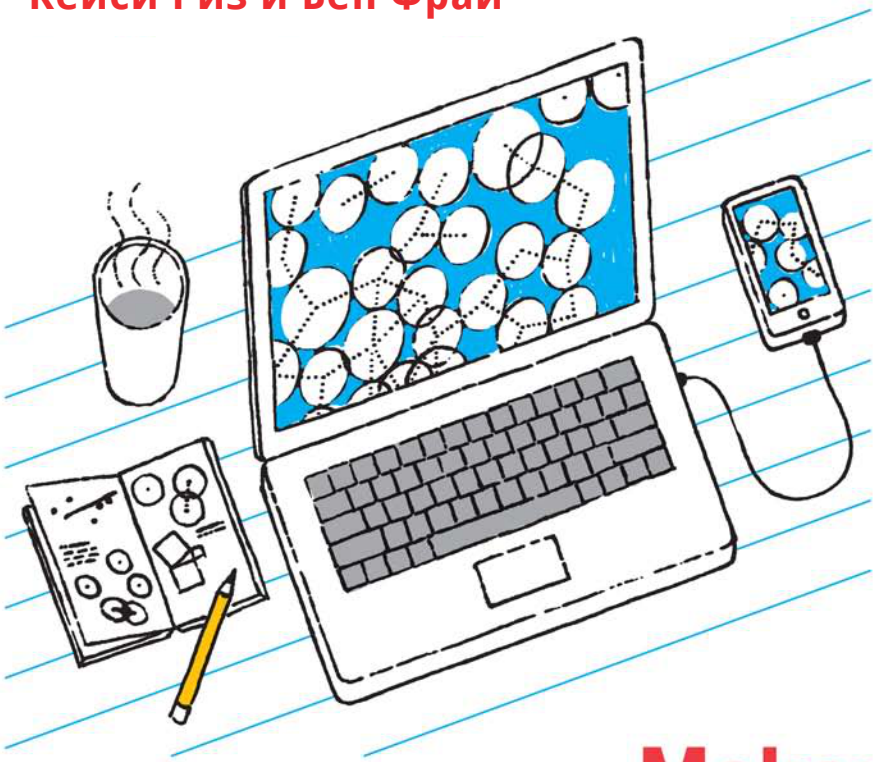
Make: PROJECTS

A HANDS-ON INTRODUCTION TO MAKING INTERACTIVE GRAPHICS

Учимся программировать вместе с Processing

Кейси Риз и Бен Фрай



O'REILLY

Make: makezine.com

Open Source/Graphics Programming

Make: PROJECTS

Учимся программировать вместе с **Processing**

Научитесь программировать вместе с Processing, простым языком, позволяющим вам легко создавать изображения, анимацию и интерактивную графику. Обычно курсы по обучению программированию начинаются с теории, но эта книга позволит вам сразу же приступить к созданию увлекательных проектов. Книга отлично подходит для тех, кто хочет научиться программировать, а также содержит простое введение в мир компьютерной графики для тех, кто уже владеет некоторыми навыками программирования.

Написанная создателями Processing, эта книга шаг за шагом поможет вам понять основные принципы программирования. Присоединяйтесь к тысячам профессионалов, студентов и просто увлечённых людей, уже открывших для себя эту свободную платформу.

С помощью этой книги вы сможете:

» Быстро изучить основы программирования - от объявления переменных до работы с объектами
» Понять основы компьютерной графики
» Познакомиться со средой разработки Processing
» Создавать интерактивную графику на простых примерах
» Использовать свободную аппаратную платформу Arduino для управления графическими объектами в среде Processing

Кейси Риз профессор факультета Design Media Arts Калифорнийского университета в Лос-Анджелесе и выпускник лаборатории MIT Media Laboratory. Программное обеспечение, разработанное Ризом, было представлено на множестве выставок в США, странах Европы и Азии.

Бен Фрай разработчик, программист и писатель, живущий в Кэмбридже, штат Массачусетс. Получил свою докторскую степень в MIT Media Laboratory. Вместе с Кейси Ризом он разработал Processing, который принёс им Золотую Нику на фестивале Prix Ars Electronica в 2005.

Make: makezine.com

O'REILLY

US $19.99 CAN $24.99

ISBN: 978-1-449-37980-3

"Создать компьютерную программу теперь стало так же легко, как включить компьютер, набрать пару строк кода и получить в ответ “Hello”! Сейчас для этого нужно чуть более 500 страниц руководства. Не более. Эта небольшая книга Бена и Кейси научит вас рисовать линии, треугольники и окружности с помощью языка программирования Processing и создавать программу за несколько минут. Они сделали создание компьютерных программ более удобным для человека и благодаря их усилиям программирование теперь стало проще и легче - а это немалое достижение.

— Джон Маэда (John Maeda),
президент Rhode Island School of Design

"Книга *"Учимся программировать вместе с Processing"* - это не только непосредственное введение в мир программирования - это еще и отличный способ хорошо провести время! Работа с этой книгой напоминает кружок “Умелые руки” - только для взрослых. Даже если вы никогда не интересовались программированием - приобретите эту книгу, потому что вы непременно заинтересуетесь."

— Марк Аллен (Mark Allen),
основатель и директор Machine Project

"Эта книга представляет прекрасное руководство для тех, кто хочет окунуться в мир программирования интерактивной графики. Принцип learning by doing (обучение через практику), который использует эта книга, отлично подходит для художников и дизайнеров, которых отталкивает традиционный подход к обучению программированию. Низкая цена книги, а также тот факт, что среда разработки Processing Development Environment является средой с открытым исходным кодом делает Processing оптимальным выбором для студентов.

— Джиллиан Крэмптон Смит (Gillian Crampton Smith),
профессор дизайна университета IUAV University of Venice

"Processing коренным образом изменил способ обучения программированию, и это послужило одним из основных факторов успеха Arduino."

— Массимо Банци (Massimo Banzi),
сооснователь Arduino

"Книга *"Учимся программировать вместе с Processing"* Кейси Риза и Бена Фрая - это прекрасное руководство, открывающее творческим людям возможности программирования интерактивной графики. Риз и Фрай точны и непосредственны, однако, как и все художники, они не боятся быть необычными и оригинальными. Это делает их уникальную методику обучения по-настоящему мощной.

— Холли Уиллис (Holly Willis), директор академических программ,
Institute for Multimedia Literacy, School of Cinematic Arts, USC

Учимся программировать вместе с Processing

Кейси Риз и Бен Фрай

перевод с английского Александры Мишутиной

По всем вопросам, замечаниям, предложениям обращайтесь на e-mail:

mishutinaalex@mail.ru

вконтакте: http://vk.com/id32542885

facebook: http://www.facebook.com/sasha.mishutina

O'REILLY

BEIJING • CAMBRIDGE • FARNHAM • KÖLN • SEBASTOPOL • TAIPEI • TOKYO

Getting Started with Processing
by Casey Reas and Ben Fry

Published by O'Reilly Media, Inc.
1005 Gravenstein Highway North, Sebastopol, CA 95472

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Print History: June 2010: First Edition.

Editor: Brian Jepson
Development Editor: Tom Sgouros
Production Editor: Rachel Monaghan
Copyeditor: Nancy Kotary
Proofreader: Rachel Monaghan
Compositor: Nancy Kotary
Indexer: Ron Strauss
Illustrations: Casey Reas and Ben Fry
Cover Designer: Karen Montgomery

ISBN: 978-1-449-37980-3
[M]

Содержание

Предисловие vii

1/Введение 1
Скетчинг и моделирование 2
Гибкость 3
Гиганты 4
Фамильное древо 5
Присоединяйтесь 6

2/Начинаем программировать 7
Ваша первая программа 8
Инструменты 10
Сохранение 11
Поделитесь достижениями 12
Примеры и справка 13

3/Рисование 15
Простые фигуры 16
Последовательность рисования 22
Свойства фигур 23
Цвет 26
Фигуры произвольной формы 30
Комментарии 33
Робот 1: Рисование 34

4/Переменные 37
Объявление переменных 39
Переменные в Processing 40
Немного математики 40
Циклы 42
Робот 2: Переменные 49

5/Интерактивность 5
Двигаем объекты мышью 53
Преобразование чисел 58
Кликаем мышью 60
Положение курсора на экране 64
Клавиатурный ввод 68
Робот 3: Интерактивность 74

6/Мультимедиа. 77
Изображения 78
Шрифты 83
Фигуры 86
Робот 4: Мультимедиа 89

7/Движение 91
Скорость и направление 92
Промежуточные изображения 95
Случайные числа 97
Таймеры 99
Движение по кругу 100
Перемещение, вращение, масштабирование 104
Робот 5: Движение 113

8/Функции 115
Как это работает 116
Создание функций 118
Возвращаемые значения 124
Робот 6: Функции 126

9/Объекты 129
Объекты и классы 130
Робот 7: Объекты 138

10/Массивы 141
Объявление массива 144
Повторения и массивы 147
Массивы из объектов 150
Робот 8: Массивы 153

11/Расширения 157
3D 158
Экспорт изображений 164
Привет, Arduino! 168
Сотрудничество 175

A/Полезные советы 177

B/Типы данных 183

C/Порядок операций 185

D/Область видимости переменных 187

Предметный указатель 189

Предисловие

Мы создали Processing, чтобы упростить программирование интерактивной графики. Мы часто думали о том, как сложно писать подобные программы на языках программирования, которые мы обычно используем (C++ и Java) и как увлекательно и легко было создавать такие программы на языках программирования нашего детства (Logo и BASIC). При создании Processing на нас повлиял Design By Numbers (DBN), язык, созданный нашим научным констультантом, Джоном Маэда; то время мы занимались поддержкой этого языка и обучением на нем.

Processing появился на свет весной 2001 на сеансе мозгового штурма, устроенном на листе бумаги. Нашей целью было придумать способ, которым мы могли бы создавать прототипы интрективных и полноэкранных приложений, которые мы разрабатывали. Мы искали более удобный способ для проверки и демонстрации наших идей, чем трудоемкая реализация их на C++ или простое обсуждение. Другая наша цель заключалась в создании языка, который было бы удобно использовать для обучения программированию студентов, занимающихся дизайном и который даст возможность студентам технических специальностей более легкий инструмент для работы с графикой. Получившийся сплав позволил усовершенствовать традиционную методику создания компьютерных программ. Мы сделали основной акцент на интерактивной графике, а не на типах данных и текстовом консольном выводе.

Processing разрабатывался довольно долго: с августа 2002 по апрель 2005 он находился в стадии альфа-версии, а затем распространялся в стадии бета-версии до ноября 2008. В течение этого времени он постоянно использовался для обучения и создания программ тысячами людей по всему миру. На протяжении этого времени язык, среда разработки и способ подачи материала непрерывно совершенствовались.

Какие-то из наших первоначальных идей остались прежними, какие-то изменились. Мы разработали систему программных расширений, библиотек (library), позволивших людям расширять возможности Processing в невероятных, неизведанных направлениях. (В настоящее время существует более 100 библиотек). 29 ноября 2008 года мы выпустили версию 1.0. Это была первая стабильная версия языка, явившаяся результатом 7 лет работы.

Сейчас, за 9 лет своего существования Processing вышел за рамки своих первоначальных задач и мы изучаем возможность его применения в других областях. Поэтому эта книга написана для новой аудитории - программистов-любителей, увлеченных людей и для всех тех, кто хотел бы испытать возможности программирования на Processing, не изучая массивных справочников. Мы надеемся, что эта книга вдохновит вас на занятие программированием и вы получите удовольствие от работы с ней. Эта книга - всего лишь старт.

Мы, Кейси и Бен, ведем корабль под названием Processing в течение последних 9 лет, однако мы не преувеличим, если скажем, что Processing является результатом совместных усилий сообщества людей. От написания библиотек расширений до публикации кода и помощи начинающим, сообщество пользователей Processing расширяло границы его применения, выводя его за рамки первоначальных задач. Без поддержки сообщества пользователей Processing не стал бы таким, какой он есть сейчас.

Как организована эта книга

Главы в этой книге организованы следующим образом:

» Глава 1, "Hello": Начальная информация о языке Processing.

» Глава 2, "Начинаем программировать": Создание вашей первой программы.

» Глава 3, "Рисование": Изображаем простые фигуры.

» Глава 4, "Переменные": Хранение, изменение и повторное использование данных.

» Глава 5, "Интерактивность": Управление программой с помощью мыши и клавиатуры.

» Глава 6, "Мультимедиа": Загружаем и выводим на экран изображения, шрифты, векторные файлы.

» Глава 7, "Движение": Двигаем фигуры.

» Глава 8, "Функции": Создание новых фрагментов кода.

» Глава 9, "Объекты": Создание модулей, содержащих переменные и функции.

» Глава 10, "Массивы": Упрощаем работу с переменными.

» Глава 11, "Расширения": Информация о 3D, экспорте изображений и получении данных с платы Arduino.

Целевая аудитория

Эта книга написана для всех тех, кто хотел бы иметь под рукой простое и краткое введение в мир компьютерного программирования, для тех, кто хотел бы создавать изображения и несложные интерактивные программы. Эта книга поможет понять тысячи примеров кода на Processing, свободно распространяемых в интернете, а также более подробную документацию о Processing. Книга *"Учимся программировать вместе с Processing"* не является учебником по программированию, как можно было бы подумать, она всего лишь дает вам старт. Эта книга написана для подростков, их бабушек и дедушек, увлеченных людей и всех остальных.

Эта книга пригодится и людям с опытом программирования, которые хотели бы изучить основы программирования интерактивной компьютерной графики. Книга *"Учимся программировать вместе с Processing"* содержит технические приемы, применимые для создания компьютерных игр, анимации и интерфейсов.

Условные обозначения

В книге используются следующие условные обозначения:

» *Курсив*: используется для выделения новых терминов, имён файлов; также используется в тексте книги для указания переменных, имён функций, типов данных и ключевых слов.

» `Моноширинный шрифт`: используется для написания текстов программ.

ПРИМЕЧАНИЕ: используется для обозначения примечаний.

Использование примеров кода

Определённо можно сказать, что эта книга предназначена для облегчения выполнения вами своей работы. В общем, вы можете использовать представленные в книге примеры кода для своих проектов и документации к ним. Вы не должны связываться с нами для получения разрешения, за исключением случаев, когда вы используете значительную часть кода. Например, написание программы, в которой используются небольшие части кода, не требует разрешения. Но продажа или распространение CD-ROM с примерами из книг издательства O'Reilly требует разрешения. Цитирование и публикация примеров кода из этой книги при ответе на вопросы не требует разрешения.Но включение значительной части кода из текста книги в документацию вашего программного продукта требует разрешения.

Мы будем признательны за указание книги в качестве образца используемой литературы, но не настаиваем на этом. Обычно указываются название, автор, издатель и ISBN. Например: *"Учимся программировать вместе с Processing*, Кейси Риз и Бен Фрай. Все права защищены. 2010. Casey Reas and Ben Fry, 978-1-449-37980-3."

Если вы считаете, что используете примеры кода не совсем честно, свяжитесь с нами по адресу *permissions@oreilly.com* для получения разрешения.

Как с нами связаться

Комментарии и вопросы, касающиеся этой книги, можно направлять издателю по адресу:

O'Reilly Media, Inc.
1005 Gravenstein eighway North
Sebastopol, CA 95472
800-998-9938 (в США или Канаде)
707-829-0515 (международные и местные звонки)
707-829-0104 (факс)

У нас есть собственная веб-страница, посвященная этой книге, где мы публикуем список опечаток, примеры кода и другую полезную информацию. Страница доступна по адресу: *http://oreilly.com/catalog/0636920000570*

Задать технический вопрос или прислать комментарий о книге, можно отправив электронное письмо по адресу: *bookquestions@oreilly.com*

За более подробной информацей о наших книгах, конференциях, ресурсных центрах и о сети O'Reilly Network, заходите на наш сайт:

» *http://oreilly.com*

Благодарности

Благодарим Брайана Джепсона (Brian Jepson) за его поддержку, неиссякаемую энергию и профессионализм. Также благодарим Нэнси Котари (Nancy Kotary), Рейчел Монаган (Rachel Monaghan) и Сумиту Мукхерджи (Sumita Mukherji), которые изящно довели эту книгу до финишной линии.

Спасибо Тому Сгоуросу (Tom Sgouros) за подробную редакцию и Дэвиду Хамфри (David Humphrey) за подробную техническую рецензию.

Мы не представляем эту книгу без *"Getting Started with Arduino"* (издательство O'Reilly), от Массимо Банци (Massimo Banzi). Прекрасная книга Массимо легла в основу этой книги.

На протяжении долгих лет небольшие группы энтузиастов вкладывали свои силы и энергию в Processing. Благодарим Флориан Дженнет (Florian Jenett) за тщательный поиск информации в интернете и умение создавать прекрасные программы, Андреаса Шлегеля (Andreas Schlegel) за создание инфраструктуры для построения и документирования библиотек расширения и Дена Шиффмана (Dan Shiffman) за написание отличных примеров и организацию онлайн-тьюториалов. На протяжении этого времени многие другие люди поддерживали Processing. Среди них Карстен Шмидт (Karsten Schmidt), Эрик Джордан (Eric Jordan) и Джонатан Фейнберг (Jonathan Feinberg). Помощь в администрировании и поддержке активности дискуссионного форума оказали PhiLho, Cedric и antiplastik.
Мы восхищены невероятной работой тех, кто создавал библиотеки расширения и делился ими с сообществом. Спасибо вам всем! Особого упоминания заслуживают библиотеки расширения GLGraphics and GSVideo Андреса Колубри (Andres Colubri), Minim sound library Дэмиена Ди Феде (Damien Di Fede) и toxiclibs - обширное и вдохновляющее собрание библиотек расширения Карстена Шмидта (Karsten Schmidt).

Выпуск версии Processing 1.0 состоялся благодаря поддержке университета Майами (Miami University) и Oblong Industries. Armstrong Institute for Interactive Media Studies университета Майами спонсировал Oxford Project - серию семинаров по разработке Processing. Эти встречи стали возможны благодаря большой работе, которую проделала Ира Гринберг (Ira Greenberg). Результатом этих четырехдневных встреч в Оксфорде, штат Огайо и Питтсбурге, штат Пенсильвания стал выпуск версии Processing 1.0 в ноябре 2008. Компания Oblong Industries спонсировала деятельность Бена Фрая по разработке Processing в течение лета 2008 и это явилось решающим фактором выпуска версии 1.0.

Эта книга написана на основе опыта преподавания Processing в Калифорнийском университете. Чендлер Маквильямс (Chandler McWilliams) оказал большую помощь при организации этих занятий. Кейси благодарит студентов факультета Design Media Arts Калифорнийского университета за их энергию и энтузиазм. Его ассистенты приняли участие в разработке методики обучения Processing. Снимаем шляпу перед Тацуей Сайто (Tatsuya Saito), Джоном Хоуком (John Houck), Тайлером Адамсом (Tyler Adams), Аароном Сигелом (Aaron Siegel), Кейси Альт (Casey Alt), Андресом Колабри (Andres Colubri), Михаэлем Контопулосом (Michael Kontopoulos), Дэвидом Эллиотом (David Elliot), Кристо Аллегра (Christo Allegra), Питом Хоуксом (Pete Hawkes) и Лорен Маккарти (Lauren McCarthy).

Мы благодарим Синана Эссиоглу (Sinan Ascioglu) за Open Processing - прекрасный ресурс для публикации свободно распространяемых примеров кода на Processing.

Следствием создания языка Processing и распространения открытой системы Web, предполагающей привлечение большого числа пользователей к совершенствованию и многократной выверке информационного материала, явился открытый язык программирования Processing.js. Троекратное ура в честь Джона Резига (John Resig), Эла Макдональда (Al MacDonald), Дэвида Хамфри (David Humphrey) и центра по разработке открытых технологий колледжа Сенека (Seneca College's Centre for Development of Open Technology (CDOT)), Роберта О'Рурка (Robert O'Rourke) и фонда Mozilla Foundation.

Основав Aestetics and Computational Group (1996-2002) в лаборатории MIT Media Lab Джон Маэда сделал этот проект возможным.

1/Введение

Язык программирования Processing предназначен для написания графических, анимационных и интерактивных программ. Принцип написания программ заключается в следующем: вы пишете всего одну строчку кода и на экране появляется кружок. Добавляете еще несколько строк - и ваш кружок начитает двигаться вслед за курсором. Ещё строка - и кружок будет менять цвет при нажатии на кнопку мыши. Мы называем этот процесс скетчинг. Вы пишете одну строчку, потом ещё одну и так далее. В результате перед вами окажется готовая программа.

Как правило, изучение программирования сосредоточено в первую очередь на теории и описании структуры программ. Все, что касается графики - интерфейсы, анимация - преподносится на десерт после того, как вы справитесь с основным блюдом - изучением алгоритмов и методов программирования, занимающим несколько недель. Мы знаем много случаев, когда наши друзья брались за подобный курс, но бросали его после первой лекции или после долгой бессонной ночи перед сдачей зачета. Всех их объединяло то, что интерес к созданию программ, работающих на их цели, угасал, поскольку они не видели пути от изучения основ программирования до создания желаемой программы.

Processing дает возможность научиться программированию через написание интерактивных программ. Существует множество путей освоения программирования, но наиболее интересными и мотивирующими для студентов являются те, которые позволяют получить немедленный результат. Возможность получения быстрого результата сделала Processing одним из популярных способов обучения программированию. В следующих разделах мы расскажем о том, что такое скетчинг, о возможности визуализации ваших приложений и о сообществе пользователей Processing.

Скетчинг и моделирование

Скетчинг - это способ мыслить. Он эффективен и похож на игру. Основная цель - оценить большое количество идей за короткий промежуток времени. Мы начинаем наши проекты с эскиза на бумаге, а затем переносим результат в код. Для анимационных и интерактивных проектов мы составляем эксизы наподобие комиксов с заметками. После создания программного эскиза лучшие идеи отбираются для прототипа проекта (Рис. 1.1). Все это - циклический процесс создания, тестирования и совершенствования, продвигающий нас к цели.

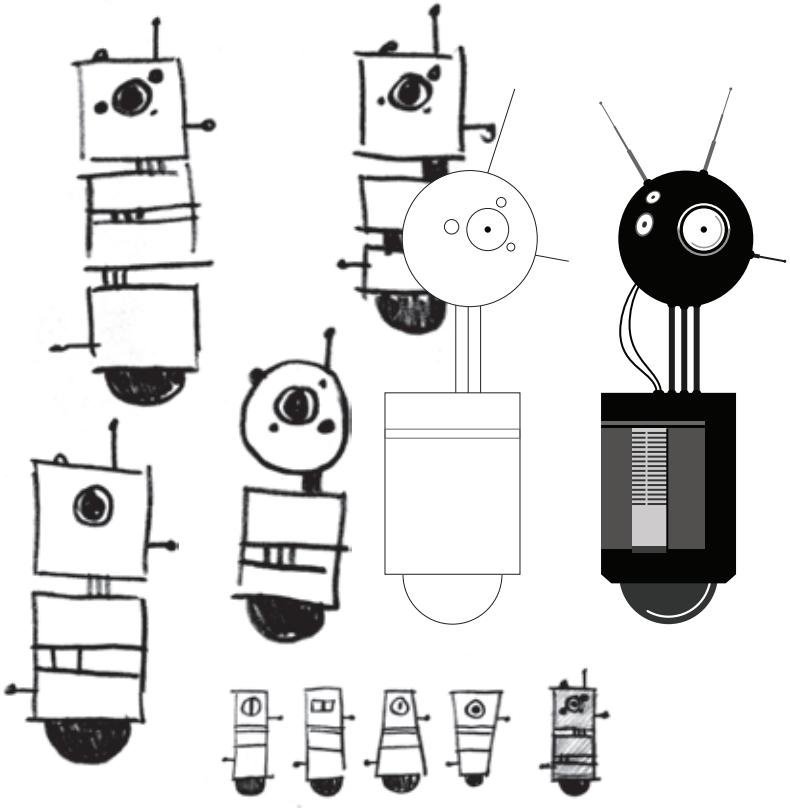


Рис. 1-1. Когда рисунок переносится с блокнота на экран, открываются новые возможности.

Гибкость

Как и любое программное обеспечение, Processing состоит из большого количества компонентов, работающих вместе. Он может быть использован как для простых поделок, так и для подробного исследования. Программа на Processing может составлять от одной до нескольких тысяч строк кода, поэтому вы всегда сможете улучшить и расширить его функционал. Существует более 100 библиотек расширения, позволяющих применять Processing для обработки звука, исследований в области машинного зрения и технических расчетов (Рис. 1-2).

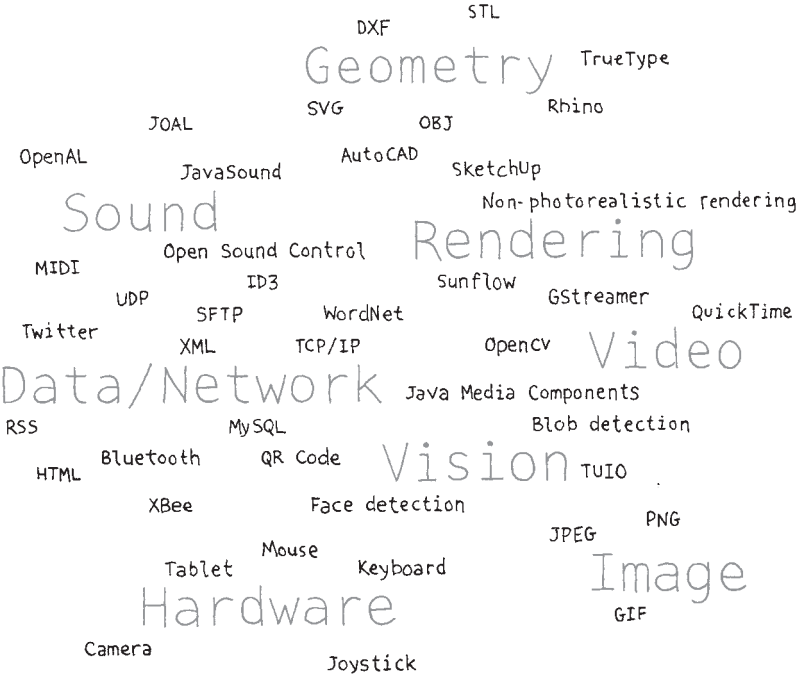


Рис. 1-2. Спектр технологий, охватываемый Processing

Гиганты

История создания изображений с использованием компьютеров начинается в 1960-х годах (Рис 1-3). В нашей повседневной жизни мы используем труд огромного числа людей, работавших до нас. Мы как бы опираемся на плечи гигантов, и Processing в этом не стал исключением. Титанами Processing были мыслители из числа инженеров, художников, архитекторов, маркетологов. Взгляните на Sketchpad (1963) Айвена Сазерленда (Ivan Sutherland), Dynabook (1968) Алана Кея (Alan Kay), на художников из книги "Художник и компьютер" (Artist and Computer)[1] под ред. Рут Ливитт (Ruth Leavitt) - издательство Harmony Books, 1976. Архивы ассоциации ACM SIGGRAPH содержат увлекательный обзор истории компьютерной графики и програмного обеспечения.

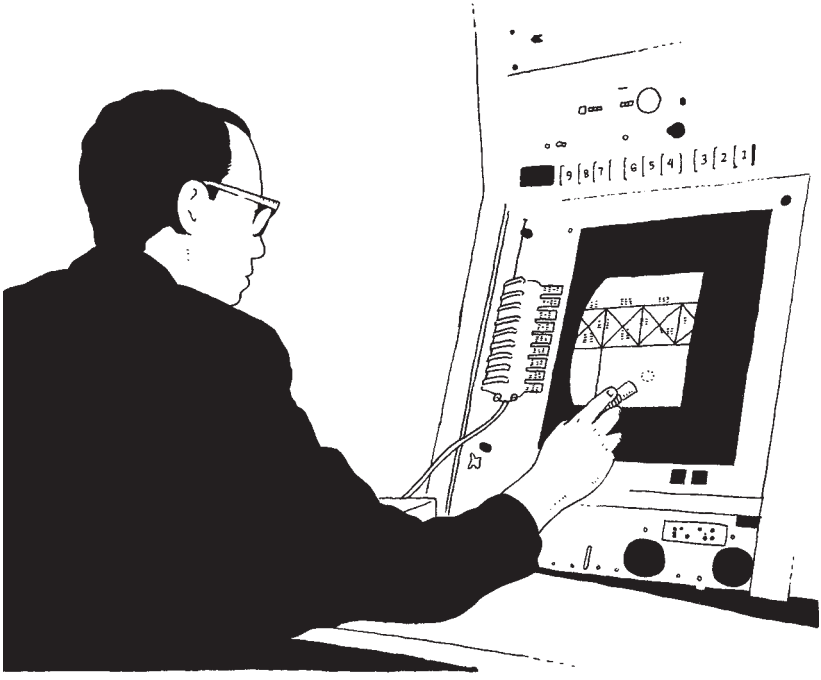


Рис. 1-3. П*рекрасные идеи и неординарные исследователи вдохновили нас на создание Processing.*

[1] *http://www.atariarchives.org/artist/*

Фамильное древо

Языки программирования, как и человеческие языки, объединяются в группы родственных языков. Processing - это диалект языка программирования под названием Java; он имеет почти тот же синтаксис, но дополнен специальными командами для работы с графикой и внешними устройствами (Рис. 1-4). Графические объекты Processing родственны системе PostScript, послужившей основой для формата PDF, и OpenGL (графическая библиотека для написания приложений, использующих 3D-графику). Processing содержит в себе особенности многих языков программирования и поэтому может послужить хорошим введением в программирование на других языках с использованием иных инструментов разработки.

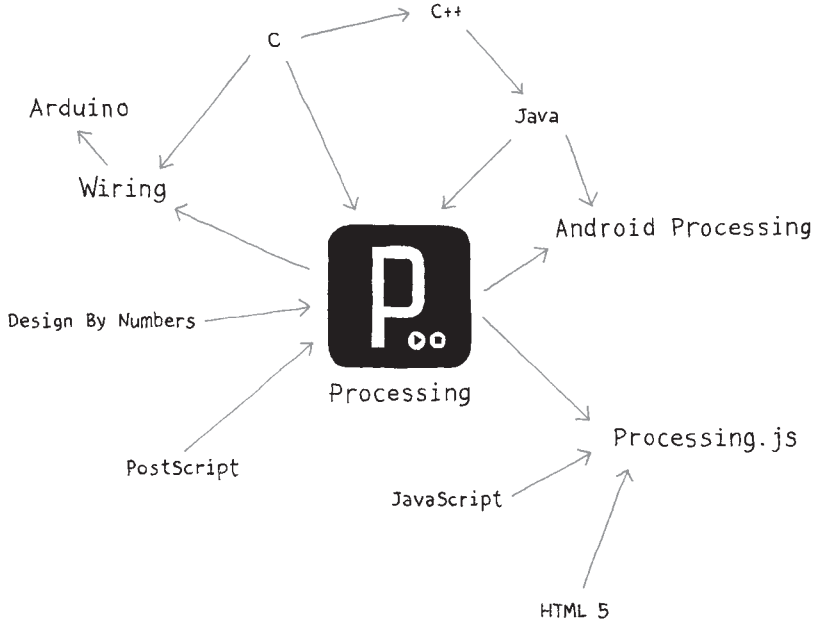


Рис. 1-4. Processing имеет большое количество родственных языков и инструментов разработки.

Присоединяйтесь

Тысячи людей по всему миру ежедневно используют Processing. Вы можете бесплатно скачать Processing и присоедениться к ним. Кроме того, вы можете модифицировать исходный код Processing для своих целей. Проект Processing разработан по принципу FLOSS (free/libre/open source software), и мы будем рады, если вы поделитесь своими знаниями и интересными проектами на сайте Processing.org и на многих других сайтах, посвященных особенностям работы в Processing. Ссылки на них вы найдете на сайте Processing.org.

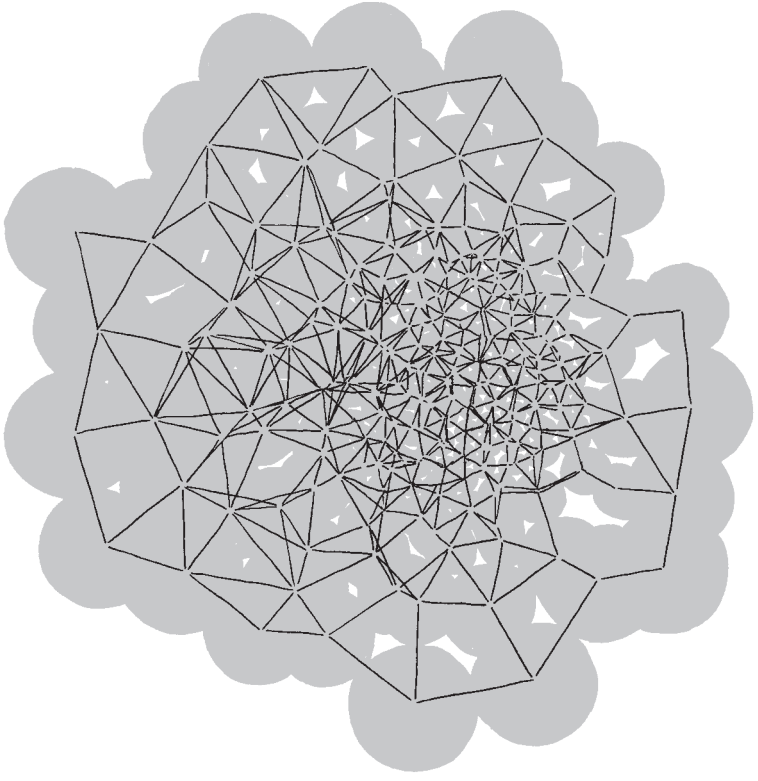


Рис. 1-5. Processing поддерживается тысячами людей из разных уголков мира. Эмблема Processing показывает, как люди взаимодействуют друг с другом.

2/Начинаем программировать

Чтобы получить максимум из этой книги, вам придется не только читать ее, но и экспериментировать. Невозможно освоить язык программирования просто читая о нем - нужно использовать его. Для начала, скачайте Processing и напишите свой первый скетч.

Посетите сайт http://processing.org/download и выберите свою операционную систему: Mac, Windows или Linux.

- Для Windows вы получите .zip файл. Кликните по нему дважды и перетащите в нужное место на жестком диске. Это может быть папка Program Files или Рабочий стол - главное, чтобы папка processing была извлечена из .zip-архива. Двойным щелчком по файлу processing.exe запустите приложение.

- Пользователи Mac OS X получат файл образа диска (.dmg). Перетащите иконку Processing в папку Applications. Если вы работаете на чужом компьютере и не можете внести изменения в папку Applications, перетащите ее на рабочий стол. Затем запустите приложение двойным щелчком.

- Для Linux это будет файл .tar.gz, знакомый большинству пользователей Linux. Загрузите файл в директорию home, откройте окно терминала и наберите:

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

  (Где xxxx в конце названия файла это номер версии). Это создаст папку processing-1.0 или нечто вроде того. Затем переключитесь на эту папку:

  ```
  cd processing-xxxx
  ```

и запустите его:

```
./processing
```

В случае успешного выполнения, вы увидите главное окно (Рис. 2-1). В каждом случае установка осуществляется по-разному, и если у вас возникли трудности с запуском программы, посетите страницу troubleshooting для устранения неполадок: http://wiki.processing.org/index.php/Troubleshooting.

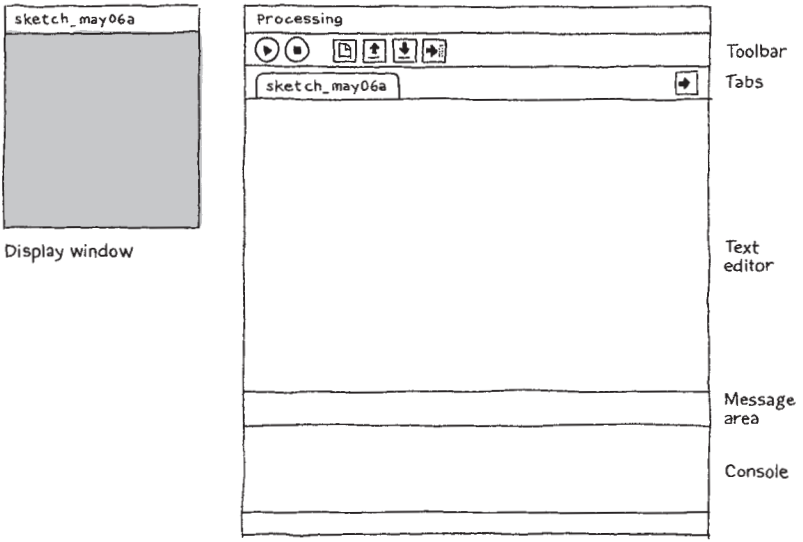


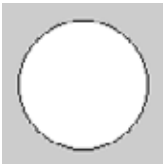
Рис. 2-1. Среда разработки Processing.

Ваша первая программа

Итак, перед вами среда разработки Processing. Выглядит она скромно: большую часть окна занимает текстовый редактор, над ним находится ряд кнопок - это панель инструментов. Под текстовым редактором располагается область вывода сообщений, ниже - консоль. В области вывода сообщений отображаются сообщения в режиме реального времени, консоль использ-уется для особых задач.

Пример 2-1: Рисуем эллипс

В текстовом редакторе наберите следующее:



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

Эта строка кода обозначает: “нарисуй эллипс с центром на расстоянии 50 пикселей от левой границы и 50 пикселей от верхней, с шириной и длиной в 80 пикселей.” Кликните на кнопку Run, которая выглядит так:

Если вы сделаете все правильно, вы увидите эллипс. Однако если допущена ошибка, область вывода сообщений станет красной и вы увидите сообщение об ошибке. Убедитесь, что вы точно скопировали код: числа должны быть заключены в скобки и разделены запятыми, строка заканчивается точкой с запятой.

Вы должны быть очень внимательны к синтаксису в вашем коде, это одна из трудностей для начинающего программиста. Среда Processing не всегда догадается, что вы имеете в виду и может быть очень требовательна к пунктуации. Немного попрактиковавшись, вы привыкнете к этому.

А теперь перейдем к более интересному скетчу.

Пример 2-2: Рисуем круги

Удалите текст предыдущего примера и попробуйте этот:

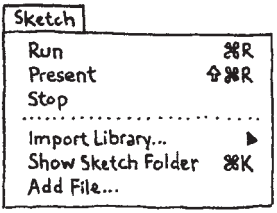
```
void setup() {
  size(480, 120);
  smooth();
}

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

Эта программа создаст окно шириной 480 пикселей и 120 пикселей высотой, и нарисует круг в месте расположения курсора. При нажатой кнопке мыши цвет круга меняется на черный. Позднее мы объясним все подробности этого кода. А сейчас запустите программу, подвигайте мышью и нажмите на кнопку, чтобы посмотреть как это работает.

Инструменты

До сих пор мы использовали только кнопку Run, но вы, вероятно, уже догадались, что делает кнопка Stop, расположенная рядом:

Вы также можете использовать меню Sketch или комбинацию клавиш Ctrl-R (Cmd-R для Mac) для быстрого запуска команды Run. Под кнопкой Run в меню Sketch находится кнопка Present, которая удаляет пустые строки в вашем сектче для более удобного просмотра текста:



Вы можете запускать команду Present из панели инстументов сочетанием клавиш Shift и Run.

Сохранение

Еще одна важная для нас команда это Save. Она выглядит как стрелка, направленная вниз:

Вы сможете найти ее также в меню File. По умолчанию, все ваши программы сохраняются в папке sketchbook, откуда вы сможете быстро извлечь их, кликнув на кнопку Open на панели инструментов. Кнопка Open выглядит как стрелка, направленная вверх и после запуска этой команды вы увидите список ваших скетчей из папки “sketchbook” и список примеров программ, загруженных вместе в программой Processing:

Сохраняйте копии ваших скетчей на разных этапах их написания. Перед тем, как попробовать что-то новое, сохраните копию вашей программы под особым именем, чтобы иметь возможность вернуться к ранней версии. Это поможет вам, если что-то пойдет не так.
С помощью команды Show Sketch Folder в меню Sketch вы можете узнать место расположения ваших скетчей на жестком диске.

Для создания нового скетча нажмите кнопку New на панели инструментов:

Скетч в текущем окне будет изменен на пустой скетч. Удерживание кнопки Shift во время нажатия кнопки New создаст новый скетч в отдельном окне, точно так же, как выбор команды File→New. Кнопка Open работает по такому же принципу.

Поделитесь достижениями

Еще одна функция Processing - возможность поделиться своей работой. Кнопка Export на панели инструментов:

упакует ваш код в папку applet которую можно загрузить на сервер (Рис 2-2). После завершения загрузки папка applet откроется на рабочем столе. Файл PDE - это исходный код, JAR - это программа, HTML-файл это веб-страница, и GIF - это изображение, отображаемое в браузере во время загрузки программы. Двойной клик по файлу index.html откроет созданную веб-страницу с вашим скетчем.

Applet

Name	Date Modified	Size	Kind
Ex_02_02.jar	Today	228 KB	Java JAR File
Ex_02_02.java	Today	4 KB	Java Source File
Ex_02_02.pde	Today	4 KB	Processing Source File
index.html	Today	4 KB	HTML Document
loading.gif	10/20/09	4 KB	Graphics Interchange Format (GIF)

Рис. 2-2 Папка applet, содержащая загруженный скетч.

Примечание: содержимое папки applet обновляется каждый раз, когда вы используете команду Export. Убедитесь, что папка сохранена перед тем, как вносить изменения в файл HTML или другие файлы внутри папки.

Рядом с кнопкой Export в меню File вы найдете кнопку Export to Application. Команда Export to Application создает, по вашему выбору, приложение для Mac, Windows, и/или Linux. Это преобразует ваш скетч в полноценное приложение, запускаемое по двойному щелчку мыши (Рис. 2-3).

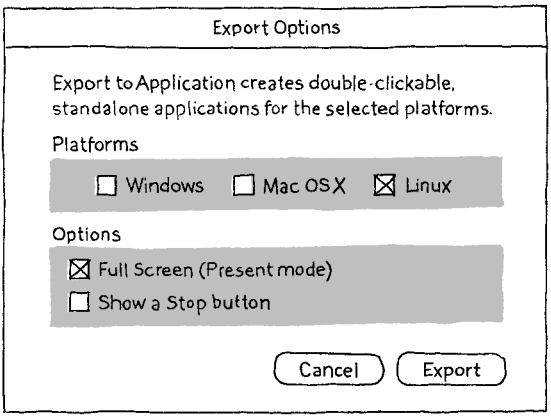


Рис. 2-3. Меню E*xport to Application.*

Вы можете запустить команду Export to Application одновременным удерживанием кнопки Shift и нажатием кнопки Export из панели инструментов.

Примеры и справка

Обучение программированию на Processing невозможно без работы с многочисленными примерами кода: запуска, модификации, прерывания и улучшения кода до тех пор, пока не сформируется нечто новое. Приняв это во внимание, мы включили в Processing десятки примеров, демонстрирующих различные возможности среды разработки. Выберите Exampes в меню File или кликните по иконке Open, чтобы открыть список примеров программ. Примеры сгруппированы по категориям в зависимости от содержащихся функций, таких как фигуры, движение, изображения. Выберите интересующую тему и поэкспериментируйте с примером.

Если вы заметили незнакомый вам фрагмент кода, выделенный оранжевым (это значит, что он принадлежит языку Processing), выделите его имя и нажмите “Find in Reference” в меню Help. Или нажмите на правую кнопку мыши (Ctrl-клик на Mac) и выберите в появившемся меню Find in Reference. Это откроет справку по выделенному фрагменту кода в вашем браузере. Справка также доступна по адресу *http://www.processing.org/reference/*. Справка Processing подробно и с примерами описывает каждый элемент кода. Образцы программ из справки значительно короче (обычно 4 или 5 строк) и воспринимаются лучше, чем длинный код из папки Examples. Рекомендуем держать справку по программе открытой, когда вы читаете эту книгу или программируете. Вы сможете быстро найти нужную функцию по названию или по алфавиту; иногда это удобнее, чем делать поиск по тексту в окне браузера.

Справка рассчитана, в первую очередь, на новичков; надеемся, что смогли сделать ее удобной и понятной. Мы признательны людям, отмечавшим ошибки в справке и сообщавшим нам о них на протежении последних лет. Если у вас есть соображения по поводу улучшения справки или вы нашли ошибку, перейдите по ссылке вверху веб-страницы и сообщите нам.

3/Рисование

Много лет назад создание рисунков на экране компьютера принципиально не отличалось от рисования на бумаге. Но со временем появилась возможность делать рисунки движущимися и интерактивными. Перед тем, как сделать этот шаг, мы начнем с основ.

Экран монитора состоит из элементов под названием *пиксели*. Каждый пиксель имеет свою позицию на экране, которую можно определить координатами. В языке Processing x-координата это расстояние, отсчитываемое от левого края окна, и y-координата - это расстояние отсчитываемое от верхнего края. Координаты пикселя записываются так: (x, y). Итак, если разрешение экрана 200x200 пикселей, координаты верхнего левого пикселя будут (0, 0), центрального (100, 100), нижнего правого (199, 199). Вы могли бы спросить: почему пискели нумеруются от 0 до 199 вместо от 1 до 200? Ответ заключается в коде: с числами в диапазоне от 0 проще производить арифметические вычисления. Мы покажем это позже.

Создать окно и отобразить в нем рисунок можно с помощью ф*ункций*. Функции являются основным инструментом для постоения программы на Processing. Результат функции зависит от введенных *параметров*. Например, почти любая программа на Processing содержит функцию *size*(), для задания размеров окна. (Если не применить функцию *size*(), то будет использовано окно размером 100x100 пикселей.)

Пример 3-1: Рисуем окно

Функция *size*() имеет 2 параметра: первый устанавливает ширину окна, второй - высоту. Чтобы получить окно в 800 пикселей в ширину и 600 в высоту, наберите:

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

Запустите строку кода и рассмотрите результат. Введите другие значения и посмотрите, что получится. Попробуйте очень малые числа и очень большие - превышающие размер экрана.

Пример 3-2: Рисуем точку

Чтобы выделить пиксель в окне, мы используем функцию *point*(). Она имеет 2 параметра, определяющие положение: x-координату и y-координату. Чтобы изобразить небольшое окно и точку в его центре, заданном координатами (240, 60), наберите:

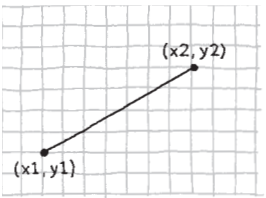
```
size(480, 120);
point(240, 60);
```

Попробуйте написать программу, которая рисует точку в каждом углу и в центре окна. Расположите точки рядом, чтобы получить горизонтальные или вертикальные линии.

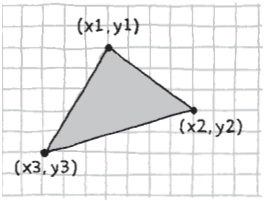
Простые фигуры

Processing включает набор функий для рисования простых фигур (Рис. 3-1). Фигуры вроде линий можно комбинировать для создания более сложных рисунков типа листа или лица.

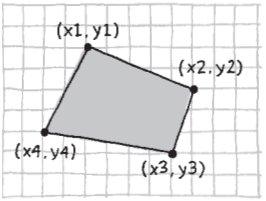
Для линии нам потребуется 4 параметра: 2 для начала линии и 2 для конца.



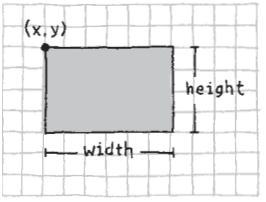

```
line(x1, y1, x2, y2)
```



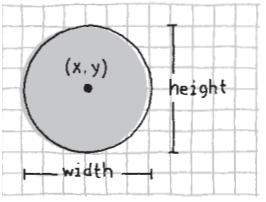

```
triangle(x1, y1, x2, y2, x3, y3)
```



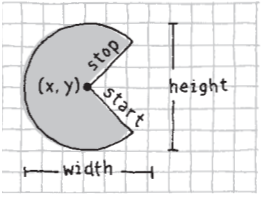

```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```




```
rect(x, y, width, height)
```




```
ellipse(x, y, width, height)
```




```
arc(x, y, width, height, start, stop)
```

Рис. 3-1. Координаты и фигуры.

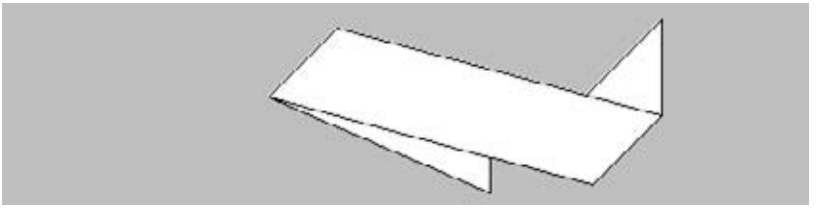
Пример 3-3: Рисуем линию

Чтобы нарисовать линию между точками с координатами (20, 50) и (420, 110) используем эту функцию:

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

Пример 3-4: Рисуем фигуры

Согласно этому рисунку, для треугольника требуется 6 параметров, для четырехугольника - 8 (по 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: Рисуем прямоугольник

Прямоугольники, как и эллипсы, задаются 4 параметрами: первый и второй являются коррдинатами опорной точки, третий и четвертый нужны для ширины и длины. Для рисования прямоугольника из точки (180, 60) шириной 220 пикселей и высотой в 40 используем функцию *rect*():

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

Пример 3-6: Рисуем эллипс

Опорная точка прямоугольника находится в верхнем левом углу, эллипса - в центре. Заметьте, что в этом примере у-координата опорной точки первого эллипса находится за пределами окна. Задание фигуры частично (или полностью) за пределами окна не считается ошибкой:

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

Processing не содержит специальных функий для рисования квадрата или круга. Для изображения этих фигур, используйте функции *ellipse*() и *rect*() и те же параметры *width* и *height*.

Пример 3-7: Рисуем часть эллипса

Функция *arc*() позволяет изобразить часть эллипса:

```
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 параметр устанавливают положение, третий и четвертый определяют ширину и высоту. Пятый параметр устанавливает угол начала дуги, шестой устанавливает конец дуги. Значение углов лучше указывать в радианах, а не в градусах. Радианы основаны на значении пи (3.14159). Рисунок 3-2 показывает соотношение между этими величинами. Как видно из данного примера, указанные 4 значения углов в радианах используются настолько часто, что в Processing им присвоены имена. Значения PI, QUARTER_PI, HALF_PI и TWO_PI можно заменить на значения в градусах 180°, 45°, 90° и 360°.

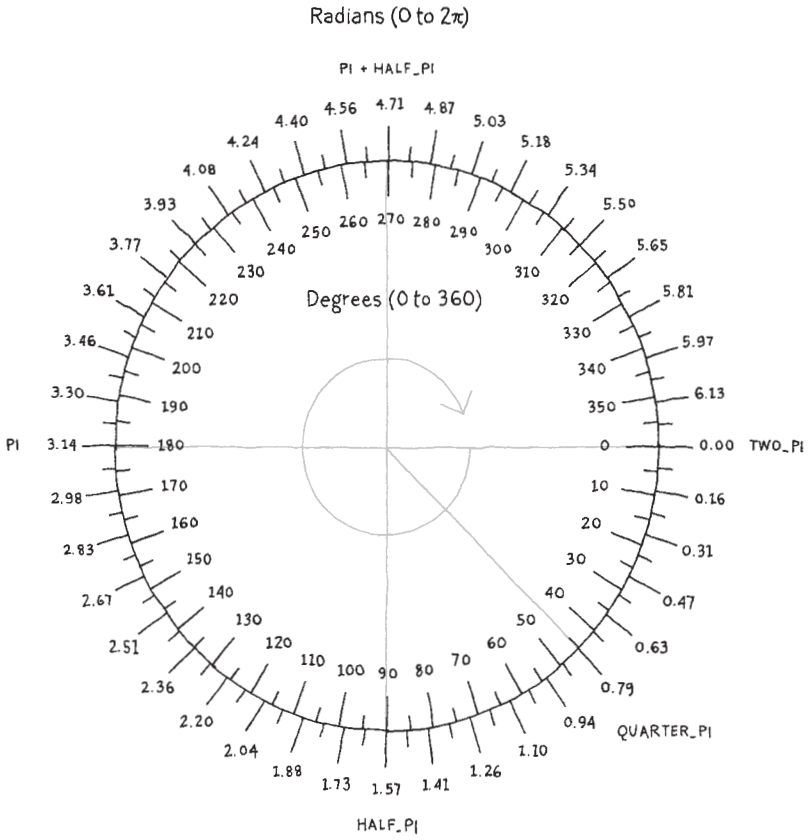


Рис. 3-2. Измерение углов в радианах и градусах.

Пример 3-8: Используем градусы

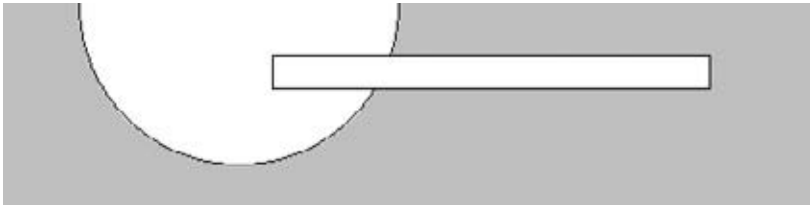
Если вы предпочитаете использовать градусы, вы можете конвертировать ваше значение в градусах в радианы функцией *radians*(). Эта функция заменяет значение угла в градусах на соответствующее значение в радианах. Следующий пример выполняет те же действия, что и пример 3-7, но для обозначения начального и конечного угла применена функция *radians*():

```
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));
```

Последовательность рисования

После запуска программы ваш компьютер начинает выполнять команды последовательно, начиная с первой строки, и останавливается, когда достигает последней. Если вы хотите, чтобы фигура была нарисована поверх остальных, соответствующая функция должна быть написана после функций, описывающих остальные фигуры.

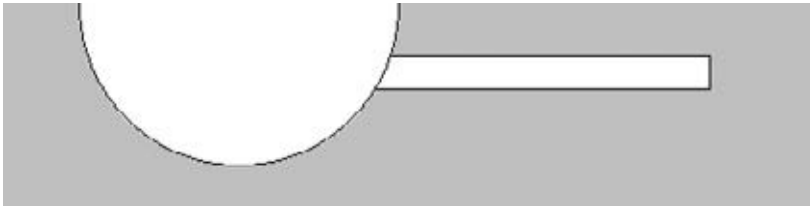
Пример 3-9: Соблюдаем последовательность



```
size(480, 120);
ellipse(140, 0, 190, 190);
// Прямоугольник расположен поверх эллипса
// потому, что в программе он стоит после него
rect(160, 30, 260, 20);
```

Пример 3-10: Напишем наоборот

Измените пример 3-9, поменяйте местами функции *rect*() и *ellipse*() и убедитесь в том, что круг нарисован поверх прямоугольника.



```
size(480, 120);
rect(160, 30, 260, 20);
// Эллипс расположен поверх прямоугольника
// потому, что в программе он стоит после него
ellipse(140, 0, 190, 190);
```

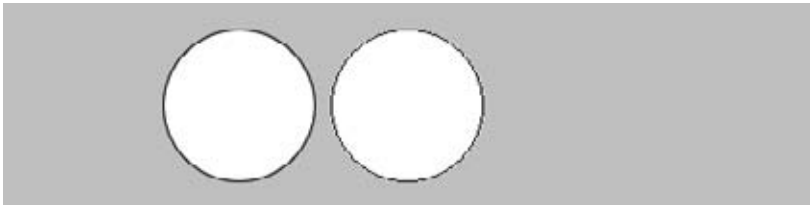
Представьте, что вы рисуете кистью или делаете коллаж. Видимым будет тот элемент, который вы добавите последним.

Свойства фигур

Самые простые и полезные свойства фигур это толщина линии и антиэлайзинг, также известный как сглаживание.

Пример 3-11: Сглаживаем линии

Функция *smooth*() сглаживает края линий, выведенных на экран, делая плавный переход между пикселями, находящимися на краях штрихов. В свою очередь, функция *noSmooth*() отключает эту функцию:

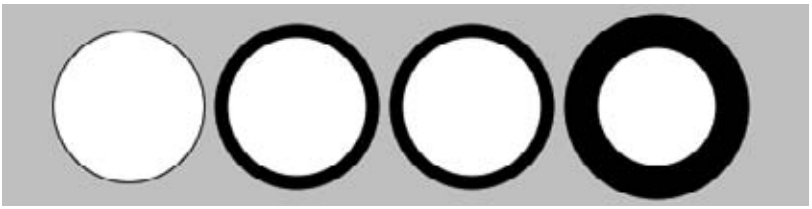


```
size(480, 120);
smooth();  // Включаем сглаживание
ellipse(140, 60, 90, 90);
noSmooth();  // Выключаем сглаживание
ellipse(240, 60, 90, 90);
```

Примечание: некоторые версии Processing (такие как версия для JavaScript) всегда сглаживают края; некоторые не поддерживают сглаживание. Иногда нельзя включить и отключить сглаживание в одном блоке команд *draw*(). За более подробной информацией по функции *Smooth*() обратитесь к справке.

Пример 3-12: Устанавливаем толщину линий.

По умолчанию, толщина линий состоавляет 1 пиксель, но вы можете изменить это значение функцией *stroke*W*eight*(). Единственный параметр функции *stroke*W*eight*() устанавливает толщину всех линий:



```
size(480, 120);
smooth();
ellipse(75, 60, 90, 90);
strokeWeight(8);  // Расширяем линию до 8 пикселей
ellipse(175, 60, 90, 90);
ellipse(279, 60, 90, 90);
strokeWeight(20);  // Расширяем линию до 20 пикселей
ellipse(389, 60, 90, 90);
```

Пример 3-13: Установить атрибуты линий

Функция *strokeJoin*() определяет вид соединения линий (углы); а функция *strokeCap*() - как будут выглядеть начальная и конечная точки линий:

```
size(480, 120);
smooth();
strokeWeight(12);
strokeJoin(ROUND);       // Скруглить углы
rect(40, 25, 70, 70);
strokeJoin(BEVEL);       // Сделать скос на углах
rect(140, 25, 70, 70);
strokeCap(SQUARE);       // Концы линий - квадратные
line(270, 25, 340, 95);
strokeCap(ROUND);        // Скруглить концы линий
line(350, 25, 420, 95);
```

Функции рисования фигур типа прямоугольника или эллипса можно настроить с помощью функций *rect*M*ode*() и *ellipse*M*ode*(). Чтобы изменить опорную точку для прямоугольника (рисовать его относительно центральной точки, а не левой верхней) или эллипса (рисовать его не относительно центра, а относительно верхней левой точки), обратитесь к справке (Help→Reference) за примерами.

Если установлены какие-либо из этих атрибутов, все фигуры, следующие за ними, будут изображаться в соответствии с ними. Обратите внимание, что в примере 3-12 вторая и третья окружность имеют одну и ту же ширину линии, несмотря на то, что атрибут линии установлен один раз перед соответствующими функциями окружностей.

Цвет

До сих пор, все фигуры, которые мы рисовали, были белыми, линии -черными, а фон окна - светло-серый. Чтобы изменить эти параметры, используйте функции *background*(), *fill*() и *stroke*(). Величина этих параметров варьируется в пределах от 0 до 255, где 255 это белый, 128 -серый, и 0 - черный. Рисунок 3-3 показывает, какому оттенку серого соответствует величина параметра.

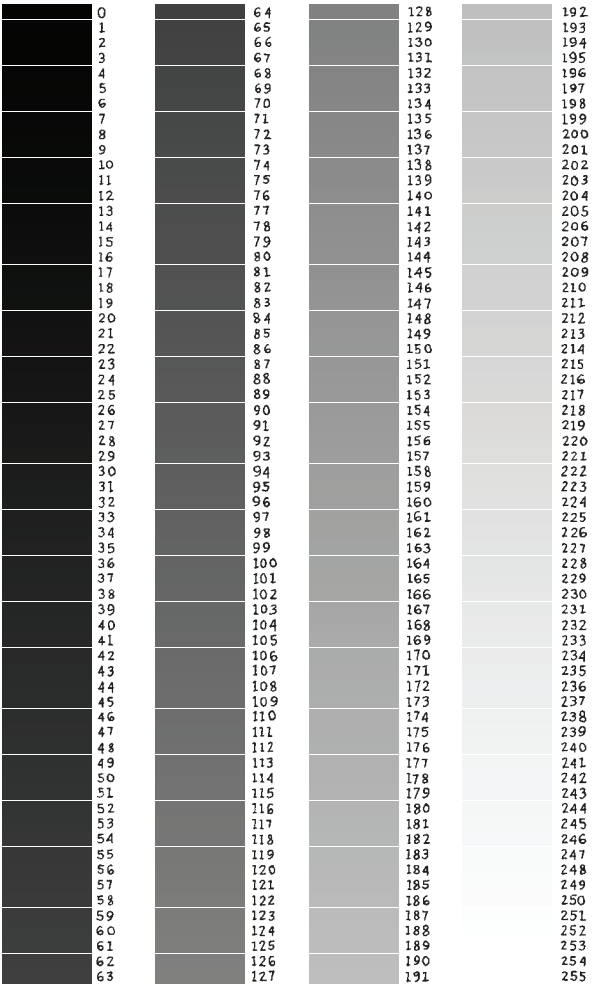


Рис. 3-3. Оттенки серого от 0 до 255.

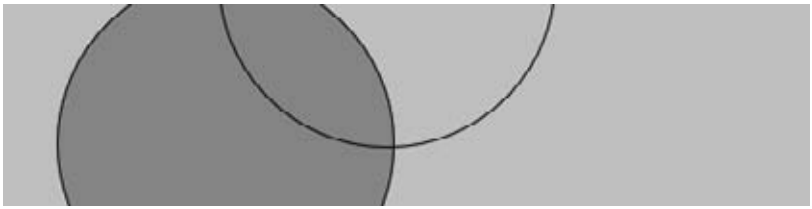
Пример 3-14: Оттенки серого

Этот пример демонстрирует 3 разных оттенка серого на черном фоне:

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

Пример 3-14: Цвета фигур и линий

Вы можете сделать контур вашей фигуры невидимым функцией *noStroke*() или сделать фигуру прозрачной (выключить заливку) функцией *noFill*():



```
size(480, 120);
smooth();
fill(153);                    // Серый
ellipse(132, 82, 200, 200);   // Серый круг
noFill();                     // Выключаем заливку
ellipse(228, -16, 200, 200);  // Контур круга
noStroke();                   // Делаем линии невидимыми
ellipse(268, 118, 200, 200);  // Этот эллипс невидим!
```

Будьте внимательны, когда применяете обе функции *fill* и *stroke*: как вы заметили в предыдущем примере, рисунок может стать невидимым.

Пример 3-16: Рисуем цветные фигуры

Чтобы не ограничиваться возможностями серого цвета, нам понадобится три параметра для описания красного, зеленого и синего компонентов цвета. Эта книга - черно-белая, поэтому вы не будете видеть здесь цветных иллюстраций. Запустите следующий код и вы получите результат в цвете:

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

Это связано с цветовой моделью RGB, которая используется для получения различных цветов на экране компьютера. Три числа задают величину красного, зеленого и синего цвета и, как и оттенки серого, находятся в пределах от 0 до 255. Использование модели RGB может не быть интуитивно понятным, поэтому используйте цветовую палитру (Tools→Color Selector); похожий инструмент, вы, вероятно, использовали в других программах (Рис. 3-4). Выберите желаемый цвет, а затем используйте параметры R, G и B для функций *background*(), *fill*() и *stroke*().

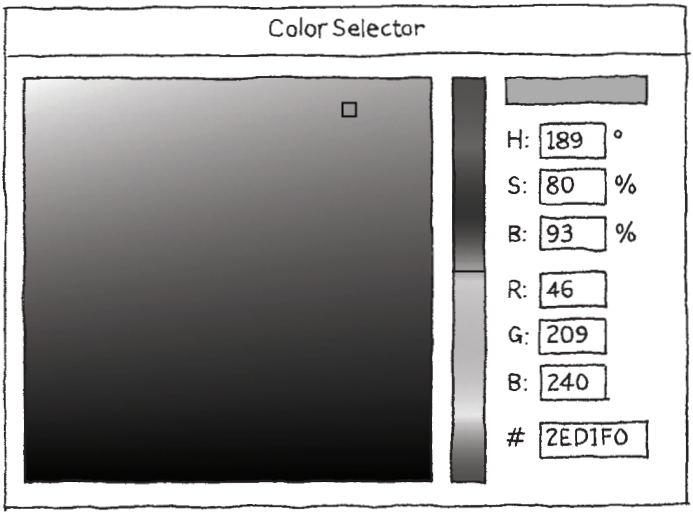


Рис. 3-4. П*алитра в Processing.*

Пример 3-17: Устанавливаем прозрачность

Добавив четвертый параметр к функциям *fill*() и *stroke*(), вы сможете регулировать прозрачность. Этот параметр, также известный как величина *alpha*, тоже использует диапазон от 0 до 255 для количественного выражения прозрачности. Величина 0 задает полностью прозрачный объект (он не отобразится), 255 - непрозрачный, а с помощью промежуточных значений можно смешивать цвета на экране:

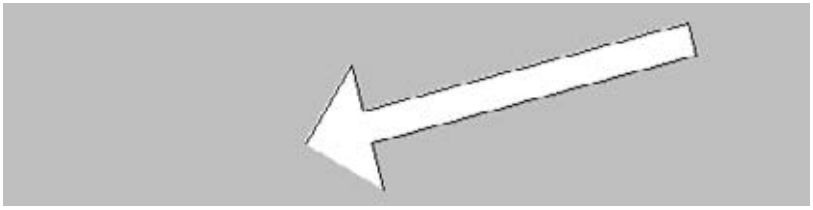
```
size(480, 120);
noStroke();
smooth();
background(204, 226, 225);     // Светло-синий
fill(255, 0, 0, 160);          // Красный
ellipse(132, 82, 200, 200);    // Красный круг
fill(0, 255, 0, 160);          // Зеленый
ellipse(228, -16, 200, 200);   // Зеленый круг
fill(0, 0, 255, 160);          //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

Фигуры произвольной формы

Если вы не хотите ограничиваться приведенными базовыми геометрическими фигурами, вы можете создать новые, соединив нужные точки линиями.

Пример 3-18: Рисуем стрелку

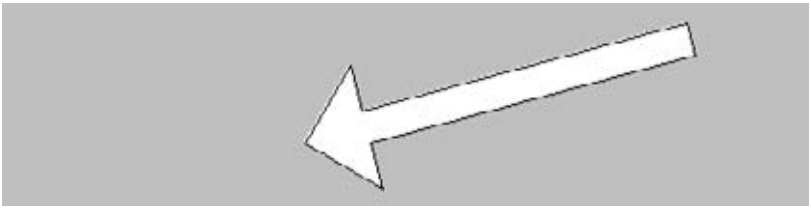
Начните создание вашей фигуры с функции *beginShape*(). Функция *vertex*() используется для определения x и y-координат фигуры. Функция *endShape*() ставится в конце описания фигуры и сигнализирует об окончании создания фигуры.



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

Пример 3-19: Заполняем пропуск

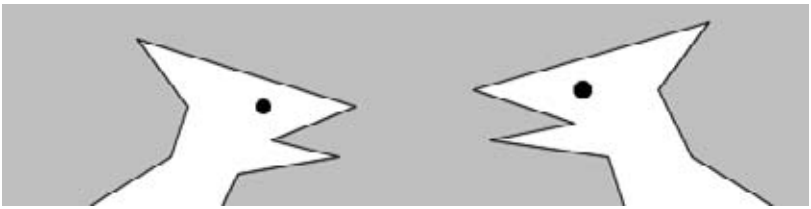
Когда вы запустите пример 3-18, вы увидите, что первая и последняя точки не соединены. Чтобы сделать это, добавьте слово C*LOS*E как параметр функции *endShape*() таким образом:



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

Пример 3-20: Создадим несколько созданий

Преимущество использования функции v*ertex*() для создания фигур заключается в возможности построения фигур со сложной структурой. Одна программа на Processing может нарисовать тысячи линий для отображения на экране самых невероятных фигур, которые создаст ваше воображение. Следующий пример воспроизводит более сложную фигуру с использованием функции v*ertex*():



```
size(480, 120);
smooth();

// Левое создание
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 создание
  fill(255); 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);
```

Комментарии

В приведенных примерах комментарии к коду, расположенные в конце строки, выделены двойной косой чертой (//). Комментарии - это та часть программы, которая игнорируется при запуске программы. Комментарии позволяют вам добавлять заметки к коду, чтобы потом вы могли сразу разобраться, как функционирует программа, и поэтому они очень удобны. Комментарии окажутся особенно важны, если ваш код будет читать кто-то другой; они помогут понять ход ваших мыслей.

Комментарии могут пригодиться вам и в других задачах, таких, например, как выбор нужного цвета. Так, предположим, что я подыскиваю нужный оттенок красного для некоторого эллипса:

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

А теперь предположим, что я захотел попробовать другой оттенок, но не хотел бы терять предыдущий. Я могу копировать и вставить строку, внести изменение, а затем превратить прежнюю строку в комментарий:

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

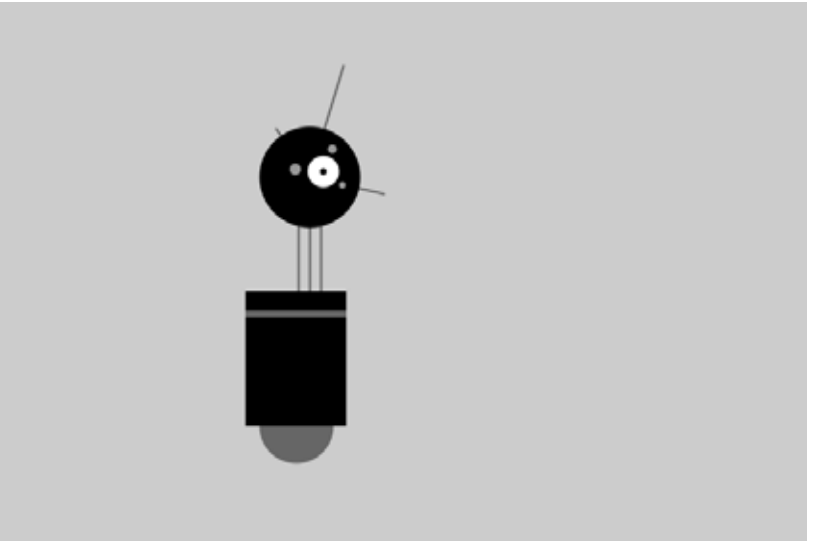
Размещение // в начале строки временно отключает ее. Я могу удалить // и поместить ее перед другой строкой, если хочу попробовать еще:

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

ПРИМЕЧАНИЕ: Вы можете использовать комбинацию клавиш Ctrl-/ (Cmd-/ на Mac) для быстрого добавления или удаления комментириев из текущей строки или из выделенного фрагмента текста. Вы также можете превратить в комментарий сразу несколько строк; подробнее об этом в Приложении A.

По мере работы в Processing вы заметите, что вам приходится перепроверять свои идеи дясятки раз; добавление комментариев или выключение фрагментов кода путем превращения их в комментарий помогут вам отследить развитие вашей идеи.

Робот 1: Рисование



Знакомьтесь: это P5, робот Processing. В этой книге содержится восемь различных скетчей, создающих статичный или анимированный вариант робота - каждый из скетчей развивает отдельную программную идею. На создание P5 нас вдохновил Спутник-1 (1957), робот Шеки (Shakey) из Стенфордского исследовательского института (Stanford Research Institute) (1966 - 1972), беспилотный истребитель из фильма “Дюна” (Dune) Дэвида Линча (David Lynch) (1984) и компьютер HAL 9000 из “Космическая одиссея 2001 года”(2001: A Space Odyssey) (1968).

Первая программа робота использует функции для рисования, описанные в предыдущей главе. Параметры функции *fill*() и *stroke*() задают черный, белый и серый цвета. Функции *line*(), *ellipse*() и *rect*() создают фигуры для шеи, антенн, туловища и головы робота. Чтобы получше освоить функции, запустите программу и измените параметры робота:

```
size(720, 480);
smooth();
strokeWeight(2);
background(204);
ellipseMode(RADIUS);

// Шея
stroke(102);                // Сделать линии серыми
line(266, 257, 266, 162);   // Левая
line(276, 257, 276, 162);   // Центральная
line(286, 257, 286, 162);   // Правая

// Антенны
line(276, 155, 246, 112);   // Малая
line(276, 155, 306, 56);    // Большая
line(276, 155, 342, 170);   // Средняя

// Туловище
noStroke();                 // Невидимые линии
fill(102);                  // Окрашивать серым
ellipse(264, 377, 33, 33);  // Шар
fill(0);                    // Окрашивать черным
rect(219, 257, 90, 120);    // Собственно туловище
fill(102);                  // Окрашивать серым
rect(219, 274, 90, 6);      // Серая полоса

// Голова
fill(0);                    // Окрашивать черным
ellipse(276, 155, 45, 45);  // Голова
fill(255);                  // Окрашивать белым
ellipse(288, 150, 14, 14);  // Большой глаз
fill(0);                    // Окрашивать черным
ellipse(288, 150, 3, 3);    // Зрачок
fill(153);                  // Окрашивать в светло-серый
ellipse(263, 148, 5, 5);    // Маленький глаз 1
ellipse(296, 130, 4, 4);    // Маленький глаз 2
ellipse(305, 162, 3, 3);    // Маленький глаз 3
```

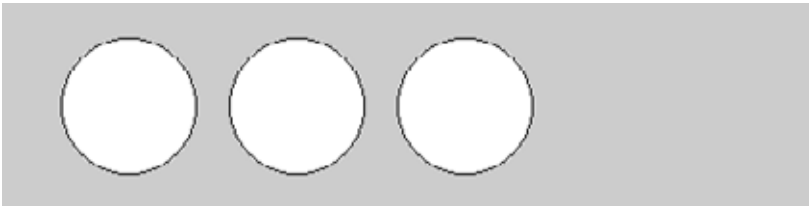
4/Переменные

Переменные сохраняют свое значение в памяти и поэтому могут быть использованы в программе неоднократно. Переменная может использоваться в одной программе много раз и изменять свое значение во время выполнения программы.

Главная причина для использования переменных - избежать повторения фрагментов кода. Если вы вводите одно и то же число более одного раза, рассмотрите возможность применения переменной вместо него; это сделает код более наглядным и удобным для внесения изменений.

Пример 4-1: Повторное использование значений

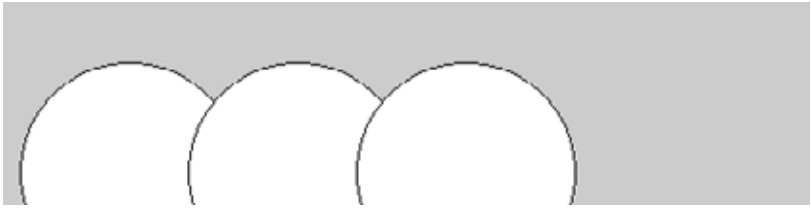
Если вы создадите переменные для хранения y-координаты и диаметра для кругов из следующего примера, то для параметров функции эллипса будет использована одна и та же величина:



```
size(480, 120);
smooth();
int y = 60;
int d = 80;
ellipse(75, y, d, d);   // Левый
ellipse(175, y, d, d);  // Центральный
ellipse(275, y, d, d);  // Правый
```

Пример 4-2: Изменяем значение переменной

Простое изменение переменных y и d видоизменяет все три эллипса:



```
size(480, 120);
smooth();
int y = 100;
int d = 130;
ellipse(75, y, d, d);   // Левый
ellipse(175, y, d, d);  // Центральный
ellipse(275, y, d, d);  // Правый
```

Без переменных вам пришлось бы трижды изменять значение y-координаты и шесть раз изменять диаметр. Обратите внимание, что примеры 4-1 и 4-2 различаются двумя строками в середине, а последние три строки не меняются. Переменные позволяют отделить строки кода, которые нужно изменять, от тех, которые менять не нужно, и это делает код более удобным для модификации. К примеру, если вам требуется управлять цветом и размером фигур в одном окне, то вы сможете сконцентрироваться на модификации всего нескольких строк кода и получить результат быстрее.

Объявление переменных

Когда вы объявляете новую переменную, вы указываете *имя*, *тип данных* и *значение* переменной. Имя это то, как вы будете называть переменную. Выбирайте такое имя, чтобы по нему можно было догадаться, какое значение хранит переменная, однако будьте последовательны и не слишком многословны. К примеру, когда вы вернетесь к коду спустя некоторое время, имя переменной “radius” будет более информативным, чем короткое “r”.

Диапазон значений, которые может принимать переменная, определяются *типом данных*. Например, тип данных *integer* хранит числа без дробной части (целые числа). В коде *integer* будет сокращено до *int*. Существуют типы данных для хранения различных данных: целых чисел, чисел с плавающей точкой, символов, слов, изображений, шрифтов и так далее.

Чтобы использовать переменные, их нужно объявить; в результате каждой переменной будет выделена область памяти для хранения информации. При объявлении переменной вы должны указать тип данных (например, *int*), это сообщает, какой вид информации будет храниться в памяти. После указания имени и типа данных вы можете присвоить значение переменной:

```
int x;   // Объявить переменную x типа int
x = 12;  // Присвоить значение переменной x
```

Этот код делает то же самое, но короче:

```
int x = 12; // Объявить переменную x типа int и присвоить ей значение
```

Тип данных указывается один раз при объявлении переменной. Каждый раз, когда вы указываете тип данных перед именем переменной, ваш компьютер думает, что вы объявляете новую переменную. Нельзя использовать две переменные с одним именем в одной программе (см. Приложение D), поэтому в этом случае программа сообщит об ошибке:

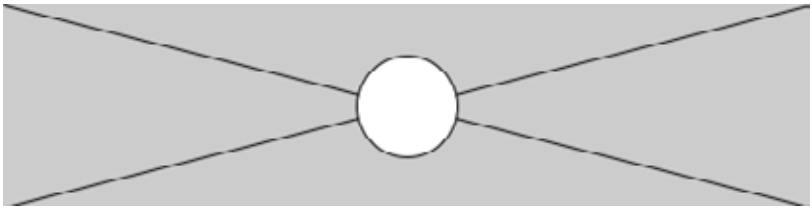
```
int x;       // Объявить переменную x типа int
int x = 12;  // ОШИБКА! Нельзя использовать две переменные с именем x
```

Переменные в Processing

Processing имеет ряд специальных переменных, хранящих информацию о программе в процессе ее работы. Например, ширина и высота окна сохраняются в переменных под названием *width* и *height*. Значения переменной устанавливаются в функции *size*(). Они могут быть использованы для изображения объектов относительно размера текщего окна, даже если параметры функции *size*() изменяются.

Пример 4-3: Корректируем размер окна

В этом примере мы изменим параметры функции *size*():



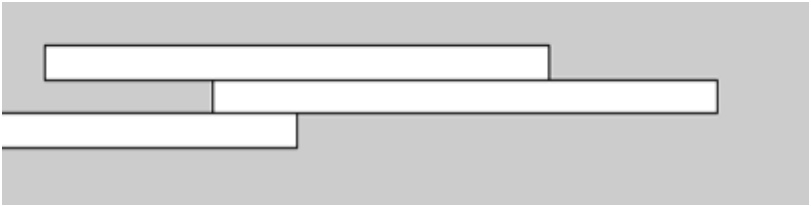
```
size(480, 120);
smooth();
line(0, 0, width, height); // Линия от (0,0) до (480,120)
line(width, 0, 0, height); // Линия от (480,0) до (0,120)
ellipse(width/2, height/2, 60, 60);
```

Существуют другие специальные переменные, отслеживающие состояние мыши, клавиатуры и т.д. Они обсуждаются в главе 5.

Немного математики

Считается, что математика и программирование это, по сути, одно и то же. Знание математики может быть полезным для написания некоторых видов программ, однако для реализации основных операций вам потребуется не больше, чем знание простой арифметики.

Пример 4-4: Простая арифметика



```
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
```

Символы +, - и * в коде называются *операторами*. Если вы записываете их между двумя значениями, то вы получаете *выражение*. Запись 5 + 9 или 1024 -512 является выражением. Ниже приведены операторы для простых арифметических операций:

+	Сложение
-	Вычитание
*	Умножение
/	Деление
=	Присвоение

Processing имеет набор правил, определяющих приоритет операторов, то есть какие операторы будут выполняться первыми, какие - вторыми, третьими, и так далее. Эти правила определяют порядок выполнения арифметических операций. Знание этих правил позволит вам понять, как работает, к примеру, эта строка кода:

```
int x = 4 + 4 * 5;  // Присвоение переменной x значения 24
```

Выражение 4 * 5 вычисляется первым, поскольку умножение имеет высший приоритет. Затем, прибавление к произведению 4 дает 24. Наконец, поскольку оператор присвоения (знак равенства) имеет низший приоритет, число 24 присваивается переменной x. Более наглядная запись содержит скобки, но имеет тот же результат:

```
int x = 4 + (4 * 5); // Присвоение значения 24 переменной x
```

Если вы хотите, чтобы сумма вычислялась первой, добавьте скобки. Так как выражение, заключенное в скобки имеет более высокий приоритет, чем умножение, порядок вычисления изменится, и это повлияет на результат:

```
int x = (4 + 4) * 5; // Присвоение значения 40 переменной x
```

На уроках математики часто используют мнемоническое правило: PEMDAS, что расшифровывается как Parentheses (скобки), Exponents (возведение в степень), Multiplication (умножение), Division (деление), Addition (сложение), Subtraction(вычитание), где скобки имеют наивысший приоритет, а вычитание - низший. Порядок всех операций - в Приложении C.

Некоторые арифметические операции настолько часто используются в программировании, что были придуманы способы быстрой записи этих операций; иногда бывает удобно нажимать на несколько клавиш меньше. Например, вы можете складывать или вычитать из переменной с помощью всего одного оператора:

```
x += 10; // Выражение эквивалентно x = x + 10
y -= 15; // Выражение эквивалентно y = y - 15
```

Также часто возникает необходимость прибавлять или вычитать 1 из переменной, поэтому существуют операторы для быстрого выполнения этих действий. Это операторы ++ и -- :

```
x++; // Выражение эквивалентно x = x + 1
y--; // Выражение эквивалентно y = y - 1
```

Для поиска других подобных операторов обратитесь к справке.

Циклы

По мере написания программ вы заметите, что вам часто приходится записывать подряд несколько однотипных строк с небольшими изменениями. Кодовая структура под названием *for* делает возможным запуск одной и той же строки с небольшими изменениями, что позволит вам свернуть множество однотипных строк в одну или две. Это разделит вашу программу на модули и сделает более удобной для внесения изменений.

Пример 4-5: Делаем одно и то же несколько раз

Этот пример содержит конструкцию, которую можно упростить циклом *for*:

```
size(480, 120);
smooth();
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: Используем цикл *for*

То же самое можно записать короче, использовав цикл *for*:

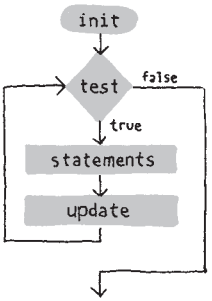
```
size(480, 120);
smooth();
strokeWeight(8);
for (int i = 20; i < 400; i += 60) {
  line(i, 40, i + 60, 80);
}
```

Цикл *for* во многом отличается от кода, который мы писали до сих пор. Обратите внимание на фигурные скобки - символы { и }. Код, заключенный в фигурные скобки, называется *блоком*. Именно этот фрагмент кода будет повторяться в цикле *for*.

Три выражения в скобках, разделенные точкой с запятой, определяют, сколько раз будет повторяться блок кода. Эти выражения называются *initializaton* (*init*), (инициализация), *test* (проверка) и *update* (обновление):

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

Выражение *init* обычно объявляет новую переменную для использования в цикле for и присваивает ей значение. Как правило, используется переменная с именем *i*, но вы можете использовать любую другую. T*est* оценивает значение переменной, а *update* изменяет это значение. Рисунок 4-1 показывает, в каком порядке они работают и как управляют выражениями внутри блока.




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

Рис. 4-1. П*оследовательность работы цикла for.*

Особо отметим выражение *test*. Это всегда выражение отношения, сравнивающее два значения оператором сравнения. В данном примере выражение “i < 400” использует оператор сравнения < (меньше). Вот самые распространенные операторы сравнения:

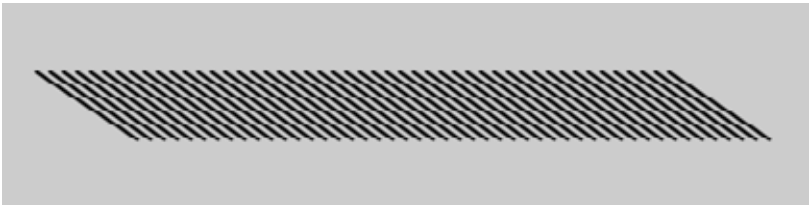
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
==	Равно
!=	Не равно

Результатом операции сравнения всегдя является *истина* или *ложь*. Например, выражение 5 > 3 это *истина*. Можно задать вопрос: “5 больше трех?” Так как ответом будет “Да”, выражение является *истиной*.

Для выражения 5 < 3 мы можем спросить: “5 меньше 3?”. Так как ответом будет “нет”, мы говорим, что выражение - *ложь*. Если выражение истинно, блок кода запускается, а если оно ложно, блок кода не запускается и цикл *for* завершается.

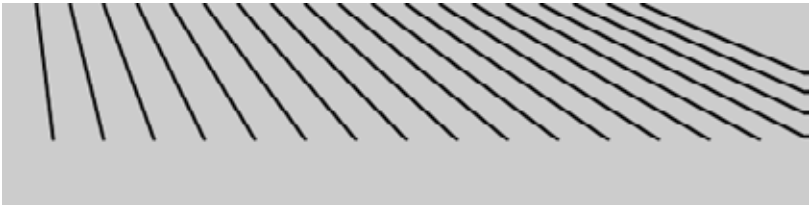
Пример 4-7: Прокачаем цикл *for*

Неоспоримое преимущество работы с циклом *for* - это возможность быстро вносить изменения в код. Блок кода внутри цикла запускается множество раз, и поэтому изменения распространяются на все итерации. Мы можем значительно изменить изображение в примере 4-6, немного изменив код:



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

Пример 4-8: Линии, исходящие из центра



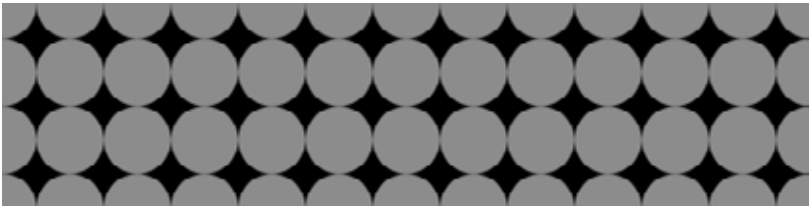
```
size(480, 120);
smooth();
strokeWeight(2);
for (int i = 20; i < 400; i += 20) {
  line(i, 0, i + i/2, 80);
}
```

Пример 4-9: Изгибаем линии

```
size(480, 120);
smooth();
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

Если вложить один цикл *for* в другой, количество повторений перемножается. Для начала рассмотрим небольшой пример, а затем реализуем его по частям в примере 4-11:



```
size(480, 120);
background(0);
smooth();
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: Ряды и колонки

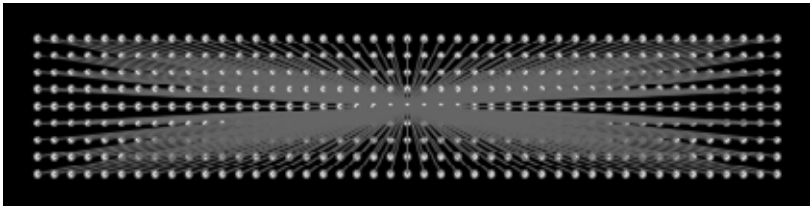
В этом примере циклы *for* расположены рядом, а не вложены один в другой. Результат выполнения программы показывает, что один цикл *for* рисует колонку из 4 кругов, другой ряд из 13 кругов:

```
size(480, 120);
background(0);
smooth();
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);
}
```

Если вложить один цикл *for* в другой, как в примере 4-10, 4 итерации первого цикла будут содержать 13 итераций второго, в результате блок кода внутри циклов будет запущен 52 раза (4x13 = 52).

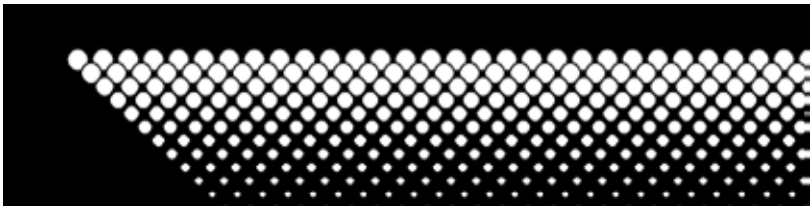
Пример 4-10 является хорошим шаблоном для создания рисунков с повторяющимися элементами. В следующих примерах мы покажем, как их можно модифицировать, но это будет всего лишь небольшая часть из того, что вы можете делать с циклом *for*. Код в примере 4-12 рисует линию от каждой точки до центра экрана. В примере 4-13 с каждой строкой происходит уменьшение размеров эллипсов и сдвиг их вправо с помощью сложения y-координаты и x-координаты.

Пример 4-12: Точки и линии



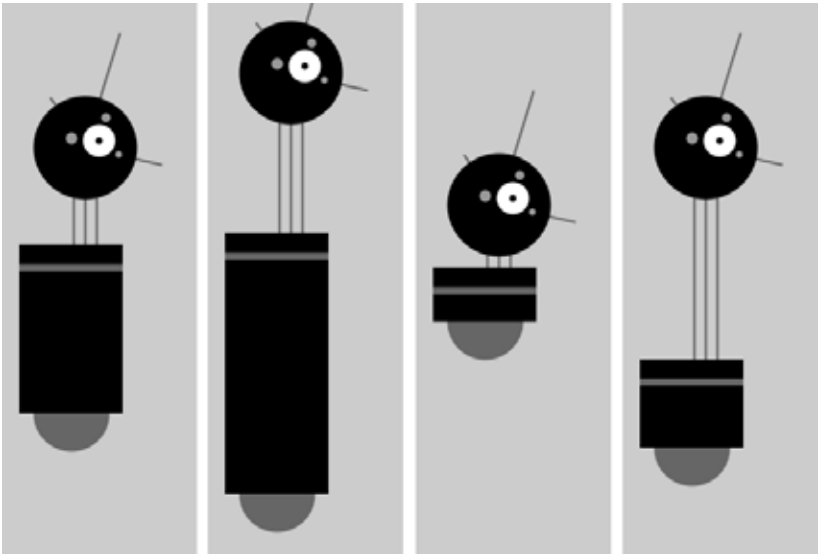
```
size(480, 120);
background(0);
smooth();
fill(255);
stroke(102);
for (int y = 20; y <= height-20; y += 10) {
  for (int x = 20; x <= width-20; x += 10) {
    ellipse(x, y, 4, 4);
    // Рисуем линии к центру экрана
    line(x, y, 240, 60);
  }
}
```

Пример 4-13: Точки и полутона



```
size(480, 120);
background(0);
smooth();
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);
  }
}
```

Робот 2: Переменные



Из-за использования переменных эта программа может показаться вам более сложной, чем Робот 1 из главы 3, но теперь у вас появилась возможность с легкостью модифицировать код, поскольку переменные, изменяющие тот или иной параметр робота, располагаются в одном месте. Например, мы можем регулировать длину шеи робота с помощью переменной *bodyHeight*. Несколько переменных в начале кода регулируют различные параметры робота, которые мы хотели бы изменить: местоположение, длину головища, длину шеи. На рисунке изображены несколько вариантов робота, ниже перечислены соответствующие значения переменных:

```
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 замене чисел на переменные подумайте, какие параметры вы хотели бы изменять и делайте изменения постепенно. Когда мы писали эту программу, мы вводили переменные по одной, чтобы облегчить модификацию кода. Как только мы убеждались, что введенная переменная работает правильно, мы добавляли следующую переменную:

```
int x = 60;            // x-координата
int y = 420;           // y-координата
int bodyHeight = 110;  // Длина туловища
int neckHeight = 140;  // Длина шеи
int radius = 45;       // Радиус головы
int ny = y - bodyHeight - neckHeight - radius;  // Шея Y

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

// Шея
stroke(102);
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(102);
ellipse(x, y-33, 33, 33);
fill(0);
rect(x-45, y-bodyHeight, 90, bodyHeight-33);
fill(102);
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);
ellipse(x, ny-8, 5, 5);
ellipse(x+30, ny-26, 4, 4);
ellipse(x+41, ny+6, 3, 3);
```

5/Интерактивность

Код, который взаимодействует с мышью, клавиатурой и другими устройствами должен выполняться непрерывно. Чтобы это стало возможным, включите фрагмент кода, который должен повторяться непрерывно, в функцию *draw*().

Пример 5-1: Функция *draw*()

Запустите этот код, чтобы посмотреть, как работает функция draw():

```
void draw() {
  // Выводит на консоль номер кадра
  println("I'm drawing");
  println(frameCount);
}
```

Вы увидите следующее:

```
I'm drawing
1
I'm drawing
2
I'm drawing
3
...
```

Код, записанный в блоке *draw*(), выполняется сверху вниз, а затем повторяется до тех пор, пока вы не выйдете из программы, нажав кнопку Stop или закрыв окно. Каждый "прогон" кода в функции *draw*() называется *кадр* (*frame*). (По умолчанию, частота "прогонов" кода равна 60 кадров в секунду, но вы можете установить другое значение. Мы сделаем это в примере 7-2.) Функция *println*() в предыдущем примере выводит текст "I'm drawing" и номер кадра с помощью специальной переменной *frameCount* (1, 2, 3, ...). Текст появляется в консольном окне - область с черным фоном под текстовым редактором Processing.

Пример 5-2: Функция *setup*()

В дополнение к функции *draw*() Processing имеет функцию *setup*(), которая запускается однократно при запуске программы:

```
void setup() {
  println("I'm starting");
}

void draw() {
  println("I'm running");
}
```

Пока программа работает, на консоль выводится следующее сообщение:

```
I'm starting
I'm running
I'm running
I'm running
...
```

Текст "I'm running" будет выводиться на консоль до тех пор, пока программа не будет остановлена.

В типичной программе код внутри функции *setup*() используется для задания начальных значений переменных. Первая строка - это, как правило, функция *size*(), далее могут быть включены функции, задающие цвета заливки и линий или функции для загрузки изображений и шрифтов. (Если вы не включите функцию *size*(), окно будет иметь размер 100x100 пикселей.)

Теперь вы знаете как использовать *setup*() и *draw*(), но это еще не все. Есть еще одна область в текстовом редакторе, куда вы можете вписывать код - это область вне функций *setup*() и *draw*(), где вы можете объявлять переменные. Переменная, объявленная в функции *setup*(), не может быть использована где-то еще. Переменная, объявленная вне этих функций, называется *глобальной переменной*, потому что она может быть использована в любом месте программы ("глобально"). Для пояснения приведем последовательность, в которой работает код:

1. Объявляются переменные вне функция *setup*() и *draw*().

2. Код внутри функции *setup*() запускается один раз.

3. Код внутри функции *draw*() работает непрерывно.

Пример 5-3: *setup*() и *draw*()

В этом примере применяются обе функции:

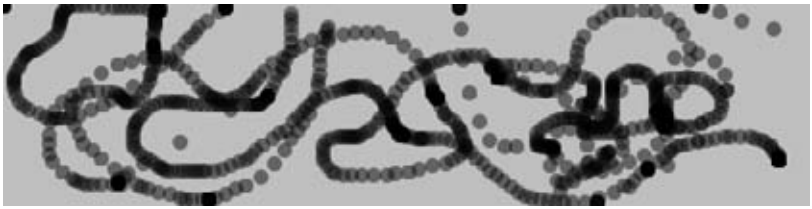
```
int x = 280;
int y = -100;
int diameter = 380;

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

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

Двигаем объекты мышью

Теперь у нас есть код, который работает непрерывно, и значит мы можем получать информацию о местоположении мыши и с помощью этих координат перемещать элементы на экране.

Пример 5-4: Следим за курсором

Переменная *mouse*X содержит x-координату, а переменная *mouse*Y содержит y-координату мыши:



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

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

В этом примере каждый раз, когда выполняется блок команд из функции *draw*(), в окне рисуется новый круг. Изображение выше получено перемещением мыши. Круги являются полупрозрачными, и по тому, где черный цвет наиболее интенсивен, можно понять, где курсор находился дольше и где он двигался с меньшей скоростью. Там, где расстояние между кругами больше, курсор перемещался быстрее.

Пример 5-5: Точка следует за курсором

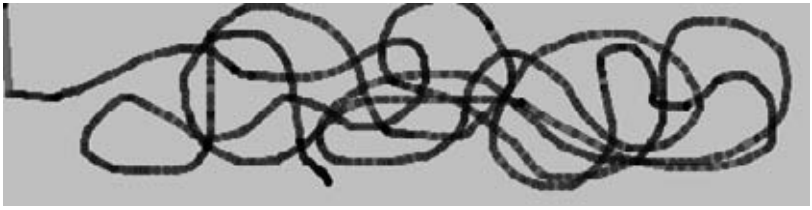
Здесь, каждый раз, когда выполняется блок команд из функции *draw*(), в окне появляется новый круг. Чтобы обновлять экран перед появлением нового круга, примените функцию *background*() в начале функции *draw*(), перед рисованием фигуры:

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

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

Функция *background*() очищает окно полностью, поэтому убедитесь, что она расположена перед другими функциями в блоке *draw*(), иначе фигуры, нарисованные перед ней, будут стерты.

Пример 5-6: Рисуем непрерывные линии

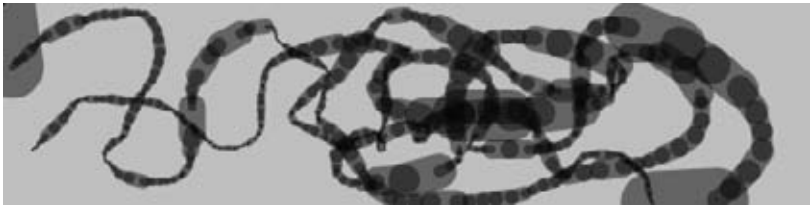
Переменные *pmouse*X и *pmouse*Y сохраняют позицию мыши из предыдущего кадра. Они обновляются при каждом запуске блока команд из *draw*(), как *mouse*X и *mouse*Y. Соединяя текущее и предыдущее положение курсора линией, можно рисовать непрерывную линию:



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

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

Пример 5-7: Меняем размер мухи

Переменные *pmouse*X и *pmouse*Y можно использовать для вычисления скорости перемещения мыши. Для этого нужно измерить расстояние между текущим и предыдущим положением мыши. Медленно двигающаяся мышь пройдет небольшое расстояние, с увеличением скорости расстояние возрастет. В следующем примере мы покажем, что функция *dist*() упрощает эти вычисления. Здесь скорость движения мыши устанавливат толщину проводимой линии:



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

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

Пример 5-8: Easing

В примере 5-7 координаты мыши непосредственно определяли положение круга на экране. Но иногда вы хотите сделать линию более плавной несмотря на неровную траекторию курсора. Эта техника называется *easing* (улучшение). Для нее нужны две величины - текущая и следующая (Рис. 5-1). С каждым шагом текущее значение приближается к следующему:

```
float x;
float easing = 0.01;
float diameter = 12;

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

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

}
```

Значение переменной x всегда стремится к *target*X. Скорость, с которой x стремится к *target*X устанавливается переменной *easing* в диапазоне от 0 до 1. Невысокое значение easing вызывает большую инерционность, чем высокое. При easing равной 1 задержки не будет. Когда вы запустите пример 5-8, обе величины будут выводиться на консоль функцией *println*(). Заметьте, что когда вы двигаете мышь, эти числа сильно различаются, но когда мышь неподвижна, они постепенно уравниваются.

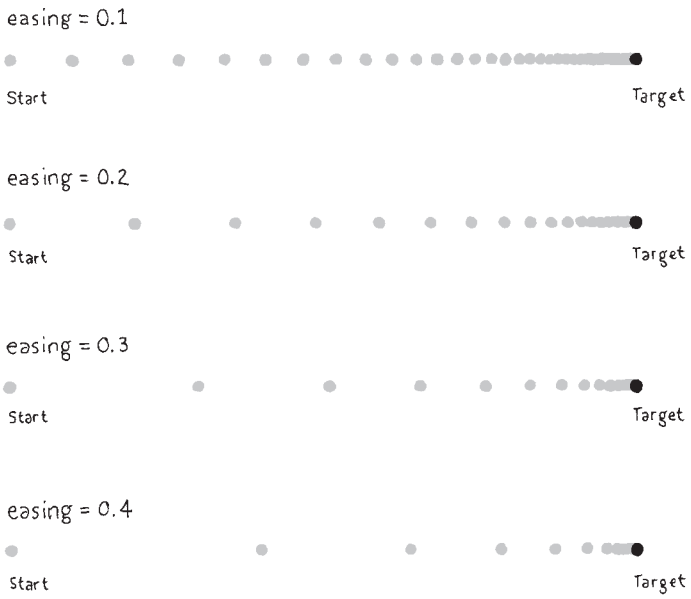
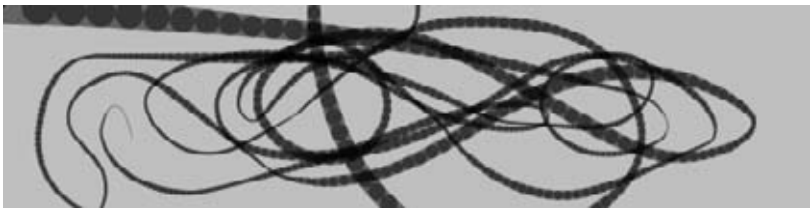


Рис. 5-1. E*asing.*

Основная часть кода в этом примере начинается с x +=. Вычисляется разность между следующим и текущим положением круга, затем умножается на переменную easing и прибавляется к x для приближения к следующему значению.

Пример 5-9: Сглаживание линий с помощью easing

В этом примере к коду из примера 5-7 применяется техника easing. Линии в этом примере более сглажены, чем в примере 5-7:



```
float x;
float y;
```

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

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

Преобразование чисел

Если вы используете числа для рисования на экране, то у вас может появиться необходимость изменить диапазон значений какой-либо переменной.

Пример 5-10: Изменяем диапазон значений

Переменная *mouse*X находится в пределах от 0 до ширины окна, но, возможно, вы захотите изменить диапазон значений переменной *mouse*X. Для этого вам потребуется разделить mouseX на некоторую величину, а затем прибавить или вычесть другую величину для реализации сдвига диапазона влево или вправо:

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

void draw() {
  background(204);
  stroke(255);
  line(120, 60, mouseX, mouseY);  // Белая линия
  stroke(0);
  float mx = mouseX/2 + 60;
  line(120, 60, mx, mouseY);      // Черная линия
}
```

Более удобный способ сделать это - функция *map*(). Она преобразует диапазон значений переменной. Первый параметр - переменная, второй и третий - минимальное и максимальное значение переменной, четвертое и пятое - желаемое минимальное и максимальное значение переменной. Все вычисления скрыты в функции *map*().

Пример 5-11: Преобразуем диапазон с функцией *map*()

Следующий пример упрощает пример 5-10 с помощью функции *map*():

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

void draw() {
  background(204);
  stroke(255);
  line(120, 60, mouseX, mouseY);  // Белая линия
  stroke(0);
  float mx = map(mouseX, 0, width, 60, 180);
  line(120, 60, mx, mouseY);      // Черная линия
}
```

Функция *map*() делает код легко читаемым, поскольку минимальное и максимальное значения заданы явно в качестве входных параметров. В приведенном примере диапазон *mouse*X от 0 до *width* преобразован в

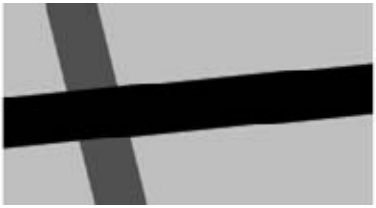
диапазон от 60 (когда *mouse*X = 0) до 180 (когда *mouse*X=*width*). Функция *map*() пригодится нам в большом количестве примеров из этой книги.

Кликаем мышью

Кроме позиции курсора Processing может отслеживать состояние кнопок мыши. Переменная *mousePressed* принимает различные значения в зависимости от того, нажата кнопка мыши или нет. В переменной *mousePressed* сохраняются данные типа *boolean*; это значит, что она принимает два значения: *истина* (*true*) и *ложь* (*false*). Когда нажата кнопка мыши, *mousePressed* принимает значение *истина*.

Пример 5-12: Кликаем мышью

Переменная *mousePressed* используется вместе с оператором *if* чтобы определить, когда должна запускаться строка кода. Прежде чем мы перейдем к дальнейшему объяснению, рассмотрите этот пример:



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

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

В приведенном примере код в блоке *if* запускается, когда нажата кнопка мыши. Если кнопка отжата, этот код игнорируется. Как и цикл *for*, который мы обсуждали в главе 4, оператор *if* имеет выражение *test*, которое оценивается как *ложь* или *истина*:

```
if (test) {
  statements
}
```

Если *test* является *истиной*, код в блоке запускается; если *ложью*, код не запускается. Ваш компьютер вычисляет выражение, заключенное в скобки и оценивает, является выражение *test истиной* или *ложью*. (Если вы хотите освежить свои знания об операторах сравнения, откройте страницу с пояснениями к примеру 4-6).

Символ == проверяет, действительно ли равны выражения слева и справа. Символ == не следует путать с оператором присвоения =. Символ == как бы спрашивает: "Эти выражения - равны?", а символ = просто присваивает значение переменной.

ПРИМЕЧАНИЕ: Часто даже опытные программисты делают распространенную ошибку: пишут = вместо ==. Processing не всегда предупреждает об этой ошибке, поэтому будьте внимательны.

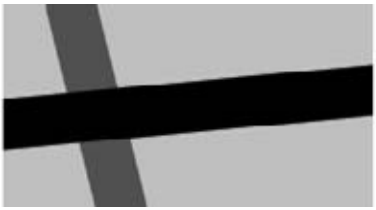
Вы можете записать выражение *test* в блоке *draw*() в примере 5-12 короче:

```
if (mousePressed) {
```

Переменные, в которых хранятся данные типа *boolean*, в том числе *mousePressed*, не требуют сравнения с помощью оператора ==, потому что они не принимают других значений кроме значений *истина* и *ложь*.

Пример 5-13: Определяем, нажата ли кнопка

Оператор *if* позволяет вам выбирать, запускать определенный блок кода или нет. Вы можете расширить возможности оператора *if*, добавив к нему *else* для выбора между двумя блоками кода. Если выражение *test* в операторе *if* является ложным, запускается блок *else*. Например, вы можете сделать цвет линий белым, когда кнопка мыши нажата и черным, когда отжата:



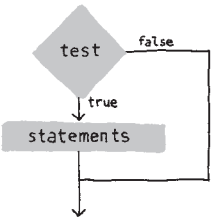
```
void setup() {
  size(240, 120);
  smooth();
  strokeWeight(30);
}

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

Пример 5-14: Работаем со всеми кнопками мыши

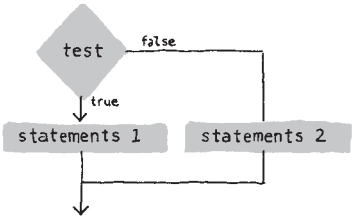
Processing имеет возможность отслеживать, какая кнопка нажата, если на вашей мыши более одной кнопки. Переменная *mouseButton* принимает три значения: LEFT, CENTER и RIGHT. Используйте оператор сравнения ==, чтобы определить, какая из кнопок нажата:

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

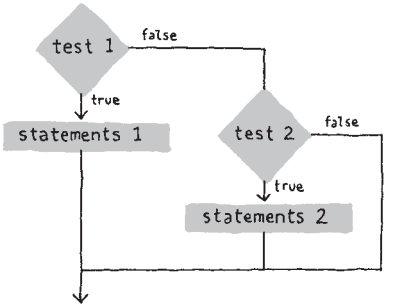
Программа может иметь и другие структуры *if* и *else* (Рис. 5-2), кроме тех, что представлены в этих примерах. Операторы могут образовывать длинную цепочку, каждое звено которой проверяет какое-либо условие, а также одни блоки *if* могут входить в другие блоки *if* для реализации сложных условий.




```
if (test) {
  statements
}
```




```
if (test) {
  statements 1
} else {
  statements 2
}
```




```
if (test 1) {
  statements 1
} else if (test 2) {
  statements 2
}
```

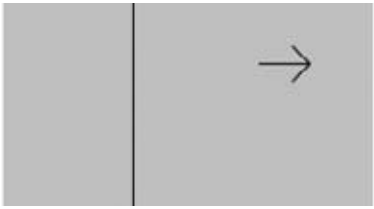
Рис. 5-2. Структура if-else принимает решение о том, какой блок кода запускать.

Положение курсора на экране

Оператор *if* может быть использован вместе в переменными *mouse*X и *mouse*Y для задания положения курсора в окне:

Пример 5-15: Найди курсор

Программа в этом примере определяет, где находится курсор - справа или слева от линии, а затем двигает линию в направлении курсора:



```
float x;
int offset = 10;

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

void draw() {
  background(204);
  if (mouseX > x) {
    x += 0.5;
    offset = -10;
  }
  if (mouseX < x) {
    x -= 0.5;
    offset = 10;
  }
  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 писать программы, имеющие графический интерфейс пользователя (кнопки, полосу прокрутки, флажки и т.д.), необходимо написать код, который знает, когда курсор находится в определенной области экрана. Следующие два примера показывают, как проверить, находится курсор в пределах круга или прямоугольника, или нет. Код составлен из отдельных модулей с переменными, поэтому вы сможете использовать его для проверки любого круга или прямоугольника просто заменив несколько величин.

Пример 5-16: Границы круга

Для проверки круга мы применим функцию *dist*() для определения расстояния от центра круга до курсора, а затем сравним это расстояние с радиусом круга (Рис. 5-3). Если оно меньше, мы будем знать, что курсор внутри круга. В следующем примере, когда курсор попадает в круг, размер круга увеличивается:

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

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

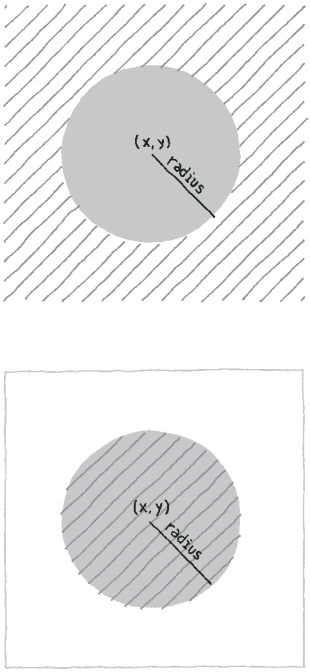


Рис. 5-3. Тест с кругом

Пример 5-17: Границы прямоугольника

Для определения, находится курсор в прямоугольнике, или нет, мы применим другой подход. Мы реализуем четыре отдельных теста, проверяющих, по какую сторону от каждой линии прямоугольника находится курсор, а затем сравним результаты - если все окажутся истинными, программа будет знать, что курсор находится внутри прямоугольника. Это иллюстрирует рис. 5-4. Все действия просты, но, собранные в одну программу, могут показаться сложными:

```
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);
}
```

Выражение в операторе *if* несколько сложнее, чем те, что мы видели раньше. Чтобы убедиться в том, что все четыре выражения (*mouse*X > x, и др.) истинны, они собраны в одно выражение с помощью логического И - символа &&. Если хотя бы одно выражение ложно, все выражение будет ложно, и прямоугольник не изменит цвет на черный. Символ && будет обсуждаться далее.

Клавиатурный ввод

Processing отслеживает нажатие кнопок на клавиатуре, а также какая из кнопок была нажата последней. Переменная *keyPressed*, как и *mousePressed*, истинна, когда нажата какая-либо кнопка, и ложна, когда не нажата ни одна кнопка.

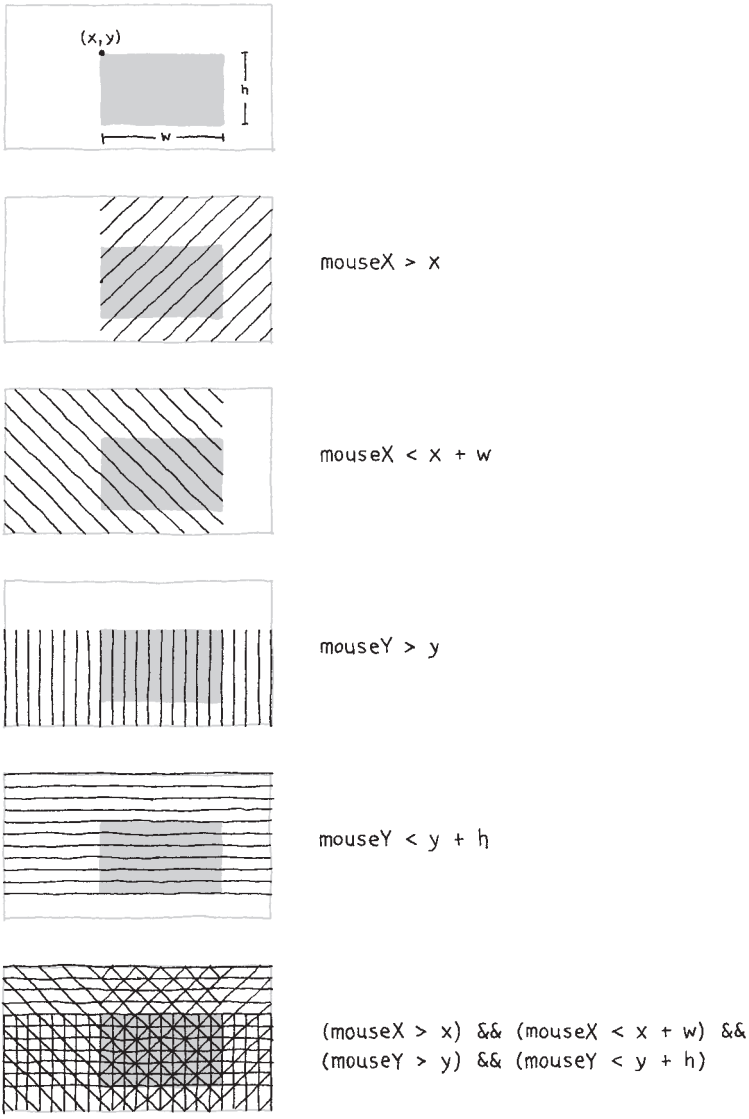
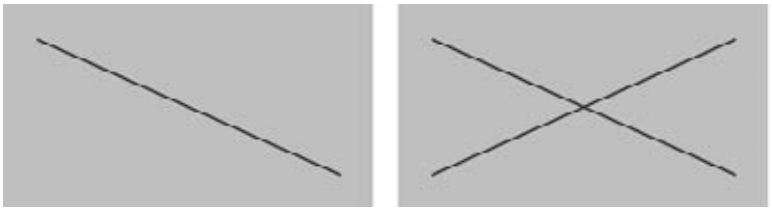


Рис. 5-4. Тест с прямоугольником.

Пример 5-18: Нажимаем на клавиши

В этом примере вторая линия будет отображаться только если нажата какая-либо из клавиш:



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

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

Переменная *key* сохраняет символ последней нажатой клавиши. *key* сохраняет символьный тип данных - *character*, сокращенно - *char*, произносится “чар”. Символьная переменная может хранить любой символ, включая буквы алфавита, цифры и различные знаки. В отличие от типа *string* (строка), (см. пример 6-8), определяемого в двойных кавычках, тип данных *char* определяется в одинарных кавычках. Вот пример объявления переменной типа *char* и присвоения ей значения:

```
char c = 'A'; // Объявление переменной c и присвоение ей значения 'A'
```

А в этих случаях вы получите сообщение об ошибке:

```
char c = "A";  // Ошибка! Нельзя присваивать переменной
               // char значение string
char h = A;    // Ошибка! Пропущены кавычки для 'A'
```

В отличие от переменной *keyPressed*, которая меняет свое значение на *ложь*, всегда, когда клавиша отжата, переменная *key* сохраняет свое значение до тех пор, пока не будет нажата следующая клавиша. В следующем примере мы используем переменную *key*, чтобы изображать на экране символы. Каждый раз, когда вы будете нажимать клавишу, значение переменной будет обновляться и вы увидите новый символ. Некоторые клавиши, вроде Shift и Alt, не имеют видимого символа, и когда вы нажмете их, вы ничего не увидите.

Пример 5-19: Рисуем буквы

Этот пример вводит функцию *textSize*(), для установки размера букв, *textAlign*() для расположения букв по центру и функцию *text*() для рисования букв. Мы обсудим эти функции подробнее на страницах 84-85.


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

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

Мы можем проверить, какая клавиша нажата, с помощью оператора *if* и нарисовать что-либо на экране в соответствии этой буквой.

Пример 5-20: Проверям нажатие клавиш

В этом примере мы будем определять, какая из клавиш нажата, H или N. Для сравнения переменной *key* с заданым значением мы используем оператор сравнения == :

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

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);
}
```

Во время нажатия клавиш H или N Caps Lock может находиться в активном состоянии или может удерживаться клавиша Shift, поэтому программа должна проверять нажатие строчной и прописной вариантов этих букв. Мы скомбинируем два этих теста в один логическим ИЛИ, символом ||. Проще говоря, оператор *if* ставит условие: “Если нажата клавиша 'h' или если нажата клавиша 'H'”. В отличие от логического И (символа &&) из примера 5-17, одного инстинного выражения достаточно, чтобы все выражение было истинным.

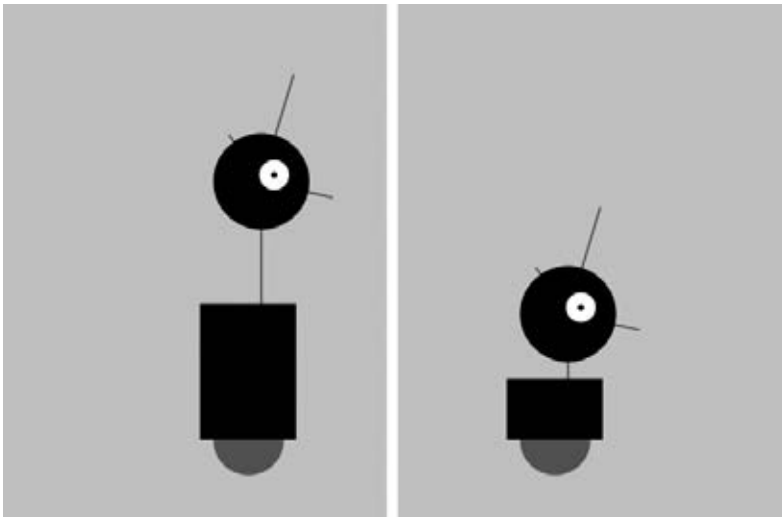
С клавишами, которые не связаны с определенной буквой (CODED), работать сложнее. Клавиши типа Shift, Alt, стрелок требуют дополнительных действий. Во-первых, мы должны проверить, связана ли клавиша с определенной буквой, а затем с помощью переменной *keyCode* проверить, какая это клавиша. Чаще всего используются клавиши ALT, CONTROL и SHIFT, а также стрелки - UP, DOWN, LEFT и RIGHT.

Пример 5-21: Управляем движением стрелками

Следующий пример показывает, двигать прямоугольник влево и вправо с помощью стрелок:

```
int x = 215;

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

void draw() {
  if (keyPressed && (key == CODED)) {  // Клавиша CODED
    if (keyCode == LEFT) {             // Стрелка влево
      x--;
    } else if (keyCode == RIGHT) {     // Стрелка вправо
      x++;
    }
  }
  rect(x, 45, 50, 50);
}
```

Робот 3: Интерактивность



Эта программа использует переменные из Робота 2 (см. “Робот 2: Переменные” в главе 4) и изменяет их значения в процессе работы программы для управления формой робота посредством мыши. Код в блоке *draw*() прокручивается множество раз за секунду. Переменные, объявленные в программе, изменяются в каждом кадре в соответствии с *mouse*X и *mousePressed*.

*mouse*X управляет местоположением робота с использованием техники easing, это делает движения робота менее резкими, а значит более естественными. Если нажата кнопка мыши, длина шеи и туловища робота уменьшаются, делая робота меньше ростом.

```
float x = 60;          // X-координата
float y = 440;         // Y-координата
int radius = 45;       // Радиус головы
int bodyHeight = 160;  // Длина туловища
int neckHeight = 70;   // Длина шеи

float easing = 0.02;

void setup() {
  size(360, 480);
  smooth();
  strokeWeight(2);
```

```
  ellipseMode(RADIUS);
}

void draw() {

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

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

  float ny = y - bodyHeight - neckHeight - radius;

  background(204);

  // Шея
  stroke(102);
  line(x+12, y-bodyHeight, x+12, 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(102);
  ellipse(x, y-33, 33, 33);
  fill(0);
  rect(x-45, y-bodyHeight, 90, bodyHeight-33);

  // 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);
}
```

6/Мультимедиа

Возможности Processing не ограничиваются рисованием простых линий и фигур. Настало время узнать, как загружать изображения, векторные файлы и шрифты, чтобы работать с фотографиями, графикой и различными шрифтами.

Для хранения таких файлов Processing имеет папку *data*, поэтому вам не придется думать об их расположении, когда вы будете создавать скетч для ПК, интернета или мобильного устройства. Мы разместили медиафайлы для примеров из этой главы по адресу: http://www.processing.org/ learning/books/ media.zip.

Скачайте этот файл, распакуйте его на рабочий стол (или в любое другое удобное место) и не забудьте, где сохранили его.

ПРИМЕЧАНИЕ: Сделайте двойной щелчок по файлу, чтобы распаковать его на Mac OS X - будет создана папка *media*. На Windows, сделайте двойной щелчок по файлу *media.zip* - откроется окно. Перетащите папку *media* на рабочий стол.

Создайте новый скетч и выберите Add File из меню Sketch. В папке, которую вы только что распаковали, найдите файл *lunar.jpg* и выделите его. Если вы сделаете все правильно, в окне сообщений вы увидите: “1 file added to the sketch.”

Для проверки выберите Show Sketch Folder из меню Sketch. Вы увидите папку *data* с копией *lunar.jpg*. Папка data создается автоматически каждый раз, когда вы добавляете файл к скетчу. Вместо использования команды меню Add File вы можете сделать то же самое, перетащив файлы в область текстового редактора Processing. Файлы точно так же скопируются в папку *data* (а сама папка *data* будет создана, если ее не будет).

Вы можете создать папку *data* в любом месте и сохранять там свои файлы. Вы не будете получать сообщения о том, что добавлен новый файл, но если вы работаете с большим числом файлов, это неплохой метод.

ПРИМЕЧАНИЕ: По умолчанию, расширение файла на Windows и Mac OS X скрыто. Будет хорошей идеей сменить эту опцию и всегда видеть название файла полностью. На Mac OS X выберите Preferences в меню Finder, а затем убедитесь, что разделе Advenced выбрано “Show all filename extensions”. На Windows найдите “Folder Options” и установите соответствующую опцию.

Изображения

Перед тем, как вы выведете изображение на экран, вам следует выполнить несколько действий:

1. Добавьте изображение в папку *data* (см. инструкцию выше).

2. Объявите переменную *PImage* для хранения изображения.

3. Загрузите изображение в переменную функцией *loadImage*().

Пример 6-1: Загружаем изображение

После выполнения указанных трех шагов, вы сможете вывести на экран изображение с помощью функции *image*(). Первый параметр функции *image*() указывает нужное изображение, второй и третий x- и y-координаты:

```
PImage img;

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

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

Дополнительные четвертый и пятый параметры устанавливают ширину и высоту изображения. Если не указаны четвертый и пятый параметры изображения, ширина и высота изображения не изменятся.

В следующих программах мы покажем, как работать с несколькими изображениями в одной программе и как менять размер изображения.

Пример 6-2: Загружаем несколько изображений

Для этого примера вам понадобится добавить к вашему скетчу файл *capsule.jpg* любым из описанных выше способов (вы найдете его в недавно загруженной папке *media*).

```
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 6-3: Мышь и изображение

Если в качестве четвертого и пятого параметров функции *image*() записать *mouse*X и *mouse*Y, то размер изображения будет меняться вместе с местоположением курсора:

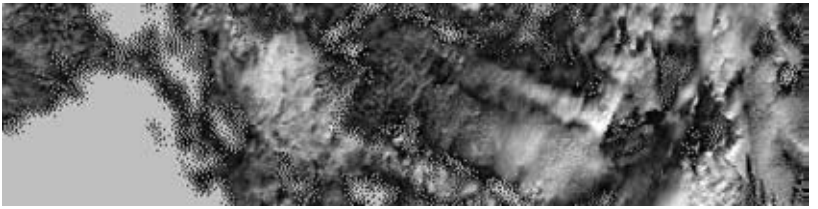
```
PImage img;

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

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

ПРИМЕЧАНИЕ: Когда изменяются первоначальные размеры изображения, оно может искажаться. Будьте внимательны при изменении размеров изображения. С помощью функции *image*() изображение может отображаться с разным размером, но файл изображения, сохраненный на вашем жестком диске, при этом не меняется.

Processing имеет возможность загружать и отображать изображения в форматах JPEG, PNG и GIF. (Векторная графика в формате SVG отображается другим способом, мы расскажем об этом далее в этой главе в разделе “Фигуры”.) С помощью таких программ, как GIMP и Photoshop вы сможете конвертировать свои изображения в форматы JPEG, PNG и GIF. Большинство современных цифровых фотоаппаратов сохраняют изображения в формате JPEG, но, как правило, нужно уменьшить их размер, прежде чем использовать в Processing. Обычный цифровой фотоаппарат создает изображение, в несколько раз большее по размеру, чем окно большинства скетчей в Processing, поэтому изменение размера изображения перед сохранением в папку *data* сделает ваши скетчи более эффективными и сохранит немного пространства на вашем жестком диске.

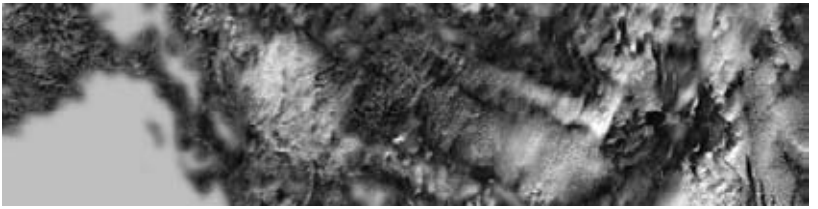
Изображения GIF и PNG поддерживают прозрачность, то есть пиксели могут быть полностью или частично невидимыми (см. обсуждение функции *color*() и величины альфа на стр. 26-29). Изображения GIF поддерживают 1-битную прозрачность - пиксели могут быть полностью прозрачными или полностью непрозрачными. Изображения PNG имеют 8-битную прозрачность - уровень прозрачности можно регулировать. Следующие примеры демонстрируют разницу между ними на примере *clouds.gif* и *clouds.png* из папки *media*. Убедитесь, что вы добавили их к скетчу перед запуском примеров.

Пример 6-4: Прозрачность в 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 6-5: Прозрачность в 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);
}
```

ПРИМЕЧАНИЕ: Не забудьте указать расширение файла *.gif*, *.jpg* или *.png* при добавлении изображения. Убедитесь также, что точно записали название файла, включая регистр. Если возникли сложности, прочитайте примечание о том, как сделать расширения файлов видимыми на Mac OS X и Windows.

Шрифты

Кроме шрифта, установленного по умолчанию, Processing может использовать множество других шрифтов. Прежде чем отображать текст в другом шрифте, вам потребуется ковертировать один из шрифтов на вашем компьютере в формат VLW, сохраняющий каждую букву как небольшое изображение. Для этого, выберите выберите Create Font в меню Tools, чтобы открыть диалоговое окно (рис. 6-1). Выберите шрифт, размер, и решите, хотите ли вы, чтобы края шрифта были сглаженными.

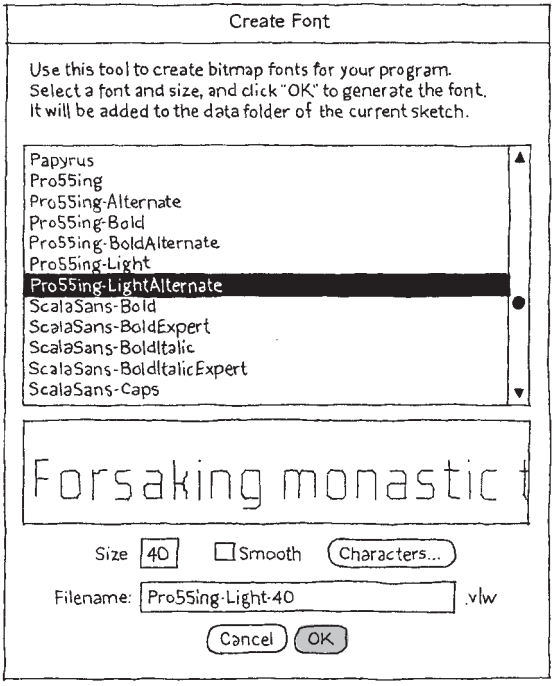


Рис. 6-1. Инструмент Create Font

ПРИМЕЧАНИЕ: При выборе размера шрифта учитывайте следующее: вы можете выбрать размер шрифта, который требуется для вашего скетча или выбрать больший размер, но помните, что размер файла для хранения шрифта при этом увеличится. Выбирайте опцию Characters только если вы используете нелатинский шрифт вроде японского или китайского, так как это тоже увеличивает размер файла.

Кликните OK в окне Create Font и ваш VLW-шрифт будет создан и сохранен в папку *data*. Теперь вы можете загрузить шрифт и добавить в скетч текст. Порядок добавления шрифтов аналогичен добавлению изображений, за исключением одного шага:

1. Добавьте шрифт в папку *data* (инструкция дана выше).
2. Объявите переменную *PFont* для хранения шрифтов.
3. Загрузите шрифт в переменную с помощью функции *loadFont*().
4. Установите текущий шрифт командой *textFont*().

Пример 6-6: Применяем различные шрифты

Теперь, с помощью функции *text*() вы можете изобажать буквы на экране, а с функцией *textSize*() менять размер шрифта. В этом примере вам потребуется применить инструмент Create Tool для создания VLW-шрифта (и изменить строчку l*oadFont*(), чтобы использовать его) или использовать *AndaleMono-36.vlw* из папки *media*:


```
PFont font;

void setup() {
  size(480, 120);
  smooth();
  font = loadFont("AndaleMono-36.vlw");
  textFont(font);
}

void draw() {
  background(102);
```

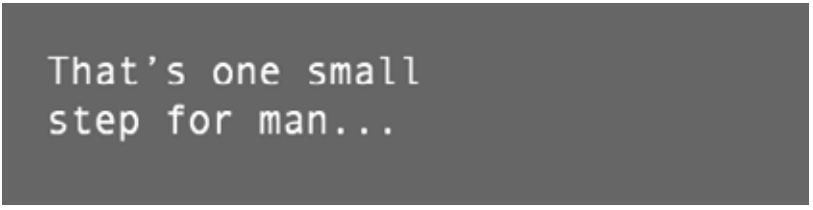
```
  textSize(36);
  text("That's one small step for man...", 25, 60);
  textSize(18);
  text("That's one small step for man...", 27, 90);
}
```

Первый параметр функции *text*() - это текст, который вы хотите написать на экране. (Заметьте, что символы заключены в кавычки.) Второй и третий параметр определяют положение по горизонтали и по вертикали. Положение задается относительно основной линии текста (см. рис. 6-2).

Рис. 6-2. Координаты текста.

Пример 6-7: Записываем текст в прямоугольнике

Добавьте четвертый и пятый параметры, чтобы записать текст в прямоугольник с определенной длиной и шириной:




```
PFont font;
void setup() {
  size(480, 120);
  font = loadFont("AndaleMono-24.vlw");
  textFont(font);
}

void draw() {
  background(102);
  text("That's one small step for man...", 26, 30, 240, 100);
}
```

Пример 6-8: Сохраняем текст в формате *String*

Текст в функции *text*() в предыдущем примере делает код трудно читаемым. Чтобы сделать код более компактным, мы сохраним этот текст в переменной. Текстовые данные сохраняются в формате *String*. Вот обновленная версия предыдущего примера с использованием формата *String*:

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

void setup() {
  size(480, 120);
  font = loadFont("AndaleMono-24.vlw");
  textFont(font);
}

void draw() {
  background(102);
  text(quote, 26, 30, 240, 100);
}
```

Существует несколько дополнительных функций для отображения букв на экране. В справке Processing, в разделе Typography, они описаны с примерами.

Фигуры

Если вы создаете векторную графику в таких программах как Inkscape или Illustrator, вы можете напрямую загрузить ее в Processing. Это удобно для таких фигур, которые вы бы не стали рисовать в Processing. Как и в случае изображений, прежде чем приступить к работе с ними, вы должны добавить их к скетчу.

Для загрузки и отображения файла SVG нужно выполнить три шага:

1. Добавьте SVG-файл в папку data.

2. Для хранения векторного файла создайте переменную *PShape*.

3. Загрузите векторный файл в переменную в помощью функции *loadShape*().

Пример 6-9: Рисуем фигуры

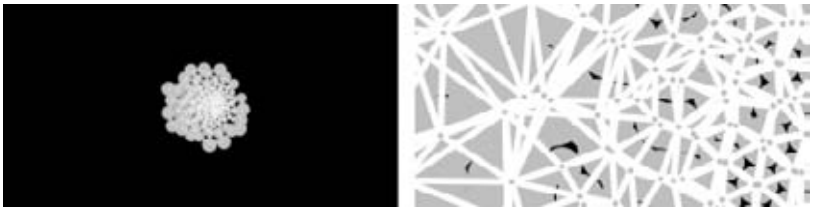
После выполнения этих шагов вы можете вывести изображение на экран функцией *shape*():

```
PShape network;

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

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

Параметры функции *shape*() аналогичны функции *image*(). Первый параметр - это название SVG-файла, следующие два устанавливают позицию изображения в окне. Дополнительные четвертый и пятый параметры задают длину и ширину изображения.

Пример 6-10: Масштабирование фигур

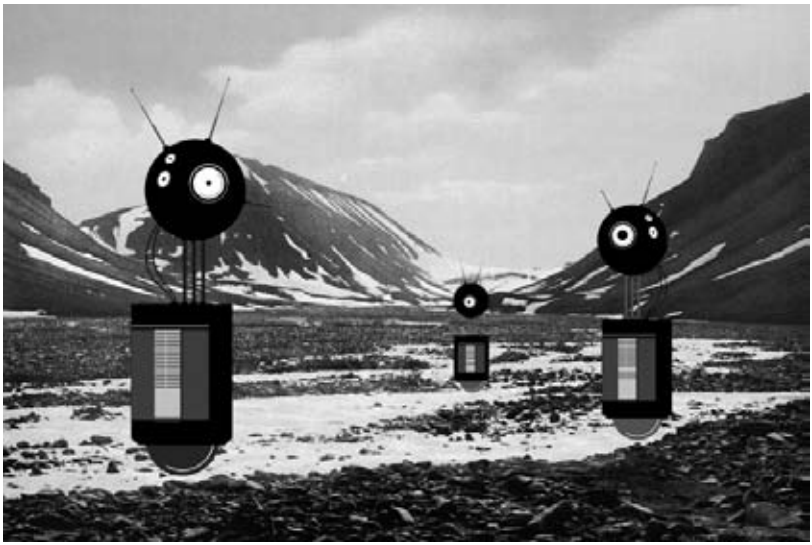
В отличие от растровых изображений, векторные фигуры могут быть увеличены до любого размера без ухудшения четкости. В этом примере масштабирование фигур осуществляется в соответствии с переменной *mouseX*, а для расположения фигуры в центре, а не в левом верхнем углу, как установлено по умолчанию, применена функция *shapeMode*():



```
PShape network;

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

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

ПРИМЕЧАНИЕ: Существуют ограничения для файлов SVG, которые вы загружаете в Processing. Processing поддержвает не все свойства SVG. За подробностями см. справку Processing по функции *PShape*.

Робот 4: Мультимедиа



В отличие от роботов, созданных из линий и прямоугольников в предыдущих главах, эти роботы созданы в специальной программе. Некоторые фигуры проще создать в таких программах как Inkscape или Illustrator, чем программировать фигуры с координатами.

Выбор техники для создания изображения как правило является компромиссом. Если фигура создана в Processing, то модифицировать ее в процессе работы программы легче. Если фигура создана где-то еще, то все, что вам остается модифицировать - это местоположение, наклон и размер. Загрузка робота в формате SVG делает невозможным изменения, которые мы делали для Робота 2 (см. “Робот 2: Переменные” в главе 4).

Изображения, созданные в других программах или полученные с фотокамеры загружаются в Processing. Используя это изображение в качестве фона, наши роботы могут исследовать формы жизни в Норвегии начала 20 века.

Файлы SVG и PNG, использованные в этом примере, вы можете загрузить здесь: 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");
  smooth();
}

void draw() {
  // Устанавливаем изображение "landscape" в качестве фона;
  // это изображение должно иметь размер окна
  background(landscape);

  // Устанавливаем сдвиг влево/вправо и применяем технику
  // easing для сглаживания движений
  float targetOffset = map(mouseY, 0, height, -40, 40);
  offset += (targetOffset - offset) * easing;

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

  // Рисуем правого робота уменьшенным с небольшим сдвигом
  float smallerOffset = offset * 0.7;
  shape(bot2, 510 + smallerOffset, 140, 78, 248);

  // Рисуем самого мальенького робота, с наименьшим сдвигом
  smallerOffset *= -0.5;
  shape(bot3, 410 + smallerOffset, 225, 39, 124);
}
```

7/Движение

Иллюзия движения на экране создается с помощью последовательности неподвижных изображений, сменяющих друг друга. Каждое изображение имеет небольшое отличие от другого. Эффект анимации достигается из-за инерционности зрения. Наш мозг воспринимает последовательность похожих изображений как движение.

Пример 7-1: Частота кадров

Processing "прокручивает" код в блоке *draw*() 60 раз в секунду для создания плавного движения. Чтобы убедиться в этом, запустите программу и понаблюдайте за значениями на консоли. Переменная *frameRate* контролирует скорость выполнения программы.

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

Пример 7-2: Устанавливаем частоту кадров

Функция *frameRate*() изменяет скорость выполнения программы. В следующей программе удалите знаки // и оцените результат:

```
void setup() {
  frameRate(30);    // 30 кадров в секунду
  //frameRate(12);  // 12 кадров в секунду
  //frameRate(2);   // 2 кадра в секунду
  //frameRate(0.5); // 1 кадр каждые 2 секунды
}
```

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

ПРИМЕЧАНИЕ: Processing запускает код 60 раз в секунду, но если для запуска блока *draw*() требуется больше времени, чем 1/60 секунды, частота кадров снизится. Функция *frameRate*() определяет максимальную частоту кадров, но реальная частота кадров зависит от компьютера, на котором запущена ваша программа.

Скорость и направление

Для написания примеров кода с плавным движением мы используем тип данных *float*. Этот тип переменной сохраняет данные с дробной частью, что обеспечивает более высокое разрешение для работы с движением. Например, когда мы используем тип *int*, самая маленькое расстояние, на которое может сместиться объект за один кадр, это один пиксель (1, 2, 3, 4, . . .), но с типом *float* вы можете передвигать объекты на любое растояние (1.01, 1.01, 1.02, 1.03, . . .).

Пример 7-3: Передвигаем фигуру

В следующем примере значение переменной x возрастает и фигура перемещается слева направо:

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

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

```
void draw() {
  background(0);
  x += speed; // Increase the value of x
  arc(x, 60, radius, radius, 0.52, 5.76);
}
```

Когда вы запустите этот код, вы заметите, что фигура выходит за правую границу окна, когда значение переменной x превышает его ширину. Величина x по-прежнему растет, но фигура уже не видна.

Пример 7-4: Разворот

Существует множество альтернативных вариантов примера 7-3, вы можете реализовать их в соответствии с вашими предпочтениями. Для начала мы попробуем усовершенствовать код так, чтобы фигура возвращалась к левому краю окна после ухода за правый край. Здесь окно представляется как развернутый цилиндр, вокруг которого вращается фигура:

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

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

void draw() {
  background(0);
  x += speed;                // Увеличение x
  if (x > width+radius) {    // Если фигура вышла за край окна,
    x = -radius;             // она возвращается к левому краю
  }
  arc(x, 60, radius, radius, 0.52, 5.76);
}
```

В каждом блоке *draw*() код сравнивает величину x с шириной экрана (плюс радиус фигуры). Если она больше, x присваивается отрицательное значение и фигура продолжает перемещаться слева направо от левой границы окна. На рисунке 7-1 изображен принцип работы программы.

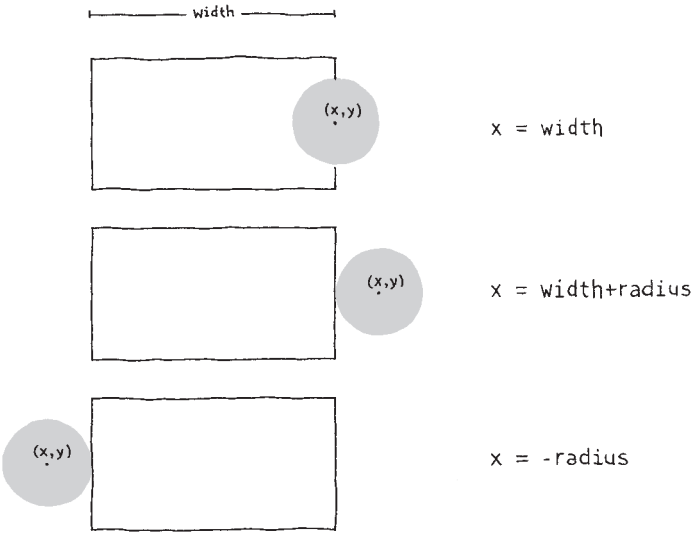


Рис. 7-1. Возврат фигуры к левому краю окна.

Пример 7-5: Фигура отскакивает от стенки

В это примере, вместо возврата фигуры к левому краю, мы добавим к примеру 7-3 возможность изменять направление движения фигуры по достижении края окна. Для этого мы объявим переменную, в которую запишем направление движения. Фигура движется вправо при значении 1, влево - при значении -1:

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

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

void draw() {
  background(0);
  x += speed * direction;
  if ((x > width-radius) || (x < radius)) {
    direction = -direction;  // Flip direction
  }
  if (direction == 1) {
    arc(x, 60, radius, radius, 0.52, 5.76); // Фигура смотрит вправо
  } else {
    arc(x, 60, radius, radius, 3.67, 8.9);  // Фигура смотрит влево
  }
}
```

Когда фигура достигает края, код изменяет знак переменной *direction*, изменяя направление движения. Например, если к моменту достижения границы значение переменной было положительно, код меняет его на отрицательное.

Промежуточные изображения

Иногда вам требуется перемещать фигуру из одной точки экрана в другую. С помощью нескольких строк кода вы можете задать начальную и конечную позицию объекта, а затем вычислить промежуточные позиции для каждого кадра.

Пример 7-6: Вычисляем промежуточные позиции

Чтобы сделать код из следующего примера более компактным, мы создали несколько переменных в начале кода. Запустите код несколько раз, изменяя значения этих переменных и посмотрите, как фигура перемещается из любой заданной точки в любую другую с различной скоростью. Переменная *step* регулирует скорость фигуры:

```
int startX = 20;     // Координата x начальной точки
int stopX = 160;     // Координата x конечной точки
int startY = 30;     // Координата y начальной точки
int stopY = 80;      // Координата y конечной точки
float x = startX;    // Текущая координата x
float y = startY;    // Текущая координата y
float step = 0.005;  // Величина шага (от 0.0 до 1.0)
float pct = 0.0;     // Доля от пройденного расстояния (от 0.0 до 1.0)

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

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

Случайные числа

В отличие от плавного, линейного движения, характерного для компьютерной графики, движение в реальном мире может быть любым. Представьте лист, падающий на землю, или муравья, ползущего по неровной дороге. Мы можем изобразить такие сложные, непредсказуемые траектории на компьютере с помощью генерации случайных чисел. Генерацию случайных чисел выполняет функция *random*(), от пользователя требуется всего лишь задать диапазон генерируемых чисел.

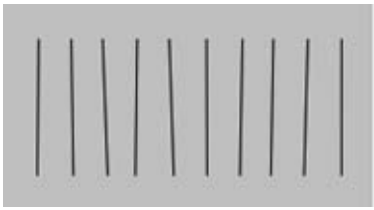
Пример 7-7: Генерируем случайные числа.

В следующем примере программа генерирует случайные числа и выводит их на консоль. Диапазон значений определяется позицией мыши. Функция *random*() всегда возвращает числа типа *float*, поэтому убедитесь, что переменная слева от оператора присвоения имеет тип *float*, как в этом примере:

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

Пример 7-8: Беспорядочное движение

Этот пример основан на примере 7-7, функция *random*() используется для изменения позиции линий на экране. Когда курсор находится в левой части окна, движение почти незаметно; по мере продвижения вправо диапазон случайных чисел, генерируемых функцией *random*(), увеличивается и линии двигаются с большим размахом. Функция *random*() заключена в блоке *for* и в каждой строке вычисляются новые координаты линий:



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

```
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 7-9: Программируем беспорядочное движение

Генератор случайных чисел можно использовать для создания более естественного движения фигур на экране. В следующем примере для изменения позиции круга на экране использованы случайные числа. Функция *background*() не включена в блок *draw*(), поэтому мы можем наблюдать траекторию движения фигуры:

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

void setup() {
  size(240, 120);
  smooth();
  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 фигурой достаточно длительное время, вы увидите, как круг выходит за пределы окна и возвращается назад. Выход круга за пределы окна - дело случая, однако мы можем добавить оператор *if* или функцию *constrain*(), чтобы круг оставался в окне. Функция *constrain*() ограничит числа до заданного диапазона, что позволит держать x и y в пределах окна. Сделав эти изменения в блоке *draw*(), вы сможете быть уверены, что круг останется в окне:

```
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);
}
```

ПРИМЕЧАНИЕ: Чтобы использовать одну и ту же последовательность случайных чисел при каждом запуске программы, существует функция *randomSeed*(). Подробное описание этой функции вы найдете в справке Processing.

Таймеры

Processing всегда отсчитывает время, прошедшее с запуска программы. Оно измеряется в миллисекундах (одна тысячная секунды); так, после секунды работы программы счетчик досчитает до 1000, после 5 сукунд - до 5000, после минуты - до 60000. Счетчик Processing удобно использовать для запуска анимированных фигур в определенное время. Функция *millis*() возвращает значение этого счетчика.

Пример 7-10: Длительность работы программы

Вы можете посмотреть, сколько времени прошло с момента запуска программы:

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

Пример 7-11: Запуск в заданный момент времени

В комплекте в оператором *if* значения из *millis*() можно использовать для запуска событий в определенном порядке. Например, вы можете запустить код в блоке *if* через 2 секунды. В следующем примере время изменения переменной x задается переменными *time1* и *time2*:

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

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

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

Движение по кругу

Если вы хорошо знакомы с тригонометрией, вы уже знаете, как прекрасны функции синус и косинус. Если нет, то мы надеемся заинтересовать вас с помощью нескольких примеров. Мы не будем обсуждать математические подробности, мы просто покажем несколько примеров для создания плавного движения.

На рисунке 7-2 изображена функция синус по отношению к углу. Обратите внимание, как изменяется синус по оси ординат в верхней и нижней частях волны, как он замедляется, а затем изменяет направление. Именно это качество кривой позволяет создать интересную траекторию движения.

Синус и косинус угла в Processing находятся в диапазоне от -1 до 1. Как и в функции *arc*(), углы следует задавать в радианах (см. пример 3-7 и 3-8 в качестве напоминания). При рисовании траекторий мы обычно умножаем числа типа *float*, которые возвращают функции *sin*() и *cos*() на некоторую величину.

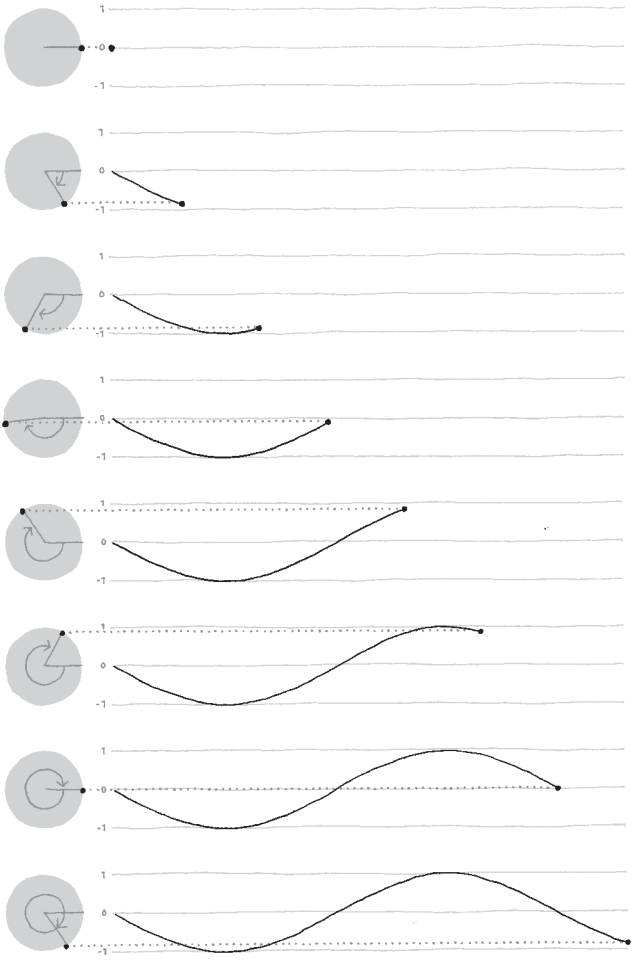


Рис. 7-2. Функции синус и косинус.

Пример 7-12: Синус

В этом примере мы покажем, как функция синус изменяется в диапазоне от -1 до 1 с увеличением угла. Вы можете установить другой диапазон значений синуса, записав число от 0 до 255 в функцию *map*(). Мы используем эту величину для программирования цвета фона в окне:

```
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 7-13: Движение по синусоидальной траектории

В следующем примере синус использован для создания траектории движения:

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

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

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 7-14: Движение по кругу

Для создания движения по кругу нужны функции *sin*() и *cos*(). x-координата изменяется по функции *cos*(), y-координата - по функции *sin*(). Обе функции умножены на переменную *scalar* для регулировки радиуса движения и суммированы с величиной, определяющей центр вращения:

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

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

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

Пример 7-15: Спирали

Сделаем небольшое изменение в программе, чтобы переменная *scalar* увеличивалась со временем, и получим спираль вместо окружности:

```
float angle = 0.0;
float offset = 60;
float scalar = 2;
float speed = 0.05;

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

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

Перемещение, вращение, масштабирование

Другой способ создать движение - изменять координаты окна. Например, вместо перемещения фигуры на 50 пикселей вправо вы можете сместить координату окна (0,0) вправо на 50 пикселей - визуально результат будет тем же. Мы можем перемещать, поворачивать, масштабировать фигуры простым изменением координат окна. Графическая интерпретация сказанного изображена на рисунке 7-3.

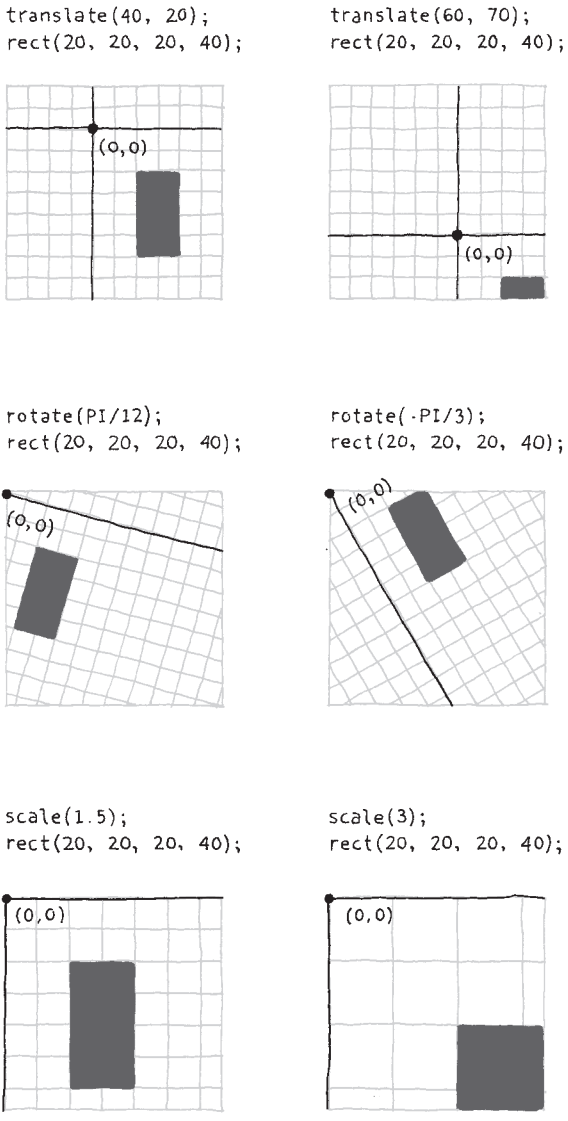


Рис. 7-3. Перемещение, поворот и масштабирование координат.

Смещение координат окна может показаться вам слишком сложным трюком, поэтому мы начнем с простой в использовании функции *translate*(). Она сдвигает координаты системы влево, вправо, вверх и вниз с помощью всего двух параметров.

Пример 7-16: Перемещение

В этом примере мы обращаем ваше внимание на то, что все прямоугольники изображены относительно точки (0,0) и перемещаются по экрану под воздействием функции *translate*():

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

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

Фактически, функция *translate*() вместо точки (0,0) записывает координаты курсора. В следующей строчке функция *rect*() рисует прямоугольник относительно новой нулевой точки, которой является координата мыши.

Пример 7-17: Перемещаем несколько фигур

Функция *translate*() применяется ко всем следующим за ней функциям рисования фигур. Но вот что происходит, когда мы добавляем еще одну команду *translate* для управления вторым прямоугольником:

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

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

Меньший прямоугольник рисуется относительно точки *mouse*X + 35 и *mouse*Y + 10.

Пример 7-18: Независимые смещения

Если вы не хотите, чтобы смещение одной фигуры влияло на позицию других фигур, нарисованных следом за ними, примените функции *push*M*atri*x() и *pop*M*atri*x(). При запуске функции *push*M*atri*x() происходит сохранение текущей координаты системы, функция *pop*M*atri*x() восстанавливает сохраненную координату:

```
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);
}
```

В этом примере меньший прямоугольник всегда будет находиться в левом верхнем углу так как действие функции *translate*(*mouse*X, *mouse*Y) прекращается командой *pop*M*atrix*().

ПРИМЕЧАНИЕ: Функции *push*M*atrix*() и *pop*M*atrix*() всегда используются вместе. Для каждой *push*M*atrix*() вам нужно применить соответствующую *pop*M*atrix*().

Пример 7-19: Вращение

Функция *rotate*() вращает систему координат окна. Для нее требуется один параметр - угол (в радианах). Вращение происходит относительно точки (0,0). Для того, чтобы запрограммировать вращение некоторой фигуры относительно ее центральной точки, сначала используйте функцию *translate*(), чтобы расположить фигуру там, где вы хотите, затем вызовите функцию *rotate*() и нарисуйте нужную фигуру с центром в точке (0,0):

```
float angle = 0.0;

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

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

Пример 7-20: Перемещение и вращение

В зависимости от того, в каком порядке вы записываете функции *translate*() и *rotate*(), результат будет различным. Следующий пример отличается от примера 7-19 тем, что функции *translate*() и *rotate*() записаны в обратном порядке. Здесь фигура вращается относительно левого верхнего угла окна, а функция *translate*() задает расстояние от угла до фигуры:

```
float angle = 0.0;

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

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

ПРИМЕЧАНИЕ: Чтобы упростить расположение фигур в центре окна, используйте функции *rect*Mode(), *ellipse*Mode(), *image*Mode() и *shape*Mode().

Пример 7-21: Масштабирование

Функция *scale*() позволяет масштабировать координаты окна. Как и *rotate*(), она работает относительно опорной точки. Следовательно, как и в случае функции *rotate*(), вам нужно сместить координаты окна с помощью функции *translate*(), масштабировать координаты, а затем нарисовать нужную фигуру с центром в точке (0,0).

```
float angle = 0.0;

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

void draw() {
  translate(mouseX, mouseY);
  scale(sin(angle) + 2);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}
```

Пример 7-22: Сохраняем толщину линий постоянной

Вы, наверное, заметили, что функция *scale*() влияет на толщину линий в примере 7-21. Чтобы толщина линий не изменялась в процессе масштабирования фигуры, разделите желаемую толщину линий на величину scalar:

```
float angle = 0.0;

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

void draw() {
  translate(mouseX, mouseY);
  float scalar = sin(angle) + 2;
  scale(scalar);
  strokeWeight(1.0 / scalar);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}
```

Пример 7-23: Движущийся манипулятор

В этом последнем и самом сложном примере мы воспользуемся последовательностью функций *translate*() и *rotate*() для создания сгибающегося и разгибающегося манипулятора. Каждая функция *translate*() перемещает нужные точки, а каждая фунцкия *rotate*() вращает их с заданным уголом поворота:

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

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

void draw() {
  background(204);
  translate(20, 25);  // Возврат к начальной позиции
  rotate(angle);
  strokeWeight(12);
  line(0, 0, 40, 0);
  translate(40, 0);   // Передвижение к следующему соединению
  rotate(angle * 2.0);
  strokeWeight(6);
  line(0, 0, 30, 0);
  translate(30, 0);   // Передвижение к следующему соединению
  rotate(angle * 2.5);
  strokeWeight(3);
  line(0, 0, 20, 0);

  angle += speed * angleDirection;
  if ((angle > QUARTER_PI) || (angle < 0)) {
    angleDirection *= -1;
  }
}
```

Мы не используем здесь *push*Matrix() или *pop*Matrix(), так как хотим, чтобы поворот и смещение распространялись на все сегменты манипулятора. Система координат устанавливается в начальное положение в начале каждого кадра, программируемого блоком *draw*().

Робот 5: Движение

Для программирования этого робота мы применили технику беспорядочного движения и движения по кругу. *background*() была удалена из программы чтобы нагляднее показать траекторию движения робота.

В каждом кадре к x-координате прибавлялось случайное число в диапазоне от -4 до 4 и к y-координате - случайное число в диапазоне от -1 до 1. Поэтому робот движется, в основном, по горизонтали, а не по вертикали.

Длина шеи изменяется в пределах от 50 до 110 пикселей в соответствии в функцией *sin*():

```
float x = 180;           // X-координата
float y = 400;           // Y-координата
float bodyHeight = 153;  // Длина туловища
float neckHeight = 56;   // Длина шеи
float radius = 45;       // Радиус головы
float angle = 0.0;       // Угол движения

void setup() {
  size(360, 480);
  smooth();
  ellipseMode(RADIUS);
  background(204);
}
```

```
void draw() {
  // Изменяем позицию на небольшое случайное число
  x += random(-4, 4);
  y += random(-1, 1);

  // Изменяем длину шеи
  neckHeight = 80 + sin(angle) * 30;
  angle += 0.05;

  // Корректируем высоту головы
  float ny = y - bodyHeight - neckHeight - radius;

  // Шея
  stroke(102);
  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(102);
  ellipse(x, y-33, 33, 33);
  fill(0);
  rect(x-45, y-bodyHeight, 90, bodyHeight-33);
  fill(102);
  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);
}
```

8/Функции

Функции являются основными строительными блоками программ Processing. Они встречались нам в каждом примере. Чаще всего мы применяли функции *size*(), *line*() и f*ill*(). В этой главе мы расскажем, как можно расширить возможности Processing путем создания новых функций.

Функции позволяют вам составлять программу из отдельных модулей, и это является их главным преимуществом. Функции работают как независимые блоки кода, которые вы можете использовать для построения более сложных программ - как из деталей конструктора LEGO, где каждая деталь сложной модели выполняет свою задачу. Как и в случае функций, преимущество такого подхода заключается в том, что вы можете составлять различные модели из типовых деталей. Один набор деталей, предназначенный для постройки космического корабля, может быть использован заново для постройки грузовика, небоскреба и многих других моделей.

Если вам требуется рисовать одну и ту же сложную фигуру наподобие дерева несколько раз, то вам будет удобнее использовать для этого особую функцию. Для написания своей функции вы можете использовать встроенные команды Processing типа *line*(). После того, как вы напишете код для своей фигуры, вам не придется вспоминать его, чтобы нарисовать ту же фигуру еще раз - вы просто напишете название своей функции, например *tree*().

Эти функции позволят вам сосредоточиться на целях более высокого уровня, например, рисовании дерева, вместо реализации деталей фигуры (команд типа *line*() для формирования дерева).

Если вы определите новую функцию, вам не придется повторять код этой функции каждый раз.

Как это работает

Компьютер выполняет программу последовательно, строка за строкой. Если запущена функция, компьютер перескакивает на фрагмент кода, в котором определена эта функция и выполняет этот код, а затем возвращается к прежнему месту.

Пример 8-1: Бросаем кости

Это действие иллюстрирует функция *rollDice*(), написанная специально для этого примера. Программа выполняет код в блоке *setup*() и останавливается. Но каждый раз, когда запускается фукция *rollDice*(), компьютер переходит к выполнению кода в блоке *rollDice*().

```
void setup() {
  println("Ready to roll!");
  rollDice(20);
  rollDice(20);
  rollDice(6);
  println("Finished.");
}

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

В двух строках кода функции *rollDice*() происходит выбор числа от 1 до числа сторон кубика, и эта величина выводится на консоль. При каждом новом запуске программы вы будете видеть разные числа, так как используется генератор случайных чисел:

```
Ready to roll!
Rolling... 20
Rolling... 11
Rolling... 1
Finished.
```

При каждом запуске функции *rollDice*() в блоке *setup*() код функции выполняется последовательно с первой строки до последней, а затем программа переходит к выполнению следующей строки в блоке *setup*().

Фунцкия *random*() (описание на странице 97) возвращает число от 0 до заданного числа (но не включая его). Так *random*(6) возвращает число от 0 до 5.99999 . . . . Мы применили *int*() для выделения целой части числа, так как *random*() возвращает числа типа *float*. Поэтому *int*(*random*(6)) будет возвращать 0, 1, 2, 3, 4 или 5. Мы прибавили 1, чтобы получить число от 1 до 6 (как если бы вы бросали кости). Когда функция *random*() возвращает числа от 0, с ними обычно проще работать; вы сможете убедиться в этом на множестве примеров из этой книги.

Пример 8-2: Второй способ

Если бы мы не использовали функцию *rollDice*(), программа выглядела бы так:

```
void setup() {
  println("Ready to roll!");
  int d1 = 1 + int(random(20));
  println("Rolling... " + d1);
  int d2 = 1 + int(random(20));
  println("Rolling... " + d2);
  int d3 = 1 + int(random(6));
  println("Rolling... " + d3);
  println("Finished.");
}
```

Код из примера 8-1, в котором мы применили функцию *rollDice*(), лучше читается и с ним проще оперировать. Смысл программы становится яснее, когда название функции отражает ее цель. В последнем примере мы видим функцию *random*() в блоке *setup*(), но общая цель программы не становится яснее. Взглянув на функцию, мы сразу можем сказать, сколько сторон в кости, которую мы бросаем: код *rollDice*(6) прямо указывает нам, что мы бросаем кость с шестью сторонами. Пример 8-1, помимо этого, удобен для модификации, так как информация не повторяется. Слово `Rolling...` выводится на консоль три раза. Если вам понадобится изменить этот текст, вам не придется редактировать один и тот же текст в трех местах, вы просто поменяете этот текст в функции *rollDice*(). Наконец, как вы увидите в примере 8-5, функции могут сделать ваш код намного короче (а следовательно удобнее и понятнее) и уменьшить количество потенциальных ошибок.

Создание функций

В этом разделе мы нарисуем сову, чтобы шаг за шагом показать, как создавать функции.

Пример 8-3: Рисуем сову

Сначала мы нарисуем сову, не создавая для этого специальных функций:

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

void draw() {
  background(204);
  translate(110, 110);
  stroke(0);
  strokeWeight(70);
  line(0, -35, 0, -65);  // Туловище
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35);  // Область левого глаза
  ellipse(17.5, -65, 35, 35);   // Область правого глаза
  arc(0, -65, 70, 70, 0, PI);   // Подбородок
  fill(0);
  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 функция *translate*() сдвигает опорную точку (0,0) на 110 пикселей в сторону и 110 пикселей вниз. Собственно сова рисуется относительно точки (0,0) и координаты фигур могут быть положительны или отрицательны. Пояснение на рисунке 8-1.

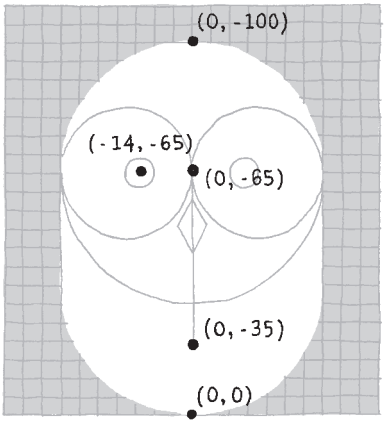


Рис. 8-1. Координаты совы

Пример 8-4: Компания из двух сов

Код из пример 8-3 хорош, если нам нужна всего одна сова, но если требуется вторая, длина кода увеличится примерно вдвое:

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

void draw() {
  background(204);

  // Левая сова
  translate(110, 110);
  stroke(0);
  strokeWeight(70);
  line(0, -35, 0, -65);  // Туловище
```

```
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35);  // Область левого глаза
  ellipse(17.5, -65, 35, 35);   // Область правого глаза
  arc(0, -65, 70, 70, 0, PI);   // Подбородок
  fill(0);
  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(0);
  strokeWeight(70);
  line(0, -35, 0, -65);  // Туловище
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35);  // Область левого глаза
  ellipse(17.5, -65, 35, 35);   // Область правого глаза
  arc(0, -65, 70, 70, 0, PI);   // Подбородок
  fill(0);

  ellipse(-14, -65, 8, 8);      // Левый глаз
  ellipse(14, -65, 8, 8);       // Правый глаз
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Клюв
}
```

Код первой совы скопирован и вставлен в код второй совы; чтобы сдвинуть вторую сову вправо на 70 пикселей добавлена функция *translate*(), в итоге количество строк в программе увеличилось с 21 до 34. Это довольно скучный и утомительный способ нарисовать вторую сову, не говоря уже о третьей. Но вам не обязательно повторять один и тот же фрагмент кода, потому что это именно та ситуация, где вам помогут функции.

Пример 8-5: Совиная функция

В этом примере для рисования двух сов применена одна функция. Если мы превратим код рисования совы в новую функцию, то этот код нужно будет набрать в программе только один раз:

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

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

void owl(int x, int y) {
  pushMatrix();
  translate(x, y);
  stroke(0);
  strokeWeight(70);
  line(0, -35, 0, -65);  // Туловище
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35);  // Область левого глаза
  ellipse(17.5, -65, 35, 35);   // Область правого глаза
  arc(0, -65, 70, 70, 0, PI);   // Подбородок
  fill(0);
  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();
}
```

Вы можете убедиться, что в примере 8-4 мы получали тот же результат, но этот пример короче, так как код рисования совы появляется в программе один раз, в функции *owl*(). Этот код будет запущен два раза, поскольку он дважды вызывается из блока *draw*(). В функции введены различные параметры x и y-координат, поэтому совы расположены в разных местах.

Параметры являются важной частью функций: они делают функции более гибкими. Мы убедились в этом на примере функции *rollDice*(), где благодаря параметру *numSides* мы можем имитировать бросание кости с 6-ю, 20-ю сторонами, или с любым количеством сторон. Все функции Processing работают по такому же принципу. Например, параметры функции *line*() позволяют провести линию между двумя произвольными точками на экране. Без использования параметров в функции *line*() мы не смогли бы нарисовать линию в любой области экрана.

Каждый параметр является переменной, которая объявляется при запуске функции, и поэтому он имеет тип данных (*int* или *float*). При первом запуске функции *owl*() значение переменной x было равно 110, переменной y - также 110. При втором запуске значение переменной x равнялось уже 180, а y - по-прежнему 110. Когда в коде функции встречается переменная, она заменяется соответствующим значением входного параметра, который вы укажете.

Убедитесь, что тип данных входного параметра функции совпадает с типом данных величины, которую вы указываете в качестве входного параметра. Рассмотрим функцию из блока *setup*() в примере 8-5:

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

Эти входные параметры функции заданы ошибочно, так как координаты x и y имеют тип *int*, а величины 110.5 и 120.2, указанные в этой функции имеют тип *float*.

Пример 8-6: Увеличиваем популяцию сов

Теперь, когда у нас есть функция для рисования совы в любом месте экрана, мы с легкостью можем нарисовать любое количество сов, записав функцию в цикле *for* так, чтобы первый параметр изменялся с каждым прогоном кода в цикле *for*:

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

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

// Сюда вставьте функцию owl() из примера 8-5
```

Вы можете добавить и другие параметры, изменяющие различные части рисунка совы. Эти параметры могут менять цвет, наклон, размер совы, диаметр ее глаз.

Пример 8-7: Совы разного размера

В этом примере мы добавили два параметра для изменения цвета и размера сов:

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

void draw() {
  background(204);
  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);  // Устанавливаем размер
  stroke(g); // Устанавливаем цвет
  strokeWeight(70);
  line(0, -35, 0, -65);  // Туловище
  noStroke();
  fill(255-g);
  ellipse(-17.5, -65, 35, 35);  // Область левого глаза
  ellipse(17.5, -65, 35, 35);   // Область правого глаза
  arc(0, -65, 70, 70, 0, PI);   // Подбородок
  fill(g);
  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();
}
```

Возвращаемые значения

Функции могут производить вычисления и возвращать результат в основную программу. Мы уже использовали функции такого типа, включая *random*() и *sin*(). Обратите внимание, что значение, возвращаемое этими функциями, присваивается переменной:

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

Здесь *random*() возвращает число в промежутке между 1 и 10 и затем оно присваивается переменной r.

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

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

Числа из функции *random*() не присваиваются переменной в этом примере - они выступают в роли входных параметров функции *point*() и используются для расположения точки в случайном месте окна.

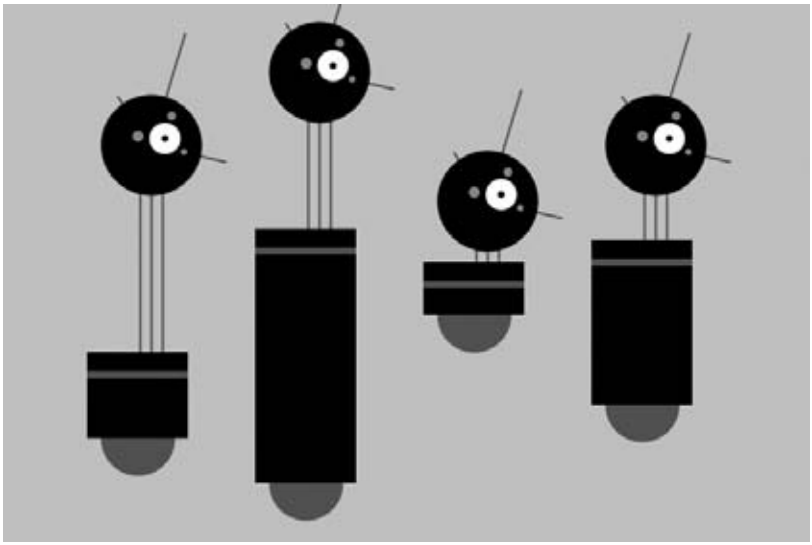
Пример 8-8: Получаем результат

Для создания функции, возвращающей некоторое число, замените слово v*oid* на тип данных, которые будет возвращать функция. С помощью команды *return* определите данные, которые вы хотели бы получить из вашей функции. Для примера рассмотрим функцию *calculate*M*ars*(), вычисляющую вес человека или какого-либо предмета на соседней планете:

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

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

Обратите внимание на слово *float* перед именем функции - это значит, что функция возвращает данные с плавающей точкой. В последней строке блока указывается, что функция возвращает значение переменной *new*W*eight*. Во второй строке блока *setup*() это значение присваивается переменной *mars*W*eight*. (Введите свой вес в переменную *your*W*eight*, чтобы узнать вес своего тела на Марсе).

Робот 6: Функции



В отличие от Робота 2 (см. “Робот 2: Переменные” в главе 4), в этом примере мы использовали функции, чтобы нарисовать четырех разных роботов в одной программе. Функция *drawRobot*() встречается в блоке *draw*() четыре раза и запускается четыре раза, каждый раз с новым набором параметров, регулирующих местоположение и высоту робота.

Заметьте, что глобальные переменные из Робота 2 теперь находятся в функции *drawRobot*(). Эти переменные используются исключительно для рисования робота и поэтому входят в код функции *drawRobot*(), заключенный в фигурные скобки. Переменная *radius* не изменяется в ходе выполнения программы, поэтому мы не определили ее как параметр. Вместо этого мы объявили ее в начале функции *drawRobot*():

```
void setup() {
  size(720, 480);
  smooth();
  strokeWeight(2);
  ellipseMode(RADIUS);
}

void draw() {
  background(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;

  // Шея
  stroke(102);
  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(102);
  ellipse(x, y-33, 33, 33);
  fill(0);
  rect(x-45, y-bodyHeight, 90, bodyHeight-33);
  fill(102);
  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);
  ellipse(x, ny-8, 5, 5);
  ellipse(x+30, ny-26, 4, 4);
  ellipse(x+41, ny+6, 3, 3);
}
```

9/Объекты

Объектно-ориентированное программирование (ООП) - это иной подход к созданию программ. Термин “объектно-ориентированное программирование” звучит несколько пугающе, но у нас есть хорошая новость: вы начали работать с объектами еще в 6 главе, когда применяли *PImage*, *PFont*, *String* и *PShape*. В отличие от обычных типов данных вроде *boolean*, *int* и fl*oat*, которые содержат одну величину, объекты могут содержать множество. Но это только верхушка айсберга. Объекты позволяют объединять переменные с соответствующими функциями. Вы уже знаете, как работать с переменными и функциями; объекты объединят все, что вы знаете в более понятный пакет.

Объекты ценны тем, что превращают идеи в маленькие строительные блоки. Это похоже на живой организм, где органы состоят из тканей, ткани из клеток и т.д. По мере усложнения кода вам все больше приходится оперировать сложными структурами, состоящими из более простых. Гораздо проще составлять код из небольших, понятных фрагментов, нежели писать длинный, подробный код.

Объект в мире программирования - это комплекс переменных и функций. В контексте объектов, переменная называется *полем данных* (*поле класса*, *артибут* или *переменная-член*), а функция называется *метод*. Поля данных и методы работают точно так же, как переменные и функции, которыми мы занимались в предыдущих главах; сейчас мы просто применим новые термины, чтобы показать, что они являются частью объектов.

Иначе говоря, объекты комбинируют данные, относящиеся к задаче (поля данных) и действия над ними (методы). Идея заключается в том, чтобы собрать воедино нужные данные и операции, совершаемые с этими данными. Чтобы, к примеру, смоделировать радио, представим, какие параметры нам придется настраивать и какие действия для этого нужны:

Поля данных: громкость, частота, модуляция (FM, AM), питание (выкл, вкл).
Методы: установка громкости, установка частоты, установка модуляции.

Сравним моделирование работы простого механического устройства и живого организма типа муравья или человека. Невозможно сократить всю сложность живого организма до нескольких полей данных и методов; однако можно создать модель упрощенную, но достаточную для того, чтобы симулировать интересный процесс. Хороший пример тому - видеоигра Sims. Игра заключается в управлении виртуальными персонажами. Персонаж обладает достаточным количеством возможностей, чтобы сделать игру с ним увлекательной, но не более того. Фактически, им присущи пять индивидуальных качеств: аккуратность, дружелюбие, активность, игривость и вежливость. С учетом возможности создания упрощенной модели сложного организма, мы попробуем запрограммировать муравья с помощью нескольких полей данных и методов:

Поля данных: тип (рабочий, солдат), вес, длина.
Методы: передвижение, захват, выброс феромонов, питание.

При составлении полей данных и методов вам придется ограничиться небольшой частью всех возможностей муравья. Но помните: не существует правильной или неправильной модели, ведь каждая модель создается для своей цели.

Объекты и классы

Прежде чем создать объект, вам нужно определить класс. Класс - это спецификация объекта. Используя архитектурную аналогию, класс является чертежом дома, а объектом является собственно дом. Дома, потроенные по чертежу, могут отличаться друг от друга, ведь чертеж это всего лишь описание дома, а не готовое здание. Один дом может быть голубым, а другой красным, в одном может быть устроен камин, а в другом - нет. Подобно объектам, класс определяет типы данных и операции, но каждый объект (дом), построенный на основе класса (чертежа) имеет переменные (цвет, наличие камина) принимающие разные значения. Выражаясь технически, каждый объект представляет собой пример класса; пример, в свою очередь, имеет свои поля данных и методы.

Определяем класс

Перед записью класса мы рекомендуем составить небольшой план. Подумайте, какие поля данных и методы будут в вашем классе. Устройте небольшой мозговой штурм: перечислите все возможные операции, выберите нужные и подумайте, как это будет работать. По ходу программирования вы будете делать изменения, но хороший старт не помешает.

Выберите информативные имена и тип данных для полей данных. Поля данных в классе могут иметь любой тип данных. Класс может одновременно содержать данные типа *boolean*, *float*, *image*, *string*, и т.д. Помните, что классы создаются для того, чтобы сгруппировать данные для одной задачи. Для методов также выберите информативные имена и решите, какие значения будут возвращаться (если будут). Методы изменяют значения полей данных и совершают операции с данными.

Преобразуем пример 7-9, рассмотренный ранее, в класс. Начнем с составления списка полей из примера:

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

Следующий шаг - посмотреть, какие методы могут быть полезны для класса. Глядя на функцию *draw*() из примера, мы видим два главных действия. Происходит изменение позиции фигуры с последующим отображением ее на экране. Создадим два метода для нашего класса:

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

Ни одно из этих действий не возвращает число, поэтому оба имеют тип возвращаемого числа v*oid*. Теперь для создания класса на основе списка полей данных и методов нужно выполнить четыре шага:

1. Создать блок.

2. Добавить поля.

3. Написать конструктор (см. далее) для присвоения значений полям данных.

4. Добавить методы.

Сначала создадим блок:

```
class JitterBug {

}
```

Обратите внимание, что команда *class* написана строчными буквами, а имя *JitterBug* - с прописными. Записывать имя класса прописными буквами не обязательно, но это делается (и мы рекомендуем это вам) для обозначения класса. (Команда *class*, однако, должна быть написана строчными, таково правило этого языка программирования.)

Далее добавим поля. Для этого нужно решить, какие поля будут использовать конструктор, - специальный метод, созданный для этих целей. На практике те значения полей, которые должны быть разными для каждого класса проходят через конструктор, а остальные значения могут быть присвоены при объявлении. Для *JitterBug* мы решили, что x, y и *diameter* будут проходить через конструктор. Вот как мы определили поля:

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

Теперь мы добавляем конструктор. Конструктор всегда имеет то же имя, что и класс. Задача конструктора - присвоить начальные значения полям при создании объекта (примера класса), см. рис. 9-1. При создании объекта запускается код в блоке конструктора. Как обсуждалось ранее, при инициализации объекта мы пропускаем через конструктор три параметра. Каждое значение присваивается временной переменной, которая существует только во время выполнения кода в конструкторе. Для прояснения мы добавили имя *temp* к каждой из этих переменных, но вы можете назвать эти величины как хотите. Они используются для присвоения значения полям, являющимся частью класса. Заметьте также, что конструктор не возвращает никаких величин, поэтому перед ним не требуется указывать тип данных вроде *void*. 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;
  }

}
```

Наконец мы добавляем методы. Это просто, почти как написание функций, но здесь они заключены в класс. Не забудьте про пробелы. Каждая строка в классе записана с отступом на несколько пробелов чтобы показать, что она является содержимым блока. В конструкторе и методах код размечается отступами, чтобы ясно 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);
  }

}
```

Рис. 9-1. Величины в конструкторе.

Пример 9-1: Создание объекта

Мы определили класс и чтобы использовать его, нам нужно создать объект из этого класса. Это делается в два шага:

1. Объявляется переменная объекта.

2. Ключевым словом *new* создается (инициализируется) объект.

Продемонстрируем, как это работает на примере скетча Processing, а затем объясним подробнее каждую часть:

```
JitterBug bug;  // Создание объекта

void setup() {
  size(480, 120);
  smooth();
  // Создание объекта с параметрами
  bug = new JitterBug(width/2, height/2, 20);
}

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

// Здесь - копия класса JitterBug
```

Каждый класс по сути - тип данных, каждый объект - переменная. Мы объявляем переменные объекта так же, как простые типы данных, например *boolean*, *int* и f*loat*. Для объявления объекта указывается сначала тип данных, затем имя переменной:

```
JitterBug bug;
```

Далее инициализируем объект с помощью ключевого слова *new*. Это зарезервирует память в компьютере под поля. За именем конструктора, записанным справа от ключевого слова *new*, следуют входные параметры конструктора, если таковые имеются:

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

Три числа в скобках являются тремя входными параметрами конструктора класса J*itter*B*ug*. Количество и типы данных параметров должны совпадать с указанными в конструкторе.

Пример 9-2: Создадим несколько объектов

В примере 9-1 мы видим еще кое-что: знак препинания (точка), который используется для доступа к методам объекта в блоке *draw*(). Оператор “точка” используется для объединения имени объекта с его полями данных и методами. Это наглядно показано в данном примере, где два объекта принадлежат к одному классу. Команда j*it.move*() относится к методу *move*() и объекту с именем j*it*, а *bug.move*() относится к тому же методу *move*() и объекту *bug*:

```
JitterBug jit;
JitterBug bug;

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

// Здесь - копия класса JitterBug
```

Теперь класс существует отдельно от остальной программы и любые изменения в нем приведут к изменениям объектов из этого класса. Например, вы можете добавить к *JitterBug* поле данных, управляющее цветом или размером. Эти величины могут быть введены с помощью конструктора или каких-либо других методов типа *setColor*() или *setSize*(). И, поскольку класс *JitterBug* - независимый модуль, вы можете использовать его в другом скетче.

Теперь будет полезным узнать о вкладках в среде Processing (рис. 9-2). Вкладки позволяют вам распределять код по нескольким файлам. Так вам легче будет управлять и редактировать длинный код. Обычно для каждого класса создается своя вкладка: это упрощает работу с классами по модульному принципу и позволяет лучше ориентироваться в коде.

Для создания вкладки кликните на стрелку в правой части панели вкладок. Если вы выберете в меню New Tab, вам будет предложено назвать новую вкладку в окне сообщений. Попробуйте улучшить код из этого примера, создав для класса *JitterBug* новую вкладку.

ПРИМЕЧАНИЕ: Каждая вкладка отображает отдельный .pde файл

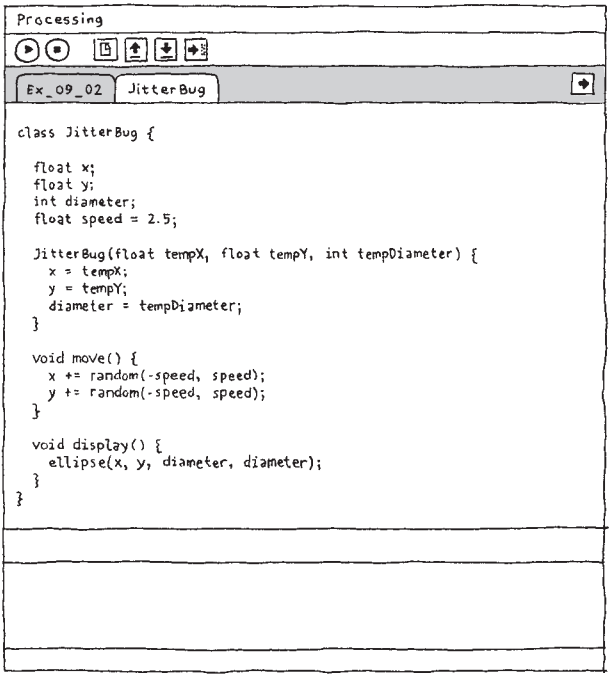
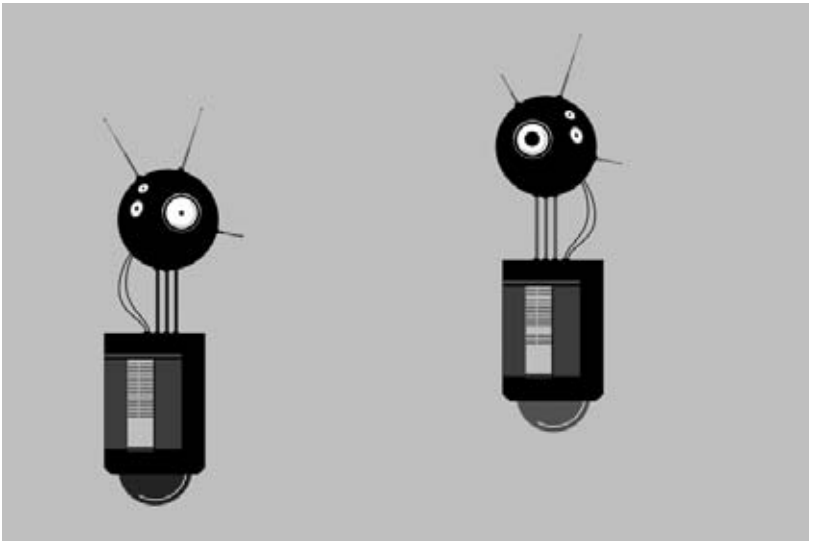


Рис. 9-2. Для удобства код можно разделить на несколько вкладок

Робот 7: Объекты



Программный объект сочетает в себе методы (функции) и поля данных (переменные). Класс R*obot* определяет все объекты (всех роботов) из этого класса. Каждый объект класса R*obot* содержит поля данных для хранения позиции и изображения, выводимого на экран. Каждый также содержит методы для обновления позиции и изображения на экране.

Параметры из *bot1* и *bot2* в блоке *setup*() определяют x и y-координаты, а файл *.svg* используется для отбражения робота. Параметры *temp*X и *temp*Y проходят через конструктор и присваиваются полям данных x*pos* и *ypos*. Параметр *svg*N*ame* используется для загрузки соответствующего изображения. Объекты (*bot1* и *bot2*) рисуются каждый на своем месте с помощью различных изображений; каждый из них имеет свои значения, проходящие через конструкторы:

```
Robot bot1;
Robot bot2;

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

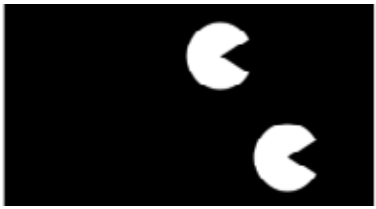
```
void draw() {
  background(204);

  // 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 конструктора
  Robot(String svgName, float tempX, float tempY) {
    botShape = loadShape(svgName);
    xpos = tempX;
    ypos = tempY;
    angle = random(0, TWO_PI);
  }

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

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

10/Массивы

В каждой из предыдущих глав мы разрабатывали новую программную идею (переменные, функции, объекты) и наконец дошли до последнего шага - массивов! Массив - это список переменных под одним именем. Массивы позволяют работать с большим количеством переменных, не объявляя каждую по отдельности. Это делает код хорошо читаемым, лаконичным и удобным для редактирования.

Пример 10-1: Набор переменных

Чтобы понять, что мы имели в виду, рассмотрите пример 7-3. Этот код отлично работает для одной фигуры, но что если мы хотим перемещать две? Придется нам объявить новую переменную x и обновлять ее в блоке *draw*():



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

void setup() {
  size(240, 120);
  smooth();
  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 10-2: Очень много переменных

Код из предыдущего примера можно редактировать, но что если мы хотим иметь 5 фигур? Придется добавить еще три переменные к уже имеющемуся коду:

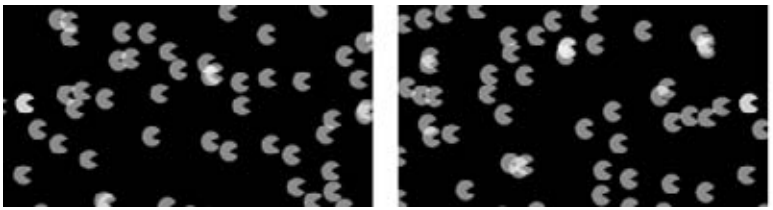
```
float x1 = -10;
float x2 = 10;
float x3 = 35;
float x4 = 18;
float x5 = 30;

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

Похоже, этот код начинает выходить из под контроля.

Пример 10-3: Массивы вместо переменных

Только представьте, что будет, если мы захотим 3000 фигур. Это значило бы объявление 3000 переменных и обновление каждой по отдельности. Сможете ли вы уследить за таким количеством переменных? И хотели бы вы это делать? Давайте используем массив вместо этого:



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

void setup() {
  size(240, 120);
  smooth();
  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 главы мы посвятим обсуждению деталей, которые сделали этот код возможным.

Объявление массива

Каждый компонент массива называется э*лемент*; элементы имеют *индекс* для обозначения позиции в массиве. Как и значения координат на экране, значения индекса начинаются от 0. Например, первый элемент массива имеет индекс 0, второй - 1 и т.д. Если в массиве 20 элементов, индекс последнего равен 19. Структура массива показана на рисунке 10-1.

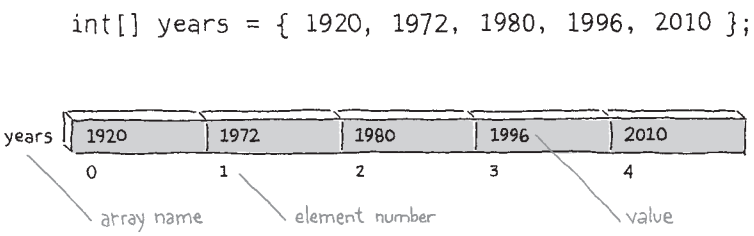


Рис. 10-1. Массив - это список из одной или нескольких переменных под одним именем.

Работа с массивами в сущности не отличается от работы с переменными; вы можете использовать те же приемы. Как вы уже знаете, переменная x типа integer объявляется так:

```
int x;
```

Для объявления массива вам осталось добавить скобки после типа данных:

```
int[] x;
```

Прелесть объявляения массивов заключается в возможности объявить 2, 10 или 100000 переменных одной строкой кода. Вот, к примеру, строка кода, объявляющая массив из 2000 переменных типа integer:

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

Вы можете создавать массивы из любых типов данных Processing: *boolean*, *float*, *String*, *PShape* и т.д., а также из любых классов, которые вы определите. Следующий пример объявляет массив из 32 переменных типа *PImage*:

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

Объявляя массив, начните с типа данных и скобок, следующих за ним. Далее следует имя массива, оператор присвоения (знак равенства), ключевое слово *new* и снова тип данных с размером массива в скобках. Этот алгоритм работает для объявления массивов любого типа.

ПРИМЕЧАНИЕ: Массив может хранить данные только одного типа (*boolean*, *int*, fl*oat*, *PImage*, и т.д.). Вы не можете объединять в одном массиве данные разных типов. Вместо этого применяйте объекты.

Давайте остановимся и подробно разберемся, как работать с массивами перед тем как идти дальше. Для объявления с массивов вам потребуется выполнить три шага, так же как и с объектами:

1. Объявите массив и укажите тип данных.

2. С помощью ключевого слова *new* создайте массив и укажите его длину.

3. Присвойте каждому элементу значения.

Эти действия могут быть выполнены в три строки или собраны в одну. Каждый из следующих трех примеров покажет вам вариант объявления массива x для хранения двух целых чисел, 12 и 2. Обратите внимание на то, что происходит перед блоком *setup*() и в самом блоке *setup*().

Пример 10-4: Объявление и присвоение

Для начала объявим массив вне блока *setup*() и присвоим ему значения в блоке *setup*(). Выражение x[0] относится к первому элементу массива, x[1] - ко второму:

```
int[] x;           // Объявление массива

void setup() {
  size(200, 200);
  x = new int[2];  // Создание массива
  x[0] = 12;       // Присвоение первого значения
  x[1] = 2;        // Присвоение второго значения
}
```

Пример 10-5: Объявление массива в краткой форме

Здесь объявление и создание массива выполняется в более краткой записи - в одну строку; присвоение значений происходит в блоке *setup*():

```
int[] x = new int[2];  // Объявление и создание массива

void setup() {
  size(200, 200);
  x[0] = 12;           // Присвоение первого значения
  x[1] = 2;            // Присвоение второго значения
}
```

Пример 10-6: Присвоение значений в один прием

Если значение массива можно записать одним выражением, вы также можете присвоить значения массиву при его создании:

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

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

ПРИМЕЧАНИЕ: Нежелательно создавать массивы в блоке *draw*(), так как создание массива в каждом кадре уменьшит частоту кадров.

Пример 10-7: Открывая заново первый пример

В качестве образца использования массивов мы поместили сюда пример 10-1. Здесь мы увидим некоторые важные особенности работы с массивами, хотя и пропустим хорошие приемы из примера 10-3:

```
float[] x = {-20, 20};

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

void draw() {
  background(0);
  x[0] += 0.5;  // Увеличение первого элемента
  x[1] += 0.5;  // Увеличение второго элемента
  arc(x[0], 30, 40, 40, 0.52, 5.76);
  arc(x[1], 90, 40, 40, 0.52, 5.76);
}
```

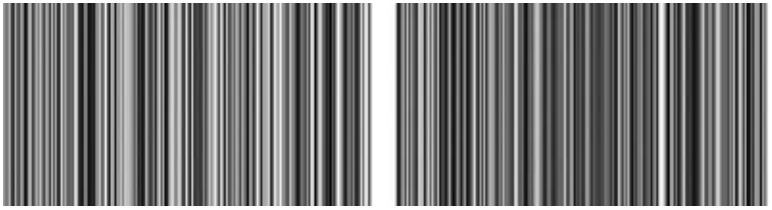
Повторения и массивы

Цикл *for*, представленный в разделе “Циклы” главы 4 упрощает работу с большими массивами, сокращая объем кода. Идея в том, чтобы написать цикл, перебирающий элементы массива один за другим. Для этого нужно знать длину массива. Поле *length* хранит количество элементов некоторого массива. Для доступа к этой величине мы используем оператор “точка” и имя массива. Например:

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

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

Пример 10-8: Заполнение массива в цикле *for*

Цикл *for* может использоваться для заполнения массива числами или для чтения чисел из него. В этом примере массив заполняется случайными числами, а затем эти числа используются в блоке *setup*() для установки толщины линий в блоке *draw*(). При каждом запуске в массиве формируется 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 10-9: Отслеживаем движения мыши

В этом примере есть два массива для хранения позиции мыши - один для координаты x, другой - для координаты y. В этих массивах хранится местоположение мыши за последние 60 кадров. С каждым кадром прежние значения x и y координат удаляются и заменяются текущими значениями *mouse*X и *mouse*Y. Новые значения размещаются на первой позиции массива, но перед этим каждое значение массива сдвигается вправо на одну позицию (от новых к старым), освобождая место для новых чисел. Этот пример наглядно демонстрирует описанное.

Также в каждом кадре используются координаты последних 60 кадров для рисования на экране эллипсов:

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

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

void draw() {
  background(0);
  // Сдвиг массива чисел на одну позицию
  for (int i = x.length-1; i > 0; i--) {
    x[i] = x[i-1];
    y[i] = y[i-1];
  }
  x[0] = mouseX;  // Установка первого элемента
  y[0] = mouseY;  // Установка первого элемента
  for (int i = 0; i < x.length; i++) {
    fill(i * 4);
    ellipse(x[i], y[i], 40, 40);
  }
}
```

ПРИМЕЧАНИЕ: В этом примере и на рисунке 10-2 показан способ хранения буфера сдвига в массиве. Но он менее эффективен, чем альтернатиный способ с использованием оператора % (деление по модулю). Этот способ поясняется в примере, включенном в Processing: Examples→Basics→Input→StoringInput.

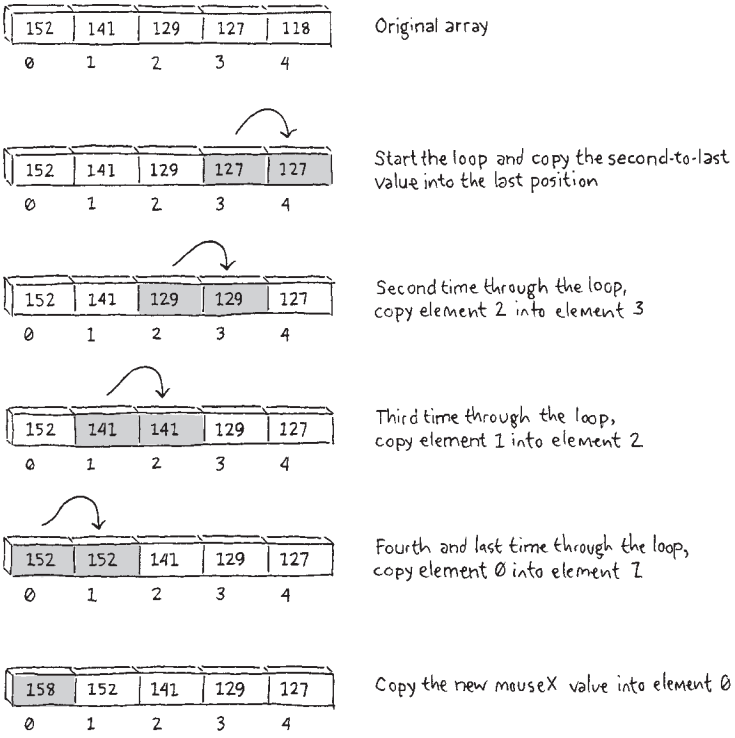


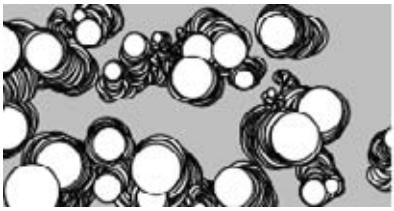
Рис. 10-2. Сдвиг чисел в массиве на одну позицию вправо.

Массивы объектов

Два коротких примера из этого раздела содержат все основные приемы, изложенные в этой книге: переменные, повторения, условия, функции, объекты и массивы. Массивы из объектов почти ничем не отличается от массивов из элементов, предствленных ранее, за исключением одного соображения: так как каждый элемент является объектом, он должен быть создан ключевым словом *new* (как и любой другой объект) перед присвоением массиву. С классом, определенным пользователем, таким, например, как *JitterBug* (см. главу 9), требуется установить каждый элемент с помощью ключевого слова *new* перед присвоением его массиву. Со встроенным в Processing классом вроде *PImage* перед присвоением нужно использовать функцию *loadImage*() для создания объекта.

Пример 10-10: Управление множеством объектов

В этом примере создается массив из 33 объектов класса *JitterBug*, которые обновляются и отображаются в блоке *draw*(). Чтобы этот пример работал, к коду нужно добавить класс *JitterBug*:

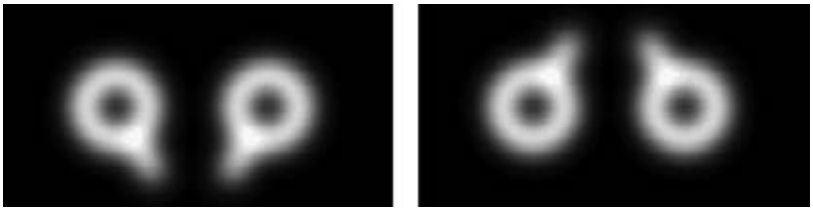


```
JitterBug[] bugs = new JitterBug[33];

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

// Сюда скопируйте класс JitterBug
```

Последний пример с массивами загружает последовательность изображений и сохраняет ее как массив объектов класса *PImage*.

Пример 10-11: Последовательность изображений

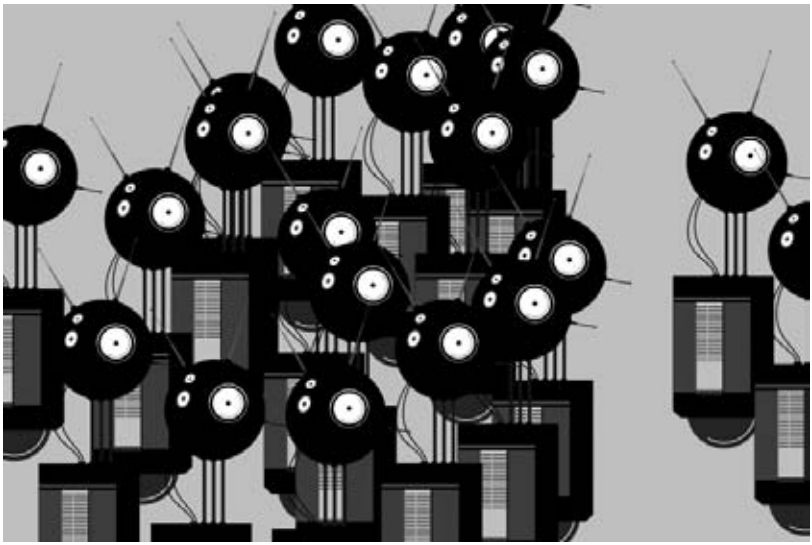
Для запуска этого примера распакуйте изображения из файла media.zip как это описано в главе 6. Изображения будут пронумерованы (frame-0001.png, frame-0002.png, и т.д.), что позволит нам вызывать файлы по имени непосредственно в цикле *for* в восьмой строке:



```
int numFrames = 12;  // Число кадров
PImage[] images = new PImage[numFrames];  // Создание массива
int currentFrame = 1;

void setup() {
  size(240, 120);
  for (int i = 1; i < images.length; i++) {
    String imageName = "frame-" + nf(i, 4) + ".png";
    images[i] = loadImage(imageName);  // Загрузка изображения
  }
  frameRate(24);
}

void draw() {
  image(images[currentFrame], 0, 0);
  currentFrame++;      // Следующий кадр
  if (currentFrame >= images.length) {
    currentFrame = 1;  // Возврат к первому кадру
  }
}
```

Функция *nf*() форматирует числа так, что *nf*(1, 4) возвращает строку “0001”, а nf(11, 4) возвращает “0011”. Эти значения объединяются с началом имени файла (“frame-”) и окончанием (“.png”), образуя полное имя файла в виде переменной типа *String*. В следующей строке файлы загружаются в массив. В блоке *draw*() изображения последовательно отображаются на экране. После отображения последнего изображения программа возвращается к началу массива и снова отображает последовательность изображений.

Робот 8: Массивы



Массивы упрощают написание программ с большим количеством элементов. В этом примере в начале кода объявлен массив объектов класса Robot. Затем массив появляется в блоке *setup*(), где в цикле *for* создается объект класса Robot. Другой цикл *for* применяется в блоке *draw*() для обновления и отображения каждого элемента массива *bots*.

Комбинация цикла *for* и массива - это удобный инструмент. Обратите внимание, что несмотря на небольшое различие в коде этого примера и примера Робот 7 (см. “Робот 7: Объекты” в главе 9) результаты заметно отличаются. Как только вы создали массив и написали цикл *for*, работа с 3000 элементами станет так же проста, как с 3.

Принципиальное отличие от Робота 7 заключается в том, что SVG файл загружен в блоке *setup*(), а не в классе Robot. Поэтому файл загружается один раз, а не множество - по количеству элементов массива (в нашем случае 20 раз). Этот код будет быстрее запускаться и занимать меньше оперативной памяти, так как загрузка файла требует времени, а нам требуется загрузить и сохранить всего один файл. Каждый элемент массива *bot* обращается к одному и тому же файлу.

```
Robot[] bots;  // Объявление массива элементов класса Robot

void setup() {
  size(720, 480);
  PShape robotShape = loadShape("robot1.svg");
  // Создание массива элементов класса Robot
  bots = new Robot[20];
  // Создание каждого объекта
  for (int i = 0; i < bots.length; i++) {
    // Генерация случайной x-координаты
    float x = random(-40, width-40);
    // Присвоение y-координаты
    float y = map(i, 0, bots.length, -100, height-200);
    bots[i] = new Robot(robotShape, x, y);
  }
  smooth();
}

void draw() {
  background(204);
  // Обновление и отображение 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;

  // Установка начальных значений конструктора
  Robot(PShape shape, float tempX, float tempY) {
    botShape = shape;
    xpos = tempX;
    ypos = tempY;
    angle = random(0, TWO_PI);
  }
```

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

  // Вывод робота на экран
  void display() {
    shape(botShape, xpos, ypos + yoffset);
  }

}
```

11/Расширения

Язык Processing создан для программирования интерактивной графики, и поэтому эта книга сосредоточена именно на этом. Но программирование имеет множество других возможностей, и, как правило, программный проект выходит за рамки графики. Processing может использоваться для управления механизмами, экспорта изображений для фильмов в формате HD и моделей 3D-печати.

За последнее десятилетие Processing был использован для создания видеоклипов для Radiohead и R.E.M., иллюстраций к журналам Nature и New York Times, скульптур для выставок, управления видеостеной размером 120x12 футов, вязания свитеров и многого другого. Processing обладает такой гибкостью благодаря своей системе библиотек.

Библиотека Processing это набор кода, который расширяет возможности программ за пределы основных функций и классов. Библиотеки сыграли важную роль в популяризации проекта, так как они позволяли разработчикам быстро добавлять новые функции. Применение библиотек, являющихся небольшими, самостоятельными проектами, оказалось более удобным, чем встраивание тех же функций в среду разработки.

В дополнение к библиотекам, включенным в Processing (они называются основными библиотеками), существует более 100 специальных библиотек доступных на сайте Processing. Список библиотек находится по адресу http://processing. org/reference/libraries/.

Выберите Import Library в меню Sketch для доступа к библиотекам. После выбора библиотеки к текущему скетчу будет добавлена строка кода, показывающая, что эта библиотека будет использована. Например, если вы выберете библиотеку OpenGL Library, вверху скетча появится такая надпись:

```
import processing.opengl.*;
```

Перед импортированием дополнительных библиотек из меню Sketch они должны быть скачаны с соответствующего сайта и размещены на компьютере в папке *libraries*. Папка с вашими библиотеками находится в скетчбуке. Расположение скетчбука вы можете найти открыв раздел Preferences. Сохраните скачанную библиотеку в папку *libraries* вашего скетчбука. Если этой папки не существует, создайте ее.

Существует более 100 библиотек Processing и мы не сможем рассказать здесь обо всех. Но мы выбрали несколько, по нашему мнению интересных и полезных для понимания этой главы.

3D

Существует два способа отобразить объемный объект в Processing - и оба требуют добавления третьего параметра к функции *size*() для изменения способа отображения графики. По умолчанию Processing использует двумерный рендерер, который очень точный, но медленный. Это рендерер JAVA2D. Не такая качественная, но иногда более быстрая его версия - P2D, рендерер Processing 2D. Вы можете сменить рендерер на рендерер Processing 3D, также называемый P3D, или на OpenGL, и ваши программы будут рисовать в трех измерениях, с добавленной осью z. (см. рисунок 11-1).

Визуализация с Processing 3D делается так:

```
size(800, 600, P3D);
```

А с OpenGL вот так:

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

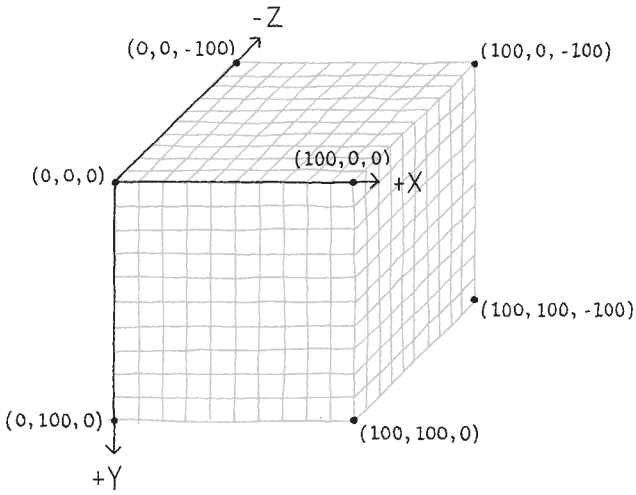


Рис. 11-1. Трехмерная система координат в Processing

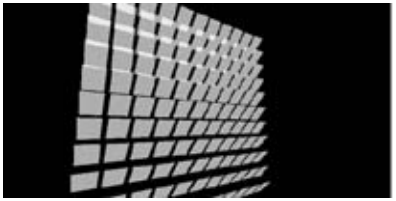
Рендерер P3D встроенный; рендерер OpenGL является библиотекой и требует команды импорта в коде, такой, как в начале примера 11-1. Рендерер OpenGL использует быстродействующее графическое аппаратное обеспечение, установленное на большинстве современных компьютеров.

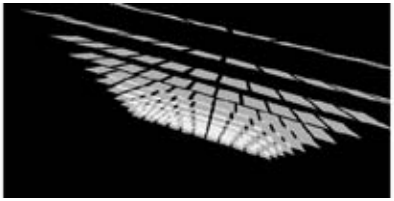
ПРИМЕЧАНИЕ: Рендерер OpenGL не всегда будет работать быстрее; за подробностями обращайтесь к справке по функции *size*().

Многие из функций, о которых рассказывалось в этой книге можно применять для работы в 3D. Например, к основным функциям рисования point(), line() и vertex() просто добавляется z-параметр после x и y-параметров, которые обсуждались ранее. Преобразования translate(), rotate() и scale() также хорошо работают в 3D.

Пример 11-1: Первая программа в 3D

В справке Processing вы найдете множество 3D функций; здесь мы приведем один пример для начала:



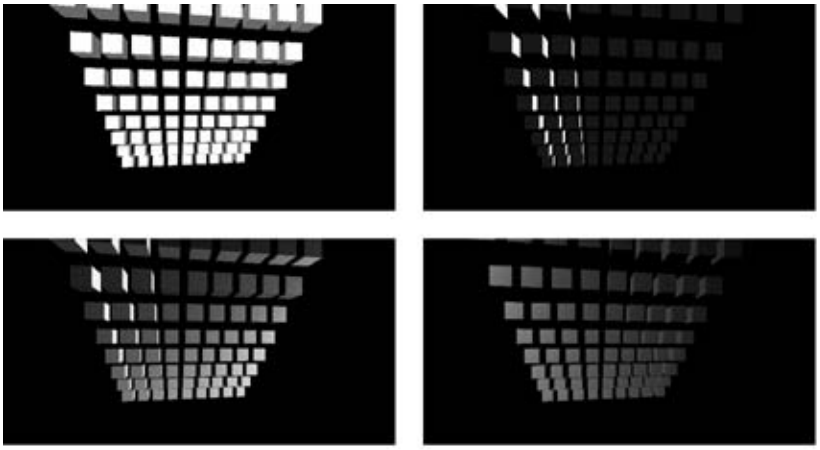


```
import processing.opengl.*;

void setup() {
  size(440, 220, OPENGL);
  noStroke();
  fill(255, 190);
}

void draw() {
  background(0);
  translate(width/2, height/2, 0);
  rotateX(mouseX / 200.0);
  rotateY(mouseY / 100.0);
  int dim = 18;
  for (int i = -height/2; i < height/2; i += dim*1.2) {
    for (int j = -height/2; j < height/2; j += dim*1.2) {
      beginShape();
      vertex(i, j, 0);
      vertex(i+dim, j, 0);
      vertex(i+dim, j+dim, -dim);
      vertex(i, j+dim, -dim);
      endShape();
    }
  }
}
```

Когда вы начнете работать в 3D, перед вами откроется море функций для исследования. Появится возможность менять ракурс, освещение, свойства материала, рисовать трехмерные сферы и кубы.

Пример 11-2: Освещение

Этот пример построен на основе примера 11-1 путем замены прямоугольников на кубы и добавления нескольких видов освещения. Испытайте различные виды освещения и их комбинацию, превращая ненужные строки кода в комментарий:



```
import processing.opengl.*;

void setup() {
  size(420, 220, OPENGL);
  noStroke();
  fill(255);
}

void draw() {
  lights();
  //ambientLight(102, 102, 102);
  //directionalLight(255, 255, 255,  // Цвет
  //                 -1, 0, 0);      // Направление XYZ
  //pointLight(255, 255, 255,        // Цвет
  //          mouseX, 110, 50);      // Местоположение
  //spotLight(255, 255, 255,         // Цвет
  //          mouseX, 0, 200,        // Местоположение
  //          0, 0, -1,              // Направление XYZ
  //          PI, 2);                // Плотность

  rotateY(PI/24);
  background(0);
```

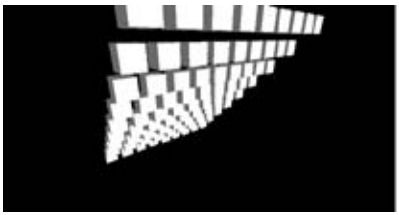
```
    translate(width/2, height/2, -20);
    int dim = 18;
    for (int i = -height/2; i < height/2; i += dim*1.4) {
      for (int j = -height/2; j < height/2; j += dim*1.4) {
        pushMatrix();
        translate(i, j, -j);
        box(dim, dim, dim);
        popMatrix();
      }
    }
  }
```

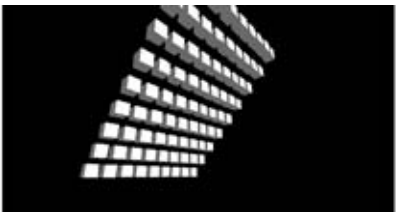
В Processing есть четыре типа освещения: spot (пятно), point (точка), directional (направленное) и ambient (рассеянное). Освещение типа spot распространяется в форме конуса, оно имеет направление, источник и цвет. Освещение point исходит из одной точки как из лампы любого цвета. Направленное освещение проецируется в одном направлении и создает резкие тени. Рассеянное освещение создает равномерную освещенность любого цвета всей композиции и как правило используется вместе с другими типами освещения. Функция *lights*() создает рассеянное и направленное освещение, установленное по умолчанию. Чтобы получить стабильный результат, освещение необходимо переустанавливать каждый раз в начале блока *draw*().

Работа в 3D не обходится без идеи ракурса (“camera”), направленного в созданном трехмерном пространстве. Как и обычная фотокамера, он отображает трехмерное простанство и двумерной форме. Изменение ракурса, по сути, меняет способ отображения 3D координат на вашем двумерном экране.

Пример 11-3: “Фотокамера” в Processing

По умолчанию мы рассматриваем фигуры в Processing как бы находясь в центре экрана, поэтому фигуры, удаленные от центра, мы видим в перспективе. Функция *camera*() предлагает управлять местоположением зрителя и ориентацией (вверх, вниз, под углом). В следующем примере для управления местоположением зрителя используется мышь:





```
import processing.opengl.*;

void setup() {
  size(420, 220, OPENGL);
  noStroke();
}

void draw() {
  lights();
  background(0);
  float camZ = (height/2.0) / tan(PI*60.0 / 360.0);
  camera(mouseX, mouseY, camZ,        // Местоположение зрителя
         width/2.0, height/2.0, 0,    // Направление обзора
         0, 1, 0);                    // Угол обзора
  translate(width/2, height/2, -20);
  int dim = 18;
  for (int i = -height/2; i < height/2; i += dim*1.4) {
    for (int j = -height/2; j < height/2; j += dim*1.4) {
      pushMatrix();
      translate(i, j, -j);
      box(dim, dim, dim);
      popMatrix();
    }
  }
}
```

Этот раздел предствил вам всего лишь верхушку айсберга возможностей 3D. В дополнение к упомянутым здесь функциям существует множество библиотек Processing, помогающих при создании трехмерных форм, загрузке и экспорте объемных фигур, а также их отображению на экране.

Экспорт изображений

Анимацию, созданную в программе Processing, можно перевести в последовательность файлов с помощью функции *saveFrame*(). Если в конце блока *draw*() стоит функция *saveFrame*(), то при выходе из программы она сохранит пронумерованные изображения в формате TIFF под названиями screen-0001.tif, screen-0002.tif и т.д. в папку скетча. Эти файлы могут быть импортированы в видео или в анимационную программу и сохранены как видеофайл. Вы можете сами указать имя файла изображения и его формат с помощью строки кода такого типа:

```
saveFrame("output-####.png");
```

ПРИМЕЧАНИЕ: При использовании функции *saveFrame*() в блоке *draw*() в каждом кадре создается новый файл, поэтому будьте внимательны - папка с вашим скетчем может очень быстро заполниться тысячами файлов.

Используйте символы # (октоторп или хэш) для указания места, где будет номер в имени файла. При сохранении они будут заменены на номер кадра. Вы также можете указать папку, в которую будут сохраняться ваши изображения, что очень удобно при работе с большим числом изображений кадров:

```
saveFrame("frames/output-####.png");
```

Пример 11-4: Сохранение изображений

В этом примере мы покажем, как сохранять изображения для создания двухсекундного видеоролика. Здесь программа работает в частотой 30 кадров в секунду и завершается после 60 кадров:

```
float x = 0;

void setup() {
  size(720, 480);
  smooth();
  noFill();
  strokeCap(SQUARE);
  frameRate(30);
}

void draw() {
  background(204);
  translate(x, 0);
  for (int y = 40; y < 280; y += 20) {
    line(-260, y, 0, y + 200);
    line(0, y + 200, 260, y);
  }
  if (frameCount < 60) {
    saveFrame("frames/SaveExample-####.tif");
  } else {
    exit();
  }
  x += 2.5;
}
```

Processing запишет изображения на основе указанного вами расширения файла (.png, .jpg и .tif являются встроенными, некоторые платформы поддерживают и другие). Изображение .tif сохраняется без компрессии; это быстро, но занимает много дискового пространства. .png и .jpg создадут меньшие по объему файлы, но из-за процедуры компрессии для этого потребуется больше времени, что замедлит работу вашего скетча.

Если выходными файлами является векторная графика, вы можете записать их в формате PDF для большего разрешения. Библиотека PDF Export позволяет записывать PDF-файлы непосредственно из скетча. Этот векторный графический файл может быть масштабирован до любого размера без потери разрешения, что делает его идеальным для печати - как постеров и баннеров так и целых книг.

Пример 11-5: Рисуем в PDF

Этот пример построен на основе примера 11-4 для изображения нескольких шевронов с различной толщиной линии, но без движения. Он возвращает файл PDF под названием Ex-11-5.pdf, как указано в третьем и четвертом параметре функции size():

```
import processing.pdf.*;

void setup() {
  size(600, 800, PDF, "Ex-11-5.pdf");
  noFill();
  strokeCap(SQUARE);
}

void draw() {
  background(255);
  for (int y = 100; y < height - 300; y+=20) {
    float r = random(0, 102);
    strokeWeight(r / 10);
    beginShape();
    vertex(100, y);
    vertex(width/2, y + 200);
    vertex(width-100, y);
    endShape();
  }
  exit();
}
```

Результат не будет выводиться на экран, он запишется напрямую в PDF файл, который сохранится в папку скетча. Код в этом примере запускается один раз и останавливается в конце блока *draw*(). Выходной файл показан на рисунке 11-2.

В среду разработки Processing включено множество примеров с экспортом в файл PDF. Загляните в раздел PDF Export примеров Processing и вы найдете большое количество приемов работы с PDF.

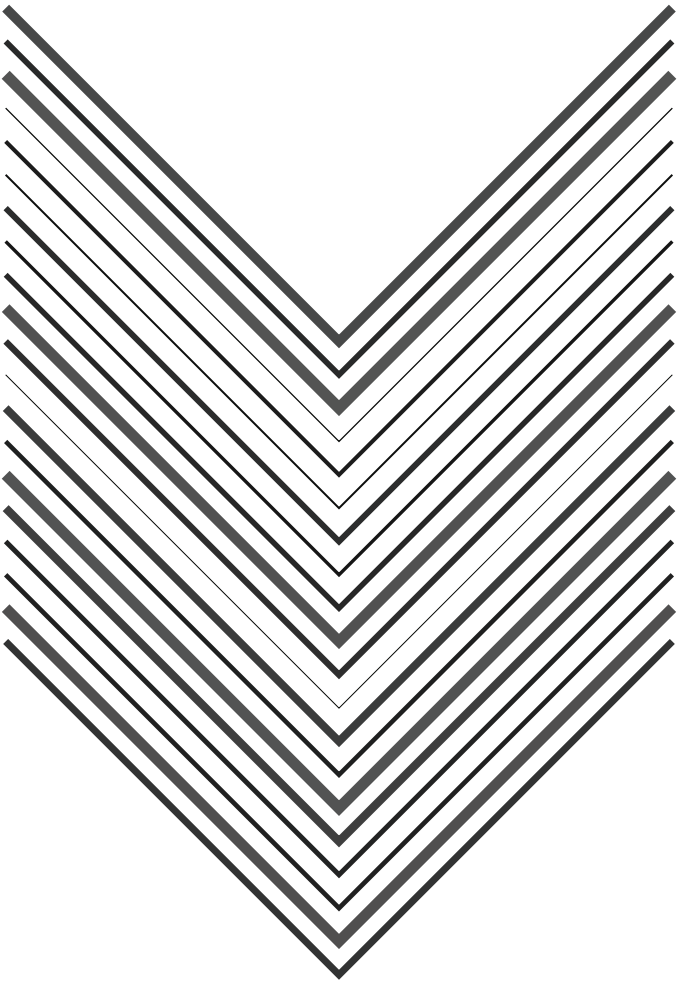


Рис. 11-2. Экспорт в PDF из примера 11-5.

Привет, Arduino!

Arduino - это платформа для электронного прототипирования, состоящая из серии плат с микроконтроллерами и программного обеспечения для их программирования. Processing и Arduino являются родственными проектами с похожими идеями и целями и, хотя они работают в разных областях, имеют долгую историю сотрудничества. Программное обеспечние и синтакс языков обоих проектов практически не различаются и вам будет легко переключаться с одного проекта на другой и использовать одни и те же приемы работы.

В этом разделе мы займемся чтением данных с платы Arduino, передачей их в Processing и визуализацией этих данных на экране. Это позволит программам Processing обрабатывать данные из новых источников, а программистам Arduino получать данные с сенсоров в виде графиков. Источником данных может быть все, что вы сможете подключить к плате Arduino: от дальномеров и компасов до сети датчиков температуры.

Мы предполагаем в этом разделе, что у вас имеется плата Arduino и вы владеете базовыми знаниями для работы с ней. Если это не так, обратитесь к сайту http://www.arduino.cc или к прекрасной книге "Getting Started with Arduino" от Массимо Банци (O'Reilly). (Русский перевод: "Arduino для начинающих волшебников". Изд. "Рид Групп") Как только вы овладеете основами, вы сможете узнать больше об обмене данных между Processing и Arduino из другой замечательной книги, "Making Things Talk" от Тома Айгоу (Tom Igoe) (O'Reilly).

Обмен данными между Processing и Arduino происходит с помощью Processing Serial Library. *Serial* (последовательный) - это формат передачи данных, при котором информация передается байт за байтом. В мире Arduino байт - это тип данных, представляющий собой число от 0 до 255; он аналогичен формату *int*, но находится в меньшем диапазоне. Большие числа разбиваются на несколько байтов и передаются по отдельности, а затем собираются на приемной стороне.

Мы возьмем самый простой код для Arduino и оставим его без изменения во всех последующих примерах, а сами займемся программированием на Processing. Мы визуализируем данные, поступающие из платы Arduino, а именно - последовательность байтов. Надеемся, что эти приемы, изложенные в этой книге, а также сотни других, размещенных в интернете, послужат вам хорошим стартом.

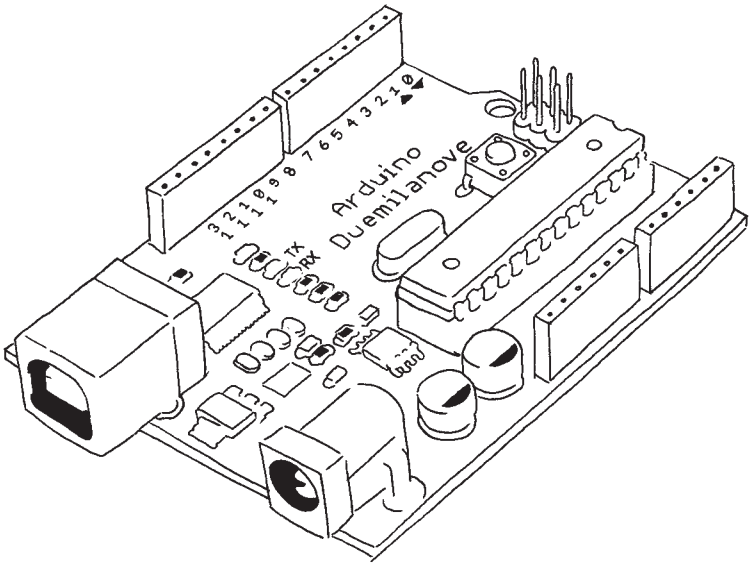


Рис. 11-3. П*лата Arduino Duemilanove*

Пример 11-6: Чтение данных с сенсора

В последующих трех примерах мы используем следующий код Arduino:

```
// Примечание: это код для Arduino, а не для Processing

int sensorPin = 0;     // Выбор входного вывода
int val = 0;

void setup() {
  Serial.begin(9600);  // Запуск последовательного порта
}

void loop() {
  val = analogRead(sensorPin) / 4;  // Чтение данных с сенсора
  Serial.print(val, BYTE);          // Отправка переменной в порт
  delay(100);                       // Задержка 100 мс
}
```

Два важных замечания о коде для Arduino. Во-первых, он требует подключения сенсора к аналоговому выходу 0 на плате Arduino. Вы можете использовать датчик освещенности (называемый также фоторезистор) или любой другой аналоговый датчик вроде термистора (температурно-зависимый резистор), резистора изгиба или резистора давления (резистор, в котором сопротивление зависит от приложенной силы). Схема подключения и макетная плата с установленными на ней компонентами изображены на рисунке 11-4. Во-вторых, обратите внимание, что получаемая с помощью функции *analogRead*() величина делится на 4 перед присвоением переменной *val*. Значения, получаемые от *analogRead*() находятся в пределах от 0 до 1023 и для передачи в формате одного байта мы преобразуем их в диапазон от 0 до 255 делением на 4.

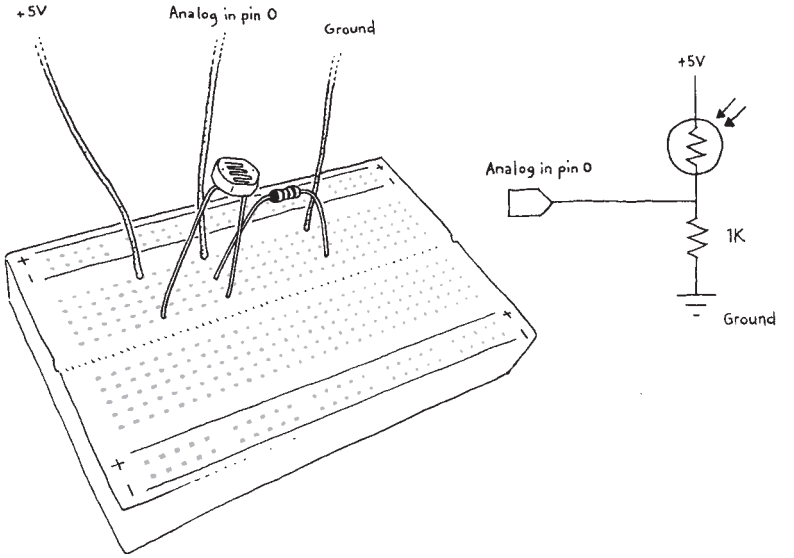


Рис. 11-4. Подключение датчика освещенности к аналоговому входу 0

Пример 11-7: Чтение данных из порта

Первый пример визуализации покажет вам, как считывать последовательные данные с платы Arduino и как преобразовывать их в величины, удобные для отображения на экране:

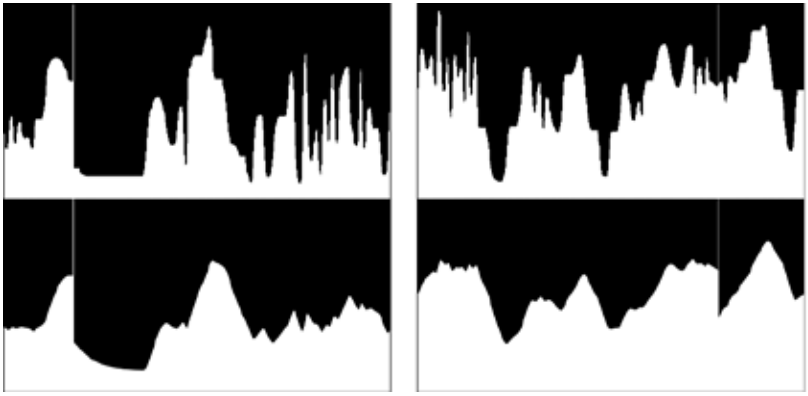
```
import processing.serial.*;

Serial port;  // Создание объекта класса Serial
float val;    // Данные, полученные из последовательного порта

void setup() {
  size(440, 220);
  // Важное замечание:
  // Первым последовательным портом, указанном в Serial.list()
  // должна быть ваша Arduino. Если это не так, удалите // в
  // в следующей строке. Запустите скетч заново и просмотрите
  // список последовательных портов. Затем запишите вместо 0
  // между символами [ и ] номер порта, к которому подключена
  // ваша плата Arduino.
  //println(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();                 // они считываются в val
    val = map(val, 0, 255, 0, height); // Преобразование величины
  }
  rect(40, val-10, 360, 20);
}
```

Библиотека Serial импортируется в первой строке, в блоке *setup*() запускается последовательный порт. Осуществить обмен данными между скетчем Processing и платой Arduino может быть сложно, а может быть легко - это зависит от качества собранной схемы. Зачастую скетч Processing обменивается данными с несколькими устройствами. Если ваш проект не заработал с первого раза, внимательно прочитайте комментарий в блоке *setup*() и выполните действия по инструкции.

В блоке *draw*() значения заносятся в программу методом *read*() объекта port из класса Serial. Программа считывает данные из последовательного порта только после получения нового байта. Метод *available*() проверяет готовность нового байта и возвращает количество доступных байт. Программа написана так, что при каждом прохождении блока *draw*() будет считываться один байт. Функция *map*() преобразует диапазон принятого значения из начального (от 0 до 255) в диапазон от 0 до высоты экрана; в данном случае это от 0 до 220.

Пример 11-8: Визуализация потока данных

Теперь наши данные передаются нормально и мы можем заняться их визуализацией. Значения, полученные напрямую с сенсора, часто не очень точны и бывает полезно сгладить их путем усреднения. Здесь, в верхней половине представлен несглаженный сигнал с датчика освещенности, изображенного на рисунке 11-4, а в нижней части - тот же сглаженный сигнал:



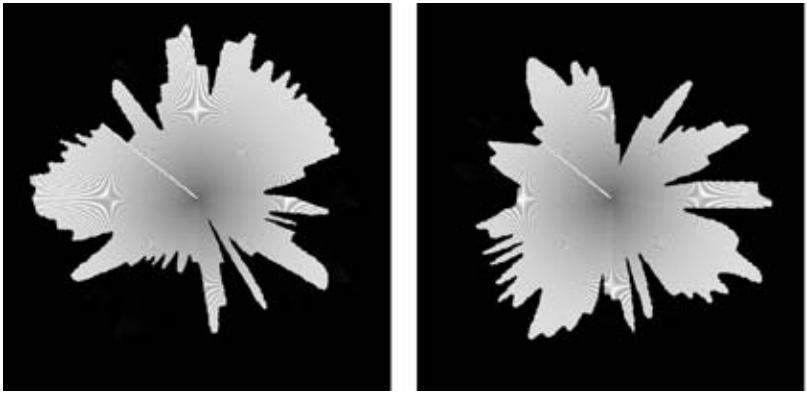
```
import processing.serial.*;

Serial port;  // Создание объекта класса Serial
float val;    // Данные, полученные из последовательного порта
int x;
float easing = 0.05;
float easedVal;

void setup() {
  size(440, 440);
  frameRate(30);
  smooth();
  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();                 // они считываются в val
    val = map(val, 0, 255, 0, height); // Преобразование величины
}
```

```
    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);             // Несглаженный сигнал
    line(x, 440, x, easedVal + 220);  // Усредненный сигнал

    x++;
    if (x > width) {
      x = 0;
    }
  }
```

Наряду с примерами 5-8 и 5-9 в этом скетче тоже используется техника *easing*. Каждый поступающий из Arduino байт является следующим значением; вычисляется разность между текущим и следующим значением и текущее значение корректируется в соответствии с следующим. Характер сглаживания входных данных можно регулировать, изменяя значение переменной *easing*.

Пример 11-9: Альтернативный способ визуализации.

На создание этого примера нас вдохновили экраны радаров. Данные так же считываются с платы Arduino, но теперь, с помощью фунций *sin*() и *cos*(), обсуждавшихся в примерах 7-12 и 7-15 они представлены круговой диаграммой:



```
import processing.serial.*;

Serial port;  // Создание объекта класса Serial
float val;    // Данные, полученные из последовательного порта
float angle;
float radius;

void setup() {
  size(440, 440);
  frameRate(30);
  strokeWeight(2);
  smooth();
  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();          // они считываются в val
    // Преобразование величин для установки радиуса
    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* непрерывно обновляется для вращения линии по кругу, а переменная v*al* изменяет длину линии, устанавливая нужное расстояние до центра экрана. После совершения полного оборота новые значения записываются поверх старых.

Мы восхищены потенциалом, которым обладает совместное использование Processing и Arduino в деле объединения мира программирования и элетроники. В отличие от примеров, приведенных здесь, обмен данными может быть двунаправленным. Определенные элементы на экране используются для управлением платой Arduino. Это значит, что вы можете использовать программы Processing как интерфейс между компьютером и моторами, громкоговорителями, фотокамерами, сенсорами, различной светотехникой и всем, что только может управляться посредством электрического сигнала. Напомним, что больше информации об Arduino вы найдете на сайте http://www.arduino.cc.

Сотрудничество

Мы хорошо поработали для того, чтобы вы могли с легкостью экспортировать программы Processing и делиться ими с другими. Во второй главе мы рассказали, как поделиться программой, экспортировав ее. Мы верим, что распространение хороших результатов мотивирует к обучению и сотрудничеству. Как только вы начнете писать свои скетчи, модифицируя скетчи из книги, мы рекомендуем вам поделиться своей работой с другими, указав на раздел книги, которым вы воспользовались. В настоящее время, лучшими местами для общения и обмена достижениями являются группы на OpenProcessing, Vimeo, Flickr и Processing Wiki. Поисковой запрос #Processing и Processing.org выдает интересные результаты. Это очень живые и интересные сообщества. Заходите на главный сайт Processing (http://www.processing.org), а также на следующие сайты за свежими ссылками:

» http://www.openprocessing.org

» http://www.vimeo.com/tag:processing.org

» http://www.flickr.com/groups/processing/

» http://www.delicious.com/tag/processing.org/

A/Полезные советы

Программирование - это разновидность писательства. Как и любой вид письма, оно имеет свои правила. Для сравнения мы приведем примеры из английского языка, о которых вы, возможно, не задумывались, потому что они стали вашей второй натурой. Одни из самых незаметных правил это написание текста слева направо и разделение слов пробелами. Другие очевидные правила это соблюдение орфографии, написание имен и названий мест с большой буквы и даже использование пунктуации в конце предложений для передачи акцента! Нарушение одного или нескольких из этих правил не помешает пониманию текста. Например, “прив бен. как дел” почти так же хорошо передает нашу мысль как “Привет, Бен. Как дела?” Однако гибкость разговорной речи не распространяется на программирование. Вы должны быть былее кратки и внимательны, так как ваша запись адресована компьютеру, а не другому человеку. От одного пропущенного символа может зависеть исход всей программы - заработает она или нет.

Processing старается подсказать вам, где вы допустили ошибку и в чем она заключается. Если в вашем коде есть опечатки (мы называем из багами), то после нажатия кнопки Run вы увидите красный цвет в области сообщений, а строку кода, в которой предположительно есть проблема, Processing подсветит. Строка кода, содержащая ошибку, как правило находится строкой ниже или выше подсвеченной, но иногда это она и есть. Текст в области сообщений выдается для помощи и предполагает, в чем может быть проблема, но иногда сообщение слишком сложно для понимания. Такие сообщения об ошибках не порадуют новичка. Вспомните, что Processing - это всего лишь маленькая программа, старающаяся быть полезной вам, но ее способность понять, что вы хотите, ограничена.

Длинные сообщения об ошибках более подробно выводятся на консоль и просмотр этого текста нередко дает хорошую подсказку. Кроме того, Processing умеет находить только одну ошибку за один раз. Если в вашей программе имеется много ошибок, вам потребуется запускать ее снова и снова и исправлять их по одной.

Внимательно прочитайте следующие советы - они научат вас писать понятный код и не допускать ошибок.

Функции и параметры

Программы состоят из множества небольших частей, собранных вместе в большую структуру. Похожая система и в английском языке: фразы состоят из слов, объединяются в предложения, а предложения группируются в параграфы. В коде та же идея, но все части имеют разные имена и выполняют разные функции. Функции и параметры являются важными частями любой программы. Функции - это основные строительные блоки программ Processing. Параметры - это значения, определяющие, как будет работать функция.

Рассмотрим функцию *background*(). Как вы, наверное, догадались, она устанавливает цвет фона окна. У функции есть три параметра, определяющих цвет. Эти числа задают количество красного, зеленого и синего компонентов нужного цвета. Например, этот код задает синий фон:

```
background(51, 102, 153);
```

Внимательно посмотрите на эту строку кода. Ключевая деталь здесь - это скобки после имени функции, в которых записаны числа, разделенные запятыми и точка с запятой в конце строки. Точка с запятой используется как точка в обычных предложениях. Она сигнализирует компьютеру, что выражение закончилось и можно приступать к следующему. Все это должно присутствовать в коде для правильной работы. Сравните предыдущий пример с неработающими версиями той же строки:

```
background 51, 102, 153;   // Ошибка! Нет скобок
background(51 102, 153);   // Ошибка! Пропущена запятая
background(51, 102, 153)   // Ошибка! Нет точки с запятой
```

Компьютер не прощает даже самые маленькие опечатки и пропуски и не распознает нужных команд. У вас будет меньше багов, если вы будете помнить об этом. Но если, как и мы, вы забудете что-нибудь, это не проблема. Processing уведомит вас о проблеме и программа прекрасно запустится когда вы поправите код.

Подсветка синтаксиса

Среда разработки Processing подсвечивает различные части кода. Слова, являющиеся ключевыми словами Processing, обозначаются синим или оранжевым чтобы отличить их от слов пользователя. Уникальные для вашей программы слова, например, имена переменных и функций отображаются черным. Базовые символы вроде (), [] и > также отображаются черными.

Комментарии

Комментарии - это заметки, которые вы оставляете в коде для себя (или для других людей). Они используются для пояснения функций кода и несут дополнительную информацию о названии и авторе программы. Комментарий начинается с двойной косой черты (//) и продолжается до конца строки:

```
// Это - строка комментария
```

Многострочный комментарий начинается с /* и заканчивается */. Например:

```
/*   Этот комментарий
     состоит из
     нескольких строк.
*/
```

Привильно введенный комментарий подсвечивается серым. Вся область комментариев выделяется серым и вы можете видеть весь комментарий от начала до конца.

Строчная и прописная

Processing различает строчные и прописные буквы, следовательно слово “Hello” отличается от слова “hello”. Не получится нарисовать прямоугольник с помощью функции *rect*(), если набрать R*ect*(). Вы увидите, распознал Processing ваш код или нет, проверив цвет текста.

Стиль

Для Processing не важно, сколько пробелов вы запишете, форматируя свой код. Вы можете написать:

```
rect(50, 20, 30, 40);
```

или:

```
rect (50,20,30,40);
```

или:

```
rect     (       50,20,
 30,   40)            ;
```

Однако, в ваших интересах сделать код легко читаемым. Это становится особенно важным когда объем кода возрастает. Хорошо отформатированный код становится разборчивым, а плохо отформатированный может запутать проблему. Выработайте привычку писать ясный код. Существует много способов хорошо отформатировать код, и выбор способа зависит от ваших персональных предпочтений.

Консоль

Консоль находится в нижней части среды разработки Processing. Выводить сообщения на консоль можно с помощью функции *println*(). Следующий код выводит на консоль сообщение с текущим временем:

```
println("Hello Processing.");
println("The time is " + hour() + ":" + minute());
```

Консоль удобно применять для наблюдения за тем, что происходит в процессе выполнения программ. Она используется для вывода значений переменных и отслеживания их, чтобы вы могли проверить наступление событий и заметить проблему в программе.

Шаг за шагом

Мы рекомендуем писать код небольшими фрагментами и почаще его запускать, чтобы убедиться, что не происходит накопления багов, о которых вы не знаете. Даже самые амбициозные программы писались по одной строчке. Разбейте ваш проект на простые подпроекты и разработайте каждый по отдельности: это превратит процесс создания в череду маленьких успехов вместо разбора кучи багов. Если обнаружился баг, попробуйте изолировать часть кода, где, по вашему мнению, находится проблема. Подойдите к исправлению бага как к решению увлекательной головоломки. Если вы раздосадованы и решение так и не приходит, сделайте паузу, чтобы освежить голову или попросите друга помочь вам. Иногда ответ находится у вас перед носом, но чтобы найти его, требуется свежий взгляд.

B/Типы данных

Существуют разные типы данных. Возьмем например данные из удостоверения личности. В нем хранятся данные о весе, росте, дате рождения, домашнем адресе и почтовом индексе. Оно содержит слова имени человека и названия города. Также там имеется фотография и решение о донорстве органов, то есть ответ да/нет. Processing использует разные типы данных для хранения различных данных. Каждый тип данных подробно объясняется в разных главах книги; приведем здесь полную информацию в виде таблицы:

Тип	**Расшифровка**	**Диапазон значений**
int	целые числа	от –2,147,483,648 до 2,147,483,647
f*loat*	числа с плавающей точкой	от –3.40282347E+38 до 3.40282347E+38
boolean	логические веражения	истина или ложь
char	одиночный символ	A–Z, 0–9 и символы
String	последовательность символов	любая буква, слово, предложение
PImage	изображение PNG, JPG, GIF	не определен
PFont	шрифт VLW; для создания используйте Create Font	не определен
PShape	файл SVG	не определен

Ориентировочно число *float* имеет точность около четырех десятичных разрядов после точки. Для счета событий или пошагового выполнения обычно используются числа *int*, а для обработки результата применяются *float*.

Есть и другие типы данных кроме упомянутых здесь, но в обычных задачах в Processing эти используются наиболее часто. Фактически, как мы говорили в главе 9, количество типов данных бесконечно велико, так как каждый класс является самостоятельным типом данных.

C/Порядок операций

При выполнении в программе математических вычислений арифметические операции имеют свой порядок. Порядок операций гарантирует нам, что при каждом запуске код будет работать одинаково. В этом отношении они не отличаются от операций в арифметике или алгебре, но в программировании есть несколько специфических операторов.

В приведенной таблице операторы, расположенные выше, выполняются раньше тех, что расположены ниже. Так, операции в скобках будут выполняться первыми, а оператор присвоения - последним.

Имя	**Символ**	**Пример**
Скобки	`()`	`a * (b + c)`
Постфикс, унарный	`++ -- !`	`a++ --b !c`
Мультипликативные	`* / %`	`a * b`
Аддитивные	`+ -`	`a + b`
Отношение	`> < <= >=`	`if (a > b)`
Равенство	`== !=`	`if (a == b)`
Логическое И	`&&`	`if (mousePressed && (a > b))`
Логическое ИЛИ	`||`	`if (mousePressed || (a > b))`
Присвоение	`= += -= *= /= %=`	`a = 44`

D/Область видимости переменных

Правило видимости переменных гласит: к переменной, объявленной внутри блока (код, заключенный в скобки **{** и **}**), можно обратиться только из этого блока. Это означает, что переменная, объявленная в блоке *setup*() работает только в блоке *setup*(), а переменная, объявленная в блоке *draw*(), может использоваться только в блоке *draw*(). Исключение составляют переменные, объявленные вне блока *setup*() и *draw*(). Эти переменные доступны для вызова как из *draw*(), так и из *setup*() (или из любой функции, которую вы создадите). Представьте область вне *setup*() и *draw*() как внешний блок кода. Мы называем эти переменные *глобальными* переменными: они могут использоваться в любом месте программы. Переменную, работающую исключительно в пределах одного блока, мы называем *локальной* переменной. Вот вам пара примеров, хорошо разъясняющих эту концепцию. Первый:

```
int i = 12;    // Объявление глобальной переменной i равной 12

void setup() {
  size(480, 320);
  int i = 24;  // Объявление локальной переменной i равной 24
  println(i);  // Вывод на консоль числа 24
}

void draw() {
  println(i);  // Вывод на консоль числа 12
}
```

Второй:

```
void setup() {
  size(480, 320);
  int i = 24;  // Объявление локальной переменной i равной 24
}

void draw() {
  println(i);  // ОШИБКА! Переменная i является локальной в setup()
}
```

Предметный указатель

Символы

3D, 158–163
> (больше, чем), оператор, 44
>= (больше или равно), оператор, 44
// (двойной слеш) комментарии, 33
< (меньше), оператор, 44
<= (меньше или равно), оператор, 44
!= (не равно), оператор, 44
|| (оператор логическое ИЛИ), 72
&& (оператор логическое И), 68
% (оператор целочисленное деление), 149
== (равенство), оператор, 72
(хэш), символ, 164

A

ACM SIGGRAPe архивы, 4
alpha величина, 29
analogRead() функция, 170
applet папка, 12–13
arc() функция, 20
Arduino, платы
  визуализация потока данных, 172–173
  визуализация в круговых диаграммах, 173–174
  обзор, 168–169
  чтение данных с последовательного порта, 170–171
  чтение с сенсоров, 169–170

B

background() функция, 26, 54
beginShape() функция, 30
boolean, тип данных, 60–62, 183

C

camera() функция, 162–163
char тип данных, 70
CLOSE параметр, 31–32
constrain() функция, 99

D

dist() функция, 55–56, 66

E

easing
  endShape() функция, 30
  движение мыши, 56–57
  сглаживание линий, 57

F

fill() функция, 26
float, тип данных, 92
library, папка 158
for, цикл
  заполнение массива значениями, 148
  примеры, 42–47
  частота кадров, 91–92

G

Getting Started with Arduino (O'Reilly), 168
GIF формат изображений, 12, 81–82
GIMP, программа, 81

H

HTML файлы в папке applet, 12

I

if, оператор, 62–63

J

JAR, файл в папке applet, 12
JAVA2D, рендерер, 158
JPEG, формат изображений, 81–82

K

keyCode, переменная, 73
keyPressed, переменная, 68–70

L

lights() функция, 162
Linux, установка Processing, 7–8
loadFont() функция, 84
loadImage() функция, 78

M

Mac OS X, установка Processing, 7
map() функция, 59–60

N

new, ключевое слово, 134, 145
nf() функция, 152
noSmooth() функция, 23

O

OpenGL, 5

P

P5, робот Processing. См. программы робота (примеры)
PDE, файл в папке applet, 12
PDF Export, библиотека, 165
PDF-файлы, рисование, 166–167
PEMDAS акроним, 42
PFont, тип данных, 84
PImage, тип данных, 78
PNG формат изображений, 81–82
point (точка), освещение 162
popMatrix()/pushMatrix() функции, 107
PostScript, 5
Present, команда, 11
println() функция, 181
Processing
  библиотеки, 157–158
  версии, установка, 7–8
  встроенные переменные, 40
  гибкость, 3
  загрузка/запуск, 7–8
  история, vii–viii, 4
  определение, 1
  похожие языки и среды, 5
  примеры и справка, 13
  сайт для публикации работ, 6
  сообщество пользователей, 175
  Среда разработки Processing (PDE), 8
pushMatrix()/popMatrix() функции, 107

R

random() функция, 97, 113–114, 117
randomSeed() функция, 99
rectMode() функция, 25
RGB цвет, 28
rotate() функция, 108
Run, кнопка, 10–11

S

saveFrame() функция, 164
scale() функция, 110
Serial Library (Processing), 168, 171
setup() функция, 52–53
sin() и cos(), фукнции, 101–104
size() функция, 15–16, 40
spot (пятно), освещение, 162
strokeCap() функция, 25
stroke() функция, 26
strokeJoin() функция, 25
SVG-файлы, загрузка и рисование 87
SVG, формат векторной графики, 81

T

TIFF формат, 164
Color Selector,28–29
translate() функция, 106–107

V

vertex() функция, 30–32
VLW, формат, 83

А

анимация
вычисление промежуточных позиций, 95–96
вращение системы координат, 108–109
движение по окружности с sin и cos, 103
движущиеся фигуры, 92–93
масштабирование фигур, 110–113
произвольное движение/рисование фигур, 97–99
синусоидальное движение/значения,102–103
спирали, создание, 104
фигуры, отскакивающие от границ экрана, 94–95
частота кадров, 91–92
антиэлайзинг (сглаживание), 23
арифметические операции
операторы, 41–42
порядок, 185

Б

библиотеки, Processing, 157–158
блок else, 62–63
блоки, цикл for, 43

В

ввод с клавиатуры, 70–71
векторная графика, 165
векторные графические файлы
загрузка в Processing, 86
масштабирование, 88
формат SVG, 81
вложенный цикл, 46
возвращаемые значения, 124–125
вращение системы координат, 108–109
выражения арифметические, 41-42
вычисление промежуточных изображений (анимация), 95–96

W

width/height, переменные, 40
Windows (Microsoft), установка Processing, 7

Г

глобальные переменные, 52

Д

движение по окружности пример, 113–114
движение фигур на экране, 93–94
двойной слеш (//) для комментариев, 33
деление по модулю (%), 149
длительность работы, 99

З

запуск по времени, 100
загрузка Processing, 7–8

И

измерение в градусах, 20–22
изображения
image() функция, 79
вывод на экран, 78–80
изменение размера, 80–81
сохранение, 164–165
экспорт, 164–167

К

клавиши
определение нажатия, 72–73
набор, 70–71

классы объектов, 130–134
кнопки
  мыши, 60–63
  отслеживание, 63–64
код
  использование переменных, 49–50
  повторения в цикле for, 42–45
  полезные советы 177–181
  разбиение на вкладки, 137
комментарии
  добавление с помощью //, 33–34
  многострочный, 179
  советы, 179
консоль
  конструктор (ООП), 132
  курсор, определение положения 65–68
  определение, 8
  функции, 181

Л

линиии
  и точки, создание, 48
  рисование, 18
  рисование непрерывных, 55
  рисование сглаженных, 23–24
  сглаживание с помощью easing, 57
  установка толщины, 55–56
логические операторы
  И (&&), 68
  ИЛИ (| |), 72
локальные переменные, 187

М

массивы
  заполнение массива в цикле for, 148
  Маэда, Джон, vii
  объектов, 150–152
  определение, 141
  отслеживание движений мыши, 148–149
  повторения, 147
  последовательность изображений, 152– 153
  создание (примеры), 144–147
  стрелки(клавиши), обнаружение, 73
масштабирование фигур, 88, 110–113
медиа-файлы, загрузка, 7
методы (ООП), 129–134
мышь
  mousePressed, переменная, 60
  изменение размера изображений, 80–81
  кнопки, отслеживание нескольких, 63–64
  нажатие, 60–63
  отслеживание движений с помощью массива, 148–149
  отслеживание движений, 53–54
  сглаживание траектории, 56–57
  скорость мыши, вычисление, 55–56

Н

начальное условие, 44
нелатинские символы, 84
непрерывные линии, рисование 55

О

область видимости переменных, 187
область вывода сообщений, 8
объекты
  и классы, 130–134
  массив, 150–152
  основные сведения, 129–130
  создание, 134–135
  создание нескольких, 136–137
объявление переменных объекта, 134
окно отображения результата, 15
окружность
  зависящая от позиции мыши, 66–67
  рисование, 10, 19
  с помощью синуса и косинуса, 103
ООП (объектно-ориентированное программирование),129
оператор точка (.), 136
операторы
  арифметические, 41
  отношения, 44
  сравнения (==), 72
освещение
  в 3D (пример), 161–162
  рассеянное, 162
  направленное, 162
отслеживание
  движения мыши, 53–54
  состояния кнопок мыши, 63–64
оттенки серого, 27–28

П

папка data, 77–78
панель инструментов, 8
параметры
  основные сведения, 122–123
  полезные советы, 178
перемещение, 106–109
пиксель, основные сведения, 15
порядок вычислений 185
произвольные фигуры, 30–33
последовательность рисования, 22–23
полезные советы по написанию кода, 180
примеры и справка (Processing), 13–14
пример класса (объект), 130
присвоение имен переменным, 39
приоритет операторов (вычисления), 185
преобразование диапазонов, 58-59
прямоугольники
  зависящие от положения курсора, 67– 68
  и массивы, 141–143
  изменение значений, 38
  рисование, 19
прозрачность
  в изображениях GIF/PNG, 81–83
  установка, 29
переменные
  frameRate, переменная. См. частота кадров
  вычисление скорости мыши, 55–56 создание, 39
  глобальные, 52
  модификация кода с помощью, 49
  область видимости, 187
  объекта, 129
  объявление переменных объекта, 134
  повторное использование величин, 37
  поля 129–134

Р

радианы
  определение, 20–21
  рисование с помощью, 22
разворот за границы экрана, 93–94
растровые изображения в форматах JPEG/PNG/GIF, 81
рендерер OpenGL , 159
рисование
  draw() функция, 51–52
  в 3D, 158–163
  в файл PDF, 166–167
  в цвете, 26–30
  изображений на экране, 78–80 квадратов и кругов, 19 кругов, 10
  линий, 18
  непрерывных линий, 55
  окна, 16
  основных фигур, 16–18
  последовательность, 22–23
  прямоугольников, 19
  с помощью радиан, 22
  с помощью шрифта, 84–85
  сглаженных линий, 23–24
  точек, 16
  фигур случайное, 97–99
  эллипсов, 9, 19–20
робот, программы (примеры)
  drawRobot() функция, 126–128
  движение по кругу и случайное, 113–114 загрузка изображений SVG/PNG 89–90 класс и объекты Robot, 138–139
  массив объектов Робот, 153–155 модификация кода и переменные 49–50
  рисование робота P5, 34–35
  управление фигурами мышью, 74–76

С

сайт
  библиотек Processing, 157
  для загрузки медиа-файлов, 77
  для размещения программ, 175
  с дополнительной информацией по плате Arduino, 168
свойства вкладок (Среда разработки Processing), 137
свойства фигур, 23–25
сглаживание линий с easing, 57
символы клавиатуры, размер 71
синусоида, 102–103
синтаксис, 9
скетчинг
  Sketch, меню, 11
  обзор, 2
  скетчбук, 11
  создание/сохранение скетчей, 11–12
экспорт скетчей, 12–13

события, срабатывающие по времени, 100
совиная функция, создание, 118–124
сокращенная запись выражений, 42
сообщество Processing, 175
сохранение
  Save, команда, 11–12
  saveFrame(), функция, 164
  изображений, 164–165
спирали, создание, 104
справка и примеры (Processing), 14
стрелки, рисование, 30
строка, сохранение текста, 86
строчные и прописные буквы в коде, 180
строки и столбцы, создание в цикле for, 47
считываение с сенсоров (Arduino), 169–170

Т

текст
  textAlign() функция, 71
  textFont() команда, 84
  text() функция, 71, 84
  textSiz e() функция, 71, 84
  сохранение в строках, 86
  текстовый редактор, 8
типы данных
  boolean, 60–62, 183
  char, 183
  float, 183
  int, 183
  PFont, 183
  PImage, 183
  PShape, 183
  String, 183
  общая информация, 183– 184
  параметры, 122
  переменных, 39
толщина штриха
  постоянная, 111–112
  установка, 24
точка, рисование, 1
точки и линии, создание, 48

У

управляющие клавиши, 73
условное выражение, 44

Ф

фигуры
  shape() функция, 87
  движение, 92–93
  основы рисования, 16–18
  произвольные, 30–33
  рисование с помощью фигур, 87
  свойства, 23–25
  случайное рисование, 97–99
  управление мышью, 74–75
  масштабирование, 88
функции
  вычисление и получение значений, 124–125
  определение, 15
  основы, 116–117
  полезные советы, 178
  создание, 118–124

Ц

цвет
  Color Selector, 28–29
  кодирование, 179
  рисование, 26–30

Ш

шрифт
  рисование, 84–85
  создание, 83–84

Э

экспорт
  изображений, 164–167
  скетчей, 12–13
элементы (массива), 144
эллипс
  ellipseMode() функция, 25
  рисование, 9, 19–20

Об авторах

Кейси Риз - профессор факультета Design Media Arts Калифорнийского университета в Лос-Анджелесе и выпускник лаборатории MIT Media Laboratory. Программное обеспечение, разработанное Ризом, было представлено на множестве выставок в США, странах Европы и Азии. Совместно с Беном Фраем он основал Processing в 2001 году. Он соавтор книги Processing: A Programming eand-book for Visual Designers and Artists (MIT Press, 2007) и Form+Code in Design, Art, and Architecture (Princeton Architectural Press, 2010). Узнать больше о его работах можно по адресу www.reas.com.

Бен Фрай - получил свою докторскую степень в MIT Media Laboratory. Вместе с Кейси Ризом он разработал Processing, который принёс им Золотую Нику на фестивале Prix Ars Electronica в 2005. Работа Фрая принесла ему стипендию New Media фонда Рокфеллера и была показана в Нью-Йоркском музее современного искусства, в Ars Electronica, на биеннале Уитни в 2002 году и фестивале Cooper eewitt Design в 2003 году.