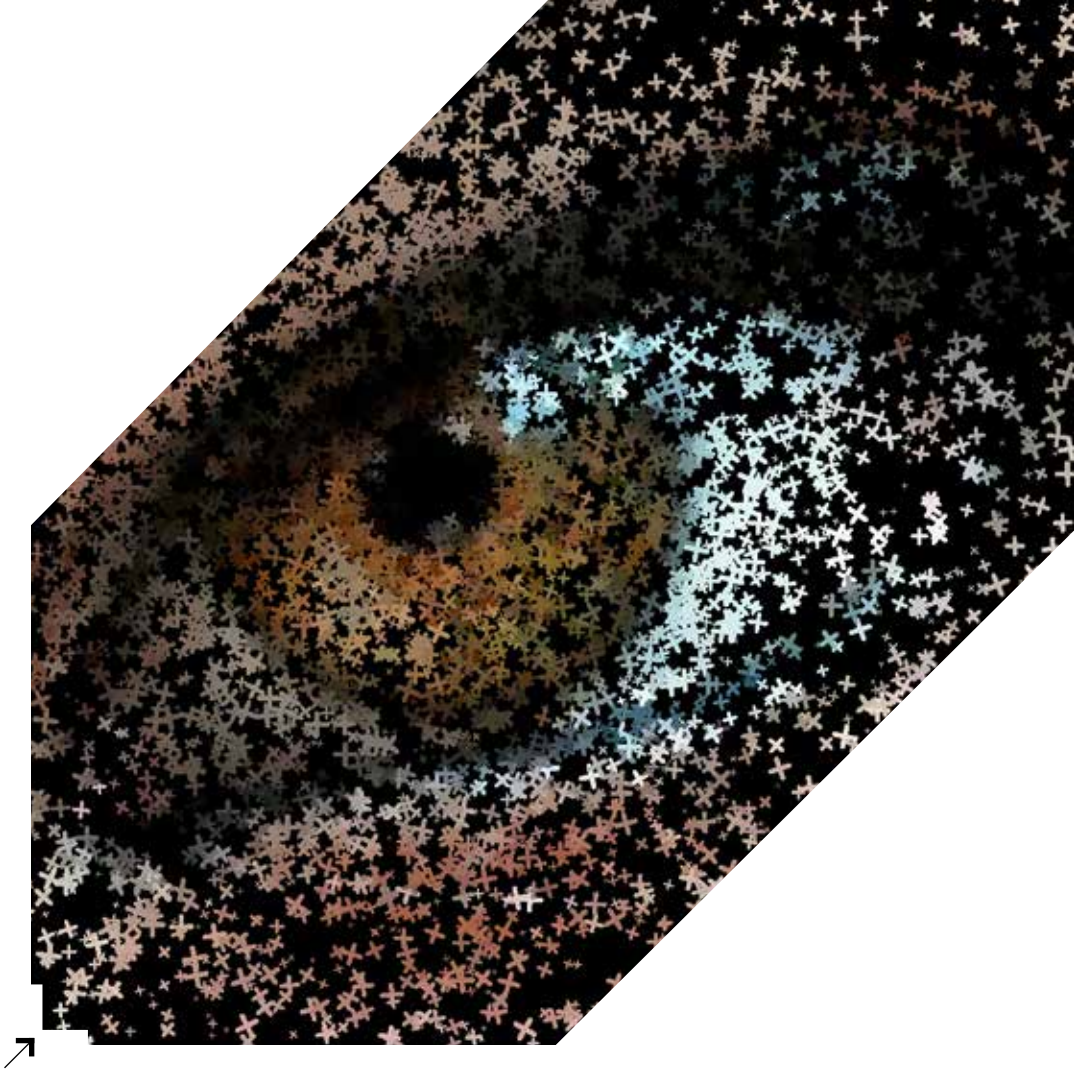


П.А. Орлов

ПРОГРАММИРОВАНИЕ ДЛЯ ДИЗАЙНЕРОВ

Учебное пособие

Co-funded by the
Tempus Programme
of the European Union

П.А. Орлов

ПРОГРАММИРОВАНИЕ ДЛЯ ДИЗАЙНЕРОВ

Учебное пособие для студентов высших учебных заведений,
обучающихся по программам магистерской подготовки
по направлению «Дизайн»

Под редакцией профессора В.М. Иванова

2015

УДК 004.92
ББК 32.973-018.2
О66

Рецензенты:
доктор технических наук, профессор Ю.Л. Липовка (Сибирский федеральный университет)
доктор технических наук, профессор С.В. Мещеряков (Санкт-Петербургский политехнический университет Петра Великого)
доктор филологических наук, профессор В.И. Заботкина (Российский государственный гуманитарный университет)

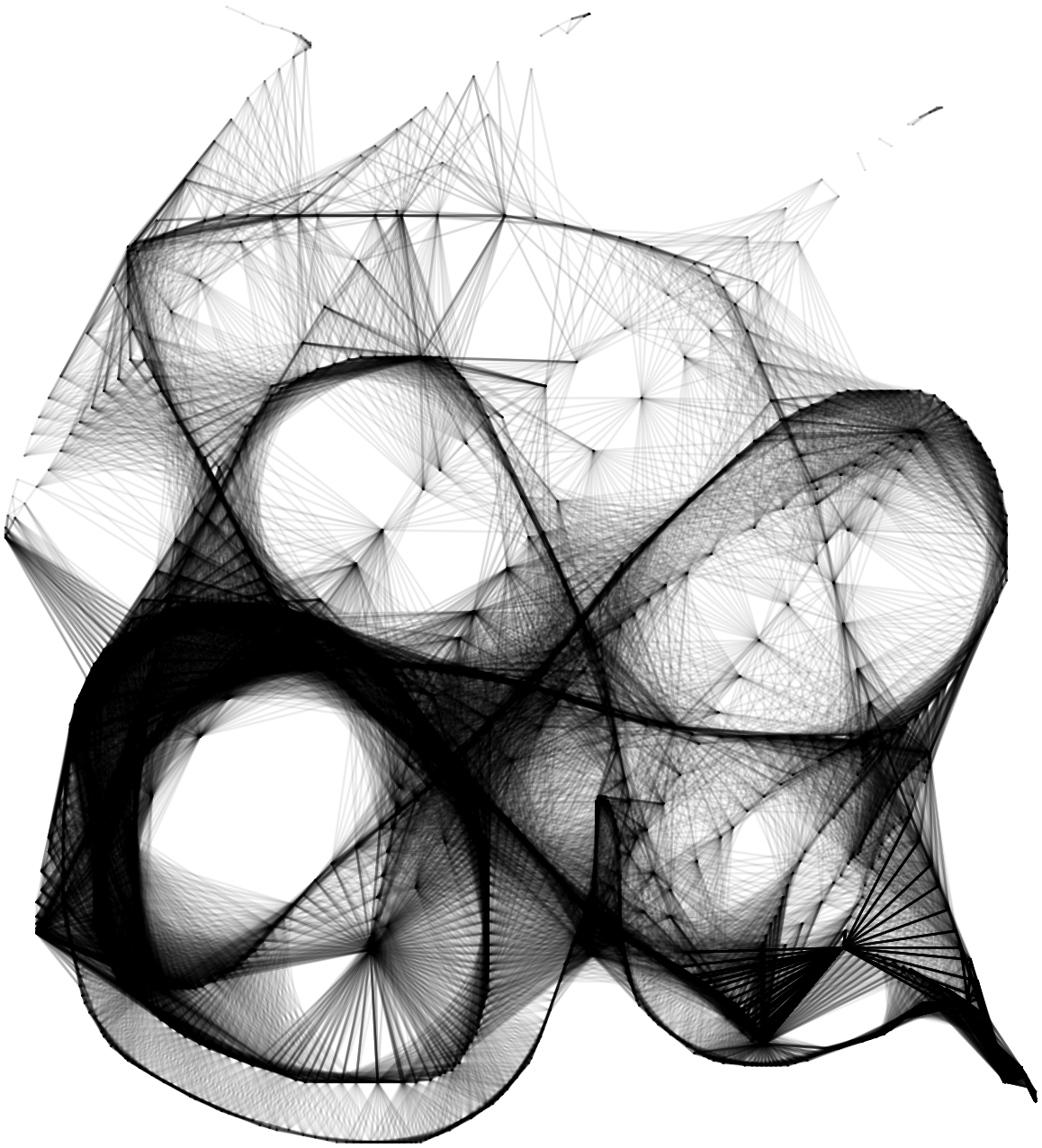
Орлов Павел Анатольевич
О66 Программирование для дизайнеров : учеб. пособие /
П.А. Орлов ; под ред. проф. В.М. Иванова — М. : АВАТАР, 2015. — 247 с.
ISBN 978-5-903781-16-4

Учебное пособие предназначено для подготовки магистров в области компьютерного дизайна, а также для профессиональных дизайнеров-инфографов, арт-директоров, редакторов издательств и СМИ, студентов и преподавателей профильных учебных заведений.
В издании описывается программирование на языке Processing для дизайнеров и цифровых художников. Уделено внимание установке необходимых программных средств и библиотек для работы с 2D графикой, анимацией, интерактивностью, векторной и растровой графикой, видеопотоками и системами частиц. Описаны возможности применения Processing для визуализации данных и разработки художественных инсталляций и приложений, последовательно рассматриваются основы программирования на графических примерах. Книга рассчитана на широкий круг читателей, интересующихся генеративной графикой и разработкой произведений цифрового искусства.
Учебное пособие обобщает опыт, полученный в рамках Европейского проекта по программе Tempus. Содержание данной публикации является предметом ответственности авторов и не отражает точку зрения Европейского Союза.

УДК 766:003.63
ББК 85.88

ISBN 978-5-903781-16-4

Co-funded by the
Tempus Programme
of the European Union



Programming for Artists

Pavel A. Orlov

Содержание

1 Введение

Во все времена художники пытались усовершенствовать существующие инструменты для творчества или создать новые с целью достижения собственных уникальных результатов. Так появлялись необычные кисти,распылители, новые красители. Компьютерные технологии открыли новую область творчества - цифровое искусство: современные художники получили возможность создавать собственные инструменты с помощью языков программирования.

В этой книге вы познакомитесь с новым инструментом современного художника – *креативным программированием*. Часто программирование воспринимается как сложный процесс, освоение которого доступно не каждому человеку. Изучение программирования действительно требует усилий, как, впрочем, и освоение любого другого инструмента. Однако в определенном смысле освоить его проще, чем, например, академический рисунок.

Вначале мы поговорим о самом понятии креативное программирование и укажемособенности языков для креативного программирования.

Митчел и Боун определяют креативное программирование как процесс, базирующийся на феномене открытия, включающий в себя исследование, повторение и рефлексию, использование кода как основного «посредника» для получения артефактов искусства (Mitchell and Bown, 2006).

Креативное программирование обладает набором свойств, первое из которых напрямую связано с особенностями творческого процесса: часто художник начинает творить, имея лишь замысел, а не детально разработанное задание, и результат может оказаться неожиданным для самого творца. Таким образом, первое свойство креативного программирования заключается в отсутствии формализованного технического задания. Второе свойство связано с возможностью создания индивидуальных художественных средств с помощью языков программирования. Например, Харе и Дейдумронг на основе одного из языков программирования представили техническое описание каркаса для инсталляций «интерактивного» искусства (Hare and Dejdumrong, 2009). Еще один пример - работа Нассера, где он описывает язык программирования, как новый инструмент (Nasser, 2011).

Свойства креативного программирования вносят свои особенности как в процесс программирования, так и в инструменты программирования. Сформулируем особенности, характерные для языков креативного программирования: во-первых, это ясный синтаксис программного языка; во-вторых, возможность работать в режиме live coding (реального

времени), который позволяет видеть результат программы сразу после ввода строки в редактор; в-третьих, простота установки; в-четвертых, полная «комплектация» языка (включая управление сторонними библиотеками и совместимостью версий) для решения художественных задач. Нассер учитывает перечисленные выше особенности для создания Zajal – языка программирования для художественных задач (Nasser, 2011).

В этой книге креативное программирование описано на основе языка Processing, для которого характерны все сформулированные особенности(Reas and Fry, 2007; Colubri and Fry, 2012). Возможность режима live coding в Processing достигается с помощью дополнительных библиотек (Bergstrom and Lotto, 2015). Для начала работы с языком Processing не требуются специальных знаний, умений и навыков, таким образом, процесс воплощения художественных замыслов идет быстро и просто (Pyshkin, 2011). Благодаря этому, Processing приобрел большую популярность в образовательной среде (Greenberg et al., 2012). Возможность работы в режиме live coding позволяет сделать из акта программирования перформанс. В визуальном искусстве это реализуется, например, в виджеинге, когда художественный результат производится артистом «вживую» в сочетании с музыкой, записанной заранее или звучащей в живом исполнении.

В этой книге продемонстрированы возможности креативного программирования в области создания генеративных художественных произведений 2D графики и интерактивных приложений. Вы научитесь базовым конструкциям программирования и объектно-ориентированного подхода на примере языка Processing. Освоение материала, представленного в книге, не потребует специальных навыков программирования, однако необходимы базовые знания математики и геометрии. И, конечно, не обойтись без амбиций художника!

Список литературы

Bergstrom, I. and Lotto, R. B. (2015). Code Bending: A new creative coding practice. *Leonardo*, pages 1–12.

Colubri, A. and Fry, B. (2012). Introducing Processing 2.0. In *ACM SIGGRAPH 2012 Talks*, page 12. ACM.

Greenberg, I., Kumar, D., and Xu, D. (2012). Creative coding and visual portfolios for CS1. *Proceedings of the 43rd ACM technical symposium on Computer Science Education - SIGCSE '12*, page 247.

Hare, T. N. and Dejdumrong, N. (2009). A framework on the applications of interactive art. *Proceedings of the 2009 6th International Conference on Computer Graphics, Imaging and Visualization: New Advances and Trends, CGIV2009*, pages 83–88.

Mitchell, M. C. and Bown, O. (2006). Towards a Creativity Support Tool in Processing : Understanding the Needs of Creative Coders Bolstering creative engagement via computational building. pages 143–146.

Nasser, R. (2011). The zajal programming language. *Proceedings of the 8th ACM conference on Creativity and cognition*, pages 413–414.

Pyshkin, E. (2011). Teaching programming: What we miss in academia. In *Software Engineering Conference in Russia (CEE-SECR), 2011 7th Central and Eastern European*, pages 1–6.

Reas, C. and Fry, B. (2007). *Processing: A Programming Handbook for Visual Designers and Artists*. Working paper series (National Bureau of Economic Research). NBER.

2 Установка необходимых компонентов

Для установки необходимых компонентов нам понадобится или стационарный компьютер, или ноутбук. Processing работает на Mac, Linux и на Windows системах. Скачать Processing абсолютно бесплатно можно на его официальном сайте: https://www.processing.org/

Прежде чем описывать работу с Processing IDE, необходимо отметить, что в сети Интернет существует большое сообщество людей, неравнодушных к Processing. Вы всегда можете найти ответы на большую часть ваших вопросов, касающихся использования Processing, и примеры кода на следующих вебсайтах:

1. http://funprogramming.org/
2. http://www.openprocessing.org/
3. http://www.learningprocessing.com/
4. http://stackoverflow.com/
5. https://vk.com/docs?oid=-13537660
6. http://btk.tillnagel.com/

Установив Processing IDE штатными средствами вашей платформы, запускаем его. Processing – это язык программирования, а Processing IDE – это среда разработки, т.е. то место где вы будете набирать исходный код и проверять работоспособность своих программ.

На Рисунке 1 показано стартовое окно программы Processing IDE. В Processing IDE можно посмотреть номер строки: кликните по строке и в левом нижнем углу будет показан её номер. Однако программы на языке Processing можно писать и в других IDE, например, в Интернет-редакторе Sketchpad (http://sketchpad.cc/). Визуально стили редактирования исходного текста отличаются, но не сильно:например,цветом ключевых слов и переменных, нумерацией строк, расположенной с левой стороны и т.п. Поэтому в книге мы не считаем нужным придерживаться одного стиля выбранного IDE, а будем использовать свой визуальный стиль с нумерацией строк для более удобного поиска их в листинге исходного кода.

Задание 1. Скачайте Processing IDE с официального сайта https://www.processing.org и установите его себе на компьютер.

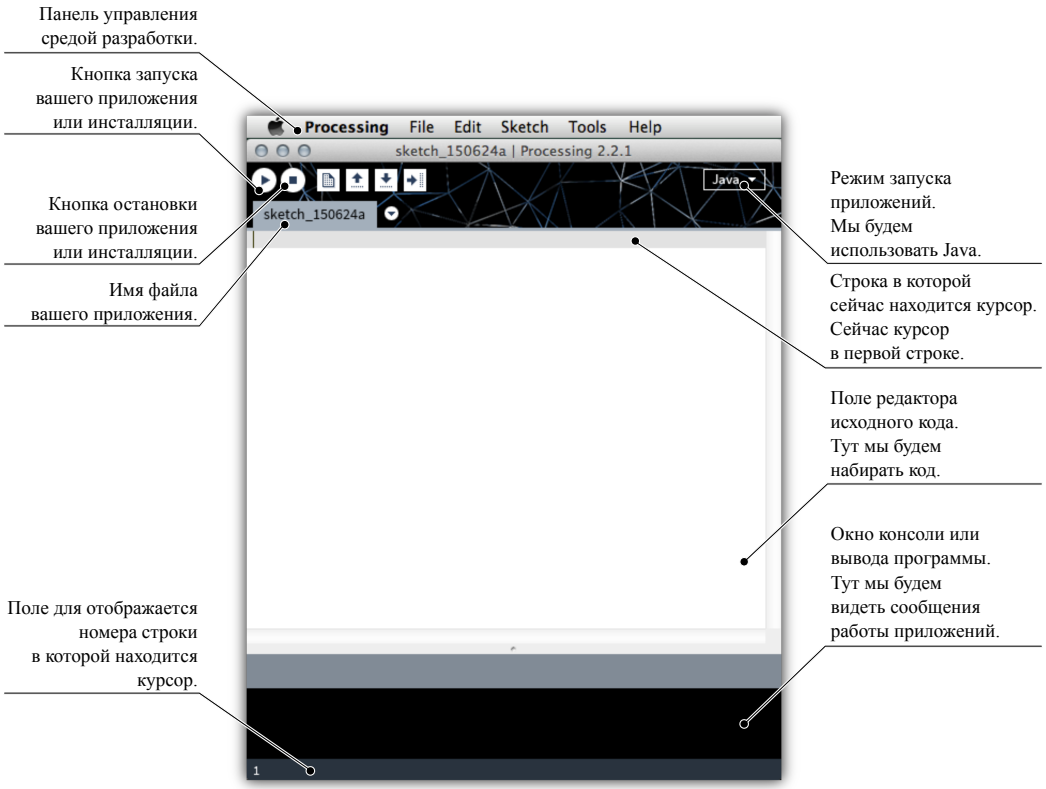


Рис. 1: Стартовое окно Processing IDE.

В меню *File -> Examples...* есть множество примеров приложений на языке Processing. Предлагаю вам открыть любой из них, например *Basics -> Color -> Brightness*. Проект откроется и в новом окне редактора появится исходный код примера. Запустите приложение, нажав на кнопку *Run*. Пока мы не будем останавливаться на объяснении этого примера: важно, чтобы вы увидели, что программы работают!

Как вы видите, программы из примеров работают, а значит все необходимые компоненты для них уже установлены. Давайте посмотрим, какие еще функции предоставляет Processing IDE.

Меню *File* содержит стандартные функции по открытию проекта, его сохранению и закрытию. Проект Processing представляет из себя папку, в которой находятся необходимые файлы. Основной файл проекта – это файл с расширением *.pde*, в котором находится исходный код. Второй файл в проекте – это *sketch.properties*, который создается автоматически (мы не будем править его самостоятельно). Файлов *.pde* в проекте может

быть несколько. Также в проекте часто размещаются дополнительные файлы, например, такие, как изображения, файлы векторной графики или файлы аудиоформатов.

Проект на языке Processing не является приложением, которое вы можете запустить без Processing IDE. Для того, чтобы получить именно работающее приложение (а не проект) его необходимо экспортировать. Для этого используется пункт меню *File -> Export Application...* Экспортировать таким образом приложения очень удобно и просто.

Задание 2. Откройте пример из меню *File -> Examples...* и далее *Basics -> Color -> Saturation*. Экспортируйте этот проект как приложение для вашей операционной системы. Запустите его, убедитесь, что это приложение работает отдельно от Processing IDE.

Раздел меню *Edit* традиционно служит для управления редактированием исходного текста. Раздел *Sketch* объединяет функции для работы с проектом: запуск, остановка и показ. В режиме *показ*, ваше приложение запускается на фоне полноэкранной серой заливки. Этот режим удобно использовать для демонстрации работы готового приложения. Выйти из режима *показ* можно, кликнув на *stop* в левой нижней части экрана (или нажав кнопку *Escape* на клавиатуре).

Задание 3. Откройте пример из меню *File -> Examples...* и далее *Basics -> Math -> Arctangent*. Запустите его в режиме *показ*.

Раздел меню *Sketch* содержит также функции по добавлению файлов в проект (например, изображений), открытию папки проекта и работе с библиотеками. Библиотеки для Processing играют важную роль, расширяя творческие возможности. Некоторые библиотеки уже установлены, другие вы можете установить. Например, вы можете установить библиотеку для компьютерного зрения или для работы с контроллером Kinect.

Раздел *Tools* содержит полезные утилиты для работы с Processing. Базовые утилиты, такие как *Create Font...* и *Color Selector*, установлены по умолчанию. Добавить новые утилиты можно с помощью функции *Tools -> Add Tool...* . Эти утилиты отличаются от библиотек раздела *Sketch* тем, что библиотеки используются непосредственно в вашем приложении, а утилиты решают вспомогательные задачи. Например, утилита выбора цвета служит удобным приложением для выбора цвета в

визуальном режиме и после того, как вы выбрали цвет, приложение можно закрывать. Библиотеки же, напротив, подключаются в исходный код вашего приложения.

Функция *Movie Maker* из раздела *Tools* служит для создания видеоролика из набора файлов изображений. Прямой записи видеоролика с экрана вашего приложения в Processing по умолчанию нет. Однако вы можете сохранять каждый кадр работы вашего приложения и потом формировать из них видеофайл. Это очень удобно, когда вы хотите показать работу вашего приложения, например, выложив видеоролик на YouTube.

Раздел *Help* содержит функции помощи и ссылки на обучающие сайты, документацию и другие интернет-источники.

Итак, в этом разделе вы познакомились с окружением Processing для креативного программирования. Настоятельно рекомендую вам обращаться к примерам исходных кодов, запускать их, копировать и модифицировать. Processing распространяется под открытой лицензией, так что любое улучшение кода и использование его в своих целях приветствуется.

3 Геометрическое формообразование

В этом разделе мы познакомимся с использованием Processing для базового рисования. Рисовать будем постепенно, начнем с линий и квадратов. Processing можно рассматривать как один из возможных инструментов для цифрового художника. Более привычные инструменты, такие как векторные редакторы (Inkscape, Adobe Illustrator) интересно использовать в сочетании с Processing, достигая желаемых художественных эффектов. Processing может работать с форматами векторной графики SVG, файлы которой вы можете создавать в векторных редакторах.

3.1 Линия

Начнем с рисования линии и на этом примере обсудим все появляющиеся вопросы. Итак, создаем новый *скетч*, копируем код Листинга 1 и выполняем наш *скетч*.

Листинг 1: Первая линия в Processing

```
1  void setup(){
2    size(300,300);
3    background(0);
4    smooth();
5    noLoop();
6  }
7
8  void draw(){
9    strokeWeight(30);
10   stroke(100);
11   line(100,100, 200, 200);
12 }
```

При выполнении *скетча* вы должны увидеть окно с темным фоном и диагонально расположенной линией, как на Рисунке 2.

Прежде чем погружаться в описание каждой строки кода, необходимо вспомнить, что для однозначного определения отрезка (в виду того, что термин «отрезок» не упоминается в официальной дукументации по Processing, мы будем использовать термин «линия» на ряду с ним, хотя конечно линия и отрезок суть разные понятия) на плоскости следует задать координаты его концов. А именно, надо указать координату по горизонтальной оси и координату по вертикальной оси точки начала отрезка и точки его конца.

11 строка Листинга 1 отвечает за указание координат точек начала и конца отрезка:числа 100 и 100 отвечают соответственно за X и Y начальной точки, а 200 и 200 – за X и Y конечной точки.

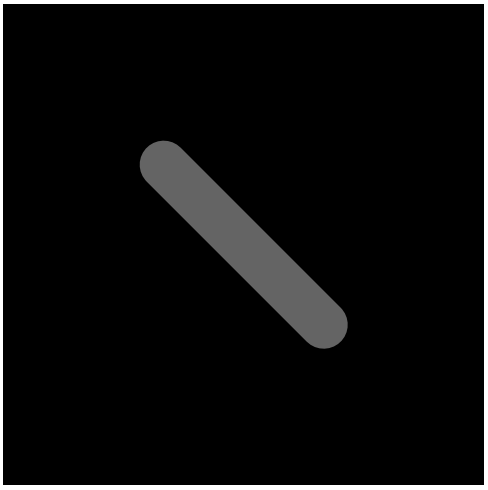


Рис. 2: Результат выполнения Листинга 1. Диагональный отрезок на черном фоне.

Задание 1. Измените код (Листинг 1), нарисовав отрезок из точки с координатами 50, 50 в точку с координатами 250, 250. Подсказка: изменить требуется только 11-ю строку.

Для того, что бы ориентироваться на плоскости холста, мы должны знать правила расположения его системы координат. По умолчанию в Processing начало системы координат находится в верхнем левом углу. Вправо (по горизонтальной оси) идет положительный отсчет оси X. Вниз (по вертикальной оси) идет положительный отсчет оси Y.

Задание 2. Измените код (Листинг 1), нарисовав второй отрезок, чтобы получилась картина как на Рисунке 3. Подсказка: требуется добавить одну строку кода после текущей 11-й строки.

Кратко объяснить работу кода Листинга 1 можно следующим образом:команда на отрисовку отрезка выполняется в 11-й строке, отрезок рисуется цветом, который мы указываем в команде в 10-й строке (100 при ограничении от 0 до 255, от черного к белому). Толщину отрезка мы задаем, выполнив команду в 9-й строке. Цифра 30 означает толщину обводки в 30 пискелей.

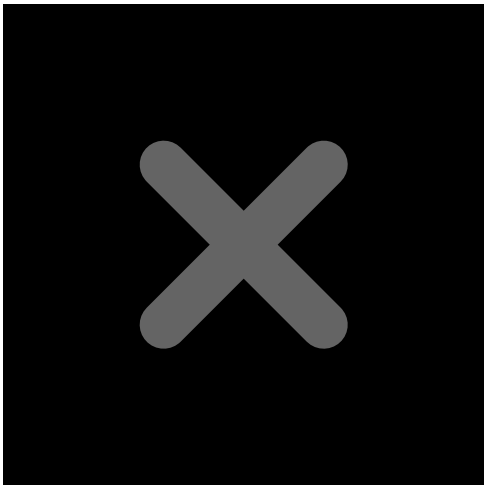


Рис. 3: Результат выполнения задания 2.
Две линии крестом на черном фоне.

Отрезок должн быть отрисован на холсте, размеры, параметры и свойства которого мы задаем в строках со 2-й по 5-ю. Указываем размер холста *size(300,300)*, заливаем фон черным цветом *background(0)*.

Исходный код Листинга 1 можно понимать более детально и полно следующим образом: код строка за строкой обрабатывается компьютером, и если нет синтаксических ошибок (и ошибок уровня компиляции), то вначале выполняется блок кода под названием *setup*. В блоке *setup*, так же как и в блоке *draw*, исходный код помещается внутри фигурных скобок. Фигурная скобка открывается в строке 1 и закрывается в строке 6 для блока *setup*. Блок кода такого вида принято называть методом.

Итак, метод *setup()* выполняется программой первым и один раз. Конечно, вы можете написать программу которая будет принудительно его вызывать, но сейчас мы этого касаться не будем. Далее вызывается метод *draw()*. В нашем примере метод *draw()* вызывается тоже один раз.

Обратите внимание, что вызовов методов *setup* и *draw* вы не видите. Таким образом, вам остается только поверить мне или документации к Processing.

Из названия метода *setup()* можно сделать вывод, что в нем вызываются методы (другие блоки кода), которые относятся к настройке нашего скетча. Действительно, во 2-й строке мы принудительно (то есть в открытую для нас с вами) вызываем метод *size()*. Причем в круглых скобках мы написали две цифры 300 и 300 (через запятую). Документация подскажет, что эти две цифры отвечают за размеры окна (холста для рисования). В примере мы создаем холст размером 300 на 300 пикселей.

Методы *setup()* и *draw()* объявляются нами вручную. Мы формируем логику работы этих методов. А, например, методы *size()*, *noLoop()* и *stroke()* за нас объявили разработчики Processing. И нам даже не обязательно знать, где они объявлены и как именно они реализованы – в этом и заключается простота такого подхода программирования: мы только читаем документацию, в которой написано, например, что вызов метода *background()* с аргументом 0, означает заливку всего поля холста черным цветом (в Processing принято отсутствие цвета брать за 0, а максимальное его значение за 255).

Задание 3.
В 3-й строке кода Листинга 1 необходимо поменять значение аргумента метода *background()* на 150, запустить *скетч*, затем поменять значение на 255 и снова запустить *скетч*. Вы должны увидеть, как меняется цвет фона вашего холста.

В 4-й строке Листинга 1 мы вызываем метод *smooth()*, который «включает» режим сглаженного рисования. В принципе, его можно не вызывать совсем, но тогда наш рисунок будет менее привлекательным.

В 5-й строке мы вызываем метод *noLoop()*. К этому методу мы будем возвращаться при работе с анимацией. Сейчас же давайте примем, что вызов этого метода означает, что наш холст будет отрисовываться один раз на экране. За отрисовку холста будет отвечать метод *draw()*, который Processing вызывает тоже один раз

В методе *draw()* вызываются методы: *strokeWeight()* – отвечает за установление толщины линии в пикселях: в нашем случае толщина линии 30 пикселей; *stroke()* – отвечает за цвет линии (обводки): в нашем случае цвет – 100 по градации серого (от 0 до 255, как у *background*). И далее идет непосредственно отрисовка линии с помощью вызова метода *line()* с четырьмя аргументами в 11-й строке Листинга 1.

В этом параграфе мы создали свое первое приложение в Processing IDE на языке программирования Processing. Мы нарисовали одну пару отрезков, для чего познакомились с несколькими методами. Полное описание метода для работы с отрезками вы можете найти на странице официальной документации https://processing.org/reference/line_.html. Обводка представляет особый интерес: для управления ею можно использовать ряд методов: *strokeWeight()*, *strokeJoin()*, *strokeCap()*.

3.2 Квадрат и прямоугольник

Processing имеет полноценный набор встроенных методов для рисования не только отрезков линий. Следующий шаг – будет формирование из линиий геометрических фигур: работа с прямоугольником и квадратом.

Прямоугольник можно рассматривать как четыре линии, расположенные определенным способом, а можно задавать его и другими параметрами. Если договориться, что одна из сторон прямоугольника будет параллельна оси, то мы можем задать его левую верхнюю вершину и ширину с высотой. Этих параметров вполне достаточно, и они наиболее естественны для описания прямоугольника. Рассмотрим Листинг 2.

Листинг 2: Рисование прямоугольников

```
1  void setup(){
2    size(400, 400);
3    smooth();
4    noLoop();
5    background(10);
6    strokeWeight(10);
7    stroke(150);
8  }
9
10 void draw () {
11   fill(250);
12   rect(100, 100, 100, 100);
13   fill(50);
14   rect(200, 200, 50, 100);
15 }
```

Результат работы кода Листинга 2 показан на Рисунке 4.

Результатом выполнения кода Листинга 2 будут два прямоугольника разных цветов. Цвет заливки фигуры, в нашем случае прямоугольника, определяется в строках 11 и 13. Метод *fill()*, аналогично методу *stroke()*, принимает аргументы, отвечающие за цвет. Интерпретировать работу метода *fill()* можно следующим образом: мы погрузили виртуальную кисть в определенную краску и все следующие фигуры будем заливать этой краской. В 13-й строке мыобмакнули нашу виртуальную кисть в другую краску, и второй прямоугольник получил заливку другим цветом.

Задание 4. Измените код Листинга 2 так, чтобы светлый прямоугольник располагался ниже, а более темный прямоугольник – выше, чем светлый.

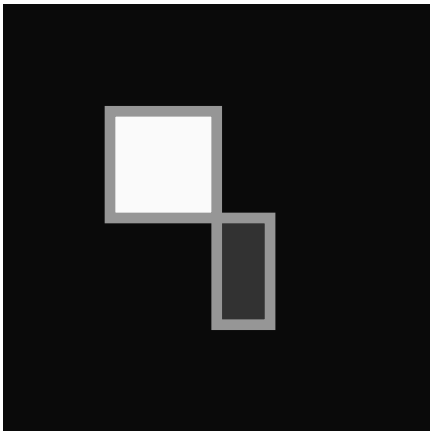


Рис. 4: Результат работы кода Листинга 2.
Два прямоугольника на черном фоне.

Кроме самих прямоугольников следует обратить внимание на обводку. Ширина обводки задается в строке 6 уже знакомым нам методом *strokeWeight()*. А цвет заливки в 7-й строке задается с помощью вызова метода *stroke()*.

Однако не всегда бывает удобно отрисовывать прямоугольники, начиная с верхнего левого угла. В Processing предусмотрели эту ситуацию и создали метод *rectMode()*. С его помощью можно менять смысл (т.е. последующую внутреннюю обработку) аргументов, принимаемых методом *rect()*. На странице официальной документации https://processing.org/reference/rectMode_.html указано, что можно использовать четыре варианта отрисовки: *CORNER* – вариант по умолчанию, *CORNERS* – вторые два аргумента (метода *rect()*) обрабатываются как координаты правого нижнего угла (а не ширина с высотой), *CENTER* – первая пара аргументов отвечает за центр прямоугольника (а не за верхний левый угол), *RADIUS* – так же как и *CENTER*: вторые два аргумента отвечают за половину соответственно ширины и высоты.

3.3 Круг, овал и точка

Рассмотрим Листинг 3. С 1-й по 9-ю строки мы, уже по традиции, объявляем метод *setup()*. В нем, в 6-й строке задаем цвет заливки будущих фигур, в 7-й строке указываем цвет обводки,в 8-й строке – толщину обводки.

Листинг 3: Рисуем 4 круга

```
1  void setup() {
```

```
2     size(500, 500);
3     smooth();
4     background(255);
5     noLoop();
6     fill(50, 80);
7     stroke(100);
8     strokeWeight(3);
9   }
10
11  void draw() {
12    ellipse(250,200,100,100);
13    ellipse(250-50,250,100,100);
14    ellipse(250+50,250,100,100);
15    ellipse(250,250+50,100,100);
16  }
```

С 11-й по 16-ю строки объявляем метод *draw()*. Тут начинается самое интересное:в 12-й, 13-й, 14-й и 15-й строках вызывается один и тот же метод *ellipse()*, но с разными аргументами. В каждой из строк передается по четыре аргумента, причем последние два одинаковые (100, 100). Взгляните на результат выполнения кода Листинга 3 (Рисунок 5).

Рисунок 5 показывает, что в результате работы нашего приложения отрисовались четыре круга полупрозрачного серого цвета с темной обводкой, причем размеры кругов одинаковые: их ширина и высота по 100 пикселей. Действительно, если обратиться к документации по методу *ellipse()* по адресу https://processing.org/reference/ellipse_.html, то мы увидим, что последние два аргумента метода отвечают за ширину и высоту. А первые два аргумента отвечают за расположение центра круга по оси X и Y соответственно. Тут следует отметить, что, так же как и при рисовании прямоугольников (*rect()*), мы можем изменить смысл передаваемых аргументов с помощью метода *ellipseMode()*. Подробную информацию о методе *ellipseMode()* вы найдете на странице официальной документации https://processing.org/reference/ellipseMode_.html.

Обратимся к детальному рассмотрению значений аргументов. В 12-й строке первый аргумент 250 – это половина ширины нашего окна приложения, которую мы задаем в строке 2. Значит, первый круг будет отрисован по центру и сдвинут вниз на 200 пикселей – второй аргумент. Далее, в 13-й строке первый аргумент 250-50 означает, что круг будет отрисован на 50 пикселей влево относительно середины окна приложения. Второй аргумент в 13-й строке говорит нам, что вертикально круг будет расположен по центру. Аналогичная логика работает по отношению к 14-й строке, только круг мы сдвигаем вправо. 15-я строка работает так же, как12-ая, но только круг сдвинут на 50 пикселей вниз от центра. Почему же мы вводим цифру 50? Несложно заметить, что 50 – это радиус

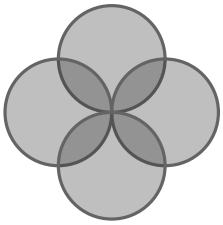


Рис. 5: Результат работы Листинга 3.
Четыре круга на белом фоне.

круга, другими словами, половина ширины или половина высоты.

В строках 13, 14 и 15 мы использовали арифметические операторы сложения и вычитания. В Processing есть возможность использовать следующий набор операторов: сложение (+), вычитание(−), инкремент – увеличение на единицу (++), декремент – уменьшение на единицу (−−), умножение (∗), деление (/), вычисление остатка от деления (%). Также есть возможность использовать ассоциативные операторы, такие как *+=*: например после такого оператора *i += 2*, значение переменной *i* увеличится на 2 единицы. Более детально операторы представлены на страницах официальной документации https://processing.org/reference/ раздел *Operators*.

Мы дали описание всех строк кода Листинга 3 отдельно от самого кода. Однако когда вы вернетесь к нему через несколько дней или недель, то вам, возможно, будет сложно вспомнить эту логику отрисовки. Поэтому предлагается внести описание в код, изменить его, сохранив логику и результат. Рассмотрим код Листинга 4.

Листинг 4: Рисуем четыре круга еще раз

```
1  int windowWidth = 500;
2  int windowHeight = 500;
3  int ellipseSize = 100;
4
5  void setup() {
6    size(windowWidth, windowHeight);
```

```
7     smooth();
8     background(255);
9     fill(50, 80);
10    stroke(100);
11    strokeWeight(3);
12    noLoop();
13  }
14
15  void draw() {
16    ellipse(windowWidth/2, windowHeight/2 - ellipseSize/2,
          ellipseSize, ellipseSize);
17    ellipse(windowWidth/2 - ellipseSize/2, windowHeight/2,
          ellipseSize, ellipseSize);
18    ellipse(windowWidth/2 + ellipseSize/2, windowHeight/2,
          ellipseSize, ellipseSize);
19    ellipse(windowWidth/2, windowHeight/2 + ellipseSize/2,
          ellipseSize, ellipseSize);
20  }
```

Первое отличие кода Листинга 4 можно заметить в 1-й, 2-й и 3-й строках. В них мы определяем переменные *windowWidth*, *windowHeight* и *ellipseSize*. Имена переменных подскажут вам их функции: они отвечают за ширину и высоту окна и за размер круга. Иными словами, в этих переменных мы будем, как в коробках, хранить соответствующие значения. Имена переменным мы придумываем сами, лучше использовать ясные и короткие названия. Однако надо понимать, что некоторые имена уже используются в Processing, например, *setup* и *draw*.

В Processing мы должны указать тип переменных, которые мы создаем. В нашем случае мы указали тип переменных как *int*. Это означает, что мы хотим, чтобы в нашей переменной хранились целые числа. Т.е. если мы попытаемся написать, например *int windowWidth = 5.67;*, то это будет ошибкой: программа на заработает и Processing выдаст нам сообщение на эту тему. Целые числа относятся к так называемым примитивным типам данных.

Processing может работать со следующими примитивными типами данных: целочисленные типы – *byte, int, long* ; числа с плавающей запятой – *float, double*, причем плавающая запятая обозначается точкой, например *5.6*, а не *5,6* ; символьный тип – *char* ; логический тип – *boolean*. Типы для работы с цветом – *color* и со строками – *String* стоят особняком, но также активно используются в Processing. Более подробно вы можете посмотреть типы данных на странице официальной документации https://processing.org/reference/ в разделе Data.

Мы объявили переменные *windowWidth*, *windowHeight* и *ellipseSize* вне методов *setup()* и *draw()* для того, чтобы эти переменные были до-

ступны как в методе *setup()*, так и в методе *draw()*. Переменные можно было объявить и в методе *setup()*, но тогда бы они были бы доступны только в нем, а обратиться к ним из метода *draw()* было бы невозможно. Конечно, вы можете попробовать, но Processing выдаст вам сообщение об исключительной ситуации (*exception*), и программа не заработает.

Задание 5. Измените значение переменной *ellipseSize* в 3-й строке кода Листинга 4 на 200. Вы должны увидеть, как круги увеличатся в размерах, сохранив пропорциональную картину. Далее программно измените размер окна приложения.

Метод *draw()* немного поменяет свой вид. Разберем для примера 16-ю строку кода Листинга 4. В ней мы также вызываем метод *ellipse()* и передаем ему четыре аргумента. Но в отличие от Листинга 3 аргументы являются не константами, а переменными. Первый аргумент *windowWidth/2*, отвечающий за координату X центра нашего первого круга, – это половина ширины окна; *windowHeight/2* – это половина высоты окна приложения и т. д.. Таким образом, вы можете не просто смотреть на код, но и понимать его, а это очень важно. Видя код, вы, как бы выступая в роли самого Processing, можете «проигрывать» строку за строкой у себя в голове, отрисовывая тот или иной объект, окрашивая его в разные цвета: таким образом вы должны представлять результат работы программы до ее выполнения.

Видоизменим код Листинга 4, добавив в него два свойства *ellipseWidth* и *ellipseHeight* в коде Листинга 5.

Листинг 5: Рисуем четыре овала

```
1  int windowWidth = 500;
2  int windowHeight = 500;
3  int ellipseSize = 100;
4  int ellipseWidth = 200;
5  int ellipseHeight = 300;
6
7  void setup() {
8    size(windowWidth, windowHeight);
9    smooth();
10   background(255);
11   fill(50, 80);
12   stroke(100);
13   strokeWeight(3);
14   noLoop();
```

```
15  }
16
17  void draw() {
18    ellipse(windowWidth/2, windowHeight/2 - ellipseSize/2,
          ellipseWidth, ellipseHeight);
19    ellipse(windowWidth/2 - ellipseSize/2, windowHeight/2,
          ellipseWidth, ellipseHeight);
20    ellipse(windowWidth/2 + ellipseSize/2, windowHeight/2,
          ellipseWidth, ellipseHeight);
21    ellipse(windowWidth/2, windowHeight/2 + ellipseSize/2,
          ellipseWidth, ellipseHeight);
22  }
```

Код Листинга 5 практически полностью повторяет предыдущий пример, во всяком случае, по логике отрисовки геометрических фигур. Мы внесли изменения в последние аргументы метода *ellipse()* в каждом его вызове. Теперь вместо кругов мы получили эллипсы. Результат работы кода Листинга 5 показан на Рисунке 6.

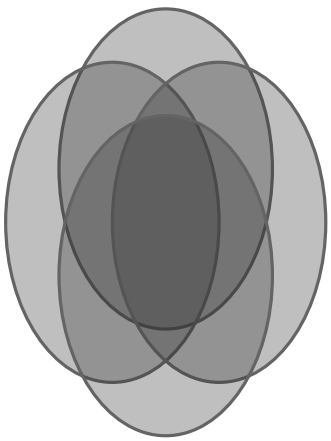


Рис. 6: Результат выполнения Листинга 5.
Четыре овала на белом фоне.

Нарисовать точку от руки на бумаге проще простого, однако в Processing для отрисовки точки придется использовать уже знакомые вам принципы. Для того, чтобы нарисовать точку, вам требуется знать ее координаты. Рассмотрим код Листинга 6.

Листинг 6: Рисуем четыре круга разного размера

```
1  void setup() {
```

```
2     size(500, 500);
3     noLoop();
4   }
5
6   void draw() {
7     smooth();
8     background(100);
9     stroke(255);
10    strokeWeight(10);
11    point(200, 200);
12    strokeWeight(30);
13    point(300, 200);
14    strokeWeight(50);
15    point(200, 300);
16    strokeWeight(80);
17    point(300, 300);
18  }
```

Результат работы кода Листинга 6 представлен на Рисунке 7. В 10-й строке кода мы вызывает метод *strokeWeight()* с аргументом 10, таким образом, наша точка будет размером 10 пикселей. Точку отрисовываем с помощью вызова метода *point()*, аргументами передаем координаты желаемой точки. В строках 12, 14 и 16 мы последовательно увеличиваем размер обводки и получаем в результате точки разной величины.

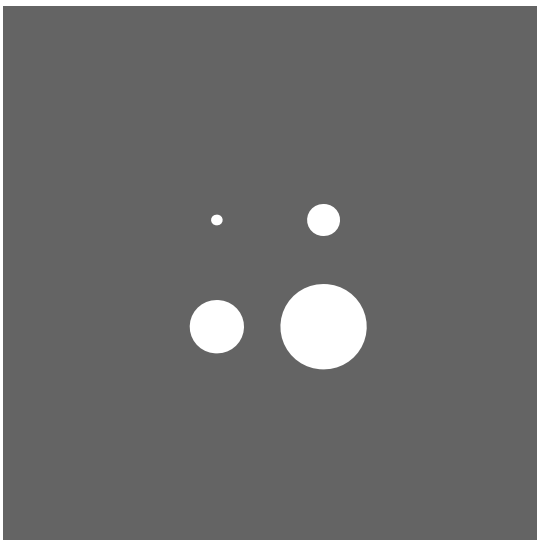


Рис. 7: Результат выполнения Листинга 6. Четыре круга разного размера на сером фоне.

Точки могут отрисовываться друг на друге и иметь разные размеры как и любые другие геометрические примитивы Processing.

Задание 6. Измените код Листинга 6 таким образом, что бы в результате его выполнения получилось изображение как на Рисунке 8.

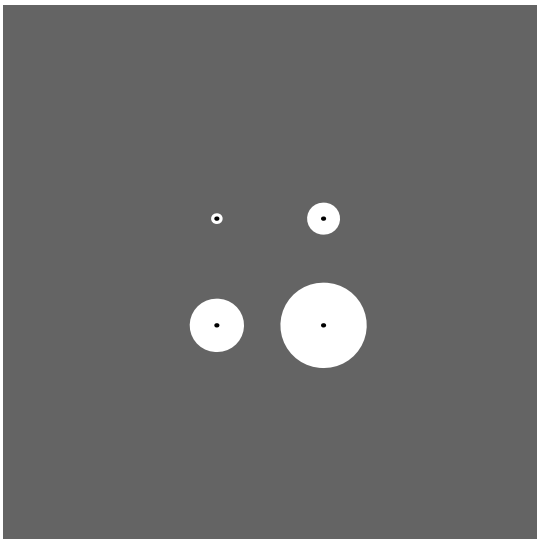


Рис. 8: Четыре круга разного размера на сером фоне с черными точками в центрах.

В конце главы хочется отметить, что рисование в Processing многим может показаться непривычным, однако оно просто и понятно. Да, пока наше рисование больше напоминает черчение, но это только начало. В этой главе мы не коснулись таких 2D примитивов, как сектора, произвольные четырехугольники и треугольники. Мы сделали это намеренно: чем больше углов и параметров рисования, тем больше их числовых значений в коде и тем сложнее создание кода. К кривым мы обратимся в соответствующей главе. Настоятельно рекомендую вам читать документацию по Processing, где в полном объеме описываются возможности этого языка и редактора.

В этой главе были рассмотрены базовые возможности языка Processing. Не стоит сравнивать Processing с профессиональными редакторами компьютерной графики: во-первых, рисование примитивов в Processing – это не самая интересная задача, скорее это просто хорошая возможность; во-вторых для статических изображений , действительно, проще нарисовать геометрию полностью вручную, не используя

Processing. Но если вам требуется «оживить» ваше произведение искусства, то тогда вам просто не обойтись без анимирования и интерактивного взаимодействия, в чем вам и поможет Processing и о чем пойдет речь в следующих главах.

4 Система координат

Когда мы обсуждали первую линию (Листинг 1), мы говорили о системе координат Processing. В этой главе мы остановимся на системе координат более подробно.

По умолчанию в Processing используется двухмерная система координат с нулем в верхнем левом углу окна приложения. Ось X направлена вправо, ось Y вниз. Система координат, в том числе, зависит и от типа механизма отрисовки (от типа *renderer*). Типе отрисовки можно указывать третим аргументом метода *size()*. Более детально метод *size()* описан на странице официальной документации https://processing.org/reference/size_.html.

4.1 Перемещение и вращение

В Processing с системой координат можно работать как с физическим объектом. Иногда бывает удобно, например, поворачивать несколько объектов сразу, группой, а не вращать их по очереди. Такая же задача может стоять при необходимости перемещения объектов.

В коде Листинга 7 представлена логика перемещения и вращения системы координат Processing.

Листинг 7: Перемещение и вращение системы координат

```
1
2  void setup() {
3    size(600, 600);
4    noLoop();
5  }
6
7  void draw() {
8
9    background(100);
10   smooth();
11   strokeWeight(50);
12
13   translate(width/2, height/2);
14   stroke(210);
15   line(0,0,250,0);
16
17   rotate(PI/4);
18   stroke(175);
19   line(0,0,250,30);
20
21   rotate(PI/4);
```

```
22      stroke(140);
23      line(0,0,250,30);
24
25      rotate(PI/4);
26      stroke(105);
27      line(0,0,250,30);
28
29      rotate(PI/4);
30      stroke(70);
31      line(0,0,250,30);
32
33      rotate(PI/4);
34      stroke(35);
35      line(0,0,250,30);
36
37      rotate(PI/4);
38      stroke(0);
39      line(0,0,250,30);
40  }
```

Результат выполнения кода Листинга 7 можно увидеть на Рисунке 9. Со 2-й по 5-ю строку кода Листинга 7 мы объявляем метод *setup()*, далее с 7-й по 40-ю – метод *draw()*. В методе *draw()* мы делаем, по сути, однотипные операции, а именно, вызываем метод *rotate()*, затем *stroke()* и *line()*. Обратите внимание, что аргументы метода *line()* остаются одинаковыми, но линия, при этом отрисовывается в разном положении на экране.

В 9-й строке кода мы вызываем метод *background()* с аргументом 100 для того, чтобы закрасить фон окна приложения в серый цвет. Далее идут вызовы методов *smooth()* и *strokeWeight()* для установки параметров обводки.

Перемещение начала системы координат происходит в 13-й строке. Мы вызываем метод *translate()*, которому передаем два аргумента, два значения половины ширины и половину высоты окна приложения. Метод *translate()* переместит начало системы координат в эту точку. В нашем примере начало системы координат переместится в точку с координатами 300, 300 (по первоначальной системе координат). В 14ой и 15ой строках мы отрисовываем первую, самую светлую линиию. Все вызовы, связанные с координатами, который будут сделаны после перемещения системы координат в новую точку, будут пересчитаны Processing в рамках нового положения системы координат.

Вращение системы координат мы впервые производим в строке 17. Метод *rotate()* принимает аргументом число как угол (в радианах), на который требуется повернуть систему координат по часовой стрелке.

Вращение происходит относительно начала системы координат. Если вы передаете отрицательные числа методу *rotate()*, то вращение будет происходить против часовой стрелки. В Processing существуют заранее объявленные константы, например *PI*. Константа – это наименование неизменного значения: знак 2, например, всегда обозначает количество «два» – и это константа. Константы удобно использовать, когда речь идет о радианной мере угла.

Трансформации с системой координат суммируются, т.е. в нашем примере к 18-й строке общая трансформация системы координат может быть представлена как «смещение + вращение». После того как мы повернули систему координат, мы отрисовали линию в строке 19. Так как окно приложения мы не вращаем, то линия отрисовалась под углом к предыдущему отрезку.

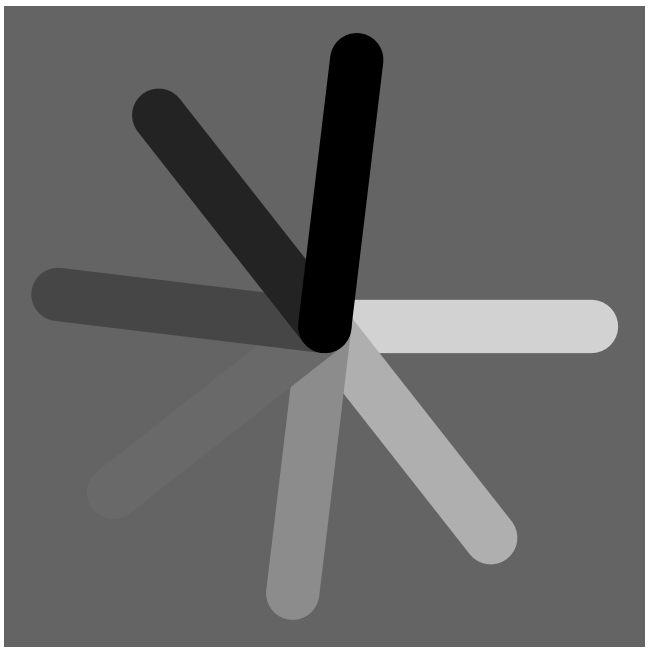


Рис. 9: Результат работы Листинга 7. Отрезки отрисованные при смещении и вращении системы координат.

Второй поворот системы координат мы производим в 21-й строке. Вращаем ее на тот же угол *PI/4*. Теперь общая трансформация системы координат представляет собой «смещение + вращение + вращение», так что отрисованный отрезок линии в 23-й строке будет еще больше отклонен от первого отрезка. Мы повторяем эти операции для демонстрации суммирования трансформаций и далее.

Задание 1. Измените код Листинга 7 таким образом, чтобы вращение системы координат происходило против часовой стрелки.

Еще одним видом трансформации является масштабирование. Масштабирование системы координат в Processing возможно с помощью вызова метода *scale()*. Рассмотрим применение масштабирования системы координат на примере кода Листинга 8.

Листинг 8: Рисуем три отрезка в разном масштабе системы координат

```
1  void setup() {
2    size(600, 600);
3    noLoop();
4  }
5
6  void draw() {
7
8    background(100);
9    smooth();
10   strokeWeight(50);
11   stroke(200);
12
13   translate(width/2, height/2 - 100);
14   line(-100,0,100,0);
15
16   translate(0, 100);
17   scale(1.5, 1.5);
18   line(-100,0,100,0);
19
20   translate(0, 100);
21   scale(1.5, 1.5);
22   line(-100,0,100,0);
23 }
```

Мы не будем детально останавливаться на уже знакомых нам методах кода, таких как *size()*, *background()*, *stroke()* и т.д. Сразу перейдем к сути примера. В 13-й строке мы переместили центр системы координат в центр окна приложения и отрисовали отрезок линии в 14-й строке. После чего мы еще раз переместили систему координат вниз на 100 пикселей (строка 16). Прежде чем отрисовать новый отрезок, мы выполнили масштабирование нашей системы координат в строке 17, т.е. мы как бы увеличили ее в 1.5 раза, ввиду чего отрезок, отрисованный в 18-й строке толще и больше, чем первый. Результат выполнения кода Листинга 8 можно увидеть на Рисунке 10.

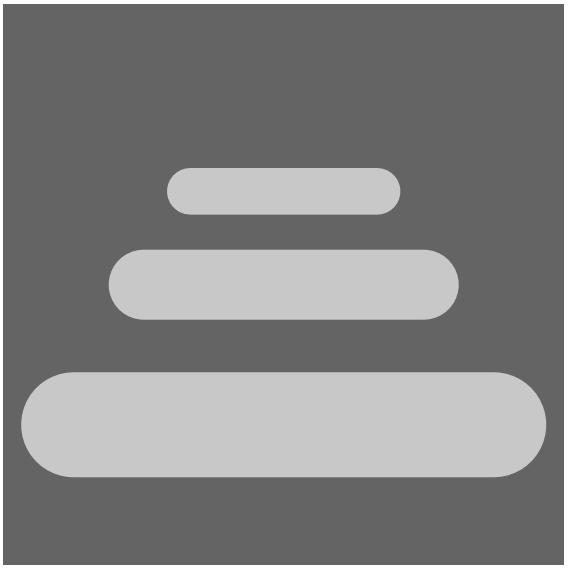


Рис. 10: Результат выполнения Листинга 8. Рисуем три отрезка в разном масштабе системы координат.

Операции трансформации в этом примере к 22-й строке можно представить как «смещение + смещение + масштабирование + смещение + масштабирование», поэтому третий отрезок отрисовывается ниже, толще и больше, чем предыдущее

Задание 2. Измените код Листинга 8 таким образом, чтобы третий отрезок отрисовывался выше первого, но с его же степенью масштабирования.

В этом параграфе мы с вами рассмотрели возможность трансформации системы координат. Мы представили ее как последовательное сложение типов трансформации. При необходимости возвращения к предыдущему состоянию мы можем сделать обратные действия. Однако чаще всего стоит задача в запоминании текущего состояния системы координат с возможность его последующего применения.

4.2 Состояния системы координат

Задачу запоминания текущего состояния системы координат в Processing решает метод *pushMatrix()*. Он сохраняет текущие трансформации. После того как вы сохранили нужные вам трансформации, вы можете со-

вершать новые, а когда потребуется, вернуть сохраненные трансформации с помощью метода *popMatrix()*. Рассмотрим пример такой работы в коде Листинга 9.

Листинг 9: Состояния системы координат в действии

```
1
2  void setup() {
3    size(600, 600);
4    noLoop();
5  }
6
7  void drawMyScene(float myColor){
8    fill(myColor);
9    rect(0,50, 150, 50);
10   rect(50,0, 50, 150);
11 }
12
13 void draw() {
14   background(20);
15   smooth();
16   noStroke();
17
18   pushMatrix();
19   translate(100, 0);
20   rotate(PI/4);
21   drawMyScene(180);
22   popMatrix();
23
24   pushMatrix();
25   translate(220, 110);
26   rotate(PI/6);
27   scale(2);
28   drawMyScene(220);
29   popMatrix();
30
31   pushMatrix();
32   translate(520, 350);
33   rotate(PI/3);
34   scale(1.4);
35   drawMyScene(80);
36   popMatrix();
37
38 }
```

Прежде чем разбирать этот пример, вернемся к предыдущему (Листинг 7). В нем мы одну и ту же строку кода с рисованием отрезка писали несколько раз, что не очень удобно. Указание цвета тоже можно было бы оптимизировать. Вообще, чем меньше кода, тем лучше! Итак, если тре-

буется выполнить какую-либо часть кода несколько раз, то можно обойтись без копирования: просто вынесите её в отдельный метод, например, *setup()* или *draw()*, а потом вызывайте по мере необходимости. Более того, этому методу можно передавать аргументы. В коде Листинга 9 мы разберем пример объявления метода и пример сохранения трансформаций системы координат. Результат показан на Рисунке 11.

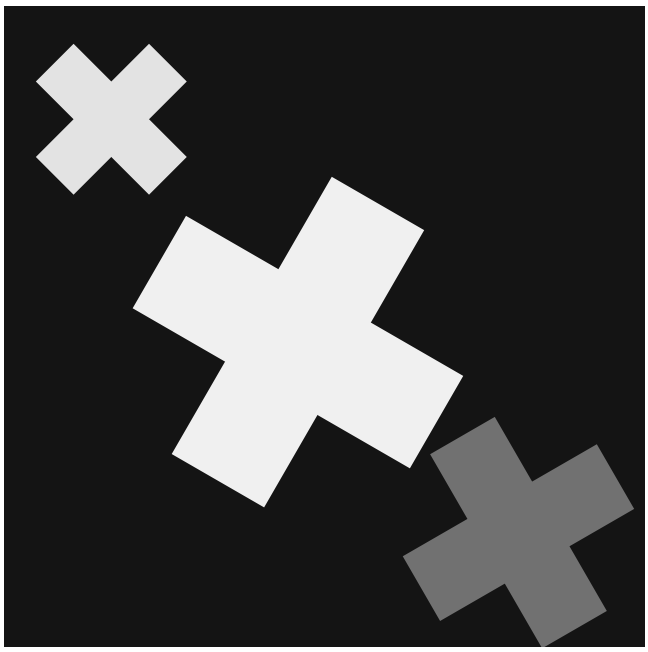


Рис. 11: Результат работы Листинга 9.
Три крестика на черном фоне.

Методы в Processing объявляются так, как показано в 7-й и 11-й строках. Вначале мы должны указать тип возвращаемого значения. Возвращаемое значение – это, например, число, которое метод подсчитал и хочет вернуть обратно вместо себя в той строке коде, где он был вызван. В нашем случае метод *drawMyScene()* не будет ничего возвращать в код, поэтому мы указываем пустой тип возвращаемого значения *void*. Наш метод принимает один аргумент – переменную *myColor* типа *float* и будет отрисовывать два прямоугольника. Причем в 8-й строке мы используем переменную *myColor* для указания цвета отрисовки в методе *fill()*. Имя переменной аргумента нашего метода, так же как и имя метода, мы могли выбрать любое (с известными ограничениями). Вызывать наш метод мы будем в строках 21, 28 и 35.

В самом приложении мы делаем три группы вызовов методов. Все группы занимаются схожими задачами, поэтому разберем работу толь-

ко первой из них. Первая группа, с 18-й по 22-ю строку, начинается с того, что мы сохраняем текущее состояние системы координат с помощью вызова метода *pushMatrix()* в 18-й строке. Далее мы сдвигаем систему координат, вращаем ее и затем, в 21-й строке, вызываем наш метод *drawMyScene()* с аргументом 180. После того как отрисован крестик, мы возвращаем систему координат в исходное состояние с помощью вызова метода *popMatrix()* в 22-й строке.

Задание 3. Измените код Листинга 9 таким образом, чтобы крестики вращались на *PI/4* по отношению друг к другу. Вызов метода *rotate()* перенесите в метод *drawMyScene()*.

Детально о возможностях трансформации системы координат можно узнать на странице официальной документации https://processing.org/reference/ , в разделе *Transform*.

Код Листинга 9 наглядно показывает, чем отличается объявление метода и его вызов. В строках с 7-й по 10-ю мы объявляем метод *drawMyScene()*, реализовываем его функционал. В строках 21, 28 и 35 мы вызываем метод *drawMyScene()*, передавая ему аргументы и ожидая адекватного результата. Мы видим исходный код метода *drawMyScene()*, но нам достаточно знать, что он «делает», какие аргументы принимает и что возвращает. Точно так же объявлены и остальные методы, такие как *size()*, *fill()* и другие, разница лишь в том, что мы не видим их исходный код.

В этой главе мы детально разобрали работу с системой координат Processing. Сравнивая координатные системы Processing и реального холста художника можно отметить больше различий, чем сходств. Система координат Processing подвижна, вы можете вращать и масшабировать ее со всеми объектами. У классического холста таких возможностей нет. Однако можно найти параллель с рабочим столом художника-мультипликатора (так называемых «рисованных» мультипликационных фильмов), когда персонажи могут быть совмещены с разными системами координат.

5 Цвет, ритм и движение

Важность цветовых и световых эффектов в работе современных цифровых художников сложно переоценить. Однако получение цвета на экране компьютера непростой процесс. В компьютерной графике на техническом уровне цвет создается от свечения пикселей экрана, управлять которым вам поможет богатый арсенал Processing.

Еще одной важной составляющей современных произведений является ритм. Мы встречаем ритм повсюду: от повторяющейся смены времен года и движения солнца до музыкального ритма знакомой мелодии. Реализация ритмических мотивов возможна в Processing без каких-либо сложностей.

С ритмом тесно связано такое понятие, как движение. К теме движения обращались художники всех стилей, жанров и эпох: вопросы передачи движения на холсте всегда были актуальны. Современные художники и дизайнеры могут дать свой ответ цифровыми произведениями, созданными с помощью новейших технических средств, таких, например, как Processing.

5.1 Работаем с цветом

В этом параграфе вы научитесь работать с цветом в Processing, используя уже знакомые вам геометрические объекты. В Processing они могут иметь цвет и прозрачность разной степени.

Мы уже задавали в качестве аргумента, например, метода *stroke()* градации серого цвета для обводки. В документации (https://processing.org/reference/stroke_.html) указано, что этот метод может принимать разное количество аргументов:

1. *stroke(rgb)*
2. *stroke(rgb, alpha)*
3. *stroke(gray)*
4. *stroke(gray, alpha)*
5. *stroke(v1, v2, v3)*
6. *stroke(v1, v2, v3, alpha)*

Градации серого попадают в 3-ий и 4-ый вариант использования. В Processing существует 255 градаций серого от 0 до 255. Максимально

черный цвет – это 0, минимально черный, т.е. белый цвет, имеет числовое значение 255.

Остальные цвета в компьютерной графике представляются триадой Red (Красный), Green (Зеленый) и Blue (Синий). Смешивая эти цвета в различных комбинациях, вы можете получить любой требуемый вам цвет. Эта аддитивная цветовая модель не единственно возможная и не совсем полная, но очень сильно распространенная в «цифровом» мире.

Первый вариант использования метода *stroke()* как раз и связан с триадой Red, Green и Blue в так называемой шестнадцатиричной форме (*hexadecimal notation*). Этот формат чаще всего используется в CSS и HTML. Рассмотрим Листинг 10.

Листинг 10: Рисуем три цветных линии

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    noLoop();
5  }
6
7  void draw() {
8    background(100);
9
10   stroke(#881DCB);
11   strokeWeight(110);
12   line(100, 150, 400, 150);
13
14   stroke(#CB1DA3);
15   strokeWeight(60);
16   line(100, 250, 400, 250);
17
18   stroke(#CB1D1D);
19   strokeWeight(110);
20   line(100, 350, 400, 350);
21 }
```

В 10-й строке мы вызываем метод *stroke()* с первым аргументом *#881DCB*. Этот аргумент и является шестнадцатиричной формой записи цвета. Результат выполнения кода Листинга 10 можно увидеть на Рисунке 12.

Несмотря на свою компактность, шестнадцатиричная форма записи чаще всего неудобна для художественных задач: художнику привычнее смешивать краски. Поэтому есть смысл остановиться на возможности смешивать значения Red, Green, Blue в так называемой «аддитивной палитре». Посмотрим, как это делается в Processing на примере кода Листинга 11.

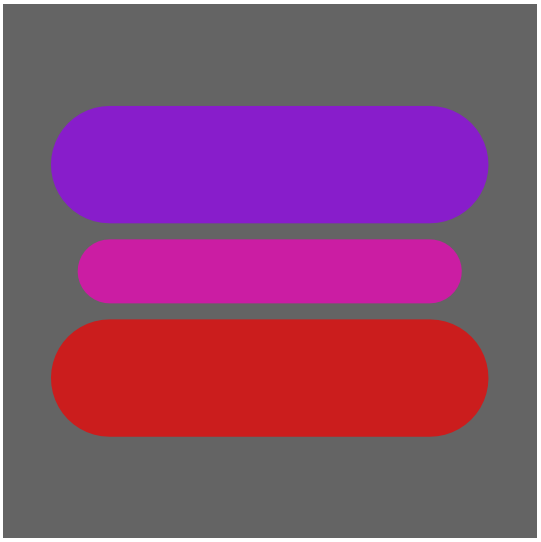


Рис. 12: Результат работы Листинга 10.
Три цветных линии на сером фоне.

Листинг 11: Рисуем три линии цветом RGB

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    noLoop();
5  }
6
7  void draw() {
8    background(100);
9
10   stroke(94, 206, 40);
11   strokeWeight(110);
12   line(100, 150, 400, 150);
13   stroke(59, 129, 25);
14   strokeWeight(60);
15   line(100, 250, 400, 250);
16   stroke(35, 77, 15);
17   strokeWeight(110);
18   line(100, 350, 400, 350);
19 }
```

В 10-й строке вызываем метод *stroke()* с тремя аргументами, задавая значение цветов Red, Green, Blue в более привычных цифрах (от 0 до 255). Число градаций каждого из Red, Green, Blue так же, как и у черного цвета, состоит из 255. Результат работы кода Листинга 11 представлен на Рисунке 13.

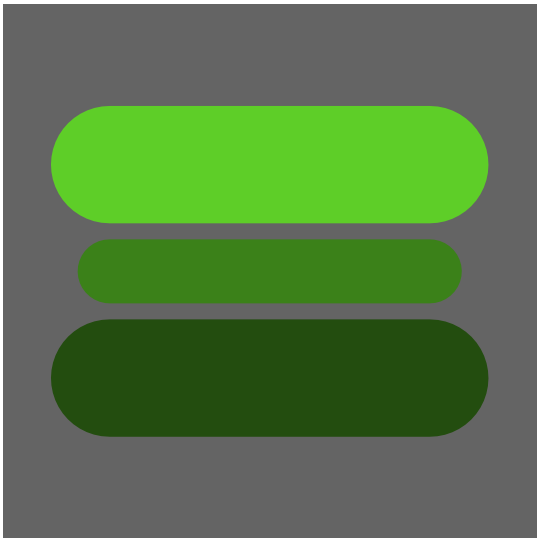


Рис. 13: Результат работы Листинга 11.
Три линии цвет которых задан в RGB режиме.

Определение необходимого вам цвета в RGB режиме может вызывать трудности, поэтому разработчики Processing включили в редактор возможность визуального выбора цвета из палитры. Для того, чтобы определить числовое значение цвета, вы можете вызвать палитру из панели управления *Tools -> Color Selector*. Появившееся окно позволит работать с выбором цвета интуитивно понятно.

Метод *stroke* может принимать и четвертый аргумент, который будет отвечать за прозрачность. Для того, чтобы рассмотреть прозрачность, мы нарисуем отрезки, наложенные друг на друга, как на Рисунке 14 (Листинг 12).

Листинг 12: Рисуем четыре отрезка разной прозрачности

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    noLoop();
5  }
6
7  void draw() {
8    background(50);
9
10   stroke(94, 206, 40, 50);
11   strokeWeight(140);
12   line(100, 100, 400, 100);
13
```

```
14      stroke(94, 206, 40, 100);
15      strokeWeight(140);
16      line(100, 200, 400, 200);
17
18      stroke(94, 206, 40, 150);
19      strokeWeight(140);
20      line(100, 300, 400, 300);
21
22      stroke(94, 206, 40, 250);
23      strokeWeight(140);
24      line(100, 400, 400, 400);
25  }
```

На Рисунке 14 мы видим, что ширина обводки линии больше, чем расстояние между ними, ввиду чего отрезки перекрывают друг друга. Код Листинга 12 отличается от предыдущего тем, что во всех вызовах метода *stroke()* (в 10-й,14-й, 18-й и 22-й строках) указывается четыре аргумента. В качестве четвертого аргумента, отвечающего за прозрачность, мы передаем последовательно увеличивающееся значение (50, 100, 150 и 250). Так достигается эффект прозрачности разной плотности.

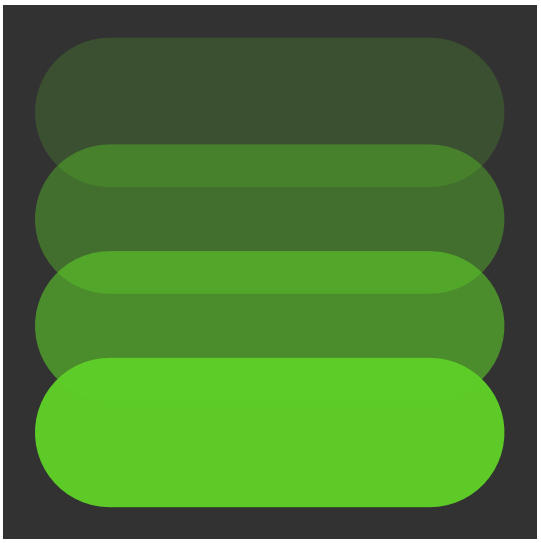


Рис. 14: Результат выполнения Листинга 12.
Четыре отрезка разной прозрачности.

Задание 1. Измените код Листинга 12 таким образом, чтобы отрисовывались не отрезки, а окружности. Прозрачность при этом должна увеличиваться книзу.

В этом параграфе мы рассмотрели возможности языка Processing в области работы с цветом в шестнадцатиричном и RGB формате и с прозрачностью. Форматы цвета в Processing определены не случайно. Processing, как инструмент художника, должен быть совместим на уровне понимания и логики с другими цифровыми инструментами – редакторами компьютерной графики. Так что, например, в Inkscape или в Abode PhotoShop вы увидите эти же цветовые модели.

5.2 Ритм с помощью циклов

В этом параграфе мы рассмотрим конструкции Processing для создания ритма. В программировании ритм используется постоянно, поэтому работа с ритмом включена практически во все языки программирования.

В Листинге 13 показан код, который очень похож на код Листинга 1. Однако метод *draw()* оставлен максимально пустым, все операции по настройке вынесены в метод *setup()*. Результат выполнения кода Листинга 13 представлен на Рисунке 15.

Листинг 13: Рисуем две наклонные линии

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    background(255);
5    strokeWeight(30);
6    noLoop();
7  }
8
9  void draw() {
10   stroke(20);
11   line(50, 200, 150, 300);
12   line(100, 200, 200, 300);
13 }
```

Вызов метода *line()* в 11-й строке и вызов метода *line()* в 12-й строке отличаются лишь значением аргументов. В первом случае это числа 50, 200, 150, 300, а во втором – 100, 200, 200, 300. Первая и третья цифры отвечают за координату Y начальной точки и координату X конечной точки соответственно. Таким образом мы сдвинули линию вправо на 50 пикселей.

Задание 2. *Измените код Листинга 13 так, чтобы отрисовывалось три линии. Третью линию, параллельную первым двум,*

Рис. 15: Результат выполнения Листинга 13.
Две диагональные линии line(50,200, 150,300) и line(110,200, 210,300).

> *необходимо расположить со сдвигом вправо на 50 пикселей, как на Рисунке 16.*

Мы можем представить первую и третью цифры в 12-й строке так: $100 = 50 + 50$ и $200 = 150 + 50$. Тогда 12-я строка будет выглядеть так: $line(50 + 50, 200, 150 + 50, 300);$. При выполнении этой строки кода вначале срабатывают операции сложения, затем вызывается метод *line()*, аргументами которого являются результаты сложения. Заметим, что результат при наших манипуляциях остается прежний: линии на холсте. Продолжим модифицировать наш пример в Листинге 14.

Листинг 14: Рисуем три наклонных линии

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    background(255);
5    strokeWeight(30);
6    noLoop();
7  }
8
9  void draw() {
10   stroke(20);
11   line(50, 200, 150, 300);
```

```
12      line(50 + 50, 200, 150 + 50, 300);
13      line(50 + 50 + 50, 200, 150 + 50 + 50, 300);
14  }
```

При запуске кода мы получаем результат (см. Рисунок 16).

Рис. 16: Результат выполнения Листинга 14.
Три диагональных линии, сдвинутые друг от друга на 50 пискелей.

Несложно заметить, что $50 + 50$, в 12-й строке – это не что иное, как $2 * 50$. Тогда строка 12 Листинга 14 может быть модифицирована так: $lin(2 * 50, 200, 150 + 50, 300);$, а строка 13 так: $lin(3 * 50, 200, 150 + 2 * 50, 300);$. Чтобы нарисовать четвертую линию, сдвинутую вправо на 50 пискелей от крайней правой, нам требуется лишь скопировать 13-ю строку кода и поменять в ней цифру 3 на цифру 4, а цифру 2 на 3, так: $lin(4 * 50, 200, 150 + 3 * 50, 300);$.

Задание 3. *Измените код Листинга 14 таким образом, что бы на холсте ыло отрисовано 7 линий, как на Рисунке 17.*

Проведем еще ряд модификаций нашего кода. В Листинге 15 показаны все операции, описанные выше. Обратите внимание на строку 11: в ней есть два интересных момента. Первый – это $1*50$, второй – $(1-1)*50$. Они приведены только для того, чтобы сделать ясной логику перехода от 11-й строке к 12-й и далее к 13-й. Эти три строки очень похожи друг

на друга. Они отличаются лишь тем, что одна «цифра» в них меняет свое значение: 1 на 2, затем на 3.

Листинг 15: Рисуем три наклонных линии. Вариант 2

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    background(255);
5    strokeWeight(30);
6    noLoop();
7  }
8
9  void draw() {
10   stroke(20);
11   line(1*50, 200, 150 + (1-1)*50, 300);
12   line(2*50, 200, 150 + (2-1)*50, 300);
13   line(3*50, 200, 150 + (3-1)*50, 300);
14 }
```

Сделаем оговорку: фраза «одна «цифра» в них меняет свое значение», конечно, не совсем корректна, так как цифра 2 всегда есть цифра 2, она не может стать тройкой или четверкой (цифра – это константа). Поэтому лучше использовать другое слово для обозначения этого воображаемого «ящика», куда помещается цифра, или, как говорят, помещается значение. Назовем этот «ящик» *переменной*. Мы уже пробовали объявлять переменные, и теперь нам от них точно никуда не деться! Теперь, надеюсь, всё стало более понятно: есть переменная, и она меняет свое значение с 1 на 2, потом на 3. Таких переменных в коде может быть несколько, и каждая меняет свое значение по-своему, т.е. так, как вы хотите. Как люди различаются по именам, так и каждой переменной присваивается своё имя. Посмотрим на код Листинга 16.

Листинг 16: Рисуем три наклонных линии. Вариант 3

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    background(255);
5    strokeWeight(30);
6    noLoop();
7  }
8
9  void draw() {
10   stroke(20);
11   int i = 1;
12   line(i*50, 200, 150 + (i-1)*50, 300);
13   i = 2;
```

```
14      line(i*50, 200, 150 + (i-1)*50, 300);
15      i = 3;
16      line(i*50, 200, 150 + (i-1)*50, 300);
17    }
```

В строке 11 Листинга 16 мы объявили переменную i и присвоили ей значение 1. Другими словами, мы вообразили некий условный «ящик», назвали его i и положили туда цифру 1. И эту переменную («ящик») мы используем дальше в коде.

Если нам требуется извлечь значение из переменной (достать цифру из «ящика»), то мы просто пишем имя «ящика» и Processing извлечет значение переменной (достанет цифру из ящика). Например, в 12-й строке мы извлекаем дважды значение перемененной i. В первый раз, что бы произвести операцию $i * 50$, второй раз операцию $(i - 1) * 50$. И в первом случае и во втором, было извлечено значение переменной равное 1.

В 13-й строке мы поменяли значение переменной i, присвоили ей значение 2. Теперь в том же «ящике» лежит навая цифра, цифра 2. В 14-й строке из переменной i будет извлекаться значение 2, а не 1, как в 12-й строке.

Эта же логика описывает 15-ю и 16-ю строки.

Отметим, что переменную требуется объявлять один раз, в нашем случае в 11-й строке. И при объявлении мы указали тип переменной – *int*: это означает, что в переменной (в этом «ящике») будут храниться только целые числа, такие как 1, 22, 213 и т.д.

И теперь самое интересное! Обратите внимание, что в Листинге 16 строки 12, 14 и 16 абсолютно одинаковые по написанию. Только значение переменной i при выполнении программы у них разное. Если строки выглядят одинаково, то нет смысла их повторять. Можно написать всего одну строку и указать программе выполнить ее несколько раз. Рассмотрим Листинг 17.

Листинг 17: Рисуем отрезки в цикле

```
1   void setup() {
2     size(500, 500);
3     smooth();
4     background(255);
5     strokeWeight(30);
6     noLoop();
7   }
8
9   void draw() {
10    stroke(20);
11    for(int i = 1; i < 8; i = i + 1){
12      line(i*50, 200, 150 + (i-1)*50, 300);
```

```
13    }
14  }
```

Результат работы кода из Листинга 17 представлен на Рисунке 17. Прежде чем разбирать конструкцию в 12-й строке, поменяем этот код. А именно, изменим метод *draw()* добавлением только одной строки (см. Листинг 18).

Рис. 17: Результат выполнения Листинга 17.
Семь линий отрисованных в цикле for.

Листинг 18: Рисуем решетку отрезками в цикле

```
1   void setup() {
2     size(500, 500);
3     smooth();
4     background(255);
5     strokeWeight(30);
6     noLoop();
7   }
8
9   void draw() {
10    stroke(20);
11    for(int i = 1; i < 8; i = i + 1){
12      line(i*50, 200, 150 + (i-1)*50, 300);
13      line(i*50 + 100, 200, 50 + (i-1)*50, 300);
14    }
15  }
```

Результат работы кода из Листинга 18 представлен на Рисунке 18.

Задание 4. Измените код строки 12 Листинга 18 следующим образом: *stroke(20, 100);*, что бы на холсте было отрисовано 7 крестов, как на Рисунке 19.

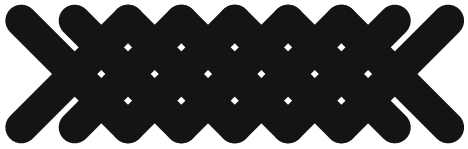


Рис. 18: Результат выполнения Листинга 18.
Семь крестов отрисованных в цикле for.

Задание 5. Измените 13-ю строку кода в Листинге 18: вместо $i = i + 1$ напишите $i = i + 2$, чтобы получить изображение, как на Рисунке 20.

Теперь можно обсудить, что происходит в 12-й строке Листинга 17 или в 13-й строке Листинга 18 (суть одна и та же). Программа, встречая конструкцию *for*, исполняет тело цикла (то, что внутри цикла, внутри фигурных скобок) до тех пор, пока проверка условия выхода из цикла проходит с положительным результатом. Оператор $i < 8$ будет возвращать *true* при значениях $i = 1, 2, 3, 4....$ А когда i примет значение 8, то оператор $i < 8$ вернет значение *false*, так как 8 не меньше 8-и ! В этот момент программа выйдет из цикла *for*.

Строки 12 и 13 Листинга 18 выполняются по очереди столько раз, сколько «срабатывает» цикл. И при каждом таком «срабатывании» (ите-

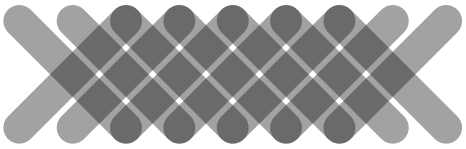


Рис. 19: Результат выполнения задания 6.
Семь полупрозрачных крестов отрисованных в цикле *for*.

рации) значение переменной i меняется, и мы отрисовываем линии с разными координатами.

Естественно, что в теле цикла можно вызывать не только метод *line()*. Если по вашей задумке требуется выполнить однотипные методы, например, с разными параметрами, то можете смело использовать циклы.

Так, например, изменим код Листинга 18, добавив в него строку с вызовом метода *stroke()*. В Листинге 19 приведен пример исходного кода, где в цикле устанавливается цвет линии и затем отрисовываются сами линии. Результат работы кода Листинга 19 приведен на рисунке Рисунке 21.

При первом вызове метода *stroke()* в 11-й строке кода мы устанавливаем цвет линий $20 * 1$ (почти полностью черный), так как при первой итерации $i = 1$. Далее, при второй итерации, цвет линии устанавливается 40, при третьей – 60 и так далее.

Листинг 19: Рисуем крестики разных оттенков в цикле

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    background(255);
5    strokeWeight(30);
6    noLoop();
7  }
8
```

Рис. 20: Результат выполнения задания 7.
Четыре полупрозрачных креста отрисованных в цикле for.

```
9  void draw() {
10   for (int i = 1; i < 8; i = i + 1) {
11     stroke(20*i);
12     line(i*50, 200, 150 + (i-1)*50, 300);
13     line(i*50 + 100, 200, 50 + (i-1)*50, 300);
14   }
15 }
```

Задание 6. Напишите в строку между 12-й и 13-й строками Листинга 19 следующий вызов метода: $stroke(160 - 20 * i)$;, что бы на холсте было отрисовано изображение, как на Рисунке 22.

Если вам требуется нарисовать сетку, т.е. расположить объекты последовательно не только по оси X, но и по оси Y, то в тело цикла нужно поместить еще один цикл со своим счетчиком итераций. Рассмотрим Листинг 20 и его результат на Рисунке 23.

Листинг 20: Полупрозрачная сетка

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    background(255);
```

```
5     strokeWeight(30);
6     noLoop();
7     stroke(0,50);
8   }
9
10  void draw() {
11    for (int i = 1; i < 8; i = i + 1) {
12      for (int k = 1; k < 4; k = k + 1) {
13        line(i*50, 100*k, 150 + (i-1)*50, 100 + 100*k);
14        line(i*50 + 100, 100*k, 50 + (i-1)*50, 100 + 100*k);
15      }
16    }
17  }
```

Рис. 21: Результат выполнения Листинга 19.
Крестики разных оттенков серого.

Циклы можно вкладывать друг в друга, как практически и все остальные конструкции языка. Так, в Листинге 20 в 11-й строке объявляется цикл (по 14-ю строку включительно), который находится внутри другого цикла, объявленного с 10-й строки по 15-ю.

Работа вложенных циклов подчиняется тем же правилам. Пройдем первые итерации так, как будто мы с вами сама программа. В 10-й строке мы попали в цикл, присвоили значение 1 переменной i. Затем проверили условие $i < 8$. Так как 1 действительно меньше 8, то, получив ответ *true*, мы попали в тело цикла. В теле цикла, в 11-й строке, мы вошли во второй цикл: присвоили значение 1 переменной k, проверили условие выхода из второго цикла $k < 4$. На текущей итерации $1 < 4$ – это *true*.

Рис. 22: Результат задания 6.
Разноокрашенные крестики.

Таким образом мы попали в тело второго цикла, в строку 12.

Как только мы отрисовали две линии в 12-й и 13-й строках, мы попали в конец внутреннего цикла, а значит, увеличили значение k на 1, как это указано в объявлении внутреннего цикла в строке 11, а именно, $k = k + 1$. Мы получили $k = 2$, проверили, что $2 < 4$ и снова попали в тело внутреннего цикла. Важно отметить, что переменная i все еще сохраняет значение 1 и пока мы не пройдем все итерации внутреннего цикла, мы из него не выйдем, а значит, не перейдем на следующую итерацию внешнего цикла и не присвоим $i = i + 1$.

Задание 7. Напишите недостающие 12-ю и 14-ю строки кода Листинга 21, чтобы на холсте было отрисовано изображение, как на Рисунке 24.

Когда мы перебрали все допустимые условием $k < 4$ значения k, то, выйдя из внутреннего цикла, мы попали в конец тела внешнего цикла, т.е. в строку 14. Мы как бы стоим перед фигурной скобкой в строке 14. По правилам цикла увеличиваем значение нашей переменной i на 1, проверяем условие выхода из внешнего цикла в строке 10. Так как $2 < 8$ – *true*, то мы снова попадаем в тело первого цикла, в котором снова заходим во внутренний цикл. Вот такой «круговорот», проход последовательно по каждому значению переменных i и k обеспечивает нам

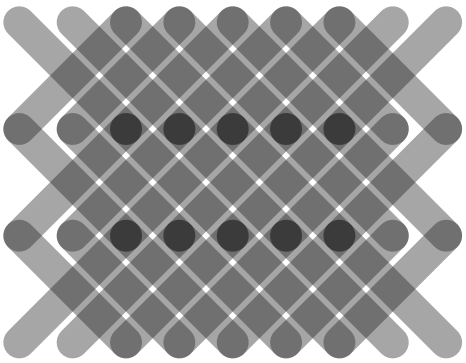


Рис. 23: Результат выполнения Листинга 20.
Полупрозрачная сетка.

сравнительно простую, в пять строк кода, возможность отрисовывать 2D сетки.

Листинг 21: Рисуем задание 7

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    background(255);
5    strokeWeight(30);
6    noLoop();
7  }
8
9  void draw() {
10   for (int i = 1; i < 8; i = i + 1) {
11     for (int k = 1; k < 4; k = k + 1) {
12
13       line(i*50, 100*k, 150 + (i-1)*50, 100 + 100*k);
14
15       line(i*50 + 100, 100*k, 50 + (i-1)*50, 100 + 100*k);
16     }
17   }
18 }
```

Также как и рисование групп линий, отрисовка групп любых фигур может быть выполнена в цикле.

Рассмотрим код Листинга 22.

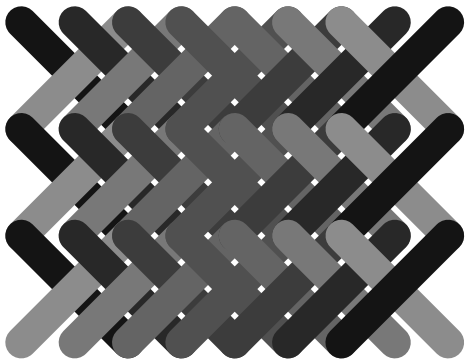


Рис. 24: Результат работы приложения задания 7.
Сетка крестов разных размеров и оттенков.

Листинг 22: Рисуем прямоугольники без обводки

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    background(255);
5    noStroke();
6    noLoop();
7  }
8
9  void draw() {
10   for(int i = 0; i < 10; i++){
11     fill(i*20);
12     rect(i*40+50, 220, 35, 35);
13   }
14 }
```

Результат работы Листинга 22 представлен на Рисунке 25.

В 5-й строке Листинга 22 мы вызываем метод *noStroke()* для того, чтобы выключить обводку. В этом примере мы рисуем прямоугольники без нее.

Логика отрисовки десяти прямоугольников проста. В цикле *for* мы объявляем переменную i, присваиваем ей значение 1, определяем условие выхода из цикла $i < 10$ и шаг $i++$ ($i++$ – это унарный оператор, который в нашем случае означает то же самое, что и оператор $i = i+1$, но пишется короче, поэтому его и используют). В теле цикла мы задаем цвет

Рис. 25: Результат работы Листинга 22.
Прямоугольники, отрисованные в цикле, без обводки.

для заливки и отрисовывам прямоугольник в зависимости от значения переменной i.

Задание 8. Измените код Листинга 22, так, чтобы прямоугольники отрисовывались на холсте в виде сетки, как на Рисунке 26

Рассмотрим пример, когда один цикл работает внутри другого цилка. Такая необходимость может возникнуть, например,если требуется построить двухмерную сетку геометрических форм с определенным шагом, как на Рисунке 27, где показан результат работы кода Листинга 23.

Листинг 23: Рисуем сетку из эллипсов

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    noLoop();
5    noStroke();
6    ellipseMode(CENTER);
7  }
8
9  void draw() {
10   background(255);
11   float border = 50;
```

```
12      float nw = width-2*border;
13      float nh = height-2*border;
14      float number = 5;
15      float nWstep = nw / number;
16      float nHstep = nh / number;
17      for (int i = 0; i < number; i++) {
18        for (int j = 0; j < number; j++) {
19          float x = border + j*nWstep + nWstep/2;
20          float y = border + i*nHstep + nHstep/2;
21          float size = 5 + (j+i)*10;
22          float mColor = size*1.5;
23          fill(mColor, 20, 50);
24          ellipse(x, y, size, size);
25          fill(250);
26          ellipse(x, y, 3, 3);
27        }
28      }
29    }
```

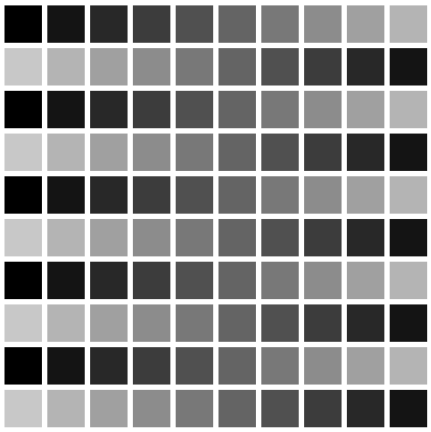


Рис. 26: Результат задания 8.
Сетка из прямоугольников разных тонов.

Начнем разбирать код Листинга 23 с метода *draw()*. В 11-й строке мы определяем переменную *border*, где будут храниться значения полей вокруг нашей будущей сетки. Например,нужно, чтобы сетка отступала на 50 пикселей от каждой стороны окна. Значит, оставшееся поле для сетки будет равно ширине окна за вычетом размера двух полей. Эти рассуждения записаны в 12-ю строку. Переменная *nw* отвечает за ширину сетки, соответственно, переменная *nh* отвечает за высоту сетки (строка 13).

Количество ячеек мы объявили в строке 14, в нашем случае их будет пять и это значение будет храниться в переменной *number*. Далее мы определяем габариты ячейки в строках 15 и 16, записываем их значения в соответствующие переменные *nWstep* и *nHstep*.

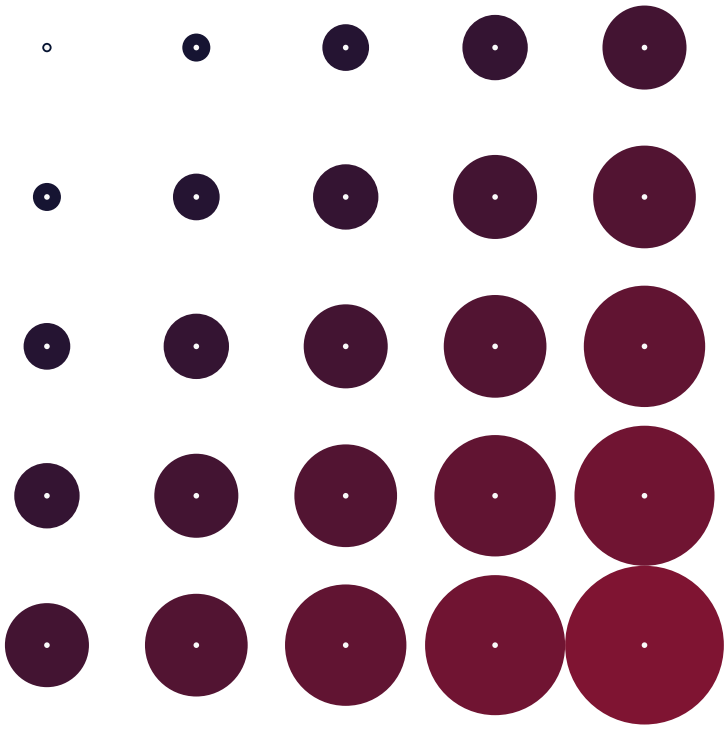


Рис. 27: Результат работы Листинга 23.
Сетка из эллипсов.

Теперь у нас все готово для того, чтобы нарисовать сетку. Как показано на Рисунке 27, нам нужно отрисовать в центрах ячеек эллипсы разного размера и цвета: очевидно, что для этого придется определить две новые переменные. Но и отрисовка эллипса требует определения его координат, значит, нужны еще две переменные.

В строке 17 мы пишем цикл for, от 0 до *number* (количество ячеек) для того, чтобы указать ряд будущих ячеек. Как только мы указали ряд, нам требуется обойти все ячейки этого ряда и отрисовать в них эллипсы.

Итак, переменная i будет отвечать за смещение по оси Y (т.е. выбор номера ряда), переменная j будет отвечать за смещение по оси X (т.е. за выбор ячейки внутри ряда). Рассмотрим определение этих переменных.

Задание 9. Измените код Листинга 23 так, чтобы эллипсы отрисовывались от большего к меньшему и были раскрашены в тона другого цвета, например синего.

Начнем определять координаты наших эллипсов. При каждой итерации внутреннего цикла у нас есть в наличии значения переменных i и j, размер полей отступа – *border* и размер ячейки. Этого материала достаточно, чтобы определить центр текущий ячейки. Так мы определяем переменные центра ячейки X и Y в строках 19 и 20. Размер эллипса будем определять в зависимости от расположения его ячейки, а именно, от переменных i и j в 22-й строке. В этой же строке будем определять цвет как переменную *mColor*.

Как только мы определили все необходимые нам переменные, мы занимаемся отрисовкой эллипса (строки 23 и 24). В нашем случае удобно воспользоваться отрисовкой не от левого верхнего угла, а от центра, поэтому мы в еще в строке 6 вызвали метод *ellipseMode()* с аргументом *CENTER*, что «включило» нам требуемый режим. Чтобы показать, что эллипсы находятся в строгой прямоугольной сетке, мы отрисовываем в центре каждого эллипса еще один поменьше и белого цвета (строки 25, 26).

В этом параграфе мы рассмотрели работу циклами с ритмом. Мы использовали цикл *for*, однако в Processing существуют и другие возможности для работы с циклами: *while* и *for each*. О них мы будем говорить далее, но не так подробно. Информацию о них вы можете найти на официальном сайте https://processing.org/reference/ в разделе *Iteration*. Там же, в разделе *Conditionals*, дано описание возможности остановки работы цикла и пропуска текущей итерации.

5.3 Движение и анимация

Описывая процесс рисования на языке Processing, мы подразумеваем, что при каждом запуске программы (скетча) наш холст является пустым, как, впрочем, пуст холст и перед классическим художником, готовящимся к созданию своего произведения. Однако если с помощью карандашей и красок художник создает свой шедевр на одном холсте один

раз, то цифровой холст используется совсем иначе: при каждом запуске программы картина создается заново, даже если в код не были внесены никакие изменения. Таким образом, у цифрового художника появляется выбор: он может остаться в привычных рамках статического произведения, сохранив и распечатав созданную им на холсте картину, а может использовать возможности многократного обновления на цифровом холсте созданного им произведения. разработали «робота художника», который каждый раз, когда его запускают, рисует одну и туже картину по вашим правилам. Эти правила вы и пишите в виде исходного кода.

В том случае, если ваша цель – создание статического графического произведения, то вам потребуется просто сохранить результат в файл. Вы можете вызвать метод *saveFrame("myArt.png");*, в аргументе которого в двойных кавычках указывается путь и имя файла для записи картинки с холста приложения. Также обязательно нужно указать расширение файла, например, TIFF (.tif), JPEG (.jpg), или PNG (.png). Вызов метода можно расположить, как в Листинге 20, после 15-й строки.

Однако если вы хотите создать интерактивное произведение, которое, например, позволит зрителям принимать участие в том, что появляется на холсте, вам не обойтись без динамики и анимации.

Рассмотрим код Листинга 24. Он очень прост и напоминает наш самый первый скетч за исключением нескольких моментов. Первое отличие – это строка 6. В ней комментарием дан вызов метода *noLoop()*: две наклонных черты (два слеша) служат специальным символом, делая строку невидимой для программы, но не для программиста. Поставив два слеша, мы сделали строку 6 пустой для программы, но если возникнет необходимость снова вызвать метод *noLoop()*, то нужно будет просто удалить эти два слеша.

Листинг 24: Рисуем крестик и печатаем в консоль

```
1  void setup () {
2    size(300, 300);
3    smooth();
4    strokeWeight(30);
5    stroke(100);
6    //noLoop();
7  }
8
9  void draw () {
10   background(0);
11   line(100,100, 200, 200);
12   line(200,100, 100, 200);
13   println(frameCount);
14 }
```

Второе отличие кода Листинга 24 – строка 13. В ней мы вызываем метод *println()* (от английского *print line*), и передаем ему аргументом переменную *frameCount*. Переменная *frameCount* объявлена не нами, а разработчиками Processing. Но по сути своей это такая же переменная, как и объявленные нами переменные i и k.

Метод *println()* выводит свои аргументы в черное поле внизу скетча в редакторе Processing (см. Рисунок 28): в это поле, которое называют консолью, выводятся цифры – номера фреймов. Но прежде чем начать разбираться с этими цифрами, давайте изменим код и посмотрим на Листинг 25.

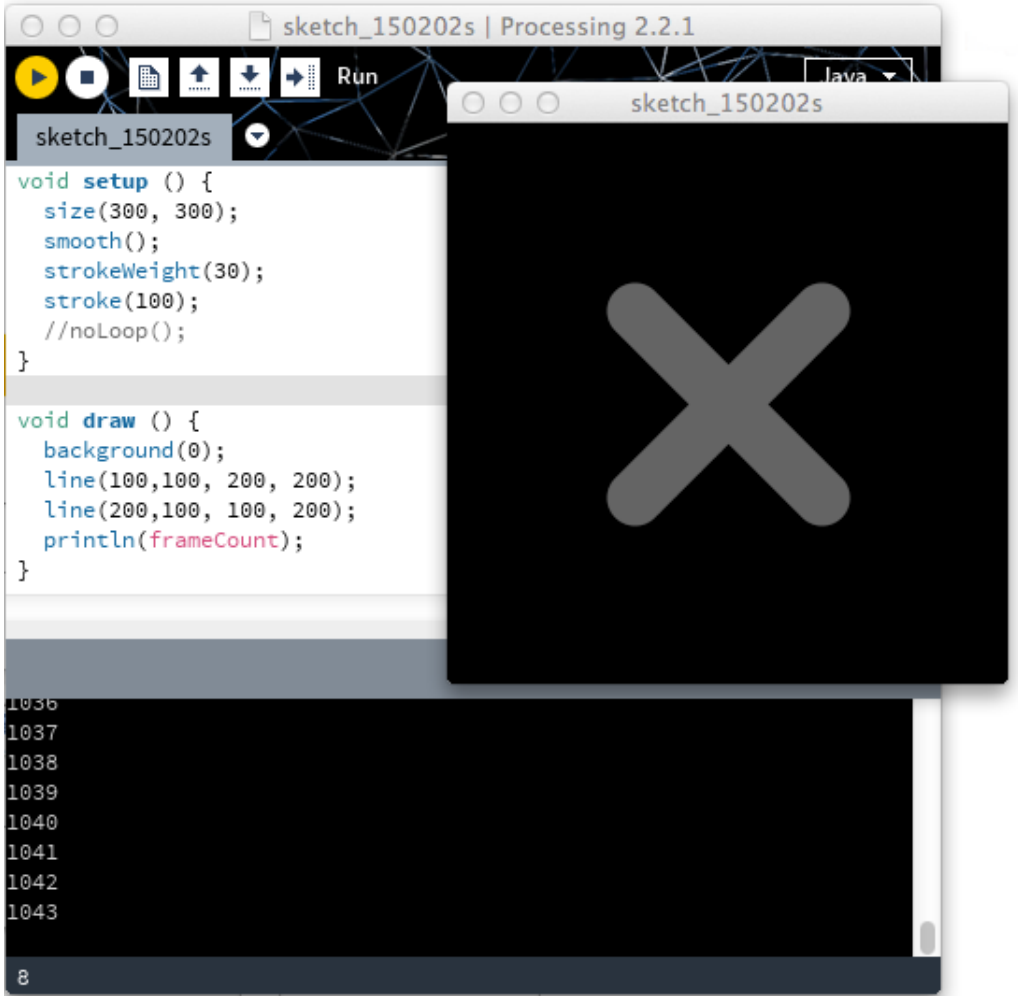


Рис. 28: Результат работы Листинга 24. Снимок экрана.

Запустив код Листинга 25, вы увидите, что крестик начнет двигаться.

Это движение происходит по двум причинам. Во-первых, мы используем переменную, которая постоянно растет на 1. Во-вторых, вызов метода *draw()* происходит не один раз, а в цикле. Более того, метод *draw()* вызывается бесконечно, пока вы не выключите программу или компьютер. При этом мы не видим, в какой именно строке и в каком файле происходит вызов метода *draw()*, полагаясь только на разработчиков Processing. Каждый раз, когда вызывается метод *draw()*, крестик отрисовываетсяался с координатами при значении переменной *frameCount = 1*. Эта переменная и содержит номер текущего кадра, который отрисовывается на экране. И далее, как в классической мультипликации, мы рисуем следующий кадр. В следующем кадре мы заливаем холст черным в строке 10 и заново рисуем крестик при *frameCount = 2*.

Листинг 25: Анимируем крестик

```
1  void setup () {
2    size(300, 300);
3    smooth();
4    strokeWeight(30);
5    stroke(100);
6    //noLoop();
7  }
8
9  void draw () {
10   background(0);
11   line(frameCount,100, 100+frameCount, 200);
12   line(100+frameCount,100, frameCount, 200);
13   //println(frameCount);
14 }
```

Важно отметить, что при таком покадровом подходе к анимации, каждый следующий кадр «ничего не знает» о предыдущем. Когда мы с вами смотрим на анимацию крестика, то мы видим ОДИН крестик, который движется по экрану, а на самом деле этого ОДНОГО крестика нет. Каждый раз отрисовка происходит заново, программа перерисовывает экран новой картинкой. Чтобы продемонстрировать это, давайте рассмотрим код Листинга 26.

Листинг 26: Анимириуем крестик без заливки фона

```
1  void setup () {
2    size(300, 300);
3    smooth();
4    strokeWeight(30);
5    background(0);
6  }
7
```

```
8  void draw () {
9    stroke(frameCount);
10   line(frameCount,100, 100+frameCount, 200);
11   line(100+frameCount,100, frameCount, 200);
12 }
```

Код Листинга 26 отличается от Листинга 25 тем, что метод *background(0)* вызывается один раз в методе *setup()*, а вот метод *stroke()*, напротив, вызывается методом *draw()* так же часто, как и сам метод *draw()*. Аргументом а метода *stroke()* мы передаем *frameCount*. Таким образом с каждым новым кадром мы меняем цвет линии и не заливаем наш холст черной краской после предыдущего кадра. Мы как бы оставляем предыдущий мультипликационный кадр и рисуем прямо на нем (см. Рисунок 29).

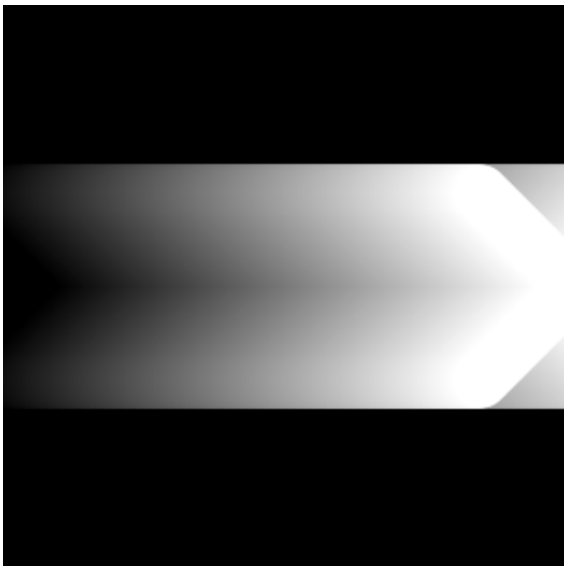


Рис. 29: Результат работы Листинга 26. Крестик и след его движения по горизонтали.

Задание 10. Измените 10-ю и 11-ю строки кода Листинга 26 так, чтобы на холсте было отрисовано изображение, как на Рисунке 30. Крестик должен двигаться по диагонали от левого верхнего угла в правый нижний угол.

При помощи прямоугольника (а на самом деле при помощи любой заливаемой цветом фигуры) можно добиться интересных художественных

Рис. 30: Результат выполнения задания 10. Крестик и след его движения по диагонали.

эффектов, как это продемонстрировано в коде Листинга 27.

Листинг 27: Движущийся шарик

```
1
2  void setup() {
3    size(500, 500);
4    smooth();
5  }
6
7  float counter;
8
9  void draw() {
10   noStroke();
11   fill(10, 50);
12   rect(-1,-1,width+1,height+1);
13
14   float ny = sin(counter)*100+200;
15   float nx = counter*10;
16
17   stroke(250);
18   strokeWeight(20);
19   line(nx, ny, nx, ny);
20
21   counter = counter + 0.1;
22
23   if(nx > width){
```

```
24        counter = 0;
25      }
26    }
27
28    void keyPressed() {
29      if (key=='s') saveFrame("myProcessing.png");
30    }
```

Результат выполнения кода Листинга 27 представлен на Рисунке 31. Мы отрисовываем отрезок «нулевой длины»: координаты его начальной и конечной точек равны. Координата по оси Y изменяется по закону синуса: в 14-й строке мы вызываем метод *sin()* и передаем ему переменную *counter*, которая увеличивается с шагом 0.1 в 21-й строке. Координата по оси X изменяется линейно, в 15-й строке ей дается приращение в виде текущего значения переменной *counter*, умноженного на 10. Как только значение этой координаты становится больше ширины холста, мы обнуляем счетчик (см. строки 23 и 24). Логика изменения координаты Y более сложная: мы меняем координату Y по синусоидному закону. Метод *sin()* возвращает значение синуса, принятого в радианах аргумента (строка 14). Более детально работа с тригонометрическими функциями будет рассмотрена в отдельной главе.

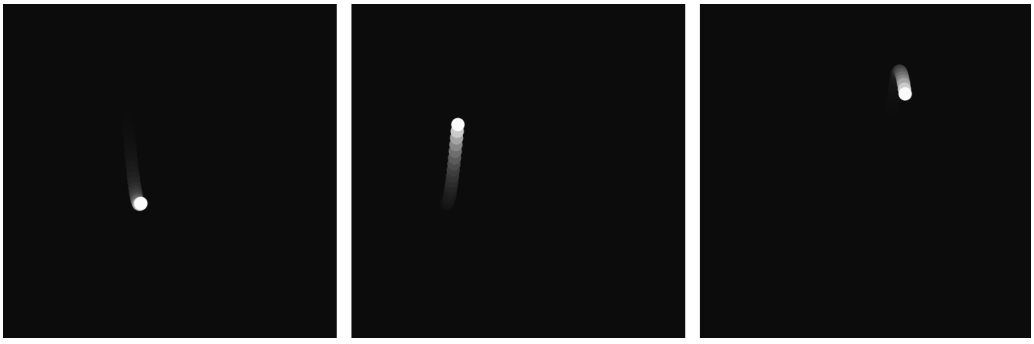


Рис. 31: Результат выполнения Листинга 27. Фазы движения шарика.

Логика движения нашего нулевого отрезка довольно проста. Интерес, как нам кажется, представляет художественный эффект в виде шлейфа. Шлейф тянется за объектом, повторяя его движение и растворяясь в черном фоне холста. Такого эффекта мы добились, используя отрисовку прямоугольника в 12-й строке. Прямоугольник мы отрисовываем с заливкой *fill(10,50)* – черного цвета и с прозрачностью.

Задание 11. Измените код Листинга 27 так, чтобы прямоугольник отрисовывался на холсте не во всю ширину, а только на половину ширины холста, как на Рисунке 32. Тогда вы увидите шлейф на левой части холста, а на правой будет оставаться полный след от движения объекта.

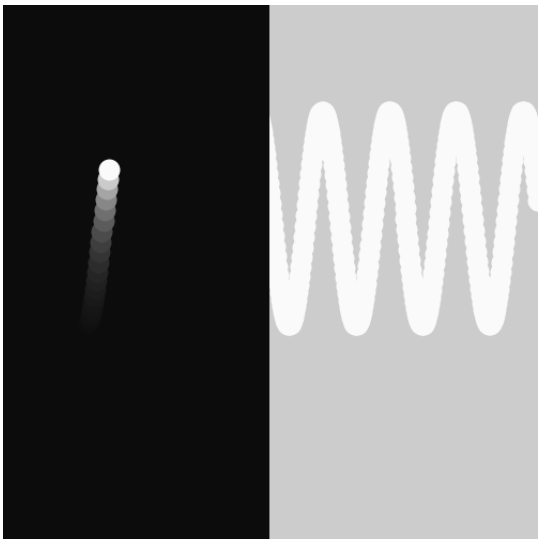


Рис. 32: Результат выполнения задания 11. Два типа шлейфа от движения шарика.

Таким образом, каждый кадр нашей анимационной последовательности мы как бы накрываем полупрозрачным черным прямоугольником, из-за чего белые объекты становятся чуть темнее, серые – еще более темными, а почти черные объекты сливаются с фоном. Таким образом мы наблюдаем эффект «исчезновения» объектов, их плавного растворения в черном фоне холста.

В этом параграфе мы рассмотрели базовые принципы работы анимации в Processing. Создание динамических композиций, возможность «оживить» статические полотна является отличительной особенностью искусства нового времени – наверное,так же рассуждали и создатели кинематографа в конце XIX века. Однако программируемое цифровое искусство не стоит на месте. Далее мы рассмотрим возможность влияния на динамику произведения не только со стороны его автора, но и со стороны его зрителей.

Но перед этим в следующем параграфе ненадолго вернемся к работе с цветом, чтобы разобрать еще один очень важный для художника аспект – цветовую модель HSB

5.4 Цвет в формате HSB

Для работы с цветом, кроме формата RGB, существует и другой, более понятный формат – цветовая модель HSB, в которой цвет задается следующим образом: (покажем на примере темно-красного цвета) цвет = красный, насыщенность = 50%, яркость = 10%. Рассмотрим Листинг 28.

Листинг 28: Рисуем цветную сетку HSB

```
1  int stepX;
2  int stepY;
3
4  void setup(){
5    size(500, 500);
6    background(0);
7  }
8
9  void draw(){
10   colorMode(HSB, width, height, 100);
11
12   stepX = mouseX+2;
13   stepY = mouseY+2;
14   for (int gridY=0; gridY<height; gridY+=stepY){
15     for (int gridX=0; gridX<width; gridX+=stepX){
16       stroke(gridX, height-gridY, 100);
17       strokeWeight(stepX);
18       line(gridX, gridY, stepX+gridX, stepY+gridY);
19     }
20   }
21 }
```

В этом примере мы немного забежали вперед и использовали две переменных: *mouseX* и *mouseY*. В них Processing хранит координаты курсора мыши. Работа с ними позволяет нам создавать интерактивные приложения, но об этом мы поговорим в следующей главе.

В 10-й строке мы вызываем метод *colorMode()*, первым аргументом которого передаем константу HSB. Спецификация Processing утверждает, что первый аргумент может быть или RGB, или HSB. В первом случае это соответствует цветовой модели Red, Green, Blue. Во втором случае – модели Hue, Saturation, Brightness: ветовой тон, Насыщенность и Яркость – это три параметра, с помощью которых можно управлять значением цвета. По умолчанию включен режим RGB. Если вы меняете

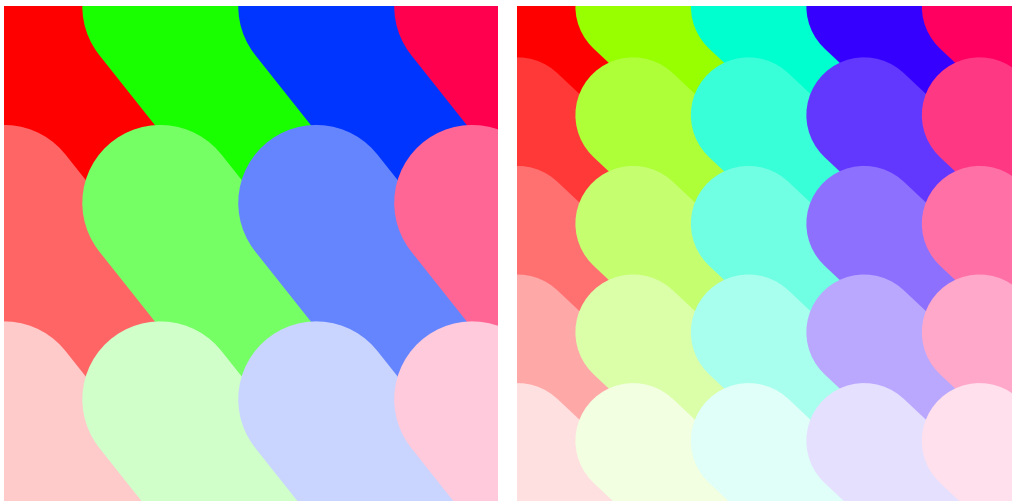


Рис. 33: Результат работы Листинга 28. Две фазы работы приложения с цветной сеткой HSB.

режим, то во всех методах значение аргументов переключатся с RGB на HSB.

Четвертым параметром при указании цвета в цифровом искусстве является прозрачность. Без него мы не могли бы в полной мере воплощать художественные амбиции. Безусловно его использование, например, при отрисовке линий, подразумевает математический пересчет цвета холста под линией и получение нового цвета. Однако все эти операции Processing берет на себя, не обременяя этим художника.

Значения прозрачности, так же как и значения RGB и HSB, изменяются в Processing от 0 до 255. Значение 0 означает полностью прозрачный, а значение 255 – полностью непрозрачный. Рассмотрим Листинг 29.

Листинг 29: Диагональное движение полупрозрачных линий

```
1  void setup() {
2    background(255);
3    size(500, 500);
4    smooth();
5  }
6
7  int l1x1 = 0;
8  int l1y1 = 0;
9  int l1x2 = 500;
10 int l1y2 = 500;
11 int flug = 1;
12
```

```
13  void draw() {
14    background(255);
15    strokeWeight(50);
16    stroke(10, 150, 100, 20);
17    line(l1x1, l1y1, l1x2, l1y2);
18    for(int i = 1; i < 20; i = i + 1){
19      int k = i*50;
20      stroke(10, 150, 100, 20+i*10);
21      line(l1x1 + k, l1y1, l1x2, l1y2 - k);
22      line(l1x1, l1y1 + k, l1x2 - k, l1y2);
23    }
24
25    l1x1 = l1x1 + flug;
26    l1y1 = l1y1 + flug;
27    l1x2 = l1x2 - flug;
28    l1y2 = l1y2 - flug;
29    if(l1y2 < 0 || l1y2 > 500){
30      flug = flug*(-1);
31    }
32  }
```

Работа кода Листинга 29 использует также принципы ритма и анимации. Мы отрисовываем линии в цикле *for* с 18-й по 23-ю строки. Расчет координат концов отрезков происходит в строках с 25-й по 28-ю с использованием дополнительной переменной *flug*. Эта переменная выполняет роль не только шага приращения координаты, но и переключателя. В 29-й строке мы проверяем два условия с помощью логического оператора || – логическое «ИЛИ». Это означает следующее: мы попадем в строку 30 в том случае, если будет истинно хотя бы одно из условий: первое из них – если мы уйдем «в минус»; второе – если мы уйдем за пределы экрана Существует еще один часто используемый логический оператор &&, который обозначает логическое «И». Подробнее о логических операторах можно узнать по адресу https://processing.org/reference/ раздел Logical Operators. Результат работы кода Листинга 29 показан на Рисунке 34.

Работа с прозрачностью происходит в строке 20 цикла *for*. В зависимости от значения переменной i мы устанавливает значение прозрачности для обводки. Далее, в строках 21 и 22 мы используем обводку этого цвета для отрисовки двух отрезков.

Задание 12. Измените код Листинга 29 так, чтобы линии на холсте были разных цветови по достижении центральной линией минимального размера происходила смена цвета композиции, например, как на Рисунке 35.

Рис. 34: Результат работы Листинга 29. Диагональное движение полупрозрачных линий.

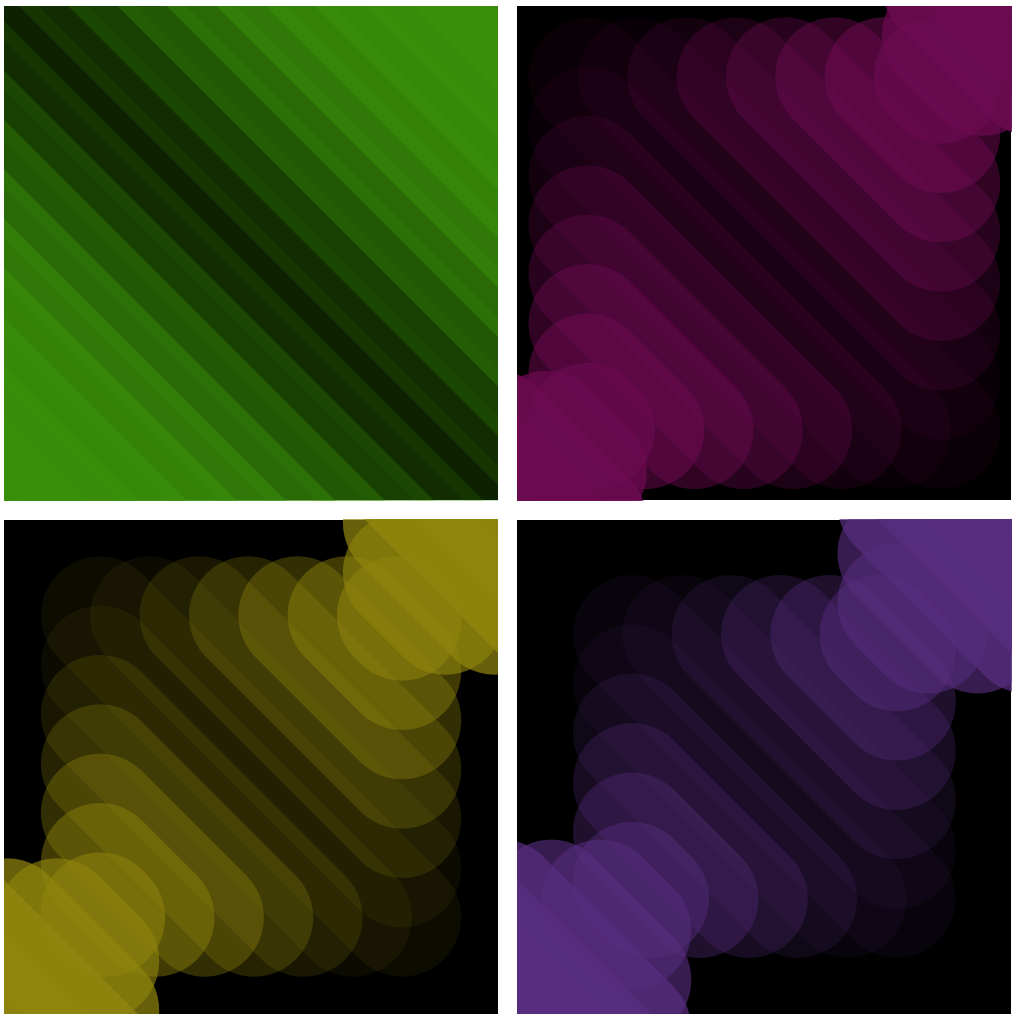


Рис. 35: Результат выполнения задания 12. Диагональные полупрозрачные линии разных цветов.

В качестве подсказки для выполнения задания 12 мы приводим Листинг 30. Подробно рассматривать этот пример мы не будем, отметим только суть его работы: она заключается в том, что смена цвета происходит при каждом «схлопывании» (сжатии до минимума) отрезка на холсте (см. Рисунок 35).

Листинг 30: Диагональные полупрозрачные линии меняющие цвет

```
1  void setup() {
2    background(255);
3    size(500, 500);
4    smooth();
```

```
5  }
6
7  int l1x1 = 0;
8  int l1y1 = 0;
9  int l1x2 = 500;
10 int l1y2 = 500;
11 int flug = 1;
12
13 float mr = 10;
14 float mg = 150;
15 float mb = 100;
16
17 void draw() {
18   background(0);
19   strokeWeight(120);
20   stroke(mr, mg, mb, 25);
21   line(l1x1, l1y1, l1x2, l1y2);
22   for(int i = 1; i < 11; i = i + 1){
23     int k = i*50;
24     stroke(mr, mg, mb, (255/11)*i);
25     line(l1x1 + k, l1y1, l1x2, l1y2 - k);
26     line(l1x1, l1y1 + k, l1x2 - k, l1y2);
27     if(l1x1 == l1x2 || (l1x1 + k) == l1x2 || l1x1 == (l1x2
          - k)){
28       mr = random(0,150);
29       mg = random(0,150);
30       mb = random(0,150);
31     }
32   }
33
34   l1x1 = l1x1 + flug;
35   l1y1 = l1y1 + flug;
36   l1x2 = l1x2 - flug;
37   l1y2 = l1y2 - flug;
38   if(l1y2 < 0 || l1y2 > 500){
39     flug = flug*(-1);
40   }
41 }
```

В этом параграфе мы рассмотрели возможность работы с интуитивно понятной цветовой моделью HSB, а также примеры динамической смены цвета в произведении. В одном из примеров мы реализовали возможность управления мышью. Остановимся на этом подробнее в следующей главе.

6 Интерактивное взаимодействие

Современные произведения искусства вышли за рамки не только статических, но и динамических художественных форм. Возможность создавать покадровую анимацию не была бы полноценной, если бы мы не могли влиять на нее в режиме реального времени – здесь и сейчас: современное произведение искусства может взаимодействовать со зрителем в прямом смысле этого слова.

Интерактивность в цифровом искусстве как понятие, которое раскрывает характер и степень взаимодействия между объектами, подразумевает взаимодействие зрителя с произведением художника. Это взаимодействие может быть настолько тесным, что зритель получает возможность стать соавтором или даже частью произведения. Безусловно, использование таких возможностей выгодно отличают работы цифровых художников.

В этой главе мы рассмотрим интерактивное взаимодействие, которое реализует Processing в своей базовой комплектации. Наиболее привычное взаимодействие – это управление приложением мышью. Однако представьте, что движения курсора могут быть связаны не обязательно с мышью, а, например, с перемещением зрителя у вашей картины.

6.1 Базовое взаимодействие

Начнем работу с примеров простого взаимодействия – это работа с мышью и клавиатурой. Далее можно усложнять варианты человеко-компьютерного взаимодействия, например, управлять приложением движением руки (через контроллер Kinect), кисти (через контроллер Leap Motion) и даже глазами (взор-содержащие интерфейсы).

Самое простое, что мы можем сделать – это заставить наш крестик из предыдущих глав двигаться не по заданной траектории, а по положению курсора мыши. Обратимся к коду Листинга 31.

Листинг 31: Двигаем крестик по экрану мышью

```
1  void setup () {
2    size(300, 300);
3    smooth();
4    strokeWeight(30);
5    background(0);
6  }
7
8  void draw () {
9    stroke(200, 20);
```

```
10      line(mouseX-50,mouseY-50, 100+mouseX-50, 100+mouseY-50);
11      line(100+mouseX-50,mouseY-50, mouseX-50, 100+mouseY-50);
12  }
```

В Листинге 31 в 10-й и 11-й строках мы используем две переменные $mouseX$ и $mouseY$. Эти переменные также встроены в Processing как и $frameCount$. Значения, хранимые в этих переменных, как вы догадались, – это координаты курсора мыши. Результат работы кода приведен на Рисунке 36.

Рис. 36: Результат выполнения Листинга 31.
Перемещение крестика по экрану мышью.

Для того чтобы курсор мыши был расположен посередине крестика необходимо в 10-й и 11-й строках сместить крестик по диагонали влево в верхний угол на 50 пикселей (мы специально оставили -50 и не стали производить вычитание для наглядности. Конечно, можно написать вместо $100 + mouseX - 50$ строку $50 + mouseX$).

Принцип работы приложения остался таким же. Метод *draw()* вызывается системой так быстро, как позволяют ресурсы. И каждый раз при его вызове переменные $mouseX$ и $mouseY$ получают новые значения положения курса мыши. Если вы двигаете мышью, то значения будут меняться. Если вы мышью не двигаете, то крестик стоит на месте.

Добавим художественного эффекта в наше произведение. Взгляните на код Листинга 32. В нем мы вводим две переменные i и k между методами $setup()$ и $draw()$. Это требуется для двух вещей.

Во-первых, если мы объявим переменную в каком-нибудь методе, например, в *setup()*, то эта переменная будет видна только внутри этого метода, так как область видимости переменной ограничивается фигурными скобками метода.

Во-вторых, если переменную объявить вне метода, вне области видимости фигурных скобок, то она будет видна методу, например, *draw()*. Причем, эта переменная будет инициализирована один раз, т.е. строки 8 и 9 будут выполнены программой один раз. И далее, метод *draw()* будет выполняться бесконечное число раз, не вызывая строки 8 и 9, что позволяет нам хранить значение переменной между вызовами метода *draw()*.

Листинг 32: Управление крестиком с динамическим цветом

```
1  void setup () {
2    size(500, 500);
3    smooth();
4    strokeWeight(30);
5    background(0);
6  }
7  int i = 0;
8  int k = 1;
9  void draw () {
10   stroke(i, 20);
11   line(mouseX-50,mouseY-50, 100+mouseX-50, 100+mouseY-50);
12   line(100+mouseX-50,mouseY-50, mouseX-50, 100+mouseY-50);
13   i = i + k;
14   if(i == 255){
15     k=-1;
16   }
17   if(i == 0){
18     k=1;
19   }
20 }
```

Результат выполнения кода Листинга 32 показан на Рисунке 37. Строка 15 работает так же, как и в циклах, увеличивая переменную i на 1 с каждой итерацией(с каждым вызовом метода *draw()*).

Далее, в 16-й строке логический оператор, такой же как в условии выхода из цикла, оператор проверяет условие в круглых скобках. В нашем случае $i == 255$ можно понимать так: «не равно ли i 255?». Логический оператор возвращает или истину (*true*), или ложь (*false*). Если мы получили истину, то отправляемся в фигурные скобки, а именно, в строку 16. После этого программа следует дальше, с 18-й строки. Если получили ложь, то в фигурные скобки не попадаем и продолжаем сразу с 18-й строки. Обратите внимание на двойной знак равенства. Если бы мы поставили одиночный знак равенства, т.е. $i = 255$, то всегда срабатыва-

ла бы операция присвоения и далее, логический оператор возвращал бы всегда *true*.

Так как переменная i у нас используется для указания цвета в строке 10, и ее значение постоянно меняется, то цвет мы видим как крестик плавно меняет свой цвет от светлого к темному и обратно.

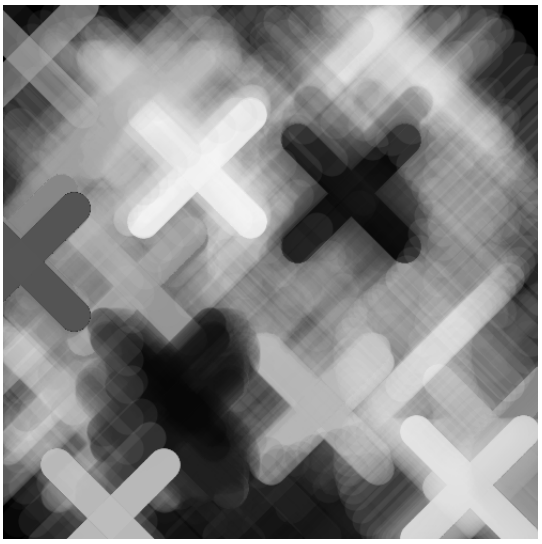


Рис. 37: Результат выполнения Листинга 32.
Управление крестиком с динамическим цветом.

Линии и отрезки как художественный объект могут раскрыть себя именно в интерактивных приложениях. Такие приложения, как из Листинга 32, можно рассматривать в виде новой кисти художника, которой он творит свои произведения.

Давайте создадим еще несколько таких кистей и сохраним наши произведения в файл. Рассмотрим Листинг 33.

В 4-й строке мы уменьшили толщину линии, а в 13-й строке мы рисуем линию: первая точка отрезка – от курсора мыши, вторая – имеет координату по оси X как результат выполнения метода *random(0,500)*. Метод *random(0,500)* с двумя атрибутами при каждом вызове, возвращает случайное число. Аргументы указывают максимальное и минимальное значение этого числа. В нашем случае координата X второй точки отрезка будет с каждым кадром разная: от 0 до 500. Координата Y остается со значением 500.

С 23-й по 27-ю строку мы объявляем метод *keyPressed()*. Этот метод работает по такой же логике, как и методы *setup()* и *draw()*: от нас требуется только реализовать их, а вызываются они без нашего вмешательства

в код. Но не в работу программы! Так, если мы нажмем на клавиатуре букву *s*, то, как можно догадаться, программа попадет в строку 25. Это происходит потому, что переменная *key* в 24-й строке также является встроенной переменной Processing. В нее записывается значение – символ, который нажат на клавиатуре.

Листинг 33: Рисование линий с вершиной в месте курсора

```
1  void setup () {
2    size(500, 500);
3    smooth();
4    strokeWeight(1);
5    background(0);
6  }
7
8  int i = 0;
9  int k = 1;
10
11 void draw () {
12   stroke(i, 20);
13   line(mouseX, mouseY, random(0,500), 500);
14   i = i + k;
15   if(i == 255){
16     k=-1;
17   }
18   if(i == 0){
19     k=1;
20   }
21 }
22
23 void keyPressed() {
24    if (key=='s') {
25        saveFrame("myProcessing.png");
26    }
27 }
```

Задание 1. Измените код Листинга 2 так, чтобы на холсте было отрисовано изображение, как на Рисунке 39 и Рисунке 40. Требуется, чтобы в первом случае «привязанная» вершина линии находилась не внизу, а вверху; во втором – чтобы линии были привязаны не к одной какой-то стороне, а ко всем четырем сторонам.

В 25-й строке выполнится метод *saveFrame("myProcessing.png");* и после него программа снова попадет в метод *draw()*. Метод

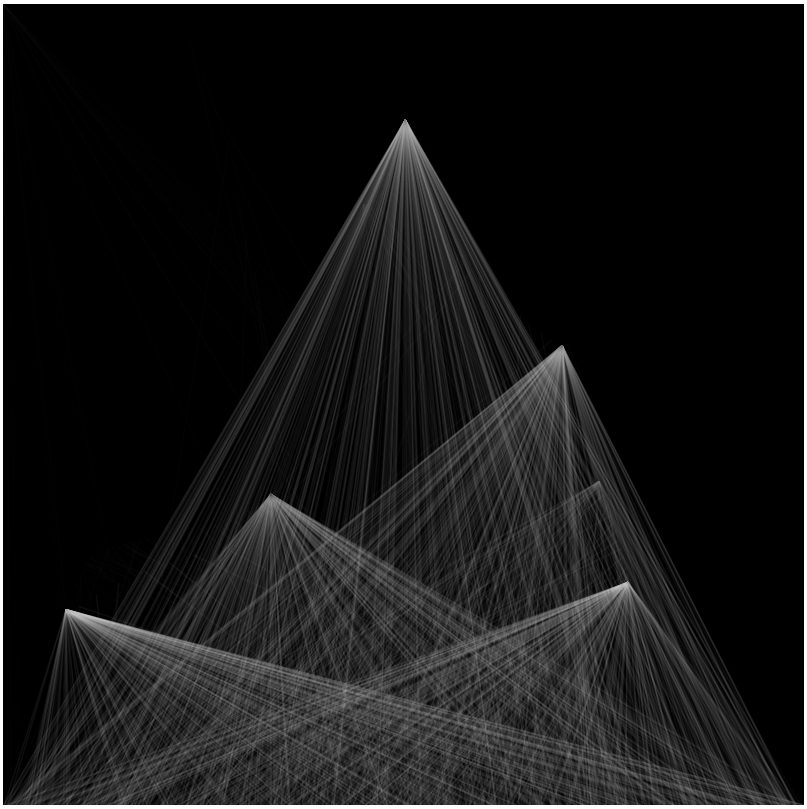


Рис. 38: Результат выполнения Листинга 33. Рисование линий с вершиной в точке курсора.

saveFrame("myProcessing.png"); запишет вашу картину в файл. И произойдет это достаточно быстро: вы, возможно, даже не заметите, как ваша интерактивная программа замрет для этой записи.

Теперь если нужно открыть папку с вашим *скетчем*, это можно сделать в Processing IDE в меню *Sketch->Show Sketch Folder*. Если вы и так помните, где сохраняете свои скетчи, то, естественно, этим меню можете не пользоваться.

Клавиатуру также можно использовать в интерактивном взаимодействии. Посмотрим на Листинг 34.

В 10-й строке мы объявили еще одну переменную и присвоили ей значение: $int flug = 1;$. В 31-й строке мы меняем значение этой переменной, причем если мы первый раз попали в строку 31, то значение переменной *flug* стало -1, если попали второй раз, то значение переменной *flug* стало 1 и так далее. Такое поведение переменной часто называют тригером или переключателем.

Листинг 34: Рисуем линии и управляем с помощью клавиатуры

```
1  void setup(){
2    size(500, 500);
3    smooth();
4    background(0);
5    strokeWeight(1);
6  }
7
8  int i = 0;
9  int k = 1;
10 int flug = 1;
11
12 void draw () {
13   stroke(i, 20);
14   if(flug == 1){
15     line(mouseX, mouseY, 500, random(0,500));
16   } else {
17     line(mouseX, mouseY, 0, random(0,500));
18   }
19
20   i = i + k;
21   if(i == 255){
22     k=-1;
23   }
24   if(i == 0){
25     k=1;
26   }
27 }
28
29 void keyPressed() {
30   if (key=='q'){
31     flug = flug*(-1);
32   }
33    if (key=='s') {
34     saveFrame("myProcessing.png");
35    }
36 }
```

Этот тригер (см. Листинг 34) мы используем в методе *draw()*: если тригер в одном положении, то работает строка 15, если тригер в другом положении, то работает строка 17.

Логический оператор *if* превратился в более полную конструкцию $if(...)\{...\}else\{...\}$, смысл которой предельно ясен: если условие в круглых скобках возвращает *true*, то программа попадает в первые фигурные скобки; если условие в круглых скобках возвращает *false*, то программа попадает во вторые фигурные скобки. После того как программа отработала или в первых, или во вторых фигурных скобках, работа продол-

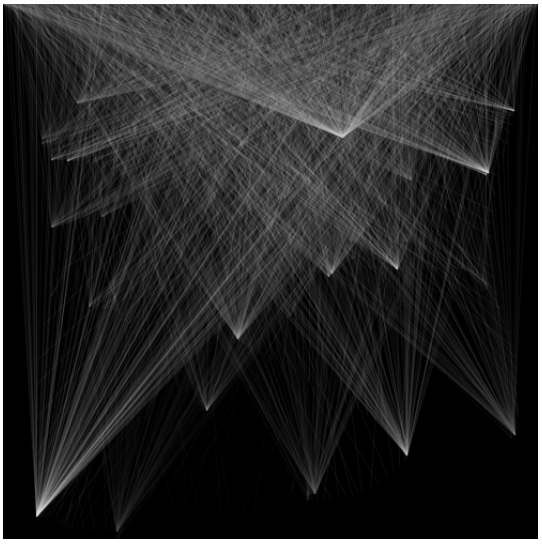


Рис. 39: Результат выполнения задания 1.
Линии отрисовываются с низу вверх.

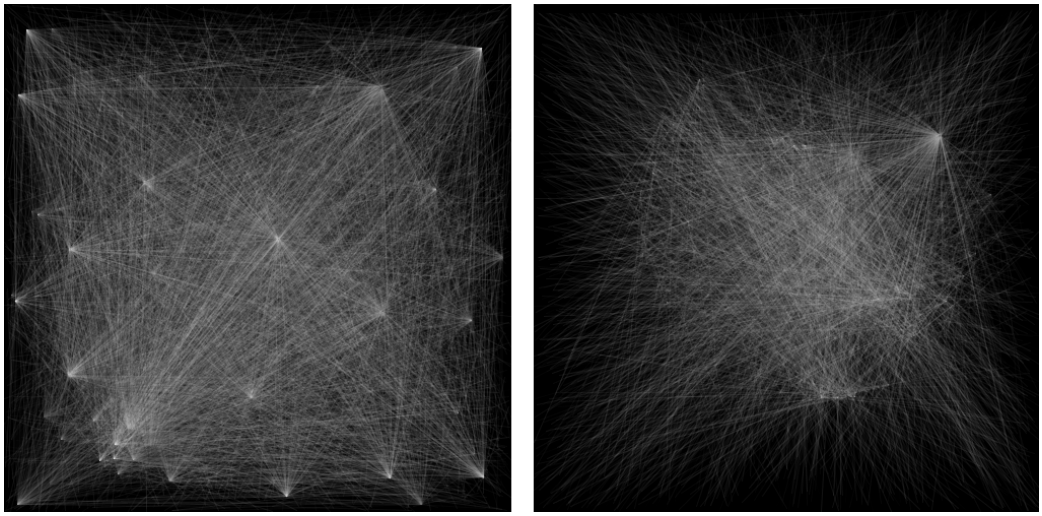


Рис. 40: Результат выполнения задания 1.
Линии отрисовываются с разных сторон окна приложения.

жается дальше со следующей строки, в нашем примере со строки 19.

Таким образом мы можем влиять на запуск 15-й и 17-й строк, и тем самым влиять на наше художественное решение – пример на Рисунке 41.

Метод *random()* помогает добиться большей реалистичности, избавиться от «компьютерных линий», так как вносит элемент случайности. Известно, что в природе нельзя найти два абсолютно одинаковых объ-

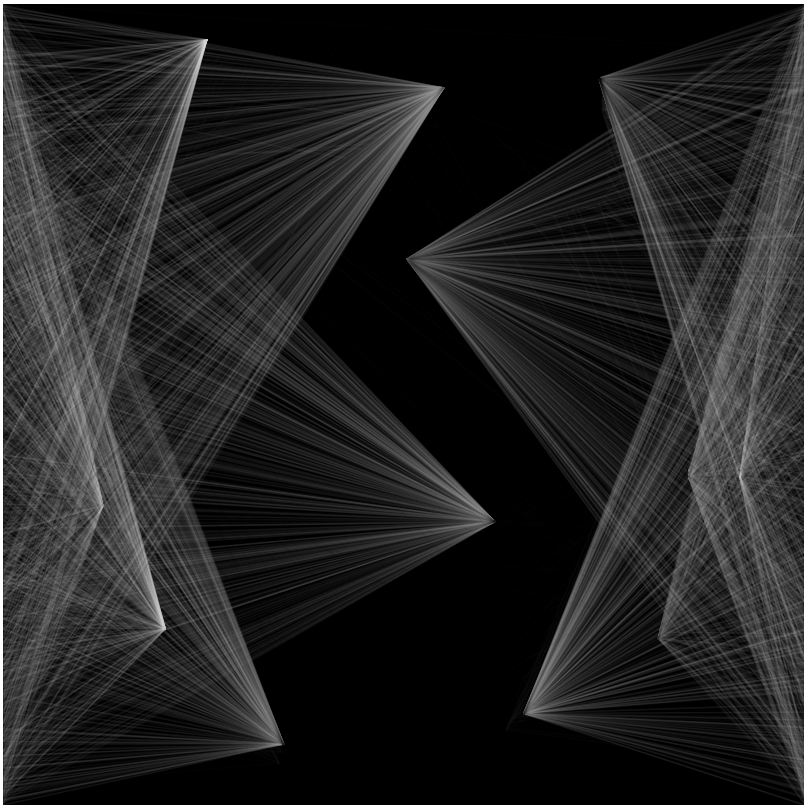


Рис. 41: Результат выполнения Листинга 34.
Рисуем мышью линии и управляем с клавиатуры.

екта: даже очень похожие деревья, облака будут чем-то отличаться. А в компьютерной графике с помощью копирования можно легко умножить количество абсолютно одинаковых объектов. Это кажется удобным, однако, как показывает практика, человеческий глаз легко замечает это, из-за чего произведение художника может подвергаться критике.

Прежде чем рассмотреть еще один пример использования метода *random()*, предлагаю взглянуть на код Листинга 34. Вы, наверняка, обратили внимание, что объем кода постепенно увеличивается: строки с 13-й по 18-ю отвечают за отрисовку, строки с 20-й по 26-ю – за изменение значений дополнительных переменных. И все это написано в одном месте, что может создать путаницу.

Предлагаю улучшить код без изменения результата программы. Такой процесс часто называется рефакторингом. Рассмотрим код Листинга 35. С 12-й по 20-ю строки мы объявили новый метод *upDate()*: название придумано нами, поэтому Processing не будет его выполнять самостоятельно – мы сами должны вызывать это метод. Больше в коде ничего не

поменялось, а метод *draw()* стал гораздо «чище».

Листинг 35: Улучшаем читабельность кода

```
void setup(){
  size(500, 500);
  smooth();
  background(0);
  strokeWeight(1);
}

int i = 0;
int k = 1;
int flug = 1;

void upDate(){
  i = i + k;
  if(i == 255){
    k=-1;
  }
  if(i == 0){
    k=1;
  }
}

void draw () {
  stroke(i, 20);
  if(flug == 1){
    line(mouseX, mouseY, random(0,500), 0);
  } else {
    line(mouseX, mouseY, random(0,500), random(0,500));
  }
  upDate();
}

void keyPressed() {
  if (key=='q'){
    flug = flug*(-1);
  }
   if (key=='s'){
    saveFrame("myProcessing.png");
  }
}
```

Рассмотрим еще один пример использования метода *random()* в Листинге 36, результат выполнения которого представлен на Рисунке 42 слева.

Листинг 36: Рисуем линии в горизонтальной плоскости

```
1  void setup(){
2    size(500, 500);
3    smooth();
4    background(0);
5    strokeWeight(1);
6  }
7  int i = 0;
8  int k = 1;
9  int flug = 1;
10
11 void upDate(){
12   i = i + k;
13   if(i == 255){
14     k=-1;
15   }
16   if(i == 0){
17     k=1;
18   }
19 }
20
21 void draw () {
22   stroke(i, 20);
23   float myRandom = random(-20,20);
24   float myY1 = mouseY - myRandom;
25   float myY2 = mouseY + myRandom;
26   line(0, myY1, 500, myY2);
27   upDate();
28 }
```

В Листинге 36 новой является строка 24, где мы объявляем переменную *myRandom* и присваиваемой ей значение, которое возвращает метод *random()*. Переменную *myRandom* мы объявляем как *float*. Float – это тип данных, который описывает числа с десятичной запятой, т.е. дроби.

Каждый раз, когда будет запускаться (вызываться) метод *draw()*, мы будем создавать эту переменную заново и заново присваивать ей случайное значение: от -20 до 20.

Мы это делаем, чтобы в 25-й и в 26-й строках отрисовывать линии относительно центра холста, для чего нам требуется складывать с $mouseY$ и вычитать из него одно и тоже число. Если бы это были два разных случайных числа, то результат был другим.

Задание 2. Измените код Листинга 36 в строках 25 и 26, так, что бы вместо переменной *myRandom* было два разных случайных числа.

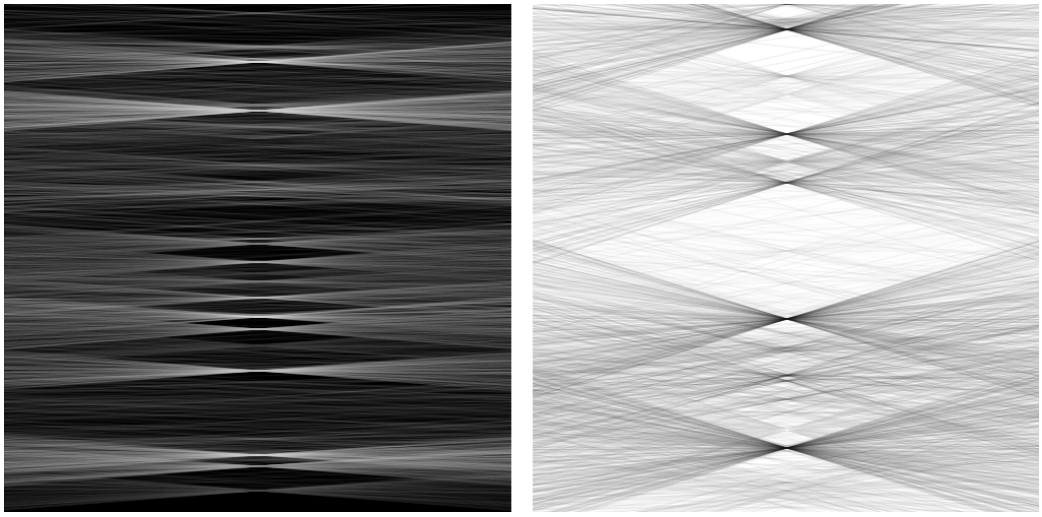


Рис. 42: Результат работы Листинга 36. Темные линии на светлом фоне, светлые – на темном.

Справа на Рисунке 42 показан результат выполнения Листинга 36, но с заменой цвета фона на белый и увеличенной величиной разброса линий относительно положения курсора мыши.

Задание 3. Измените код Листинга 36 так, чтобы на холсте было отрисовано такое же изображение, как на Рисунке 42 справа.

Рассмотрим код Листинга 37. В нем мы видим отрисовку двухмерной сетки прямоугольников. Причем если в 13-й строке идет перебор счетчика цикла до значения 10, то в 14-й строке – только до 5.

Листинг 37: Управляем цветной сеткой

```
1  void setup() {
2    size(500, 500);
3    smooth();
4    background(255);
5    noStroke();
6    colorMode(HSB);
7  }
8
9  boolean flug = true;
10
11 void draw() {
12   if(flug){
```

```
13        for (int i = 0; i < 10; i++) {
14          for (int j = 0; j < 5; j++) {
15            fill(10, random(0, 255), random(10, 250));
16            rect(j*40+50, i*40+50, 35, 35);
17            rect((10-j)*40+10, i*40+50, 35, 35);
18          }
19        }
20      }
21    }
22
23    void mouseClicked(){
24      flug = !flug;
25    }
```

Результат работы Листинга 37 представлен на Рисунке 43. Из рисунка становится ясна логика работы: квадраты отрисованы симметрично относительно вертикальной центральной оси. В 15-й строке мы задаемся цветом и отрисовываем два квадрата. Положение второго квадрата рассчитывается «обратным ходом» в 17-й строке. Из общего числа квадратов в строке (10 штук) мы вычитаем номер текущего квадрата, и так как номер текущего квадрата ограничен цифрой 4 ($i < 5$), то за пределы мы не выходим.

Условие в 12-й строке позволяет нам управлять этим пестрым ковром квадратов. Если мы кликнули мышью, то срабатывает метод *mouseClicked()* в строке 23 и мы попадаем в строку 24, где меняем значение свойства *flug*.

Переменная *flug* – это переменная логического типа (булевая), у которой может быть только два значения: истина и ложь (*true* и *false*). Мы использует логический оператор отрицания – ! для того, чтобы поменять значение переменной. Оператор отрицания «!» называется логическое «НЕ» и повсеместно используется в программировании. Например, если мы хотим сделать проверку по типу «не равно ли...», то можем писать *if(x != 20) { ... }*. Тогда если x не равен 20, то результат – истина (TRUE).

Итак, мы рассмотрели базовую работу с интерактивностью в Processing. Надо отметить, что методов работы с мышью и клавиатурой в Processing достаточное количество (весь список можно увидеть на сайте официальной документации). С развитием тактильных интерфейсов на мобильных устройствах становится актуальной работа с жестами пальцев. Processing поддерживает это на уровне подключаемых библиотек. В следующем параграфе мы посмотрим более интересные варианты использования интерактивности.

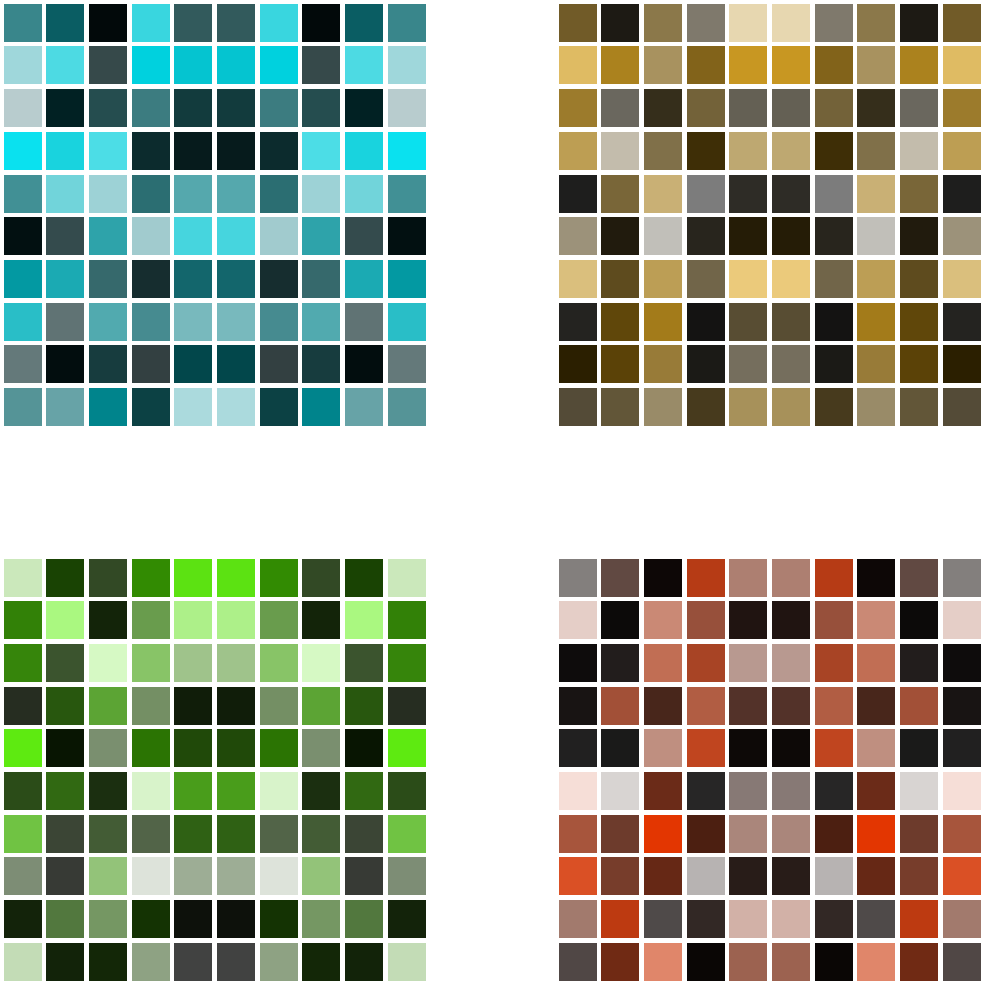


Рис. 43: Результат выполнения Листинга 37.
Управляем цветной симметричной сеткой.

6.2 Индивидуальные инструменты

Имея возможность использовать клавиатуру и мышь в своих приложениях, вы получаете свободу в создании собственных уникальных инструментов рисования. Результаты рисования могут быть сохранены с помощью метода *saveFrame()* и использованы вами, например, для подготовки полиграфической продукции. Взгляните на Листинг 38.

В 14-й строке происходит отрисовка линии из точки с координатами

курсора мыши в точку с рандомными координатами. Рандомные (полученные с помощью метода *random()*) координаты определяются в 12-й и 13-й строках.

Интерес представляет 11-я строка, в которой мы вызываем метод $stroke(flug, 20)$. Метод принимает два аргумента, первый из которых – переменная $flug$, которая при старте программы принимает значение 1 в 8-й строке. Таким образом как только программа начинает работать, линии отрисовываются черным цветом с небольшой прозрачностью (так как второй аргумент как раз отвечает за прозрачность).

Листинг 38: Рисуем сами

```
1  void setup(){
2    size(500, 500);
3    smooth();
4    background(150);
5    strokeWeight(1);
6  }
7
8  int flug = 1;
9
10 void draw () {
11     stroke(flug, 20);
12     float myY2 = mouseY + random(-10,10)*10;
13     float myX2 = mouseX + random(-10,10)*10;
14     line(mouseX, mouseY, myX2, myY2);
15 }
16
17 void keyPressed() {
18   if (key=='w') {
19     flug = 255;
20   }
21
22   if (key=='b') {
23     flug = 0;
24   }
25
26   if (key=='s') {
27     saveFrame("myProcessing.png");
28   }
29 }
```

Вы можете двигать мышью и тем самым создавать вашей уникальной кистью произведение генеративной графики. Если мышью не двигать, то плотность цвета усилится, если же двигать сравнительно быстро, то линии будут более равномерно и мягко заполнять пространство холста.

Обратите внимание на объявление метода *keyPressed()* в 17-й строке.

В нем есть три оператора проверки условия. Первый из них проверяет, не нажата ли кнопка *w*. Если да – то переменная *flug* получает значение 255 и тогда в 11-й строке цвет линии изменится с черного на белый. Второй оператор проверяет, не нажата ли кнопка *b* и переключает переменную *flug* снова в значение 1. Вы можете сохранить картинку своих трудов, нажав на кнопку *s* за этот функционал отвечает третий оператор с 26-й по 28-ю строку. Результаты использования такой кисти представлены на Рисунке 45.

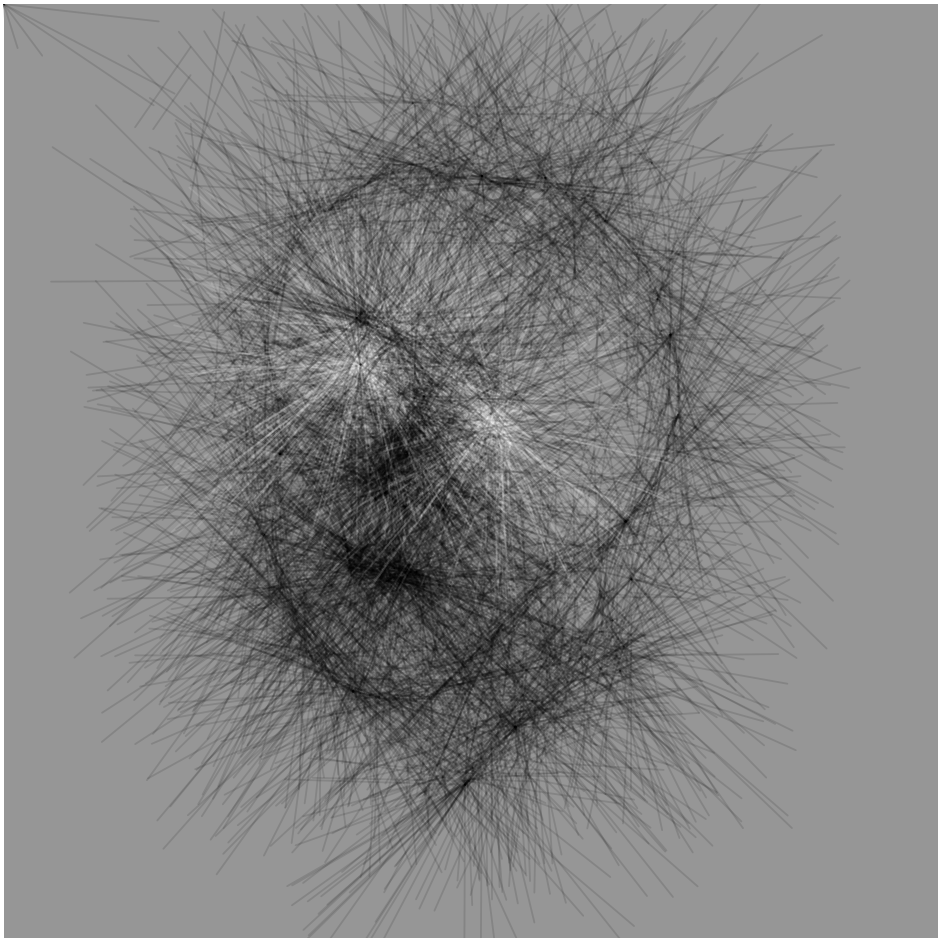


Рис. 44: Результат выполнения Листинга 38. Персонаж.

Задание 4. Измените код Листинга 38 так, чтобы на холсте отрисовывался шлейф за мышью из эллипсов разного размера и прозрачности.

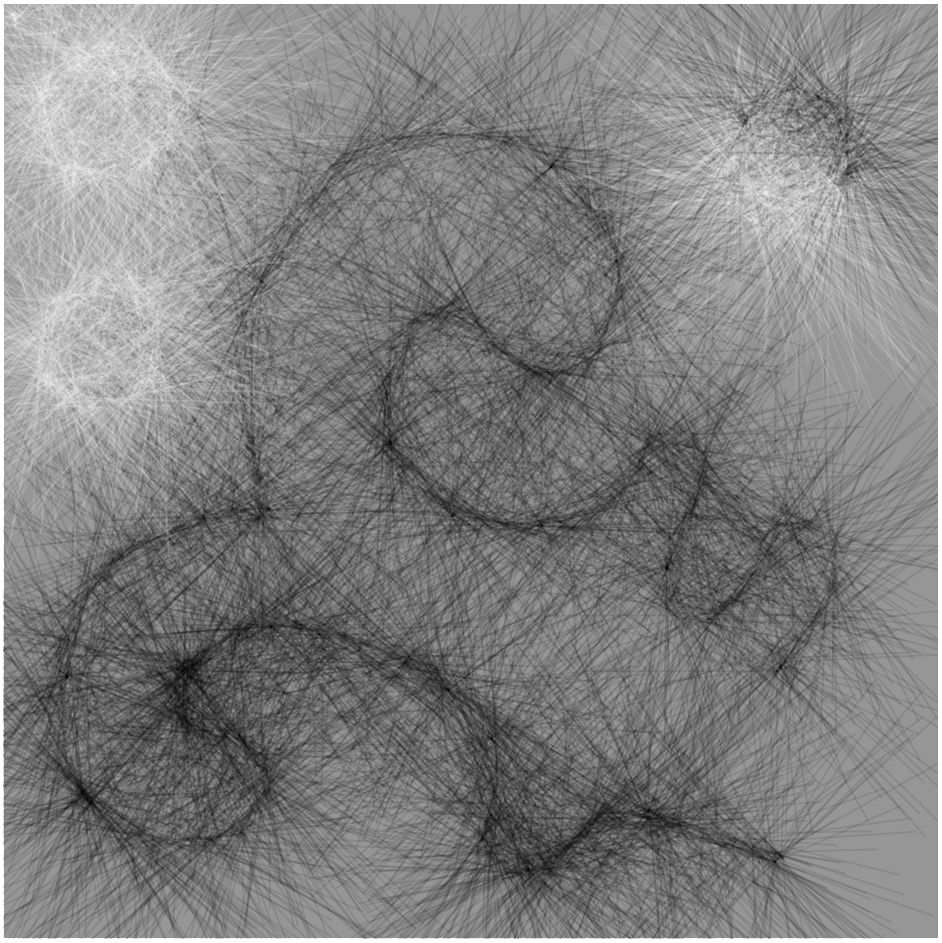


Рис. 45: Результат выполнения Листинга 38. Погремушка.

В этом разделе вы познакомились с богатыми возможностями такого, казалось бы, простого элемента, как линия. В Processing вы уже можете создавать свои *скетчи*, динамические приложения генеративной графики и даже собственные художественные инструменты.

7 Тригонометрия

Тригонометрические функции, знакомые нам со школьной скамьи, в Processing оказываются не такими уж сухие и скучными формулами: мы можем раскрыть их потенциал в работе с ритмом и гармонией при создании динамических произведений.

В этой главе мы рассмотрим работу с синусом и косинусом и, далее, с кривыми Безье. Мы разберем примеры использования тригонометрии как для создания генеративной графики, так и для реализации динамических интерактивных приложений.

7.1 Игры с синусом и косинусом

Работа художника без применения знаний о симметрии была бы лишена геометрического лаконизма и точности. Известно, что центральной симметрией обладает окружность. Что бы работать отрезками с окружностью, необходимо остановиться на базовых тригонометрических функциях. Рассмотрим код Листинга 39.

Листинг 39: Рисуем точки по кругу

```
1
2  void setup() {
3    size(500, 500);
4    smooth();
5    background(50);
6    strokeWeight(5);
7    stroke(250);
8    noLoop();
9  }
10
11 float cx = 250;
12 float cy = 250;
13 float cRadius = 200;
14
15 void draw() {
16   for(float i = 0; i < 2*PI; i = i + 2*PI/12){
17     float x1 = cos(i)*cRadius + cx;
18     float y1 = sin(i)*cRadius + cy;
19     line(x1, y1, x1, y1);
20   }
21   line(cx, cy, cx, cy);
22 }
23
24 void keyPressed() {
25   if (key=='s') saveFrame("myProcessing.png");
```

```
26 }
```

Со 2-й по 9-ю строки мы объявляем общие настройки нашего приложения. В них стоит обратить внимание только на строку 8, в которой мы вызываем метод *noLoop()*. Таким образом, наша программа будет отрисовывать только один кадр (метод *draw()* вызовется только один раз).

С 15-й по 22-ю строки идет более интересная часть исходного кода, а именно, объявление метода draw. В нем мы объявили цикл с условием выхода из цикла $i < 2 * PI$ и шагом $i = i + 2 * PI/12$. Processing «знает» константное значение математической величины ПИ, которую в исходном коде мы будем вызывать по имени. Число ПИ – это математическая константа, равная отношению длины окружности к длине её диаметра. Другими словами, эта константа обозначает, сколько раз диаметр окружности поместится в длину той же окружности: 3 раза и еще чуть-чуть. Более точное математическое описание этого «чуть-чуть» ведется с 1761 года, а сама константа представляет собой иррациональное, трансцендентное число. В Processing, чтобы получить его значение с определенной точностью, требуется всего лишь написать PI.

Итак, возвращаясь в 16-ю строку, мы рассматриваем условие выхода из цикла $i < 2 * PI$. Т.е. как только счетчик i превысит $2 * PI$, то цикл прекратит свою работу. Что же такое $2 * PI$? А это не что иное, как часть формулы длины окружности: $C = 2 * PI * R$, где C – длина окружности, а R – радиус. Если радиус принять за единицу, т.е. работать с единичной окружностью, то и останется $2 * PI$.

Таким образом, наш цикл как бы обходит окружность целиком, но не по каждой точке, а с шагом $2 * PI/12$ (что равноценно $PI/6$). В каждой такой точке единичной окружности мы определяем ее положение для реальной окружности с радиусом *cRadius* и положением центра окружности в точке cx, cy. Эти операции мы проводим в строке 17 для оси X и в строке 18 для оси Y, соответственно.

В строке 17, для вычисления координаты *x1* мы вычисляем косинус от *i*. Рассмотрим рисунок 47 А, поясняющий нашу логику.

На окружности радиуса R косинусом угла альфа является отношение (см. формулы 1 и 2):

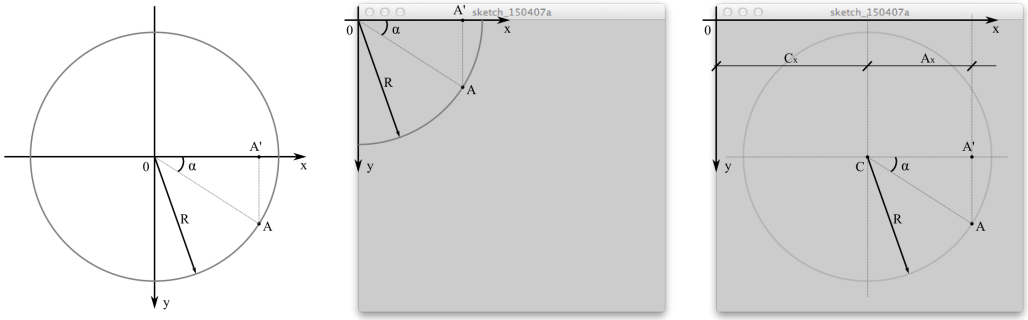
$$sin(\alpha) = \frac{R}{0A'} \tag{1}$$

или

$$sin(\alpha) = \frac{0A}{0A'} \tag{2}$$

Для воплощения тригонометрических изысканий в поле нашего приложения совместим начало координат *0* осей окружности с началом ко-

Рис. 46: Результат выполнения Листинга 39.
Белые точки по кругу.



A B C

Рис. 47: Принцип построения: три шага построения.

ординат поля нашего приложения, как показано на рисунке 47 B. Тогда чтобы найти длину отрезка $0A'$, необходимо выполнить умножение: $cos(\alpha) * R$.

Так как центр окружности в нашем приложении располагается не в начале координат, точка A сдвинется с текущего места на такое же расстояние, на котором находится и точка *0*. Рассмотрим рисунок 47 C. Центр окружности теперь обозначим точкой C, который сдвинулся по оси x на расстояние C_x. Тогда координата точки A стала равняться сумме C_x и A_x. Подитоживая, для определения координаты x точки A,

необходимо выполнить: $cos(i) * cRadius + cx$, что и написано в строке 17 Листинга 39. Эта же логика описывает получение координаты y, с использованием синуса.

Таким образом мы отрисовали 12 точек на равном расстоянии по окружности с центром cx, cy и с радиусом *cRadius*. С первого взгляда кажется, что это можно рассматривать лишь как заготовку для циферблата, однако использование синуса и косинуса дает возможность создания очень интересных с художественной точки зрения работ. Давайте их посмотрим

Задание 1. Измените код Листинга 39 так, чтобы линии, выходящие из центра окружности к ее границе, были разных цветов, как на Рисунке 48.

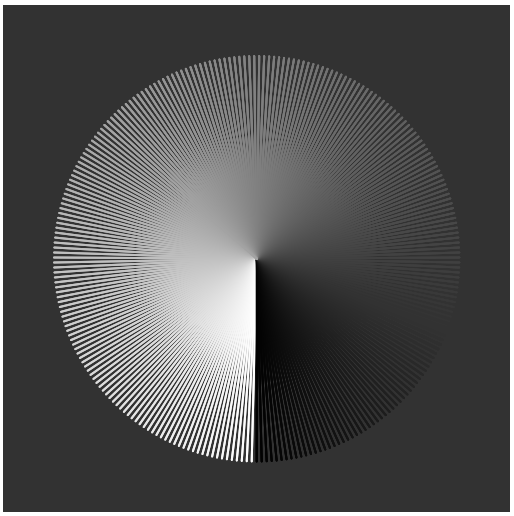


Рис. 48: Линии на холсте выходящие из центра по кругу.

Если нам требуется отрисовывать линии в интерактивном режиме, то цикл по всей окружности $2 * PI$ есть смысл «перенести» на метод *draw()*, как показано в коде Листинга 40.

Листинг 40: Линии на холсте выходящие из центра по кругу

```
1
2  void setup() {
3    size(500, 500);
4    smooth();
```

```
5     background(50);
6     strokeWeight(2);
7     stroke(250);
8   }
9
10  float counter;
11  float cx = 250;
12  float cy = 250;
13  float cRadius = 200;
14  int mcolor;
15
16  void draw() {
17
18    float x1 = sin(counter)*cRadius + cx;
19    float y1 = cos(counter)*cRadius + cy;
20    mcolor = mcolor + 1;
21    stroke(mcolor);
22    line(cx, cy, x1, y1);
23    counter = counter + 2*PI/255;
24
25    if(counter > 2*PI){
26      counter = 0;
27      mcolor = 0;
28      background(50);
29    }
30  }
31
32  void keyPressed() {
33    if (key=='s') saveFrame("myProcessing.png");
34  }
```

Результат выполнения кода Листинга 40 показан на рисунке 48 (но в отличие от задания, в этом случае мы получили анимированную картину).

Таким образом, варьируя координаты отрезка с помощью функций тригонометрии, мы получаем возможность создавать сложные графические композиции. Рассмотрим код Листинга 41 с результатом выполнения на Рисунке 49

Листинг 41: Рисуем геометрический цветок

```
1
2   void setup() {
3     size(500, 500);
4     smooth();
5     background(255);
6     strokeWeight(1);
7   }
8
```

```
9  float counter, counter1;
10 float cx = 250;
11 float cy = 250;
12 float cRadius = 200;
13
14 void draw() {
15   stroke(0, 30);
16
17   float nx = sin(counter1)*cRadius + cx;
18   float ny = cos(counter1)*cRadius + cy;
19
20   float x1 = nx - sin(counter)*200;
21   float y1 = ny - cos(counter)*200;
22   float x2 = nx + sin(counter)*200;
23   float y2 = ny + cos(counter)*200;
24
25   line(x1, y1, x2, y2);
26
27   counter += 0.1;
28   if (counter > 2*PI) {
29     counter = 0;
30   }
31
32   counter1 += 0.01;
33
34   if (counter1 > 2*PI) {
35     counter1 = 0;
36   }
37 }
38
39 void keyPressed() {
40   if (key=='s') {
41     saveFrame("myProcessing.png");
42   }
43 }
```

Отличие кода Листинга 41 от предыдущего кода проявляется в использовании двух счетчиков *counter* и *counter1*. Счетчики объявлены вне метода *draw()* (в строке 9), поэтому сохраняют свое состояние между вызовами метода (*draw()*). Приращение у счетчиков разное: счетчик *counter* изменяет свое значение в 27-й строке на *0.1*, *counter1* – в 32-й строке на *0.01*. Но оба счетчика обнуляются, как только становятся больше значения *2*PI*.

Счетчик *counter1* используется для вычисления координат *nx* и *ny*. Воображаемая точка *n* с этими координатами двигается по окружности с радиусом *cRadius* и центром в точке *cx*, *cy*. Эту точку n мы не отрисовываем, а используем ее для получения следующих координат.

Итак, у нас есть точка *n*, которую мы принимаем за центр новой окружности с радиусом 200. По этой окружности мы рассчитываем координаты двух точек *x1* и *y1* и *x2* и *y2*, которые используем для отрисовки отрезка в строке 25. Получается, что мы отрисовываем отрезок, концы которого вращаются вокруг точки, которая в свою очередь вращается вокруг центра с координатами *cx* и *cy*. В результате мы получаем изображение, представленное на Рисунке 49.

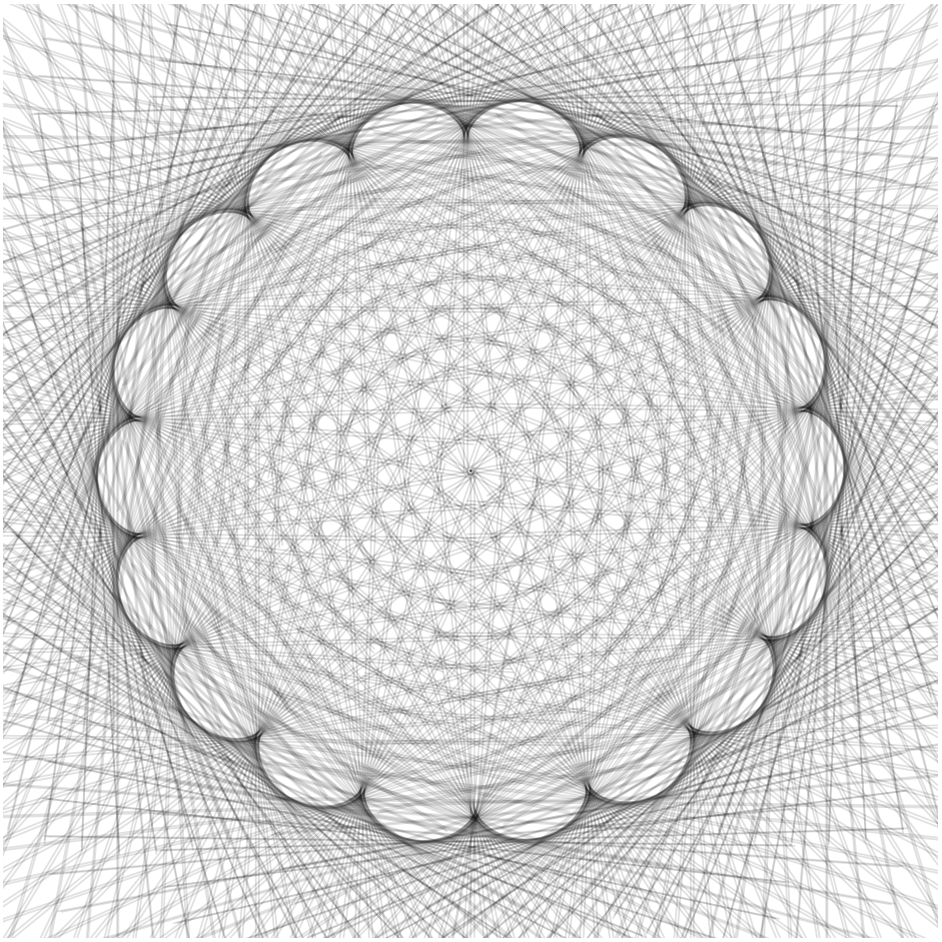


Рис. 49: Результат выполнения Листинга 41.
Геометрический цветок.

Вынесем логику отрисовки линий в отдельный метод *oneLineDraw()*, как показано в Листинге 42. И добавим еще и изменение радиуса первой окружности, по которой вращается воображаемая точка *n*. Для изменения радиуса введем свойство (переменную) *cRadius*, которая будет зависеть от *counter* (см. строку 42).

Метод *oneLineDraw()* принимает два аргумента x и y. Эти аргументы он использует для формирования координат и отрисовки линии в строках с 15-й по 19-ю. Логика вычисления координат очень похожа на логику кода предыдущего Листинга: так же используются методы для вычисления синуса и косинуса. Координаты вычисляются, исходя из радиуса, равного 100 и счетчика *counter1*. Вы, конечно, можете изменить логику вичисления координат, что бы достичь нового художественного результата отрисовки.

Листинг 42: Рисуем синусоидную звезду на черном фоне

```
1
2  float counter, counter1, cx, cy, nx, ny;
3  float cRadius = 20;
4
5  void setup() {
6    size(500, 500);
7    smooth();
8    background(0);
9    strokeWeight(1);
10   cx = width/2;
11   cy = height/2;
12 }
13
14 void oneLineDraw(float ncx, float ncy) {
15   float x1 = ncx - sin(counter1)*(100);
16   float y1 = ncy - cos(counter1)*(100);
17   float x2 = ncx + sin(counter1)*(100);
18   float y2 = ncy + cos(counter1)*(100);
19   line(x1, y1, x2, y2);
20 }
21
22 void draw() {
23   stroke(200, 25);
24
25   nx = sin(counter1)*cRadius + cx;
26   ny = cos(counter1)*cRadius + cy;
27
28   float x1 = nx - sin(counter)*(150);
29   float y1 = ny - cos(counter)*(150);
30   float x2 = nx + sin(counter)*(150);
31   float y2 = ny + cos(counter)*(150);
32
33   oneLineDraw(x2, y2);
34   oneLineDraw(x1, y1);
35
36   counter += 0.1;
37   if (counter > 2*PI) {
```

```
38        counter = 0;
39      }
40
41      counter1 += 0.01;
42      cRadius += counter/20;
43
44      if (counter1 > 2*PI) {
45        counter1 = 0;
46      }
47    }
```

Результат работы кода Листинга 42 показан на Рисунках 50 и 51 (с измененным размером центральной части).

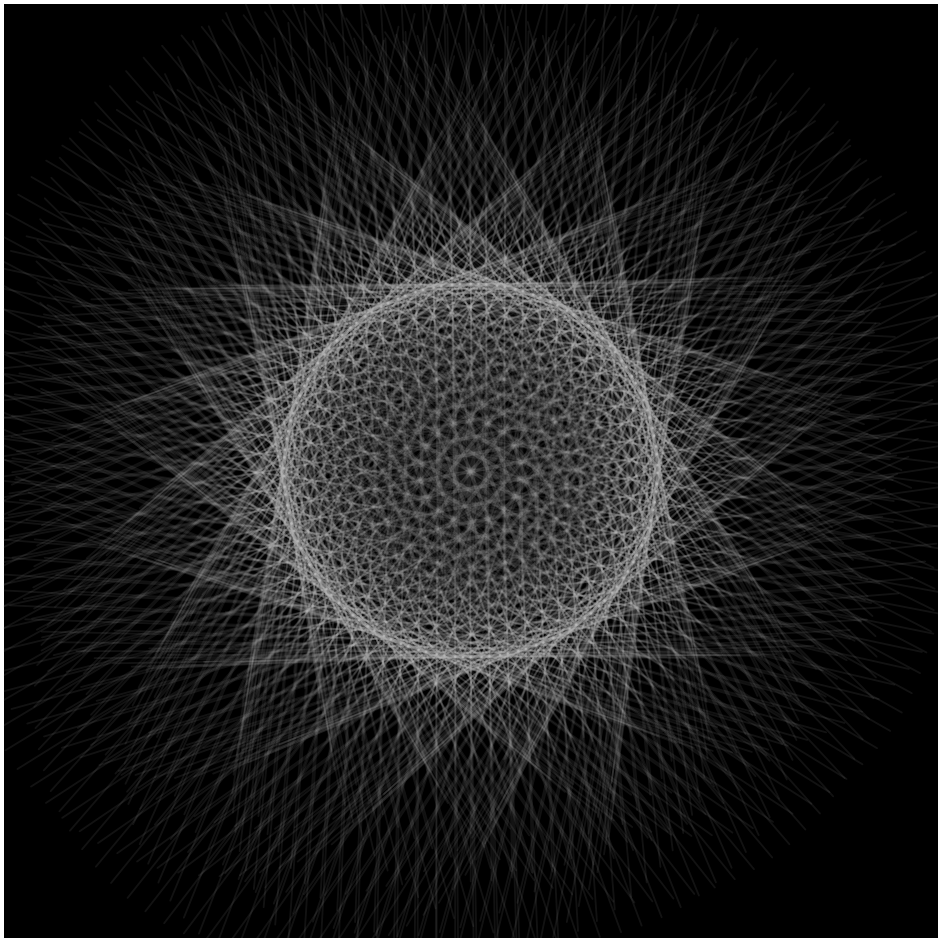


Рис. 50: Результат выполнения Листинга 42.
Синусоидная звезда на черном фоне 1.

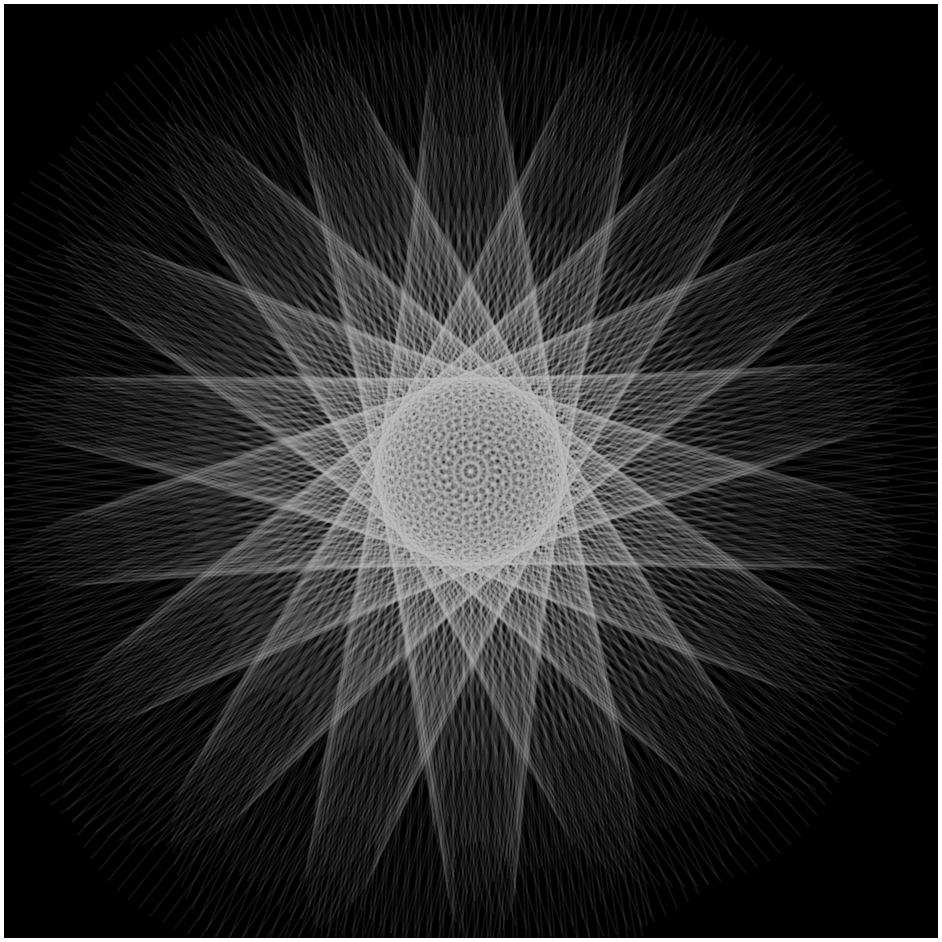


Рис. 51: Результат выполнения Листинга 42.
Синусоидная звезда на черном 2.

На правой части рисунка показаны первые кадры отрисовки. Рендеринг (отрисовка) происходит не сразу: каждый кадр приложение рисует одну линию, как бы вышивая белый узор на черном поле холста. Это «прорастание», постепенное формирование графики и является признаками генеративного метода получения изображений.

Изменяя параметры отрисовки, например, толщину линий, размер окружности и длину линий в методе *oneLineDraw()*, можно получить интересные эффекты (см. Рисунок 52).

Задание 2. Измените код Листинга 42 так, чтобы линии на холсте формировали рисунок, похожий на Рисунок 52.

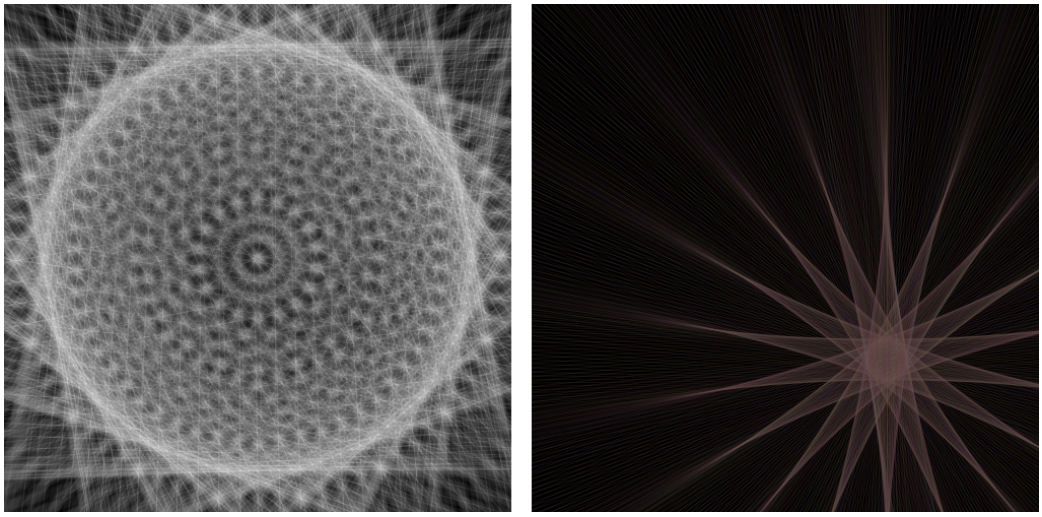


Рис. 52: Результат задания 2. Вариации на тему звезды.

Приведем еще один пример использования тригонометрических функций. В Листинге 43 мы строим прямоугольную сетку точек. Сетка строится двумя циклами *for*, вложенными друг в друга (см. строки 15 и 16).

Каждый раз, когда вызывается метод *draw()*, циклы отрабатывают свои задачи. А именно, в 17-й и 18-й строках формируются координаты текущей точки сетки; с 20-й по 23-ю строку формируются координаты начала и конца отрезков в зависимости от значения счетчика *counter*.

Листинг 43: Рисуем диагональные звезды

```
1
2  void setup() {
3    size(500, 500);
4    smooth();
5    background(0);
6    strokeWeight(1);
7  }
8
9  float counter, nx, ny;
10 float cx = 250;
11 float cy = 250;
12
13 void draw() {
14   stroke(200, 5);
15   for (float si = 0; si < 6; si+=1) {
16     for (float ci = 0; ci < 6; ci+=1) {
17       nx = ci*80 + 50;
18       ny = si*80 + 50;
```

```
19
20          float x1 = nx - sin(counter)*(50);
21          float y1 = ny - cos(counter)*(50);
22          float x2 = ny + sin(counter)*(50);
23          float y2 = nx + cos(counter)*(50);
24          line(x1, y1, x2, y2);
25        }
26      }
27
28      counter += 0.1;
29      if (counter > 2*PI) {
30        counter = 0;
31      }
32
33    }
```

В результате выполнения кода Листинга 43 мы получаем изображение, показанное на Рисунке 53.

Рис. 53: Результат выполнения Листинга 43.
Диагональные звезды на черном фоне.

Задание 3. Измените код Листинга 43: после 18-й строки вставьте отрисовку точки сетки (т.е. необходимо отрисовывать точки с координатами nx и ny).

На примере кода Листинга 44 можно увидеть использование переключателя *switcher*. В строке 27 переключатель меняет свой знак. От знака этого переключателя зависит формирование координат nx и ny. Логическое условие в строке 17 обеспечивает исполнение строк или 18 – 20, или 22 – 24. Таким образом при каждом вызове метода *draw()* отрисовыется или светлый отрезок, или темный.

Листинг 44: Синусоидный орнамент из крупных отрезков

```
1
2  void setup() {
3    size(500, 500);
4    smooth();
5    background(0);
6    strokeWeight(50);
7  }
8
9  float counter, counter1, nx, ny;
10 float cx = 250;
11 float cy = 250;
12 float cRadius = 10;
13 int switcher = 1;
14
15 void draw() {
16
17   if (switcher > 0) {
18     nx = cos(counter1)*cRadius + cx;
19     ny = sin(counter1)*cRadius + cy;
20     stroke(0, 50);
21   } else {
22     nx = sin(counter1)*cRadius + cx;
23     ny = cos(counter1)*cRadius + cy;
24     stroke(250, 50);
25   }
26
27   switcher *= -1;
28
29   float x1 = nx - sin(counter)*(20);
30   float y1 = ny - cos(counter)*(20);
31   float x2 = nx + sin(counter)*(20);
32   float y2 = ny + cos(counter)*(20);
```

```
33
34      line(x1, y1, x2, y2);
35
36      counter += 0.1;
37      if (counter > 2*PI) {
38        counter = 0;
39      }
40
41      counter1 += 0.01;
42
43      cRadius += counter/50;
44
45      if (counter1 > 2*PI) {
46        counter1 = 0;
47      }
48    }
```

Результат работы кода Листинга 44 показан на рисунке 54. Этот пример показывает возможности перерисовки изображения, что в конечном итоге формирует интересное художественное решение.

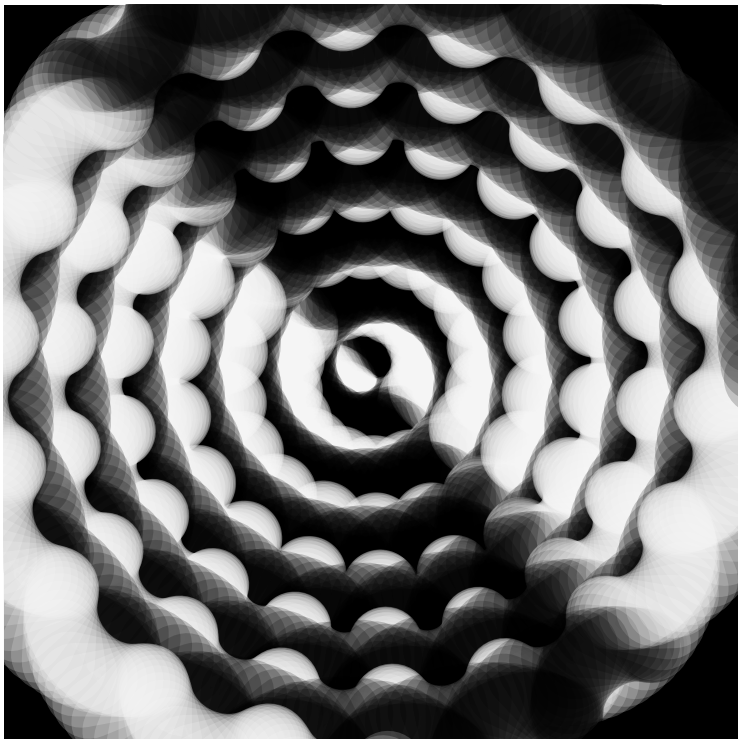


Рис. 54: Результат выполнения Листинга 44.
Синусоидный орнамент из крупных отрезков.

Изменяя толщину линий в строке 6, как показано в коде Листинга 45, можно добиться абсолютно другого художественного результата. Наше приложение начинает плести узор линий, накладывая их друг на друга, дополняя друг друга, а не перерисовывая, как в предыдущем случае.

Листинг 45: Тонкие линии по спирали

```
1
2  void setup() {
3    size(500, 500);
4    smooth();
5    background(120);
6    strokeWeight(1);
7  }
8
9  float counter, counter1, nx, ny;
10 float cx = 250;
11 float cy = 250;
12 float cRadius = 10;
13
14 int switcher = 1;
15
16
17 void draw() {
18
19   if (switcher > 0) {
20     nx = cos(counter1)*cRadius + cx;
21     ny = sin(counter1)*cRadius + cy;
22     stroke(0, 50);
23   } else {
24     nx = sin(counter1)*cRadius + cx;
25     ny = cos(counter1)*cRadius + cy;
26     stroke(250, 50);
27   }
28
29   switcher *= -1;
30
31
32   float x1 = nx - sin(counter)*(30);
33   float y1 = ny - cos(counter)*(30);
34   float x2 = nx + sin(counter)*(30);
35   float y2 = ny + cos(counter)*(30);
36
37   line(x1, y1, x2, y2);
38   counter += 0.1;
39   if (counter > 2*PI) {
40     counter = 0;
41   }
42
```

```
43      counter1 += 0.01;
44
45      cRadius += counter/50;
46
47      if (counter1 > 2*PI) {
48        counter1 = 0;
49      }
50    }
```

Результат работы кода Листинга 45 показан на Рисунке 55.

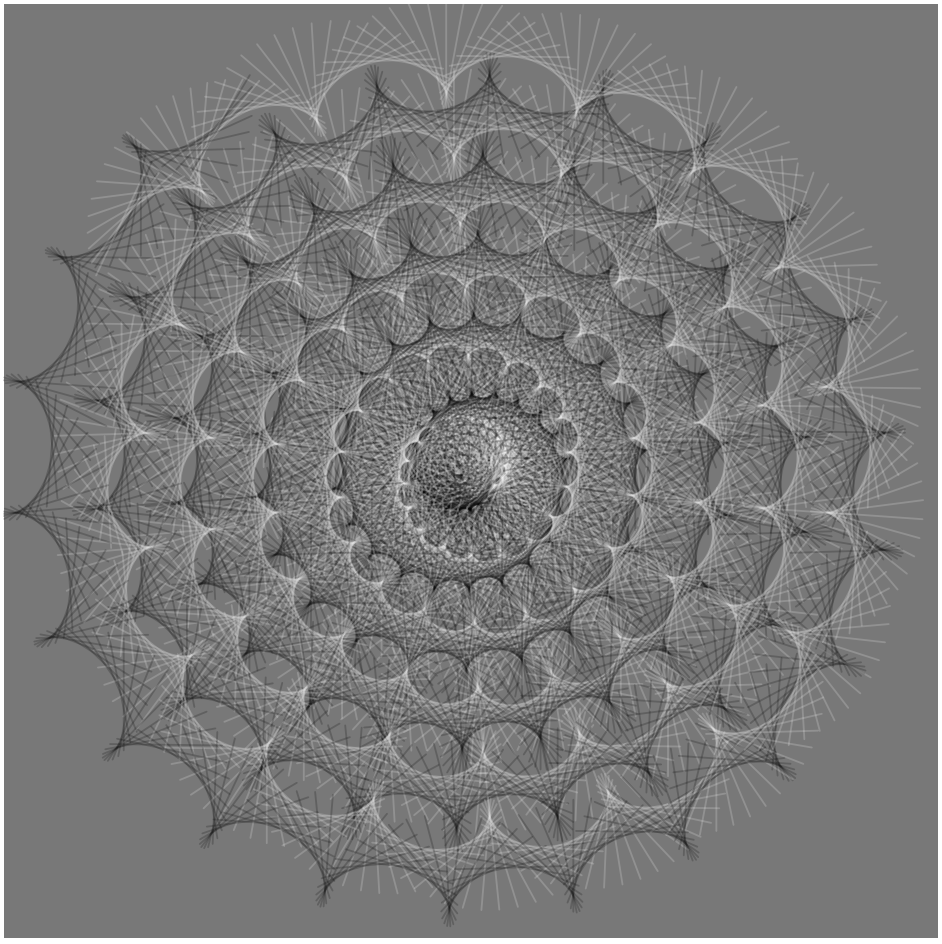


Рис. 55: Результат выполнения Листинга 45.
Тонкие линии по спирали.

Традиционным примером работы с тригонометрическими функциями считается отрисовка спирали. Вариации на эту тему представлены в коде Листинга 46. Переменная *cRadius* с каждым вызовом метода *draw()* меняет свое значение. Это приращение реализовано в строке

31: $cRadius+ = counter1/30;$. Таким образом, при обнулении счетчика *counter1* значение радиуса не уменьшается и далее продолжает расти.

Значение *cRadius* используется в строках 16 и 17 для формирования значения переменных координат *nx* и *ny*.

Листинг 46: Рисуем белый цветок на белом фоне

```
1
2  void setup() {
3    size(500, 500);
4    smooth();
5    background(255);
6    strokeWeight(1);
7  }
8
9  float counter, counter1;
10 float cx = 250;
11 float cy = 250;
12 float cRadius = 10;
13 void draw() {
14   stroke(0, 50);
15
16   float nx = sin(counter1)*cRadius + cx;
17   float ny = cos(counter1)*cRadius + cy;
18
19   float x1 = nx - sin(counter)*(50);
20   float y1 = ny - cos(counter)*(50);
21   float x2 = nx + sin(counter)*(50);
22   float y2 = ny + cos(counter)*(50);
23   line(x1, y1, x2, y2);
24
25   counter += 0.1;
26   if (counter > 2*PI) {
27     counter = 0;
28   }
29
30   counter1 += 0.01;
31   cRadius += counter1/30;
32
33   if (counter1 > 2*PI) {
34     counter1 = 0;
35   }
36 }
37
38 void keyPressed() {
39   if (key=='s') saveFrame("myProcessing.png");
40 }
```

В результате выполнения кода Листинга 46 мы получаем спиралевидное изображение, показанное на рисунке 56.

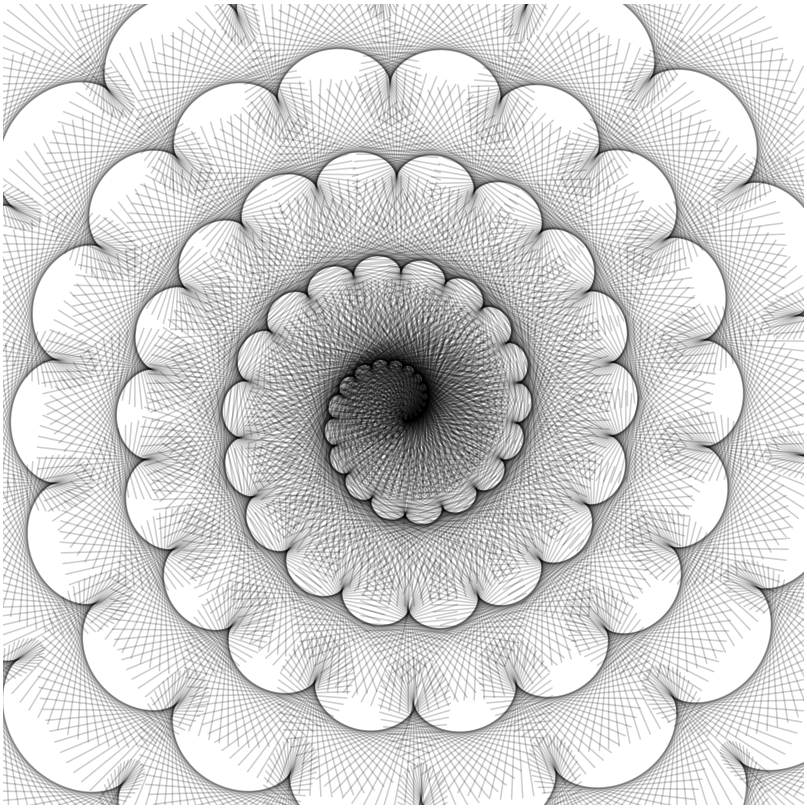


Рис. 56: Результат выполнения Листинга 46.
Белый цветок на белом фоне.

Модифицируем код Листинга 46 так, как показано в Листинге 47. Изменение коснулось формирования значения счетчика *counter1* в 32-й строке. Теперь значение счетчика зависит от горизонтального положения курсора мыши, таким образом мы добавили в наше приложение элемент интерактивного взаимодействия: пользователь может управлять свойствами отрисовки и получать результаты, как, например, на Рисунке 57.

Листинг 47: Рисуем симметрию на белом

```
1
2  void setup() {
3    size(500, 500);
4    smooth();
5    background(255);
6    strokeWeight(1);
```

```
7  }
8
9  float counter, counter1;
10 float cx = 250;
11 float cy = 250;
12 float cRadius = 100;
13
14 void draw() {
15   stroke(0, 30);
16
17   float nx = sin(counter1)*cRadius + cx;
18   float ny = cos(counter1)*cRadius + cy;
19
20   float x1 = nx - sin(counter)*300;
21   float y1 = ny - cos(counter)*300;
22   float x2 = nx + sin(counter)*300;
23   float y2 = ny + cos(counter)*300;
24
25   line(x1, y1, x2, y2);
26
27   counter += 0.1;
28   if (counter > 2*PI) {
29     counter = 0;
30   }
31
32   counter1 = counter1 + mouseX/((float)width*2);
33
34   if (counter1 > 2*PI) {
35     counter1 = 0;
36   }
37 }
38
39 void keyPressed() {
40   if (key=='s') saveFrame("myProcessing.png");
41 }
```

Результат работы кода Листинга 47 показан на рисунке 57.

Задание 4. Измените код Листинга 46 так, чтобы вместо линий отрисовывались эллипсы разных оттенков и цветов.

Задание 5. Измените код Листинга 46 так, чтобы степень «закрученности» спирали зависела от положения курсора мыши на горизонтальной оси.

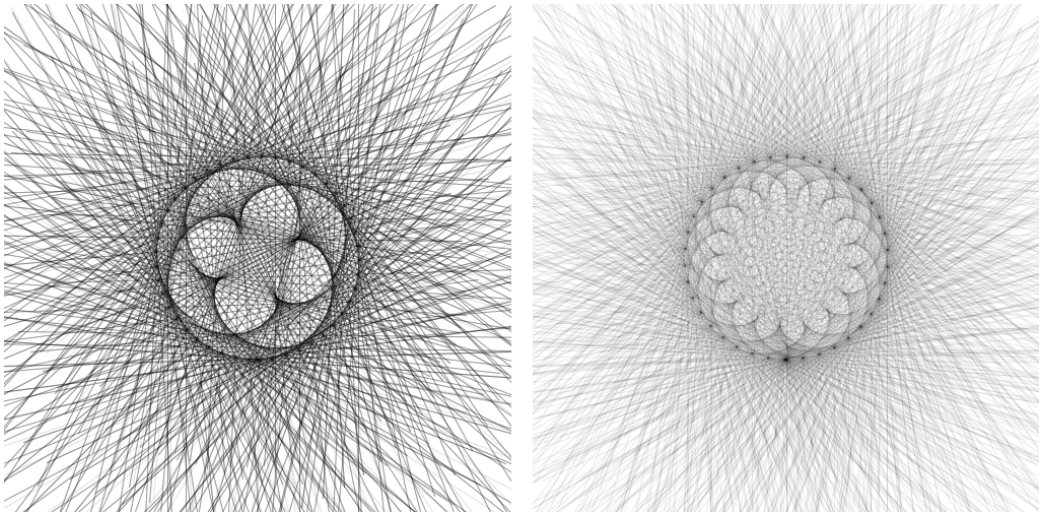


Рис. 57: Результат выполнения Листинга 47. Симметрия на белом.

В этом параграфе мы рассмотрели варианты работы с тригонометрическими функциями, разобрали примеры построения спиралей. В последующих главах мы еще вернемся к функциям синуса и косинуса для демонстрации интересных художественных решений с использованием синусоидальных траекторий движения.

7.2 Кривые Безье

Кривые Безье часто используются в компьютерной графике: например, векторная графика (о которой мы еще будем говорить отдельно) активно использует их для описания геометрии. Математический аппарат кривых Безье мы рассматривать не будем, лишь взглянем на Рисунок **??**. На нем можно увидеть, что кривая Безье состоит из опорных точек и кривой (путем). В Processing опорные точки для упрощения разделили на якоря (anchors), через которые кривая проходит, и контрольные точки (control points), которые задают форму кривой. На Рисунке 58 видно, что кривая начинается в якоре A и стремиться к якорю B, аформа кривой задается положением контрольных точек C и D.

В редакторах растровой графики всегда есть схожие элементы управления визуальной настройкой кривой Безье. В Processing нам придется указывать координаты для каждой опорной точки, т.е. 8 цифр.

Рассмотрим пример отрисовки кривой Безье. В коде Листинга 48 отрисовываем кривую Безье в 23-й строке с помощью вызова метода *bezier()*. Этот метод принимает 8 аргументов. Первые два аргумента –

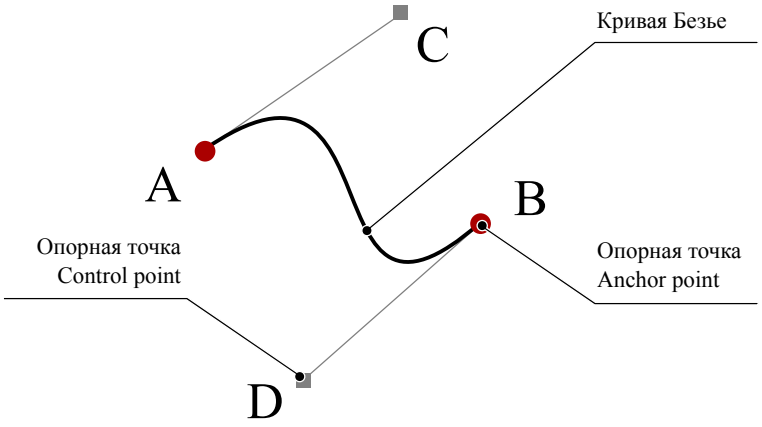


Рис. 58: Визуализация кривой Безье с опорными точками.

это X *и* Y координаты первой якорной точки, вторая пара X *и* Y – координаты первой контрольной точки, третья пара, соответственно, – вторая контрольная точка и, наконец, четвертая пара аргументов – координаты второй якорной точки.

Листинг 48: Управляем опорными точками кривой Безье

```
1  void setup(){
2    size(660, 400);
3    smooth();
4  }
5
6  void draw(){
7    background(255);
8    stroke(100);
9    strokeWeight(5);
10
11   float anc1X = mouseX;
12   float anc1Y = mouseY;
13   float cont1X = width/2;
14   float cont1Y = height/2 - 100;
15   float cont2X = width/2;
16   float cont2Y = height/2 + 100;
17   float anc2X = width - mouseX;
18   float anc2Y = height - mouseY;
19
20   noFill();
21   bezier(anc1X,anc1Y,cont1X,cont1Y,cont2X,cont2Y,anc2X,
       anc2Y);
```

```
22
23      stroke(50);
24      strokeWeight(1);
25      line(anc1X,anc1Y,cont1X,cont1Y);
26      line(anc2X,anc2Y,cont2X,cont2Y);
27
28      fill(150);
29      noStroke();
30      rectMode(CENTER);
31      rect(cont1X,cont1Y, 6,6);
32      rect(cont2X,cont2Y, 6,6);
33
34      fill(170,0,0);
35      noStroke();
36      ellipseMode(CENTER);
37      ellipse(anc1X,anc1Y,10,10);
38      ellipse(anc2X,anc2Y,10,10);
39  }
```

Координаты опорных точек мы определяем в строках с 11-й по 18-ю. Первый якорь будет располагаться в месте курсора мыши, второй якорь будет зеркально отражен (см. строку 17 и 18) по горизонтали и вертикали. Контрольные точки будут зафиксированы в центре по горизонтали и равноудалены от центральной горизонтальной оси. Кроме отрисовки самойсобственно кривой (пути), мы отрисуем и опорные точки, и управляющие линии. Результат работы кода Листинга 48 можно увидеть на Рисунке 59.

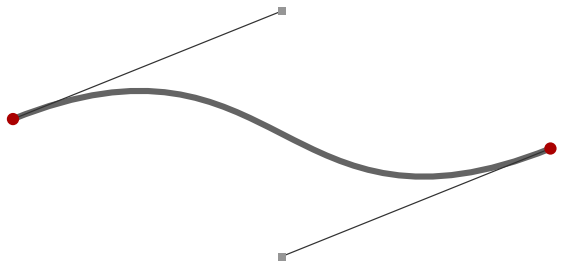


Рис. 59: Результат выполнения Листинга 48.
Управление опорными точками кривой Безье.

Рассмотрим интересный пример использования кривых Безье Михаэлем Пинном (Michael Pinn, http://www.openprocessing.org/sketch/159891), который разработал приложение и опубликовал исходные коды под открытой лицензией. Кривые Безье в коде Листинга 49 слагают симметричный ансамбль формы.

Листинг 49: http://www.openprocessing.org/sketch/159891

```
1
2  float num, pathR, pathG, pathB;
3
4  void setup(){
5    size(500, 500);
6  }
7
8  void draw(){
9    fill(0, 30);
10   rect(-1, -1, width+1, height+1);
11
12   float maxX = map(mouseX, 0, width, -150, 150);
13   float maxY = map(mouseY, 0, height, -150, 150);
14
15   translate(width/2, height/2);
16
17   for(int i = 0; i < 360; i+=2){
18     float angle = sin(i+num);
19     float x = sin(radians(i)) * (maxX + angle*30);
20     float y = cos(radians(i)) * (maxX + angle*30);
21
22     float x2 = sin(radians(i+num*50)) * (maxY + angle*60);
23     float y2 = cos(radians(i+num*50)) * (maxY + angle*60);
24
25     pathR = 50+angle+125*sin(PI+num*3);
26     pathG = 50+angle+125*sin(TWO_PI+num*3);
27     pathB = 50+angle+125*sin(HALF_PI+num*3);
28
29     stroke(pathR, pathG, pathB, 50);
30     fill(pathR, pathG, pathB, 30);
31
32     bezier(x, y, x+x, y+y, x+x2, y+y2, x+x2, y+y2);
33   }
34   num += 0.01;
35 }
```

Результат работы кода Листинга 49 показан на Рисунке 60.

В коде Листинга 49 за каждый вызов метода *draw()* происходит отрисовка 180 (360 с шагом 2) кривых Безье в цикле *for* (см. строку 17).

Кривая Безье строится по двум якорным точками, координаты которых определяются в 19-й – 20-й и 22-й – 23-й строках соответственно. В вычислении координат участвует свойство *num*, которое определено во 2-й строке и увеличивается на 0.01 в 34-й строке с каждым вызовом метода *draw()*. Таким образом происходит не только синусоидальное изменение координат из-за переменной *i*, но и изменение угла (строка 18).

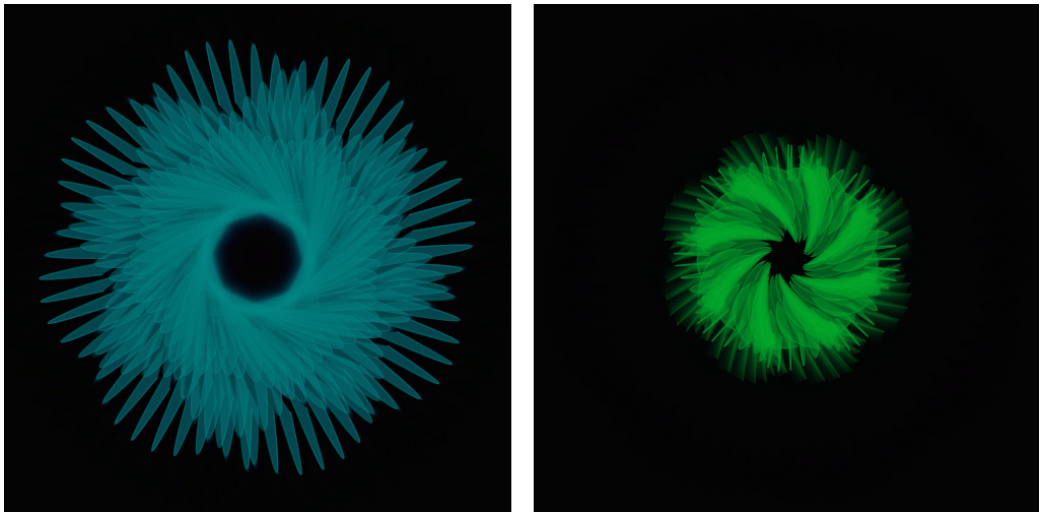


Рис. 60: Результат выполнения Листинга 49.

Изменение цвета кривых и заливки происходит в строках 25–27. Значения трех цветовых каналов меняются также в зависимости от переменной num. Изменения каналов происходят со сдвигом на PI, TWO_PI и HALF_PI.

Одной из вариаций на тему этого кода является следующий пример: Михаэль Пинн опубликовал его так же под открытой лицензией (Michael Pinn, http://www.openprocessing.org/sketch/160305).

Листинг 50: http://www.openprocessing.org/sketch/160305

```
1
2  float amount = 20, num;
3
4  void setup() {
5    size(640, 640);
6    stroke(0, 150, 255, 100);
7  }
8
9  void draw() {
10   fill(0, 40);
11   rect(-1, -1, width+1, height+1);
```

```
12
13    float maxX = map(mouseX, 0, width, 1, 250);
14
15    translate(width/2, height/2);
16    for (int i = 0; i < 360; i+=amount) {
17      float x = sin(radians(i+num)) * maxX;
18      float y = cos(radians(i+num)) * maxX;
19
20      float x2 = sin(radians(i+amount-num)) * maxX;
21      float y2 = cos(radians(i+amount-num)) * maxX;
22      noFill();
23      bezier(x, y, x-x2, y-y2, x2-x, y2-y, x2, y2);
24      bezier(x, y, x+x2, y+y2, x2+x, y2+y, x2, y2);
25      fill(0, 150, 255);
26      ellipse(x, y, 5, 5);
27      ellipse(x2, y2, 5, 5);
28    }
29    num += 0.5;
30  }
```

Результат работы кода Листинга 50 показан на Рисунке 61. В этой работе нет изменения цвета. В отличие от предыдущего кода, здесь отрисовываются две кривые Безье (строки 23 и 24) и эллипсы (строки 26 и 27).

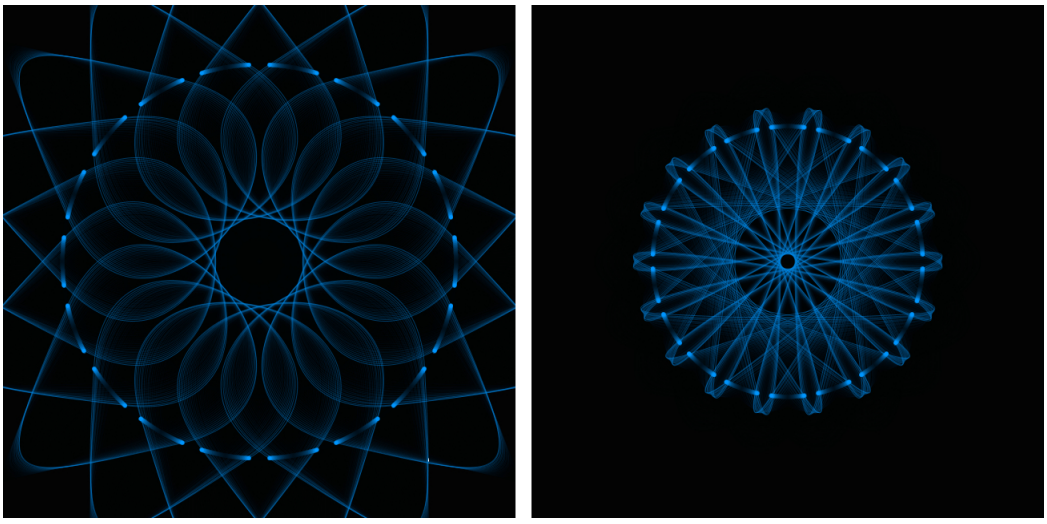


Рис. 61: Результат выполнения Листинга 50.

Первый эллипс (строка 26) отрисовывается в координатах первой якорной точки кривой Безье (строка 23), второй эллипс – во второй. Причем якорные точки обеих кривых совпадают на координатной плоскости, однако положение контрольных точек симметрично.

Задание 6. Измените код Листинга 47 так, чтобы вместо линий отрисовывались кривые Безье.

Можно отметить, что плавность линий и изящество построений кривых Безье расширяют возможности создания простых геометрических фигур. Конечно, кривые Безье требуют много параметров: необходимо думать о расположении якорей и контрольных точек. Тем не менее эти кривые – уникальное средство в руках цифрового художника. Мы будем использовать кривые Безье в дальнейшем, так что у вас еще появится возможность увидеть их «в действии».

В этой главе мы рассмотрели применение тригонометрических функций и работу с кривыми Безье в Processing. Удобство использования этих функций и кривых позволяет включать их в работу на любом этапе. Особенно интересным представляется использование их для отрисовки не собственно линий, а путей, по которым будут отрисовываться объекты и формироваться художественные произведения. Фундаментальному вопросу о том, какие вообще объекты могут отрисовываться на ваших полотнах будет посвящена следующая глава.

8 Классы и объекты

В предыдущих главах вы освоили рисование фигур путем обращения к методу *draw()*, объявленному в Processing: каждый раз при вызове этого метода программа отрисовывает, например, прямоугольник. Однако важно понимать, что при каждом вызове прямоугольник рисуется заново, более того, самого прямоугольника в коде нет: есть только метод *rect()*, который отрисовывает прямоугольник на экране.

В этом отличие программной отрисовки от рисования на бумаге, где вы, начертив прямоугольник один раз, воспринимаете его как стабильный объект, который никуда не исчезнет, пока вы его не сотрете ластиком.

Такие действия, как вращение, перемещение мы тоже привыкли делать с объектами, однако отсутствие стабильного объекта в Processing приводит к логическому пробелу между деятельностью в реальным мире и в программировании.

Заполнить этот пробел, принести логику окружающей нас реальности в логику программирования поможет так называемый Объектно Ориентированный Подход (ООП). В нем есть возможность создавать объекты и манипулировать с ними.

8.1 Класс и его объекты

В миренас окружают объекты разной природы, или разных типов, или разных классов. Так, например, есть класс автомобилей и есть сами, конкретные машины, которые могут отличаться друг от друга, скажем, типом двигателя, но при этом принцип классификации не нарушается: обе остаются автомобилями.

Рассмотрим код Листинга 51, в котором мы отрисовываем прямоугольник. Отрисовка прямоугольника в том месте, где располагается курсор мыши, дает ощущение, что мы имеем дело со стабильной фигурой, которая перемещается по экрану, как будто мы вырезали её из бумаги.

Листинг 51: Рисуем прямоугольник в позиции курсора

```
1
2  void setup() {
3    size(600, 600);
4    smooth();
5    noStroke();
6    rectMode(CENTER);
7  }
```

```
8
9  void draw() {
10   background(255);
11   fill(50);
12   rect(mouseX, mouseY, 50, 50);
13 }
```

На самом деле никакого прямоугольника у нас нет: мы отрисовываем прямоугольник на каждом фрейме за каждый вызов метода *draw()*. Модифицируем наш код, введя в него объявление класса нашего прямоугольника.

В коде Листинга 52 мы вводим класс *FunnyRect* описания нашего прямоугольника. Объявление класса происходит с помощью ключевого слова *class*. Мы объявили класс со 2-й по 17-ю строку. В нашем классе в 3-й строке мы объявили свойства *cx, cy и fsize*, который будут отвечать за положение центра прямоугольника и за его размер. По сути, свойства класса (иногда их называют поля) – это обычные переменные.

Листинг 52: Рисуем объект FunnyRect

```
1
2  class FunnyRect {
3    float cx, cy, fsize;
4
5    void setCenter(float x, float y){
6      cx = x;
7      cy = y;
8    }
9
10   void setSize(float size){
11     fsize = size;
12   }
13
14   void render(){
15     rect(cx, cy, fsize, fsize);
16   }
17 }
18
19 FunnyRect funnyRectObj = new FunnyRect();
20
21 void setup() {
22   size(600, 600);
23   smooth();
24   noStroke();
25   rectMode(CENTER);
26   funnyRectObj.setSize(50);
27 }
28
```

```
29  void draw() {
30    background(255);
31    fill(50);
32    funnyRectObj.setCenter(mouseX, mouseY);
33    funnyRectObj.render();
34  }
```

Далее в 10-й строке мы объявили метод *setSize()*, который принимает один аргумент *float size*. Название метода и аргументы мы можем придумывать сами (при условии, что они еще не используются). Так, вполне логично назвать *setSize()* метод , который принимает аргументом размер квадрата (переменную *size*) и записывает ее значение в свойство *fsize*.

Такая же логика связана с методом *setCenter()*. Метод *setCenter()* принимает два аргумента и присваивает их значения соответствующим свойствам класса *FunnyRect*.

Логика метода *render()* немного отличается от предыдущих. Этот метод ничего не принимает – он занимается отрисовкой нашего прямоугольника.

Итак, мы описали объявление класса. Далее идет работа с объектами этого класса. По факту объекты будут создаваться в памяти компьютера, когда вы запустите программу, т.е. программист пишет только классы. В 19-й строке мы объявили свойство *funnyRectObj* – это ссылка, которая связывается с объектом класса *FunnyRect* с помощью знака равенства. Чтобы создать объект класса необходимо использовать ключевое слово *new*.

Мы не случайно назвали *funnyRectObj* свойством, хотя эта ссылка не объявлена внутри какого-то «видимого нам» класса. По сути, все что мы пишем в Processing, можно представлять как место внутри встроенного класса: как будто бы в 1-й строке кода у нас объявлен *class ProcessingApplet {* , а в 35ой строке закрывающую фигурную скобку *}*.

В 26-й строке мы уже работаем с объектом(!) класса *FunnyRect* – устанавливаем его размер; в строке 32 – координаты центра; далее, в 33-й строке, мы отрисовываем именно этот объект.

Вызов метода объекта пишется как «ссылка-точка-метод(аргументы через запятую)». Последовательность аргументов и их тип, естественно, важны. Рассмотрим создание нескольких объектов одного класса на примере кода Листинга 53.

Листинг 53: Рисуем два объекта (один из них крутится)

```
1
2  class FunnyRect {
3    float cx, cy, fsize;
4
```

```
5     void setCenter(float x, float y){
6       cx = x;
7       cy = y;
8     }
9
10    void setSize(float size){
11      fsize = size;
12    }
13
14    void render(){
15      rect(cx, cy, fsize, fsize);
16    }
17  }
18
19  FunnyRect funnyRectObj = new FunnyRect();
20  FunnyRect funnyRectObj1 = new FunnyRect();
21  float counter = 0;
22
23  void setup() {
24    size(600, 600);
25    smooth();
26    noStroke();
27    rectMode(CENTER);
28    funnyRectObj.setSize(50);
29    funnyRectObj1.setSize(20);
30  }
31
32  void draw() {
33    background(255);
34    fill(50);
35
36    float objX = mouseX + sin(counter)*150;
37    float objY = mouseY + cos(counter)*150;
38
39    funnyRectObj.setCenter(mouseX, mouseY);
40    funnyRectObj.render();
41
42    funnyRectObj1.setCenter(objX, objY);
43    funnyRectObj1.render();
44
45    counter += 0.05;
46  }
```

Код Листинга 53 немного изменился по сравнению с предыдущим. Так, в 20-й строке мы создали еще один объект класса *FunnyRect* и связали его со ссылкой *funnyRectObj1*, в 29-й строке мы установили размер второго объекта, в 42-й строке – координаты центра. Эти координаты отличаются от координат первого объекта: они формируются в 36-й и

37-й строке каждый раз заново, исходя из значений синуса и косинуса. Таким образом второй объект – второй квадрат – будет крутиться вокруг курсора мыши.

Задание 1. Измените класс FunnyRect в Листинге 53, добавив в него свойство, отвечающее за цвет. Второй объект отрисуйте другим цветом.

Следует отметить, что мы создали два объекта одного (!) класса. Они имеют одинаковый набор свойств: у каждого есть размер и положение центра. Значения этих свойств у каждого объекта свои, и изменение значений свойств одного объекта не приводит к изменению свойств второго объекта (хотя принципиальная возможность есть). Код в целом получился более длинным: увеличилось количество строк, но читать его стало проще и, главное, интуитивно понятнее.

Работа с объектами удобна, когда нам требуется совершить какое-либо действие с геометрической фигурой: если, например, мы захотим по клику мыши увеличивать размеры второго объекта, то нам придется всего лишь добавить строки (см. Листинг 54).

Листинг 54: Фрагмент кода mouseClicked

```
1
2  void mouseClicked(){
3    float currentSize = funnyRectObj1.fsize;
4    currentSize += 5;
5    funnyRectObj1.setSize(currentSize);
6  }
```

Код Листинга 54 можно добавить в Листинг 53, начиная с 47-й строки. При клике мы попадаем в метод *mouseClicked()*, в начале которого мы обращаемся к свойству *fsize* объекта *funnyRectObj1*. Если свойства объекта «открыты» (а они открыты по умолчанию), то к ним можно обращаться через «ссылка-точка-свойство». Значение свойства мы присваиваем переменной *currentSize*, увеличиваем его не 5 и затем обновленное значение устанавливаем для объекта *funnyRectObj1* с помощью вызова метода *setSize()*.

Задание 2. Добавьте код Листинга 54 в Листинге 53, проверьте работу клика мыши.

Обратите внимание, что при этом изменении логики работы приложения мы не затронули метод *draw()*. В нем как происходил, так и происходит вызов отрисовки объектов. Новая логика появилась в поведении самого объекта.

8.2 Объекты для генеративной графики

Объявляя свои собственные классы, мы получаем возможность создавать свои собственные уникальные геометрические объекты. Если раньше мы были ограничены геометрическими примитивами Processing, то теперь можем описать класс своей собственной линии. Рассмотрим код Листинга 55.

Листинг 55: Рисуем свой тип объектов

```
1
2  class MyLine {
3    float centerX, centerY, length, angle, weight;
4
5    public void render() {
6      float x1 = centerX - cos(angle)*length/2;
7      float y1 = centerY + sin(angle)*length/2;
8      float x2 = centerX + cos(angle)*length/2;
9      float y2 = centerY - sin(angle)*length/2;
10     stroke(50, 100);
11     strokeWeight(weight);
12     line(x1, y1, x2, y2);
13     strokeWeight(5);
14     stroke(50);
15     line(x2, y2, x2, y2);
16     line(x1, y1, x1, y1);
17   }
18 }
19
20 MyLine lineObj;
21 float counter;
22
23 void setup() {
24   size(500, 500);
25   smooth();
26   background(255);
27   lineObj = new MyLine();
28   lineObj.centerX = width/2;
29   lineObj.centerY = height/2;
30   lineObj.length = 350;
31   lineObj.angle = PI/4;
32   lineObj.weight = 1;
```

```
33  }
34
35  void draw() {
36    counter += 0.05;
37    if (counter > TWO_PI) {
38      counter = 0;
39    }
40    lineObj.centerX = width/2 + sin(counter)*50;
41    lineObj.centerY = width/2 + cos(counter)*50;
42    lineObj.angle = counter*2;
43    lineObj.render();
44  }
```

Результат работы Листинга 55 представлен на Рисунке 62. При запуске приложения мы видим, что линия двигается по определенной траектории. Причем линия имеет уникальную форму с круглым окончанием. Линия – это объект класса *MyLine*, который мы объявляем со 2-й по 18-ю строку.

Класс *MyLine* содержит несколько свойств и один метод *render()*. Этот метод служит для отрисовки линий. Логика отрисовки продолжает параграф «Игры с синусом и косинусом».

Объект класса *MyLine* создается в 27-й строке (справа от знака равенства) и затем с помощью знака равенства связывается со ссылкой *lineObj*, которую мы объявили в 20-й строке. После этого с 28-й по 32-ю строку мы устанавливаем значение свойств нашего объекта *lineObj*.

В методе *draw()* происходит обновление счетчика counter для того, чтобы сформировать координаты для центра линии. Центр линии устанавливается в строках 40 и 41. В 42-й строке мы устанавливаем новое значение свойства *angle* и отрисовываем линию в 43-й строке. При следующем вызове метода *draw()* мы обновим счетчик и у той же самой линии поменяем ее центр и угол.

Таким образом, мы работаем с одним объектом, т.е. с одной и той же линией: мы как бы сохраняем ее между вызовами метода *draw()*.

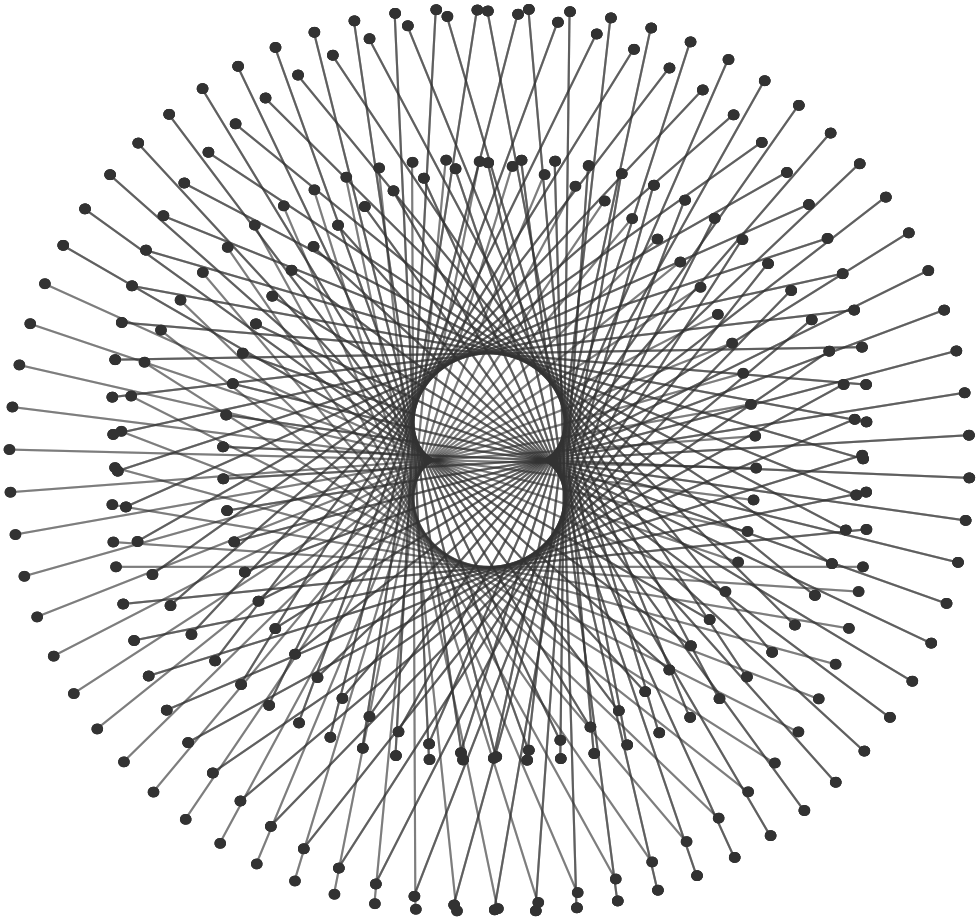


Рис. 62: Результат выполнения Листинга 55.
Генеративная графика своим типом объектов.

Изменим код Листинга 55, добавив движение от центра холста к его краям (см. Листинг 56).

Листинг 56: Рисуем более сложную графику

```
class MyLine {
  float centerX, centerY, length, angle, weight;

  public void render() {
    float x1 = centerX - cos(angle)*length/2;
    float y1 = centerY + sin(angle)*length/2;
    float x2 = centerX + cos(angle)*length/2;
    float y2 = centerY - sin(angle)*length/2;
    stroke(50, 100);
    strokeWeight(weight);
    line(x1, y1, x2, y2);
    strokeWeight(5);
    stroke(50);
    line(x2, y2, x2, y2);
    line(x1, y1, x1, y1);
  }
}

MyLine lineObj;
float counter, radius;

void setup() {
  size(500, 500);
  smooth();
  background(255);

  lineObj = new MyLine();
  lineObj.centerX = width/2;
  lineObj.centerY = height/2;
  lineObj.length = 350;
  lineObj.angle = PI/4;
  lineObj.weight = 1;
  radius = 50;
}

void draw() {
  counter += 0.05;
  if (counter > TWO_PI) {
    counter = 0;
    radius = radius + 50;
  }
  lineObj.centerX = width/2 + sin(counter)*radius;
  lineObj.centerY = height/2 + cos(counter)*radius;
  lineObj.angle = counter*2;
```

```
45      lineObj.render();
46  }
```

Результат работы Листинга 56 представлен на Рисунке 63. Мы внесли незначительные изменения: добавили свойство *radius* (строка 39) и использовали его при формировании координат центра нашей линии (строки 43 и 44). В остальном код остался прежним.

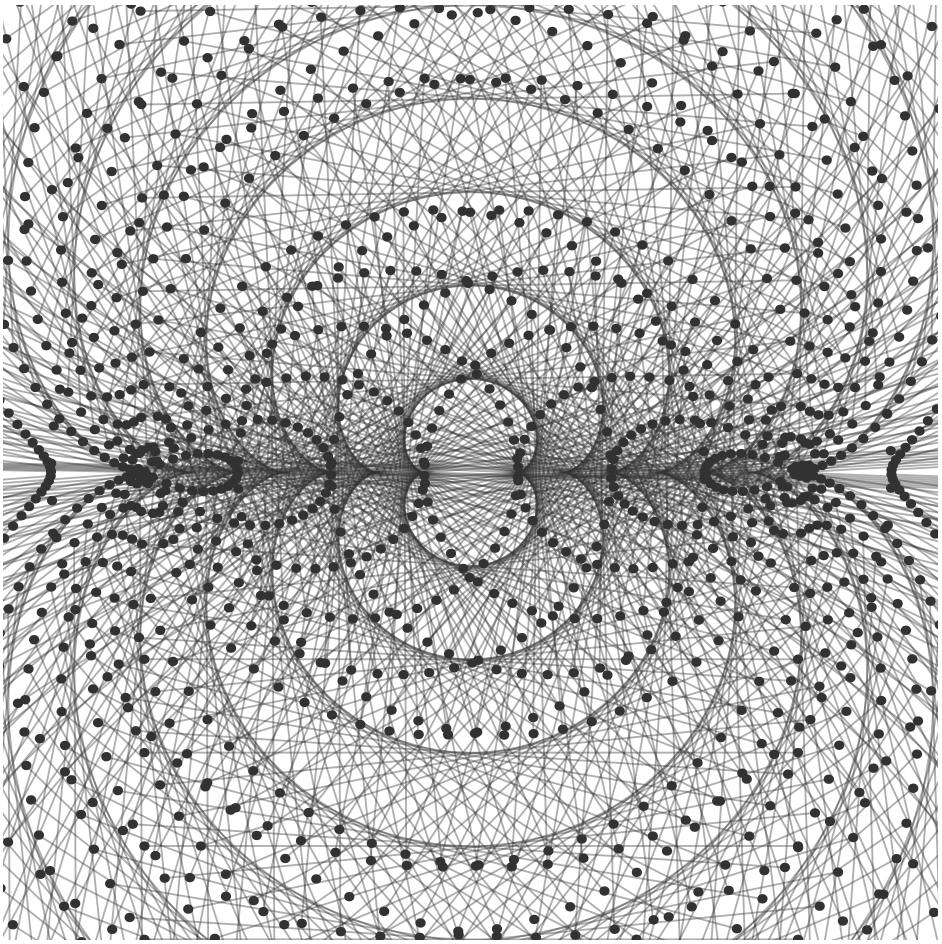


Рис. 63: Результат выполнения Листинга 56.
Более сложная генеративная графика.

Видоизменяя параметры отрисовки в коде Листинга 56, можно добиться интересных результатов, например, как на Рисунках 64 и 65. Для того чтобы получился результат Рисунка 64, необходимо включить логику работы с цветом в класс *MyLine*. Вы можете менять значения цветов применяя принципы тригонометрии.

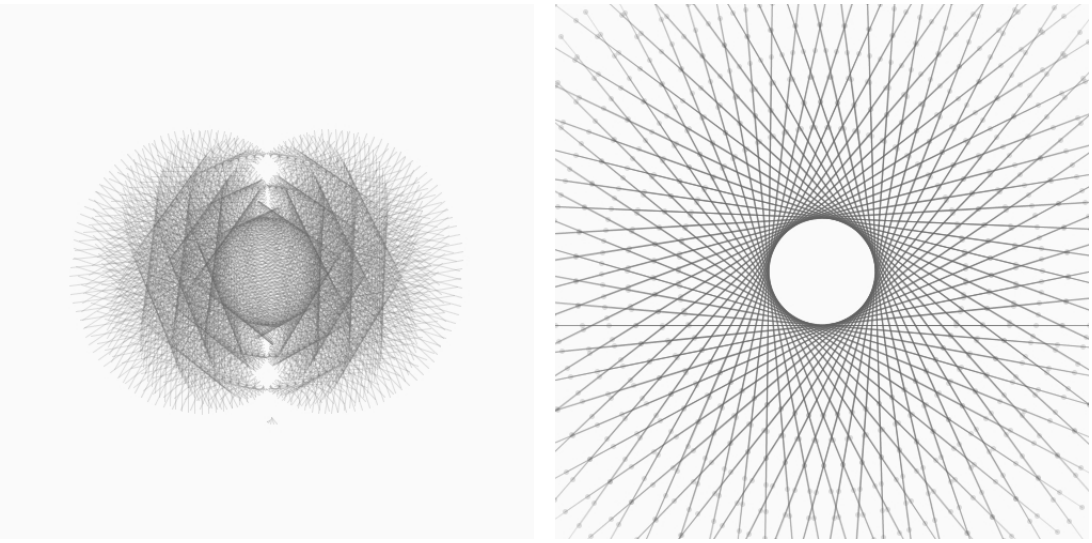


Рис. 64: Варианты генеративной графики.

Задание 3. *Измените код Листинга 56, реализовав в нем работу с цветом, таким образом, чтобы в результате у вас получилось изображение схожее с Рисунком 65.*

В этом параграфе мы познакомились с возможностями описания своих собственных классов объектов, создания самих объектов и манипулирования ими. Класс – это сущность (фрагмент кода), который определяет структуру (набор свойств) всех объектов этого класса. Все объекты одного класса должны иметь одинаковый набор методов, одинаковый набор свойств, но значения свойств могут быть различными. Объект можно представить как единый пакет данных и функциональности. Как правило, единственный путь добраться до данных (свойств) объекта – вызвать одну из предоставляемых им функций. Эти функции называются методами.

Строк программного кода получается больше, но его ясность становится выше. Взаимодействие с объектами происходит на интуитивном уровне за счет методов. При работе с Processing чаще всего придется использовать уже разработанные классы, а не создавать собственные. Однако все зависит от ваших амбиций и задач.

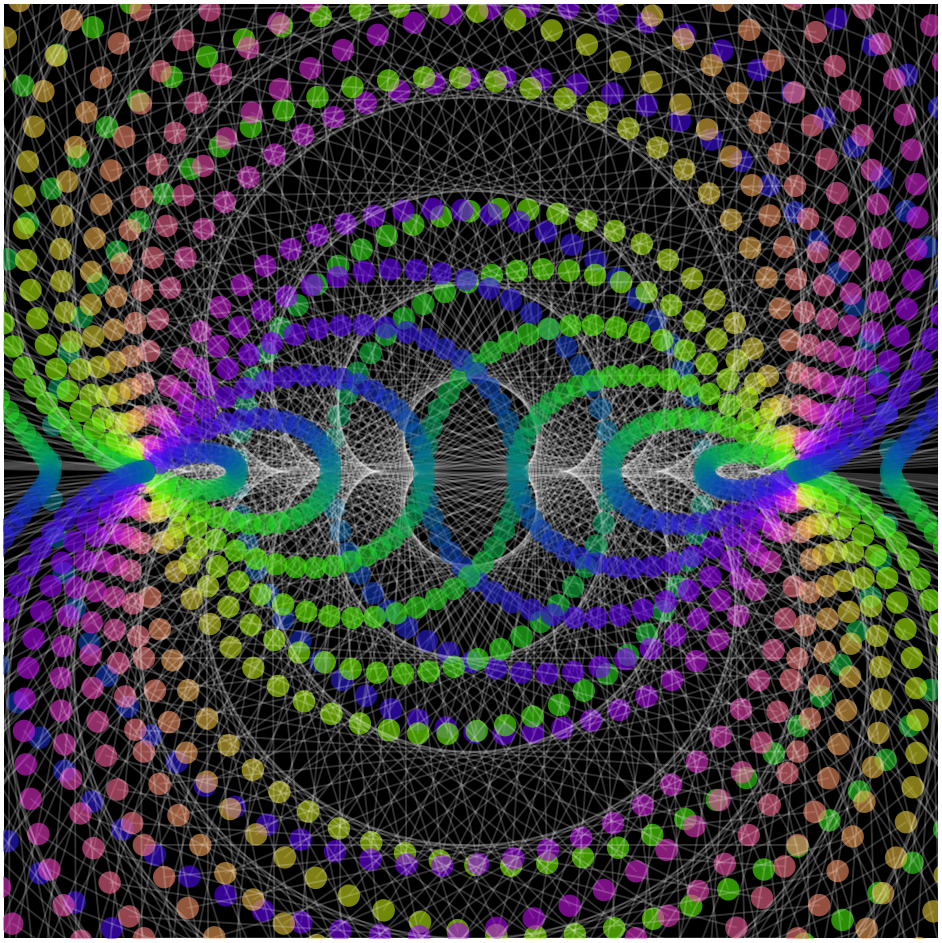


Рис. 65: Результат задания 3.
Сложная генеративная графика с цветными эллипсами на черном.

8.3 Конструктор класса и свойство this

Часто при создании объекта класса нам требуется указать определенные параметры. Например, если мы хотим создать эллипс, то сразу задаем его ширину и высоту: согласитесь, странно создавать безразмерный эллипс и только потом устанавливать его параметры. В Processing если вы определили свойство класса, то при создании объекта этого класса (когда вы пишите new), все свойства принимают нулевые значения (если не указано другое). Однако если вы задумали задать более сложную логику инициализации свойств, то вам потребуется опеределенное место в коде, где вы смогли бы реализовать эту логику.

Вы можете определять специальные методы, вызываемые всякий раз при создании объекта класса. Эти методы называются конструкторами

(*constructors*). Задача конструктора – инициализация свойств объекта нужным программисту образом, до использования этого объекта. Рассмотрим код Листинга 57.

Листинг 57: Работа с конструктором

```
1
2  class MyEllipse {
3    float centerX, centerY, angle, size, weight;
4
5    MyEllipse(float cX, float cY, float cA, float eS, float
        Ew){
6      centerX = cX;
7      centerY = cY;
8      angle = cA;
9      size = eS;
10     weight = Ew;
11   }
12
13   public void render() {
14     fill(200, size/20);
15     float x1 = centerX - cos(angle)*size/2;
16     float y1 = centerY + sin(angle)*size/2;
17
18     stroke(250, 100);
19     strokeWeight(weight);
20     ellipse(x1, y1, size, size);
21   }
22 }
23
24 MyEllipse ellipseObj;
25
26 void setup() {
27   size(500, 500);
28   smooth();
29   background(10);
30   ellipseObj = new MyEllipse(width/2, width/2, 0, 150, 1);
31 }
32
33 float counter;
34
35 void draw() {
36   counter += 0.01;
37   if (counter > TWO_PI) {
38     counter = 0;
39   }
40   ellipseObj.size = sin(counter*4f)*mouseX;
41   ellipseObj.render();
42 }
```

В Листинге 57 мы как и раньше объявляем класс со 2-й по 22-ю строку. В 3-ей строке мы объявляем набор свойств – при создании объекта они будут инициализированы нулями. Далее сработает конструктор, который мы объявили с 5-й по 11-ую строку. У конструкторов в Processing должно быть то же имя, что и у самого класса (чтобы Processing «понял», что этот метод и есть конструктор). В нашем примере конструктор принимает 5 аргументов и производит присвоение их значений соответствующим свойствам.

В 30-й строке мы создаем объект класса *MyEllipse* и передаем его конструктору требуемые аргументы. Если мы объявили конструктор, то при создании объекта класса требуется работать с ним (или с ними: можно объявить несколько конструкторов, но это уже более сложное ООП). Когда программист создает объект, вызывается его конструктор, который должен вернуть управление, прежде чем разработчик сможет выполнить над объектом другое действие. Именно поэтому при объявлении конструктора мы не пишем тип возвращаемого значения (даже не пишем *void*). Итак, в 30-й строке мы создали объект и связали его со свойством *ellipseObj*.

В методе *draw()* мы производим манипуляции с объектом *ellipseObj*, изменяя его размер в 40-й строке. В строке 41 мы отрисовываем объект.

Результат работы Листинга 57 представлен на Рисунке 67.

Задание 4. Измените код Листинга 57 так, чтобы окружности на холсте перемещались по кругу, как на Рисунке 66

В коде Листинга 57 при объявлении конструктора нам пришлось придумывать наименования аргументов, при том, что свойства класса у нас уже имеют имена, которые нас вполне устраивают. Когда аргументов и свойств много, может возникнуть путаница. Взглянем на код Листинга 58: в нем мы опеределяем только конструктор, в место объявления в Листинга 57 (все остальные строки остаются прежними, изменяются только строки с 5-й по 11-ю).

Листинг 58: Изменения в конструкторе

```
1
2  MyEllipse(float centerX, float centerY, float angle, float
       size, float weight){
3       this.centerX = centerX;
4       this.centerY = centerY;
5       this.angle = angle;
```

```
6         this.size = size;
7         this.weight = weight;
8     }
```

При таком объявлении конструктора аргументы носят наименования свойств класса. Чтобы отличить аргументы от свойств, мы используем ссылку на текущий объект класса – *this*. Не смотря на то, что объектов у нашего класса может быть сколько угодно, ссылка *this* всегда будет вести к текущему объекту (т.е. это ссылка на объект внутри самого объекта). Эта ссылка существует в каждом классе Processing, даже если вы не видите объявление класса (в стартовом поле редактирования кода Processing нет явного объявления класса, но мы уже говорили, что фактически он есть). Вернемся к этому вопросу в следующих главах.

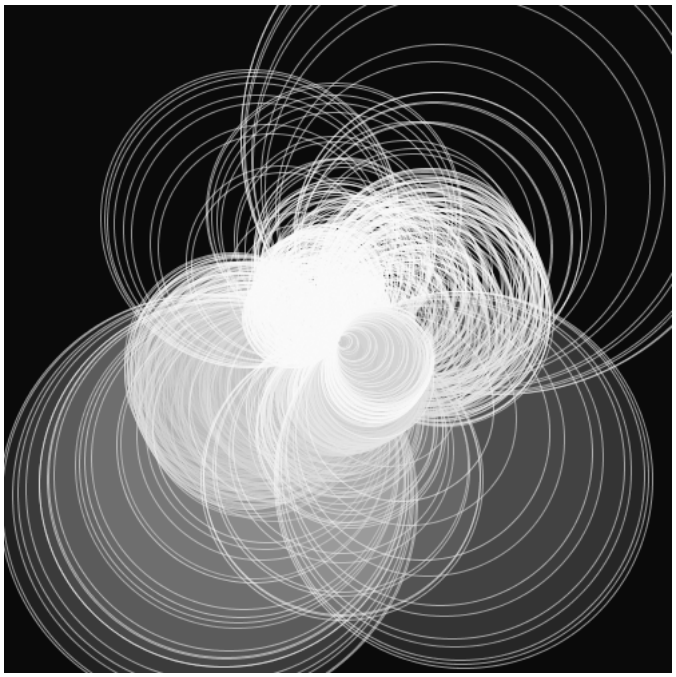


Рис. 66: Результат задания 4.
Эллипсы на черном отрисованы по кругу.

Обратите внимание, что использование ссылки *this* никак не повлияло на создание объекта класса: строка 30 кода Листинга 57 никак не поменялась. Как и раньше мы создаем объект, передавая в его конструктор 5 аргументов в определенном порядке.

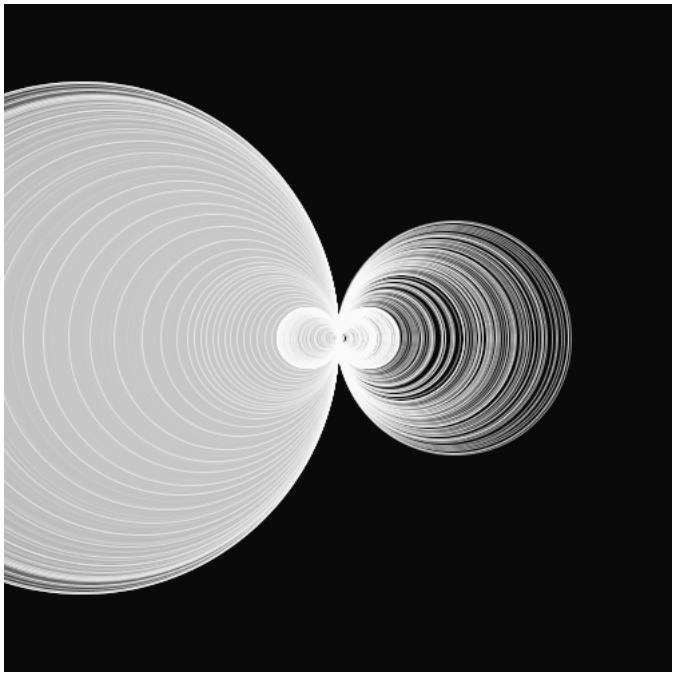


Рис. 67: Результат выполнения Листинга 57.
Эллипсы на черном. Работа с конструктором.

8.4 Взаимодействие объектов

Наличие объектов сближает логику работы программы с логикой привычного нам мира, тем более, что, как и в реальной действительности, в ООП объекты могут взаимодействовать друг с другом: это происходит с помощью методов. Мы уже знаем примеры использования методов для установки значений свойств и для вызова отрисовки. Рассмотрим код Листинга 59.

Листинг 59: Рисуем взаимодействие

```
1
2  class FunnyEllipse {
3    int myColor = 200;
4    float size, x, y;
5    boolean noise;
6
7    FunnyEllipse() {
8      size = random(50, 150);
9    }
10
11   void render() {
12     rectMode(CENTER);
```

```
13      fill(myColor, 100);
14      noStroke();
15      if(noise){
16        fill(random(150,255), 150);
17        float nx = x + noise(x)*10;
18        float ny = y + noise(y)*10;
19        ellipse(nx, ny, size, size);
20      } else {
21        ellipse(x, y, size, size);
22      }
23
24    }
25
26    boolean isLine(FunnyEllipse box) {
27      boolean result = false;
28      if (dist(x,y, box.x, box.y) < 150) {
29        result = true;
30        box.noise = true;
31      } else {
32        box.noise = false;
33      }
34      noise = result;
35      return result;
36    }
37
38    void drawLine(FunnyEllipse box) {
39      stroke(255);
40      strokeWeight(1);
41      line(x, y, box.x, box.y);
42      noStroke();
43      fill(10);
44      ellipseMode(CENTER);
45      ellipse(x, y, 10, 10);
46      ellipse(box.x, box.y, 10, 10);
47    }
48  }
49
50  FunnyEllipse obj_1, obj_2;
51  float counter = 0;
52
53  void setup() {
54    size(500, 500);
55    obj_1 = new FunnyEllipse();
56    obj_2 = new FunnyEllipse();
57  }
58
59  void draw() {
60    background(10);
61    obj_1.x = map(sin(counter),-1,1,100,width-100);
```

```
62      obj_2.x = map(cos(counter),-1,1,100,width-100);
63      obj_1.y = map(cos(counter + 0.1),-1,1,100,width-100);
64      obj_2.y = map(sin(counter - 0.1),-1,1,100,width-100);
65      obj_1.render();
66      obj_2.render();
67      if (obj_1.isLine(obj_2)) {
68        obj_1.drawLine(obj_2);
69      }
70      counter += 0.01;
71    }
```

Результат выполнения кода Листинга 59 представлен на Рисунке 68. Если программист не определяет конструктор класса, то к классу автоматически добавляется «пустой» конструктор по умолчанию – конструктор без аргументов. Заметьте, что если программист сам создает один или несколько конструкторов, то автоматического добавления конструктора без аргументов не происходит. В 7-й строке мы объявили конструктор без аргументов – он и будет вызываться при создании объектов в строках 55 и 56.

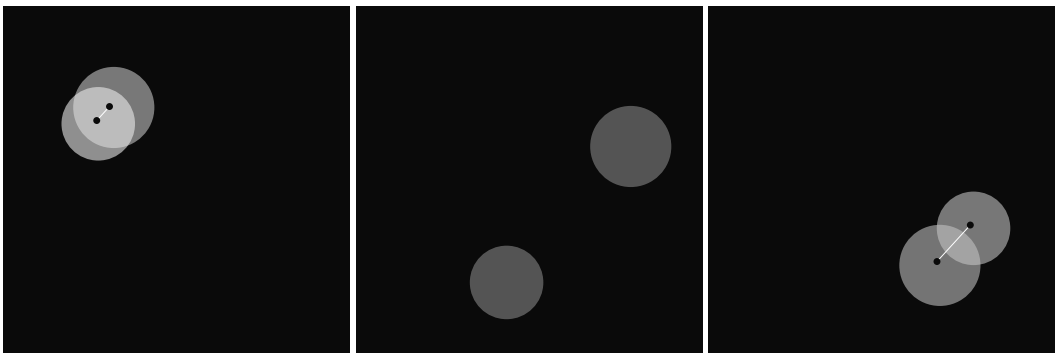


Рис. 68: Результат выполнения Листинга 59,
Фазы взаимодействия объектов

С 61-й по 64-ю строки мы формируем координаты наших объектов. Использование тригонометрических функций позволит объектам вращаться по кругу. В 65-й и 66-й строках мы отрисовываем объекты на экране, вызывая метод *render()*, который объявлен в классе *FunnyEllipse* с 11-й по 24-ю строки. Метод *render()* занимается логикой отрисовки объекта: если значение свойства *noise* – истина, то срабатывают строки 16 – 19 и объект отрисовывается с шумом и цветом; если значение свойства *noise* – ложь, то объект отрисовывается в обычном режиме.

В 67-й строке происходит вызов метода *isLine()* от объекта *obj_1*. Этот метод принимает аргументом второй объект *obj_2* того же класса

FunnyEllipse. Метод *isLine()* мы объявили с 26-й по 36-ю строки в классе *FunnyEllipse*. Он принимает объект класса *FunnyEllipse* как аргумент и работает с ним по ссылке *box*. Если текущий объект находится близко (менее чем не 150 пикселей) от центра второго объекта (который мы приняли аргументом), тогда срабатывают строки 29 и 30 (включаем флаг *noise*), если далеко – срабатывает строка 32 (выключаем флаг *noise*). Метод *isLine()* возвращает булевое значение: если нужно отрисовывать линию – метод возвращает истину (TRUE), если нет – ложь (FALSE).

Если линию между объектами отрисовывать нужно, то мы попадаем в строку 68, где вызываем метод *drawLine()*. В принципе этот метод можно вызывать от любого их двух объектов, передавая аргументом его «коллегу» – влюбом случае мы получим линию между центрами объектов.

Задание 5. Измените код Листинга 59 так, чтобы при взаимодействии объекты уменьшались.

Итак, объекты взаимодействуют: они перемещаются по кругу в разных направлениях и как только приближаются друг к другу на определенное расстояние, то между ними отрисовывается линия и объекты начинают трястись и мигать.

Задание 6. Добавьте в код Листинга 59 еще два объекта, одним из которых управляйте мышью.

В этом разделе мы рассмотрели объектно ориентированный подход к созданию приложений. Безусловно, возможность работать в программном коде с принципами логики окружающего нас мира вносит ясность и прозрачность в код. Однако объем исходного кода чаще всего увеличивается, поэтому объявление классов принято выносить в отдельные файлы и называть файл именем класса.

ООП является универсальной концепцией разработки и программирования в целом, т.е. реализация принципов ООП в языках программирования носит свою специфику. Мы прикоснулись к основам работы с классами и объектами – более подробное описание возможностей ООП в Processing остается, к сожалению за рамками этого издания.

9 Массивы

Полотна художников искусства всех направлений: от иконографической традиции до нонкомформизма сегодняшних дней – чаще всего демонстрируют многообъектные композиции(конечно, есть и исключения: например, «Черный квадрат» Каземира Малевича на картине находится «в одиночестве»). Если мы хотим работать с несколькими объектами, наборами координат или просто линейным вектором чисел, то мы можем использовать такую структуру данных, как массив.

Массив – фундаментальная конструкция – структура данных языка программирования. В массиве хранятся последовательности переменных (или ссылок) одного типа (или класса). Таким образом, массив может хранить не только цифры, но и объекты классов.

Только хранением данных мы не ограничиваемся. Мы хотим не просто хранить в массиве переменные (ссылки), а получать быстрый доступ к каждой конкретной переменной (ссылке). Такой доступ к переменной (ссылке) осуществляется посредством её номера. Переменные или ссылки, хранимые в массиве, носят название *элементов* массива.

9.1 Одномерные массивы

В одномерном массиве хранятся последовательно значения одного типа, как в спичечном коробке хранятся спички. Отличие лишь в том, что спички в коробке не пронумерованы (или они не лежат в определенный ячейках). Если у нас N переменных (ссылок), нумерацию начинаем с нуля до N–1. Рассмотрим объявление массива и работу с его использованием на примере кода Листинга 60.

Листинг 60: Отрисовываем массив эллипсов

```
1
2  int[] xCoordinate = new int[10];
3
4  void setup() {
5    size(500, 500);
6    smooth();
7    noStroke();
8    for(int i = 0; i < xCoordinate.length ; i++){
9      xCoordinate[i] = 35*i + 90;
10   }
11 }
12
13 void draw() {
14   background(50);
```

```
15    for(int coordinate : xCoordinate){
16      fill(200);
17      ellipse(coordinate, 250, 30, 30);
18      fill(0);
19      ellipse(coordinate, 250, 3, 3);
20    }
21  }
22
23  void keyPressed() {
24    if (key=='s') saveFrame("myProcessing.png");
25  }
```

В коде Листинга 60 мы объявили массив во 2-й строке. Объявление одномерного массива заключается в постановке квадратных скобок у типа переменной, причем в левой части равенства и квадратные скобоки пустые, а в правой в них указывается количество элементов массива. Если вы создаете массив, вы должны обязательно указать количество его элементов. Количество элементов массива в Processing может быть динамическим, т.е. вы можете произвести инициализацию массива случайным числом. Например, так, как показано в Листинге 61:

Листинг 61: Фрагмент кода

```
1  int k = (int)random(5,15);
2  int[] xCoordinate = new int[k];
```

В синтаксис объявления массива в Processing включается использование ключевого слова new по аналогии с созданием объектов класса. После объявления массива, например, с десятью элементами, как в нашем примере, все элементы получают нулевые значения. Таким образом, после создания массива нам требуется его заполнить.

В нашем случае мы заполняем массив координатами по оси X для того, чтобы отрисовать 10 эллипсов. Результат выполнения кода Листинга 60 представлен на Рисунке 69.

С 8-й по 10-ую строки кода Листинга 60 мы заполняем массив в цикле. Обращение к элементу массива идет по его индексу. Индекс – это целое положительное число. Если вы обратитесь к элементу с несуществующим индексом, например, превысите размерность масива, то программа при работе выдаст ошибку. В 9-й строке мы обращаемся к элементу массива с индексом i, т.е. при первой итерации цикла у переменной i есть значение 0, значит, мы обращаемся к нулевому элементу массива, при следующей итерации цикла $i = 1$ и обращение идет к первому элементу и так далее.

Вернемся в 8-ю строку, где мы указали условие выхода из цикла как *xCoordinate.length*. При создании массива мы не случайно использовали

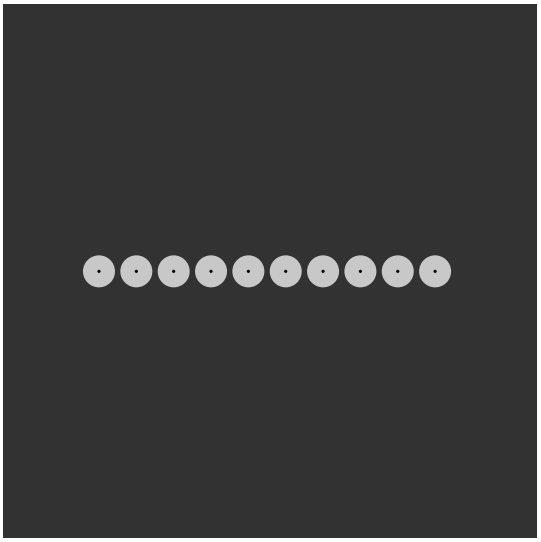


Рис. 69: Результат выполнения Листинга 60. Массив эллипсов.

ключевое слово new. Дело в том, что массив по своей сути – это объект. Но объект какого именно класса? Ведь нам могут потребоваться массивы не только *int*, но и *float* и т.д. Отдавая дань традициям программирования и с целью упрощения синтаксиса, в Processing принят только такой «[ ]» синтаксис. Поэтому *xCoordinate* в нашем примере – это не что иное, как ссылка на объект массива. А у объектов, как мы уже знаем, есть свойства и методы, которые объявлены в классе. Например, свойство *length* хранит в себе длину массива, т.е. количество его элементов.

Итак, в методе *setup()* мы заполнили массив числовыми значениями. В методе *draw()* мы должны последовательно перебрать все элементы массива или, как говорят, «обойти массив». Так как обход массива – очень популярная операция, то для нее упростили синтаксис цикла *for*: в 15-й строке мы и воспользовались упрощенной конструкцией. Читать ее можно так: каждый элемент массива *xCoordinate* последовательно копирует свое значение в переменную *coordinate*. Такое прочтение синтаксиса характерно только для массивов, элементы которого являются примитивными типами данных (*int, float, boolean* и т.п.). При первой итерации цикла мы скопировали значение нулевого элемента в переменную coordinate и отрисовали два эллипса (серый и черный), при второй итерации мы скопировали значение первого элемента массива в переменную *coordinate* и отрисовали еще два эллипса и так далее. Обойдя весь массив, мы отрисовали 10 серых эллипсов, каждый из которых содержит по одному черному эллипсу.

Рассмотрим операцию обхода массива с помощью традиционного цикла *for* на примере кода Листинга 62

Листинг 62: Рисуем массив эллипсов с увеличением размера

```
1  int[] xCoordinate = new int[10];
2
3  void setup() {
4    size(500, 500);
5    smooth();
6    noStroke();
7    for(int i = 0; i < xCoordinate.length ; i++){
8      xCoordinate[i] = 35*i + 90;
9    }
10 }
11
12 void draw() {
13   background(50);
14   for(int i = 0; i < xCoordinate.length; i++){
15     fill(200, 40);
16     ellipse(xCoordinate[i], 250, 15*i, 15*i);
17     fill(0);
18     ellipse(xCoordinate[i], 250, 3, 3);
19   }
20 }
21
22 void keyPressed() {
23   if (key=='s') saveFrame("myProcessing.png");
24 }
```

Результат выполнения кода Листинга 62 представлен на Рисунке 70.

Объявление массива, его инициализация и заполнение происходят так же, как и в коде Листинга 60. Обход массива происходит в классическом цикле *for* с 14-й по 19-ую строку. При отрисовке серого эллипса мы изменяем его размер в зависимости от индекса элемента массива.

Задание 1. Измените код Листинга 62, добавив в него еще один массив так, чтобы на холсте отрисовывалось изображение, как на Рисунке 71.

Обход массива возможен с использованием любого вида цикла. С каким бы типом массива вы ни работали, идентификатор массива в действительности представляет собой ссылку на объект, созданный в динамической памяти. И чаще всего вам придется проверять длину массива,

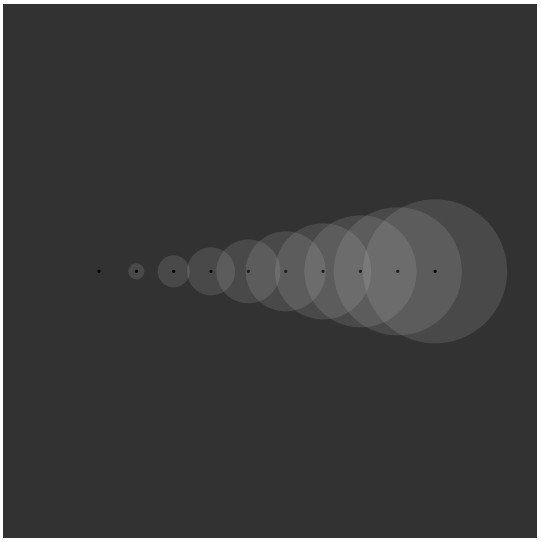


Рис. 70: Результат выполнения Листинга 62.
Массив эллипсов с увеличением размера.

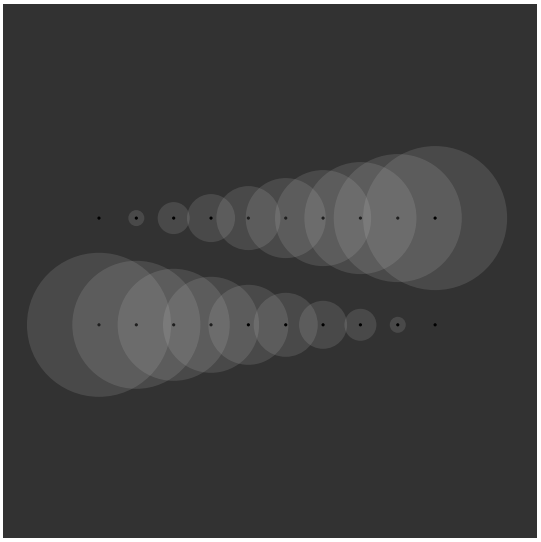


Рис. 71: Результат выполнения задания 1.
2 массива эллипсов с изменением размеров.

как это мы уже делали посредством доступного только для чтения свойства *length*, которое указывает, сколько элементов может храниться в объекте массива. Как и прежде, весь доступ к объекту массива ограничивается синтаксисом квадратных скобок *[ ]*.

Массивы объектов содержат ссылки, а массивы примитивов содержат сами примитивные значения. Значения элементов массива могут быть не известными для вас, и, чтобы их посмотреть, можно использовать распечатку в консоли, как показано на примере кода Листинга 63.

Листинг 63: Рисуем облако эллипсов

```
1
2  int[] xCoordinate = new int[30];
3
4  void setup() {
5    size(500, 500);
6    smooth();
7    noStroke();
8    myInit();
9  }
10
11 void myInit(){
12   println("New coordinates: ");
13   for (int i = 0; i < xCoordinate.length; i++) {
14     xCoordinate[i] = 250 + (int)random(-100,100);
15     println(xCoordinate[i]);
16   }
17 }
18
19 void draw() {
20   background(30);
21   for (int i = 0; i < xCoordinate.length; i++) {
22     fill(20);
23     ellipse(xCoordinate[i], 250, 5, 5);
24     fill(250, 40);
25     ellipse(xCoordinate[i], 250, 10*i, 10*i);
26   }
27   if(mouseX > 250){
28     myInit();
29   }
30 }
31
32 void keyPressed() {
33   if (key=='s') saveFrame("myProcessing.png");
34 }
```

В коде Листинга 63 мы вынесли заполнение массива в отдельный метод *myInit()*, чтобы не засорять метод *setup()*. В 12-й строке мы распечатываем в консоль текст, тем самым контролируя, в каком месте сейчас находится выполнение программы. Далее с 13-й по 16-ю строки идет заполнение массива *xCoordinate* случайными числами. Каждое число распечатывается в консоли в строке 15.

Если курсор мыши попадает в правую половину окна приложения (строка 27), то вызывается снова и снова метод *myInit()*. Происходит заполнение массива новыми значениями и мы отрисовываем уже обновленные эллипсы. В этом случае работа с массивом даже примитивных типо абсолютно такая же, как в примерах, которые мы рассматривали при работе с объектами. Мы как бы сохраняем от фрейма к фрейму (от вызова метода *draw()* до следующего *draw()*) состояние объекта, а в нашем случае под состоянием мы понимаем его координаты.

Результат выполнения кода Листинга 63 представлен на Рисунках 72 и 73.

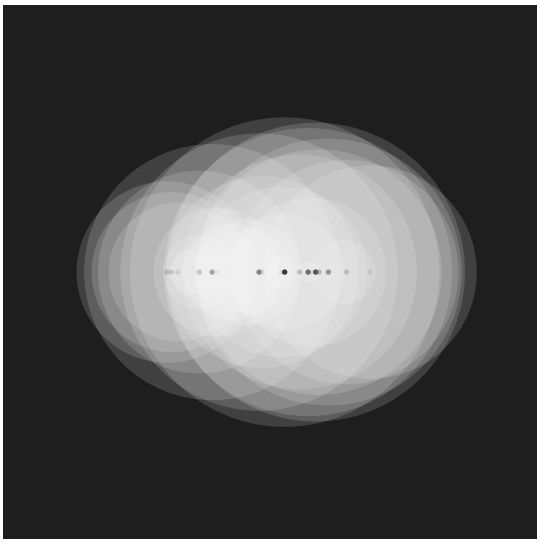


Рис. 72: Резульат выполнения Листинга 63.
Рисуем облако эллипсов.

Задание 2. Измените код Листинга 63. Так, что бы найти, отрисовать и распечатать в консоль элемент(и его индекс) с максимальным значением координаты x.

Еще один пример использования одномерных массивов идет в комплекте с processing IDE. Откроем проект из примеров: *File -> Examples...* и далее: *Basics -> Input -> StoringInput*. Код примера показан в Листинге 64.

Рассмотрим код Листинга 64 подробнее. В 12-й и 13-й строках объявляются массивы для хранения элементов типа *float*. Эти массивы, как

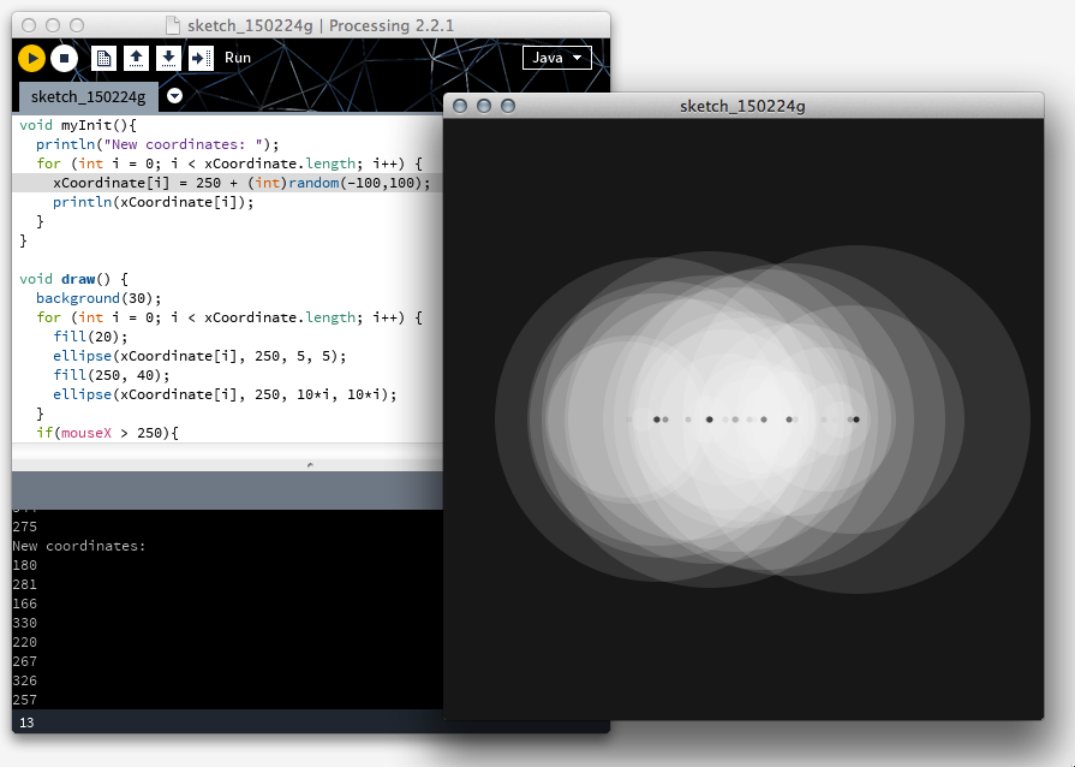


Рис. 73: Результат выполнения Листинга 63.
Снимок экрана и облако эллипсов.

можно догадаться из имен ссылок, с которыми они связаны, будут использоваться для хранения координат X и Y.

Листинг 64: Storing Input

```
1  /**
2   * Storing Input.
3   *
4   * Move the mouse across the screen to change the position
5   * of the circles. The positions of the mouse are recorded
6   * into an array and played back every frame. Between each
7   * frame, the newest value are added to the end of each
        array
8   * and the oldest value is deleted.
9   */
10
11  int num = 60;
12  float mx[] = new float[num];
13  float my[] = new float[num];
14
```

```
15  void setup() {
16    size(640, 360);
17    noStroke();
18    fill(255, 153);
19  }
20
21  void draw() {
22    background(51);
23
24    // Cycle through the array, using a different entry on
          each frame.
25    // Using modulo (%) like this is faster than moving all
          the values over.
26    int which = frameCount % num;
27    mx[which] = mouseX;
28    my[which] = mouseY;
29
30    for (int i = 0; i < num; i++) {
31      // which+1 is the smallest (the oldest in the array)
32      int index = (which+1 + i) % num;
33      ellipse(mx[index], my[index], i, i);
34    }
35  }
```

В строке 26 мы определяем переменную *which*. Ей присваивается значение остатка от деления текущего номера фрейма на количество элементов массива. Идея состоит в следующем: в каждый фрейм (при каждом вызове метода *draw()*) мы записываем координаты курсора мыши в элемент массива. Таким образом мы как бы сохраняем в памяти последние шестьдесят положений курсора, чтобы отрисовать в этих положениях эллипсы так, как показано на Рисунке 74.

Для того, чтобы отрисовать эллипсы по координатам массива, нам требуется обойти массив. Обход массива будем проводить с помощью цикла *for*, который начинается в 30-й строке. Первый (не по индексу) эллипс для отрисовки должен быть самым маленьким. Причем, индекс элемента массива с координатой этого эллипса не всегда будет 0. Например, если текущий индекс равен 5 (т.е. переменная $which = 5$), то эллипс с координатами из 6-го элемента массива должен быть самый маленький, а значит, мы должны отрисовать его первым, если хотим задавать его размер переменной цикла i (см. строку 33). Итак, первый индекс для отрисовки будет *which+1*, второй – *which+2*, в итоге получается *which+1+i* (если i считается от нуля). Чтобы не выйти за пределы массива, мы снова используем операцию остатка от деления: как только мы дойдем до последнего отрисованного индекса, мы перескакиваем на начало массива (на *0*) и продолжаем отрисовку до тех пор, пока не отрисуем последний,

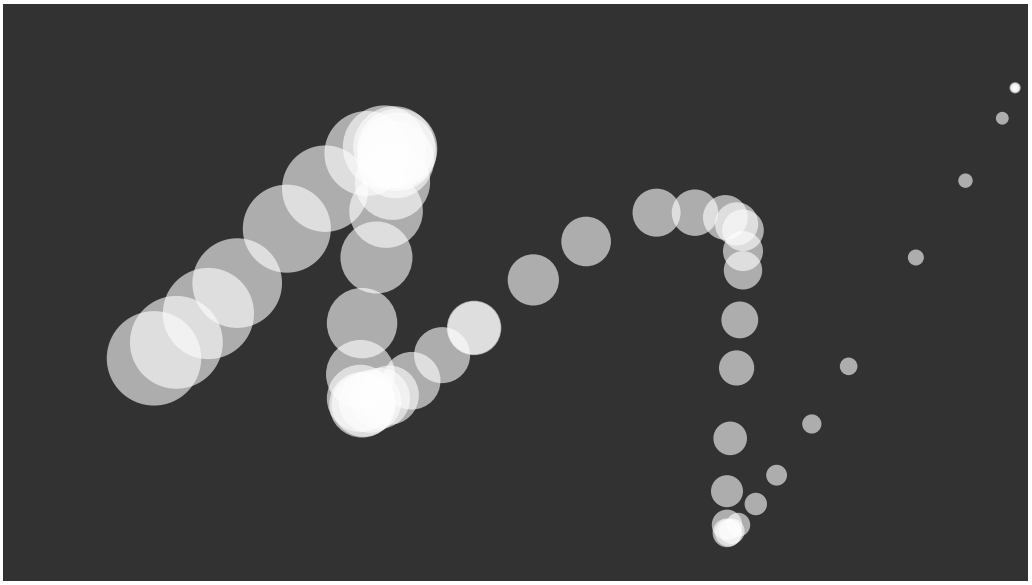


Рис. 74: Результат выполнения Листинга 64. Storing Input.

самый большой эллипс с текущим индексом *which*.

Код Листинга 64 кажется совсем небольшим, но он потребовал достаточно подробного объяснения. Так что не стоит думать, что меньший по объему код обязательно проще!

Итак, мы рассмотрели одномерные массивы. Несмотря на то, что одномерные массивы используются очень часто, в последнем примере мы были вынуждены создать два массива для хранения координат, по сути, одних и тех же точек. Рассмотрим, как поступать в таких случаях.

9.2 Многомерные массивы

Многомерные массивы часто используются для более ясного представления данных реального мира в компьютерных программах. Двумерные массивы, например, могут отражать табличные данные или данные о координатах точки в двумерном пространстве.

В Processing нет многомерных массивов, а есть массивы, базовыми типами которых являются также массивы. Например, тип *int[]* означает «массив чисел», а *int[][]* означает «массив массивов чисел». Отдавая дань традиции программирования, мы будем использовать термин «двумерный массив».

Рассмотрим на примере кода Листинга 65 работу с двумерным массивом.

Листинг 65: Рисуем эллипсы из многомерного массива

```
1
2  int numberOfellipse = 10;
3  int step = 50;
4  int[][] a = {
5    {50,133,40},
6    {132,87,90},
7    {12,287,10},
8    {300,407,30},
9    {232,187,190},
10   {448,300,79},
11   {460,450,32}
12 };
13
14 void setup() {
15   size(500, 500);
16   smooth();
17   noStroke();
18
19 }
20
21 float counter = 0;
22 void draw(){
23   background(127);
24   for(int i = 0; i < a.length; i++){
25     float angle = counter/(float)(1+i);
26     float myC = map(sin(angle), -1, 1, 0, 255);
27     fill(myC);
28     ellipse(a[i][0], a[i][1], a[i][2], a[i][2]);
29     counter+=0.01;
30   }
31 }
```

В 4-й строке мы объявили ссылку *a* на двумерный массив целых чисел. Для создания массивов можно использовать *инициализаторы* – в этом случае применяется столько вложенных фигурных скобок, сколько требуется. С 5-й по 11-ю строку мы записали семь групп цифр. В каждой группе по три цифры. Такой тип инициализации хорош, когда у нас есть значения элементов массива.

Результат выполнения кода Листинга 65 представлен на Рисунке 75. Выражение a.length имеет значение 7 – длину массива массивов. С помощью выражений *a[i].length* можно узнать длину каждого вложенного массива чисел, при условии, что *i* не отрицательно и меньше *a.length*, а также *a[i]* не равно *null* – иначе будут возникать ошибки во время выполнения программы (в текущем примере нам это не понадобилось).

Обход массива мы начинаем в 24-й строке с помощью цикла *for*. В

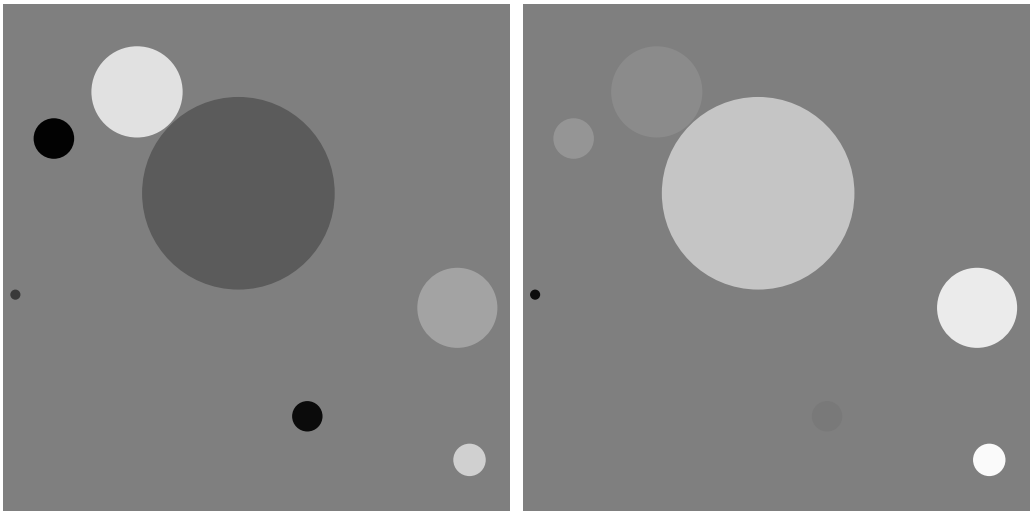


Рис. 75: Результат выполнения Листинга 65.
Эллипсы из многомерного массива.

28-й строке мы обращаемся к элементам массива. Элемент с индексом *0* текущего массива i будет отвечать за координату X, элемент с индексом 2 – за Y, и последний элемент – за размеры эллипса. Цвет эллипса меняется по синусоидальному закону в зависимости от счетчика counter и текущего значения переменной i.

Все рассмотренные примеры и утверждения одинаково верны для многомерных массивов, основанных как на примитивных, так и на ссылочных типах.

Рассмотрим создание массива с помощью конструкции *double[][] a = new double[M][N];* на примере кода Листинга 66. В нем мы хотим отрисовывать эллипсы на определенных местах, цвет отрисовки и размер будет зависеть от положения курсора мыши.

Листинг 66: Рисуем эллипсы и управляем ими мышью

```
1
2  float [][] a = new float[500][2];
3
4  void setup() {
5    size(500, 500);
6    for(int i = 0; i < a.length; i++){
7      for(int j = 0; j < a[i].length; j++){
8        a[i][j] = random(10,490);
9      }
10   }
11 }
```

```
12
13  void draw() {
14    smooth();
15    noStroke();
16    background(0);
17
18    for(int i = 0; i < a.length; i++){
19        float eDist = dist(mouseX, mouseY, a[i][0], a[i][1]);
20        float eSize = map(eDist, 0, 200, 5, 100);
21        float eColor = map(eDist, 0, 200, 50, 255);
22        fill(eColor, 200);
23
24        float cx = noise(mouseX)*10 + a[i][0];
25        float cy = noise(mouseY)*10 + a[i][1];
26
27        ellipse(cx, cy, eSize, eSize);
28    }
29  }
```

Результаты работы кода Листинга 66 в зависимости от количества элементов массива, представлены на Рисунках 76 и 77.

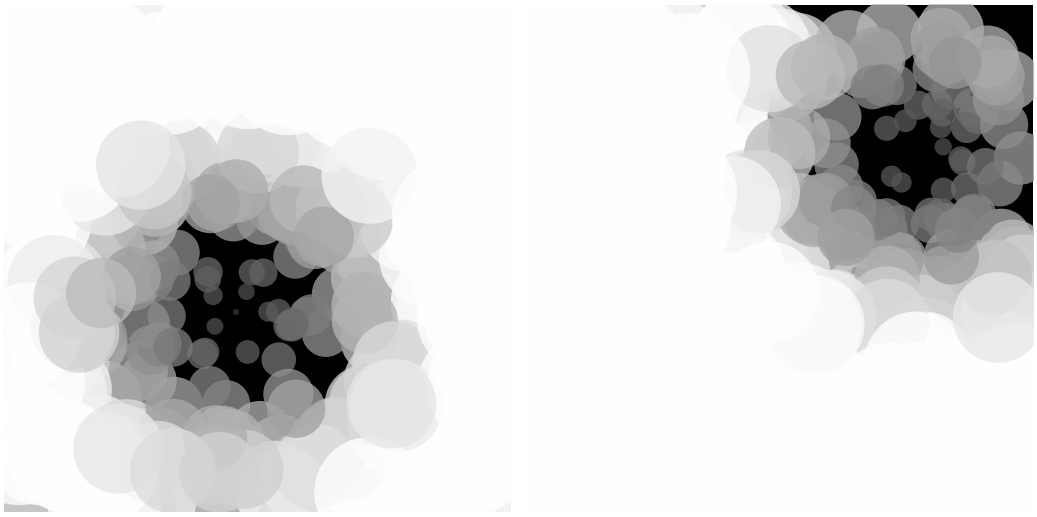


Рис. 76: Результат выполнения Листинга 66.
Фазы работы с эллипсами мышью.

Во 2-й строке кода Листинга 66 происходит инициализация массива и связывание его со ссылкой *a*. Такая инициализация предполагает указание количества элементов в общем массиве (массиве массивов) и во внутреннем. В нашем случае мы указали, что у нас будет пятьсот элементов, каждый из который будет представлять собой массив с двумя элементами типа *float*.

Формально допустимо создавать и такое: *double[][][][][][][][] foo = new double[N][2][2][3][3];* но очевидно, что человеку работать с этим будет очень сложно.

С 6-й по 9-ю строку происходит заполнение массива. Мы обходим массив a с помощью двух циклов *for*. В строке 6 мы ограничиваемся длиной общего массива *a.length* (500), во внутреннем цикле – длиной частного массива *a[i].length* (2). Чтобы обратиться к элементу массива, достаточно использовать квадратные скобки, как показано в строке 8. После заполнения массива в нем лежат случайные пары цифр – координаты отрисовки эллипсов.

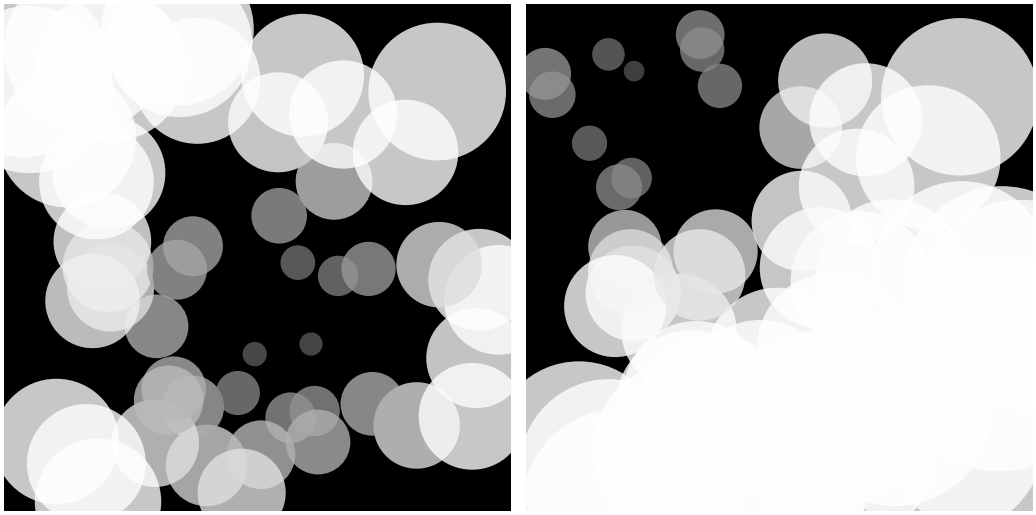


Рис. 77: Результат выполнения Листинга 66.
Фазы работы с эллипсами мышью (2).

Обход массива для отрисовки эллипсов начинается в 18-й строке. Мы используем один цикл *for*, а для доступа к элементам внутреннего массива мы используем значения индексов напрямую, например, обращения *a[i][0]* и *a[i][1]*. В 19-й строке мы определяем расстояние между курсором мыши и координатой отрисовки эллипса.

Зная расстояние между курсором и текущим эллипсом, мы формируем его размер в 20-й строке. По той же логике далее формируется цвет заливки.

В 24-й и 25-й строке мы определяем координаты для отрисовки. Использование метода *noise()* дает интересный динамический эффект.

Задание 3. Измените код Листинга 66 так, чтобы вместо эллипсов отрисовывались линии (отрезки) с направлением в сторону положения курсора мыши.

Мы рассмотрели работу с многомерными массивами на примере работы с двумерными. До сих пор мы работали с массивами, которые можно представить как прямоугольные таблички данных. В следующем параграфе вы увидите, что на самом деле массивы куда более интересны.

9.3 Визуализация массивов

Если создать двумерный массив и связать его со ссылкой *a*, то, используя эту ссылку и два числа, каждое в паре квадратных скобок (например, *a[0] [0]*), можно обратиться к любому элементу двумерного массива. Но в то же время, используя *a* и одно число в паре квадратных скобок, можно обратиться к одномерному массиву, который является элементом двумерного массива. Его можно инициализировать новым массивом с другой длиной и таблица перестанет быть прямоугольной – она примет произвольную форму. В частности, можно одному из одномерных массивов присвоить даже значение *null*.

Рассмотрим фрагмент кода, представленный в Листинге **??**. В 1-й строке мы создаем массив 3x5 и связываем его со ссылкой *a*. Во второй строке мы можем связать со ссылкой *a[0]* объект массива другой длины, например, из семи элементов.

Листинг 67: Фрагмент кода

```
1  int a[][]=new int[3][5];
2  a[0]=new int[7];
3  a[1]=new int[0];
4  a[2]=null;
```

Таким образом, прямоугольная таблица, которая первоначальна была прообразом двумерного массива, исчезает. В 3-й строке следующий элемент мы связываем с пустым массивом, а последний элемент (*a[2]*) отвязываем от всех объектов. После таких операций массив, на который ссылается переменная *a*, назвать прямоугольным никак нельзя. Зато хорошо видно, что это просто набор одномерных массивов или «пустых ссылок» (значений *null*).

Помните, что при создании многомерных массивов с помощью *new* необходимо указывать все пары квадратных скобок, соответственно ко-

личеству измерений. Но заполненной обязательно должна быть лишь крайняя левая пара: это значение задаст длину верхнего массива массивов. Если заполнить следующую пару, то этот массив заполнится не значениями по умолчанию *null*, а новыми созданными массивами с меньшей на единицу размерностью. Если заполнена вторая пара скобок, то можно заполнить третью, и так далее.

Чтобы увидеть, как выглядят рассмотренные нами структуры данных, их необходимо визуализировать, однако, как мы уже отметили, табличная форма визуализации будет не самой уместной. Рассмотрим пример кода Листинга 68.

Листинг 68: Визуализируем массивы

```
1
2  private int[][] a = new int[10][];
3  private int step = 30;
4
5  void setup() {
6    size(500, 500);
7    smooth();
8    noStroke();
9    myInit();
10 }
11
12 void myInit(){
13   for (int i = 0; i < a.length; i++) {
14     a[i] = new int[(int)random(0, 10)];
15     for (int j = 0; j < a[i].length; j++) {
16       a[i][j] = (int)random(0, 30);
17     }
18   }
19 }
20
21 void draw() {
22   fill(180, 50);
23   background(10);
24   for (int i = 0; i < a.length; i++) {
25     for (int j = 0; j < a[i].length; j++) {
26       stroke(100);
27       strokeWeight(1);
28       fill(50);
29       rect(i*step + 100, j*step + 100, step, step);
30       noStroke();
31       fill(250, 90);
32       ellipse(i*step+115, j*step+115, a[i][j], a[i][j]);
33     }
34   }
```

```
35  }
36
37  void mouseClicked(){
38    myInit();
39  }
```

Результат выполнения кода Листинга 68 представлен на Рисунке 78. Во 2-й строке мы создаем объект массива с десятью ссылками на пустые объекты массивов для хранения целочисленных значений (*int*). Длину внутренних массивов мы не знаем, поэтому пока она не указана.

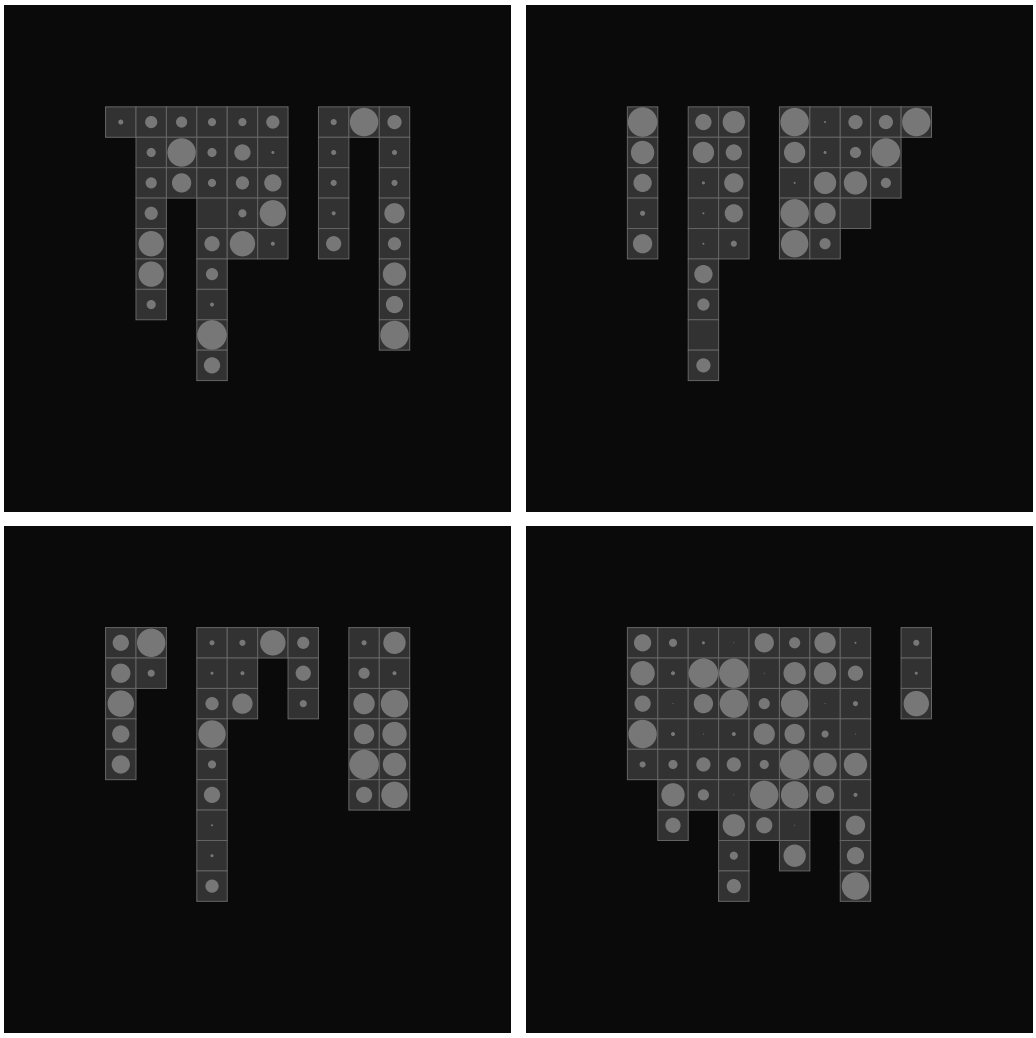


Рис. 78: Результат выполнения Листинга 68.
Визуализация массивов.

С 12-й по 18-ую строки мы объявляем метод *myInit()*, в котором про-

водим заполнение массивов. В 14-й строке мы создаем объект одномерного массива, причем его длина задается случайным образом от 0 до 10. В 16-й строке записываем целочисленное значение в элемент только что созданного массива.

Обратите внимание, что если метод *myInit()* вызвать еще раз, то массив *a* (объемлющий) получит новые объекты внутренних массивов, а старые объекты безвозвратно исчезнут. Метод *myInit()* мы вызываем, как в 9-й строке, внутри метода *setup()*, так и по нажатию мыши, в 38-й строке.

После заполнения массива мы попадаем в работу метода *draw()*. В нем мы обходим двумерный массив привычным способом вложенных циклов *for*. В 29-й строке мы отрисовываем квадрат, который символизирует ячейку элемента. Квадрат имеет одинаковые размеры для каждого элемента, за которые отвечает свойство *step* (строки 3 и 29). В 32-й строке мы отрисовываем эллипс поверх своего квадрата. Размер эллипса формируем в зависимости от значения элемента массива.

Задание 4. Измените код Листинга 68 так, чтобы визуализация данных многомерного массива подчеркивала, что массив не является прямоугольной таблицей. Например, как на Рисунке 79.

Таким образом, при визуализации массива мы показали и его структуру в виде квадратов, и значение его элементов, от которых зависят размеры эллипсов. У нас еще осталась величина цвета, которую мы не использовали в отображении данных, так что можете сами подумать на эту тему.

9.4 Массивы объектов

Массивы объектов представляют особый интерес для реализации любых художественных замыслов, ведь даже мазки кисти можно рассматривать, как объекты, а произведение – массивом этих объектов. Обратите внимание на массив объектов класса *MySuperBall*, который мы объявили в коде Листинга 69.

Класс *MySuperBall* представляет своим объектам ссылку на предыдущий объект такого же класса. Объекты у нас будут в массиве. В 4-й строке кода объявляется ссылка *MySuperBall previousBall*, которую мы будем использовать для того, чтобы связать с предыдущим объектом массива. Как видите, свойствами класса могут быть ссылки на объекты любых классов, в том числе и ссылки на объекты этого же класса. В 3-й

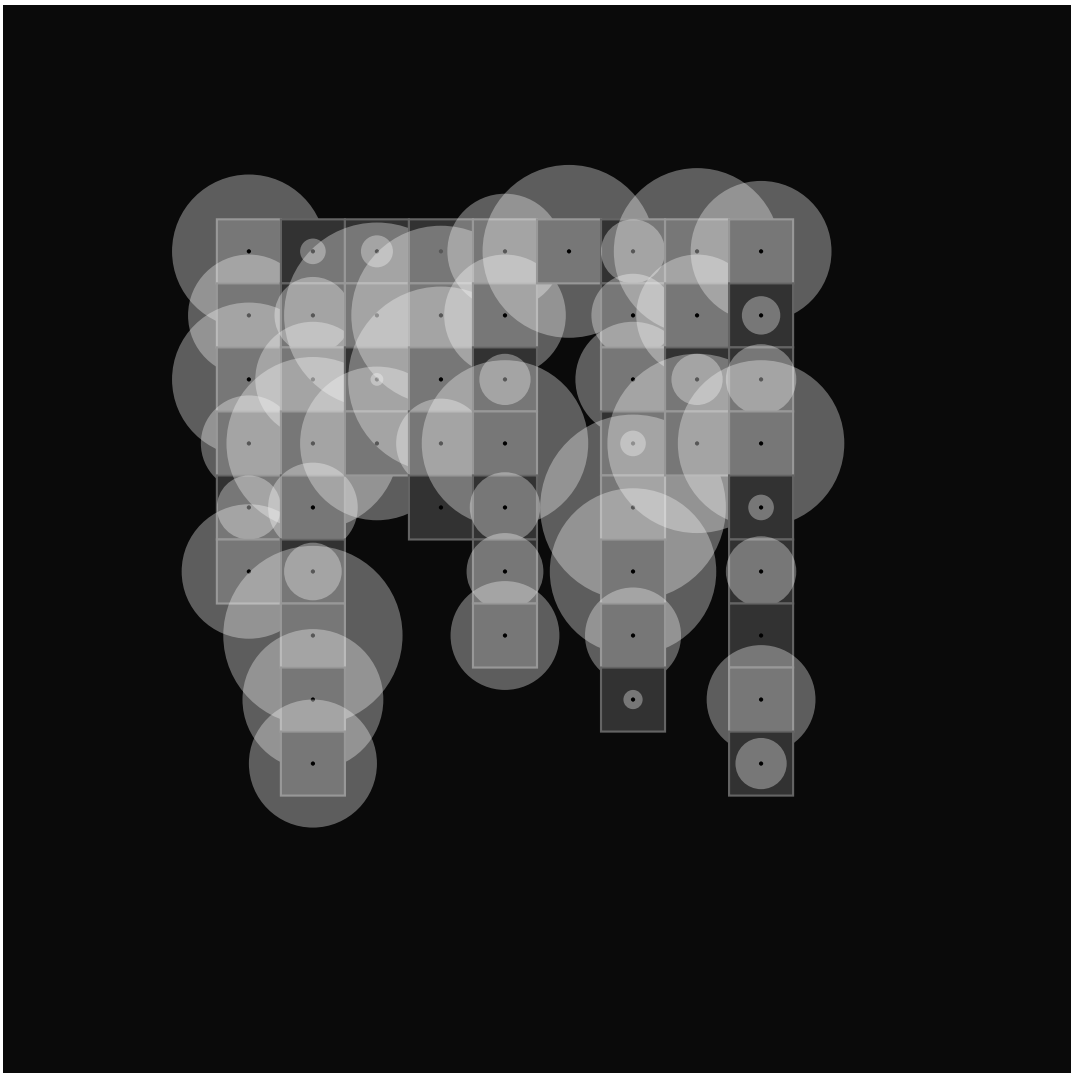


Рис. 79: Результат выполнения задания 4.
Дизайнерская визуализация массива.

строке мы объявляем свойства *x*, *y*, *radius* и *speed*, для хранения в них координат, размера и величины скорости. В 5-й строке мы объявляем целочисленное свойство *vector*. Мы будем использовать его для изменения направления движения.

Движение объектов *MySuperBall* подчиняется свойству скорости (*speed*) и вектору направления (*vector*). Наши объекты будут двигаться только вертикально. Метод *upDate()*, объявленный в строках 20–25 отвечает за изменение координат в зависимости от указанных параметров. Величина координаты *Y* растет линейно (строка 21), но когда координата *Y* стремится покинуть экран вверх или вниз, то свойству *vector* мы меняем знак (строка 23). Таким образом объект удерживается в пределах от 0 до 500.

Метод *render()*, который мы объявляем в классе *MySuperBall* с 6-й по18-ю строку, обеспечивает отрисовку объекта на экране. Объект будет отрисован как эллипс (9-ая строка) в точке с координатами *X, Y* и радиусом *radius*. Эти три переменные являются свойствами класса *MySuperBall*. В 12-й строке мы проверяем, не является ли ссылка на предыдущий объект пустой. Если ссылка не пустая, то мы попадаем в 13-ю строку, где отрисовываем линию между центром текущего объекта и предыдущего. В заключение мы отрисовываем небольшой черный эллипс в центре текущего объекта. На этом описание класса *MySuperBall* заканчивается.

Листинг 69: Создаем MySuperBall

```
1
2  class MySuperBall{
3    float x,y,radius,speed;
4    MySuperBall previousBall;
5    int vector = 1;
6    void render(){
7      noStroke();
8      fill(200, 100);
9      ellipse(x,y,radius,radius);
10     stroke(10);
11     strokeWeight(2);
12     if(previousBall != null){
13       line(x,y,previousBall.x,previousBall.y);
14       noStroke();
15     }
16     fill(0);
17     ellipse(x,y,6,6);
18   }
19
20   void upDate(){
```

```
21      y = y + speed*vector;
22      if(y > 500 || y < 0){
23        vector = vector*(-1);
24      }
25    }
26  }
27
28  int step = 30;
29  MySuperBall[] ballArray_one, ballArray_two;
30
31  void setup() {
32    size(500, 500);
33    smooth();
34    myInit();
35  }
36
37  void myInit(){
38    ballArray_one = new MySuperBall[15];
39    for(int i = 0; i < ballArray_one.length; i++){
40      MySuperBall tmp_obj = new MySuperBall();
41      float variable = random(5,30);
42      tmp_obj.x = variable + step*i + 20;
43      tmp_obj.y = random(-100,100) + 100;
44      tmp_obj.radius = variable*2 + 10;
45      tmp_obj.speed = random(0.2, 10);
46      if(i > 0){
47        tmp_obj.previousBall = ballArray_one[i-1];
48      }
49      ballArray_one[i] = tmp_obj;
50    }
51    ballArray_two = ballArray_one;
52  }
53
54  void draw(){
55    background(50);
56    for(MySuperBall curentBall: ballArray_one){
57      curentBall.upDate();
58      curentBall.render();
59    }
60  }
61
62  void keyPressed() {
63    if (key=='a') myInit();
64    if (key=='q') ballArray_two[0].radius = 300;
65  }
```

После того как класс наших объектов описан, мы переходим к работе с ними. В 29-й строке мы создаем две ссылки на массив объектов класса *MySuperBall*. Синтаксис создания ссылок на массив объектов по-

хож на тот, который используется при работе с массивами примитивных типов: те же квадратные скобки *[ ]*. Вся работа будет вестись по ссылке *ballArray_ one*, а ссылку *ballArray_ two* мы будем использовать для демонстрации одной особенности ООП, о которой подробно расскажем далее.

Всю работу по заполнению массива мы вынесли в метод *myInit()* и объявили его с 37-й по 52-ю строки. В 38-й строке мы создаем массив из 15 объектов. При этом синтаксис остается прежним: мы указываем количество элементов массива в квадратных скобках, затем в 39-й строке начинаем цикл *for* для обхода и заполнения массива.

В 40-й строке создаем объект класса *MySuperBall*, связываем его со ссылкой *tmp_ obj*. Далее по этой ссылке мы устанавливаем объекту его свойства: в 42-й и 43-й строке – координаты X, Y ; в 44-й – размер; в 45-й – значение скорости. В 46-й строке мы проверяем, не является ли наш объект первым (т.е. с нулевым индексом). Если он не первый, то связываем его свойство *tmp_ obj.previousBall* с предыдущим объектом – элементом массива с индексом *i-1* (строка 47).

Как только мы настроили объект, т.е. заполнили свойства, которые нам требуются (в принципе не обязательно заполнять абсолютно все свойства объекта), мы связываем его с соответствующей ссылкой – iым элементом массива *ballArray_ one*, в строке 49.

После обхода всех пятнадцати элементов массива мы попадаем в строку 51, где связываем ссылку *ballArray_ one* со ссылкой *ballArray_ two* (используем операцию присваивания или связывания, т.е. там, где участвует знак равенства, читается справа налево). Эта связка поможет нам наглядно показать, что здесь не происходит копирования объектов, а к одному объекту (массиву объектов класса *MySuperBall*) теперь привязаны две ссылки, и работать с этим объектом можно как через ссылку *ballArray_ one*, так и через ссылку *ballArray_ two* – такую работу мы будем проводить в 64-й строке. Если при нажатии на клавишу «q» (в английской раскладке) будет изменен радиус у первого эллипса, то это означает, что мы действительно обратились к объекту (массиву объктов класса *MySuperBall*) и по второй ссылке.

И, наконец то, небольшой метод *draw()*, который только и делает, что обходит массив объектов класса *MySuperBall* по ссылке *ballArray_ one*. У каждого объекта мы вызываем метод *upDate()*, а затем *render()*.

Результат выполнения кода Листинга 69 представлен на Рисунке 80.

Таким образом, наши объекты с каждым вызовом метода *draw()* изменяют свое положение и затем отрисовываются на экране. Причем движение зависит ин индивидуальных значений свойств объекта, от его скорости и направления вектора движения. Каждый объект «знает» своего

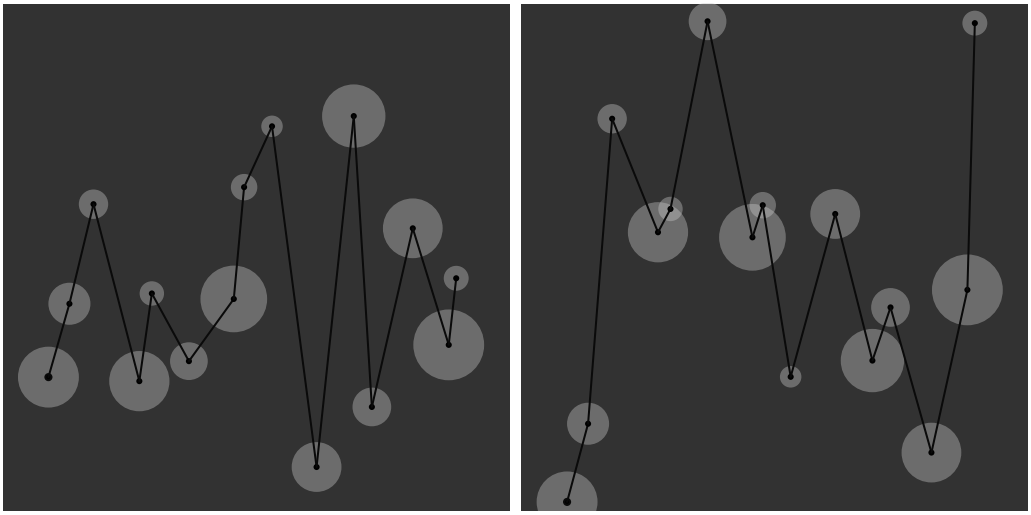


Рис. 80: Результат выполнения Листинга 69.
Объекты класса MySuperBall.

соседа и рисует к нему линию. В заключение нажмите на «q» и объекты пройдут заново путь создания и настройки; нажмите на «q», и вы увидите, как первый объект изменит свой вид.

Продолжим изменять код, воздействуя на динамику движения: обратите внимание на код Листинга 70.

Листинг 70: Изменяем динамику объектов

```
1
2  class MySuperBall{
3    float x,y,radius,speed,counter;
4    MySuperBall previousBall;
5    int vector = 1;
6    void render(){
7      noStroke();
8      fill(200, 100);
9      ellipse(x,y,radius,radius);
10     stroke(0);
11     strokeWeight(1);
12     if(previousBall != null){
13       line(x,y,previousBall.x,previousBall.y);
14       noStroke();
15     }
16     fill(0);
17     ellipse(x,y,6,6);
18   }
19
```

```
20    void upDate(){
21      counter += speed*vector/500;
22      y = 250 + sin(counter)*200;
23      if(counter > TWO_PI){
24        vector = vector*(-1);
25      }
26    }
27  }
28
29  MySuperBall[] ballArray_one, ballArray_two;
30
31  void setup() {
32    size(500, 500);
33    smooth();
34    myInit();
35  }
36
37  void myInit(){
38    int number = 125;
39    float step = (float) width / (float) (number);
40    ballArray_one = new MySuperBall[number];
41    for(int i = 0; i < ballArray_one.length; i++){
42      MySuperBall tmp_obj = new MySuperBall();
43      float variable = random(0,5);
44      tmp_obj.x = variable + step*i;
45      tmp_obj.y = random(-100,100) + 250;
46      tmp_obj.radius = variable*10 + 5;
47      tmp_obj.speed = random(0.2, 10);
48      if(i > 0){
49        tmp_obj.previousBall = ballArray_one[i-1];
50      }
51      ballArray_one[i] = tmp_obj;
52    }
53    ballArray_two = ballArray_one;
54  }
55
56  void draw(){
57    background(40);
58    for(MySuperBall curentBall: ballArray_one){
59      curentBall.upDate();
60      curentBall.render();
61    }
62  }
63
64  void keyPressed() {
65    if (key=='a') myInit();
66    if (key=='q') ballArray_two[0].radius = 300;
67    if (key=='s') saveFrame("myProcessing.png");
68  }
```

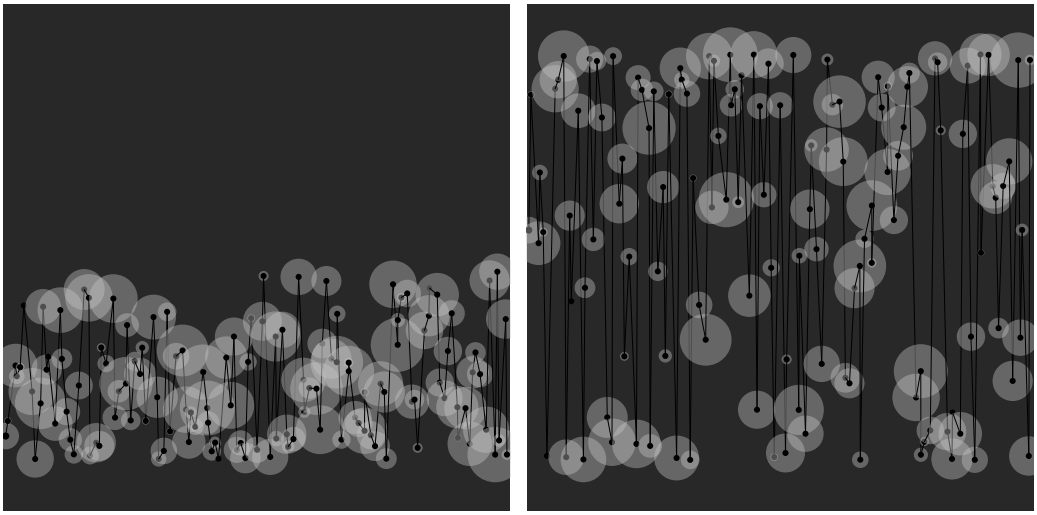


Рис. 81: Результат выполнения Листинга 70.
Новая динамика объектов класса MySuperBall.

Результат выполнения кода Листинга 70 представлен на Рисунке 81. Мы добавили свойство *counter* в 3-ю строку. Значения этого счетчика уникальны для каждого объекта, так же как, впрочем, и значения остальных наших свойств. Счетчик мы обновляем в 21-й строке, когда работаем над всем обновлением координат. Теперь движение наших эллипсов будет происходить более плавно. Остальной код не поменялся, за исключением 39-й строки, в которой мы поменяли логику определения переменной *step*.

Задание 5. Измените код Листинга 70 так, чтобы цвет отрисовки объектов зависел от их положения по оси Y: в крайних положениях цвет должен быть светлым, а посередине – темным. Для этого необходимо создать свойство в классе *MySuperBall*, отвечающее за цвет, например, *myColor* и использовать его в методе *render()*.

Итак, каждый объект имеет ссылку на своего «соседа». Давайте попробуем влиять на один объект, например, двигать его – тогда остальными можно будет управлять по цепочке. Рассмотрим пример кода Листинга 71.

В объявлении класса *MySuperBall* вводим еще одно свойство – center, которое мы будем использовать как центр для синусоидального движения. В методе *upDate()* в строке 22 мы используем это свойство, чтобы

относительно него вычислять координату Y с учетом синуса от свойства counter. В 24-й строке мы устанавливаем значение свойства center, записывая в него координату Y предыдущего объекта, в том случае, если он существуют (строка 23). Это все изменения, которые мы внесли в класс *MySuperBall*.

Листинг 71: Управление объектами по цепочке

```
1
2  class MySuperBall{
3    float x,y,radius,speed,counter,center = 250;
4    MySuperBall previousBall;
5    int vector = 1;
6    void render(){
7      noStroke();
8      fill(0, 150);
9      ellipse(x,y,radius,radius);
10     stroke(250);
11     strokeWeight(1);
12     if(previousBall != null){
13       line(x,y,previousBall.x,previousBall.y);
14       noStroke();
15     }
16     fill(250);
17     ellipse(x,y,6,6);
18   }
19
20   void upDate(){
21     counter += speed*vector/300;
22     y = center + sin(counter)*30;
23     if(previousBall != null){
24         center = previousBall.y;
25     }
26     if(counter > TWO_PI){
27       vector = vector*(-1);
28     }
29   }
30 }
31
32 MySuperBall[] ballArray_one;
33
34 void setup() {
35   size(500, 500);
36   smooth();
37   myInit();
38 }
39
40 void myInit(){
```

```
41    int number = 50;
42    float step = (float) width / (float) (number);
43    ballArray_one = new MySuperBall[number];
44    for(int i = 0; i < ballArray_one.length; i++){
45      MySuperBall tmp_obj = new MySuperBall();
46      float variable = random(0,5);
47      tmp_obj.x = variable + step*i;
48      tmp_obj.y = random(-100,100) + 250;
49      tmp_obj.radius = variable*10 + 5;
50      tmp_obj.speed = random(0.2, 10);
51      if(i > 0){
52        tmp_obj.previousBall = ballArray_one[i-1];
53      }
54      ballArray_one[i] = tmp_obj;
55    }
56  }
57
58  void draw(){
59    background(150);
60    ballArray_one[0].y = mouseY;
61    for(int i = 0; i < ballArray_one.length; i++){
62      MySuperBall curentBall = ballArray_one[i];
63      if(i != 0){
64        curentBall.upDate();
65      }
66      curentBall.render();
67    }
68  }
69
70  void keyPressed() {
71    if (key=='a') myInit();
72    if (key=='s') saveFrame("myProcessing.png");
73  }
```

Заполнение массива объектами, настройка свойств объектов в строках с 40-й по 56-ю остались без принципиальных изменений. Изменения произошли в объявлении метода *draw()*. В 60-й строке свойству y первого элемента (с нулевым индексом) массива мы присваиваем значение – координату Y курсора мыши. Этот первый элемент будет «вести за собой» все остальные объекты.

Мы обходим все объекты массива *ballArray_ one* с помощью цикла *for*, иесли текущий объект не первый (т.е. с индексом большим, чем нуль), то вызываем у него метод *upDate()*. После чего отрисовываем объект, вызвав метод *render()*.

Результат выполнения кода Листинга 71 представлен на Рисунке 82.

Наши объекты теперь меняют свое положение по очере ди слева направо. У них осталось как свое собственное движение, так и движение

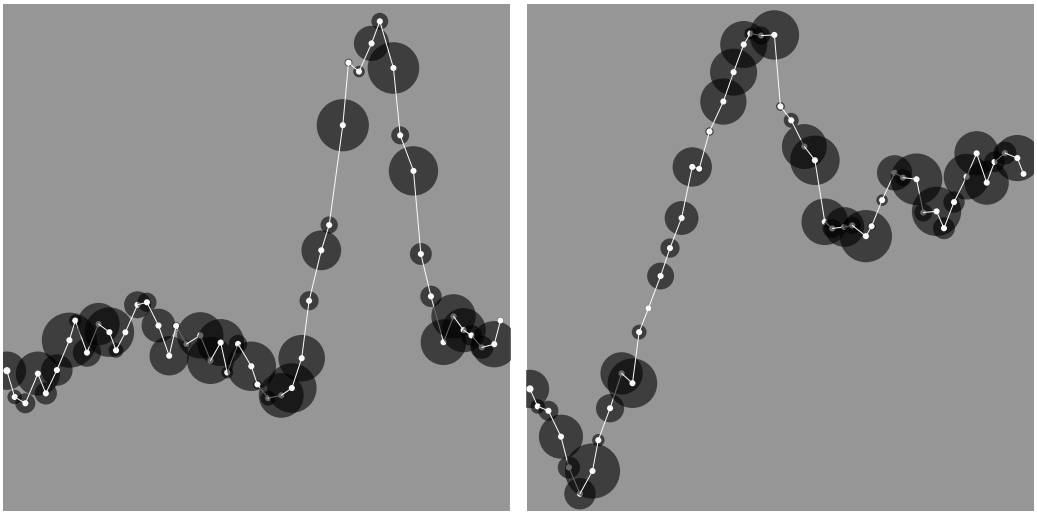


Рис. 82: Результат выполнения Листинга 71.
Фазы управления объектами по цепочке.

«подчинения» от предыдущего объекта. При определенном движении мышью можно заметить, что объекты попадают в резонанс.

9.5 «Умные» массивы

Работа с массивами определенной, фиксированной длины не всегда удобна. Часто бывает, что нам требуется добавлять все новые и новые элементы в массив, и мы не можем заранее предположить их количество. Конечно, можно каждый раз, когда мы хотим увеличить длину массива, создавать второй массив с длиной на один элемент больше и копировать туда все элементы первого, однако в Processing есть более изящные решения, одно из которых – это класс *ArrayList*.

ArrayList может менять свой размер во время исполнения программы. При создании объекта класса *ArrayList* мы не указываем количество его элементов. Элементы *ArrayList* могут быть любых типов. Давайте посмотрим, как с ним работать, на примере кода Листинга 72.

Листинг 72: ArrayList для объектов MyPoint

```
1  class MyPoint{
2      float x, y;
3
4      MyPoint(float newX, float newY){
5        x = newX;
6        y = newY;
```

```
7       }
8
9       boolean isDistanceLessThen(float dist, MyPoint point){
10        boolean result = false;
11        float currentDist = dist(point.x, point.y, x, y);
12        if(currentDist < dist){
13          result = true;
14        }
15        return result;
16      }
17    }
18
19    ArrayList<MyPoint> myPoints = new ArrayList<MyPoint>();
20
21    MyPoint currentPoint;
22    float currentDist = 10.0;
23
24    void setup() {
25      size(500, 500);
26      smooth();
27      background(255);
28      strokeWeight(1);
29    }
30
31    void draw() {
32      stroke(0, 20);
33      currentPoint = new MyPoint(mouseX, mouseY);
34      myPoints.add(currentPoint);
35      upDate();
36    }
37
38    void upDate() {
39      for(MyPoint pointFromArray : myPoints){
40        if(pointFromArray.isDistanceLessThen(currentDist,
              currentPoint)){
41          line(pointFromArray.x, pointFromArray.y, currentPoint
                .x, currentPoint.y);
42        }
43      }
44    }
45
46
47    void keyPressed() {
48      if (key=='s') saveFrame("myProcessing.png");
49      for(int i = 1; i < 10; i=i+1){
50        if(key == Integer.toString(i).charAt(0)){
51          currentDist = 10*i;
52        }
53      }
```

```
54  }
```

Суть приложения состоит в том, что положение курсора мыши постоянно записывается в одно большое хранилище (в *ArrayList*). Если текущее положение курсора мыши достаточно близко к любой уже записанной точке, то происходит отрисовка линии между этими двумя точками. Результат работы нашего приложения показан на Рисунках 83 и 84.

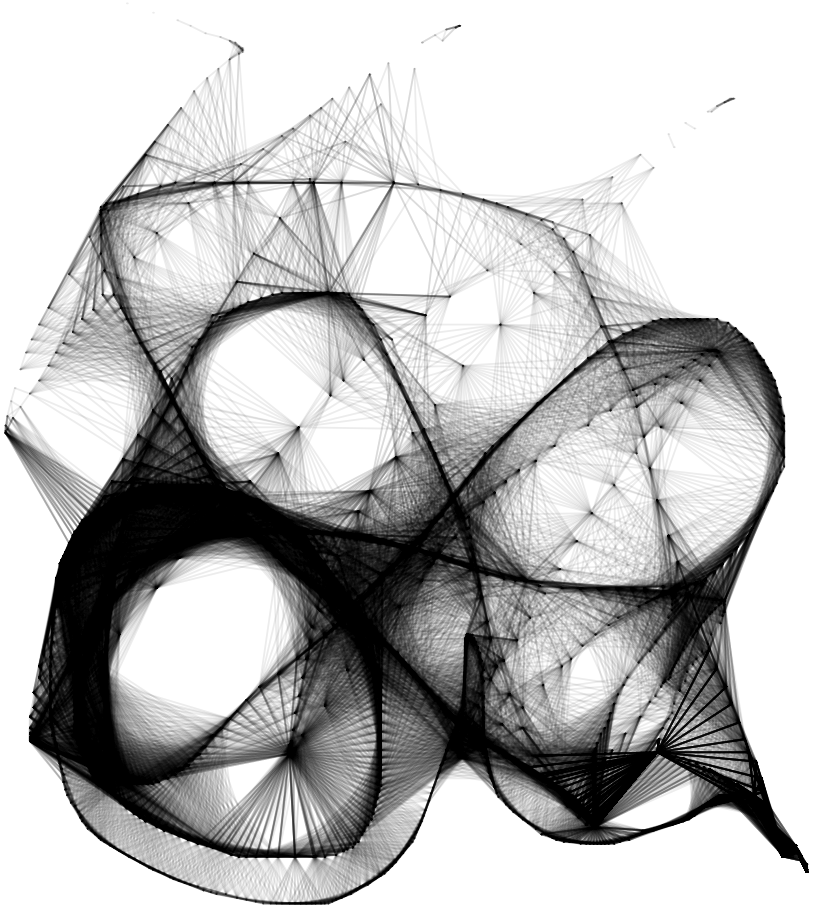


Рис. 83: Результат выполнения Листинга 72.
ArrayList для объектов MyPoint с позицией курсора мыши. Фаза 1.

Начнем рассматривать код Листинга 72 с объявления класса *MyPoint*. Этот класс будет предоставлять нам объекты для работы. Он объявляется с 1-й по 17-ю строки. В нем объявлены два свойства X и Y, конструктор и метод *isDistanceLessThen()*. Свойства, как понятно из их наименований, будут хранить значения координат. Конструктор служит более быстрому созданию объектов (в том смысле, что количество строк кода

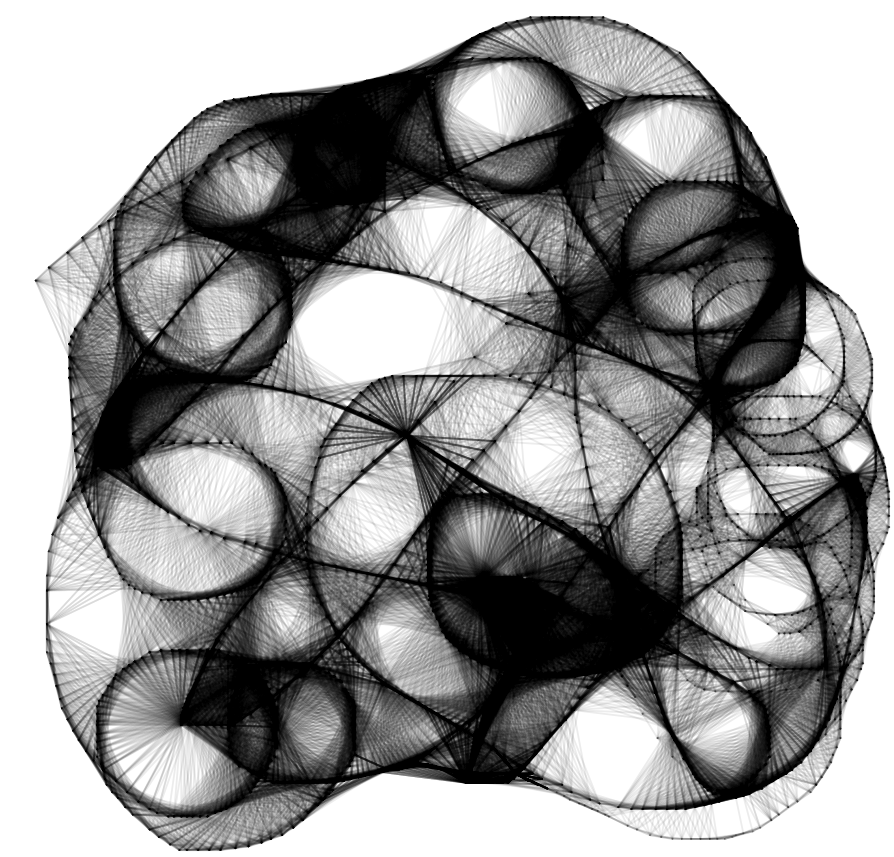


Рис. 84: Результат выполнения Листинга 72.
ArrayList для объектов MyPoint с позицией курсора мыши. Фаза 2.

будет чуть меньше). Метод *isDistanceLessThen()* принимает аргументами текущее значение дистанции и объект класса *MyPoint*, с координатами которого требуется сравнить текущие. Метод возвращает истину (TRUE), если текущее расстояние меньше исходной дистанции. На этом объявление класса заканчивается.

В 19-й строке мы объявили ссылку *myPoints* на объект класса *ArrayList*. В угловых скобках мы указываем класс объектов, которые будут храниться в нашем *ArrayList*. В правой части равенства мы создаем объект класса и с помощью знака равенства связываем объект со ссылкой *myPoints*. Как видите, работа по инициализации объекта класса *ArrayList* не столь сложна.

В 21-й строке мы объявили свойство *currentPoint*: с этой ссылкой мы

будем связывать объект, содержащий текущие координаты курсора мыши. Свойство *currentDist* в строке 22 объявлено для указания дистанции отрисовки линий.

Каждый раз когда вызывается метод *draw()* в 33-й строке мы создаем новый объект класса *MyPoint*. В его конструктор передаем текущие координаты курсора мыши и связываем объект со ссылкой *currentPoint*. В следующий вызов метода *draw()* мы опять создадим новый объект класса *MyPoint* с новыми координатами и свяжем со ссылкой *currentPoint*, при этом старая связь потеряется. Объект, с которым порвалась связь, никуда не исчез, так как в 34-й строке мы добавили его в *ArrayList*, в «умный» массив *myPoints*. У класса *ArrayList* есть множество методов, один из которых – *add()*. Его работа заключается в том, чтобы принимать аргументами объекты и добавлять их в массив. После того как мы добавили текущую точку в массив, в 35-й строке мы вызываем метод *upDate()*.

Мы не стали объявлять метод *upDate()* в классе *MyPoint*, чтобы показать, что логику работы вашего приложения вы строите сами. Если вам кажется более обоснованным располагать метод обновления координат в классе изменяемого объекта, то вы можете это сделать. Если вам кажется, что управление объектом лучше отделить от самого объекта, то располагайте управляющие методы вне класса объекта – все зависит от вас! Итак, в методе *upDate()* мы обходим *ArrayList* с помощью универсального цикла *for*, его принято называть *for each* (строка 39).

Логика работы цикла *for each* такая же, как и при работе с одномерным массивом примитивных типов данных. С каждой итерацией мы связываем ссылку *pointFromArray* со следующим элементом массива, т.е. следующим объектом *ArrayList*.

Для каждой точки массива в 40-й строке мы вызываем метод *isDistanceLessThen()* для того, чтобы получить ответ на вопрос, требуется ли рисовать линию между текущей точкой массива и точкой, где сейчас находится курсор мыши. Если мы получаем от метода *isDistanceLessThen()* результат «истина» (т.е. возвращаемое значение – TRUE), то мы попадаем в строку 41 и рисуем линию.

Описанный процесс происходит с каждым вызовом метода *draw()*, количество объектов в *ArrayList* растет с каждой секундой работы. И тем не менее, даже не очищая память, приложение может проработать несколько десятков минут без каких бы то ни было проблем. А вы получаете возможность творить своим собственным инструментом!

В этом разделе мы рассмотрели важную тему работы с массивами. Массивы, объекты и «умные» массивы открывают художнику практически все возможности для цифрового творчества. Работа с ними может

показаться, на первый взгляд, непростой, но стоит только загореться интересной творческой идеей, как тут же все сложности отступят (в крайнем случае вы всегда можете найти решение в сети Интернет).

10 Растр и вектор

10.1 Растровая графика

Для решения вопросов работы с файлами графических форматов, в Processing существует несколько методов, которые «прячут» от нас (или инкапсулируют) все сложности работы с файловой системой и чтением разных типов форматов. Processing в базовой комплектации может работать с графическими файлами таких форматов, как *jpg(jpeg), png, gif, tga* (в зависимости от операционной системы, базовая комплектация Processing предусматривает работу и с большим числом форматов). Рассмотрим код Листинга 73.

Листинг 73: Отрисовываем растровое изображение

```
1
2  PImage img;
3
4  void setup() {
5   background(100);
6   smooth();
7   size(800, 800);
8   img = loadImage("000.jpg");
9  }
10
11 void draw() {
12  background(100);
13  image(img, mouseX, mouseY);
14 }
```

Результат работы кода Листинга 73 показан на Рисунке 85.

Создаем свойство *img* – ссылку на объект класса *PImage* во второй строке кода Листинга 73. Класс *PImage* специально разработан для работы с изображениями в форматах GIF, JPG, TGA и PNG. Изображения при этом могут отображаться в 2D и 3D пространстве.

Чтобы использовать изображение в своей инсталляции, необходимо его загрузить. Загрузка чаще всего происходит с помощью метода *loadImage()*. Метод *loadImage()* принимает своим агрументом строку – путь к файлу с изображением. Чтобы не указывать полный (абсолютный) путь до картинки, можно воспользоваться возможностью добавления файлов через панель управления: *Sketch -> Add File...*. При добавлении файла в проект вы можете обращаться к нему только по имени, без указания пути от корня файловой системы.

Метод *loadImage()* возвращает объект класса *PImage*. Объект класса *PImage* содержит свойства для ширины и высоты изображения, а так-

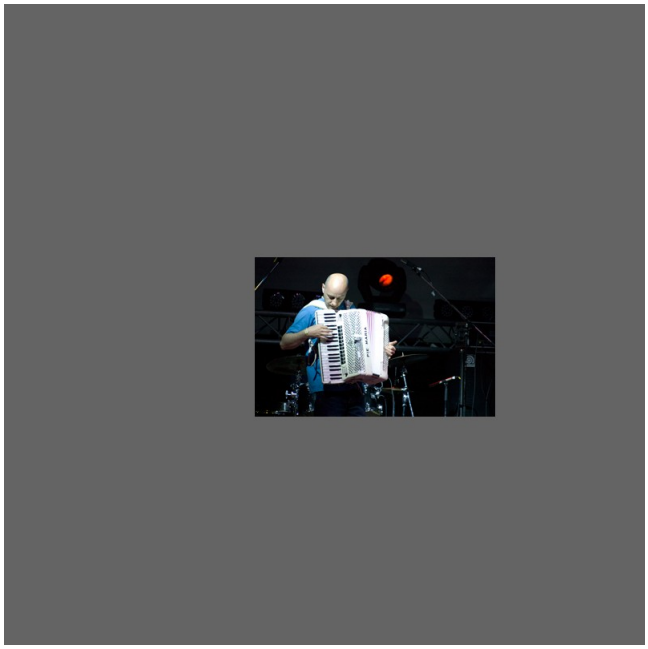


Рис. 85: Результат выполнения Листинга 73.
Отрисованное растровое изображение.

же массив с пикселями, указывающий значения цвета для каждого пикселя в изображении. Большое количество методов класса обеспечивает свободный доступ к пикселям изображения и альфа-каналу и упрощает процесс композитинга. В 8-й строке кода Листинга 73 мы связываем объект, который вернул метод *loadImage()* со свойством *img*, в 13-й строке работаем с объектом по этой (*img*) ссылке.

Вызов метода *image()* требуется для отрисовки изображения на экране вашего приложения. Это происходит в 13-й строке. В нашем примере метод *image()* принимает три аргумента: ссылку на объект *PImage* и координаты левого верхнего угла отрисовки изображения. В целом в Processing метод *image()* может принимать пять аргументов: к трем перечисленным выше могут быть добавлены ширина и высота прямоугольника изображения, который должен быть отрисован в случае, если нужно отрисовать не целое изображение, а только его часть. По умолчанию при указании всех трех аргументов изображение отрисовывается полностью.

Используя даже базовые возможности отрисовки изображений можно получать интересные результаты.

Взглянем на код Листинга 74.

Рис. 86: Результат выполнения Листинга 74.
Управляемая мышью анимация изображений.

Листинг 74: Управляем анимацией изображений

```
1
2  PImage img0;
3  PImage img1;
4  PImage img2;
5  PImage img3;
6
7  void setup() {
8   background(100);
9   smooth();
10  img0 = loadImage("i000.png");
11  img1 = loadImage("i001.png");
12  img2 = loadImage("i002.png");
13  img3 = loadImage("i003.png");
14  size(720, 500);
15 }
16
17 void draw() {
18  background(100);
19  image(img0, 0, 0);
20  image(img1, mouseX * 0.7 - 150, 100);
21  image(img2, 0, 0);
22  image(img3, width - mouseX * 1.5, 35);
23 }
```

Результат работы кода Листинга 74 показан на Рисунке 86.

В коде Листинга 74 мы используем как бы послойную отрисовку изображений, получая эффект 2.5D графики. На заднем слое отрисовывается изображение по ссылке *img0*. Свойство (ссылка) *img0* объявлена в строке 2. С этой ссылкой связан объект класса *PImage*, который мы получили в строке 10. В строках 11-13 мы получаем еще три объекта класса *PImage* и связываем их со ссылками *img1*, *img2*, *img3*.

Задний слой отрисовывается на одном и том же месте с каждым вызовом метода *draw()*. Отрисовываемое на следующем слое по ссылке *img1* изображение зависит от координаты X курсора мыши. Третий слой – ссылка *img2* – также статичен. Последний слой – ссылка *img3* – отрисовывается в обратной зависимости от координаты X курсора мыши (*width* – *mouseX...*).

Таким образом мы получаем многослойную структуру, слои которой отрисовываются по-разному в зависимости от положения курсора мыши. Два слоя статичны, два – динамичны. На каждом слое отрисовывается изображение из файла *png*, который поддерживает прозрачность. Благодаря этому слои видно друг под другом. Для приведенного в Листинге 74 примера было необходимо обтравить вручную все изображения (в отдельном растровом редакторе), кроме i000.png, которое располагается на заднем плане. Фотографии моделей были взяты с публичных профилей сайта fashionbank.ru (FashionBank – портал моделей и фотографов).

10.2 Покадровая анимация

Работа с покадровой анимацией на базе библиотеки Processing возможна, но не стоит воспринимать её как основную функцию Processing: дело в том, что эта библиотека – не самый простой инструмент для создания мультипликационных фильмов. Мы рассмотрим основные принципы работы с анимационными последовательностями изображений на примере кода Листинга 75.

Листинг 75: Рисуем покадровую анимацию

```
1
2  PImage img1;
3  PImage img2;
4  PImage img3;
5  boolean isAnimate = true;
6  int currentFrame = 1;
7
8  void setup() {
9   background(100);
10  smooth();
11  size(800, 800);
12  frameRate(12);
13  img1 = loadImage("000.jpg");
14  img2 = loadImage("001.jpg");
15  img3 = loadImage("002.jpg");
16 }
17
18 void draw() {
```

```
19    background(100);
20    if(isAnimate){
21      switch(currentFrame) {
22        case 1:
23          image(img1, mouseX, mouseY);
24        break;
25        case 2:
26          image(img2, mouseX, mouseY);
27        break;
28        case 3:
29          image(img3, mouseX, mouseY);
30        break;
31      }
32
33      currentFrame++;
34      if(currentFrame > 3){
35        currentFrame = 1;
36      }
37    } else {
38      image(img1, mouseX, mouseY);
39    }
40  }
41
42  void keyPressed() {
43    isAnimate = !isAnimate;
44  }
```

Смена кадров анимации в коде Листинга 75 совпадает со сменой кадров (фреймов) всего приложения, т.е. с каждым вызовом метода *draw()* мы отрисовываем новый кадр анимации. В нашей анимации три кадра: три изображения загружены из файлов в объекты класса *PImage* и связаны со ссылками *img1, img2, img3*. Ссылки мы объявили в строках 2, 3, 4, объекты создали с помощью вызовов метода *loadImage()* в строках 13, 14, 15.

В 6-й строке мы определили свойство (переменную) *currentFrame*: в ней мы будем хранить номер текущего кадра анимации и с каждым фреймом (с каждым вызовом метода *draw()*) мы будем обновлять его, увеличивая на единицу. Как только значение *currentFrame* станет больше, чем количество кадров (в нашем примере больше трех), то отсчет начнется заново. Таким образом мы зацикливаем анимационную последовательность. Операции по обновлению производим в строке 33, где увеличиваем значение *currentFrame* на единицу. В строках 34-36 мы проверяем, не вышло ли значение *currentFrame* на максимальное количество кадров, и если вышло, то «сбрасываем» его до единицы.

Управление анимацией происходит посредством нажатия на клавиатуру. Нажатие клавиши перехватывается методом *keyPressed()* (объяв-

лен с 42–44 строки). В этом методе выполняется строка 43, в которой мы устанавливаем новое значение свойства (переменной) *isAnimate*. Причем новое значение мы получаем с помощью логического оператора отрицания *!* . Вспомним, что логический оператор отрицания работает с булевым типом данных по следующему принципу: если мы отрицаем значение TRUE, то получаем FALSE, и наоборот: если мы отрицаем FALSE, то получаем TRUE. Таким образом, при каждом нажатии клавиши клавиатуры мы переключаем значение свойства *isAnimate* с TRUE на FALSE и обратно. В 20-й строке мы проверяем это значение, и если оно TRUE, то отрисовываем соответствующий кадр анимации.

Задание 1. Используя свои изображения, реализуйте управление анимационной последовательностью на базе кода Листинга 75: если мышь движется вправо, анимация должна проигрываться в обычном порядке, если влево – в обратном, если курсор мыши не двигается, то анимация должна остановиться.

Отрисовка кадров происходит с помощью конструкции *switch*. Конструкция *switch* по своей сути аналогична оператору *if() { ... } else { ... },* но является более удобной, если нам необходимо проверять значение одной переменной. В строке 21 мы указываем переменную (*currentFrame*), значение которой нам требуется проверять. В строке 22 мы проверяем, не равно ли значение переменной *currentFrame* единице. Если нет, попадаем в строку 25, где проверяем, не равно ли значение переменной *currentFrame* двум и т.д. Если, например, значение переменной *currentFrame* равно единице, то выполняем строку 23 и следующие строки до команды *break*. Обратите внимание, что команда break играет существенную роль и ее использование обязательно. В строке 23 мы отрисовываем кадр анимации с помощью вызова метода *image()*.

После того как мы отрисовали необходимый кадр анимации, обновляем счетчик кадров в строках 33-36. Если значение свойства *isAnimate* является FALSE, то мы не отрисовываем кадры анимации, а попадаем в строку 38. В строке 38 происходит отрисовка только изображения *img1* при каждом вызове метода *draw()* – этим мы останавливаем анимацию. Таким образом, мы получили управляемую покадровую анимацию.

10.3 Режимы наложения

Каждый вызов метода *draw()* генерит новое изображение для вывода на экран. Изображение создается последовательно, выполняя строку за

строкой в методе *draw()*. Вы можете управлять тонированием каждого шага формирования изображения в указанные цвета или устанавливая прозрачность (установив альфа-составляющую). По аналогии с указанием текущего цвета методом *fill()* вы можете указать текущее значение тонирования с помощью вызова метода *tint()*. Тонирование используется при работе с растровыми изображениями. Рассмотрим код Листинга 76

Листинг 76: Рисуем изображения с тонированием в разные цвета

```
1
2  PImage img0;
3  PImage img1;
4
5  void setup() {
6   size(1200, 595);
7   background(100);
8   smooth();
9   colorMode(HSB);
10  img0 = loadImage("mm3.jpg"); //138x595
11  img1 = loadImage("mm4.jpg");
12 }
13
14 void draw() {
15  background(100);
16  for(int i = 0; i < 10; i++){
17    tint(i*25, 150, 255);
18    if(mouseX < i*120 + 120 && mouseX > i*120){
19      noTint();
20      image(img1, i*120, 0);
21    } else {
22      image(img0, i*120, 0);
23    }
24  }
25
26 }
```

Результат выполнения кода Листинга 76 показан на Рисунках 87 и 88. Для этого примера, выполненного в духе работ Энди Уорхала было использовано два изображения Мерлин Монро в градациях серого. Первое изображение связано со ссылкой *img0*, второе с *img1*. С 16-й по 23-ю строки описана логика отрисовки изображений.

В цикле из десяти итераций (строка 16) мы каждый раз устанавливаем новое значение для режима наложения. Для этого в строке 17 мы вызываем метод *tint()* с аргументами, зависящими от текущего счетчика цикла i. В нашем случае аргументы метода *tint()* являются значениями цвета в формате HSB, так как в 9-й строке мы вызвали метод *colorMode(HSB)*. С каждой итерацией цикла значение режима наложе-

ния будет разным. Это продемонстрировано нашем приложении на Рисунках 87 и 88.

Логика строк с 18-й по 23-ю обеспечивает отрисовку изображения и второго изображения, в том случае, если курсор мыши находится в соответствующем положении. Если курсор расположен над текущим изображением, то мы попадаем в 19-ю строку, где «сбрасываем» режим наложения, а затем в 20-ю строку, в которой отрисовываем второе изображение.

Таким образом мы получили интерактивное приложение, в котором использовали режимы наложения и управляли ими.

Чтобы сделать изображение прозрачным без изменения его цвета, используйте белый оттенок цвета и укажите значение альфа-канала. Например, *tint(255, 128)* сделает изображение на 50% прозрачным.

Даже простой эффект плавного изменения прозрачности может дать необходимый результат. В нашем примере все зависит от семантики изображений, прозрачность которых мы меняем.

Рис. 87: Результат выполнения Листинга 76. Изображения с тонированием в разные цвета. Фаза 1.

Рис. 88: Результат выполнения Листинга 76. Изображения с тонированием в разные цвета. Фаза 2.

Рассмотрим пример кода Листинга 77. Результат выполнения кода Листинга 77 показан на Рисунке 89.

Листинг 77: Рисуем политиков

```
1
2  PImage img1;
3  PImage img2;
4
5  void setup() {
6   background(100);
7   smooth();
8   img1 = loadImage("vp.jpg");
9   img2 = loadImage("bo.jpg");
10  size(784, 600);
11 }
12
13 void draw() {
14  float  myTintBO = map(mouseX, 0, width, 0, 255);
15  float  myTintVP = map(mouseX, 0, width, 255, 0);
16
17  tint(255, myTintVP);
18  image(img1, 0, 0);
19  tint(255, myTintBO);
20  image(img2, 0, 0);
21 }
```

Принцип работы приложения заключается в том, что когда курсор мыши находится в левой части экрана, то первая картинка отрисовывается прозрачной, а вторая - непрозрачной. При перемещении курсора мыши по экрану слева направо степень прозрачности одной картинки уменьшается, а второй, соответственно, увеличивается. Если мы перемещаем курсор мыши справа налево, то получаем обратный эффект.

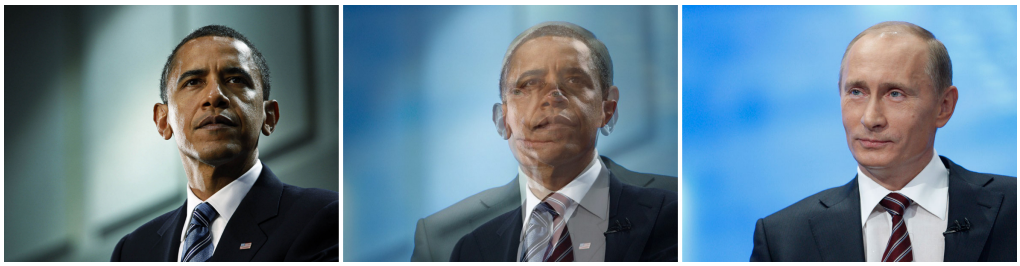


Рис. 89: Результат выполнения Листинга 77.
Фазы изменения прозрачности изображений.

Так как размер окна приложения может быть разным, а в нашем случае он зависит от размеров изображения (784 на 600, строка 10), то

нам необходим механизм пересчета значения координаты курсора мыши в диапазон прозрачности, т.е. в значения от 0 до 255. Другими словами, по горизонтали курсор меняет свою координату от 0 до 784, а нам требуется пересчитать это значение в диапазоне от 0 до 255. В Processing эта возможность реализована в методе *map()*, который мы используем в 14-й и 15-й строках. Метод *map()* принимает 5 аргументов. Первый из них – значение, которое требуется пересчитать: в нашем случае это значение координаты X курсора мыши. Второй и третий аргумент – это предельные значения системы для пересчета, т.е. входящие значения: в нашем случае это 0 и ширина изображения. Четвертый и пятый аргумент – это предельные значения системы, в которую требуется пересчитать: у нас это 0 и 255 в 14-й строке, 255 и 0 в 15-й строке.

Задание 2. На базе кода Листинга 77, используя свои изображения, реализуйтеь увеличение прозрачности одного изображения со временем. При клике мышью прозрачность должна плавно уменьшаться.

Таким образом, мы управляем прозрачностью изображений с помощью курсора мыши посредством метода *tint()* в 17-й и 19-й строках кода Листинга 77.

10.4 Работа с пикселями

Режимы наложения оперируют с цветом каждой точки изображения. В компьютерной графике мельчайшую точку изображения принято называть *пикслелем*. Растровые изображения в компьютерной графике представляют из себя массив (таблицу) точек изображения, которые отрисовываются на экране с помощью пикселей. Каждая точка изображения должна содержать информацию о цвете, например, в формате RGB. Массив точек изображения для простоты часто называют массивом пикселей.

По идее этот массив пикселей изображения должен быть двумерным, т.е. если мы увеличим маленький фрагмент растрового изображения (6x4 пикселя), то пиксели (будем принимать их за квадраты) будут выглядеть как показано на Рисунке 90.

Пронумеруем пиксели для наглядности (Рисунок 90). Мы это делаем только для того, чтобы мысленно вытянуть этот массив элементов в одну линию, соблюдая нумерацию. Должно получиться примерно то, что показано на Рисунке 91.

Рис. 90: Пиксели изображения при увеличении.

Рис. 91: Пиксели изображения при разложении в цепочку.

При такой раскладке массива что бы найти пиксель с координатами двумерного массива $x = 4$, $y = 2$, т.е. пиксель с номером 16, нужно проделать следующую операцию: $index = x + y * img.width$, где $img.width$ – количество пикселей в ширину картинки: $index = 4 + 2 * 6 = 16$.

Рассмотрим пример использования знания о пикселях в реальном приложении – в коде Листинга 78.

Листинг 78: Рисуем в стиле пуантилизма

```
1
2  PImage img0;
3  PImage img1;
4
5  void setup() {
6   background(100);
7   smooth();
8   noStroke();
9   img0 = loadImage("012.jpg");
10  img1 = loadImage("012black.jpg");
11  size(600, 800);
12 }
13
14 void draw() {
15   if(frameCount == 1){
16     image(img1, 0, 0);
17   }
18
19  float pointSize = map(mouseX, 0, width, 0, 100);
20  float pointAlpha = map(mouseY, 0, height, 0, 255);
21
22  int x = (int) random(img0.width);
```

```
23    int y = (int) random(img0.height);
24    int loc = x + y*img0.width;
25    img0.loadPixels();
26    float r = red(img0.pixels[loc]);
27    float g = green(img0.pixels[loc]);
28    float b = blue(img0.pixels[loc]);
29
30    fill(r,g,b,pointAlpha);
31    ellipse(x,y,pointSize,pointSize);
32
33    tint(255, 2);
34    image(img1, 0, 0);
35  }
36
37  void keyPressed() {
38    saveFrame("myProcessing" + frameCount + ".jpg");
39  }
```

Результат выполнения кода Листинга 78 показан на Рисунке 92. Идея работы приложения состоит в том, что мы, вдохновившись стилем пуантилизма, формируем наше изображение с помощью точек разного размера и цвета. За основу мы взяли фотографию с публичного профиля сайта fashionbank.ru (FashionBank – портал моделей и фотографов). Предварительно мы обработали фотографию в растровом редакторе, переведя ее в градации серого. На черно-белой фотографии мы отрисовываем точки, цвет которых берем из цветной фотографии. Размер точек и их прозрачность зависят от положения курсора мыши.

Файл с цветным изображением мы загрузили и связали со ссылкой *img0* в 9-й строке, а черно-белое изображение – со ссылкой img1 в 10-й строке. Размер окна приложения соответствует размеру самого изображения – в 11-й строке мы вызываем метод *size(600, 800)*.

Для того чтобы отрисовать черно-белое изображение только один раз, мы пишем строки с 15-й по 17-ю. В 15-й строке идет проверка номера кадра. Если текущий кадр (номер текущего вызова метода *draw()*) равняется единице, т.е. приложение в первый раз запустило метод *draw()*, то мы попадаем в строку 16 и отрисовываем изображение по ссылке *img1* – черно-белую картинку. В следующий вызов метода *draw()* мы не отрисовываем эту картинку, а рисуем объекты поверх нее.

Чтобы управлять размером точек и их прозрачностью с помощью координат курсора мыши, мы соотносим их значения с соответствующими переменными в строках 19 и 20. Размер точки – *pointSize* – будет варьироваться в пределах от 0 до 100. Прозрачность – *pointAlpha* – от 0 до 255.

Точки мы будем отрисовывать по очереди: каждую точку в каждом

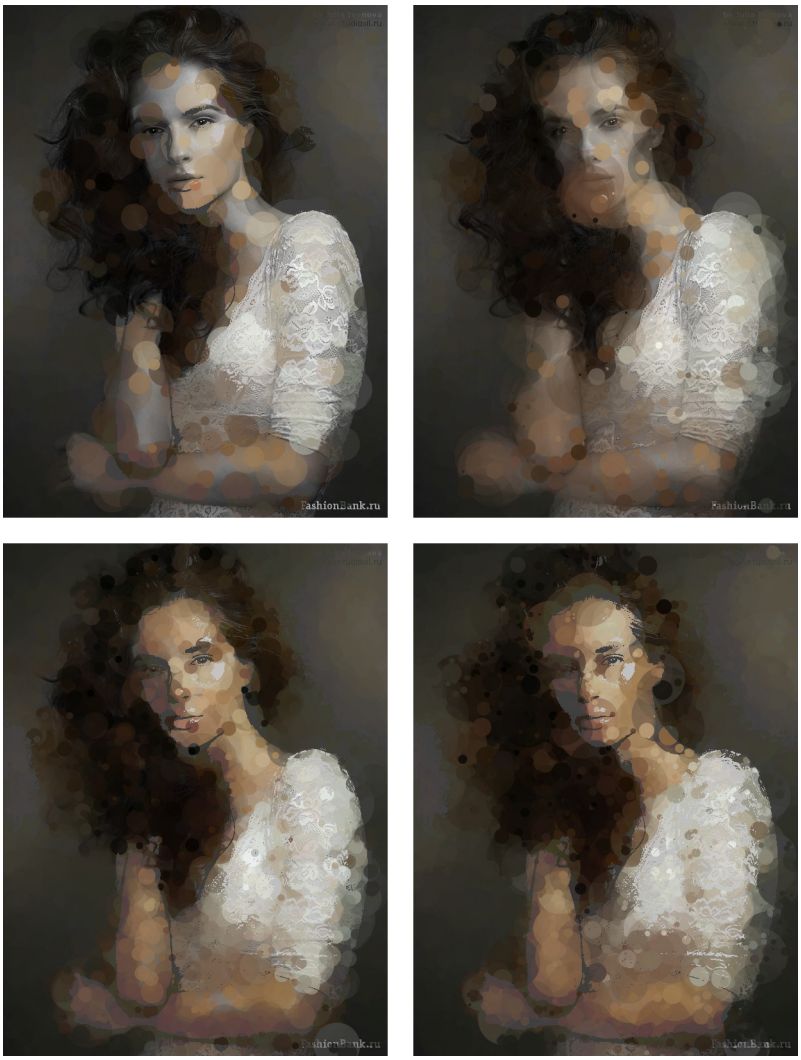


Рис. 92: Результат выполнения Листинга 78.
Фазы работы в стиле пуантилизма.

кадре (т.е. в каждом вызове метода *draw()* будем отрисовывать одну точку). Положение точки будет случайным (строки 22 и 23). В строке 24 происходит вычисление индекса точки в массиве пикселей изображения *img0* (мы рассматривали его на примере Рисунка 91). В 25-й строке мы вызываем метод *loadPixels()* от объекта по ссылке *img0*. Этот метод заполняет массив пикселей *pixels[]*, который, в свою очередь, является свойством класса *PImage*. В принципе, в нашем случае, мы можем вызвать это метод только один раз, т.е. не в *draw()*, а в *setup()*: так как изображение по ссылке *img0* не меняется в ходе работы приложения, то нет смысла загружать одни и те же пиксели в массив с каждым вызовом метода *draw()*.

Задание 3. На базе кода Листинга 78, используя свое изображение, отрисуйте не эллипсы, а линии или крестики.

Чтобы получить цветовое значение каждого пикселя из RGB канала, нам необходимо вызывать методы *red(), green() и blue()*, которые должны получить аргументом пиксель. Зная его индекс в массиве, мы указываем его в квадратных скобках (строки 16, 17, 18). Когда мы получили цвет текущего пикселя со случайными координатами из цветного изображения, мы можем задать заливку (строка 20) и отрисовывать круг необходимого размера с заливкой. Для более яркого художественного эффекта мы отрисовываем черно-белое изображение с большой прозрачностью по верх, чтобы проявились его черты и нюансы.

10.5 Режимы смешивания и фильты

При работе с растровыми изображениями часто используются так называемые режимы смешивания. При этих режимах происходит обработка пикселей изображения тем или иным алгоритмом. В Processing есть возможность использовать как режимы смешивания, так и растровые фильтры.

Рассмотрим пример использования растрового фильтра в коде Листинга 79. В основу этого примера лег код Джима Бумгарднера (Jim Bumgardner, http://krazydad.com/tutorials/circles_js/), который мы немного изменили.

Листинг 79: Динамические эллипсы с эффектом размытия

```
1
```

```
2  float lg_diam, lg_rad, lg_circ, sm_diam, cx, cy;
3
4  void setup() {
5   background(100);
6   smooth();
7   size(500, 400);
8   noStroke();
9   lg_diam =  width * .55;     // large circle's diameter
10  lg_rad  = lg_diam/2;        // large circle's radius
11  lg_circ =  PI * lg_diam;    // large circumference
12  cx = width/2;
13  cy = height/2;
14  colorMode(HSB);
15 }
16
17 void draw() {
18   fill(0,10);
19   rect(0,0,width,height);
20
21   int nbr_circles = (int) map(mouseX, 0, width, 6, 50);
22   sm_diam = (lg_circ / nbr_circles);      // small circle's
         diameter
23
24   int myColor = (int) map(mouseY, 0, height, 150, 255);
25
26   filter(BLUR, 3);
27
28   fill(myColor,180,190,100);
29
30   for (int i = 1; i <= nbr_circles; ++i) {
31     float angle = i * TWO_PI / nbr_circles;
32     float x = cx + cos(angle) * lg_rad;
33     float y = cy + sin(angle) * lg_rad;
34     ellipse(x, y, sm_diam, sm_diam);
35   }
36 }
37
38 void keyPressed() {
39   if (key=='s') saveFrame("myProcessing" + frameCount + ".
        jpg");
40 }
```

В этом примере мы вычисляем количество эллипсов, которые будут отрисованы по окружности. Вычисление происходит в 21-й строке с помощью вызова метода *map()*. Минимальное количество эллипсов 6, максимальное – 50. Их количество связывается с горизонтальным смещением курсора мыши. В 22-й строке, исходя из количества эллипсов, мы вычисляем их размер, зная длину окружности, по которой они бу-

дут располагаться – *lg_circ* (строка 11, вычисление длины окружности как умножение $\pi * lg\_daiam$). В 24-й строке идет определение цвета: мы работаем в режиме HSB (строка 14).

Задание 4. Измените код Листинга 79: перенесите вызов метода *filter(BLUR, 3)* из 26-й строки в строку после 35-й, после отрисовки эллипсов в цикле.

В 26-й строке мы применяем к нашему изображению целиком фильтр размытия – BLUR. Применение фильтра происходит с помощью вызова метода *filter()* с передачей аргумента BLUR (это константа, определенная в Processing). Вторым аргументом метода идет степень размытия.

Результат выполнения кода Листинга 79 показан на Рисунке 93.

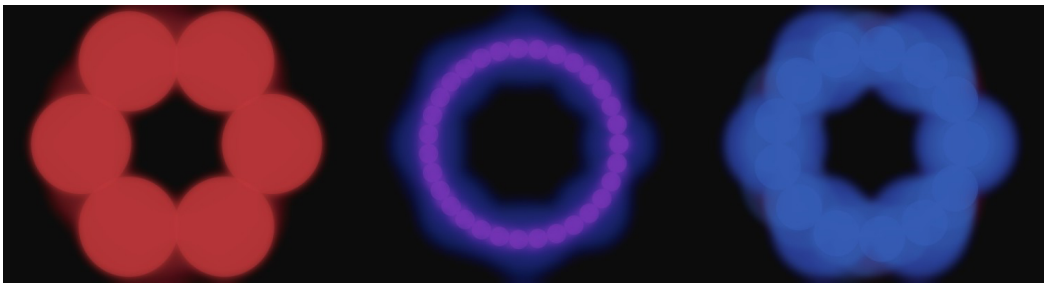


Рис. 93: Результат выполнения Листинга 79. Фазы динамики эллипсов с эффектом размытия.

После того как мы определили количество, размер и цвет эллипсов, нам остается их отрисовать. Отрисовка эллипсов происходит в цикле for с 30-й по 35-ю строки. В 32-й и 33-й строке мы вычисляем значения координат X и Y с применением уже знакомых нам тригонометрических функций. Угол *angle* зависит от количества эллипсов (строка 31), переменная *lg_rad* отвечает за ширину отступа эллипсов от центра.

Таким образом мы получили плавный эффект размытия. Движение эллипсов получило новую эмоциональную окраску. Обратите внимание, что фильтр применяется ко всему произведению целиком.

Задание 5. Примените фильтр *BLUR* к любому из предыдущих ваших приложений.

Режимы смешивания в Processing задаются с помощью вызова метода *blendMode()* и работают только с растровыми изображениями. Рассмотрим его использование на примере кода Листинга 80.

Листинг 80: Инсталляция на тему бетонных многоэтажек

```
1
2  PImage img;
3  PImage img2;
4  PImage img3;
5  PImage img4;
6
7  void setup() {
8   background(100);
9   smooth();
10  size(670, 436);
11  noStroke();
12  img = loadImage("build.jpg");
13  img4 = loadImage("build1.gif");
14 }
15
16 void draw() {
17   if(frameCount == 1){
18     image(img,0,0);
19   }
20
21   int val2 = (int) random(0,150);
22   int val3 = (int) random(0,150);
23
24   img2 = img.get(mouseX + val2,0,20,height);
25   img3 = img.get(mouseX + val3,0,5,height);
26
27   blendMode(SUBTRACT);
28   tint(255,20);
29   image(img2,mouseX + val2, random(0,height));
30
31   noTint();
32   blendMode(BLEND);
33   image(img3,mouseX - val3, 0);
34
35   image(img4,0, 0);
36 }
37
38 void keyPressed() {
39   if (key=='s') saveFrame("myProcessing" + frameCount + ".
       jpg");
40 }
```

Принцип работы этой инсталляции состоит в следующем. Вначале

отрисовывается первое изображение по ссылке *img*. Оно отрисовывается целиком только один раз при первом фрейме (строки с 17-й по 19-ю). Затем мы копируем из него вертикальные прямоугольные площади рандомной (случайной) ширины (строки 21,22 и 24,25) в новые объекты класса *PImage*, после чего происходит связывание этих объектов со ссылками *img2* и *img3*, которые объявлены как свойства в строках 3 и 4. На этом этапе у нас есть исходное изображение, отрисованное при запуске программы, и два *PImage* объекта с прямоугольника из этого изображения во всю его высоту.

В 27-й строке мы включаем режим наложения SUBTRACT с помощью вызова метода *blendMode()*. Этот метод может принимать различные константы, которые управляют применяемым режимом наложения. В 33-й строке мы отрисовываем *img2*. Отрисовка происходит не в том месте, откуда он был скопирован.

В 32-й строке мы устанавливаем другой режим смешивания и отрисовываем второй объект *PImage* по ссылке *img3*.

В заключение мы отрисовываем *img4*. Эта ссылка связана с объектом *PImage* еще в 13-й строке. Мы использовали *gif* файл с прозрачностью, чтобы крыши домов на переднем плане инсталляции не менялись при работе приложения.

Результат выполнения кода Листинга 80 показан на Рисунке 94.

Рис. 94: Результат выполнения Листинга 80.
Инсталляция на тему бетонных многоэтажек, фрагменты.

Так как скопированные прямоугольники отрисовываются не в месте своего копирования, мы получаем эффект перестройки бетонных многоэтажек. Причем общий вид застройки, не меняясь, остается унылым: режим смешивания SUBTRACT подчеркивает неряшливый вид изношенного бетона. Надо отметить, что эта тематика не чужда современным художникам: например, Михаэль Вульф (Michael Wolf) активно использует мотивы городской застройки такого типа в своих фотографи-

ях. Примеры его работ можно увидеть на сайте фотографа по ссылке http://photomichaelwolf.com/#transparent-city/14.

Работа с фильтрами и режимами смешивания была представлена нами не во всем объеме возможностей, которыми обладает Processing. Мы рассмотрели лишь принципы работы на конкретных примерах. Подробную информацию о фильтрах можно получить из официальной документации по ссылке: https://processing.org/reference/filter_.html , о режимах смешивания – по ссылке: https://processing.org/reference/blendMode_.html

10.6 Геометрия из SVG

Примеры приложений в этом параграфе были разработаны нами под влиянием творчества Джона Маеда (John Maeda). Избранные работы его студии вы можете найти на сайте http://www.maedastudio.com/.

Несмотря на то что в Processing вы можете создавать статические геометрические формы, часто для этих целей используют редакторы векторной графики.

Векторная графика является средством хранения информации о геометрии рисунка. Например, когда мы пишем в коде *line(100,100,200,200);*, сохраняется информация о линии. Обратите внимание: эта линия не зависит от разрешения экрана или печатной (полиграфической) машины. Т.е. линия в нашем случае описана максимально полно, а ее отрисовка, конечно, зависит от свойств конечного устройства.

Существуют визуальные редакторы векторной графики, такие как Inkscape и Adobe Illustrator. Inkscape – редактор с открытым исходным кодом, и так же, как и Processing может быть использован бесплатно. Результат работы в векторном редакторе может быть использован в Processing.

Для того чтобы добавить SVG изображение в ваш скетч, вам необходим сам SVG файл. Далее нужно использовать объект класса *PShape*, который возвращается методом *loadShape()*, который, в свою очередь, принимает аргументом путь к SVG файлу. Проще хранить все файлы в одной Processing папке проекта: тогда вы можете выбрать в панели управления *Sketch->Add File...* и добавить ваш SVG файл. Этот файл Processing скопирует в папку *data* вашего проекта.

Прежде чем рассматривать Листинг 81, рассмотрим процесс подготовки SVG файла в редакторе Inkscape. В нем мы набрали слово ХАОС и перевели его в кривые. Каждая буква, а теперь уже каждый объект,

соответствует своему XML элементу. У каждого такого элемента есть атрибут *id*. Значение атрибута *id* для каждой буквы уникально.

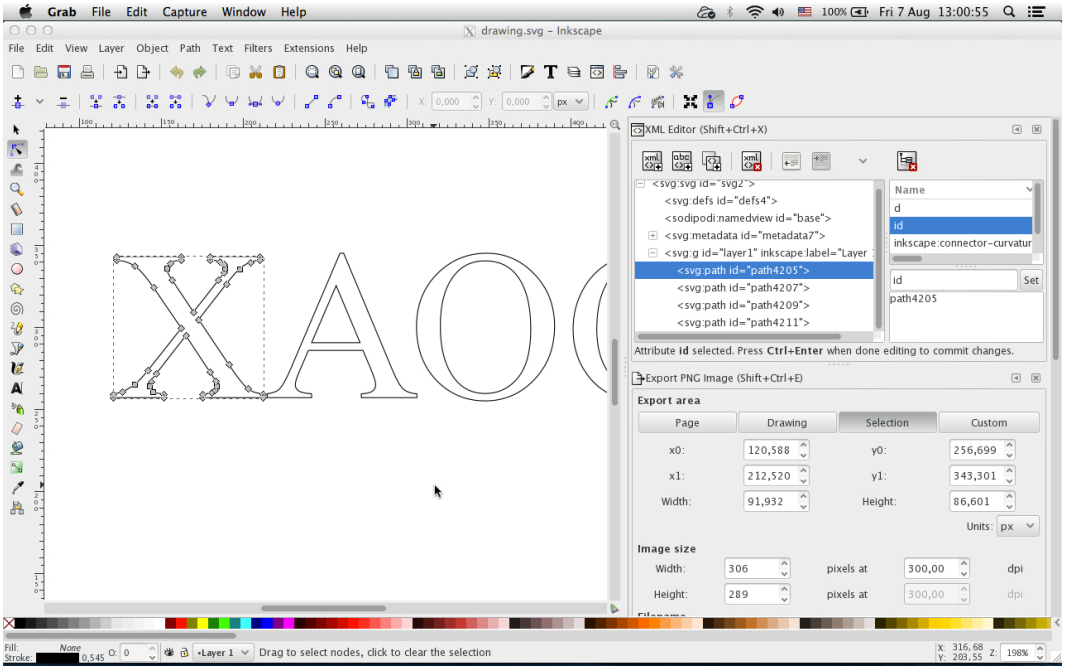


Рис. 95: Подготовка SVG файла в редакторе Inkscape

На Рисунке 95 вы можете видеть, что выделен SVG путь буквы «X». В правой части изображения открыт редактор XML и выбран соответствующий элемент в XML редакторе, мы можем видеть его *id*. Например, для буквы «X» *id* имеет значение «*path4205*».

Буква «X», как показано на Рисунке 95, содержит несколько вершин, или точек. Из этих точек мы будем отрисовывать линии в нашем приложении, код которого и представлен в Листинге 81.

Листинг 81: Работаем с SVG файлом и линиями

```
1
2  PShape drawingSVG;
3
4  void setup() {
5   background(100);
6   smooth();
7   strokeWeight(1);
8   size(600, 600);
9   drawingSVG = loadShape("drawing.svg");
10  size((int)drawingSVG.width,(int)drawingSVG.height);
11
```

```
12  }
13
14  void draw() {
15    fill(10, 15);
16    rect(0,0,width,height);
17    float mCursor = map(mouseX, 0, width, 100, 155);
18    fill(10, 100);
19    drawingSVG.disableStyle();
20    shape(drawingSVG, 0, 0);
21
22    PShape border = drawingSVG.getChild("path4205");
23    for(int i = 0 ; i < border.getVertexCount(); i++){
24      float vx = border.getVertexX(i) - 45; //122 120
25      float vy = border.getVertexY(i) - 60; //343  497.54
26      float lx = vx + random(-150, 150);
27      float ly = vy + random(-150, 150);
28      float lineColor = mCursor + random(-100, 100);
29      stroke(lineColor,100);
30      line(vx, vy, lx, ly);
31      fill(200, 50);
32      noStroke();
33      ellipse(vx, vy, 3,3);
34    }
35
36    PShape border1 = drawingSVG.getChild("path4207");
37    for(int i = 0 ; i < border1.getVertexCount(); i++){
38      float vx = border1.getVertexX(i) - 45; //122 120
39      float vy = border1.getVertexY(i) - 60; //343  497.54
40      float lx = vx + random(-150, 150);
41      float ly = vy + random(-150, 150);
42      float lineColor = mCursor + random(-100, 100);
43      stroke(lineColor,100);
44      line(vx, vy, lx, ly);
45      fill(200, 50);
46      noStroke();
47      ellipse(vx, vy, 3,3);
48    }
49
50    PShape border2 = drawingSVG.getChild("path4209");
51    for(int i = 0 ; i < border2.getVertexCount(); i++){
52      float vx = border2.getVertexX(i) - 45; //122 120
53      float vy = border2.getVertexY(i) - 60; //343  497.54
54      float lx = vx + random(-150, 150);
55      float ly = vy + random(-150, 150);
56      float lineColor = mCursor + random(-100, 100);
57      stroke(lineColor,100);
58      line(vx, vy, lx, ly);
59      fill(200, 50);
60      noStroke();
```

```
61        ellipse(vx, vy, 3,3);
62      }
63
64      PShape border3 = drawingSVG.getChild("path4211");
65      for(int i = 0 ; i < border3.getVertexCount(); i++){
66        float vx = border3.getVertexX(i) - 45; //122 120
67        float vy = border3.getVertexY(i) - 60; //343  497.54
68        float lx = vx + random(-150, 150);
69        float ly = vy + random(-150, 150);
70        float lineColor = mCursor + random(-100, 100);
71        stroke(lineColor,100);
72        line(vx, vy, lx, ly);
73        fill(200, 50);
74        noStroke();
75        ellipse(vx, vy, 3,3);
76      }
77
78    }
79
80    void keyPressed() {
81      if (key=='s') saveFrame("myProcessing" + frameCount + ".
            jpg");
82    }
```

Вначале нам необходимо загрузить SVG файл. За работу с ним в Processing отвечает класс *PShape*, ссылку на который мы объявляем во 2-й строке в виде свойства *drawingSVG*. Чтобы получить объект этого класса и загрузить в него информацию из нашего SVG файла, мы воспользуемся методом *loadShape()* в строке 9.

У класса *PShape* есть множество методов и свойств: например, свойства *width* и *height* содержат информацию о размере изображения. Мы используем их, чтобы задать размер нашему окну в строке 10.

Работа с объектом *PShape* продолжается в 19-й и 20-й строках: там мы сбрасываем все художественные настройки нашего объекта, которые у него были и затем отрисовываем его. Отрисовка происходит с помощью вызова метода *shape()* по аналогии с методом *image()* для отрисовки растровых изображений. Результат выполнения кода Листинга 81 показан на Рисунке 96.

Мы обходим букву за буквой, а в ней вершину за вершиной и рисуем линии от текущей вершины в рандомную точку. В самой вершине мы отрисовываем небольшой эллипс.

Чтобы обратиться к конкретному XML элементу, а в нашем случае, к конкретной букве, мы будем использовать id. Метод *getChild()*, который мы вызываем от объекта класса *PShape* по ссылке *drawingSVG*, принимает аргументом значение атрибута *id*. Именно для этого необходимо

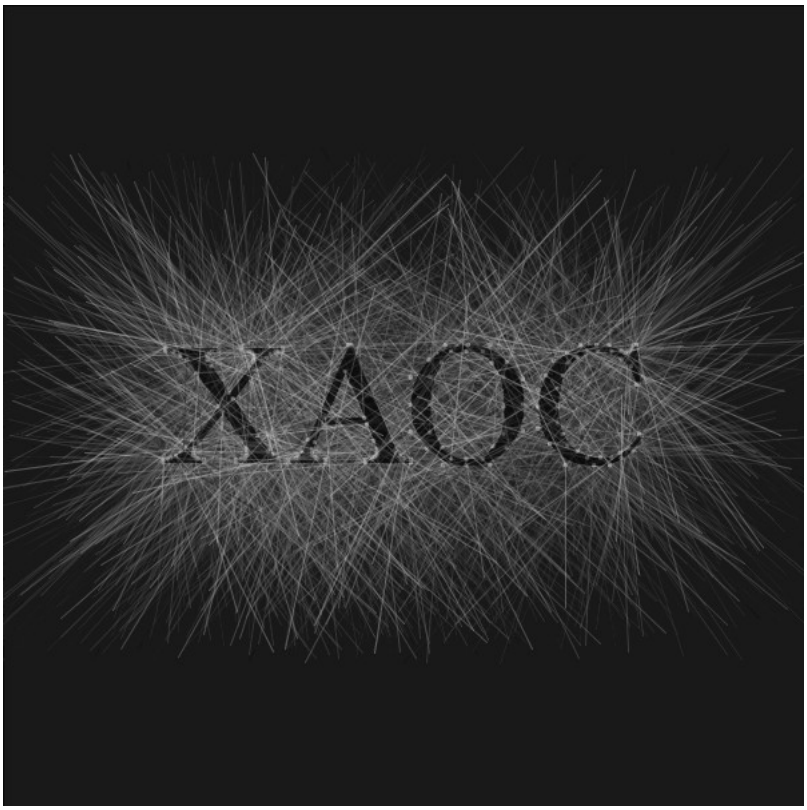


Рис. 96: Результат выполнения Листинга 81. Загрузка SVG файла и обработка его с помощью линий.

знать *id* (напомню, что его можно посмотреть в Inkscape редакторе). В 22-й строке мы вызываем этот метод и связываем возвращенный им объект класса *PShape* со ссылкой *border*. Сейчас она связана только с нашей буквой «X».

Количество вершин мы получаем с помощью вызова метода *getVertexCount()* и используем это значение в условии выхода из цикла *for* в строке 23. В цикле нам требуется определить координаты текущей вершины. Методы *getVertexX()* и *getVertexY()* с аргументом в качестве номера вершины мы вызываем по ссылке *border* и формируем координаты X и Y (нам приходится немного «подгонять» координаты «в ручную»). На этом работа с SVG-частью заканчивается и от этой координаты мы отрисовываем линию. Эту логику мы применяем ко всем остальным буквам.

Продолжая работу на кодом Листинга 81, рассмотрим его модификацию на примере кода Листинга 82, результат работы которого представлен на Рисунке 97.

Листинг 82: Работаем с SVG файлом и кривыми Безье

```
1
2  PShape drawingSVG;
3
4  void setup() {
5   background(100);
6   smooth();
7   strokeWeight(1);
8   size(600, 600);
9   drawingSVG = loadShape("drawing.svg");
10  size((int)drawingSVG.width,(int)drawingSVG.height);
11
12 }
13
14 void shapeDraw(PShape myShapeToDraw){
15   for(int i = 0 ; i < myShapeToDraw.getVertexCount(); i++){
16     float vx = myShapeToDraw.getVertexX(i) - 45; //122 120
17     float vy = myShapeToDraw.getVertexY(i) - 60; //343
          497.54
18     float lx1 = vx + random(-150, 150);
19     float ly1 = vy + random(-150, 150);
20     float lx = mouseX + random(-150, 150);
21     float ly = mouseY + random(-150, 150);
22     float mCursor = map(mouseX, 0, width, 0, 255);
23     stroke(mCursor, 10);
24     noFill();
25     bezier(vx, vy, lx, ly, lx1, ly1, vx, vy);
26   }
```

```
27  }
28
29  void draw() {
30    fill(10, 10);
31    drawingSVG.disableStyle();
32    shape(drawingSVG, 0, 0);
33
34    PShape border = drawingSVG.getChild("path4205");
35    shapeDraw(border);
36
37    PShape border1 = drawingSVG.getChild("path4207");
38    shapeDraw(border1);
39
40    PShape border2 = drawingSVG.getChild("path4209");
41    shapeDraw(border2);
42
43    PShape border3 = drawingSVG.getChild("path4211");
44    shapeDraw(border3);
45
46  }
47
48  void keyPressed() {
49    if (key=='s') saveFrame("myProcessing" + frameCount + ".
        jpg");
50  }
```

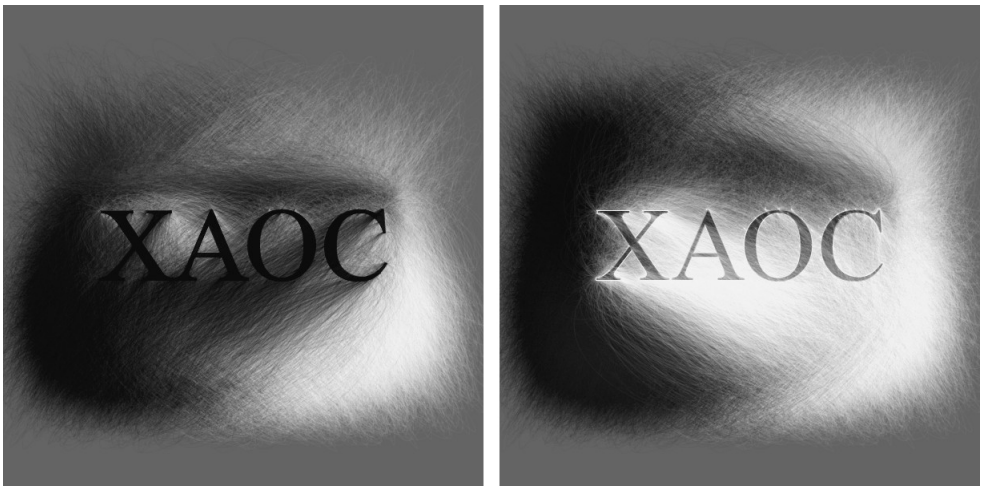


Рис. 97: Результат выполнения Листинга 82. Загрузка SVG файла и обработка его с помощью кривых Безье.

В этом примере мы вынесли логику работы с буквой в отдельный метод и назвали его *shapeDraw()*. Метод объявили с 14-й по 27-ю строки. Объявленный метод принимает ссылку *myShapeToDraw* на объект класса

PShape и ничего не возвращает (*void*). Работа этого метода по обходу SVG пути описана в разборе предыдущего примера. Но в отличие от него, логика отрисовки линий здесь заменения на логику отрисовки кривых Безье, причем отрисовка зависит еще и от положения курсора мыши.

С определением нового метода *shapeDraw()* метод *draw()* стал гораздо компактнее, и читать его стало проще, хотя, по сути, его работа не поменялась: мы также используем метод *getChild()* для получения буквы и передаем ее на отрисовку.

Задание 6. На базе Листинга 82 обработайте свои SVG файлы. Не останавливайтесь только на обработке букв, экспериментируйте с логотипами, иконками и векторным рисунками.

Итак, в этой главе мы рассмотрели работу с растровой и векторной графикой в Processing, показали на примерах варианты их использования. В совмещении растровых эффектов с геометрическими построениями и заранее подготовленными SVG файлами открывается еще одна возможность реализации художественных замыслов. В следующей главе мы продолжим использовать полученные знания и навыки.

11 Текст и шрифт

В начале было слово... Работа с типографикой, со шрифтами и интерактивными текстами содержит в себе массу интересных возможностей. Многие цифровые художники активно используют в своих работах текст . Другие, наоборот, включают интерактивные шрифтовые композиции в свои произведения.

Работа со шрифтом на прикладном уровне связана с рядом сложностей, например, в области формата шрифта: в этом формате должны храниться начертания букв и символов в векторной форме. Также нельзя забывать и о типографской традиции, которая предусматривает такие свойства шрифта, как выравнивание, интерлиньяж, межбуквенное расстояни – всё это создает дополнительнын сложности в реализации.

В Processing работа со шрифтами отдана классу PFont. Processing работает с определенным форматом шрифта (.vlw), поэтому, прежде чем реализовывать свое приложение, требуется сформировать шрифт. Для того, чтобы создать шрифт, можно выбрать в панели задач *Tools -> Create Font...*. В появившимся окне (смотри Рисунок 98) необходимо выбрать из списка системных шрифтов тот, который вам требуется. Также нужно указать размер букв, желательно с запасом, т.е. указывать больший размер, чем вам действительно требуется. Дело в том, что Processing отрисовывает каждую букву выбранного вами шрифта в растровую картинку того размера, который вы выбрали. Это означает, что каждую букву в формате .vlw можно сравнить с картинкой. Исходя из этого, надо представлять художественные возможности работы с объектами класса *PFont*. И если вы выберете размер шрифта меньше, чем требуется, то вы потеряете качество изображения буквы при масшабировании.

После того как вы сформировали шрифт, он появится в папке *data* вашего проекта (смотри Рисунок 98, нижнее изображение). Объект класса *PFont* рекомендуется получать с помощью метода *loadFont()*, который своим аргументом принимает наименование файла шрифта (с расширением .vlw). Получить список всех доступных шрифтов можно с помощью метода *list()*.

Вы можете создавать шрифты .vlw не только с помощью пользовательского интерфейса Processing. Существует метод *createFont()*, который делает это «на лету»: метод принимает аргументами наименование доступного шрифта, размер, необходимость сглаживания и массив символов конвертации, а результатом является объект класса *PFont* и файл .vlw в папке *data*.

Рис. 98: Подготовка файла шрифта к использованию в Processing.

11.1 Работа со шрифтом, PFont

Рассмотрим базовую работу с текстом на примере кода Листинга 83. В нем мы определенным шрифтом отрисуем слово в окне нашего приложения.

Во второй строке кода Листинга 83 мы объявили свойство *font* класса *PFont*. С этим свойством мы свяжем объект класса *PFont* в 9-й строке, который, в свою очередь, возвращается методом *loadFont()*. Пока нам не требуется анимация, так что мы вызвали метод *noLoop()*, и значит, метод *draw()* сработает только один раз.

Далее, в 10-й строке мы устанавливаем текущий размер шрифта (вызываем метод *textFont()*), так же как и текущий цвет заливки в строке 11. Заливка будет действовать в том числе и на буквы.

Листинг 83: Рисуем предложение

```
1
2  PFont font;
3
4  void setup(){
5    size(600, 180);
6    smooth();
7    background(0);
8    noLoop();
9    font = loadFont("Bauhaus93-48.vlw");
10   textFont(font, 48);
11   fill(178, 7, 157);
12
13 }
14
15 void draw(){
16   text("The world is here", 120, 100);
17 }
```

Отрисовка непосредственно текста происходит с помощью вызова метода *text()* в 16-й строке. Метод принимает строку для отрисовки и положение, где именно она будет отрисована. Таким образом мы отрисовали строку текста. Если нам требуется нарисовать две строки, то необходимо или еще раз вызвать метод *text()* с другими значениями аргументов, или использовать символ перевода строки: \n.

Результат работы кода Листинга 83 показан на Рисунке 99.

Модифицируем код Листинга 83. Мы будем отрисовывать два слова, причем разными цветами и в разных местах приложения. Работая на контрастах, мы «закрутим» нашу композицию так, чтобы слова перекрывали друг друга, как бы боролись и в то же время сливались в одном ритме.

Рис. 99: Результат выполнения Листинга 83.
Предложение.

Листинг 84: Крутим слова

```
1
2  PFont font;
3  float rotateCounter = 0;
4
5  void setup(){
6    size(600, 600);
7    smooth();
8    background(0);
9
10   font = loadFont("Bauhaus93-48.vlw");
11   textFont(font, 48);
12 }
13
14 void draw(){
15   translate(width/2, height/2);
16
17   pushMatrix();
18   rotate(rotateCounter);
19   fill(255);
20   text("Black", mouseX - width/2, mouseY - height/2);
21   popMatrix();
22
23   pushMatrix();
24   rotate(-rotateCounter*1.5);
25   fill(0);
26   text("White", width/2 - mouseX, height/2 - mouseY);
27   popMatrix();
28
29   rotateCounter+=0.05;
30 }
```

Результат работы кода Листинга 84 показан на Рисунках 100 и 101. В виду того что класс *PFont* работает с растровым форматом (.vlw)

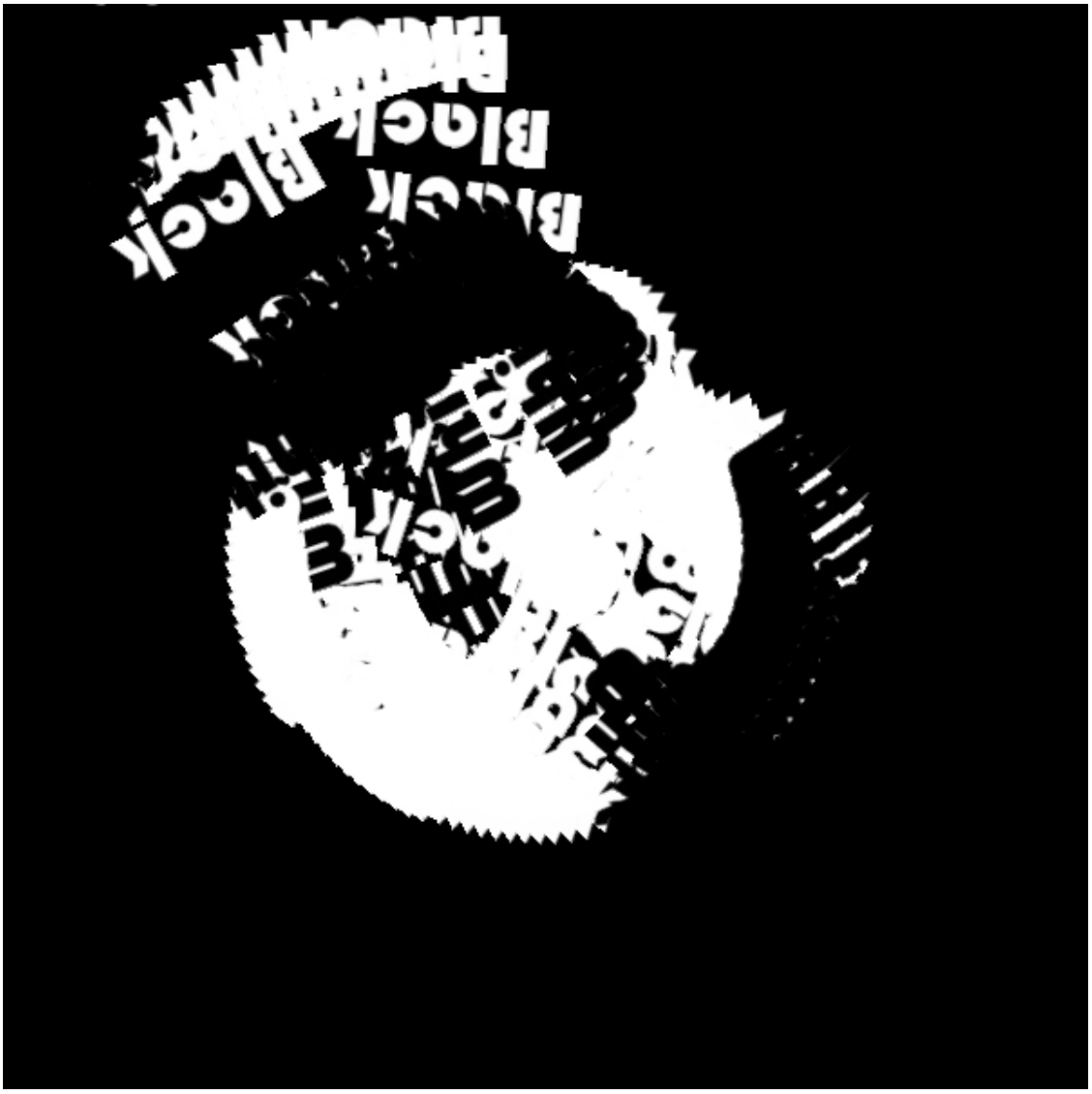


Рис. 100: Результат выполнения Листинга 84.
Вращение двух слов *Black* и *White*. Фаза 1.

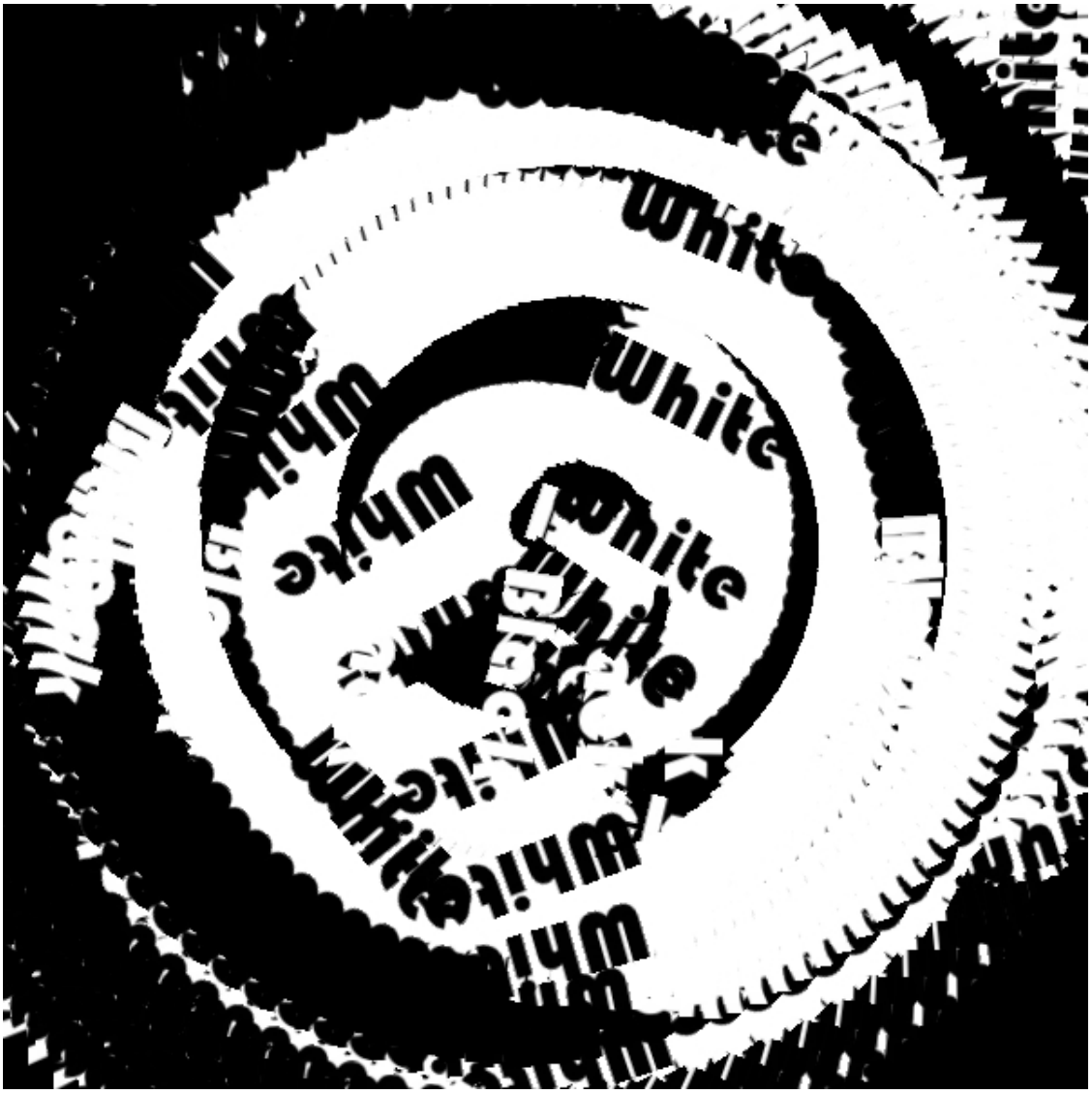


Рис. 101: Результат выполнения Листинга 84.
Вращение двух слов *Black* и *White*. Фаза 2.

шрифта, то у нас нет возможности дать букве например, обводку. Возможна работа и с кириллическим текстом (так же, как и работа со всеми шрифтами кодировки utf8): для этого вам потребуется при создании шрифта зайти в подменю *Characters...* и выбрать *All Characters*. Тогда в файле .vlw будут сформированы кирилические символы (конечно, выбранный вами шрифт должен содержать кириллические глифы). Кстати, сам редактор Processing (версия 2) не поддерживает кириллические символы и по умолчанию они отображаются прямоугольными значками. Пример использования кириллицы можно увидеть на Рисунке 102.

В Processing существует ряд методов для работы с объектами класса *PFont*. Методы призваны решать такие задачи типографики, как выравнивание текста – метод *textAlign()*, управление межстрочным интервалом – метод *textLeading()*, управление размером шрифта – метод *textSize()*, получение ширины символа – метод *textWidth()*.

Задание 1. На базе Листинга 84 создайте приложение, в котором используйте два разных шрифта. К изображению примените фильтры, тонирование или режимы смешивания.

Метод *textMode()* с аргументом *SHAPE* используется для формирования векторных глифов взамен растровых изображений. Этот режим работы со шрифтом применяется для сохранения исходного результата в форматы pdf и dxf. В Processing версии 2 невозможно использовать этот режим для отрисовки на экране, тем не менее рассмотрим работу с тестом из формата SVG, где мы можем получить доступ до векторного формата глифа.

11.2 Работа с буквами из формата SVG

Если вам требуется более изящная работа с линиями начертания буквы, то есть векторное представление глифов, то класс *PFont* вам не подойдет. Тогда удобнее работать с глифами напрямую, например, преобразовав текст в кривые в любом векторном редакторе. Дальнейший процесс аналогичен процессу работы с форматом SVG.

Начнем с подготовки нашего SVG изображения. Этот пример мы будем рассматривать также с помощью редактора Inkscape. В нем мы создали SVG документ размерами 600x300 пикселей и набрали слово «Visual» определенным шрифтом. К сожалению, импортирование SVG файлов, содержащих тексты и шрифты, в Processing невозможно, поэтому мы перевели слово в кривые и в каждой букве увеличили количество

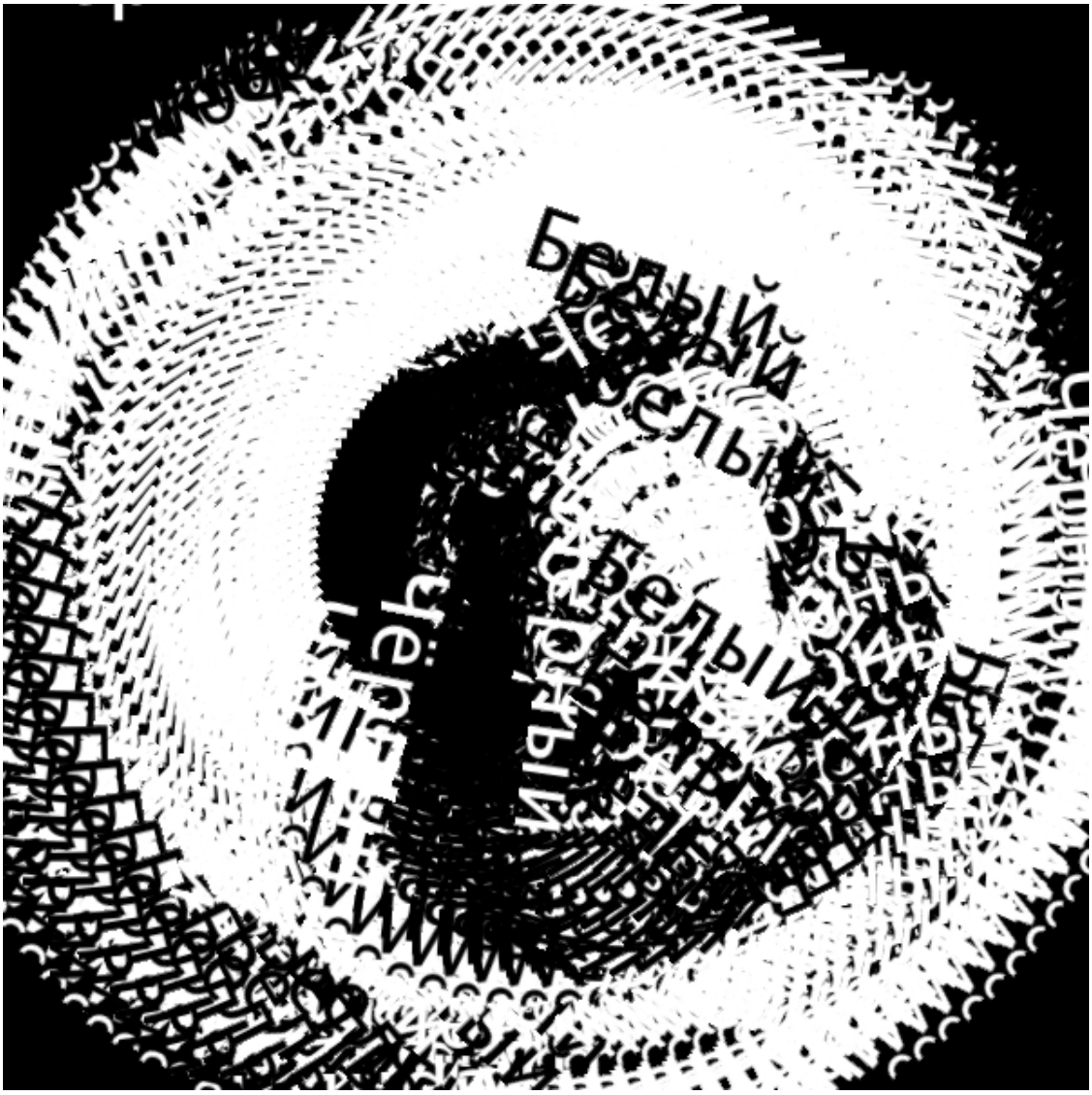


Рис. 102: Результат выполнения Листинга 84.
Вращение двух слов *Белый* и *Черный*.

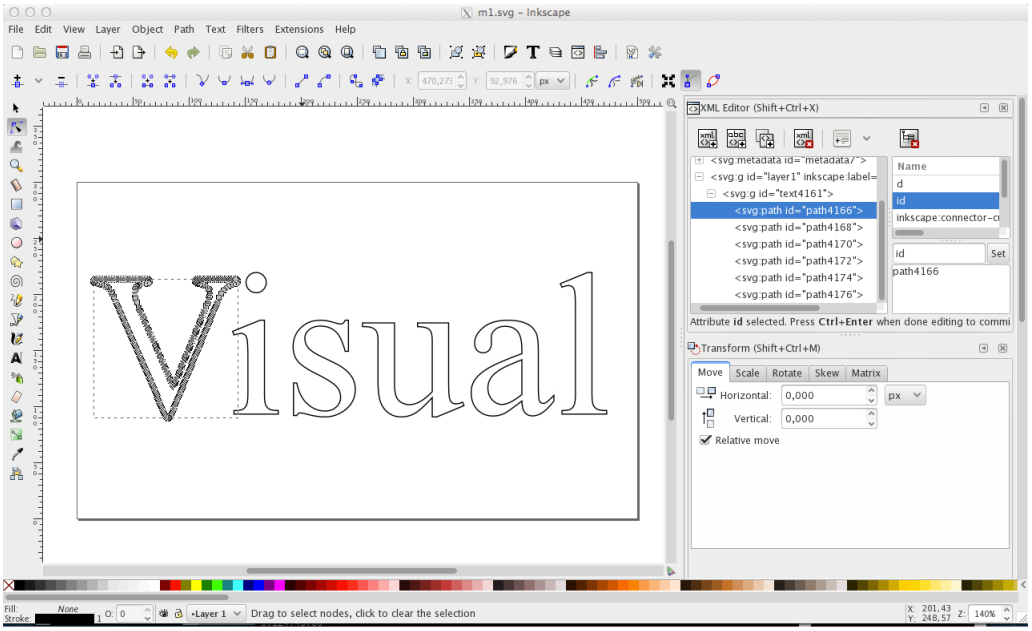


Рис. 103: Снимок экрана с процессом редактирования в *Inkscape*.

точек на кривой. Снимок экрана с процессом редактирования в Inkscape показан на Рисунках 103 и 104.

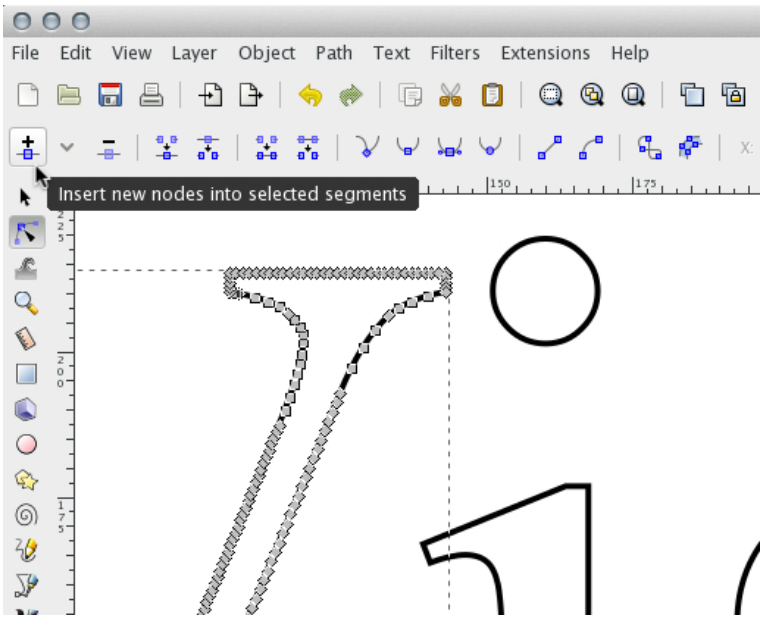


Рис. 104: Размещение точки между двумя соседними в редакторе *Inkscape*.

На Рисунке 103 можно видеть примерное количество точек, которое должно быть на кривой. Дело в том, что далее мы будем отрисовывать в этих точках те или иные фигуры, поэтому плотность точек должна быть сравнительно большой. Разместить точку между двумя соседними точками в редакторе Inkscape можно с помощью команды *Insert new nodes into selected segments* (см. Рисунок 104).

Так как каждая буква переведена в кривые, то она имеет свой уникальный идентификатор – *id*. Посмотрите на Рисунок 103: в его правой части открыто окно XML Editor, в котором можно видеть 6 тегов XML SVG. Каждый тег соответствует соответствующей букве и имеет атрибут *id* с уникальным значением. Например, сейчас выделена буква «V» и ей соответствует тег с *id = path4166*. По этому *id* мы будем обращаться к букве из Processing приложения. Далее мы должны сохранить наше SVG изображение и добавить его в проект Processing.

Рассмотрим пример кода Листинга 85, который будет обрабатывать наш подготовленный SVG файл.

Листинг 85: Рисуем переливающиеся буквы

```
1
2  PShape[] borders;
3  int vertexCount;
4  float sinCounter = 0;
5  float eSize = 15;
6
7  void setup(){
8    size(600, 300);
9    smooth();
10   background(0);
11   noStroke();
12   PShape drawingSVG = loadShape("m1.svg");
13   borders = new PShape[6];
14   borders[0] = drawingSVG.getChild("path4166");
15   borders[1] = drawingSVG.getChild("path4168");
16   borders[2] = drawingSVG.getChild("path4170");
17   borders[3] = drawingSVG.getChild("path4172");
18   borders[4] = drawingSVG.getChild("path4174");
19   borders[5] = drawingSVG.getChild("path4176");
20 }
21
22 void draw(){
23   background(0);
24   for(int j = 0; j < 6; j++){
25     vertexCount = borders[j].getVertexCount();
26     for(int i = 0; i < vertexCount; i+=1){
27       float vx = borders[j].getVertexX(i) + 50;
```

```
28          float vy = borders[j].getVertexY(i) - 750;
29          float ellipseColor = map(sin(sinCounter),-1,1,0,255);
30          fill(ellipseColor);
31          float ellipseSize = map(i,0,vertexCount,eSize,eSize
               +40);
32          ellipse(vx, vy, ellipseSize, ellipseSize);
33
34          if(sinCounter < TWO_PI){
35            float cv = map(mouseX,0,width,0,1);
36            sinCounter+=cv;
37          } else {
38            sinCounter = 0;
39          }
40        }
41      }
42      //filter(BLUR, 1);
43  }
```

Этот пример был написан под влиянием работы Ekstasy Type Club проекта http://paperdove.com/ от Dae In Chung. Исходные коды работы не опубликованы, поэтому представляю вам свой вариант. Результат работы кода показан на Рисунках 105 и 106.

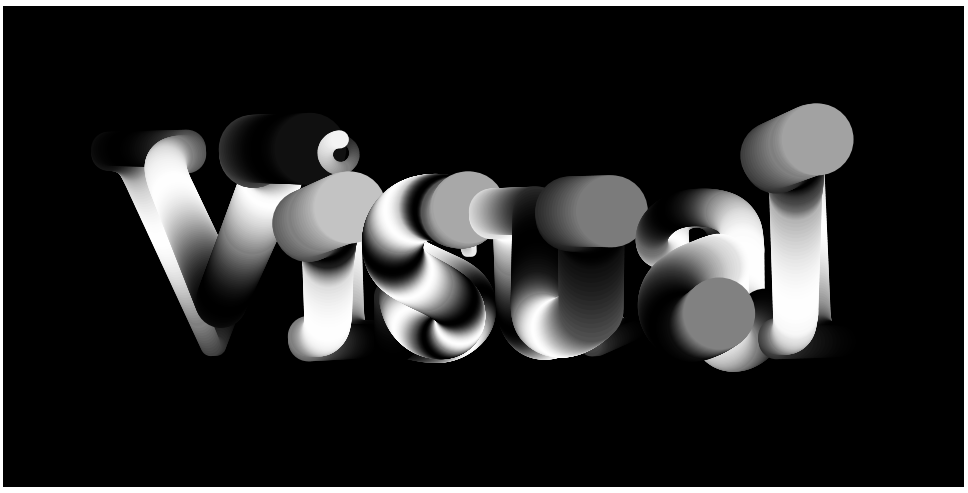


Рис. 105: Результат выполнения Листинга 85.
Переливающиеся буквы. Фаза 1.

Общий принцы работы кода Листинга 85 состоит в том, что мы «обходим» букву за буквой и в каждой букве «обходим» вершины (точки). Для каждой вершины мы отрисовываем свой эллипс, причем цвет эллипса изменяется по синусоидальному закону от черного к белому. Размер эллипса уменьшается от первой вершины буквы к последней.

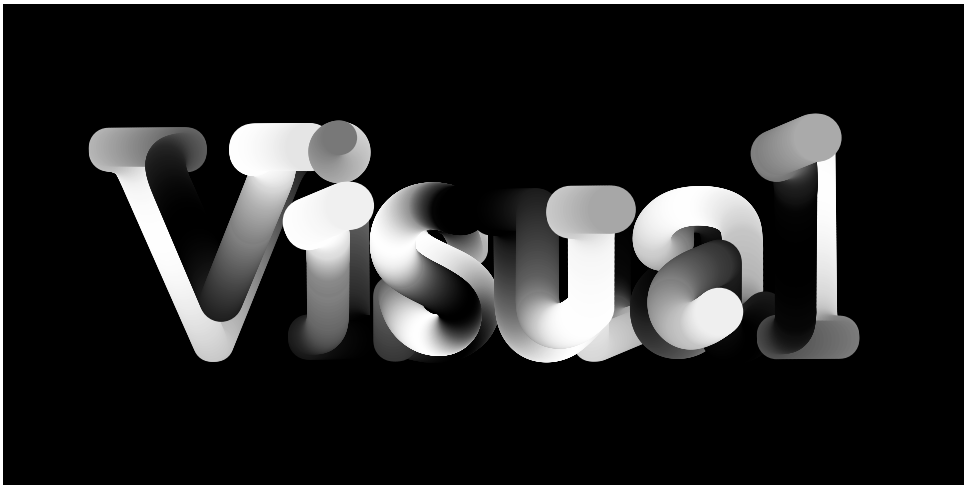


Рис. 106: Результат выполнения Листинга 85.
Переливающиеся буквы. Фаза 2.

Так как операции по отрисовке эллипсов идентичны для каждой буквы, то логично прописать их один раз и применять последовательно к каждой букве. Для этого удобнее хранить буквы в массиве. Ссылку на этом массив мы объяляем во 2-й строке. Мы знаем, что букв у нас шесть, поэтому в 13-й строке мы создали массив для объектов класса *PShape* из 6 элементов.

В 12-й строке мы загружаем SVG файл с помощью метода *loadShape()* и связываем его со ссылкой *darwingSVG*. С 14-й по 19-ю строку мы заполняем наш массив объектами класса *PShape*, т.е. нашими буквами. Зная id тега SVG для буквы, мы вызываем метод *getChild()* от объекта *darwingSVG*. Например, в 14-й строке мы написали *drawingSVG.getChild("path4166")*, так как «path4166» – это *id* буквы «V» в нашем слове (см. Рисунок 103).

Затем мы обходим букву за буквой: перебираем в цикле элементы массива *borders*. Цикл *for* объявляется с 24-й строки. Для каждой буквы нам требуется знать количество вершин, чтобы дальше перебрать и их тоже. Количество вершин мы получаем в 25-й строке с помощью вызова метода *getVertexCount()* от текущего элемента массива borders (т.е. от текущей буквы).

Обход вершин также осуществляем с помощью цикла *for*. Следите, чтобы во вложенных циклах наименование переменных, отвечающих за шаг цикла, было разное. В нашем случае это переменные *j* и *i*. Итак, в 27-й и 28-й строках мы записали значение координат текущей вершины в переменные *vx* и *vy* (нам пришлось их подправить: +50 и -750 – чтобы

они адекватно располагались на экране).

Цвет эллипса мы задаем по значению синуса счетчика *sinCounter*, соотнеся методом *map()* значение синуса с цветовой шкалой от 0 до 255. Счетчик *sinCounter* мы объявили в 4-й строке и с каждым вызовом внутреннего цикла *for* он увеличивается на определенное значение – *cv*. Значение *cv* зависит от положения курсора мыши. В этом приложении наиболее интересный эффект получается, если курсор мыши располагается в левой части окна приложения.

Размер эллипса мы определяем в 31-й строке, в зависимости от количества вершин в букве. Мы пересчитываем размер текущего эллипса, исходя из минимального значения, за которое отвечает переменная *eSize* в 5ой строке, и максимального *eSize+40*.

В 32-й строке мы отрисовываем эллипс. Таким образом, с каждым вызовом метода *draw()* мы отрисовываем все эллипсы на тех же местах. Координаты положения эллипса каждый раз (каждый *draw()*) мы получаем заново из SVG. Если бы значения SVG менялись, то мы видели бы движение геометрии, а в нашем примере мы видим только движение цвета. Чтобы придать размытие нашему художественному решению, можно открыть 42-ю строку, которая дана в виде комментария.

Работая с SVG файлом можно получать различные художественные эффекты. Рассмотрим код Листинга 86.

Листинг 86: Рисуем второй художественный эффект

```
1
2  PShape drawingSVG;
3  PShape[] borders;
4  int vertexCount;
5  float sinCounter = 0;
6
7  void setup(){
8    size(600, 300);
9    smooth();
10   background(0);
11   drawingSVG = loadShape("m1.svg");
12   borders = new PShape[6];
13   borders[0] = drawingSVG.getChild("path4166");
14   borders[1] = drawingSVG.getChild("path4168");
15   borders[2] = drawingSVG.getChild("path4170");
16   borders[3] = drawingSVG.getChild("path4172");
17   borders[4] = drawingSVG.getChild("path4174");
18   borders[5] = drawingSVG.getChild("path4176");
19 }
20
21 void draw(){
```

```
22    noStroke();
23    fill(0, 10);
24    rect(0,0,width,height);
25
26    stroke(200, 150);
27    strokeWeight(1);
28    fill(100,200);
29    drawingSVG.disableStyle();
30    //shape(drawingSVG , 50, 0);
31    noFill();
32
33    for(int j = 0; j < 6; j++){
34      vertexCount = borders[j].getVertexCount();
35      for(int i = 0; i < vertexCount-1; i+=2){
36        float vx = borders[j].getVertexX(i) + 50;
37        float vy = borders[j].getVertexY(i) - 750;
38        float vx1 = borders[j].getVertexX(i+1) + 50;
39        float vy1 = borders[j].getVertexY(i+1) - 750;
40
41        float vcx1 = vx + random(-10,10);
42        float vcx2 = vx1 + random(-10,10);
43        float vcy1 = vy + random(-10,10);
44        float vcy2 = vy1 + random(-10,10);
45
46        float letterSColor = map(sin(sinCounter),-1,1,0,255);
47        stroke(letterSColor, 50);
48
49        if(sinCounter < TWO_PI){
50          float cv = map(mouseX,0,width,0,0.5);
51          sinCounter+=cv;
52        } else {
53          sinCounter = 0;
54        }
55
56        bezier(vx, vy, vcx1, vcy1, vcx2, vcy2, vx1, vy1);
57      }
58    }
59    //filter(BLUR, 1);
60  }
```

Результат работы кода Листинга 86 представлен на Рисунках 107 и 108.

Код Листинга 86 отличается от предыдущего только логикой отрисовки. Обратите внимание на 35-ю строку: в ней шаг цикла имеет значение 2, а не 1, как в предыдущем случае. Мы делаем это намеренно, так как отрисовываем кривую Безье между двумя соседними точками. Текущая вершина – это элемент массива *i*, а следующая вершина – *i+1* (строки 36 – 39). Значения координат узлов кривой Безье мы задаем раномно

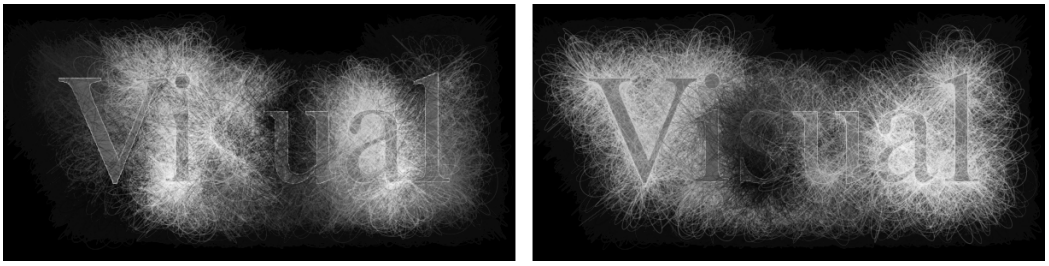


Рис. 107: Результат выполнения Листинга 86.
Фазы применения художественного эффекта с кривыми Безье

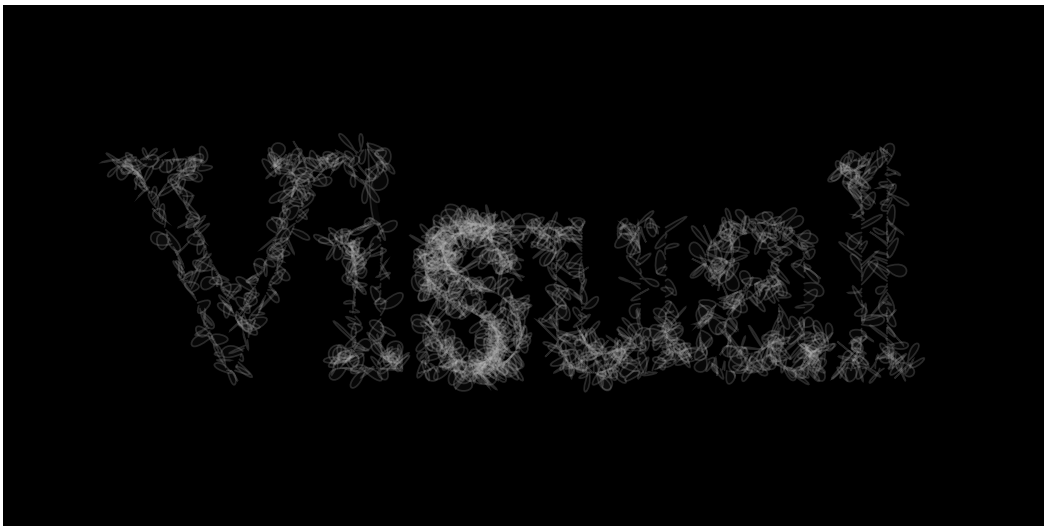


Рис. 108: Результат выполнения Листинга 86.
Художественный эффект с кривыми Безье. Фаза 3.

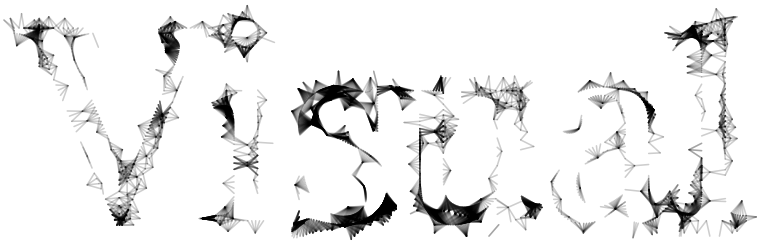


Рис. 109: Результат выполнения Листинга 87.
Слово нарисованное мышью по SVG пути. Фаза 1.

(строки 41 – 44). Изменяя граничные значения рандомизации, можно получать интересные эффекты. Финальная отрисовка кривой происходит в 56-й строке.

Рассмотрим еще одну вариацию на тему нашего SVG слова: на Рисунках 109 и 110 показаны результаты выполнения кода Листинга 87.

Рис. 110: Результат выполнения Листинга 87.
Слово нарисованное мышью по SVG пути. Фаза 2.

Суть работы приложения Листинга 87 заключается в том, что мы отрисовываем линии от вершин букв к курсору мыши. Отрисовка про-

исходит только в том случае, если курсор мыши находится на достаточно близком расстоянии от вершины.

Листинг 87: Рисуем слово мышью

```
1
2  PShape drawingSVG;
3  PShape[] borders;
4  int vertexCount;
5
6  void setup(){
7    size(600, 300);
8    smooth();
9    background(255);
10   strokeWeight(1);
11   noFill();
12   drawingSVG = loadShape("m1.svg");
13   borders = new PShape[6];
14   borders[0] = drawingSVG.getChild("path4166");
15   borders[1] = drawingSVG.getChild("path4168");
16   borders[2] = drawingSVG.getChild("path4170");
17   borders[3] = drawingSVG.getChild("path4172");
18   borders[4] = drawingSVG.getChild("path4174");
19   borders[5] = drawingSVG.getChild("path4176");
20 }
21
22 void draw(){
23   for(int j = 0; j < 6; j++){
24     vertexCount = borders[j].getVertexCount();
25       for(int i = 0; i < vertexCount-1; i+=1){
26         float vx = borders[j].getVertexX(i) + 50;
27         float vy = borders[j].getVertexY(i) - 750;
28         stroke(0, 50);
29         if(dist(mouseX,mouseY, vx, vy) < 10){
30           line(mouseX, mouseY, vx, vy);
31         }
32     }
33   }
34   //filter(BLUR, 1);
35 }
```

По сравнению с двумя предыдущими кодами, объем кода Листинга 87 меньше: мы убрали логику изменения цвета линий, но добавили логику вычисления расстояния. В строке 29 мы вычисляем расстояние между текущей вершиной и курсором мыши с помощью метода *dist()*. Этот метод делает несложный геометрический расчет расстояния между двумя точками. Если это расстояние меньше 10, то отрисовывается линия. Вы можете менять это расстояние, получая интересные художественные эф-

фекты для своих работ.

Таким образом, мы с вами рассмотрели работу со шрифтами и текстом с использованием как стандартного класса *PFont*, так и преобразования текста в кривые с последующей работой в SVG формате.

12 Работа с вэб-камерой

Работа с веб-камерой, которая подает видеопоток в режиме реального времени, часто используется цифровыми художниками. Все варианты использования видеопотока можно разделить на две группы. Первая группа представляет из себя обработку видео с помощью библиотек компьютерного зрения. В этом случае вы получаете возможность определять объекты в видеокадре, распознавать лица, жесты и т.п. Этот принцип используется, например,в контроллере Kinect.

Вторая группа методов работы с видеопотоком не подразумевает использования библиотек компьютерного зрения: поставленные художественные задачи решают довольно простыми методами. Работа в этом случае заключается в последовательной обработке кадров видеопотока. Кадр видеопотока представляет собой растровое изображение, примеры использования которого были даны в предыдущих главах.

12.1 Базовое использование изображения с вэб-камеры

Веб-камера является самым доступным инструментом получения видеопотока. Камера работает по принципу скоростного фотоаппарата: она делает фотографии с определенной скоростью, например, 12 кадров в секунду. У каждой камеры свои характеристики: например существуют высокоскоростные камеры на тысячу кадров в секунду. Размер кадра, пропорции его сторон и его цветовая схема (черно-белая, цветая) также зависят от типа камеры.

Учитывая все это, можно поблагодарить разработчиков Processing, которые взяли на себя большой объем трудностей по организации работы с веб-камерой. Processing справится даже, например, с несовпадением частоты работы камеры и нашего приложения справляется Processing.

Базовая комплектация Processing, к сожалению, не может охватить все разнообразие креативных задач. Работа с видеопотоками может выполняться с помощью нескольких различных библиотек. Чтобы подключить библиотеки в Processing, нужно выполнить следующие два шага:

1. библиотеку необходимо загрузить и установить,
2. в первых строках файла необходимо написать директиву о включении библиотеки.

Для установки библиотеки на компьютер в панели управления требуется выбрать: *Sketch -> Import Library...* и далее *Add Library....* Пе-

ред вами откроется окно со списком всех официальных библиотек для Processing. Вам только остается выбрать необходимую и кликнуть по ней, следуя короткой инструкции. После установки новой библиотеки у вас появятся примеры ее использования. В Processing хорошим тоном разработки библиотек считается создание примеров. Они будут располагаться во разделе со всеми остальными «штатными» примерами.

После установки библиотеки нам необходимо подключить ее к себе в программу. Для этого снова выберем в панели управления *Sketch -> Import Library...* и далее ту библиотеку, которая нам требуется. По клику на имя библиотеки в нашем файле появятся строка(и) с указанием импорта: *import processing...* Таким образом, мы теперь можем работать со сторонними библиотеками.

У каждой библиотеки есть свои правила и описания работы, которые выходят за рамки этого издания, но общий принцип остается одинаковым, так что рассмотрев работу с одной библиотекой, можно в общих чертах представить принцип работы других библиотек. В наших примерах будет показана работа с одной веб-камерой, хотя в Processing есть возможность работы с несколькими камерами и переключения между ними. Мы рассмотрим примеры работы с библиотекой *processing.video*. Базовый пример использования потока с веб-камеры представлен в коде Листинга 88.

В результате работы программы вы увидите в окне приложения видеопоток с вашей веб-камеры, которая, видимо, будет снимать вас же самих (если веб-камера расположена на ноутбуке). Во второй строке Листинга 88 мы подключаем необходимую библиотеку. Отметим, что директиву включения библиотек можно писать и вручную, не прибегая к Processing IDE панели управления, но перед этим библиотека должна быть установлена. Таким образом, строка 2 является обычной строкой кода, она ничем не отличается от всех остальных.

Листинг 88: Снимаем себя на веб-камеру

```
1
2  import processing.video.*;
3
4  Capture video;
5
6  void setup() {
7    size(640, 480);
8    smooth();
9    background(0);
10   noStroke();
11   video = new Capture(this, width, height);
```

```
12    video.start();
13  }
14
15  void draw() {
16
17    if (video.available()) {
18      video.read();
19    }
20    pushMatrix();
21    scale(-1,1);
22    image(video,-width,0);
23    popMatrix();
24  }
```

В 4ой строке мы объявили свойство *video*, которое является ссылкой на объект класса *Capture*. Это класс написали разработчики библиотеки *processing.video* и снабдили его необходимой документацией, так что смотреть его исходные коды нам не придется. В 11-й строке в правой части мы создаем объект класса *Capture*. В его конструктор мы передаем три аргумента. Первый аргумент представляет собой ссылку *this*. Ссылку *this* мы уже использовали, когда говорили о понятиях класса и объекта. Тогда же мы упоминали, что ссылка *this* указывает на объект текущего класса. В основном «поле» работы Processing мы не видим объявления класса: он скрыт от нас во избежание лишней путаницы. А ссылка *this*, естественно, доступна и библиотека *processing.video* требует ее передать в конструктор класса *Capture* первым аргументом, а что он с ней будет делать, нам знать не требуется. Вторым и третьим аргументом мы передаем размеры окна приложения.

После того как мы создали объект класса *Capture*, мы связываем его знаком равенства со свойством *video*. Это свойство – ссылку *video* – мы используем для доступа к созданному нами объекту. Первое, что мы делаем – «включаем» камеру с помощью вызова метода *start()* в 12-й строке.

Как мы уже говорили, скорость работы камеры и скорость отрисовки изображения на экране не совпадают. Поэтому прежде чем получить изображение с веб-камеры, мы должны проверить, готова ли камера нам его передать. Эту проверку мы производим в строке 17. Класс *Capture* обладает методом *available()*, который возвращает истину (TRUE), если камера готова отдать нам изображение, и ложь (FALSE), если изображение не готово. Если изображение готово, то мы читаем его в строке 18.

В строке 18 мы вызываем метод *read()* от объекта по ссылке *video* класса *Capture* для того чтобы массив пикселей камеры поместить в

окружение Processing. Визуально мы не заметим этого вызова, однако если мы его не сделаем, то у нас не будет доступного изображения для работы, так как этот метод копирует изображение с камеры в нашу программу.

Изображение с веб-камеры ноутбука ничем не отличается от рассмотренных нами растровых изображений, поэтому мы можем отрисовать его с помощью вызова метода *image()* (строка 22). Обратите внимание, что веб-камера ноутбука даст вам зеркальное отображение действительности. Если вы хотите «обратить» его, то можно воспользоваться матричным преобразованием системы координат, вызвав метод *scale()* с аргументами *-1 , 1*.Таким образом мы получили базовое приложение по работе с веб-камерой в Processing.

12.2 Обработка видеопотока в режиме реального времени

Изображение с веб-камеры является растровым. С растром мы работали с помощью класса *PImage*. При работе с веб-камерой нам не требуется читать изображения из файла, как мы делали это раньше с помощью метода *loadImage()*, метод *get()* класса *Capture* вернет нам изображение из камеры.

Еще одной особенностью работы с веб-камерой является восприятие человеком скорости работы. Когда мы отрисовку собственных движений на экране, нам кажется, что приложение работает в режиме реального времени, т.е. очень быстро. Однако наши предыдущие примеры работали не медленнее, просто ваше восприятие их работы не было личностно мотивировано. Так что эффект от работы приложения с веб-камерой может быть очень сильным.

Рассмотрим работу приложения по обработке видеопотока режиме реального времени на примере кода Листинга 89.

Листинг 89: Рисуем вертикальные полосы

```
1
2  import processing.video.*;
3
4  Capture video;
5  int mySize;
6
7  void setup() {
8    size(640, 480);
9    smooth();
10   background(0);
```

```
11    noStroke();
12    video = new Capture(this, width, height);
13    video.start();
14  }
15
16  void draw() {
17    if (video.available()) {
18      video.read();
19      if (frameCount % 3 == 0 ){
20        blendMode(ADD);
21      } else {
22        blendMode(BLEND);
23      }
24      float myX = random(0, width);
25      PImage myImg = video.get((int)myX,0,mySize,height);
26      image(myImg,(int)myX,0);
27      mySize = (int)random(1,50);
28    }
29  }
```

В строках 4, 12 и 13 мы проводим описанную выше работу по включению камеры в нашу программу. Если изображение с камеры доступно, то мы попадаем в строки с 18-й по 27-ю. В 18-й строке изображение с камеры загружается в объект класса *Capture* по ссылке *video*.

Далее мы каждый третий фрейм переключаем режим наложения для достижения художественного эффекта (строки с 19-й по 23-ю). Затем в четырех строках мы описываем всю основную логику приложения, результат выполнения которого показан на Рисунке 111.

В 24-й строке Листинга 89 мы объявляем переменную *myX*, в которую копируем случайное число из диапазона от 0 до ширины окна. Таким образом, каждый вызов метода *draw()*, при условии наличия изображения с камеры, мы получаем новое случайное число. В 25-й строке мы вызываем метод *get()* по ссылке *video*. С помощью этого метода мы можем получить объект уже знакомого нам класса *PImage*. Этот метод возвращает объект класса *PImage* при четырех аргументах. Мы можем указать прямоугольную часть изображения, которую нам требуется извлечь из видеокадра. Первые два аргумента отвечают за положение левого верхнего угла прямоугольника, вторые два аргумента – за ширину и высоту прямоугольника. В строке 25 в объекте *PImage* у нас есть часть изображения – прямоугольник шириной *mySize* и высотой во все окно. Место, откуда будет копироваться этот прямоугольник, мы указываем переменной *myX*, т.е. случайным образом. Итак, мы случайно выбираем из видеокадра прямоугольник разной ширины, высотой с видеокадр.

Теперь нам осталось отрисовать его с помощью метода *image()* в 26-й

Рис. 111: Результат выполнения Листинга 89.
Фазы работы приложения с видеопотоком и вертикальными полосами.

строке и обновить значение переменной *mySize* в строке 27. В результате работы мы отрисовываем прямоугольники на экране в случайном порядке, причем на каждом прямоугольнике запечатлен свой момент времени. Мы получили динамическую, живую инсталляцию всего за несколько строк кода работы с веб-камерой.

Рассмотрим в качестве примера модификацию кода Листинга 90. Некоторые параллели с этой работой можно найти в проекте Fragmentation01 и Fragmentation02 от дизайнера-фотографа Рез Н (Rez N. aka @zproc aka @0x90). Его проект можно посмотреть по ссылкам http://lotek.fr/pulse/2015/07/20/fragmentation02/ , http://lotek.fr/pulse/2015/07/20/displacement01/

Листинг 90: Рисуем прямоугольники из видеокадров

```
1  import processing.video.*;
2
3  Capture video;
4
5  void setup() {
6    size(640, 480);
7    smooth();
8    background(0);
9    noStroke();
10   video = new Capture(this, width, height);
11   video.start();
12 }
13
14 void draw() {
15
16   if (video.available()) {
17     video.read();
18     if(mouseX > 20 && mouseY > 20){
19       if(mouseX < width - 20 && mouseY < height - 20){
20         if (frameCount \% 5 == 0 ){
21           blendMode(BLEND);
22           noTint();
23           fill(0, 10);
24           rect(0,0,width, height);
25         } else {
26           blendMode(ADD);
27       }
28
29       float x = mouseX + random(-100,100);
30       float y = mouseY + random(-100,100);
31       float wx = 20 + random(200);
32       float wy = 20 + random(200);
33       PImage tmpImg = video.get(mouseX,  mouseY, (int)wx,(
```

```
            int)wy);
34          tint(random(50, 255), random(50, 255), random(50,
               255));
35          image(tmpImg,x, y);
36          }
37        }
38      }
39    }
```

Приложение работает по схожему с предыдущим принципу: мы получаем видеокадр и, в зависимости от положения курсора мыши, копируем из него прямоугольную область. Далее, меняя режимы наложения, мы отрисовываем этот прямоугольник рядом с тем местом, где он должен быть. Результат выполнения кода Листинга 89 представлен на Рисунке 112.

Рис. 112: Результат выполнения Листинга 90.
Фазы работы приложения где рисуются тонировнанные прямоугольники из видеокадров.

В строках 18 и 19 мы проверяем, находится ли курсор мыши в «активной» области окна. Если курсор действительно находится в ней, то

мы отрисовываем полупрозрачный черный прямоугольник. Мы это делаем для того, чтобы постепенно очищать окно приложения. Однако мы можем и убрать мышь из активной области.

В строках 29 и 30 происходит формирование координат случайным образом вокруг курсора мыши. Так же, случайным образом, мы формируем ширину и высоту нашего прямоугольника для выделения области копирования изображения с видеокадра и в 33-й строке производим это копирование. После этого остается отрисовать получившееся изображение тонированным на экране с помощью вызова метода *imgae()* в 35-й строке.

Итак, мы рассмотрели два интересных примера работы с веб-камерой в Processing. Работа с любым дополнительным типом устройств, в том числе и с веб-камерой, имеет свои нюансы в зависимости от операционной системы и типа устройства. Для обеспечения работы Processing разные операционные системы могут использовать разные библиотеки. Таким образом, при переносе приложений с одного компьютера на другой вы должны быть готовы к возможным внештатным ситуациям.

Обработку видеокадров интересно совмещать со всеми возможностями Processing, о которых мы говорили выше. После изучения материала следующего раздела поле для ваших художественных экспериментов расшириться: вы сможете совмещать видеоизображения с системами частиц для создания красочных динамических композиций.

13 Системы частиц

Системы частиц – это, в каком-то смысле, вершина художественных возможностей Processing. Принципы ООП позволяют дизайнерам управлять большим количеством частиц, создавая из них оригинальные произведения. Речь идет именно об управлении, поэтому мы и вынесли разговор о частицах в главу, отдельную от темы массивов. Безусловно, системы частиц используют массивы, объекты и все другие возможности Processing, которые мы рассматривали.

Система частиц не является встроенной конструкцией языка. Она является, скорее, логической конструкцией, которая может быть реализована по-разному. Чаще всего система частиц состоит из класса частицы, в котором описываются свойства частицы, и класса контроллера, в котором описываются управляющие функции. Управление частицами выносится в контроллер, однако программист сам решает, какой объем управления необходимо вынести в контроллер.

13.1 Класс частицы и контроллера

Начнем знакомиться с системами частиц на примере кода Листинга 91. При клике мыши в этом приложении будет образовываться множество крестиков, вращающихсяь вокруг своей оси, и разлетающихся в разные стороны. Результат работы приложения показан на Рисунке 113.

Начнем рассматривать код Листинга 91 с основного потока выполнения программы, т.е. сразу с 70-й строки. В ней мы объявляем ссылку *pCont*, создаем объект класса *ParticleController* и связываем его с объявленной ссылкой. Таким образом создается контроллер системы частиц. В 75-й строке мы вызываем его метод *createParticles()* и передаем ему 3 аргумента. Согласно своему наименованию, этот метод создает сами частицы, заполняя ими хранилище контроллера.

В 83-й строке мы вызываем метод *upDate()* и передаем ему координаты курсора мыши. Очевидно, что тем самым мы посылаем сигнал контроллеру, чтобы он обновил координаты частиц в зависимости от аргументов. В 84-ой строке мы вызываем метод *render()*, сообщая контроллеру о том, что требуется отрисовка частиц.

Итак, мы рассмотрели базовую логику работы, не вдаваясь в детали реализации классов. Получается, что работа строится следующим образом: создаем контроллер частиц, создаем частицы, обновляем положения частиц, отрисовываем частицы. Причем, обновление и отрисовка происходят с каждым вызовом метода *draw()*, что позволяет нам видеть движение частиц. Обратите внимание, что мы пока не говорим, какие имен-

но это будут частицы, какое именно движение они будут совершать и т.п. – мы говорим про общую концепцию работы с системами частиц. Например, в нашем приложении частица – это крестик, а в другом приложении это будет один пиксель. Возможно также создать один контроллер для управления разными классами частиц. Однако мы пока ограничимся одни контроллером и одним классом частиц.

Листинг 91: Первая система частиц

```
class LightParticle {
  float x,y,cx,cy,size,step,dist,angle;
  float counter;

  LightParticle(float cx, float cy, float size, float step,
      float dist, float angle){
    this.cx = cx;
    this.cy = cy;
    this.size = size;
    this.dist = dist;
    this.step = step;
    this.angle = angle;
  }

  void render(){
    counter += step;
    if(counter > TWO_PI){
      counter = 0;
    }
    if(counter < 0){
      counter = TWO_PI;
    }
    float x1 = x - sin(counter)*(size/2);
    float x2 = x + sin(counter)*(size/2);
    float y1 = y - cos(counter)*(size/2);
    float y2 = y + cos(counter)*(size/2);

    line(x1,y1,x2,y2);

    float x3 = x - sin(counter + PI/2)*(size/2);
    float x4 = x + sin(counter + PI/2)*(size/2);
    float y3 = y - cos(counter + PI/2)*(size/2);
    float y4 = y + cos(counter + PI/2)*(size/2);

    line(x3,y3,x4,y4);

  }
}

```

```
39  class ParticleController {
40    ArrayList<LightParticle> ar = new ArrayList<LightParticle
        >();
41
42
43    void createParticles(float x, float y, int number){
44      for(int i = 0; i < number; i++){
45        LightParticle lObj = new LightParticle(x, y, random
            (5,50), random(-0.5,0.5), random(0.5,5), random
            (0,360));
46        ar.add(lObj);
47      }
48    }
49
50    void upDate(float x, float y){
51      for(LightParticle tmp : ar){
52        tmp.x = tmp.x + tmp.dist;
53        tmp.y = tmp.y + tmp.dist;
54      }
55    }
56
57    void render(){
58      stroke(250,200);
59      for(LightParticle tmp : ar){
60        strokeWeight(tmp.size/5);
61        pushMatrix();
62        translate(tmp.cx, tmp.cy);
63        rotate(radians(tmp.angle));
64        tmp.render();
65        popMatrix();
66      }
67    }
68  }
69
70  ParticleController pCont = new ParticleController();
71
72  void setup(){
73    size(500,500);
74    smooth();
75    pCont.createParticles(width/2,height/2,10);
76    background(50);
77  }
78
79  void draw(){
80    //fill(0,20);
81    //rect(0,0,width,height);
82    background(50);
83    pCont.upDate(mouseX, mouseY);
84    pCont.render();
```

```
85  }
86
87  void mouseClicked(){
88    pCont = new ParticleController();
89    pCont.createParticles(mouseX,mouseY,50);
90  }
```

Класс частицы *LightParticle* объявляется с 1-й по 37-ю строки. Необходимо объявить ряд свойств частицы, которые служат для хранения координат (*x, y*), координат центра (*cx, cy*), размера частицы (*size*), счетчика (*counter*), шага изменения счетчика(*step*), величины изменения положения (*dist*) и угла движения (*angle*). Координаты и значения счетчика частицы будут определяться по ходу выполнения программы, а вот остальные свойства мы устанавливаем с помощью контроллера с 5-й по 12-ю строку.

Единственный метод, который у нас есть в классе *LightParticle* – это метод *render()* (с 14-й по 36-ю строки) в котором происходит отрисовка крестика. С 15-й по 21-ю строку мы обновляем счетчик *counter*, который может как увеличиваться, так и уменьшаться в зависимости от знака свойства *step*. Чтобы отрисовать первый отрезок в 27-й строке, нам требуется определить координаты его концов. Их мы находим с 22-й по 25-ю строки. Этот отрезок будет вращаться относительно координат X и Y в зависимости от счетчика и размером *size*. Такую же логику мы используем и для второй линии, только прибавляем к счетчику *PI/2*, чтобы вторая линия была перпендикулярна первой.

Мы рассмотрели класс частицы *LightParticle*, который по большей части состоит из свойств, устанавливаемых с помощью конструктора, и метода отрисовки, где используются свойства класса для формирования изображения на экране.

Рассмотрим класс контроллера частиц *ParticleController*: он объявлен с 39-й по 68-ю строку. Как мы уже говорили, контроллер должен иметь хранилище для частиц. Роль этого хранилища играет *ArrayList*. В 40-й строке мы объявили свойство *ar*, с которым связали объект класса *ArrayList*. В нем будут храниться объекты класса *LightParticle*. Других свойств у нашего контроллера нет.

У контроллера есть три метода. Метод *cteateParticle()* принимает координаты и количество частиц. Из названия метода ясно, что он занимается созданием частиц: действительно, в 44-й строке мы попадаем в цикл, который ограничен аргументом *number* по созданию частиц. Объект класса *LightParticle* создается в 45-й строке, связываясь со ссылкой *lObj*. При его создании мы передаем в конструктор класса *LightParticle* необходимые аргументы, порядок которых указан при объявлении кон-

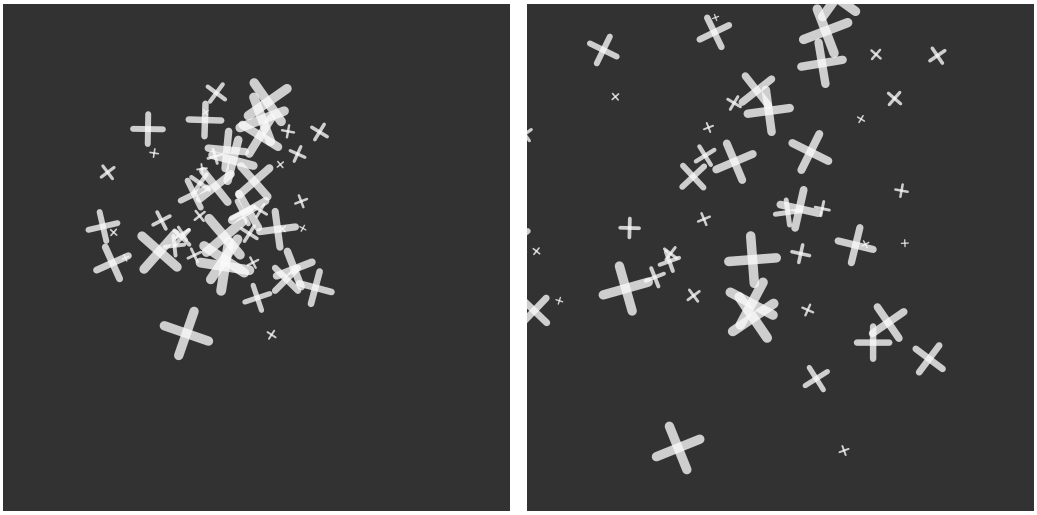


Рис. 113: Результат выполнения Листинга 91.
Система частиц, состоящая из множества крестиков.

структора в 5-й строке. Как только частица создана, мы добавляем ее в хранилище в строке 46.

Метод *upDate()* принимает два аргумента и, обходя все частицы хранилища, он изменяет координат каждой из них (строки 52 и 53).

Метод *render()*, также обходя всё хранилище частиц, отрисовывает каждую из них. В 60-й строке мы задаем толщину линии в зависимости от размеров текущей частицы. В следующей строке мы запоминаем состояние системы координат, чтобы в 62-й строке переместить ее в точку центра *cx, cy*. В нашем примере точки центра совпадают для всех частиц, но они могут иметь и уникальные координаты. В 63-й строке мы вращаем систему координат относительно точки центра на угол *angle*, и в той же строке вызываем метод *render()* нашей частицы. После этого мы возвращаем систему координат в прежнее состояние. Обратите внимание, что метод *render()* не изменяет свойств частиц, а только использует их для отрисовки.

Задание 1. Раскройте комментарии строк 80 и 81 и закройте комментарием строку 82, для получения интересные художественные эффекты. Попробуйте поменять количество частиц, размер и остальные свойства.

В коде Листинга 91 в строках с 61-й по 65ю мы использовали матрич-

ное преобразование систем координат. Однако, во первых, не обязательно его использовать, во вторых, если нам требуется определить текущие координаты частицы, мы должны сделать обратные преобразования.

В следующем примере мы оставим класс *LightParticle* без изменений. Сейчас мы хотим, чтобы наши частицы были выкрашены в определенный цвет. Цвет будем брать из заранее заготовленного изображения. Мы будем реализовывать приложение, похожее по принципу работы на код Листинга 78 (результат выполнения которого представлен на Рисунке 92). Рассмотрим код нашего приложения в Листинге 92.

Листинг 92: Система частиц с растровами изображениями

```
1
2  class LightParticle {
3    float x, y, cx, cy, size, step, dist, angle;
4    float counter;
5
6    LightParticle(float cx, float cy, float size, float step,
         float dist, float angle) {
7      this.cx = cx;
8      this.cy = cy;
9      this.size = size;
10     this.dist = dist;
11     this.step = step;
12     this.angle = angle;
13   }
14
15   void render() {
16     counter += step;
17     if (counter > TWO_PI) {
18       counter = 0;
19     }
20     if (counter < 0) {
21       counter = TWO_PI;
22     }
23     float x1 = x - sin(counter)*(size/2);
24     float x2 = x + sin(counter)*(size/2);
25     float y1 = y - cos(counter)*(size/2);
26     float y2 = y + cos(counter)*(size/2);
27
28     line(x1, y1, x2, y2);
29
30     float x3 = x - sin(counter + PI/2)*(size/2);
31     float x4 = x + sin(counter + PI/2)*(size/2);
32     float y3 = y - cos(counter + PI/2)*(size/2);
33     float y4 = y + cos(counter + PI/2)*(size/2);
34
35     line(x3, y3, x4, y4);
```

```
36    }
37  }
38
39  class ParticleController {
40    ArrayList<LightParticle> ar = new ArrayList<LightParticle
        >();
41    int counter;
42
43    void createParticles(float x, float y, int number) {
44      for (int i = 0; i < number; i++) {
45        LightParticle lObj = new LightParticle(x, y, random
            (5, 30), random(-0.5, 0.5), random(0.5, 5), random
            (0, 360));
46        ar.add(lObj);
47      }
48    }
49
50    void upDate() {
51      counter+=1;
52      for (LightParticle tmp : ar) {
53        tmp.x = tmp.cx + sin(radians(tmp.angle))*(tmp.dist*
            counter);
54        tmp.y = tmp.cy - cos(radians(tmp.angle))*(tmp.dist*
            counter);
55      }
56    }
57
58    void render(PImage pimg) {
59      for (LightParticle tmp : ar) {
60        strokeWeight(tmp.size/5);
61        if(tmp.x > 0 && tmp.x < 500){
62          if(tmp.y > 0 && tmp.y < 500){
63            int loc = (int)tmp.x + (int)tmp.y*pimg.width;
64            float r = red(pimg.pixels[loc]);
65            float g = green(pimg.pixels[loc]);
66            float b = blue(pimg.pixels[loc]);
67            stroke(r,g,b);
68          }
69        }
70        tmp.render();
71      }
72    }
73  }
74
75  ParticleController pCont = new ParticleController();
76  PImage pimg;
77  //PImage pimg1;
78
79  void setup() {
```

```
80    size(500, 500);
81    smooth();
82    pCont.createParticles(width/2, height/2, 10);
83    background(50);
84    pimg = loadImage("eye.jpg");
85  //  pimg1 = loadImage("eye.jpg");
86    pimg.loadPixels();
87  }
88
89  void draw() {
90    background(0);
91    //pimg1.filter(GRAY);
92    //image(pimg1, 0, 0);
93
94    pCont.upDate();
95    pCont.render(pimg);
96  }
97
98  void mouseClicked() {
99    pCont = new ParticleController();
100   pCont.createParticles(mouseX, mouseY, 500);
101 }
```

Результат работы кода Листинга 92 показан на Рисунках 114 и 115. Поведение частиц не изменилось по сравнению с предыдущим примером: они также разлетаются от центра системы. А вот цвет их поменялся.

Рассмотрим изменившиеся по сравнению с предыдущим кодом и добавленные строки кода Листинга 93. В классе *particleController* мы поменяли логику методов *upDate()* и *render()* и ввели вспомогательное свойство counter в 41-й строке.

Метод *upDate()* обходит хранилище частиц и устанавливает каждой значение координат. Так же как и в предыдущем примере, когда мы делали перенос системы координат в *cx, cy*, частицы будут вращаться вокруг центра. С помощь синуса и косинуса мы поворачиваемся на угол *angle* и по воображаемому вектору отходим от центра на расстояние *tmp.dist*counter*. Счетчик *counter* увеличивается с каждым вызовом метода *upDate()*. Таким образом, мы знаем координаты положения частицы, которые будем использовать в методе *render()*.

Метод *render()* объявлен с 57-й по 72-ю строку. В этом примере он принимает аргументом объект класса *PImage* по ссылке *pimg*. И далее мы обходим всё хранилище частиц. Для каждой частицы проверяем ее координаты – не вышли ли они за пределы размера изображения (строки с 61-й по 62-ю). Если координаты частицы находятся в поле изображения, то мы вычисляем цвет соответствующего пикселя (строки с 63-й по 66-ю) и затем задаем его для отрисовки обводки (строка 67). Больше

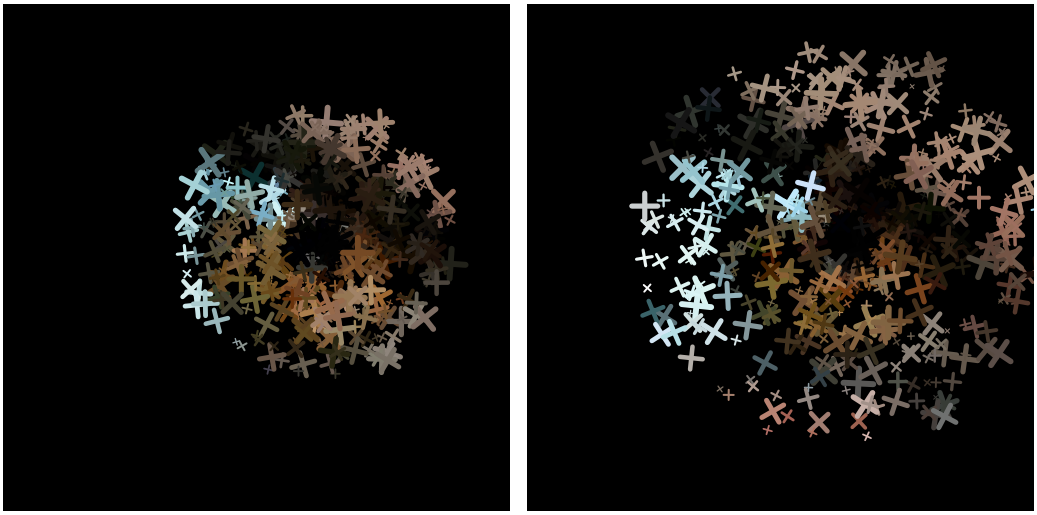


Рис. 114: Результат выполнения Листинга 92,
Фазы системы частиц с растровым изображением.

никаких изменений в классе *particleController* мы не делали.

Мы изменили общую логику приложения, добавив свойство *pimg* в 76-й строке – ссылку на объект класса *PImage*. В 84-й строке в методе *setup()*, произвели загрузку изображения и связали объект класса *PImage* со свойством *pimg*. В 86-й строке произвели загрузки пикселей. Осталось передать ссылку на наш объект *PImage* в метод *render()* нашего контроллера.

В результате мы получили черный фон и разлетающиеся частицы, окрашенные в цвета нашего изображения. Если вы откроете комментарии в строках 77, 85, 91 и 92, то увидите другую картину. В этих строках мы создаем еще один объект *PImage*, тоже загрузив изображение. В 91-й строке мы применяем к нему фильтр градаций серого,в 92-й строке отрисовываем его. Система частиц отрисовывается поверх черно-белого изображения. Таким образом цветные крестики раскрашивают черно-белый фон (см. Рисунок 116).

Если требуется работа с несколькими контроллерами системами частиц, например, так, чтобы по клику мыши появлялся новый контроллер, при этом старый никуда не исчезал, то эту задачу также поможет решить класс *ArrayList*, только в таком случае в его объекте будут храниться контроллеры.

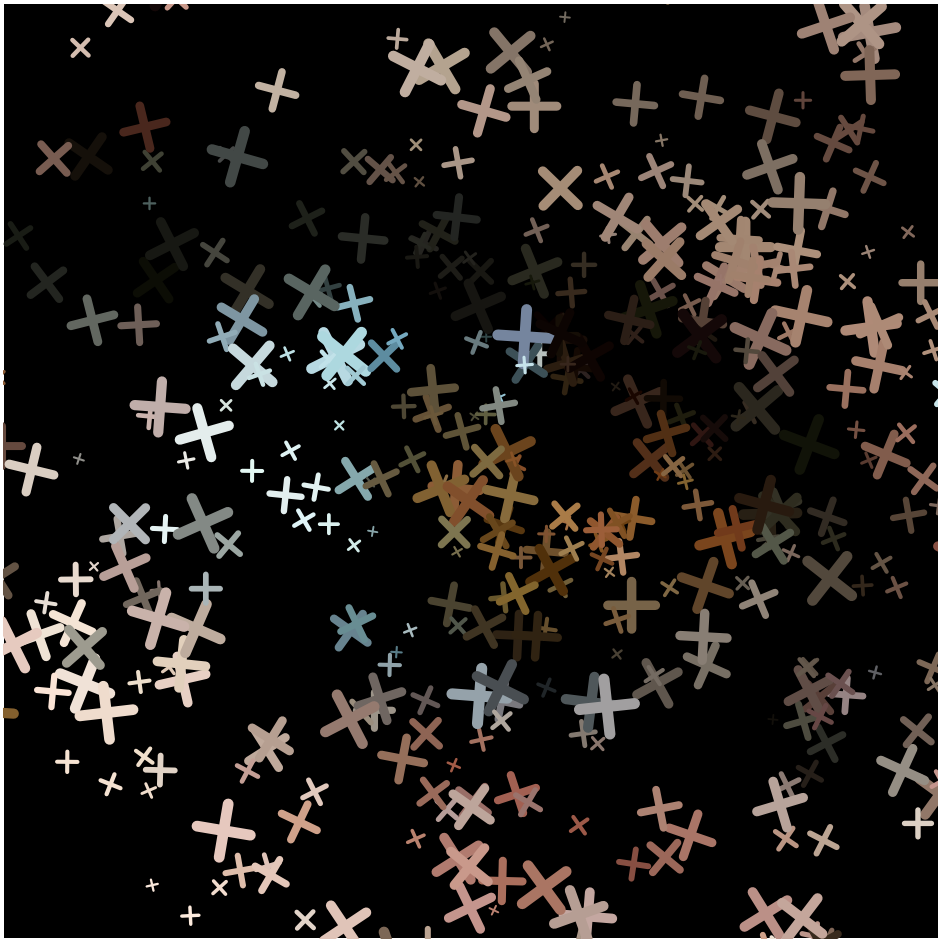


Рис. 115: Результат выполнения Листинга 92,
Система частиц с растровым изображением. Фаза 3.

13.2 Несколько контроллеров частиц

В каждом контроллере у нас сотни частиц (или столько, сколько указано), так что количество контроллеров будет безусловно влиять на производительность приложения. Нам надо позаботиться о том, чтобы удалять те частицы, координаты которых находятся за пределами экрана – тогда мы освободим ресурсы, которые они занимали впустую. В Листинге 93 приведен только код основной части программы, так как класс *LightParticle* не поменялся. Для того чтобы запустить это приложение, не забудьте скопировать класс *LightParticle* в код Листинга 93.

Во второй строке кода Листинга 93 мы объявляем свойство *pcs*, связанное с объектом класса *ArrayList*, в котором будут храниться объекты класса *ParticleController*. Пока наше хранилище пусто.

Рис. 116: Результат выполнения Листинга 92,
Система частиц рисуется по верх растрового изображения.

Заполнять хранилище мы будем по клику мыши. В 23-й строке мы создаем объект класса *ParticleController* и связываем его со ссылкой *pCont*. По ней мы вызываем метод *pCont.createParticles()*, в который передаем так же, как и в предыдущем примере, координаты мыши и количество частиц. После этого в 25-й строке мы добавляем наш контроллер в хранилище контроллеров вызовом метода *pcs.add()* по ссылке *pcs*. Если мы кликнем еще раз, то еще один контроллер добавится в хранилище. Теперь давайте разберем удаление (чистку) частиц.

В предыдущем примере в методе *render()* мы проверяли координаты частиц, и если частица была вне поля изображения, то не задавали ей цвет. В строках с 18-й по 23-ю в методе *upDate()* мы делаем схожую проверку. С целью хранения ссылок на частицы, которые мы будем удалять, в 14-й строке вводится ссылка на хранилище таких частиц. После того как мы прошли по всем частицам циклом в 15-й строке, у нас появился набор частиц для удаления. Теперь нам надо пройтись и по нему: это делается в строках 26-28. В классе *ArrayList* есть метод *remove()*, который удаляет объект из хранилища, если находит его там по ссылке, которую принимает аргументом.

Для того чтобы наглядно проверить количество элементов в хранили-

ще, мы воспользуемся еще одним методом класса *ArrayList*. Метод *size()* возвращает количество элементов хранилища. В строках с 77-й по 82-ю, если пользователь нажимает на кнопку «*q*», мы обходим хранилище контроллеров и у каждого обращаемся к свойству *ar*, вызывая метод *size()*. Количество элементов мы распечатываем в консоли. Результат работы кода Листинга 93 показан на Рисунке 117.

Листинг 93: Несколько контроллеров частиц

```
1
2  class ParticleController {
3    ArrayList<LightParticle> ar = new ArrayList<LightParticle
       >();
4    int counter;
5
6    void createParticles(float x, float y, int number) {
7      for (int i = 0; i < number; i++) {
8        LightParticle lObj = new LightParticle(x, y, random
           (2, 10), random(-0.5, 0.5), random(0.5, 5), random
           (0, 360));
9        ar.add(lObj);
10     }
11   }
12
13   void upDate() {
14     ArrayList<LightParticle> remove = new ArrayList<
         LightParticle>();
15     for (LightParticle tmp : ar) {
16       tmp.x = tmp.cx + sin(radians(tmp.angle))*(tmp.dist*
           counter);
17       tmp.y = tmp.cy - cos(radians(tmp.angle))*(tmp.dist*
           counter);
18       if(tmp.x < 0 || tmp.x > 500){
19         remove.add(tmp);
20       }
21       if(tmp.y < 0 || tmp.y > 500){
22         remove.add(tmp);
23       }
24     }
25
26     for(LightParticle tmp : remove){
27       ar.remove(tmp);
28     }
29   }
30
31   void render(PImage pimg) {
32     counter+=1;
33     for (LightParticle tmp : ar) {
```

```
34        strokeWeight(tmp.size/5);
35        if(tmp.x > 0 && tmp.x < 500){
36          if(tmp.y > 0 && tmp.y < 500){
37            int loc = (int)tmp.x + (int)tmp.y*pimg.width;
38            float r = red(pimg.pixels[loc]);
39            float g = green(pimg.pixels[loc]);
40            float b = blue(pimg.pixels[loc]);
41            stroke(r,g,b);
42          }
43        }
44        tmp.render();
45      }
46    }
47  }
48
49  ArrayList<ParticleController> pcs = new ArrayList<
       ParticleController>();
50
51  PImage pimg;
52
53  void setup() {
54    size(500, 500);
55    smooth();
56    background(50);
57    pimg = loadImage("eye.jpg");
58    pimg.loadPixels();
59  }
60
61  void draw() {
62    background(0);
63    for(ParticleController current : pcs){
64      current.upDate();
65      current.render(pimg);
66    }
67  }
68
69  void mouseClicked() {
70    ParticleController pCont = new ParticleController();
71    pCont.createParticles(mouseX, mouseY, 500);
72    pcs.add(pCont);
73  }
74
75
76  void keyPressed() {
77    if (key=='q') {
78    for(ParticleController current : pcs){
79      int numbers = current.ar.size();
80       println(numbers);
81    }
```

```
82        }
83    }
```

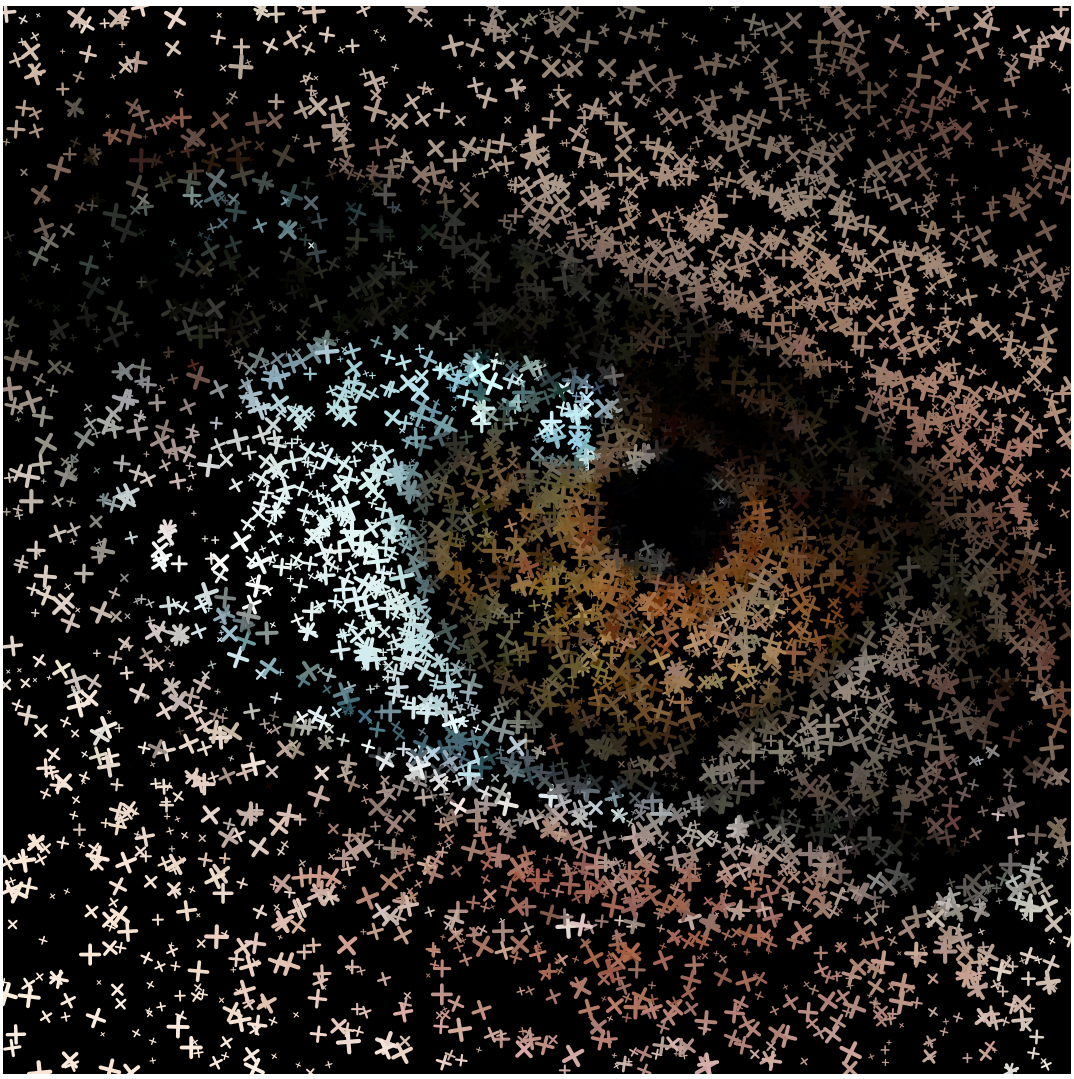


Рис. 117: Результат выполнения Листинга 93.
Несколько контроллеров частиц на одном изображении.

Таким образом можно работать с несколькими контроллерами частиц. Удаление лишних частиц увеличивает производительность программы. Однако частицы, выпущенные из центра, улетают бесконтрольно – давайте посмотрим, каким образом можно заставить их двигаться по-другому.

13.3 Управляемые частицы

В этом параграфе мы подробно рассмотрим управление частицами. Мы уже говорили, что за непосредственное управление частицами отвечает контроллер, который мы и будем использовать. Особенность работы заключается в том, что управляющую переменную, например, положение курсора мыши, лучше передавать аргументом в метод *upDate()* класса контроллера, а не использовать внутри класса контроллера «напрямую» (Листинг 94).

В строке 2 Листинга 94 мы начинаем с объявления класса *EParticle*, который будет отвечать за частицы. В нем есть уже знакомый нам ряд свойств, конструктор и метод *render()*. Конструктор (с 7-й по 13-ю строки) принимает аргументы и присваивает значения соответствующим свойствам класса. Метод *render()* в 16-й строке отрисовывает эллипс. В 18-й строке мы заканчиваем объявление класса *EParticle*.

Листинг 94: Рисуем частицы по сетке

```
1
2  class EParticle {
3    float x, y, cx, cy, size, step, dist;
4    float counter;
5    float mr, mg, mb;
6
7    EParticle(float cx, float cy, float size, float step,
        float dist) {
8      this.cx = cx;
9      this.cy = cy;
10     this.size = size;
11     this.dist = dist;
12     this.step = step;
13   }
14
15   void render() {
16     ellipse(x,y,size,size);
17   }
18 }
19
20 class ParticleController {
21   ArrayList<EParticle> ar = new ArrayList<EParticle>();
22   float counter;
23
24   void createParticles(float size, PImage pimg) {
25     for (int py = 0; py < pimg.height; py+=size) {
26       for (int px = 0; px < pimg.width; px+=size) {
27         EParticle lObj = new EParticle(px, py, size, random
              (-0.2, 0.2), random(20, 500));
```

```
28          int loc = (int)px + (int)py*pimg.width;
29          lObj.mr = red(pimg.pixels[loc]);
30          lObj.mg = green(pimg.pixels[loc]);
31          lObj.mb = blue(pimg.pixels[loc]);
32          ar.add(lObj);
33        }
34      }
35    }
36
37    void upDate(float dimmer) {
38      for (EParticle tmp : ar) {
39        tmp.counter += tmp.step;
40        tmp.x = tmp.cx + sin(tmp.counter)*(tmp.dist*dimmer);
41        tmp.y = tmp.cy - cos(tmp.counter)*(tmp.dist*dimmer);
42      }
43    }
44
45    void render() {
46      noStroke();
47      for (EParticle tmp : ar) {
48        fill(tmp.mr, tmp.mg, tmp.mb);
49        tmp.render();
50      }
51    }
52  }
53
54  ParticleController pc = new ParticleController();
55  PImage pimg;
56
57  void setup() {
58    smooth();
59    background(50);
60    pimg = loadImage("m1.png");
61    size(pimg.width, pimg.height);
62    pimg.loadPixels();
63    pc.createParticles(15, pimg);
64    ellipseMode(CENTER);
65  }
66
67  void draw() {
68    background(0);
69    float var = 0;
70    if(mouseX < width/2){
71      var = map(mouseX, 0, width/2, 1, 0);
72    } else {
73      var = map(mouseX, width, width/2, 1, 0);
74    }
75    pc.upDate(var);
76    pc.render();
```

```
77  }
```

Класс *ParticleController* очень похож на соответствующий класс из предыдущего примера. В нем есть хранилище частиц (строка 21) в виде объекта класса *ArrayList* с элементами класса *EParticle*. Также в нем есть три метода: *createParticle()* – для создания частиц, *upDate()* – для обновления координат частиц и *render()* – для отрисовки.

Результат выполнения кода Листинга 94 показан на Рисунке 118.

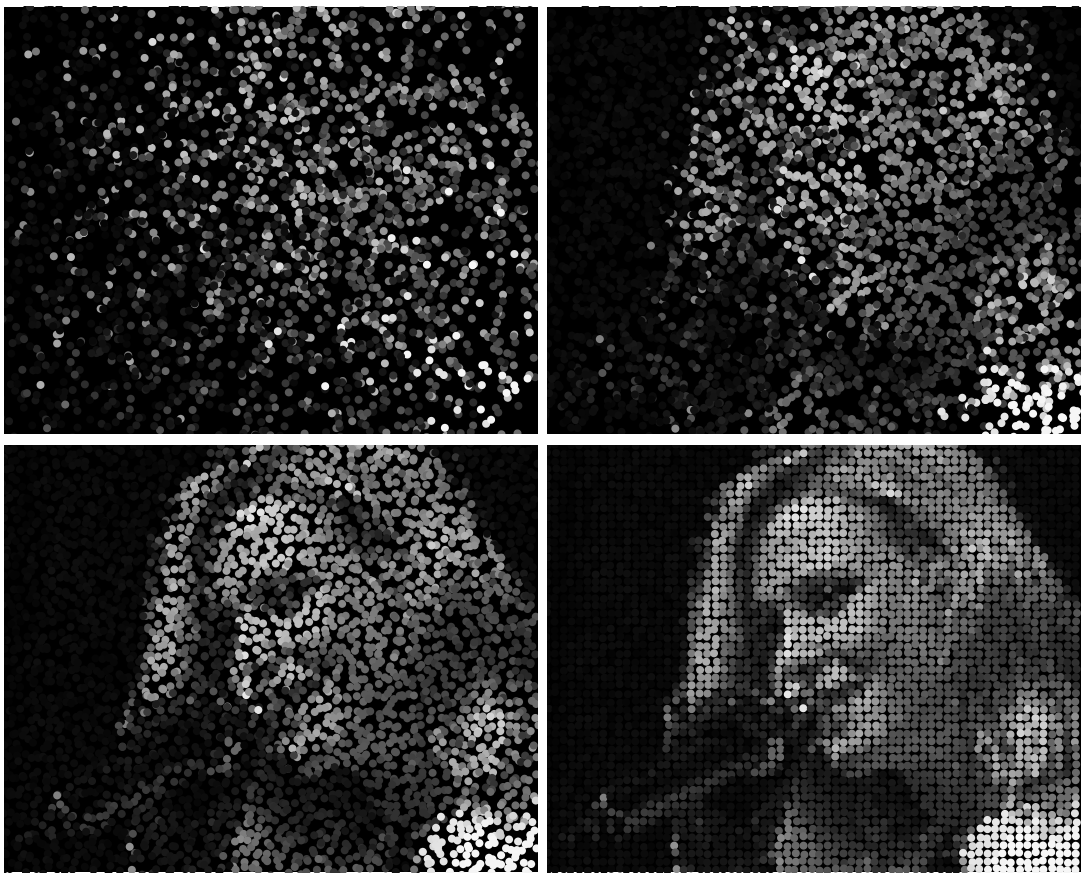


Рис. 118: Результат выполнения Листинга 94.
Фазы работы приложения с частицами формирующими изображение по сетке.

Метод *createParticle()* изменился по сравнению с предыдущим примером. Теперь в 24-й строке, он принимает два аргумента: размер и изображение. Певый аргумент отвечает за размер частицы, второй – за ссылку на объект класса *PImage*. Далее мы обходим в двойном цикле пиксели изображения с шагом – *size*. Делаем мы это для того, чтобы отрисовать

эллипсы размера *size* сеткой, размером с изображение. В 27-й строке мы создаем объект класса *EParticle* и заполняем его свойства. В его конструктор мы передаем текущие значения X и Y, чтобы частичка «запомнила», что ее центр именно в этом месте экрана. Т.е. у каждой частички теперь свой центр со своими координатами (в предыдущем примере центры всех частиц совпадали). С 28-й по 31-ю строку мы определяем цвет пикселя изображения в текущих координатах И заносим эти значения в свойства объекта по ссылке *lObj*. После этого в 32-й строке мы добавляем нашу частицу в хранилище.

Прежде чем рассмотреть метод *upDate()*, обратим внимание на работу с ним из потока программы. В методе *draw()* с 67-й по 77-ю строку мы привычно вызываем методы *upDate()* и *render()*. На этот раз мы не просто вызываем метод *upDate()* – мы передаем ему аргументом переменную *var*. Эту переменную мы определяем в строках 69-74. Если курсор мыши находится в левой части экрана, то работает одна логи метода *map()*, если курсор находится в правой части экрана, то логика как бы отражается. Когда курсор мыши находится в центре окна, то значение переменной var стремится к нулю. Эту переменную мы и передаем в метод *upDate()*.

Метод *upDate()*, который принимает значение от переменной var, копирует его в переменную *dimmer*. Так как мы работаем с простым типом данных (*float*), то тут не идет речи о ссылке, значения копируются. Dimmer мы используем в формировании координат частицы. Если *dimmer* стремится к нулю, то значение координаты стремится к центральному положению: а мы помним, что центральное положение у каждой частицы в этом примере свое. Таким образом мы управляем поведением частиц и получаем интересный художественный результат.

Возможность хранить центральное положение и управлять возвращением в него можно применять не только для работы с изображениями. Вспомните материал раздела работы с SVG файлами и рассмотрите пример совмещения системы частиц с вершинами на SVG пути – код Листинга 95.

Листинг 95: Частицы и векторная графика

```
1
2  class EParticle {
3    float x, y, cx, cy, size, step, dist;
4    float counter;
5    EParticle previousP;
6
7    EParticle(float cx, float cy, float size, float step,
       float dist) {
```

```
8       this.cx = cx;
9       this.cy = cy;
10      this.size = size;
11      this.dist = dist;
12      this.step = step;
13    }
14
15    void render() {
16      if(previousP != null){
17        stroke(200,50);
18        line(previousP.x, previousP.y, x,y);
19      }
20      ellipse(x,y,size,size);
21    }
22  }
23
24  class ParticleController {
25    ArrayList<EParticle> ar = new ArrayList<EParticle>();
26
27    void createParticles(float size, PShape border) {
28      EParticle previousObj = null;
29      for(int i = 0 ; i < border.getVertexCount(); i++){
30        float vx = border.getVertexX(i);//122 120
31        float vy = border.getVertexY(i) - 550; //343 497.54
32        EParticle lObj = new EParticle(vx, vy, size, random
              (-0.05, 0.05), random(20, 500));
33        if(i != 0){
34          lObj.previousP = previousObj;
35        }
36        ar.add(lObj);
37        previousObj = lObj;
38      }
39    }
40
41    void upDate(float dimmer) {
42      for (EParticle tmp : ar) {
43        tmp.counter += tmp.step;
44        tmp.x = tmp.cx + sin(tmp.counter)*(tmp.dist*dimmer);
45        tmp.y = tmp.cy - cos(tmp.counter)*(tmp.dist*dimmer);
46      }
47    }
48
49    void render() {
50      noStroke();
51      fill(200,200);
52      for (EParticle tmp : ar) {
53        tmp.render();
54      }
55    }
```

```
56  }
57
58  ParticleController pc = new ParticleController();
59
60  void setup() {
61    size(500, 500);
62    smooth();
63    background(50);
64    PShape drawingSVG = loadShape("a.svg");
65    PShape border = drawingSVG.getChild("path14003");
66    pc.createParticles(5, border);
67    ellipseMode(CENTER);
68  }
69
70  void draw() {
71    background(0);
72    float var = 0;
73    if(mouseX < width/2){
74      var = map(mouseX, 0, width/2, 1, 0);
75    } else {
76      var = map(mouseX, width, width/2, 1, 0);
77    }
78    pc.upDate(var);
79    pc.render();
80  }
```

В коде Листинга 95 мы немного поменяли класс частицы *EParticle*, добавив в него еще одно свойство – ссылку на объект класса *EParticle* (строка 5). С этой ссылкой мы будем связывать предыдущий объект на SVG пути. Имея эту ссылку, мы можем отрисовать линию между двумя соседними частицами.

В методе *render()* в 16-й строке мы проверяем ссылку на предыдущий объект: если она не пустая, тогда рисуем линию. Отрисовка происходит в 17-й и 18-й строках, после чего отрисовывается эллипс размера *size*.

Изменения в коде Листинга 95 небольшие, но они достаточно расширяют объем художественных возможностей. Кстати, это демонстрация того, как вы сами можете расширять классы новыми свойствами и использовать их. На Рисунке 119 показан результат выполнения кода Листинга 95.

Перейдем к рассмотрению логики выполнения программы. В строке 58 мы объявляем, как обычно, контроллер частиц – свойство *pc*, создаем объект класса *ParticleController* и связываем его со ссылкой pc. В 64-й строке мы создаем объект *PShape* с помощью вызова метода *loadShape()*, связываем его со ссылкой *drawingSVG*. Эти операции были рассмотрены нами на примерах других SVG файлов – логика тут не поменялась.

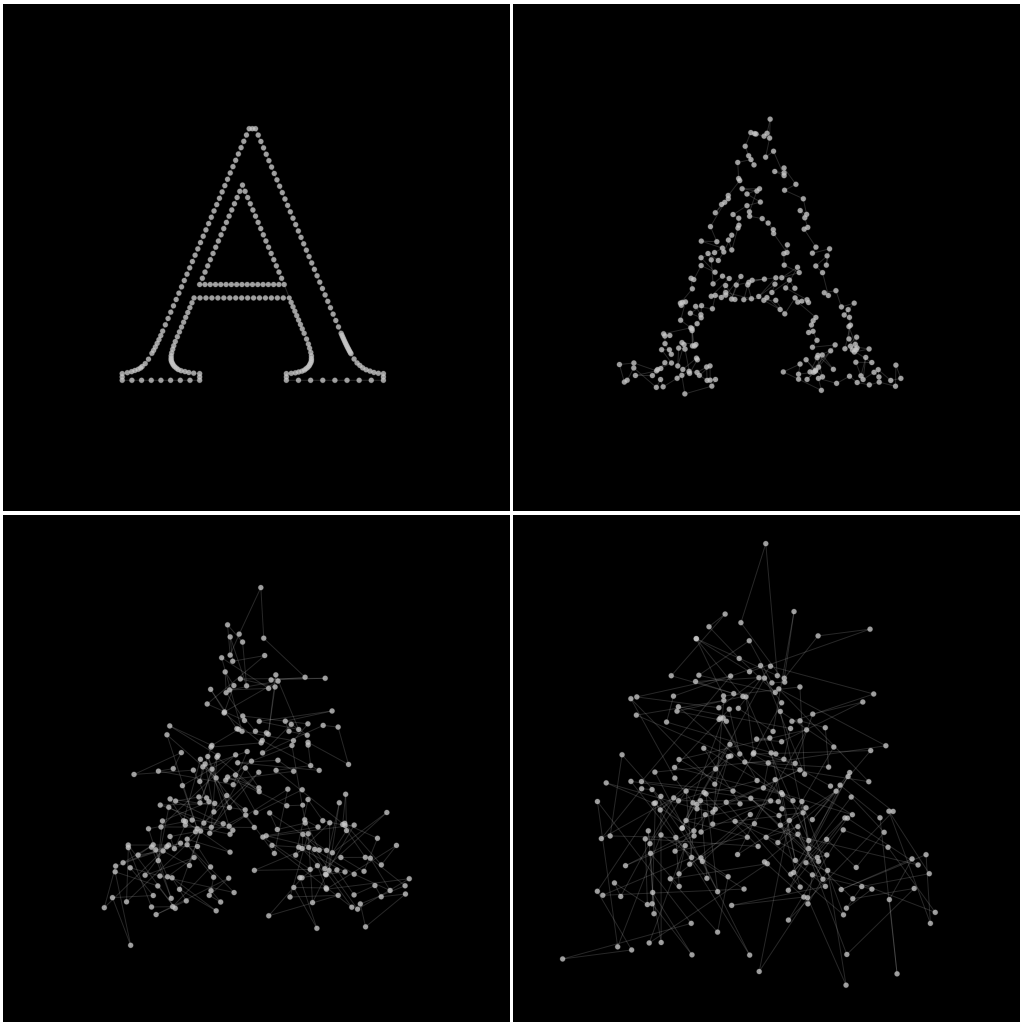


Рис. 119: Результат выполнения Листинга 95.
Фазы рисования контура векторной буквы «A» частицами.

Наш SVG файл содержит путь, который мы получили с помощью векторизации буквы «A». Мы добавили некоторое количество вершин к получившемуся пути в редакторе Inkscape. XML элемент SVG пути имеет атрибут *id*, значение которого – *«path14003»*, так что в 65-й строке у нас нет никаких препятствий, чтобы создать объект нашего пути и связать его со ссылкой *border*. Эту ссылку мы передаем аргументом в метод *createParticle()*, который мы объявили в классе *ParticleController*. Логика работы метода *draw()* не поменялась по сравнению с предыдущим примером.

Метод *createParticle()* класса *ParticleController* по сути не отличает-

ся от предыдущего примера. Только в этот раз нам предстоит обходить не массив пикселей, а массив вершин SVG пути, что мы и делаем, начиная со строки 29. Однако нам требуется еще подготовиться к записи предыдущей частицы, для чего в 28-й строке мы определили ссылку *previousObj*: с ней мы свяжем частицу, которая будет требоваться для последующей (строка 37). Если текущая частица является первой, то у нее нет предыдущей – эту ситуацию мы проверяем в строке 33. Методы *upDate()* и *render()* работают обычным способом.

Итак, мы показали возможность использования «памяти» частицы не только на координаты центрального положения, но и на предыдущий объект, что позволило нам получить требуемый результат.

Давайте рассмотрим еще один пример на взаимодействие частиц. Надо отметить, что добавление функционала к частице влечет за собой увеличение операций, что обязательно скажется на производительности, но наш пример кода в Листинге 96 работает быстро – рассмотрим его.

Листинг 96: Множественное взаимодействие частиц

```
1
2  class RectParticle {
3    float x, y, cx, cy, size, step, dist;
4    float counter;
5
6    RectParticle(float cx, float cy, float size, float step,
       float dist) {
7      this.cx = cx;
8      this.cy = cy;
9      this.size = size;
10     this.dist = dist;
11     this.step = step;
12   }
13
14   void render() {
15     ellipse(x,y,size,size);
16   }
17 }
18
19 class ParticleController {
20   ArrayList<RectParticle> ar = new ArrayList<RectParticle
       >();
21
22   void createParticles(float size, PShape border) {
23     for(int i = 0 ; i < border.getVertexCount(); i++){
24       float vx = border.getVertexX(i);
25       float vy = border.getVertexY(i) - 550;
26       RectParticle lObj = new RectParticle(vx, vy, size,
```

```
            random(-0.01, 0.01), random(20, 500));
27        ar.add(lObj);
28      }
29    }
30
31    void upDate(float dimmer) {
32      counter = dimmer;
33      for (RectParticle tmp : ar) {
34        tmp.counter += tmp.step;
35        tmp.x = tmp.cx + sin(tmp.counter)*(tmp.dist*dimmer);
36        tmp.y = tmp.cy - cos(tmp.counter)*(tmp.dist*dimmer);
37      }
38    }
39
40    void render() {
41      noStroke();
42      fill(200,200);
43      for (RectParticle tmp : ar) {
44        tmp.render();
45        for (RectParticle tmp1 : ar) {
46          stroke(200,50);
47          if(dist(tmp1.x, tmp1.y, tmp.x,tmp.y) < 40){
48            line(tmp1.x, tmp1.y, tmp.x,tmp.y);
49          }
50        }
51      }
52    }
53  }
54
55  ParticleController pc = new ParticleController();
56
57  void setup() {
58    size(500, 500);
59    smooth();
60    background(50);
61    PShape drawingSVG = loadShape("a1.svg");
62    PShape border = drawingSVG.getChild("path14003");
63    pc.createParticles(5, border);
64    ellipseMode(CENTER);
65  }
66
67  void draw() {
68    background(0);
69    float var = 0;
70    if(mouseX < width/2){
71      var = map(mouseX, 0, width/2, 1, 0);
72    } else {
73      var = map(mouseX, width, width/2, 1, 0);
74    }
```

```
75      pc.upDate(var);
76      pc.render();
77  }
```

На Рисунке 119 показан результат выполнения кода Листинга 95.

Мы упростили класс для частиц – *RectParticle*, оставив в нем только свойства – параметры, конструктор и метод *render()*. Логика работы приложения не поменялась: методы *setup()* и *draw()* остались такими же, как в предыдущем примере. Поменялся класс *ParticleController*.

В классе *ParticleController* упростился метод *createParticle()*: теперь нам не требуется запоминать предыдущую частицу. Метод *upDate()* не изменился.

В методе *render()* класса *ParticleController* мы начинаем обходить хранилище частиц в 43-й строке. Предположим, что мы прошли несколько итераций цикла и сейчас у нас одна из частиц активна. Мы хотим проверить все расстояния между этой активной частицей и всеми остальными. Для этого нам снова надо обойти все хранилище и проверить расстояния, поэтому мы снова пишем цикл *for each* в 45-й строке. В 47-й строке мы вычисляем расстояние с помощью метода *dist()*, и, если оно меньше 40 пикселей, попадаем в 48-ю строку, в которой отрисовываем линию между двумя близкими частицами.

Глядя на работу приложения, в котором частицы как будто бы знают, что приблизились друг к другу, можно представить, что они протягивают линию «дружбы» друг другу.

Мы рассмотрели возможности работы с частицами, научились определять контроллеры частиц, налаживать взаимодействие частиц. Этот материал, расширяя объем творческих возможностей, может натолкнуть дизайнера на оригинальные решения многих художественных задач. Примеры использование систем частиц можно увидеть в моем собственном проекте Visualtools:

1. Interactive particle system sketch https://vimeo.com/32408372

2. Dynamic particle type https://vimeo.com/34899632

В примерах, приведенных в главе, можно найти параллели с работами Дае Ин Чанг (Dae In Chung) в проекте Paperdove http://paperdove.com/work/generative-selfportraits.html и Робби Тилтана (@RobbieTilton) в проекте Consciousfree http://consciousfree.tumblr.com/post/109231770645/58-365 .

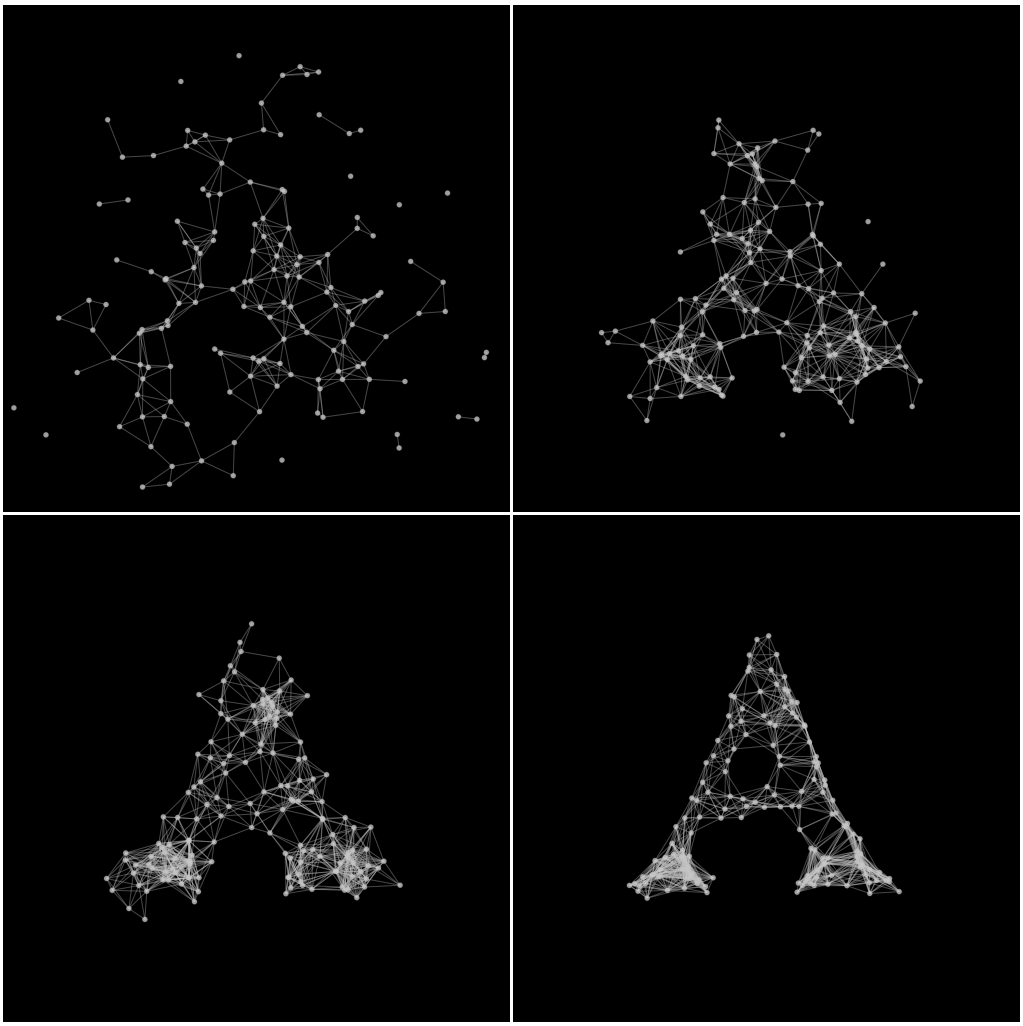
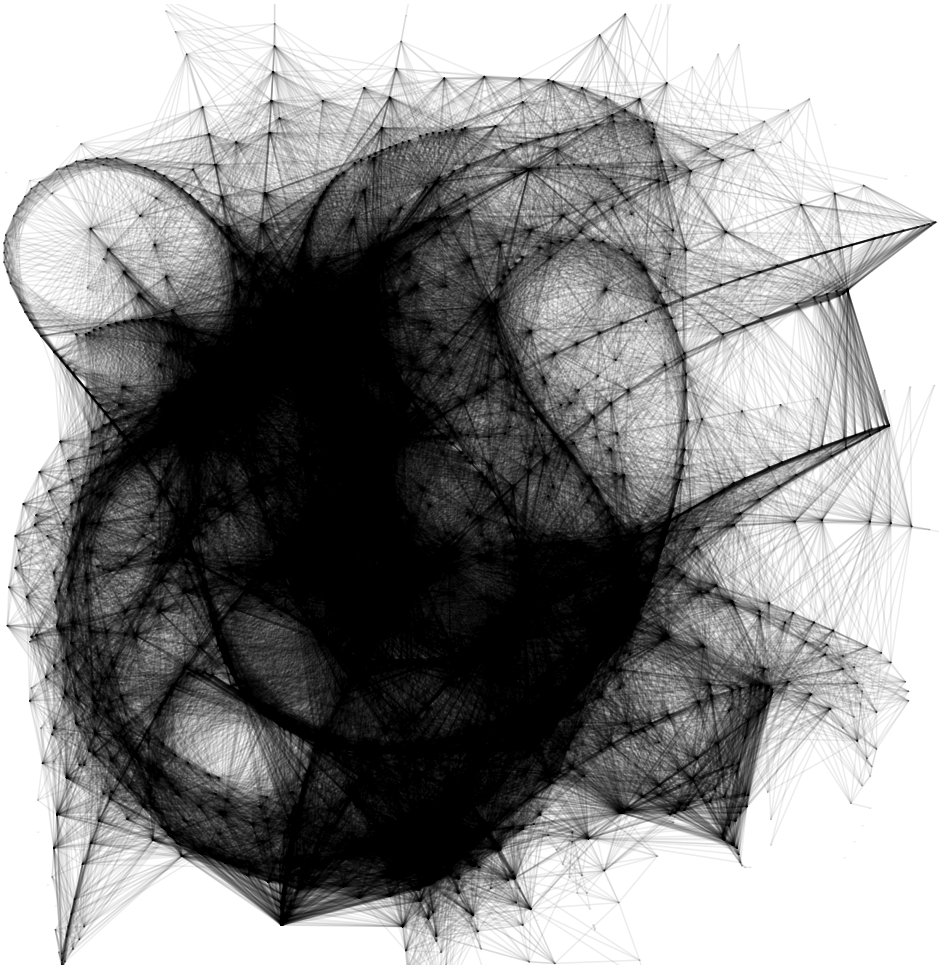


Рис. 120: Результат выполнения Листинга 96.
Фазы рисования и разбивания буквы «A».



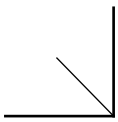
Programming for Atrists

Pavel A. Orlov

14 Заключение

Программирование как инструмент современного цифрового художника, безусловно, требует знаний и умений в обращении с ним. К сожалению, исторически сложившееся мнение о программисте как о «маге и волшебнике», умеющем «говорить» с компьютером, вызывает известные опасения у человека художественного склада ума. В этой книге предпринята попытка развенчать необоснованные страхи и сомнения: нет ничего сверхсложного в том, чтобы научиться использовать программирование в качестве универсального и мощного инструмента творчества.

Надеюсь, что эта книга будет служить началом вашего увлекательного и захватывающего путешествия в мир красоты и гармонии, искусства и программирования.



АВАТАР