например, если переменная `direction` положительна, когда фигура достигает края, код меняет значение на отрицательное.

8.3. Анимация

Иногда вам нужно анимировать фигуру, чтобы она переходила из одной точки на экране в другую. С помощью нескольких строк кода вы можете установить начальную и конечную позиции, а затем вычислить промежуточные позиции для каждого кадра.

Пример 8.6. Расчет позиций анимации

Чтобы сделать этот пример кода модульным, мы создали группу переменных. Выполните код несколько раз, меняя значения переменных и наблюдая, как этот код перемещает фигуру из одного места в другое с разной скоростью. На скорость движения влияет значение переменной `step` (шаг):

```
int startX = 20;    // Начальная координата x
int stopX = 160;    // Конечная координата x
int startY = 30;    // Начальная координата y
int stopY = 80;     // Конечная координата y
float x = startX;   // Текущая координата x
float y = startY;   // Текущая координата y
float step = 0,005; // Размер каждого шага (от 0,0 до 1,0)
float pct = 0,0;    // Процент пройденного пути (от 0,0 до 1,0)
```

```
void setup() {
  size(240, 120);
}

void draw() {
  background(0);
  if (pct < 1.0) {
    x = startX + ((stopX-startX) * pct);
    y = startY + ((stopY-startY) * pct);
    pct += step;
  }
  ellipse(x, y, 20, 20);
}
```


8.4. Случайное движение

В отличие от плавного линейного движения, характерного для компьютерной графики, движение в физическом мире обычно происходит более хаотично и непредсказуемо. Например, представьте себе лист, плывущий по земле, или муравья, ползущего по пересеченной местности. Мы можем моделировать непредсказуемое поведение реальных объектов, генерируя случайные числа. Функция `random()` вычисляет случайные значения; мы можем установить диапазон, внутри которого располагаются случайные значения.

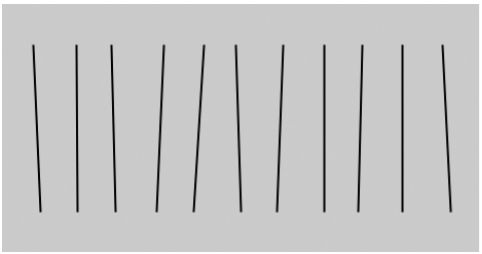
Пример 8.7. Генерация случайных значений

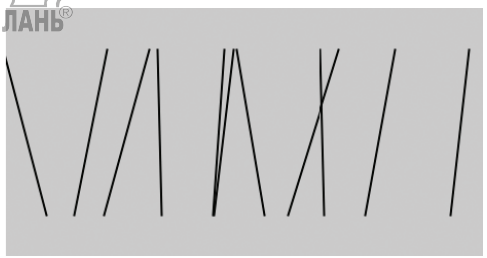
В следующем коротком примере на консоль выводятся случайные значения, диапазон которых ограничен положением мыши. Функция `random()` всегда возвращает значение с плавающей запятой, поэтому убедитесь, что переменная слева от оператора присваивания (=) относится к типу `float`, как здесь:

```
void draw () {
  float r = random (0, mouseX);
  println (r);
}
```

Пример 8.8. Рисование случайных объектов

Следующий пример является продолжением предыдущего; он использует значения из функции `random()` для изменения положения линий на экране. Когда курсор находится в левой части экрана, изменение невелико; по мере того как он перемещается вправо, значения функции `random()` увеличиваются, и линии колеблются все сильнее. Поскольку функция `random()` находится внутри цикла `for`, новое случайное значение вычисляется для каждой начальной и конечной точек каждой линии:

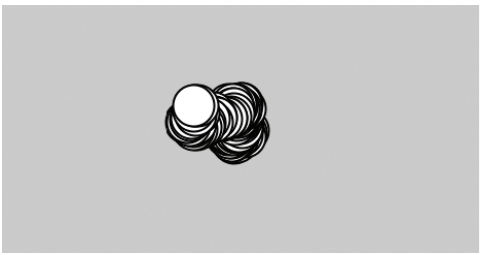




```
void setup() {
  size(240, 120);
}

void draw() {
  background(204);
  for (int x = 20; x < width; x += 20) {
    float mx = mouseX / 10;
    float offsetA = random(-mx, mx);
    float offsetB = random(-mx, mx);
    line(x + offsetA, 20, x - offsetB, 100);
  }
}
```

Пример 8.9. Произвольное перемещение фигур

Наложение случайных значений на координаты движущегося объекта помогает создавать изображения, которые выглядят более естественно. В следующем примере положение круга изменяется на случайную величину при каждом выполнении блока `draw()`. Поскольку функция `background()` не используется, предыдущие изображения сохраняются:



```
float speed = 2.5;
int diameter = 20;
float x;
float y;

void setup() {
  size(240, 120);
  x = width/2;
```

```
  y = height/2;
}

void draw() {
  x += random(-speed, speed);
  y += random(-speed, speed);
  ellipse(x, y, diameter, diameter);
}
```

Если вы будете наблюдать за работой этого кода достаточно долго, вы можете увидеть, как круг покидает окно и возвращается. Это поведение оставлено на волю случая, но мы могли бы добавить несколько структур `if` или использовать функцию `constrain()`, чтобы круг не покидал экран. Функция `constrain()` ограничивает значение определенным диапазоном, который можно использовать для удержания `x` и `y` в границах окна отображения. Заменив блок `draw()` в предыдущем коде на показанный ниже, вы убедитесь, что круг больше не покидает экран:


```
void draw() {
  x += random(-speed, speed);
  y += random(-speed, speed);
  x = constrain(x, 0, width);
  y = constrain(y, 0, height);
  ellipse(x, y, diameter, diameter);
}
```

Чтобы заставить `random()` выдавать одну и ту же последовательность чисел при каждом запуске программы, можно использовать функцию `randomSeed()`. Более подробно об этом можно прочитать в справочнике по языку Processing.

8.5. Таймеры

Каждая программа на языке Processing отсчитывает время, прошедшее с момента ее запуска. Оно считается в миллисекундах (тысячных долях секунды), поэтому через 1 секунду счетчик показывает 1000, через 5 секунд – 5000, а через 1 минуту – 60 000. Мы можем использовать этот счетчик для запуска анимации в заданное время. Функция `millis()` возвращает текущее значение счетчика.

Пример 8.10. Течение времени

Вы можете наблюдать за течением времени, запустив эту программу:

```
void draw() {
  int timer = millis();
  println(timer);
}
```

Пример 8.11. Запуск событий по времени

В сочетании с блоком if значения из millis() можно использовать для формирования последовательности анимации и событий в программе. Например, по прошествии двух секунд код внутри блока if может инициировать заданное изменение. В этом примере переменные с именами time1 и time2 определяют, когда следует изменить значение переменной x:

```
int time1 = 2000;
int time2 = 4000;
float x = 0;

void setup() {
  size(480, 120);
}


void draw() {
  int currentTime = millis();
  background(204);
  if (currentTime > time2) {
    x -= 0.5;
  } else if (currentTime > time1) {
    x += 2;
  }
  ellipse(x, 60, 90, 90);
}
```


8.6. Круговое движение

Если вы являетесь знатоком тригонометрии, вам уже известно, насколько удивительны функции синуса и косинуса. Если нет, мы надеемся, что следующие примеры вызовут у вас интерес. Мы не будем здесь подробно обсуждать математику, но покажем несколько примеров создания плавного движения.

На рис. 8.2 наглядно показано соотношение значений синусоидальной волны с величиной угла. Обратите внимание на то, как в верхней и нижней частях волны скорость перемещения по вертикальной оси замедляется, останавливается, а затем меняет направление. Именно это свойство синусоидальной кривой создает интересные эффекты.

Функции sin() и cos() в языке Processing возвращают значения от –1 до 1 для синуса или косинуса заданного угла. Как и для функции arc(), углы должны быть указаны в радианах (вернитесь к примерам 3.7 и 3.8, чтобы вспомнить, что такое радианы). Чтобы использовать для рисования значения, возвращаемые функциями sin() и cos(), их обычно умножают на относительно большое число.

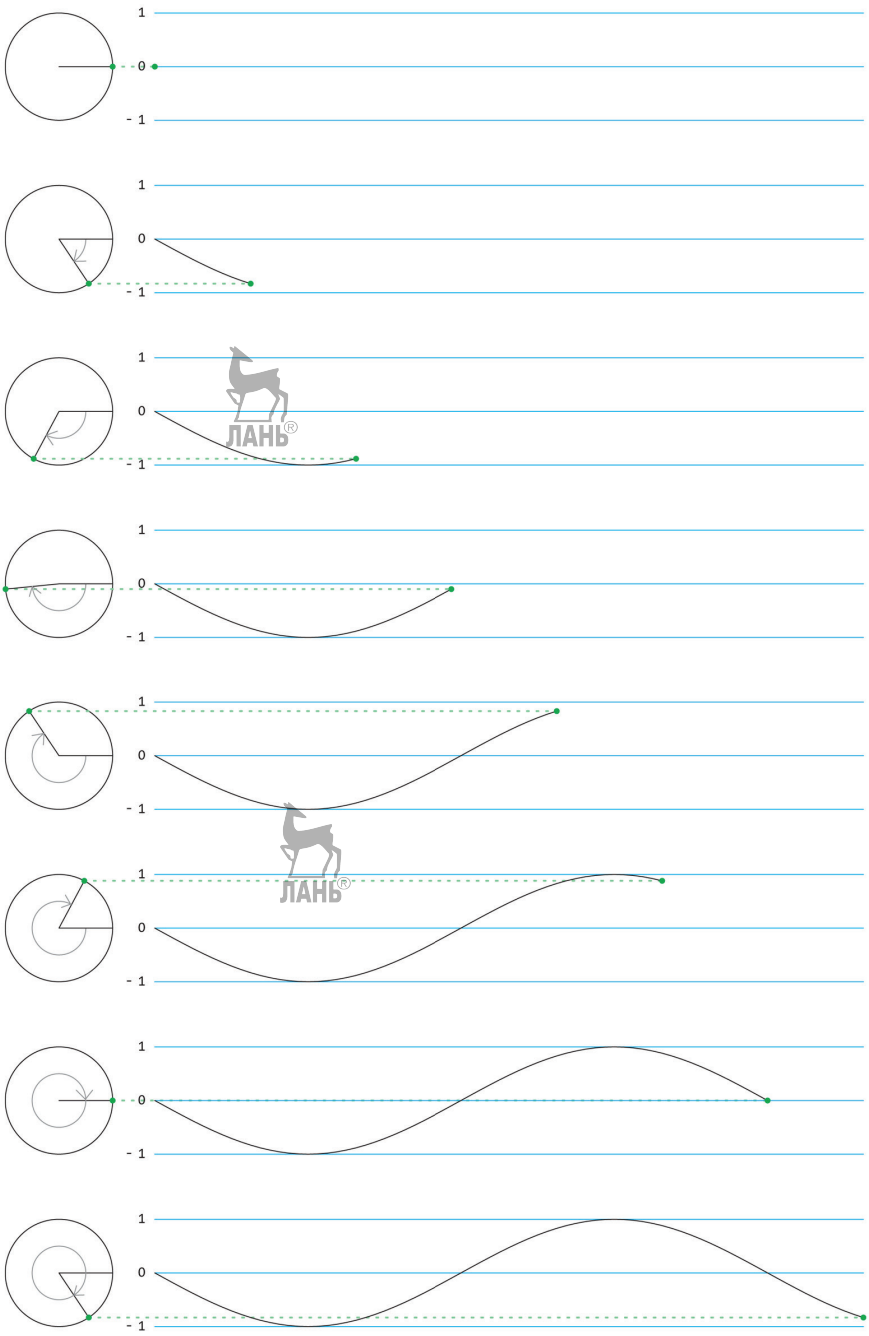


Рис. 8.2 ❖ Синусоидальная волна создается путем отслеживания значений синуса угла, образованного точкой, которая движется по окружности

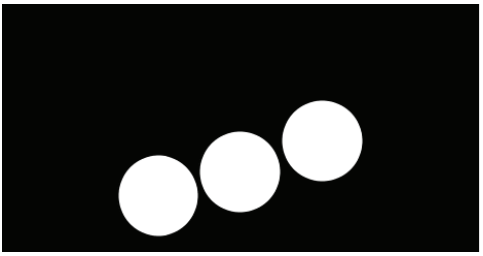
Пример 8.12. Значения синусоидальной функции

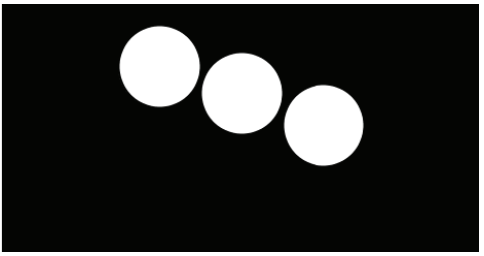
В этом примере показано, как значения функции `sin()` меняются от –1 до 1 по мере увеличения угла. С помощью функции `map()` переменная `sinval` переносится из этого диапазона в новый диапазон от 0 до 255. Это новое значение используется для установки яркости фона окна:

```
float angle = 0.0;

void draw() {
  float sinval = sin(angle);
  println(sinval);
  float gray = map(sinval, -1, 1, 0, 255);
  background(gray);
  angle += 0.1;
}
```

Пример 8.13. Движение синусоидальной волны

В этом примере показано, как значения `sin()` можно преобразовать в движение:

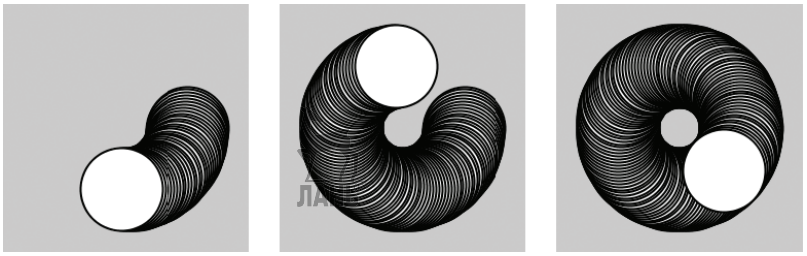




```
float angle = 0.0;
float offset = 60;
float scalar = 40;
float speed = 0.05;

void setup() {
  size(240, 120);
}

void draw() {
  background(0);
  float y1 = offset + sin(angle) * scalar;
  float y2 = offset + sin(angle + 0.4) * scalar;
  float y3 = offset + sin(angle + 0.8) * scalar;
  ellipse( 80, y1, 40, 40);
  ellipse(120, y2, 40, 40);
  ellipse(160, y3, 40, 40);
  angle += speed;
}
```

Пример 8.14. Круговое движение

Когда `sin()` и `cos()` используются вместе, их значения могут описывать круговое движение. Значения `cos()` представляют собой координаты `x`, а значения `sin()` – координаты `y`. Оба значения умножаются на переменную с именем `scalar`, которая задает радиус движения, и суммируются со значением переменной `offset`, от которой зависит расположение центра кругового движения:



```
float angle = 0.0;
float offset = 60;
float scalar = 30;
float speed = 0.05;

void setup() {
  size(120, 120);
}

void draw() {
  float x = offset + cos(angle) * scalar;
  float y = offset + sin(angle) * scalar;
  ellipse( x, y, 40, 40);
  angle += speed;
}
```

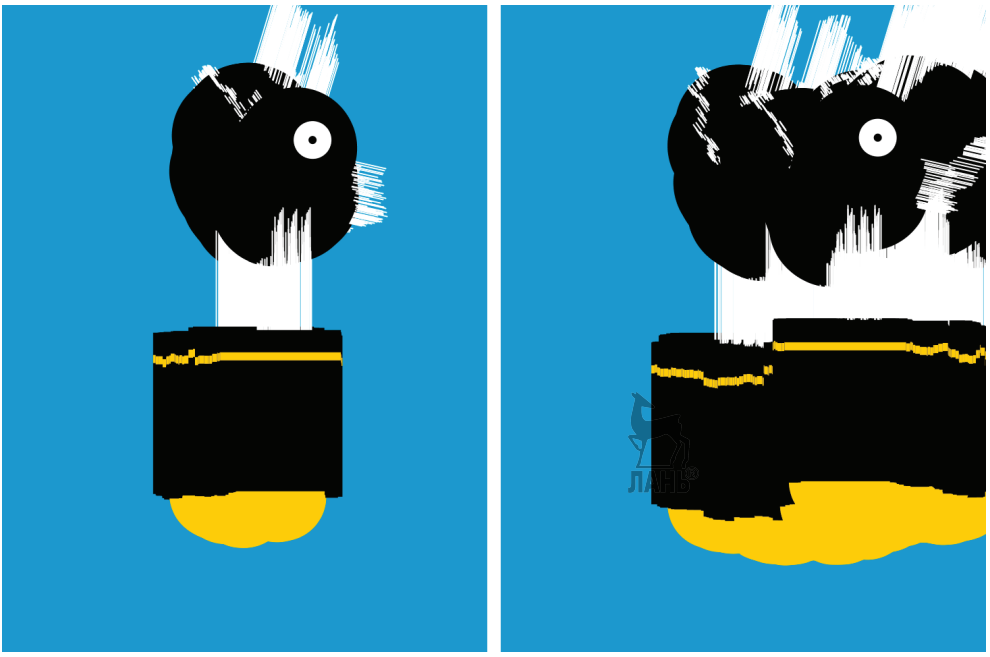
Пример 8.15. Движение по спирали

Небольшое изменение, связанное с увеличением значения переменной `scalar` в каждом кадре, дает спираль, а не круг:

```
float angle = 0.0;
float offset = 60;
```

```
float scalar = 2;
float speed = 0.05;

void setup() {
  size(120, 120);
  fill(0);
}

void draw() {
  float x = offset + cos(angle) * scalar;
  float y = offset + sin(angle) * scalar;
  ellipse( x, y, 2, 2);
  angle += speed;
  scalar += speed;
}
```


8.7. Робот 6: движение



В этом примере к роботу применяются методы случайного и кругового движений. Функция `background()` в данном примере отсутствует, чтобы было легче увидеть, как меняется положение и тело робота.

В каждом кадре случайное число от –4 до 4 добавляется к координате `x`, а случайное число от –1 до 1 добавляется к координате `y`. Это заставляет робота двигаться преимущественно слева направо, чем сверху вниз. Числа, полученные с помощью функции `sin()`, изменяют высоту шеи, поэтому ее высота колеблется от 50 до 110 пикселей:

```
float x = 180;           // Координата X
float y = 400;           // Координата Y
float bodyHeight = 153;  // Высота туловища
float neckHeight = 56;   // Длина шеи
float radius = 45;       // Радиус головы
float angle = 0.0;       // Угол для функции sin()

void setup() {
  size(360, 480);
  ellipseMode(RADIUS);
  background(0, 153, 204);  // Голубой фон
}

void draw() {
  // Изменение позиции на небольшое случайное значение
  x += random(-4, 4);
  y += random(-1, 1);

  // Изменение длины шеи
  neckHeight = 80 + sin(angle) * 30;
  angle += 0.05;

  // Вычисление высоты головы
  float ny = y - bodyHeight - neckHeight - radius;

  // Шея
  stroke(255);
  line(x+2, y-bodyHeight, x+2, ny);
  line(x+12, y-bodyHeight, x+12, ny);
  line(x+22, y-bodyHeight, x+22, ny);

  // Антенны
  line(x+12, ny, x-18, ny-43);
  line(x+12, ny, x+42, ny-99);
  line(x+12, ny, x+78, ny+15);

  // Туловище
  noStroke();
  fill(255, 204, 0);
  ellipse(x, y-33, 33, 33);
  fill(0);
  rect(x-45, y-bodyHeight, 90, bodyHeight-33);
  fill(255, 204, 0);
  rect(x-45, y-bodyHeight+17, 90, 6);

  // Голова
  fill(0);
  ellipse(x+12, ny, radius, radius);
  fill(255);
  ellipse(x+24, ny-6, 14, 14);
  fill(0);
  ellipse(x+24, ny-6, 3, 3);
}
```

Глава 9

Функции

Функции являются основными строительными блоками программ на языке Processing. Они присутствуют в каждом представленном нами примере. Например, мы часто использовали функции `size()`, `line()` и `fill()`. В этой главе показано, как писать новые функции, чтобы расширить возможности языка за пределы его встроенных функций.

Сила функций заключается в их модульности. Функции – это независимые программные блоки, которые используются для создания более сложных программ как кубики LEGO, где каждый тип кубиков служит определенной цели, а создание сложной модели требует совместного использования различных частей. Как и в случае с функциями, ценность этих кубиков заключается в возможности создавать множество различных форм из одного и того же набора элементов. Та же группа деталей LEGO, из которых собран космический корабль, может быть повторно использована для создания грузовика, небоскреба и многих других объектов.

Функции полезны, если вы собираетесь неоднократно рисовать сложную фигуру, например дерево. Функция рисования дерева будет состоять из встроенных функций Processing, таких как `line()`, которые создают фигуру. После того как код для рисования дерева написан и отлажен, вам не нужно больше думать о тонкостях рисования дерева – вы можете просто вызвать в коде функцию `tree()` (или как там вы назвали ее), чтобы нарисовать замысловатую фигуру. Функции позволяют абстрагировать сложную последовательность операторов, поэтому вы можете сосредоточиться на цели более высокого уровня (например, расположение деревьев), а не на деталях реализации (функции `line()`, которые определяют форму дерева). После определения функции код внутри функции не нужно повторять снова.

9.1. Основы работы с функциями

Компьютер выполняет программу на языке Processing по одной строке за раз. Когда вы вызываете функцию, компьютер переходит к тому месту, где она определена, и выполняет там код функции, а затем возвращается к тому месту, где он остановился.

Пример 9.1. Бросание игровых кубиков

Поведение компьютера иллюстрирует функция `rollDice()`, написанная для этого примера. После запуска программы выполняется код в блоке `setup()`, а затем выполнение завершается. Программа отклоняется от последовательного выполнения и запускает код внутри функции `rollDice()` каждый раз, когда встречает упоминание о ней:


```
void setup() {
  println("Готов к броску!");
  rollDice(20);
  rollDice(20);
  rollDice(6);
  println("Завершено.");
}

void rollDice(int numSides) {
  int d = 1 + int(random(numSides));
  println("Бросок... " + d);
}
```

Две строки кода в функции `rollDice()` выбирают случайное число от 1 до количества сторон на кубике и выводят это число на консоль. Поскольку числа случайны, вы будете видеть разные числа при каждом запуске программы:

```
Готов к броску!
Бросок ... 20
Бросок ... 11
Бросок ... 1
Завершено.
```


Каждый раз, когда внутри блока `setup()` встречается вызов функции `rollDice()`, код внутри функции выполняется сверху вниз, затем выполнение программы переходит к следующей строке в `setup()`.

Функция `random()` возвращает число от 0 до указанного числа (но не включая его). Итак, `random(6)` возвращает число от 0 до 5,99999... Поскольку `random()` возвращает значение с плавающей запятой, мы также используем функцию `int()` для преобразования его в целое число. Поэтому сочетание функций `int(random(6))` вернет 0, 1, 2, 3, 4 или 5. Затем мы прибавляем к полученному значению 1, чтобы возвращаемое число было между 1 и 6 (как у обычной игральной кости). Как и во многих других случаях в этой книге, отсчет от 0 упрощает использование результатов `random()` с другими вычислениями.

Пример 9.2. Другой способ бросить кубик

Если бы эквивалентная программа была написана без функции `rollDice()`, это могло бы выглядеть так:

```
void setup() {
  println("Готов к броску!");
```

```
  int d1 = 1 + int(random(20));
  println("Бросок... " + d1);
  int d2 = 1 + int(random(20));
  println("Бросок... " + d2);
  int d3 = 1 + int(random(6));
  println("Бросок... " + d3);
  println("Завершено.");
}
```

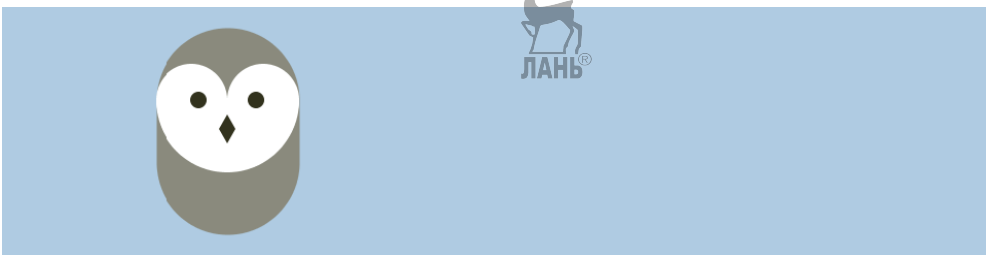
Функция `rollDice()` в примере 9.1 упрощает чтение и сопровождение кода. Такая программа более понятна, потому что в названии функции четко указано ее назначение. В этом примере мы тоже видим функцию `random()` в блоке `setup()`, но ее использование не так очевидно. Количество сторон на кубике также становится яснее благодаря использованию функции: когда в коде написано `rollDice(6)`, очевидно, что он имитирует бросок шестигранного кубика. Кроме того, код из примера 9.1 легче поддерживать и модифицировать, потому что информация не повторяется. В данном коде текст «Бросок...» повторяется трижды. Если вы хотите изменить этот текст на что-то другое, вам нужно будет обновить программу в трех местах, а не делать одно редактирование внутри функции `rollDice()`. Кроме того, как вы увидите в примере 9.5, функция также может сделать программу намного короче (и, следовательно, ее легче поддерживать и читать), что помогает снизить потенциальное количество ошибок.

9.2. Создание функции

В этом разделе мы нарисуем сову, чтобы на примере показать этапы создания функции.

Пример 9.3. Рисование совы

Сначала нарисуем сову без использования функции:



```
void setup() {
  size(480, 120);
}
```

```
void draw() {
  background(176, 204, 226);
  translate(110, 110);
  stroke(138, 138, 125);
  strokeWeight(70);
  line(0, -35, 0, -65); // Туловище
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35); // Левый обвод глаза
  ellipse(17.5, -65, 35, 35);  // Правый обвод глаза
  arc(0, -65, 70, 70, 0, PI);  // Нижний обвод
  fill(51, 51, 30);
  ellipse(-14, -65, 8, 8); // Левый глаз
  ellipse(14, -65, 8, 8);  // Правый глаз
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Клюв
}
```

Обратите внимание, что сначала мы используем функцию `translate()` для перемещения начала координат (0,0) на 110 пикселей и на 110 пикселей вниз. Затем сова рисуется относительно точки (0,0), причем координаты ее элементов как положительные, так и отрицательные, поскольку они сосредоточены вокруг новой точки (0,0), как показано на рис. 9.1.

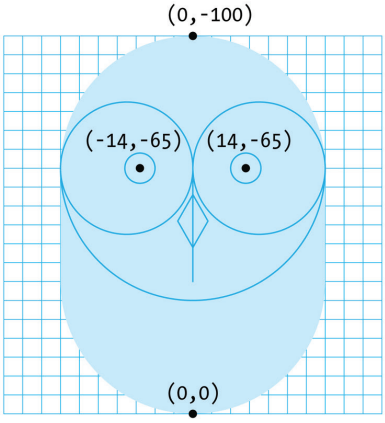


Рис. 9.1 ❖ Координаты элементов изображения совы

Пример 9.4. Компания из двух сов

Код, представленный в примере 9.3, является разумным, если нам нужна только одна сова, но когда мы рисуем вторую, длина кода почти удваивается:

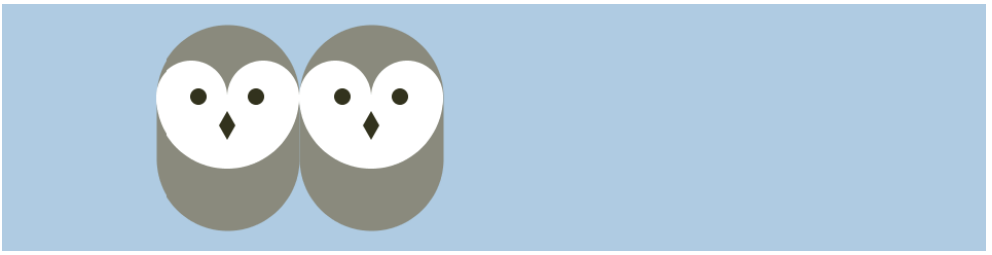
```
void setup() {
  size(480, 120);
}


void draw() {
  background(176, 204, 226);

  // Левая сова
  translate(110, 110);
  stroke(138, 138, 125);
  strokeWeight(70);
  line(0, -35, 0, -65); // Туловище
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35); // Левый обвод глаза
  ellipse(17.5, -65, 35, 35);  // Правый обвод глаза
  arc(0, -65, 70, 70, 0, PI);  // Нижний обвод
  fill(51, 51, 30);
  ellipse(-14, -65, 8, 8); // Левый глаз
  ellipse(14, -65, 8, 8);  // Правый глаз
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Клюв

  // Правая сова
  translate(70, 0);
  stroke(138, 138, 125);
  strokeWeight(70);
  line(0, -35, 0, -65); // Туловище
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35); // Левый обвод глаза
  ellipse(17.5, -65, 35, 35);  // Правый обвод глаза
  arc(0, -65, 70, 70, 0, PI);  // Нижний обвод
  fill(51, 51, 30);
  ellipse(-14, -65, 8, 8); // Левый глаз
  ellipse(14, -65, 8, 8);  // Правый глаз
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Клюв
}
```

Программа выросла с 21 строки до 34: код для рисования первой совы был скопирован и вставлен в программу; также был вставлен вызов функции `translate()`, чтобы переместить новую сову на 70 пикселей вправо. Это утомительный и неэффективный способ нарисовать вторую сову, не говоря

уже о головной боли, связанной с добавлением третьей совы этим методом. Но в дублировании кода нет необходимости, потому что это ситуация, когда нам поможет функция.

Пример 9.5. Функция рисования совы

В этом примере представлена функция для рисования двух сов с одним и тем же кодом. Если мы оформим код рисования совы в отдельную функцию, он должен появиться в программе только один раз:



```
void setup() {
  size(480, 120);
}

void draw() {
  background(176, 204, 226);
  owl(110, 110);
  owl(180, 110);
}

void owl(int x, int y) {
  pushMatrix();
  translate(x, y);
  stroke(138, 138, 125);
  strokeWeight(70);
  line(0, -35, 0, -65); // Туловище
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35); // Левый обвод глаза
  ellipse(17.5, -65, 35, 35);  // Правый обвод глаза
  arc(0, -65, 70, 70, 0, PI);  // Нижний обвод
  fill(51, 51, 30);
  ellipse(-14, -65, 8, 8); // Левый глаз
  ellipse(14, -65, 8, 8);  // Правый глаз
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Клюв
  popMatrix();
}
```

Из иллюстраций видно, что этот пример и пример 9.4 дают одинаковый результат, но этот пример короче, потому что код для рисования совы появляется только один раз внутри функции, логично названной `owl()`, т. е. «сова».

Этот код выполняется дважды, потому что он дважды вызывается внутри `draw()`. Сова нарисована в двух разных местах из-за переданных в функцию параметров, которые определяют координаты `x` и `y`.

Параметры (или аргументы) – важная часть функций, поскольку они обеспечивают гибкость. Еще один пример мы видели в функции `rollDice()`; единственный параметр с именем `numSides` позволил имитировать 6-гранную игральную кость, 20-гранную кость или кость с любым количеством сторон. Аналогично устроены и другие функции в языке Processing. Например, параметры функции `line()` позволяют провести линию от любого пикселя на экране к любому другому пикселю. Без параметров функция могла бы провести линию только от одной фиксированной точки к другой.

У каждого параметра есть тип данных (например, `int` или `float`), потому что каждый параметр – это переменная, которая создается при очередном запуске функции. Когда вы запускаете пример 9.5, при первом вызове функции `owl()` значение параметра `x` равно 110, и значение `y` также равно 110. При втором использовании функции значение `x` равно 180, а значение `y` снова равно 110. Каждое значение передается в функцию, а затем везде, где имя переменной появляется в функции, оно заменяется входящим значением.

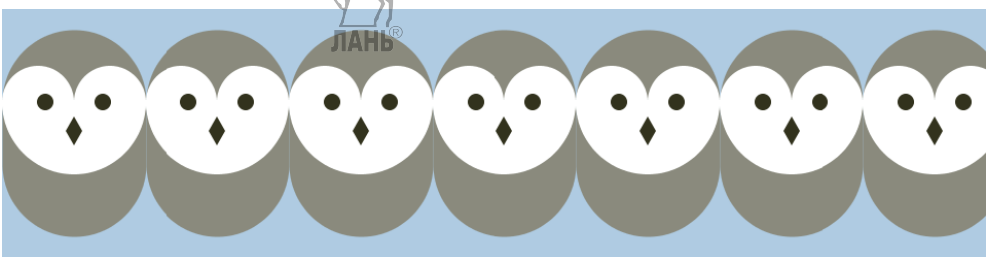
Убедитесь, что значения, переданные в функцию, соответствуют типам данных параметров. Например, если в этом примере внутри блока `setup()` встретится следующий вызов функции:

```
owl(110.5, 120.2);
```

это приведет к ошибке, поскольку параметры `x` и `y` относятся к типу `int`, а значения `110.5` и `120.2` являются значениями с плавающей запятой.

Пример 9.6. Увеличение популяции сов

Теперь, когда у нас есть базовая функция для рисования совы в произвольном месте экрана, мы можем эффективно рисовать большое количество сов, помещая функцию в цикл `for` и изменяя первый параметр при каждом прохождении цикла:



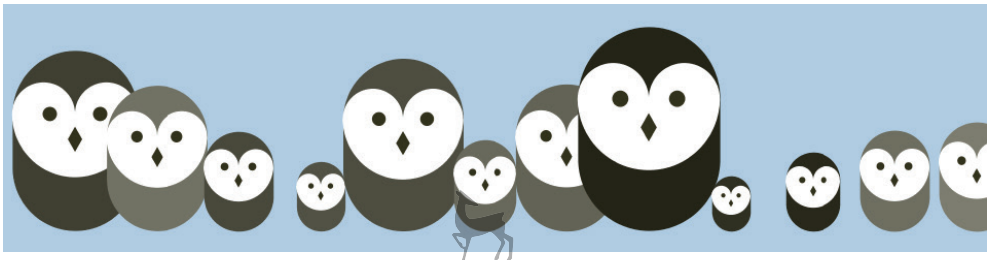
```
void setup() {
  size(480, 120);
}
```

```
void draw() {
  background(176, 204, 226);
  for (int x = 35; x < width + 70; x += 70) {
    owl(x, 110);
  }
}

// Здесь вставьте функцию owl() из примера 9.5
```

В функцию можно добавить много разных параметров, чтобы изменять различные аспекты рисования совы. Например, можно передать значения, которые определяют цвет совы, ее поворот, масштаб или диаметр ее глаз.

Пример 9.7. Совы разного размера

В этом примере мы добавили два параметра, от которых зависят яркость серого цвета и размер каждой совы:




```
void setup() {
  size(480, 120);
}

void draw() {
  background(176, 204, 226);
  randomSeed(0);
  for (int i = 35; i < width + 40; i += 40) {
    int gray = int(random(0, 102));
    float scalar = random(0.25, 1.0);
    owl(i, 110, gray, scalar);
  }
}

void owl(int x, int y, int g, float s) {
  pushMatrix();
  translate(x, y);
  scale(s); // Задаем размер
  stroke(138-g, 138-g, 125-g); // Задаем значение цвета
  strokeWeight(70);
  line(0, -35, 0, -65); // Туловище
  noStroke();
  fill(255);
```

```
  ellipse(-17.5, -65, 35, 35); // Левый обвод глаза
  ellipse(17.5, -65, 35, 35);  // Правый обвод глаза
  arc(0, -65, 70, 70, 0, PI);  // Нижний обвод
  fill(51, 51, 30);
  ellipse(-14, -65, 8, 8); // Левый глаз
  ellipse(14, -65, 8, 8);  // Правый глаз
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Клюв
  popMatrix();
}
```

9.3. Возвращаемые значения

Функции могут производить вычисления, а затем возвращать результат в основную программу. Мы уже использовали функции подобного типа, включая `random()` и `sin()`. Обратите внимание, что когда применяется этот тип функции, возвращаемое значение обычно присваивается переменной:

```
float r = random(1, 10);
```

В этом случае `random()` возвращает значение от 1 до 10, которое затем присваивается переменной `r`.

Функция, возвращающая значение, также часто используется в качестве параметра для другой функции. Например:

```
point(random(width), random(height));
```

В этом случае значения из `random()` не присваиваются переменной – они сразу передаются как параметры в `point()` и используются для позиционирования точки в окне.

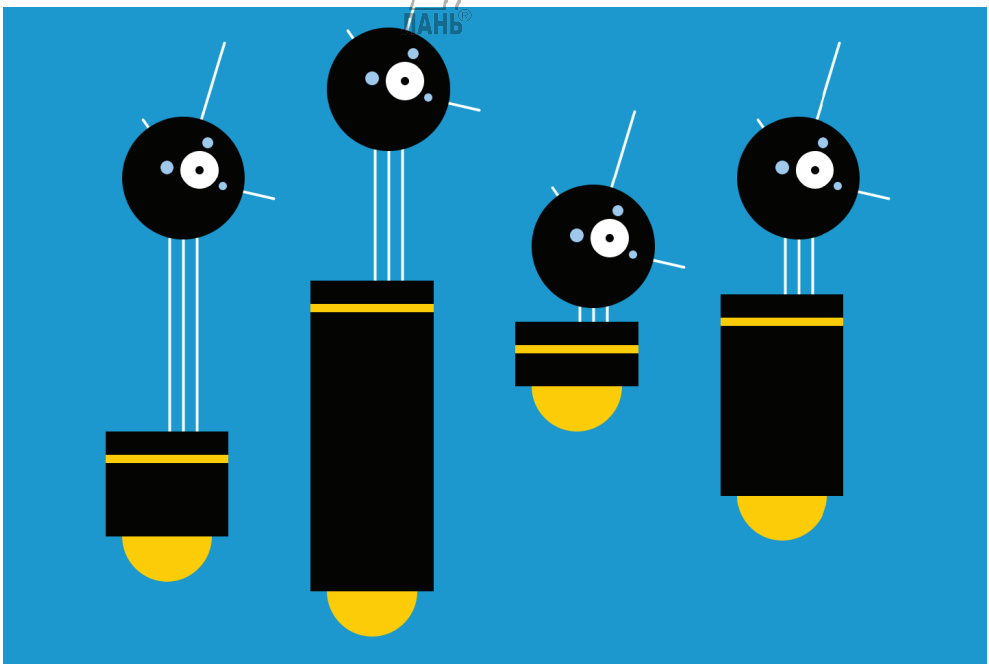
Пример 9.8. Вычисления в функции

Чтобы создать функцию, возвращающую значение, замените в объявлении функции ключевое слово `void` типом данных, который должна возвращать функция. В вашей функции укажите данные, которые нужно передать обратно, с помощью ключевого слова `return`. Следующий код содержит функцию с именем `calculateMars()`, которая вычисляет вес человека или объекта на Марсе:

```
void setup() {
  float yourWeight = 132;
  float marsWeight = calculateMars(yourWeight);
  println(marsWeight);
}

float calculateMars(float w) {
  float newWeight = w * 0.38;
  return newWeight;
}
```

Обратите внимание, что перед именем функции стоит указание на тип данных `float`, чтобы показать, что она возвращает значение с плавающей запятой. Последняя строка кода функции возвращает значение переменной `newWeight`. Во второй строке блока `setup()` это значение присваивается переменной `marsWeight`. (Чтобы узнать свой собственный вес на Марсе, измените значение переменной `yourWeight` на свой вес.)

9.4. Робот 7: функции



В отличие от робота 2 (см. раздел 4.6), в этом примере используется функция для рисования четырех вариантов робота в одной программе. Поскольку функция `drawRobot()` вызывается в блоке `draw()` четыре раза, код этой функции запускается четырежды, каждый раз с другим набором параметров для изменения положения и высоты тела робота.

Обратите внимание, что глобальные переменные робота 2 теперь изолированы в функции `drawRobot()`. Поскольку эти переменные применяются только для рисования робота, они находятся внутри фигурных скобок, определяющих функциональный блок `drawRobot()`. Поскольку значение переменной `radius` не меняется, его не обязательно передавать в виде параметра. Вместо этого оно определяется в начале кода `drawRobot()`:

```
void setup() {
  size(720, 480);
  strokeWeight(2);
```

```
  ellipseMode(RADIUS);
}

void draw() {
  background(0, 153, 204);
  drawRobot(120, 420, 110, 140);
  drawRobot(270, 460, 260, 95);
  drawRobot(420, 310, 80, 10);
  drawRobot(570, 390, 180, 40);
}

void drawRobot(int x, int y, int bodyHeight, int neckHeight) {

  int radius = 45;
  int ny = y - bodyHeight - neckHeight - radius;  // neckHeight Y

  // Шея
  stroke(255);
  line(x+2, y-bodyHeight, x+2, ny);
  line(x+12, y-bodyHeight, x+12, ny);
  line(x+22, y-bodyHeight, x+22, ny);

  // Антенна
  line(x+12, ny, x-18, ny-43);
  line(x+12, ny, x+42, ny-99);
  line(x+12, ny, x+78, ny+15);

  // Туловище
  noStroke();
  fill(255, 204, 0);
  ellipse(x, y-33, 33, 33);
  fill(0);
  rect(x-45, y-bodyHeight, 90, bodyHeight-33);
  fill(255, 204, 0);
  rect(x-45, y-bodyHeight+17, 90, 6);

  // Голова
  fill(0);
  ellipse(x+12, ny, radius, radius);
  fill(255);
  ellipse(x+24, ny-6, 14, 14);
  fill(0);
  ellipse(x+24, ny-6, 3, 3);
  fill(153, 204, 255);
  ellipse(x, ny-8, 5, 5);
  ellipse(x+30, ny-26, 4, 4);
  ellipse(x+41, ny+6, 3, 3);
}
```

Глава 10

Объекты

Объектно-ориентированное программирование (ООП) – это еще один подход к разработке программ. Хотя термин «объектно-ориентированное программирование» звучит довольно устрашающе, есть хорошие новости: вы уже работаете с объектами начиная с главы 7, когда начали использовать `PImage`, `PFont`, `String` и `PShape`. В отличие от примитивных типов данных `boolean`, `int` и `float`, которые могут хранить только одно значение, объект может хранить несколько значений. Но это только часть истории. Объекты также позволяют объединять переменные со связанными функциями. Поскольку вы уже знаете, как работать с переменными и функциями, вам остается понять, что объекты просто объединяют изученные приемы в более понятный пакет.

Объекты важны, потому что они разбивают сущности на более мелкие строительные блоки. Этот подход отражает устройство мира природы, где, например, органы состоят из тканей, ткани – из клеток и так далее. Точно так же, когда ваш код становится более сложным, вы должны разбивать его на мелкие структуры, которые складываются в более сложные. Легче написать и поддерживать более мелкие и понятные фрагменты кода, которые работают вместе, чем написать один большой фрагмент кода, который делает все сразу.

10.1. Поля и методы

Объект в программе – это набор связанных переменных и функций. В контексте объектов переменная называется *полем* (или *переменной экземпляра*), а функция – *методом*. Поля и методы работают так же, как переменные и функции, описанные в предыдущих главах, но мы будем использовать новые термины, чтобы подчеркнуть, что они являются частью объекта. Другими словами, объект объединяет связанные данные (поля) со связанными действиями и поведением (методами). Идея состоит в том, чтобы сгруппировать связанные данные со связанными методами, которые работают с этими данными.

Например, чтобы смоделировать радиоприемник, подумайте, какие параметры можно настроить и какие действия могут повлиять на эти параметры:

Поля

`volume` (громкость), `frequency` (частота), `band` (диапазон), `power` (питание).

Методы

`setVolume` (настроить громкость), `setFrequency` (выбрать частоту), `setBand` (выбрать диапазон)

Моделирование простого механического устройства проще по сравнению с моделированием организма, такого как муравей или человек. Невозможно свести такие сложные организмы к нескольким полям и методам, но можно построить достаточно точную модель, чтобы создать интересную симуляцию. Компьютерная игра The Sims – яркий тому пример. В эту игру играют, управляя повседневной деятельностью смоделированных людей. Персонажи обладают достаточной индивидуальностью, чтобы сделать игру увлекательной, но не более того. На самом деле у них всего пять личностных качеств: аккуратный, общительный, активный, игривый и милый. Зная, что можно создать очень упрощенную модель сложных организмов, мы могли бы начать моделирование муравья, используя всего несколько полей и методов:

Поля

тип(рабочий, солдат), вес, длина
`type(worker, soldier)`, `weight`, `length`

Методы

ходить, ущипнуть, выпустить феромоны, есть
`walk`, `pinch`, `releasePheromones`, `eat`

Если вы составили список полей и методов муравья, вы можете сосредоточиться на различных аспектах муравья для моделирования. Невозможно создать модель без определенной подгонки ее под цели вашей программы.

10.2. Определение класса

Прежде чем вы сможете создать объект, вы должны определить класс. *Класс* – это спецификация объекта. Если использовать аналогию с архитектурой, класс похож на проект дома, а объект похож на сам дом. Каждый дом, построенный по чертежу, может иметь отличительные особенности, и чертеж – это только спецификация, а не построенная конструкция. Например, один дом может быть синим, а другой красным; в одном доме может быть камин, а в другом его не будет. Точно так же с объектами – класс определяет типы данных и поведение, но каждый объект (дом), созданный из одного класса (чертежа), имеет переменные (цвет, камин), которым присвоены разные значения. Если использовать более технические термины, каждый объект является экземпляром класса, и каждый экземпляр имеет свой собственный набор полей и методов.

Прежде чем писать класс, мы рекомендуем продумать план. Подумайте, какие поля и методы должны быть у вашего класса. Проведите небольшой мозговой штурм, чтобы представить все возможные варианты, а затем расставьте приоритеты и выберите наилучший по вашему мнению вариант. Вы

будете вносить изменения в процессе программирования, но важно иметь хорошее начало.

Для своих полей выберите понятные имена и определите тип данных для каждого из них. Поля внутри класса могут быть данными любого типа. Класс может одновременно содержать множество изображений, логических значений, значений с плавающей запятой, текстовых строк и т. д. Имейте в виду, что одна из причин создания класса – это потребность сгруппировать связанные элементы данных. Для ваших методов тоже выберите понятные имена и определите возвращаемые значения (если есть). Методы используются для изменения значений полей и выполнения действий на основе значений полей.

В качестве первого урока мы преобразуем код примера 8.9. Начнем с составления списка полей из примера:

```
float x
float y
int diameter
float speed
```

Следующий шаг – выяснить, какие методы могут быть полезны для класса. Глядя на код блока `draw()` из примера, который мы адаптируем, мы видим два основных компонента. Положение фигуры обновляется и отображается на экране. Давайте создадим два метода для нашего класса, по одному для каждой задачи:

```
void move()
void display()
```

Ни один из этих методов не возвращает значение, поэтому оба имеют тип возвращаемого значения `void`. Далее мы приступаем к написанию класса на основе списков полей и методов и должны выполнить четыре шага:

1) создать блок;
2) добавить поля;
3) написать конструктор (поясняется вкратце) для присвоения значений полям;
4) добавить методы.

Сначала создаем блок:

```
class JitterBug {

}
```

Обратите внимание, что ключевое слово `class` написано в нижнем регистре, а имя `JitterBug` содержит прописные буквы. Начинать имя класса с прописной буквы не обязательно, но это соглашение (соблюдать которое мы настоятельно рекомендуем) используется для обозначения того, что это класс. (Однако ключевое слово `class` должно быть в нижнем регистре, потому что это правило языка программирования.)

Затем мы добавляем поля. Когда мы это делаем, мы должны решить, каким полям будут присвоены значения с помощью *конструктора* – специального

метода, используемого для этой цели. Как правило, значения полей, которые должны быть разными для каждого объекта, передаются через конструктор, а значения других полей могут быть определены при их объявлении. Для класса `JitterBug` мы решили, что будут передаваться значения полей `x`, `y` и `diameter`. Итак, поля будут объявлены следующим образом:

```
class JitterBug {
  float x;
  float y;
  int diameter;
  float speed = 0.5;
}
```

Третьим шагом мы добавляем конструктор. Конструктор всегда имеет то же имя, что и класс. Назначение конструктора – присвоить начальные значения полям при создании объекта (экземпляра класса) (рис. 10.1). Код внутри блока конструктора запускается один раз при первом создании объекта. Как обсуждалось ранее, мы передаем конструктору три параметра при инициализации объекта. Каждое из переданных значений назначается временной переменной, которая существует только во время выполнения кода внутри конструктора. Чтобы прояснить это, мы добавили имя `temp` к каждой из этих переменных, но они могут быть названы любыми именами, которые вам нравятся. Они используются только для присвоения значений полям, которые являются частью класса. Также обратите внимание, что конструктор никогда не возвращает значение и, следовательно, перед ним не ставят ключевое слово `void` или обозначение типа данных. После добавления конструктора класс выглядит так:

```
class JitterBug {

  float x;
  float y;
  int diameter;
  float speed = 0.5;

  JitterBug(float tempX, float tempY, int tempDiameter) {
    x = tempX;
    y = tempY;
    diameter = tempDiameter;
  }

}
```

Последний шаг – добавление методов. Эта часть проста; она похожа на написание функций, просто здесь они содержатся внутри класса. Также обратите внимание на интервалы в коде. Каждая строка в классе имеет отступ на несколько пробелов, чтобы показать, что она находится внутри опреде-

```
Train red, blue;

void setup() {
  size(400, 400);
  red = new Train("Red Line", 90);
  blue = new Train("Blue Line", 120);
}

class Train {
  String name;
  int distance;
  Train (String tempName, int tempDistance) {
    name = tempName;
    distance = tempDistance;
  }
}
```

Присвоить «Red Line» переменной name для красного объекта

Присвоить 90 переменной distance для красного объекта

```
Train red, blue;

void setup() {
  size(400, 400);
  red = new Train("Red Line", 90);
  blue = new Train("Blue Line", 120);
}

class Train {
  String name;
  int distance;
  Train (String tempName, int tempDistance) {
    name = tempName;
    distance = tempDistance;
  }
}
```

Присвоить «Blue Line» переменной name для синего объекта

Присвоить 120 переменной distance для синего объекта

Рис. 10.1 ❖ Передача в конструктор значений полей объекта

ленного блока. В конструкторе и методах код снова разделен интервалом, чтобы четко показать иерархию:

```
class JitterBug {

  float x;
  float y;
  int diameter;
  float speed = 2.5;

  JitterBug(float tempX, float tempY, int tempDiameter) {
    x = tempX;
    y = tempY;
    diameter = tempDiameter;
  }

  void move() {
    x += random(-speed, speed);
    y += random(-speed, speed);
  }

  void display() {
    ellipse(x, y, diameter, diameter);
  }

}
```

10.3. Создание объектов

Теперь, когда вы определили класс, чтобы использовать его в программе, вы должны определить объект, принадлежащий к этому классу. Чтобы создать объект, нужно выполнить два шага:

1) объявите объектную переменную;
2) создайте (инициализируйте) объект с ключевым словом `new`.

Пример 10.1. Создание объекта

Мы начнем наш пример с демонстрации готового скетча Processing, а затем продолжим подробным объяснением каждой части:

```
JitterBug bug; // Объявляем объект

void setup() {
  size(480, 120);
  // Создаем экземпляр объекта и передаем ему параметры
  bug = new JitterBug(width/2, height/2, 20);
}

void draw() {
  bug.move();
  bug.display();
}

class JitterBug {

  float x;
  float y;
  int diameter;
  float speed = 2.5;

  JitterBug(float tempX, float tempY, int tempDiameter) {
    x = tempX;
    y = tempY;
    diameter = tempDiameter;
  }

  void move() {
    x += random(-speed, speed);
    y += random(-speed, speed);
  }

  void display() {
    ellipse(x, y, diameter, diameter);
  }

}
```

Каждый класс – это *тип данных*, а каждый объект – это *переменная* такого типа. Мы объявляем объектные переменные аналогично переменным из примитивных типов данных, таких как `boolean`, `int` и `float`. Объявление объекта начинается с указания типа данных, за которым следует имя переменной:

```
JitterBug bug;
```

Второй шаг – инициализация объекта ключевым словом `new`. Оно освобождает место для объекта в памяти и создает поля. Имя конструктора записывается справа от ключевого слова `new`, за которым следуют параметры в конструкторе, если они есть:

```
JitterBug bug = new JitterBug(200.0, 250.0, 30);
```

Три числа в круглых скобках – это параметры, которые передаются в конструктор класса `JitterBug`. Количество этих параметров и их типы данных должны соответствовать таковым у конструктора.

Пример 10.2. Создание нескольких объектов

В примере 10.1 мы видим кое-что новое: точку, которая используется для доступа к методам объекта внутри `draw()`. Оператор «точка» нужен для соединения имени объекта с его полями и методами. Это более наглядно показано в следующем примере, где два объекта созданы из одного и того же класса. Функция `jit.move()` относится к методу `move()`, который принадлежит объекту с именем `jit`, а `bug.move()` относится к методу `move()`, который принадлежит объекту с именем `bug`:

```
JitterBug jit;
JitterBug bug;

void setup() {
  size(480, 120);
  jit = new JitterBug(width * 0.33, height/2, 50);
  bug = new JitterBug(width * 0.66, height/2, 10);
}

void draw() {
  jit.move();
  jit.display();
  bug.move();
  bug.display();
}

class JitterBug {

  float x;
  float y;
  int diameter;
  float speed = 2.5;

  JitterBug(float tempX, float tempY, int tempDiameter) {
    x = tempX;
    y = tempY;
    diameter = tempDiameter;
  }

  void move() {
    x += random(-speed, speed);
    y += random(-speed, speed);
  }
```

```
  void display() {
    ellipse(x, y, diameter, diameter);
  }
}
```

10.4. Вкладки

Теперь, когда класс существует как отдельный модуль кода, любые изменения будут изменять объекты, созданные на его основе. Например, вы можете добавить к классу `JitterBug` поле, которое управляет цветом, или другое поле, определяющее размер фигуры. Эти значения могут быть переданы с помощью конструктора или назначены с помощью дополнительных методов, таких как `setColor()` или `setSize()`. А поскольку это автономный модуль, вы также можете использовать класс `JitterBug` в другом скетче.

Сейчас подходящее время, чтобы рассказать о наличии *вкладок* в среде разработки Processing (рис. 10.2). Вкладки позволяют размещать код более чем в одном файле. Это упрощает редактирование длинного кода и делает его более управляемым в целом. Для каждого класса обычно создается новая вкладка, которая улучшает модульность классов и упрощает поиск кода.

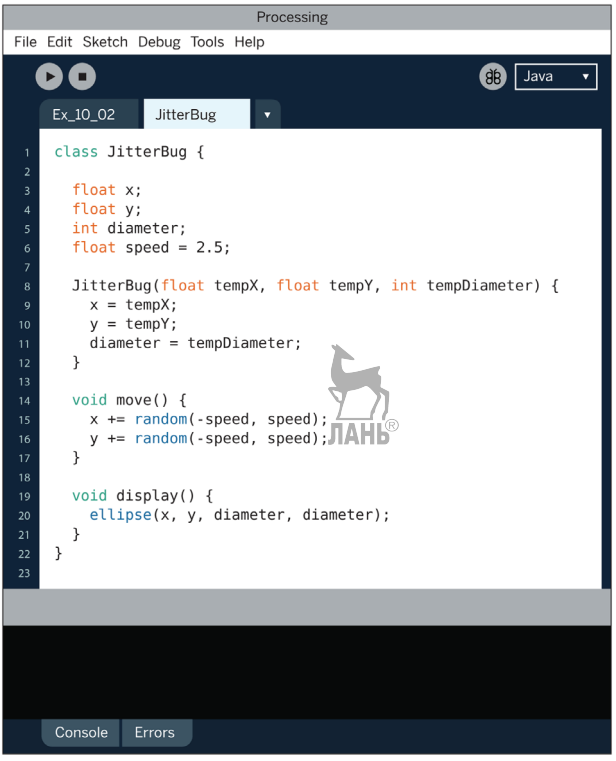
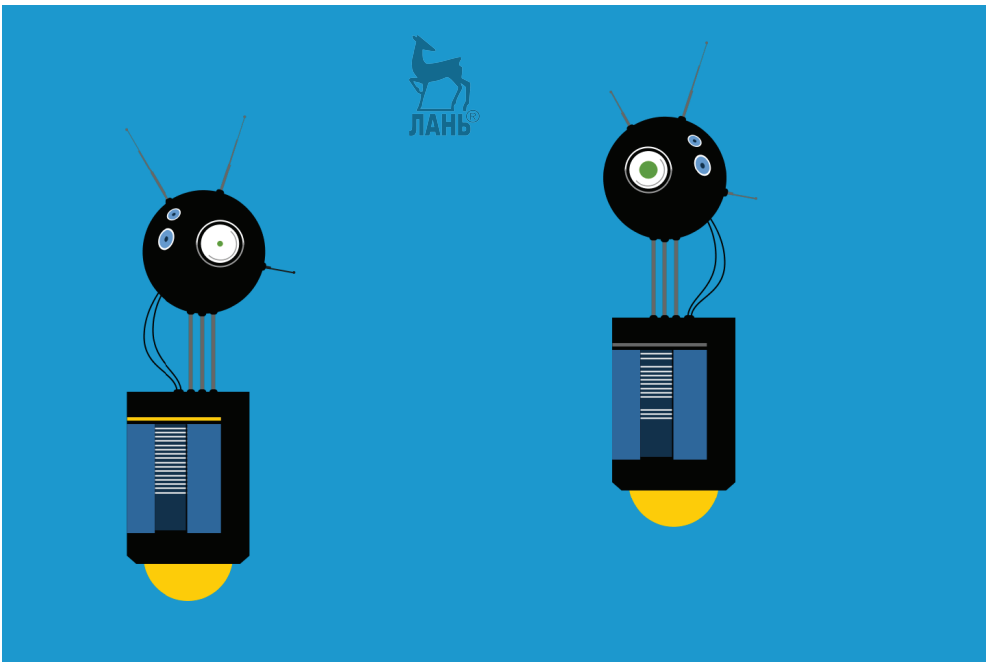


Рис. 10.2 ❖ Код можно разделить на разные вкладки, чтобы упростить управление

Чтобы создать новую вкладку, щелкните треугольную стрелку в правой части панели вкладок. Когда вы выбираете в меню опцию **Новая вкладка**, вам будет предложено придумать имя вкладки. Измените код текущего примера, создав отдельную вкладку для класса `JitterBug`.

Каждая вкладка отображается как отдельный файл *.pde* в папке скетча.

10.5. Робот 8: объекты



Программный объект объединяет методы (функции) и поля (переменные) в одно целое. Класс `Robot` в этом примере определяет все объекты роботов, которые будут созданы из него. Каждый объект `Robot` имеет свой собственный набор полей для хранения позиции и изображения, которое будет отображаться на экране. У каждого объекта есть методы для обновления положения и отображения рисунка.

Параметры `bot1` и `bot2` в `setup()` определяют координаты `x` и `y` и файл векторного изображения робота с расширением `.svg`. Параметры `tempX` и `tempY` передаются в конструктор и назначаются полям `xpos` и `ypos`. Параметр `svgName` используется для загрузки соответствующего файла изображения. Объекты (`bot1` и `bot2`) рисуются каждый в своем месте и со своим изображением, потому что каждый из них имеет уникальные значения, передаваемые в объекты через их конструкторы:

```
Robot bot1;
Robot bot2;

void setup() {
  size(720, 480);
  bot1 = new Robot("robot1.svg", 90, 80);
  bot2 = new Robot("robot2.svg", 440, 30);
}

void draw() {
  background(0, 153, 204);

  // Обновление и отображение первого робота
  bot1.update();
  bot1.display();

  // Обновление и отображение второго робота
  bot2.update();
  bot2.display();
}

class Robot {
  float xpos;
  float ypos;
  float angle;
  PShape botShape;
  float yoffset = 0.0;

  // Установка начальных значений в конструкторе
  Robot(String svgName, float tempX, float tempY) {
    botShape = loadShape(svgName);
    xpos = tempX;
    ypos = tempY;
    angle = random(0, TWO_PI);
  }

  // Обновление полей
  void update() {
    angle += 0.05;
    yoffset = sin(angle) * 20;
  }

  // Рисование робота на экране
  void display() {
    shape(botShape, xpos, ypos + yoffset);
  }
}
```

Глава 11

Массивы

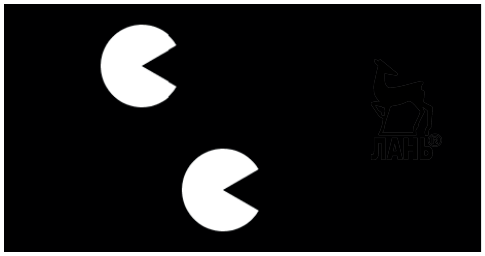
Массив – это список переменных с общим именем. Массивы полезны тем, что они позволяют работать с большим количеством переменных, не создавая для каждой из них новое имя. Это делает код короче, его легче читать и удобнее обновлять.

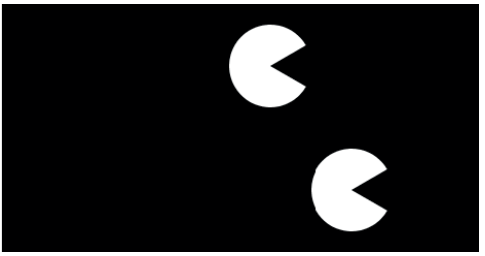
11.1. От переменных к массивам

Когда программе нужно отслеживать одно или два значения, нет нужды использовать массив. Добавление массива в такой ситуации может сделать программу более сложной, чем необходимо. Однако когда в программе много однотипных элементов (например, звездное поле в космической игре или несколько точек данных в визуализации), массивы упрощают написание кода.

Пример 11.1. Множество переменных

Чтобы лучше понять, о чем речь, вернитесь к примеру 8.3. Этот код отлично работает, если мы перемещаем только одну фигуру, но что, если мы хотим иметь две? Нам нужно создать новую переменную `x` и обновить ее в блоке `draw()`:



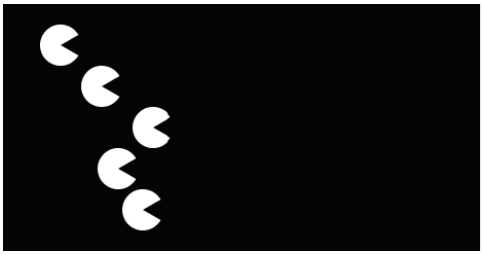


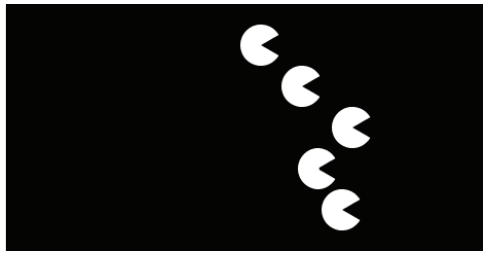
```
float x1 = -20;
float x2 = 20;

void setup() {
  size(240, 120);
  noStroke();
}
```

```
void draw() {
  background(0);
  x1 += 0.5;
  x2 += 0.5;
  arc(x1, 30, 40, 40, 0.52, 5.76);
  arc(x2, 90, 40, 40, 0.52, 5.76);
}
```

Пример 11.2. Когда переменных слишком много

Код предыдущего примера по-прежнему прост и понятен, но что, если мы хотим иметь пять кругов? Нам нужно добавить еще три переменные к двум, которые у нас уже есть:



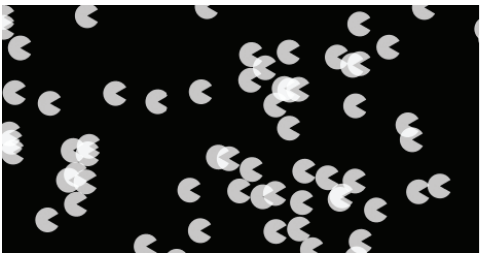


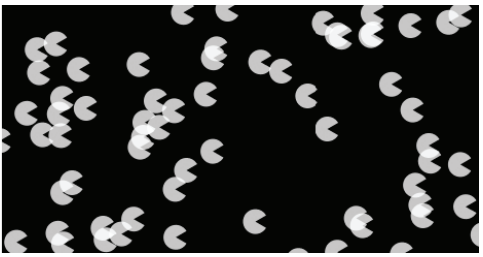
```
float x1 = -10;
float x2 = 10;
float x3 = 35;
float x4 = 18;
float x5 = 30;

void setup() {
  size(240, 120);
  noStroke();
}

void draw() {
  background(0);
  x1 += 0.5;
  x2 += 0.5;
  x3 += 0.5;
  x4 += 0.5;
  x5 += 0.5;
  arc(x1, 20, 20, 20, 0.52, 5.76);
  arc(x2, 40, 20, 20, 0.52, 5.76);
  arc(x3, 60, 20, 20, 0.52, 5.76);
  arc(x4, 80, 20, 20, 0.52, 5.76);
  arc(x5, 100, 20, 20, 0.52, 5.76);
}
```

Этот код начинает выходить из-под контроля.

Пример 11.3. Массивы вместо переменных

Представьте, что произойдет, если вы вдруг захотите иметь 3000 кругов. Это означало бы создание 3000 отдельных переменных с последующим обновлением каждой из них по отдельности. Могли бы вы отслеживать такое количество переменных? Вы бы хотели этим заниматься? Вместо этого мы используем массив:





```
float[] x = new float[3000];

void setup() {
  size(240, 120);
  noStroke();
  fill(255, 200);
  for (int i = 0; i < x.length; i++) {
    x[i] = random(-1000, 200);
  }
}

void draw() {
  background(0);
  for (int i = 0; i < x.length; i++) {
    x[i] += 0.5;
    float y = i * 0.4;
    arc(x[i], y, 12, 12, 0.52, 5.76);
  }
}
```

Остаток этой главы мы посвятим обсуждению деталей работы с массивами, лежащих в основе этого примера.

11.2. Создание массива

Каждый компонент массива называется *элементом*, а его позиция в массиве определяется *индексом*. Как и координаты на экране, значения индекса массива отсчитываются с 0. Например, первому элементу в массиве соответствует индекс 0, второму элементу соответствует индекс 1 и так далее. Если в массиве хранится 20 значений, значение индекса последнего элемента равно 19. На рис. 11.1 показана обобщенная структура массива.

```
int[] years = { 1920, 1972, 1980, 1996, 2010 };
```

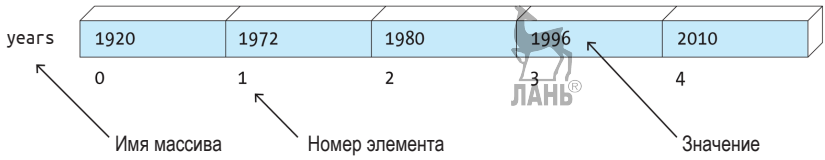


Рис. 11.1 ❖ Массив – это список из одной или нескольких переменных с одинаковыми именами

Использование массивов аналогично работе с отдельными переменными; применяются те же приемы. Как вы уже знаете, можно создать единственную целочисленную переменную с именем x с помощью следующего кода:

```
int x;
```

Чтобы создать массив, просто поставьте скобки после типа данных:

```
int[] x;
```

Прелесть создания массива заключается в возможности создать 2, 10 или 100 000 значений переменных с помощью всего одной строчки кода. Например, следующая строка создает массив из 2000 целочисленных переменных:

```
int[] x = new int [2000];
```

Вы можете создавать массивы из всех типов данных Processing: `boolean`, `float`, `String`, `PShape` и т. д., а также из любого определенного пользователем класса. Например, следующий код создает массив из 32 переменных `PImage`:

```
PImage[] images = new PImage[32];
```

Чтобы создать массив, начните с наименования типа данных, за которым следуют квадратные скобки. Далее следует имя, которое вы выбирали для массива, оператор присваивания (символ равенства), за которым идет ключевое слово `new`, потом снова наименование типа данных с количеством элементов, которые необходимо создать, помещенным в квадратные скобки. Этот шаблон работает для массивов всех типов данных.

Каждый массив может хранить данные только одного типа (`boolean`, `int`, `float`, `PImage` и т. д.). Нельзя смешивать и сопоставлять разные типы данных в одном массиве. Если вам нужно это сделать, используйте вместо массивов объекты.

Прежде чем двигаться дальше, давайте остановимся и поговорим о работе с массивами более подробно. Как и при создании объекта, работа с массивом состоит из трех шагов:

1) объявите массив и определите тип данных;
2) создайте массив с ключевым словом new и определите длину;
3) присвойте значения каждому элементу.

Каждый шаг может выполняться в отдельной строке, или все шаги могут быть собраны в одну операцию. Каждый из трех следующих примеров демонстрирует разные методы создания массива с именем `x`, в котором хранятся два целых числа, `12` и `2`. Обратите особое внимание на то, что происходит до блока `setup()` и что происходит внутри этого блока.

Пример 11.4. Объявление массива и присвоение значений

В этом примере мы объявляем массив снаружи блока `setup()`, а затем создаем и присваиваем значения внутри этого блока. Запись `x[0]` указывает на первый элемент массива, а запись `x[1]` – на второй элемент:

```
int[] x;           // Объявляем массив

void setup() {
  size(200, 200);
  x = new int[2]; // Создаем массив
  x[0] = 12;      // Присваиваем первое значение
  x[1] = 2;       // Присваиваем второе значение
}
```

Пример 11.5. Более компактное создание массива

Вот немного более компактный пример, в котором массив объявляется и создается в одной строке, а значения присваиваются в блоке `setup()`:

```
int[] x = new int[2]; // Объявляем и создаем массив

void setup() {
  size(200, 200);
  x[0] = 12;         // Присваиваем первое значение
  x[1] = 2;          // Присваиваем второе значение
}
```

Пример 11.6. Создание массива одной операцией

Вы также можете присвоить значения массиву при его создании, если он является частью одного оператора:

```
int[] x = { 12, 2 }; // Объявление, создание и присвоение

void setup() {
  size(200, 200);
}
```

Избегайте создания массивов в блоке `draw()`, потому что создание нового массива в каждом кадре приведет к снижению общей частоты кадров.

Пример 11.7. Возвращаясь к первому примеру

В качестве полного примера использования массивов мы перепишем здесь пример 11.1. Хотя мы пока не видим всех преимуществ, показанных в примере 11.3, становятся очевидны некоторые важные детали того, как работают массивы:

```
float[] x = {-20, 20};
  void setup() {
  size(240, 120);
  noStroke();
}

void draw() {
  background(0);
  x[0] += 0.5; // Увеличиваем первый элемент
  x[1] += 0.5; // Увеличиваем второй элемент
  arc(x[0], 30, 40, 40, 0.52, 5.76);
  arc(x[1], 90, 40, 40, 0.52, 5.76);
}
```

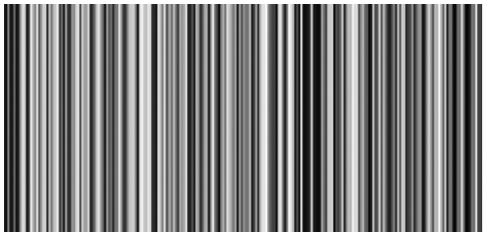
11.3. Повторение и массивы

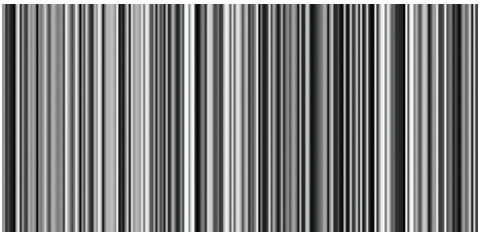
Цикл `for`, представленный в разделе 4.5, упрощает работу с большими массивами, сохраняя при этом лаконичность кода. Идея состоит в том, чтобы написать цикл для перемещения по каждому элементу массива один за другим. Для этого нужно знать длину массива. Поле `length`, связанное с каждым массивом, хранит количество его элементов. Для доступа к этому значению мы используем имя массива с оператором точки. Например:

```
int[] x = new int[2];    // Объявление и создание массива
println(x.length);       // Выводит в консоль число 2

int[] y = new int[1972]; // Объявление и создание массива
println(y.length);       // Выводит в консоль число 1972
```

Пример 11.8. Заполнение массива при помощи цикла `for`

Цикл `for` может использоваться для заполнения массива значениями или для последовательного чтения значений. В этом примере массив сначала заполняется случайными числами внутри блока `setup()`, а затем эти числа используются для установки значения градации серого цвета внутри блока `draw()`. Каждый раз при запуске программы в массив помещается новый набор случайных чисел:

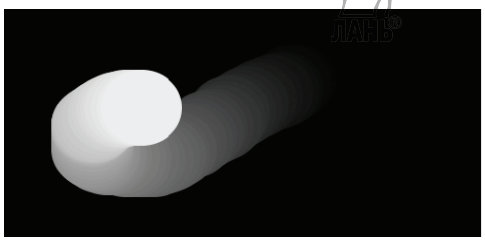


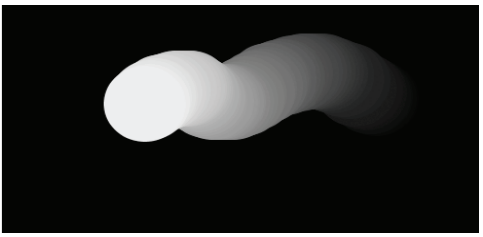


```
float[] gray;

void setup() {
  size(240, 120);
  gray = new float[width];
  for (int i = 0; i < gray.length; i++) {
    gray[i] = random(0, 255);
  }
}

void draw() {
  for (int i = 0; i < gray.length; i++) {
    stroke(gray[i]);
    line(i, 0, i, height);
  }
}
```


Пример 11.9. Отслеживание перемещений мыши

В этом примере есть два массива для хранения положения мыши: один для координаты x, а другой – для координаты y. Эти массивы хранят местоположение мыши для предыдущих 60 кадров. С каждым новым кадром самые старые значения координат x и y удаляются и заменяются текущими значениями mouseX и mouseY. Новые значения добавляются в первую позицию массива, но до того, как это произойдет, каждое значение в массиве перемещается на одну позицию вправо (от начала к концу), чтобы освободить место для новых чисел. Этот пример визуализирует данную процедуру. Кроме того, в каждом кадре все 60 координат используются для рисования серии кругов на экране:





```
int num = 60;
int[] x = new int[num];
int[] y = new int[num];
```

```
void setup() {
  size(240, 120);
  noStroke();
}

void draw() {
  background(0);
  // копирование элементов массива в обратном порядке
  for (int i = x.length-1; i > 0; i--) {
    x[i] = x[i-1];
    y[i] = y[i-1];
  }
  x[0] = mouseX; // Задаем значение первого элемента
  y[0] = mouseY; // Задаем значение первого элемента
  for (int i = 0; i < x.length; i++) {
    fill(i * 4);
    ellipse(x[i], y[i], 40, 40);
  }
}
```

Техника хранения сдвигового буфера чисел в массиве, показанном в этом примере и на рис. 11.2, менее эффективна, чем альтернативная техника, использующая оператор % (модуль). Этот прием показан в одном из примеров (Примеры > Basics > Input > StoringInput), представленных в среде разработки Processing.

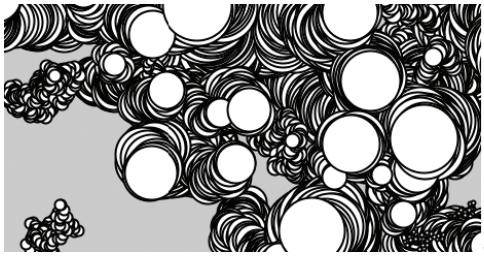
Рис. 11.2 ❖ Сдвиг значений в массиве на одну позицию вправо

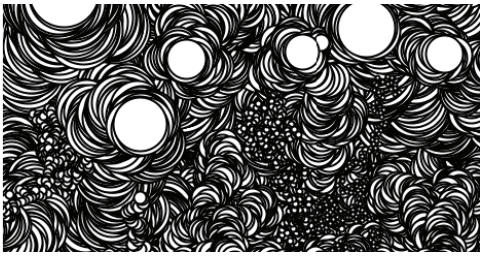
11.4. Массивы объектов

Два коротких примера в этом разделе объединяют все основные концепции программирования, о которых мы уже говорили в этой книге: переменные, итерацию, условные выражения, функции, объекты и массивы. Создание *массива объектов* – это почти то же самое, что создание массивов, о которых мы говорили на предыдущих страницах, но есть одно дополнительное соображение: поскольку каждый элемент массива является объектом, он должен быть сначала создан с ключевым словом `new` (как и любой другой объект) перед присвоением его массиву. В случае пользовательского класса, такого как `JitterBug` (см. главу 10), это означает использование `new` для объявления каждого элемента до того, как он будет назначен массиву. В случае встроенного класса Processing, такого как `PImage`, это означает использование функции `loadImage()` для создания объекта до его присвоения массиву.

Пример 11.10. Работа с множеством объектов

В этом примере создается массив из 33 объектов `JitterBug`, а затем каждый из них обновляется и отображается внутри `draw()`. Чтобы этот пример работал, вам нужно добавить в код класс `JitterBug`:





```
JitterBug[] bugs = new JitterBug[33];

void setup() {
  size(240, 120);
  for (int i = 0; i < bugs.length; i++) {
    float x = random(width);
    float y = random(height);
    int r = i + 2;
    bugs[i] = new JitterBug(x, y, r);
  }
}

void draw() {
  for (int i = 0; i < bugs.length; i++) {
    bugs[i].move();
    bugs[i].display();
  }
}

// Вставьте сюда класс JitterBug из примера 10.1
```

Пример 11.11. Новый способ управления объектами

При работе с массивами объектов можно использовать цикл другого типа, называемый «улучшенным» циклом `for`. Вместо создания новой переменной счетчика, такой как переменная `i` в примере 11.10, можно напрямую перебирать элементы массива или списка. В следующем примере каждый объект в массиве `bugs` объектов `JitterBug` поочередно присваивается временной переменной `b`, чтобы вызвать методы `move()` и `display()` для всех объектов в массиве.

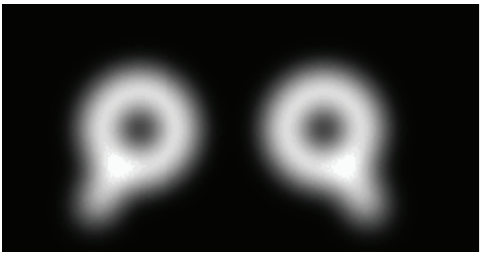
Улучшенный цикл `for` часто более эффективен, чем цикл с числом, хотя в этом примере мы не использовали его внутри блока `setup()`, потому что переменная `i` была нужна в двух местах внутри цикла, демонстрируя, как иногда полезно иметь доступ к такому числу:

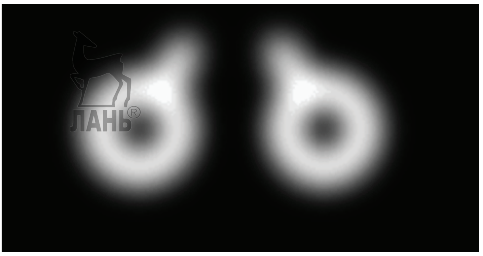
```
JitterBug[] bugs = new JitterBug[33];

void setup() {
  size(240, 120);
  for (int i = 0; i < bugs.length; i++) {
    float x = random(width);
    float y = random(height);
    int r = i + 2;
    bugs[i] = new JitterBug(x, y, r);
  }
}

void draw() {
  for (JitterBug b : bugs) {
    b.move();
    b.display();
  }
}

// Вставьте сюда класс JitterBug из примера 10.1
```

Код последнего примера этой главы загружает последовательность изображений и сохраняет каждое из них как элемент в массиве объектов `PImage`.

Пример 11.12. Последовательности изображений

Чтобы запустить этот пример, извлеките изображения из архива `media.zip`, как описано в главе 7. Изображения именованы последовательно (`frame-0000.png`, `frame-0001.png` и т. д.), что позволяет создавать имя каждого файла в цикле `for`, как показано в восьмой строке программы:

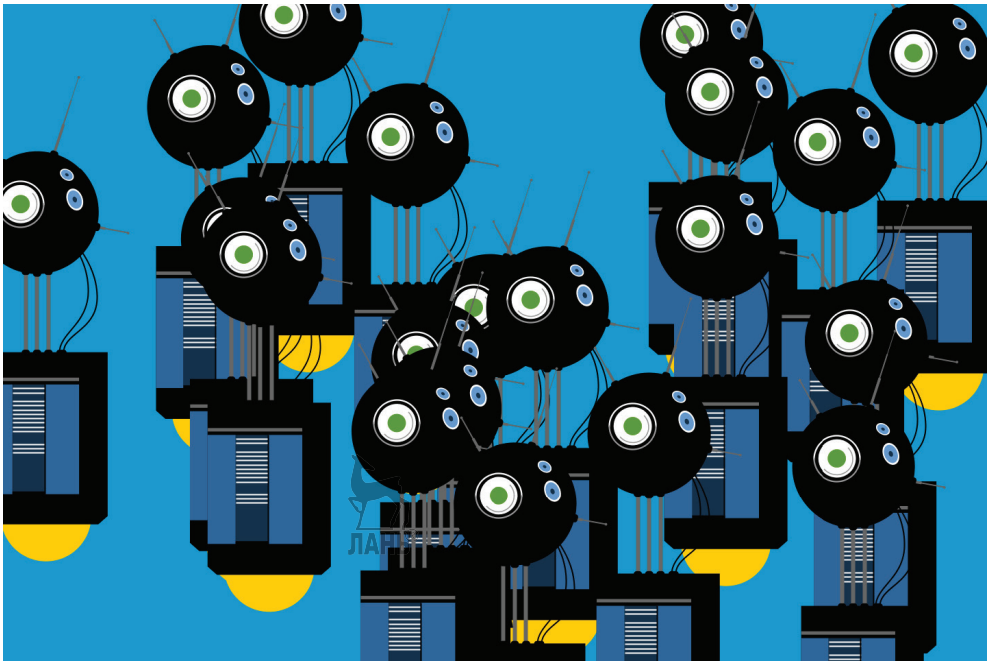




```
int numFrames = 12; // Количество кадров
PImage[] images = new PImage[numFrames]; // Создаем массив
int currentFrame = 0;

void setup() {
  size(240, 120);
  for (int i = 0; i < images.length; i++) {
    String imageName = "frame-" + nf(i, 4) + ".png";
    images[i] = loadImage(imageName); // Читаем каждое изображение
  }
  frameRate(24);
}

void draw() {
  image(images[currentFrame], 0, 0);
  currentFrame++; // Следующий кадр
  if (currentFrame >= images.length) {
    currentFrame = 0; // Возврат к первому кадру
  }
}
```

Функция `nf()` форматирует числа таким образом, что `nf(1, 4)` возвращает строку «`0001`», а `nf(11, 4)` возвращает «`0011`». Эти значения объединяются с началом (`frame-`) и концом (`.png`) имени файла для создания полного имени файла в виде строковой переменной. Файлы загружаются в массив в следующей строке кода. Изображения выводятся на экран по одному в блоке `draw()`. Когда отображается последнее изображение в массиве, программа возвращается к началу массива и снова показывает изображения по порядку.

11.5. Робот 9: массивы



Массивы упрощают работу программы со многими элементами. В этом примере в начале кода объявлен массив объектов `Robot`. Затем внутри блока `setup()` под массив выделяется место в памяти компьютера, а внутри цикла `for` создается каждый объект `Robot`. В `draw()` другой цикл `for` используется для обновления и отображения каждого элемента массива `bots`.

Цикл `for` и массив составляют мощную комбинацию. Обратите внимание на тонкие различия между кодом для этого примера и кодом робота 8 (см. раздел 10.5), которые ведут к значительным изменениям визуального результата. Как только создан массив и задан цикл `for`, работать с 3 элементами становится так же легко, как и с 3000.

Основным изменением по сравнению с роботом 8 является решение загружать файл SVG в блоке `setup()`, а не в классе `Robot`. Этот выбор был сделан по той причине, что файл загружается только один раз, а не столько раз, сколько элементов в массиве (в данном случае 20 раз). Это изменение ускоряет запуск кода, поскольку загрузка файла требует времени и к тому же экономит память, так как файл сохраняется один раз. Каждый элемент массива `bots` ссылается на один и тот же файл:

```
Robot[] bots;  // Объявляем массив объектов Robot

void setup() {
  size(720, 480);
  PShape robotShape = loadShape("robot2.svg");
  // Создаем массив объектов Robot
  bots = new Robot[20];
  // Создаем каждый объект
  for (int i = 0; i < bots.length; i++) {
    // Генерируем случайную координату x
    float x = random(-40, width-40);
    // Вычисляем и присваиваем координату y
    float y = map(i, 0, bots.length, -100, height-200);
    bots[i] = new Robot(robotShape, x, y);
  }
}

void draw() {
  background(0, 153, 204);
  // Обновляем и отображаем каждого робота в массиве
  for (int i = 0; i < bots.length; i++) {
    bots[i].update();
    bots[i].display();
  }
}

class Robot {
  float xpos;
  float ypos;
  float angle;
  PShape botShape;
  float yoffset = 0.0;

  // Задаем начальные значения в конструкторе
  Robot(PShape shape, float tempX, float tempY) {
    botShape = shape;
    xpos = tempX;
    ypos = tempY;
    angle = random(0, TWO_PI);
  }

  // Обновляем поля
  void update() {
    angle += 0.05;
    yoffset = sin(angle) * 20;
  }

  // Рисуем робота на экране
  void display() {
    shape(botShape, xpos, ypos + yoffset);
  }
}
```

Глава 12

Данные

Визуализация данных – одна из самых активно развивающихся областей на стыке кода и графики, а также одна из самых популярных областей применения языка Processing. Эта глава основана на том, что вам уже известно из этой книги о хранении и загрузке данных; далее мы расскажем о дополнительных функциях, относящихся к наборам данных, которые можно использовать для визуализации.

Существует множество приложений, способных делать стандартные визуализации, такие как гистограммы и точечные диаграммы. Однако возможность написать собственный код для создания этих визуализаций с нуля обеспечивает больший контроль над графическим представлением и побуждает пользователей придумывать, исследовать и создавать более уникальные представления данных. По нашему мнению, это отличный способ обучения программированию с использованием такого языка, как Processing, и мы считаем, что это намного интереснее, чем ограничиваться заранее подготовленными стандартными методами или инструментами.

12.1. Что мы знаем о данных

Пришло время вернуться назад и обсудить, как были представлены данные в этой книге. Переменная в скетче Processing используется для хранения фрагмента данных. Мы начали с *примитивов*. В данном случае слово «примитив» означает отдельный фрагмент данных. Например, `int` хранит одно и только одно целое число. Идея разделения данных по типам очень важна. Каждый тип данных уникален и хранится по-разному. Числа с плавающей запятой (числа с десятичными знаками), целые числа (без десятичных знаков) и буквенно-цифровые символы (буквы и цифры) относятся к разным типам данных, таких как `float`, `int` и `char` соответственно.

Для хранения списка элементов с одним именем переменной предназначены массивы. Так, в примере 11.8 хранятся сотни чисел с плавающей запятой, которые используются для установки значения яркости линии. Массивы могут быть созданы из любого примитивного типа данных, но способны хранить данные только одного типа. Если вы хотите объединить в одной структуре данные разных типов, то должны использовать класс.

Классы `String`, `PImage`, `PFont` и `PShape` хранят более одного элемента данных, и каждый из них уникален. Например, `String` может хранить более одного символа – слово, предложение или целый абзац. Кроме того, у него есть методы для получения длины данных или возврата их версий в верхнем или нижнем регистре. В качестве другого примера `PImage` предоставляет массив пикселей и переменных, в которых хранятся ширина и высота изображения.

Объекты, созданные из классов `String`, `PImage` и `PShape`, можно определить в коде или загрузить из файла в папке `data` скетча. Примеры в этой главе также загружают данные в скетчи, но они используют новые классы, которые хранят данные по-разному.

Класс `Table` введен для хранения данных в таблице, состоящей из строк и столбцов. Классы `JSONObject` и `JSONArray` предназначены для хранения данных, загружаемых через файлы в формате JSON. Форматы файлов для `Table`, `JSONObject` и `JSONArray` более подробно обсуждаются в следующем разделе.

Формат данных `XML` – это еще один встроенный формат данных для Processing, он задокументирован в справочнике по языку, но не рассматривается в этой книге.

12.2. Таблицы

Многие наборы данных хранятся в виде строк и столбцов, поэтому Processing предоставляет класс `Table`, чтобы упростить работу с таблицами (рис. 12.1). Если вы работали с электронными таблицами, то без труда освоите работу с таблицами в коде. Processing может прочитать таблицу из файла или создать новую прямо в коде. Также возможно чтение и запись в любую строку или столбец и изменение отдельных ячеек в таблице. В этой главе мы сосредоточимся на работе с табличными данными.

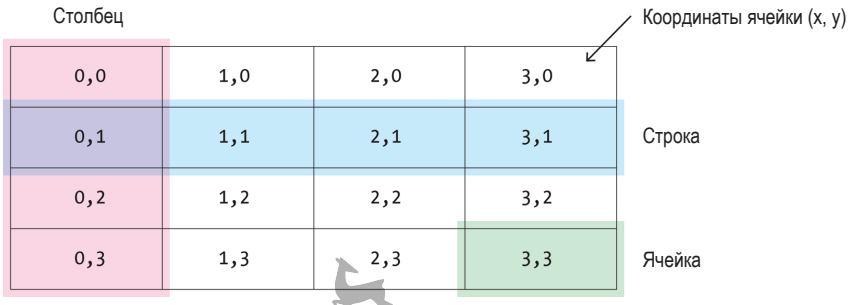


Рис. 12.1 ❖ Таблицы представляют собой сетки ячеек. Строки – это горизонтальные ряды элементов, а столбцы – вертикальные. Данные можно читать из отдельных ячеек, строк и столбцов

Табличные данные часто хранятся в текстовых файлах со столбцами, в которых в качестве разделителя используются запятые или символ табуляции. Файл значений, разделенных запятыми, сокращенно называется CSV

и имеет расширение .csv. При использовании вкладок иногда применяется расширение .tsv.

Чтобы загрузить файл CSV или TSV, сначала поместите его в data скетча, как описано в начале главы 7, а затем используйте функцию loadTable(), чтобы получить данные и поместить их в объект, созданный из класса Table.

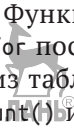
В следующих примерах показаны только первые несколько строк каждого набора данных. Если вы вводите код вручную, вам понадобится весь файл .csv, .json или .tsv, чтобы воспроизвести визуализации, показанные на иллюстрациях к примерам. Вы можете найти эти файлы в папке data примера скетча (см. раздел 2.5).

Данные для следующего примера представляют собой упрощенную версию статистики ударов игрока команды «Бостон Ред Сокс» Дэвида Ортиса с 1997 по 2014 год. Слева направо указаны год, количество хоум-ранов, количество результативных проходов и среднее количество ударов. При открытии в текстовом редакторе первые пять строк файла выглядят следующим образом:


```
1997,1,6,0.327
1998,9,46,0.277
1999,0,0,0
2000,10,63,0.282
2001,18,48,0.234
```

Пример 12.1. Чтение таблицы

Чтобы загрузить эти данные в Processing, создается объект из класса Table. Объект в этом примере называется stats. Функция loadTable() загружает файл ortiz.csv из папки data. Оттуда цикл for последовательно читает каждую строку таблицы. Он получает данные из таблицы и сохраняет их в переменных типа int и float. Метод getRowCount() используется для подсчета количества строк в файле данных. Поскольку эти данные представляют собой статистику Ортиса с 1997 по 2014 год, необходимо прочитать 18 строк данных:




```
Table stats;

void setup() {
  stats = loadTable("ortiz.csv");
  for (int i = 0; i < stats.getRowCount(); i++) {
    // Получение целого числа из строки i, столбца 0 в файле
    int year = stats.getInt(i, 0);
    // Получение целого числа из строки i, столбца 1
    int homeRuns = stats.getInt(i, 1);
    int rbi = stats.getInt(i, 2);
    // Чтение числа, содержащего десятичную точку
    float average = stats.getFloat(i, 3);
    println(year, homeRuns, rbi, average);
  }
}
```

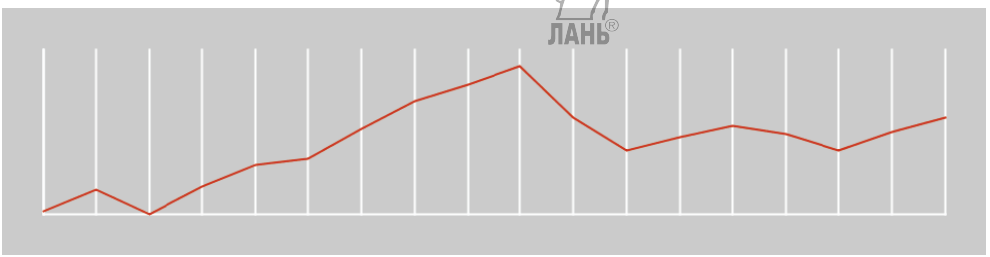
Внутри цикла for для получения данных из таблицы используются методы getInt() и getFloat(). Важно использовать метод getInt() для чтения целочисленных данных, а метод getFloat() – для чисел с плавающей запятой. Оба этих метода имеют два параметра: первый – это строка, из которой нужно читать, а второй – столбец.

Пример 12.2. Визуализация табличных данных

Следующий пример основан на данных из предыдущего примера. Сначала мы создаем в setup() массив с именем homeRuns для хранения данных после загрузки, а данные из этого массива используются в draw(). Длина homeRuns применяется трижды с кодом homeRuns.length для подсчета количества итераций цикла for.

homeRuns сначала используется в setup(), чтобы определить, сколько раз нужно извлечь целое число из таблицы. Затем это значение применяется для размещения вертикальной отметки на графике для каждого элемента данных в массиве. В третий раз оно используется для чтения каждого элемента массива по порядку и для прекращения чтения из массива в конце. После того как данные загружены в setup() и прочитаны в массив, в остальной части программы мы применяем знания, полученные в главе 11.

Этот пример представляет собой визуализацию упрощенной версии статистики игрока команды «Бостон Ред Сокс» Дэвида Ортиса с 1997 по 2014 год, взятой из таблицы:



```
int[] homeRuns;

void setup() {
  size(480, 120);
  Table stats = loadTable("ortiz.csv");
  int rowCount = stats.getRowCount();
  homeRuns = new int[rowCount];
  for (int i = 0; i < homeRuns.length; i++) {
    homeRuns[i] = stats.getInt(i, 1);
  }
}

void draw() {
  background(204);
  // Рисуем фоновую разметку
```

```
  stroke(255);
  line(20, 100, 20, 20);
  line(20, 100, 460, 100);
  for (int i = 0; i < homeRuns.length; i++) {
    float x = map(i, 0, homeRuns.length-1, 20, 460);
    line(x, 20, x, 100);
  }
  // Рисуем график на основании данных о хоум-ранах
  noFill();
  stroke(204, 51, 0);
  beginShape();
  for (int i = 0; i < homeRuns.length; i++) {
    float x = map(i, 0, homeRuns.length-1, 20, 460);
    float y = map(homeRuns[i], 0, 60, 100, 20);
    vertex(x, y);
  }
  endShape();
}
```

Этот пример настолько прост, что нет необходимости хранить эти данные в массивах, но идею можно применить к более сложным примерам, которые вы, возможно, захотите создать в будущем. Кроме того, вы можете подумать о том, как внести в график дополнительную информацию – например, указать количество хоум-ранов по вертикальной оси, а по горизонтали указать год.

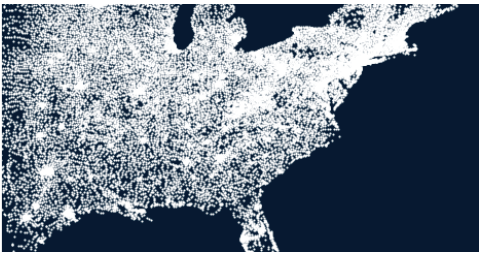
Пример 12.3. 29 740 городов

Чтобы вы лучше ощутили потенциал работы с таблицами данных, в следующем примере используется больший набор данных и представлена удобная функция. У этих данных есть особенность – первая строка в файле является заголовком. Заголовок определяет наименование для каждого столбца, чтобы прояснить контекст. Ниже показаны первые пять строк нашего нового файла данных с именем `city.csv`:

```
zip,state,city,lat,lng
35004,AL,Acmar,33.584132,-86.51557
35005,AL,Adamsville,33.588437,-86.959727
35006,AL,Adger,33.434277,-87.167455
35007,AL,Keystone,33.236868,-86.812861
```

Заголовок упрощает чтение кода, например во второй строке файла указан почтовый индекс 35004 (`zip`) города (`city`) Акмар, штат (`state`) Алабама, широта (`lat`) города 33,584132, а долгота (`lng`) –86,51557. В общей сложности файл состоит из 29 741 строки и описывает географическое местоположение и почтовые индексы 29 740 городов США.

В текущем примере эти данные загружаются в `setup()`, а затем выводятся на экран в цикле `for` внутри блока `draw()`. Функция `setXY()` преобразует данные широты и долготы из файла в точку на экране:



```
Table cities;

void setup() {
  size(240, 120);
  cities = loadTable("cities.csv", "header");
  stroke(255);
}

void draw() {
  background(0, 26, 51);
  float xoffset = map(mouseX, 0, width, -width*3, -width);
  translate(xoffset, -300);
  scale(10);
  strokeWeight(0.1);
  for (int i = 0; i < cities.getRowCount(); i++) {
    float latitude = cities.getFloat(i, "lat");
    float longitude = cities.getFloat(i, "lng");
    setXY(latitude, longitude);
  }
}

void setXY(float lat, float lng) {
  float x = map(lng, -180, 180, 0, width);
  float y = map(lat, 90, -90, 0, height);
  point(x, y);
}
```

Обратите внимание, что в блоке `setup()` функция `loadTable()` получает второй параметр «`header`». Если этого не сделать, код будет рассматривать первую строку CSV-файла как данные, а не как заголовок каждого столбца.

Класс `Table` имеет множество методов, таких как добавление и удаление столбцов и строк, получение списка уникальных записей в столбце или сортировка таблицы. Более полный список методов с короткими примерами приведен в справочнике по языку Processing.

12.3. Данные в формате JSON

Формат JSON (JavaScript Object Notation) – еще одна распространенная система хранения данных. Подобно форматам HTML и XML, элементы имеют связанные с ними *ярлыки*. Например, связанные с фильмом данные могут

включать ярлыки для названия (`title`), режиссера (`director`), года выпуска (`year`), рейтинга (`rating`) и т. д.

Эти ярлыки будут связаны с данными следующим образом:

```
"title": "Alphaville"
"director": "Jean-Luc Godard"
"year": 1964
"rating": 7.2
```

Чтобы сформировать правильный файл JSON, нам потребуются дополнительные знаки препинания для разделения элементов. Между каждой парой данных ставятся запятые, и группа связанных данных заключается в фигурные скобки. Данные, указанные в фигурных скобках, представляют собой *объект JSON*.

С этими изменениями наш правильный файл данных JSON выглядит следующим образом:

```
{
  "title": "Alphaville",
  "director": "Jean-Luc Godard",
  "year": 1964,
  "rating": 7.2
}
```

В этом коротком примере JSON есть еще одна интересная деталь, связанная с типами данных: вы заметите, что данные заголовка (`title`) и режиссера (`director`) заключены в кавычки, чтобы указать, что это строковые данные, а год и рейтинг не заключены в кавычки, чтобы определить их как числа. В частности, год является целым числом, а рейтинг – числом с плавающей запятой. Это различие становится важным после загрузки данных в скетч.

Чтобы добавить еще один фильм в список, нам придется вставить квадратные скобки вверху и внизу, указав таким образом, что данные представляют собой массив объектов JSON. Каждый объект массива разделяется запятой:

```
[
  {
    "title": "Alphaville",
    "director": "Jean-Luc Godard",
    "year": 1964,
    "rating": 7.2
  },
  {
    "title": "Pierrot le Fou",
    "director": "Jean-Luc Godard",
    "year": 1965,
    "rating": 7.7
  }
]
```

Аналогичным образом мы можем добавить в массив JSON информацию о других фильмах. На этом этапе интересно сравнить запись в формате JSON с соответствующим табличным представлением тех же данных.

В формате CSV данные выглядят так:

```
title, director, year, rating
Alphaville, Jean-Luc Godard, 1964, 9.1
Pierrot le Fou, Jean-Luc Godard, 1965, 7.7
```

Обратите внимание, что в формате CSV задействовано меньше символов, что может быть важно при работе с большими наборами данных. С другой стороны, версию JSON часто легче читать, потому что каждый элемент данных снабжен ярлыком.

Теперь, когда мы изучили основы формата JSON и его связь с табличными данными, давайте посмотрим на код, необходимый для чтения файла JSON в скетч Processing.

Пример 12.4. Чтение файла JSON

Этот скетч загружает файл JSON, показанный в начале данного раздела, который содержит только данные для одного фильма Alphaville:

```
JSONObject film;

void setup() {
  film = loadJSONObject("film.json");
  String title = film.getString("title");
  String dir = film.getString("director");
  int year = film.getInt("year");
  float rating = film.getFloat("rating");
  println(title + " by " + dir + ", " + year);
  println("Rating: " + rating);
}
```

Класс `JSONObject` используется для создания объекта для хранения данных. Как только эти данные загружены, каждый отдельный фрагмент данных может быть прочитан последовательно или путем запроса данных, относящихся к конкретной метке. Обратите внимание, что разные типы данных запрашиваются по имени типа данных. Метод `getString()` используется для извлечения названия фильма, а метод `getInt()` – для извлечения года выпуска.

Пример 12.5. Визуализация данных из файла JSON

Для работы с файлом JSON, который содержит данные более, чем об одном фильме, нам нужно ввести новый класс `JSONArray`. Файл данных, начатый в предыдущем примере, был обновлен и теперь включает все фильмы режиссера 1960–1966 годов. Название каждого фильма расположено на экране в порядке, соответствующем году выпуска, и ему присваивается значение уровня серого на основе значения рейтинга.

Есть несколько различий между этим примером и примером 12.4. Наиболее важным отличием является способ загрузки файла JSON в объекты `Film`. Файл JSON загружается в блоке `setup()`, и каждый объект `JSONObject`, представляющий один фильм, передается в конструктор класса `Film`. Конструктор назначает данные `JSONObject` полям `String`, `float` и `int` внутри каждого объекта `Film`. Класс `Film` также имеет метод для отображения названия фильма:


```
Film[] films;

void setup() {
  size(480, 120);
  JSONArray filmArray = loadJSONArray("films.json");
  films = new Film[filmArray.size()];
  for (int i = 0; i < films.length; i++) {
    JSONObject o = filmArray.getJSONObject(i);
    films[i] = new Film(o);
  }
}

void draw() {
  background(0);
  for (int i = 0; i < films.length; i++) {
    int x = i*32 + 32;
    films[i].display(x, 105);
  }
}

class Film {

  String title;
  String director;
  int year;
  float rating;

  Film(JSONObject f) {
    title = f.getString("title");
    director = f.getString("director");
    year = f.getInt("year");
    rating = f.getFloat("rating");
  }

  void display(int x, int y) {
    float ratingGray = map(rating, 6.5, 8.1, 102, 255);
```

```
    pushMatrix();
    translate(x, y);
    rotate(-QUARTER_PI);
    fill(ratingGray);
    text(title, 0, 0);
    popMatrix();
  }
}
```

Этот пример – всего лишь заготовка визуализации данных фильма. Он показывает, как загружать данные и как использовать их для визуализации, но ваша задача – отформатировать их таким образом, чтобы выделить то, что вам интересно в данных. Например, хотите ли вы показать количество фильмов, которые режиссер Годар снимает за год? А может быть, вам интереснее сравнить и сопоставить эти данные с фильмами другого режиссера? Будут ли данные легче читаться с другим шрифтом, размером экрана или соотношением сторон? Попробуйте применить полученные ранее навыки, чтобы довести этот скетч до нового уровня визуального представления данных.

12.4. Сетевые данные и API

Публичный доступ к огромным объемам данных, собранных правительствами, корпорациями, организациями и отдельными людьми, меняет нашу культуру, начиная с нашего способа общения и заканчивая нашим отношением к нематериальным идеям, таким как конфиденциальность. Доступ к этим данным чаще всего осуществляется через программные структуры, называемые API.

Термин «API» выглядит загадочно, и его значение – интерфейс прикладного программирования (application programming interface) – ненамного понятнее. Однако API-интерфейсы необходимы для работы с данными и устроены не так уж сложно. По сути, это запросы к некоторой внешней службе с просьбой предоставить данные. Когда наборы данных огромны, копировать все данные нецелесообразно или нежелательно; API позволяет программисту запрашивать только те данные, которые имеют отношение к его задаче.

Эту идею можно более четко проиллюстрировать на гипотетическом примере. Предположим, есть организация, которая ведет базу данных диапазонов температур для каждого города страны. API для этого набора данных позволяет программисту запрашивать высокие и низкие температуры для любого города, допустим, в течение октября 1972 года. Чтобы получить доступ к этим данным, запрос должен быть выполнен через определенную строку или строки кода в формате, установленном службой данных.

Некоторые API являются полностью общедоступными, но многие требуют аутентификации, которая обычно представляет собой уникальный идентификатор пользователя или ключ, чтобы служба данных могла узнать своих пользователей. У большинства API есть строгие правила относительно того, как часто можно делать запросы. Например, можно делать только 1000 запросов в месяц или не более одного запроса в секунду.

Processing может запрашивать данные через интернет, когда компьютер, на котором запущена программа, находится в сети. Файлы CSV, TSV, JSON и XML можно загрузить с помощью соответствующей функции загрузки с URL-адресом в качестве параметра. Например, текущая погода в Цинциннати доступна в формате JSON.

Внимательно изучите URL-адрес[1] и его расшифровку:

1. В адресной строке мы запрашиваем данные из поддомена `api` сайта openweathermap.org.
2. Адресная строка указывает город, для которого нужны данные (`q` – это сокращенное обозначение для запроса, которое часто используется в URL-адресах).
3. Параметры адресной строки также означают, что данные будут возвращены в имперском формате, то есть температура будет в градусах Фаренгейта. Замена имперской системы на метрическую предоставит данные о температуре в градусах Цельсия.

Просмотр этих данных из источника OpenWeatherMap – более реалистичный пример работы с данными, найденными в естественных условиях, а не с упрощенными наборами данных, представленными ранее. На момент написания книги файл, возвращенный по этому URL-адресу, выглядел так:

```
{"message":"accurate","cod":"200","count":1,"list":[{"id":
4508722,"name":"Cincinnati","coord":{"lon":-84.456886,"lat":
39.161999},"main":{"temp":34.16,"temp_min":34.16,"temp_max":
34.16,"pressure":999.98,"sea_level":1028.34,"grnd_level":
999.98,"humidity":77},"dt":1423501526,"wind":{"speed":
9.48,"deg":354.002},"sys":{"country":"US"},"clouds":{"all":
80},"weather":[{"id":803,"main":"Clouds","description":"broken
clouds","icon":"04d"}]}]}
```

Этот файл намного легче читать, если он отформатирован с разрывами строки, а объект JSON и структуры массива определены с помощью фигурных скобок:

```
{
  "message": "accurate",
  "count": 1,
  "cod": "200",
  "list": [{
    "clouds": {"all": 80},
    "dt": 1423501526,
    "coord": {
      "lon": -84.456886,
      "lat": 39.161999
    },
    "id": 4508722,
    "wind": {
      "speed": 9.48,
      "deg": 354.002
```

[1] http://api.openweathermap.org/data/2.5/find?q=Cincinnati&units=imperial.

```
    },
    "sys": {"country": "US"},
    "name": "Cincinnati",
    "weather": [{
    "id": 803,
    "icon": "04d",
    "description": "broken clouds",
    "main": "Clouds"
  }],
  "main": {
    "humidity": 77,
    "pressure": 999.98,
    "temp_max": 34.16,
    "sea_level": 1028.34,
    "temp_min": 34.16,
    "temp": 34.16,
    "grnd_level": 999.98
  }
 }]
}
```

Обратите внимание, что в разделах `list` и `weather` видны скобки, обозначающие массив объектов JSON. Хотя массив в этом примере содержит только один элемент, в других случаях API может возвращать данные за несколько дней или варианты данных с нескольких метеостанций.

Пример 12.6. Анализ погодных данных

Первым шагом в работе с данными является их изучение, а затем написание минимального кода для извлечения желаемых данных. В данном случае нас интересует текущая температура. Мы видим, что интересующее нас значение температуры составляет 34.16. Оно обозначено ярлыком `temp` и находится внутри основного объекта, который, в свою очередь, находится внутри массива `list`. Для работы с этой конкретной структурой файла JSON была написана функция `getTemp()`:

```
void setup() {
  float temp = getTemp("cincinnati.json");
  println(temp);
}

float getTemp(String fileName) {
  JSONObject weather = loadJSONObject(fileName);
  JSONArray list = weather.getJSONArray("list");
  JSONObject item = list.getJSONObject(0);
  JSONObject main = item.getJSONObject("main");
  float temperature = main.getFloat("temp");
  return temperature;
}
```

Имя файла `cincinnati.json` передается в функцию `getTemp()` внутри блока `setup()`, и там этот файл загружается. Затем, из-за вложенной структуры файла JSON, необходимо последовательно применить серию функций `getJSONArray()` и `getJSONObject()`, чтобы все глубже и глубже проникать в структуру данных и, наконец, прийти к желаемому числу. Это число сохраняется в переменной `tempersture`, а затем возвращается функцией и присваивается переменной `temp` в `setup()`, после чего выводится на консоль.

Пример 12.7. Объединение методов JSON

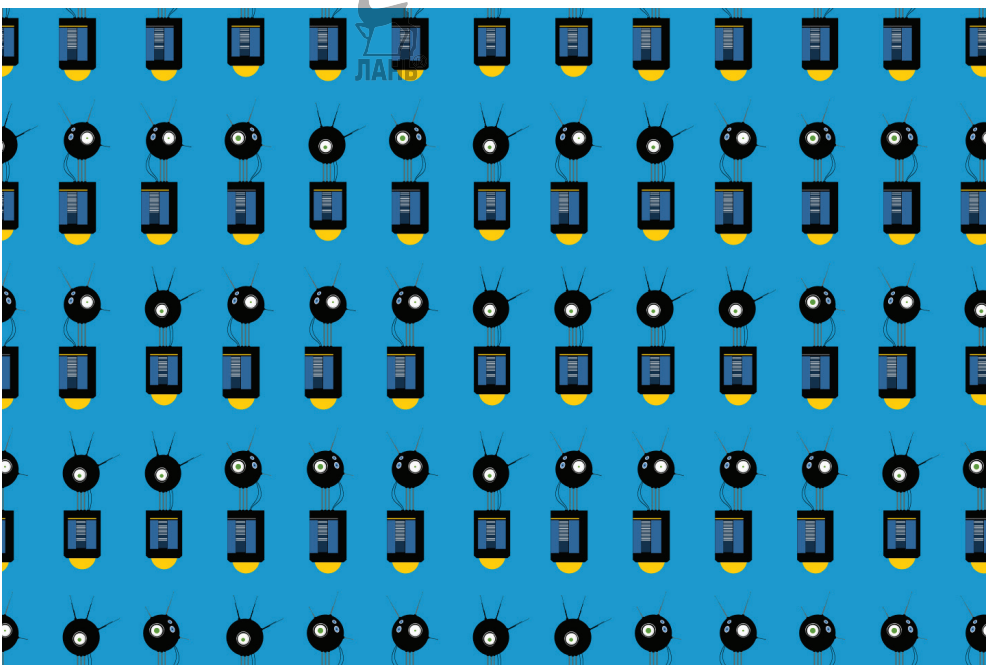
Мы можем обойтись без последовательности переменных JSON, созданных в предыдущем примере, объединив методы `get` в одном месте. Этот пример работает так же, как пример 12.6, но методы связаны оператором точки, а не выполняются по одному с созданием промежуточных объектов:

```
void setup() {
  float temp = getTemp("cincinnati.json");
  println(temp);
}

float getTemp(String fileName) {
  JSONObject weather = loadJSONObject(fileName);
  return weather.getJSONArray("list").getJSONObject(0).
  getJSONObject("main").getFloat("temp");
}
```

Также обратите внимание, как функция `getTemp()` возвращает окончательное значение температуры. В примере 12.6 создается переменная с плавающей запятой для хранения десятичного значения, затем функция возвращает это значение, и его присваивают переменной. В текущем примере данные, созданные методами `get`, возвращаются напрямую, без промежуточных переменных.

Этот пример можно изменить, чтобы получить доступ к большему количеству данных из погодного канала и написать скетч, который отображает данные на экране, а не просто выводит их в консоль. Вы также можете изменить его для чтения данных из другого онлайн-API – вы обнаружите, что данные, возвращаемые многими API-интерфейсами, имеют аналогичный формат.

12.5. Робот 10: данные



Последний пример робота в этой книге отличается от остальных, потому что он состоит из двух частей. Первая часть генерирует файл данных с использованием случайных значений и циклов for, а вторая часть считывает этот файл данных, чтобы вывести на экран армию роботов.

В первом скетче применяются два новых элемента кода: класс `PrintWriter` и функция `createWriter()`. При совместном использовании они создают и открывают файл в папке скетчей для хранения данных, созданных скетчем. В этом примере объект, созданный из `PrintWriter`, называется `output`, а файл – `botArmy.tsv`. В циклах данные записываются в файл путем запуска метода `println()` для объекта вывода. Здесь случайные значения используются для определения, какое из трех изображений робота будет нарисовано в каждой позиции на экране. Для правильного создания файла методы `flush()` и `close()` должны быть выполнены до остановки программы.

Код, который рисует эллипс, предназначен только для предварительного просмотра, чтобы показать расположение координаты на экране, но обратите внимание, что эллипс не будет записан в файл:

```
PrintWriter output;

void setup() {
  size(720, 480);
  // Создаем новый файл
  output = createWriter("botArmy.tsv");
  // Записываем строку заголовка с названиями столбцов
```

```
  output.println("type\tx\ty");
  for (int y = 0; y <= height; y += 120) {
    for (int x = 0; x <= width; x += 60) {
      int robotType = int(random(1, 4));
      output.println(robotType + "\t" + x + "\t" + y);
      ellipse(x, y, 12, 12);
    }
  }
  output.flush(); // Записываем оставшиеся данные в файл
  output.close(); // Закрываем файл
}
```

После запуска этой программы откройте файл `botArmy.tsv` в папке скетчей, чтобы посмотреть, как записываются данные. Первые пять строк этого файла будут выглядеть приблизительно так:


```
type  x   y
3     0   0
1     20  0
2     40  0
1     60  0
3     80  0
```

Первый столбец содержит информацию о том, какое изображение робота использовать, второй столбец – это координата `x`, а третий столбец – координата `y`.

Следующий скетч загружает файл `botArmy.tsv` и использует данные из файла для размещения роботов на экране:

```
Table robots;
PShape bot1;
PShape bot2;
PShape bot3;

void setup() {
  size(720, 480);
  background(0, 153, 204);
  bot1 = loadShape("robot1.svg");
  bot2 = loadShape("robot2.svg");
  bot3 = loadShape("robot3.svg");
  shapeMode(CENTER);
  robots = loadTable("botArmy.tsv", "header");
  for (int i = 0; i < robots.getRowCount(); i++) {
    int bot = robots.getInt(i, "type");
    int x = robots.getInt(i, "x");
    int y = robots.getInt(i, "y");
    float sc = 0.3;
    if (bot == 1) {
      shape(bot1, x, y, bot1.width*sc, bot1.height*sc);
    } else if (bot == 2) {
      shape(bot2, x, y, bot2.width*sc, bot2.height*sc);
    } else {
```

```
      shape(bot3, x, y, bot3.width*sc, bot3.height*sc);
    }
  }
}
```


В более кратком (и гибком) варианте этого скетча в качестве продвинутого подхода используются массивы и метод `rows()` класса `Table`:

```
int numRobotTypes = 3;
PShape[] shapes = new PShape[numRobotTypes];
float scalar = 0.3;

void setup() {
  size(720, 480);
  background(0, 153, 204);
  for (int i = 0; i < numRobotTypes; i++) {
    shapes[i] = loadShape("robot" + (i+1) + ".svg");
  }
  shapeMode(CENTER);
  Table botArmy = loadTable("botArmy.tsv", "header");
  for (TableRow row : botArmy.rows()) {
    int robotType = row.getInt("type");
    int x = row.getInt("x");
    int y = row.getInt("y");
    PShape bot = shapes[robotType - 1];
    shape(bot, x, y, bot.width*scalar, bot.height*scalar);
  }
}
```

Глава 13

Дополнительные возможности языка Processing

В этой книге основное внимание уделяется использованию Processing для интерактивной графики, потому что в этом и заключается его основное предназначение. Однако скетчи могут делать гораздо больше и зачастую являются компонентами проектов, выходящих за рамки экрана компьютера. Например, Processing использовали для управления машинами, создания изображений для художественных фильмов и экспорта моделей для 3D-печати.

За последнее десятилетие Processing сыграл свою роль при создании музыкальных клипов для групп Radiohead и REM, применялся для создания иллюстраций для таких изданий, как Nature и New York Times, для вывода изображений скульптур в галереях, для управления огромными видеостенами, для вязания свитеров и многого другого. У Processing есть такая гибкость благодаря системе библиотек.

Библиотека языка Processing – это специально разработанный код, который расширяет возможности языка за пределы его основных функций и классов. Библиотеки сыграли важную роль в развитии проекта, так как они позволяют разработчикам быстро добавлять новые функции. Поскольку библиотеки – это небольшие автономные проекты, их разработкой и модификацией легче управлять, чем если бы эти функции были интегрированы в основной проект.

Чтобы использовать библиотеку, выберите опцию **Импортировать библиотеку** в меню **Набросок** и выберите в списке нужную библиотеку. При выборе библиотеки в скетч добавляется строка кода, указывающая, какая библиотека будет использоваться с текущим скетчем.

Например, при добавлении библиотеки PDF Export в верхнюю часть скетча добавляется эта строка кода:

```
import processing.pdf.*;
```

В дополнение к библиотекам, включенным в Processing (они называются *базовыми* библиотеками), существует более 100 дополнительных библиотек,

которые можно получить по ссылкам на сайте Processing. Все библиотеки доступны в интернете по адресу http://processing.org/reference/libraries/.

Прежде чем дополнительную библиотеку можно будет импортировать через меню **Набросок**, ее необходимо добавить через Диспетчер библиотек. Выберите опцию **Импортировать библиотеку** в меню **Набросок**, а затем выберите опцию **Добавить библиотеку**, чтобы открыть интерфейс Диспетчера библиотек. Щелкните на строке с описанием библиотеки, а потом нажмите кнопку **Установить**, чтобы загрузить ее на свой компьютер.

Загруженные файлы сохраняются в папке `libraries`, которая находится в вашей папке скетчей. Вы можете найти местоположение своей папки скетчей, открыв меню **Настройки**. Диспетчер библиотек также можно использовать для обновления и удаления библиотек.

Как упоминалось ранее, существует более 100 библиотек Processing, поэтому мы не сможем обсудить их в этой книге все. Мы выбрали несколько библиотек, которые, на наш взгляд, являются достаточно интересными и полезными, чтобы рассказать о них в этой главе.

13.1. Звук

Звуковая библиотека `Sound`, представленная в Processing 3.0, может воспроизводить, анализировать и генерировать (синтезировать) звук. Эту библиотеку необходимо загрузить с помощью Диспетчера библиотек, как описано ранее. (Она не входит в основной дистрибутив Processing из-за своего размера.)

Подобно изображениям, векторным файлам и шрифтам, о которых мы говорили в главе 7, звуковой файл является еще одним вариантом медиафайла, дополняющего скетч Processing. Библиотека `Sound` может загружать файлы различных форматов, включая WAV, AIFF и MP3. После загрузки звукового файла его можно воспроизводить, останавливать, зацикливать или даже искажать с использованием различных «эффектов».

Пример 13.1. Воспроизведение звукового фрагмента

Чаще всего библиотека Sound применяется для воспроизведения звука в качестве фоновой музыки или когда на экране происходит какое-либо событие. Следующий пример основан на примере 8.5. Этот скетч воспроизводит звук, когда фигура касается краев экрана. Файл *`blip.wav`* находится в папке `media`, которую вы загрузили в главе 7 с сайта по адресу http://www.processing.org/learning/books/media.zip.

Как и в случае с другими медиафайлами, объект `SoundFile` определяется в верхней части скетча, загружается в память в блоке `setup()`, после чего его можно использовать в любом месте программы:

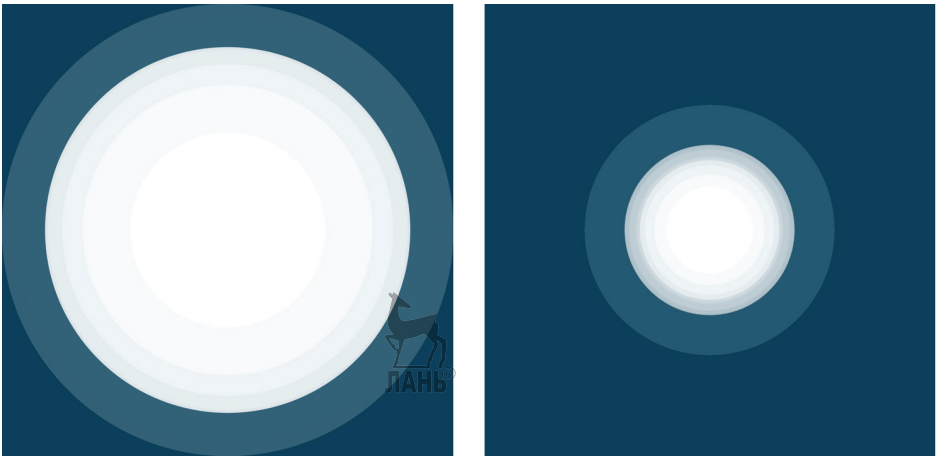
```
import processing.sound.*;

SoundFile blip;
int radius = 120;
float x = 0;
float speed = 1.0;
int direction = 1;

void setup() {
  size(440, 440);
  ellipseMode(RADIUS);
  blip = new SoundFile(this, "blip.wav");
  x = width/2; // Старт в центре
}

void draw() {
  background(0);
  x += speed * direction;
  if ((x > width-radius) || (x < radius)) {
    direction = -direction; // Смена направления
    blip.play();
  }
  if (direction == 1) {
    arc(x, 220, radius, radius, 0.52, 5.76); // Движение вправо
  } else {
    arc(x, 220, radius, radius, 3.67, 8.9); // Движение влево
  }
}
```

Звук воспроизводится каждый раз, когда срабатывает метод `play()`. Этот пример работает хорошо, потому что звук воспроизводится только тогда, когда значение переменной `x` соответствует краю экрана. Если бы звук воспроизводился при каждом проходе через `draw()`, метод перезапускался бы 60 раз в секунду и не успевал бы закончить воспроизведение. Чтобы воспроизвести более длинный звуковой файл во время работы программы, вызовите метод `play()` или `loop()` для этого звука внутри блока `setup()`, чтобы воспроизведение звука запускалось только один раз.

Класс `SoundFile` имеет множество методов для управления воспроизведением звука. Наиболее важными из них являются `play()` для однократного воспроизведения звукового файла, `loop()` для воспроизведения его от начала до конца снова и снова, `stop()` для остановки воспроизведения и `jump()` для перехода к определенному моменту в файле.

Пример 13.2. Прослушивание микрофона

Processing может не только воспроизводить, но и *слушать* звук. Если на вашем компьютере есть микрофон, библиотека `Sound` может слушать через него звуки окружающего мира. Звуки с микрофона можно анализировать, изменять и воспроизводить:



```
import processing.sound.*;

AudioIn mic;
Amplitude amp;

void setup() {
  size(440, 440);
  background(0);
  // Создание объекта аудиовхода и его запуск
  mic = new AudioIn(this, 0);
  mic.start();
  // Создание нового анализатора амплитуды и подключение к входу
  amp = new Amplitude(this);
  amp.input(mic);
}

void draw() {
  // Рисование фона, который затухает в черный цвет
  noStroke();
  fill(26, 76, 102, 10);
  rect(0, 0, width, height);
  // Метод analyze() возвращает значения между 0 и 1,
  // поэтому применяется функция map() для преобразования в диапазон больших значений
  float diameter = map(amp.analyze(), 0, 1, 10, width);
  // Рисование окружности, диаметр которой зависит от громкости
  fill(255);
  ellipse(width/2, height/2, diameter, diameter);
}
```

Измерение *амплитуды* (громкости) звука, воспринимаемого подключенным микрофоном, состоит из двух частей. Класс `AudioIn` используется для получения выборок сигнала с микрофона, а класс `Amplitude` – для измерения сигнала. Объекты обоих классов определяются в верхней части кода и создаются внутри `setup()`.

После создания объекта `Amplitude` (названного здесь `amp`) объект `AudioIn` (названный `mic`) вставляется в объект `amp` с помощью метода `input()`. После этого

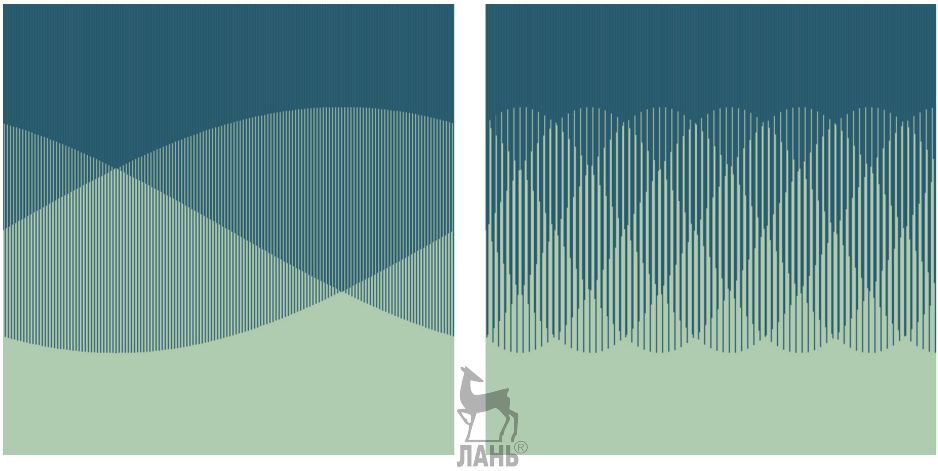
в любой момент можно запустить метод `analysis()` объекта `amp`, чтобы считать амплитуду данных микрофона в программе. В данном примере это делается при каждом проходе `draw()`, а возвращенное значение затем используется для установки размера круга.

Помимо воспроизведения и анализа звука, как показано в последних двух примерах, Processing может напрямую синтезировать звук. Основы звукового синтеза – это формы волны, которые включают синусоидальную волну, треугольную волну и прямоугольную волну.

Синусоидальная волна звучит мягко, прямоугольная волна – резко, а треугольная волна – где-то посередине между ними. Каждая звуковая волна имеет ряд свойств. *Частота*, измеряемая в герцах, определяет высоту тона. *Амплитуда* волны определяет громкость.

Пример 13.3. Генерирование синусоидальной волны

В следующем примере значение `mouseX` определяет частоту синусоидальной волны. По мере того как мышь перемещается влево и вправо, слышимая частота и соответствующая визуализация волн увеличиваются и уменьшаются:



```
import processing.sound.*;

SinOsc sine;
float freq = 400;

void setup() {
  size(440, 440);
  // Создание и запуск генератора синусоидальных колебаний
  sine = new SinOsc(this);
  sine.play();
}
```

```
void draw() {
  background(176, 204, 176);
  // Масштабируем координаты мыши в диапазон от 20Гц до 440Гц
  float hertz = map(mouseX, 0, width, 20.0, 440.0);
  sine.freq(hertz);
  // Визуализируем волну
  stroke(26, 76, 102);
  for (int x = 0; x < width; x++) {
    float angle = map(x, 0, width, 0, TWO_PI * hertz);
    float sinValue = sin(angle) * 120;
    line(x, 0, x, height/2 + sinValue);
  }
}
```

Синусоидальный объект, созданный из класса `SinOsc`, определяется в верхней части кода, а затем создается внутри `setup()`. Как и при работе с семплом, волну нужно проиграть с помощью метода `play()`, чтобы начать генерировать звук. В `draw()` метод `freq()` непрерывно устанавливает частоту сигнала в зависимости от положения мыши по горизонтали.

13.2. Экспорт изображений и PDF

Анимированные изображения, созданные программой Processing, можно превратить в последовательность файлов с помощью функции `saveFrame()`. Когда эту функцию вызывают в конце блока `draw()`, она сохраняет пронумерованную последовательность изображений в формате TIFF выходных данных программы с именами `screen-0001.tif`, `screen-0002.tif` и т. д. в папке скетча.

Эти одиночные файлы кадров можно импортировать в видео- или анимационную программу и сделать из них файл фильма. Вы также можете указать собственное имя и формат файла изображения с помощью такой строки кода:

```
saveFrame("output-####.png");
```

Используйте символ # (решетка), чтобы показать, где будут отображаться числа в имени файла. Они заменяются фактическими номерами кадров при сохранении файлов. Вы также можете указать подпапку для сохранения изображений, что полезно при работе с большим количеством кадров изображений:

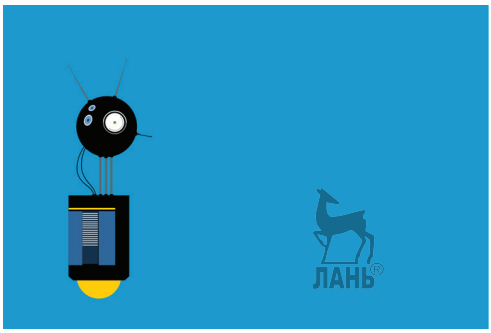
```
saveFrame("frames/output-####.png");
```

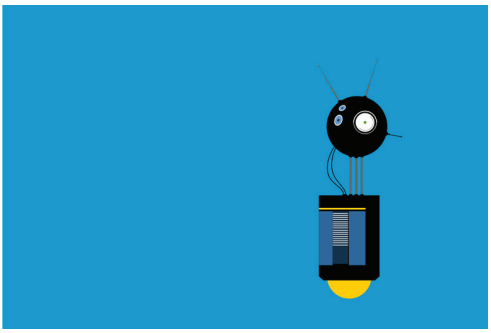
При использовании `saveFrame()` внутри `draw()` каждый кадр сохраняется в новом файле, поэтому будьте осторожны, так как можете быстро заполнить вашу папку скетча тысячами файлов.

Пример 13.4. Сохранение изображений

В этом примере показано, как сохранять изображения в количестве, достаточном для двухсекундной анимации. Код загружает и перемещает файл робота из раздела 7.4. Можете перечитать главу 7, чтобы освежить в памяти инструкции по загрузке файла `robot1.svg` и добавлению его в скетч.

В данном примере программа запускается со скоростью 30 кадров в секунду, а затем завершается через 60 кадров:





```
PShape bot;
float x = 0;

void setup() {
  size(720, 480);
  bot = loadShape("robot1.svg");
  frameRate(30);
}

void draw() {
  background(0, 153, 204);
  translate(x, 0);
  shape(bot, 0, 80);
  saveFrame("frames/SaveExample-####.tif");
  x += 12;
  if (frameCount > 60) {
    exit();
  }
}
```

Processing запишет изображение на основе расширения файла, которое вы используете (расширения `.png`, `.jpg` и `.tif` встроены в базовый дистрибутив, и некоторые платформы могут поддерживать другие расширения). Чтобы найти сохраненные файлы, перейдите в меню **Набросок** > **Показать папку с набросками**.

Изображение в формате `.tif` сохраняется без сжатия, что быстро, но занимает много места на диске. Форматы `.png` и `.jpg` создают файлы меньшего размера, но из-за сжатия обычно требуется больше времени для сохранения, из-за чего скетч работает медленнее.

Если желаемый результат – векторная графика, вы можете записать результат в файлы PDF с более высоким разрешением. Библиотека экспорта PDF позволяет создавать файлы PDF прямо из скетча. Эти файлы векторной графики можно масштабировать до любого размера без потери разрешения, что делает их идеальными для печати – от плакатов и баннеров до целых книг.

Пример 13.5. Рисование в PDF

Этот пример основан на примере 13.4, чтобы нарисовать больше роботов, но не использует движение. В верхней части скетча расположен импорт библиотеки PDF, которая дополняет возможности языка Processing записью файлов PDF.

Скетч из этого примера создает файл PDF с именем `Ex-13-5.pdf`, как указано в третьем и четвертом параметрах `size()`:

```
import processing.pdf.*;
PShape bot;

void setup() {
  size(600, 800, PDF, "Ex-13-5.pdf");
  bot = loadShape("robot1.svg");
}

void draw() {
  background(0, 153, 204);
  for (int i = 0; i < 100; i++) {
    float rx = random(-bot.width, width);
    float ry = random(-bot.height, height);
    shape(bot, rx, ry);
  }
  exit();
}
```

Рисунок не отображается на экране; он записывается непосредственно в файл PDF, который сохраняется в папке скетча. Код в этом примере запускается один раз, а затем завершается в конце `draw()`. Полученный результат показан на рис. 13.1.

В дистрибутив среды разработки Processing включены другие примеры экспорта в PDF. См. раздел **Библиотеки – PDF Export** во встроенных примерах Processing, чтобы ознакомиться с примерами использования других методов.

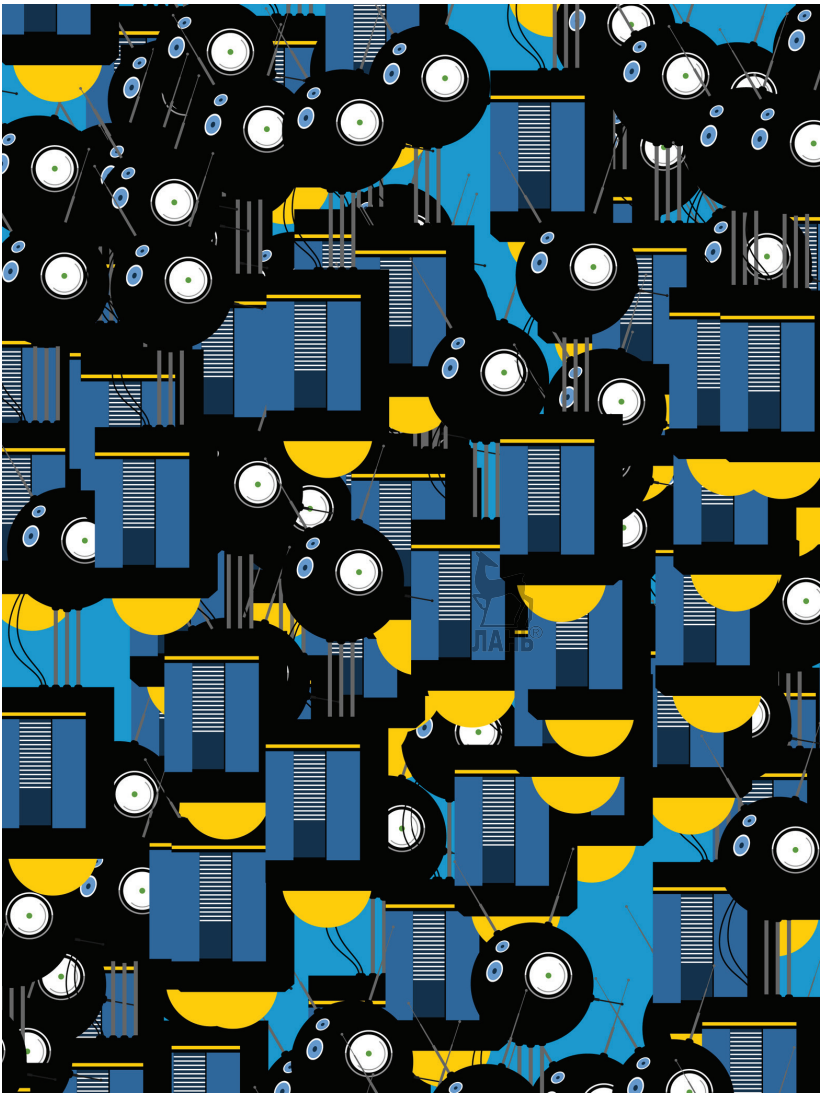


Рис. 13.1 ❖ Экспорт изображения в файл PDF из примера 3.5

13.3. Привет, Ардуино

Arduino – это платформа для создания прототипов электронных устройств с серией плат микроконтроллеров и программным обеспечением для их программирования. У Processing и Arduino долгая общая история; это родственные проекты со многими схожими идеями и целями, хотя и направлены

на разные области. Поскольку они используют один и тот же редактор, среду программирования и схожий синтаксис, между ними легко перемещаться и использовать навыки работы на той и другой платформах.

В этом разделе мы сосредоточимся на считывании данных в Processing с платы Arduino, а затем на визуализации этих данных на экране. Это позволяет использовать новые входные данные в программах на языке Processing, а разработчики в среде Arduino могут отображать данные датчиков в виде графики. Новые источники данных могут быть чем угодно, что подключается к плате Arduino. Эти устройства варьируются от датчика расстояния до компаса или ячеистой сети датчиков температуры.

Двигаясь дальше, мы предполагаем, что у вас есть плата Arduino и базовые практические навыки ее использования. Если это не так, вы можете узнать больше на сайте http://www.arduino.cc и в отличной книге Массимо Банци «Arduino для начинающих волшебников». Изучив основы, вы сможете узнать больше об обмене данными между средой Processing и Arduino в другой выдающейся книге Тома Иго «Умные вещи. Arduino, датчики и сети для связи устройств».

Для обмена данными между скетчем Processing и платой Arduino потребуется помощь библиотеки Serial, которая поддерживает последовательную передачу данных. Последовательная передача – это формат данных, который отправляет по одному байту за раз. В мире Arduino байт (`byte`) – это тип данных, который может хранить значения от 0 до 255; он работает как `int`, но с гораздо меньшим диапазоном. Большие числа отправляются путем разбиения их на последовательность байтов с последующей повторной сборкой.

В следующих примерах мы сосредоточимся на стороне скетча Processing и используем простой код Arduino. Мы визуализируем данные, поступающие с платы Arduino по одному байту за раз. Мы надеемся, что благодаря методам, описанным в этой книге, и сотням примеров проектов Arduino в интернете этого будет достаточно для начала.

Пример 13.6. Считывание показаний датчика

Вместе со следующими тремя примерами кода на языке Processing используется такой код Arduino:

```
// Это код для платы Arduino, а не среды Processing!

int sensorPin = 0; // Выбираем вывод платы для входа
int val = 0;

void setup() {
  Serial.begin(9600); // Открываем последовательный порт
}

void loop() {
  val = analogRead(sensorPin) / 4; // Читаем показания датчика
  Serial.write((byte)val); // Отправляем значение в последовательный порт
  delay(100); // Ждем 100 миллисекунд
}
```

В этом примере кода Arduino следует отметить две важные детали. Во-первых, необходимо подключить датчик к аналоговому входу на выводе 0 на плате Arduino. Вы можете использовать датчик освещенности (также называемый фоторезистором или светочувствительным резистором) или другой аналоговый резистор, например термистор (термочувствительный резистор), датчик изгиба или датчик давления (чувствительный к давлению резистор). Принципиальная схема и чертеж макетной платы с компонентами показаны на рис. 13.2. Затем обратите внимание, что значение, возвращаемое функцией `analogRead()`, делится на 4 перед присвоением переменной `val`. Значения `analogRead()` находятся в диапазоне от 0 до 1023, поэтому мы делим их на 4, чтобы перенести в диапазон от 0 до 255 и отправлять одним байтом.

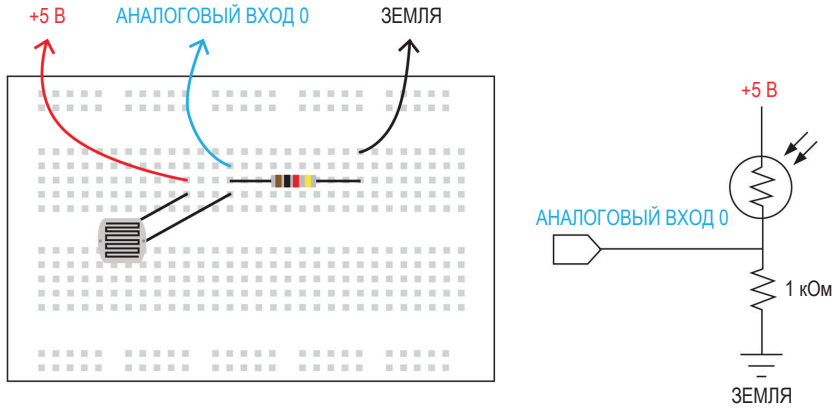


Рис. 13.2 ❖ Подключение датчика освещенности (фоторезистора) к аналоговому входу 0

Пример 13.7. Чтение данных из последовательного порта

Первый пример визуализации показывает, как считывать последовательные данные с платы Arduino и преобразовать эти данные в значения, соответствующие размерам экрана:

```
import processing.serial.*;

Serial port;  // Создаем объект класса Serial
float val;    // Данные, полученные из последовательного порта

void setup() {
  size(440, 220);
  // ВАЖНОЕ ПРИМЕЧАНИЕ:
  // Первый порт, возвращаемый методом Serial.list(),
  // должен принадлежать плате Arduino. Если это не так, раскомментируйте
  // следующую строку, удалив перед ней символы //. Запустите скетч снова,
  // чтобы увидеть список последовательных портов. Затем замените 0
```

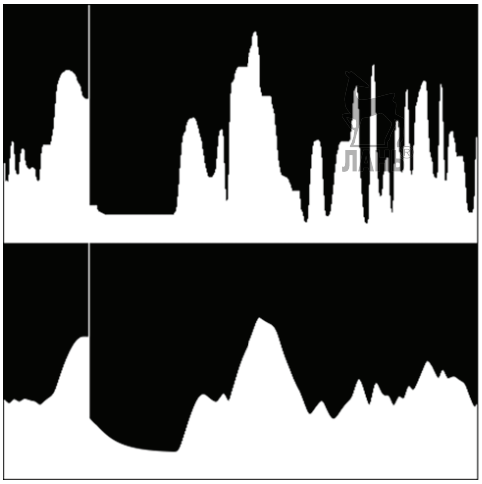
```
  // в квадратных скобках на номер порта, который используется
  // вашей платой Arduino.
  //printArray(Serial.list());
  String arduinoPort = Serial.list()[0];
  port = new Serial(this, arduinoPort, 9600);
}

void draw() {
  if (port.available() > 0) { // Если данные доступны,
    val = port.read();        // читаем их и сохраняем в val
    val = map(val, 0, 255, 0, height);  // Масштабируем значение
  }
  rect(40, val-10, 360, 20);
}
```

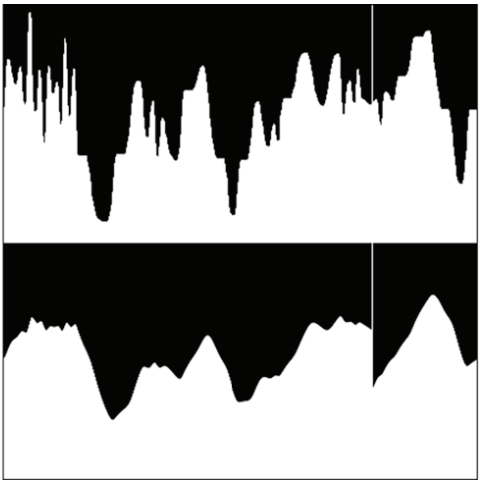
Библиотека `Serial` импортируется в первой строке, а последовательный порт открывается в `setup()`. Заставить ваш скетч Processing взаимодействовать с платой Arduino может быть непросто; это зависит от настроек вашего оборудования. Часто скетч Processing может попытаться установить связь с несколькими устройствами. Если код не работает с первого раза, внимательно прочтите комментарий в `setup()` и следуйте инструкциям.

В блоке `draw()` значение передается в программу с помощью метода `read()` объекта `Serial`. Программа считывает данные из последовательного порта только тогда, когда доступен новый байт. Метод `available()` проверяет, готов ли новый байт, и возвращает количество доступных байтов. Эта программа написана так, что каждый раз при проходе через `draw()` будет считываться один новый байт. Функция `map()` преобразует входящее значение из исходного диапазона от 0 до 255 в диапазон от 0 до высоты экрана; в этой программе от 0 до 220.

Пример 13.8. Визуализация потока данных

Теперь, когда мы организовали непрерывное получение данных, их можно визуализировать в более интересном формате. Значения, поступающие непосредственно с датчика, часто бывают ошибочными, и их полезно сгладить путем усреднения. Необработанный сигнал от датчика освещенности представлен в верхней половине следующего рисунка, а сглаженный сигнал – в нижней половине:



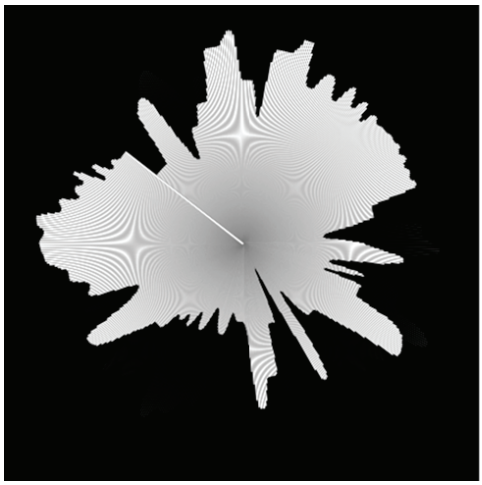


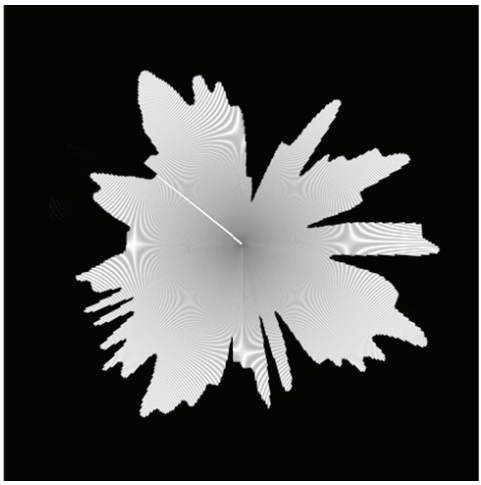
```
import processing.serial.*;

Serial port; // Создаем объект класса Serial
float val; // Данные, полученные из последовательного порта
int x;
float easing = 0.05;
float easedVal;

void setup() {
  size(440, 440);
  frameRate(30);
  String arduinoPort = Serial.list()[0];
  port = new Serial(this, arduinoPort, 9600);
  background(0);
}

void draw() {
  if ( port.available() > 0) { // Если данные доступны,
    val = port.read();         // читаем их и сохраняем в val
    val = map(val, 0, 255, 0, height/2); // Масштабируем значения
  }
  float targetVal = val;
  easedVal += (targetVal - easedVal) * easing;
  stroke(0);
  line(x, 0, x, height); // Черная линия
  stroke(255);
  line(x+1, 0, x+1, height); // Белая линия
  line(x, 220, x, val); // Исходное значение
  line(x, 440, x, easedVal + 220); // Усредненное значение
  x++;
  if (x > width) {
    x = 0;
  }
}
```

Подобно примерам 5.8 и 5.9, в этом скетче используется прием отставания. Каждый новый байт с платы Arduino устанавливается как целевое значение, вычисляется разница между текущим значением и целевым значением, а текущее значение перемещается ближе к целевому. Вы можете менять значение переменной `easing`, чтобы выбрать подходящую степень сглаживания, применяемого к входным значениям.

Пример 13.9. Еще один способ визуализации данных

Этот пример имитирует экран дисплея радиолокатора. Значения попрежнему считываются с платы Arduino, но визуализируются в виде круговой диаграммы с использованием функций `sin()` и `cos()`, представленных ранее в примерах 8.12, 8.13 и 8.15:





```
import processing.serial.*;

Serial port; // Создаем объект класса Serial
float val; // Данные, полученные из последовательного порта
float angle;
float radius;

void setup() {
  size(440, 440);
  frameRate(30);
  strokeWeight(2);
  String arduinoPort = Serial.list()[3];
  port = new Serial(this, arduinoPort, 9600);
  background(0);
}

void draw() {
```

```
  if ( port.available() > 0) { // Если данные доступны,
    val = port.read(); // читаем их и сохраняем в val
    // Преобразуем значения, чтобы получить радиус
    radius = map(val, 0, 255, 0, height * 0.45);
  }
  int middleX = width/2;
  int middleY = height/2;
  float x = middleX + cos(angle) * height/2;
  float y = middleY + sin(angle) * height/2;
  stroke(0);
  line(middleX, middleY, x, y);
  x = middleX + cos(angle) * radius;
  y = middleY + sin(angle) * radius;
  stroke(255);
  line(middleX, middleY, x, y);
  angle += 0.01;
}
```

Переменная `angle` постоянно обновляется, чтобы перемещать линию, рисующую текущее значение, по кругу, а переменная `val` определяет длину движущейся линии, чтобы установить ее расстояние от центра экрана. После одного оборота по кругу значения начинают записываться поверх предыдущих данных.

Мы очень рады возможности совместного использования Processing и Arduino – так мы соединяем миры программного обеспечения и электроники. В отличие от показанных в этой книге примеров, связь может быть двунаправленной. Элементы на экране также могут влиять на поведение платы Arduino. Это означает, что вы можете использовать программу Processing в качестве интерфейса между вашим компьютером и двигателями, динамиками, освещением, камерами, датчиками и многими другими устройствами, которыми можно управлять с помощью электрического сигнала. Посетите сайт http://www.arduino.cc для получения дополнительной информации об Arduino.

Приложение 1

Рекомендации по программированию

Программный код – это разновидность письменного текста. Как и все виды письменного текста, код программы подчиняется определенным правилам. Для сравнения мы вспомним некоторые правила английского (и русского) языка, о которых вы, вероятно, давно не задумывались, потому что они являются вашей второй натурой. Наиболее очевидные и незаметные правила говорят о том, что надо писать слева направо и ставить пробелы между словами. Более явными являются правила написания слов, использования заглавных букв в именах людей и расстановки знаков препинания. Если мы нарушим одно или несколько из этих правил при написании электронного письма другу, сообщение все равно останется понятным для него. Например, текст с ошибками «привет Бен. как дела?» мы понимаем так же хорошо, как фразу «Привет, Бен, как дела?» Однако гибкость в правилах написания разговорного языка не распространяется на программирование. Поскольку вы пишете, чтобы общаться с компьютером, а не с другим человеком, вам нужно быть более точным и аккуратным. Один неуместный символ способен изменить поведение программы на противоположное.

Processing пытается подсказать вам, где вы допустили ошибку, и угадать, в чем ошибка. Если в вашем коде есть грамматические (синтаксические) ошибки, то после нажатия кнопки **Запустить** область сообщений становится красной, и Processing пытается выделить строку кода, в которой предполагается наличие ошибки. Строка кода с ошибкой часто находится на одну строку выше или ниже выделенной строки, хотя в некоторых случаях это совсем не так. Текст в области сообщений пытается быть полезным и указывает на потенциальную проблему, но иногда сообщение выглядит слишком загадочным и недоступным для понимания. Новичка эти сообщения об ошибках могут расстраивать. Поймите, что Processing – это простая среда разработки, которая пытается быть полезной, но имеет ограниченное понятие о том, что вы пытаетесь сделать.

Длинные сообщения об ошибках содержат более подробную информацию, и иногда прокрутка этого текста в консоли может дать подсказку. Кроме того, Processing может находить только одну ошибку за раз. Если в вашей программе много ошибок, вам нужно продолжать запускать программу и исправлять их по очереди.

Пожалуйста, внимательно прочитайте и, при необходимости, перечитайте следующие разделы, чтобы научиться писать красивый и правильный код.

Функции и параметры

Программы состоят из множества небольших частей, которые сгруппированы вместе, чтобы образовать более крупные структуры. Аналогичным образом устроен разговорный язык: слова сгруппированы в фразы, которые объединяются в предложения и дальше в абзацы. Идея в коде та же, но составные части имеют разные имена и ведут себя по-разному. Две важные части языка программирования – функции и параметры. *Функции* – это основные строительные блоки программы на языке Processing. *Параметры* – это значения, определяющие поведение функции.

Рассмотрим такую функцию, как `background()`. Как следует из названия, она используется для установки цвета фона рабочего окна программы. Функция имеет три параметра, определяющих цвет. Эти числа определяют красный, зеленый и синий компоненты цвета для определения составного цвета. Например, следующий код рисует синий фон:

```
background(51, 102, 153);
```

Внимательно рассмотрите эту единственную строчку кода. Ключевые детали – это круглые скобки после имени функции, в которых заключены числа, запятые между каждым числом и точка с запятой в конце строки. Точка с запятой в программировании играет роль точки в конце предложения. Это означает, что текущая инструкция завершена, поэтому компьютер может искать начало следующей. Для успешного выполнения кода в инструкции должны присутствовать абсолютно все необходимые детали. Сравните предыдущую строку со следующими тремя ошибочными версиями той же строки:

```
background 51, 102, 153; // Ошибка! Отсутствуют скобки
background(51 102, 153); // Ошибка! Отсутствует запятая
background(51, 102, 153) // Ошибка! Отсутствует точка с запятой
```

Компьютер неумолим даже к малейшим упущениям или отклонениям от того, чего он ожидал. Если вы будете внимательно относиться к малейшим деталям, у вас будет меньше ошибок. Но если вы допустите ошибку при вводе кода, что периодически случается с любым из нас, это не беда. Processing сообщит вам о проблеме, и когда она будет устранена, программа начнет работать нормально.

Выделение цветом

Среда разработки Processing выделяет цветом разные части каждой программы. Слова, которые являются частью языка Processing, написаны синим

и оранжевым цветами, чтобы отличить их от частей программы, написанных вами. Слова, уникальные для вашей программы, такие как имена переменных и функций, отображаются черным цветом. Основные символы, такие как квадратные и круглые скобки и символы /, >, <, *, также черные.

Комментарии

Комментарии – это заметки, которые вы пишете себе (или другим людям) внутри кода. Вы должны использовать их, чтобы разъяснить простыми словами, что делает код, и предоставить дополнительную информацию, такую как название и автор программы. Комментарий начинается с двух косых черт (//) и продолжается до конца строки:

```
// Это однострочный комментарий
```

Вы можете написать многострочный комментарий, который начинается с символов /* и заканчивается символами */. Например:

```
/ * Этот комментарий
```

продолжается более чем на одной строке

```
* /
```

Если комментарий оформлен правильно, цвет текста станет серым. Вся область комментариев становится серой, поэтому вы можете четко видеть, где она начинается и где заканчивается.

Прописные и строчные буквы

Processing отличает прописные буквы от строчных и поэтому различает слова «Привет» и «привет». Если вы попытаетесь нарисовать прямоугольник с помощью функции `rect()` и вместо этого напишете `Rect()`, программа не запустится. Вы можете увидеть, распознает ли Processing ваш предполагаемый код, проверив цвет текста.

Оформление и форматирование кода

Processing позволяет гибко определять стиль оформления вашего кода. Для программы на языке Processing безразлично, если вы напишете:

```
rect(50, 20, 30, 40);
```

или же:

```
rect (50, 20, 30, 40);
```

или же:

```
rect ( 50,20,
  30, 40) ;
```

Однако в ваших интересах сделать код легкочитаемым. Это становится особенно важным по мере увеличения длины кода. Аккуратное форматирование делает структуру кода читаемой и понятной, а небрежное форматирование часто скрывает проблемы. Выработайте привычку писать чистый код. Есть много разных способов хорошо отформатировать код, и какой из них вы выберете – это вопрос вашего личного предпочтения.

Консоль

Консоль – это нижняя область среды разработки, предназначенная для вывода текстовых сообщений программы. Вы можете отправлять сообщения в консоль с помощью функции `println()`. Например, следующий код печатает сообщение, за которым идет текущее время:

```
println("Привет, Processing.");
println("Текущее время " + hour() + ":" + minute());
```

Консоль важна для просмотра того, что происходит внутри ваших программ во время их работы. В консоль полезно выводить значения переменных, чтобы вы могли отслеживать их, проверять, происходят ли ожидаемые события, и определять, где у программы возникла проблема.

Шаг за шагом

Мы рекомендуем писать несколько строк кода за раз и почаще запускать код, чтобы ошибки не накапливались без вашего ведома. Даже самые амбициозные программы пишут поэтапно. Разбейте свой проект на более простые подпроекты и запускайте их по одному, чтобы у вас в итоге получилось много небольших успехов, а не рой ошибок. Если у вас есть ошибка, попробуйте изолировать область кода, в которой, по вашему мнению, кроется проблема. Постарайтесь думать об исправлении ошибок как о решении загадки или головоломки. Если вы зашли в тупик или расстроились, обязательно сделайте перерыв, чтобы прочистить голову, или попросите помощи у друга. Иногда ответ находится прямо у вас под носом, но, чтобы найти его, нужен свежий взгляд со стороны.

Приложение 2

Типы данных

Есть разные категории данных. Для примера возьмем данные, которые содержатся в удостоверении личности. Там указана дата рождения, есть слова для хранения имени и города проживания человека. Есть также изображение (фото) и часто встречается согласие быть донором органов в виде решения «да/нет». В Processing предусмотрены разные типы для хранения различных данных. Мы рассмотрели все основные типы данных на протяжении книги, а здесь приводим краткий перечень:

Тип	Описание	Диапазон значений
`int`	Целое число (обычные числа)	От –2 147 483 648 до 2 147 483 647
`float`	С плавающей запятой	От –3.40282347E+38 до 3.40282347E+38
`boolean`	Логическое значение	Истина или ложь
`char`	Одиночный символ	A–z, 0–9 и символы
`String`	Последовательность символов (строка)	Любая буква, слово, предложение и т. д.
`PImage`	Изображение PNG, JPG или GIF	N/A
`PFont`	Используйте функцию `createFont()` или инструмент Create Font, чтобы создать шрифты для применения с Processing	N/A
`PShape`	Файл SVG	N/A

Ориентировочно, число типа `float` должно содержать около четырех десятичных знаков после запятой. Если вы отсчитываете или делаете небольшие шаги, то должны использовать значение типа `int`, а затем, возможно, масштабировать его с помощью числа с плавающей запятой, если это необходимо в дальнейшем.

На самом деле разнообразие типов данных больше, чем упомянуто здесь, но мы рассмотрели наиболее полезные для программ на языке Processing. Фактически, как упоминалось в главе 10, существует бесконечное количество типов данных, потому что каждый новый класс представляет собой уникальный тип данных.

Приложение 3

Порядок операций

Когда программа содержит математические вычисления, каждая операция выполняется в строго определенном порядке. Это гарантирует, что код всегда будет выполняться одинаково. В целом такой подход не отличается от правил, принятых в арифметике или алгебре, но в программировании есть и другие менее известные операторы.

В следующей таблице операторы, расположенные выше, имеют приоритет выполнения относительно операторов, расположенных ниже, поэтому операция в круглых скобках будет выполняться первой, а присваивание – последним:

Операция	Символ	Примеры
Круглые скобки	`()`	`a * (b + c)`
Инкремент, декремент	`++ -- !`	`a++ --b !c`
Умножение	`* / %`	`a * b`
Сложение, вычитание	`+ -`	`a + b`
Отношения	`> < <= >=`	`if (a > b)`
Эквивалентность	`== !=`	`if (a == b)`
Логическое И	`&&`	`if (mousePressed && (a > b))`
Логическое ИЛИ	`||`	`if (mousePressed || (a > b))`
Присваивание	`= += -= *= /= %=`	`a = 44`

Приложение 4

Область видимости переменных

Правило *области видимости переменной* определяется просто: переменная, созданная внутри блока (код, заключенный в фигурные скобки { и }), существует только внутри этого блока. Это означает, что переменная, созданная внутри блока `setup()`, может использоваться (видна) только внутри этого блока, и аналогично переменная, объявленная внутри блока `draw()`, может использоваться только внутри `draw()`. Исключением из этого правила является переменная, объявленная вне функций `setup()` и `draw()`. Эти переменные можно использовать как в `setup()`, так и в `draw()` (или внутри любой другой создаваемой вами функции). Считайте область за пределами `setup()` и `draw()` неявным блоком кода. Мы называем такие переменные *глобальными*, потому что их можно использовать в любом месте программы. Переменную, которая используется только в одном блоке, называют *локальной*. Ниже приведены два примера кода, которые иллюстрируют эту концепцию. Первый пример:

```
int i = 12; // Объявляем глобальную переменную i = 12

void setup() {
  size(480, 320);
  int i = 24; // Объявляем локальную переменную i = 24
  println(i); // В консоль будет выведено число 24
}

void draw() {
  println(i); // В консоль будет выведено число 12
}
```

И второй пример:

```
void setup() {
  size(480, 320);
  int i = 24; // Объявляем локальную переменную i = 24
}

void draw() {
  println(i); // ОШИБКА! Переменная i является локальной в блоке setup()
  }
```

Предметный указатель

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Кейси Риас, Бен Фрай

Знакомство с программированием на языке Processing

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Яценков В. С.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 15,76. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Знакомство с программированием на языке Processing

Изучите компьютерное программирование с помощью Processing — простого языка, предназначенного для создания рисунков, анимации и интерактивной графики. Курсы программирования обычно начинаются с теории, но эта книга сразу же переходит к творческим и занимательным проектам. Она идеально подходит для всех, кто хочет научиться программировать, и служит простым введением в компьютерную графику для людей, которые уже имеют некоторые навыки программирования.

Написанная создателями языка Processing, эта книга шаг за шагом проведет вас через процесс обучения и поможет понять основные принципы программирования. Присоединяйтесь к тысячам любителей, студентов и профессионалов, которые открыли для себя эту бесплатную образовательную платформу.

Прочитав эту книгу, вы:

- быстро изучите основы программирования, от переменных до объектов;
- освоите азы компьютерной графики;
- познакомитесь со средой разработки Processing;
- научитесь создавать интерактивную графику на простых примерах;
- научитесь применять методы визуализации данных;
- соедините между собой электронную плату Arduino и программу Processing;
- добавите звук к своим графическим творениям.

Версия Processing 3.0 и цветные иллюстрации!

Кейси Риас *— профессор факультета медиадизайна Калифорнийского университета в Лос-Анджелесе. Его программы, гравюры и инсталляции были представлены на многочисленных персональных и групповых выставках в музеях и галереях США, Европы и Азии.*

Бен Фрай *— руководитель Fathom, консалтинговой компании по дизайну и программному обеспечению, расположенной в Бостоне. Он получил докторскую степень в группе «Эстетика+вычисления» медиалаборатории Массачусетского технологического института, где проводил исследования на стыке таких областей, как информатика, статистика, графический дизайн и визуализация данных в качестве средства понимания информации.*

В 2001 году К. Риас и Б. Фрай создали язык Processing.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК "Галактика"
books@alians-kniga.ru

Make:
makezine.com

ДМК ПРЕСС
издательство
www.дмк.рф

ISBN 978-5-97060-950-7