# Технологическая реализация графических интерфейсов (на примере Canvas-JS)

В этой главе учебного пособия мы обсудим вопросы технологической реализации графических интерфейсов. Если в остальных главах мы говорим больше о том, как графический интерфейс должен выглядеть, какое влияние оказывает на работу пользователя и как формирует пользовательский опыт, то в рамках этой главы мы эти соображения ненадолго отставим в сторону. Здесь нашей задачей будет разобраться в том, как именно это всё реализуется на техническом уровне.

Нам предстоит обсудит следующие аспекты технологической реализации интерфейсов.

1. **Рендеринг** – процесс выведения на экран необходимых пикселей, из которых складываются видимые на экране элементы интерфейса. Рендеринг необходим для любой графической системы, и GUI здесь не исключение.
2. **События** – повсеместно используемый в реализациях GUI архитектурный подход, который позволяет компьютеры определять ключевые моменты в работе программы и запускать в эти моменты какие-то алгоритмы. Все вопросы, связанные с нажатиями клавиш, движения мышки, любыми другими действиями пользователя, а также любыми другими стимулами (такими как, например, таймеры или сетевые сообщения), в современных интерфейсных системах оформляются как события.
3. **Объектная структура** – категоризация элементов интерфейса по типам используемых данных и выполняемого функционала. По мере усложнения интерфейсов и появления в нём всё большего количества и разнообразия элементов, возникает необходимость выделять общие категории подобных элементов, группируя их по общему функционалу. Это необходимо как для продолжения разработки, которая без такой категоризации объектов становится неподъёмно сложной, так и для перехода к последнему шагу – разметке.
4. **Разметка –** способ описания интерфейса на языке, приближенном к человеческому и доступному не только разработчику интерфейса, но и дизайнеру. Необходимый замыкающий шаг в истории технологической разработки интерфейса, после которого могут начинать работать дизайнеры.

Мы обсудим эти четыре аспекта разработки интерфейсов на примере создания интерфейса для интерактивного Web-приложения. То есть в процессе работы в этой главе нам так же предстоит познакомится с языком программирования JavaScript и языком разметки HTML.

Результатом нашей работы будет небольшая браузерная игра, которая будет целиком существовать в рамках HTML-элемента **Canvas**. Игра будет состоять из кнопок, менюшек и картинок, и будет в этом смысле целиком воспроизводить стандартный интерфейс с горячими клавишами, реакциями на движения мышки итд. Мы так же предусмотрим возможность для гипотетического будущего дизайнера этой игры описывать и настраивать её работу путём редактирования специального текстового файла.

В процесса этой работы мы своими руками применим все основные подходы к технологической реализации всех аспектов GUI. Мы так же на каждом этапе будет специально уделять время, чтобы обсудить какие из реализованных нами технологических решений является универсальными для всех современных GUI-систем, а какие являются особенностями конкретно выбранной нами технологической связки **HTML-Canvas-JS**.

Последний момент: тестировать работу наших программ и ловить ошибки в работе JS мы будем в браузере **Chrome**. Писать код **HTML/JS** можно в любом текстовом редакторе, но на скриншотах внутри этого пособия Вы будете видеть редактор **PyCharm Community Edition**, который мы и рекомендуем для работы всем.

## Рендеринг.

В первой крупной главе нашего пособия мы разберёмся с вопросами выведения пикселей на экран в нужных нам положениях. Мы так же обучимся основам разработки программ на языке **JavaScript** и тестировании этих программ в браузере **Chrome**.

### Первый HTML файл

Для того, чтобы открыть страницу в браузере, нужно сначала получить страницу! В нашем случае, страницей будет служить лежащий в любом месте на нашем жёстком диске HTML-файл. В этом HTML-файле должен быть стандартный набор инструкций, описывающих метаданные нашей Web-страницы. В нашем случае всё, что нам нужно, это чтобы в этой странице был тэг **сanvas** и чтобы эта веб-страница загружала программу на **JavaScript**, которая могла бы с этим тэгом работать.

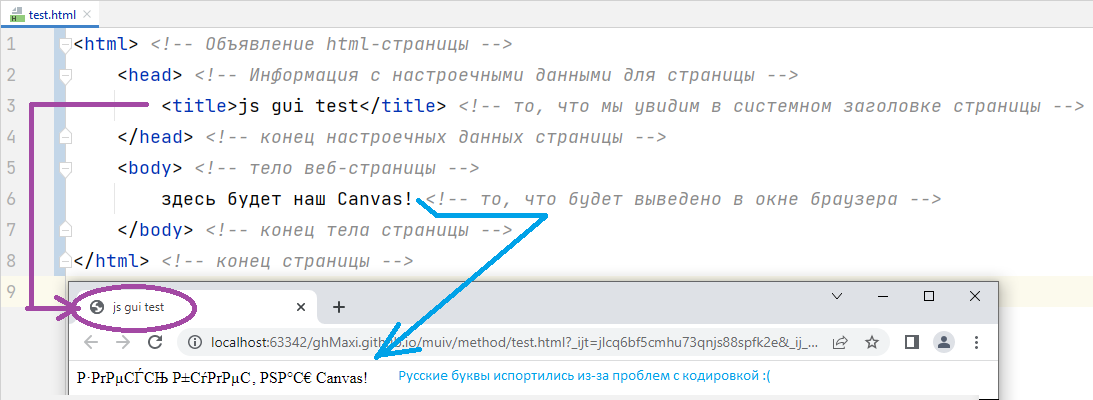
Перед тем, как мы перейдём к разбору нашего первого HTML файла, пара слов о том, как устроен язык HTML.

* Язык HTML состоит из **тэгов**, которые пишутся в **треугольных скобках**. При этом подавляющее большинство тэгов должны сначала открываться, а потом закрываться и всё, что находится между открытием и закрытием считается внутренностью этого тэга.  
  Например, вся наша исходная страница будет находится внутри тэга **<html>** и заканчиватся с закрытием тэга **</html>**.
* Внутри тэга могут находится другие тэги. Например, в нашей исходной странице будут тэги **<head>** и **<body>**, непосредственно подчинённые тэгу **<html>**. А внутри тэга **<head>** будет подчинный уже ему тэг **<title>**
* Внутри тэга может находиться текст. И тогда расположение текста на странице будет определяться тем тэгом, в котором он находится. Напримерь, текст **<title>** пойдёт в системный заголовок, а текст **<body>** пойдёт в окно браузера.

На скриншоте внизу – первый предлагаемый вариант HTML файла, а так же то, как этот файл выглядит в браузере (можете просто открыть его двойным кликом, или можете открыть его средствами PyCharm через localhost, как на скриншоте). Сразу отметим несколько проблем и особенностей этого файла:

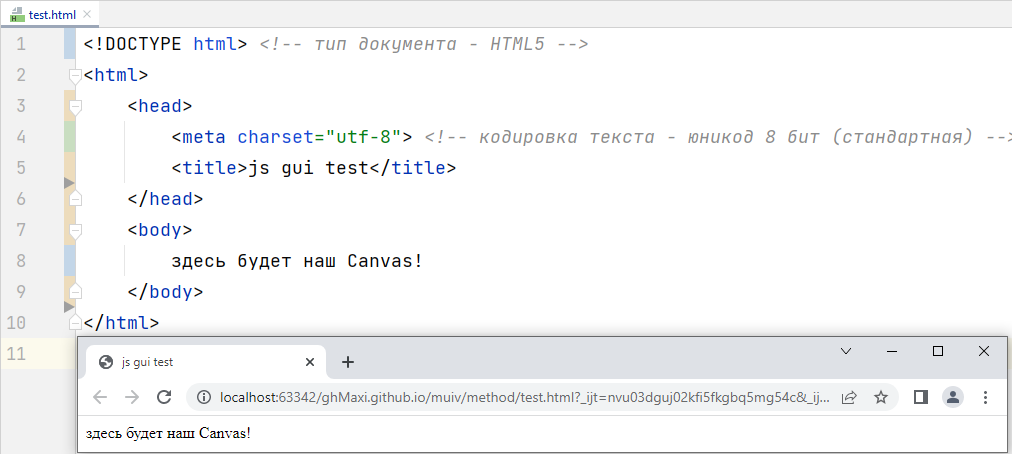
* В файле пока нет нужного нам **сanvas**
* В файле не подключен **JavaScript**
* В зависимости от того, как именно Вы откроете этот файл, могут возникнуть проблемы с отображением русских букв (как на скриншоте)
* Файл снабжён избыточными комментариями, описывающими работу каждой строки. Комментарии в HTML определяются так **<!-- комментарий -->** (на скриншоте серого цвета). Они даны больше для Вашего понимания работы и к себе их переписывать не нужно. С будущих скриншотов комментарии старых строк будут убираться.

С учётом всего выше сказанного, разберите и воспроизведите у себя первую версию файла test.html, данную на скриншоте непосредственно ниже. Дальше мы будем этот файл улучшать!

Рис.1: файл test.html, версия 1  


Добавим в наш файл справочные тэги **<!DOCTYPE>** и **<meta>** с информацией о том, с каким документом файл имеет дело и в какой кодировке должен отображаться. Эти справочные тэги не требуют закрытия, как остальные, потому что не могут иметь в себе содержимое и всю полезную информацию несут в рамках настроек в треугольных скобках.

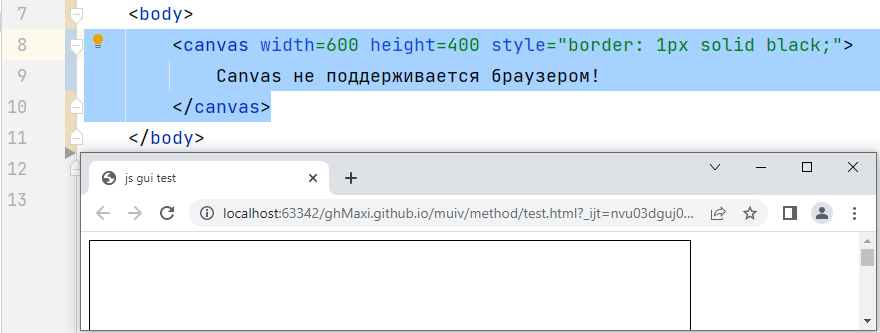
Так как на скриншоте мы работает в **pyCharm**, который по умолчанию сохраняет файлы в кодировке **utf-8**, то именно эту информацию мы и укажем в настройках тэга **<meta>**. Эта кодировка обычно настроена во всех редакторах по умолчанию, и её выставление решает проблему с отображением букв. Но, теоретически, Ваш редактор может быть настроен на сохранение в другой кодировке, и тогда мы рекомендуем найти способ переключить его на **utf-8**.

Рис.2: файл test.html, версия 2 с DOCTYPE и метаданными  


Наконец, последний шаг – добавить на нашу страницу элемент Canvas, в котором мы будем рисовать наш интерфейс!

Элемент **<canvas>** должен отображаться в теле страницы, поэтому добавляется внутрь тэга **<body>**. В его настройках мы пропишем его ширину и высоту в пикселях, а так же зададим в рамках настроек стиля отображения границу шириной в 1 пиксель. Тэг **<canvas>** предусматривает возможность наличия содержимого (и поэтому должен закрываться). Внутри тэга **<canvas>** пишут то, что выводится в браузерах, не поддерживающих этот тэг. Напишем там соответствующий текст.

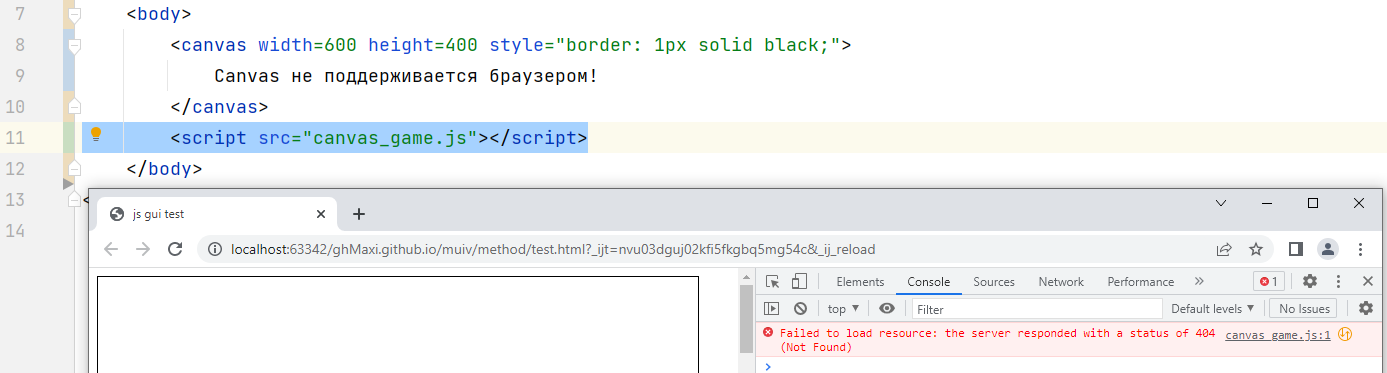
В результате в браузере не должно быть видно текста, но должна быть видна чёрная рамочка заданого размера! (Код и пример на скриншоте ниже)

Рис.3: файл test.html, версия 3 – изменения внутри тэга **<body>**  


Чтобы завершить подготовку нашего html-файла, нам осталось только сделать так, чтобы он загружал JavaScript из внешнего файла. Для этого надо добавить одну-единственную строчкусразу после **</canvas>:** загрузку скрипта с названием **canvas\_game.js**. Для этого используем специальный тэг **<script>**.

Кое-какие особенности использования тэга:

* Тэг **<script>** надо закрывать, потому что в нём теоретически может быть содержимое.
* При загрузке скрипта из внешнего файла через настройку **src** содержимое тэга **<script>** просто игнорируется. Из-за этого возникает небольшой парадокс – тэг без содержимого, который всё равно надо закрывать. Так как **HTML/JS** – достаточно старая технология, подобного рода парадоксов в ней накопилось много и с ними просто надо смириться.
* Так как пока самого файла **canvas\_game.js** у нас нет, то понятно, что ничего не загрузится. Но мы всё равно можем увидеть ошибку загрузки в консоли разработчика, которая в **Chrome** открывается через комбинацию клавиш **CTRL+SHIFT+J**.

Рис.4: файл test.html, версия 4 – добавление **<script>** после **</canvas>** и ошибка в консоли

**<!!! Другие способы открытия консоли /!!!>**

На этом предварительная работа с HTML-файлом завершена! Мы разобрали с Вами структуру этого файла, сделали в нём базовые настройки и создали элемент **canvas**, который и будет служить нам холстом для рисования в дальнейшем. Мы так же подготовили наш файл к работе со скриптом, который мы начнём писать в следующем разделе.

**Упражнения:**

1. Задайте canvas размер 200x300 (то есть ширина меньше высоты).
2. Увеличьте толщину границы **canvas** до 5px;
3. Поменяйте системный заголовок окна с **js gui test** на **canvas game.**

**Задачи для самостоятельной работы:**

1. Найдите в Интернет документацию по тэгу **<h1>** и с его помощью сделайте на странице заголовок. Расположите заголовок над **canvas**.
2. Сделайте так, чтобы граница у **canvas** была пунктирная синего цвета (для этого надо будет модифицировать его **style**, используя **css**-свойство **border**)
3. Вставьте внутри **<body>** перед **<canvas>** следующий скрипт:  
     
   <script>  
   document.write('ХАХАХАХАХАХХА ЗДЕСЬ БЫЛ <span style="color:red;">ДЖАСЬ!</span>');  
   </script>  
     
   Объясните результат.

**Общее и различное с другими архитектурами GUI.**

**Общее** состоит в том, что все архитектуры должны так или иначе получать информацию о том окне, в котором рисуется картинка и как-то определять размеры этого окна в пикселях. То есть во всех таких архитектурах возникает свой аналог **canvas**.

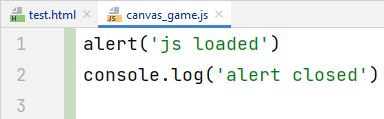
**Различие** состоит в том, что в большинстве других gui-решений этот самый gui автоматически занимает всё окно, в то время как наш canvas занимает только строго отведённое ему место в браузере. Это является следствием более фундаментального отличия, которое состоит в том, что там, где другие архитектуры могут напрямую общаться с операционной системой и видеокартой, мы вынуждены работать опосредованно через браузер.

### Первый JS файл. Рисуем пиксели на Canvas.

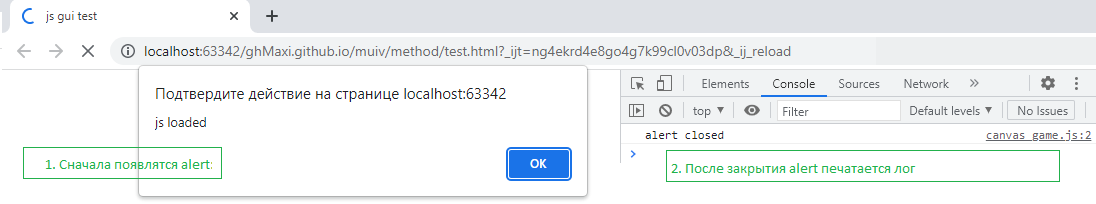
В прошлом разделе мы создали HTML файл, в котором есть элемент **<canvas>** и в которой загружается скрипт из файла **canvas\_game.js**.

Сейчас нам предстоит создать этот скрипт!

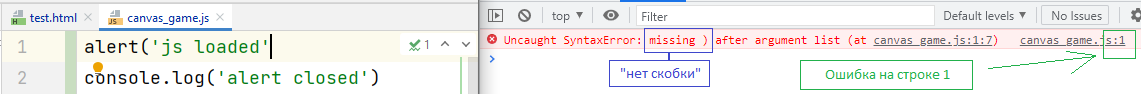
Создайте в одной папке с html-файлом файл **canvas\_game.js** и напишите в нём следующие строки:

Рис.5: файл canvas\_game.js, версия 1 – два способа выводить сообщения  


Первая строчка выводит окно с данным в скобках текстовым сообщением и останавливает работу скрипта, пока это окно активно. Вторая строчка печатает лог в уже знакомую нам по концу прошлой части консоль. Увидеть работу скрипта можно так же, как и раньше, запустив html-файл в браузере. Обратите внимание, что лог не напечатается пока не будет отработан **alert**.

Рис.6: работа скрипта при запуске test.html  


Если в скрипте будут допущены ошибки, то отчёт об ошибках будет в той же консоли. Например, если потерять закрывающую скобку на первой строке, то отчёт об ошибке будет такой:

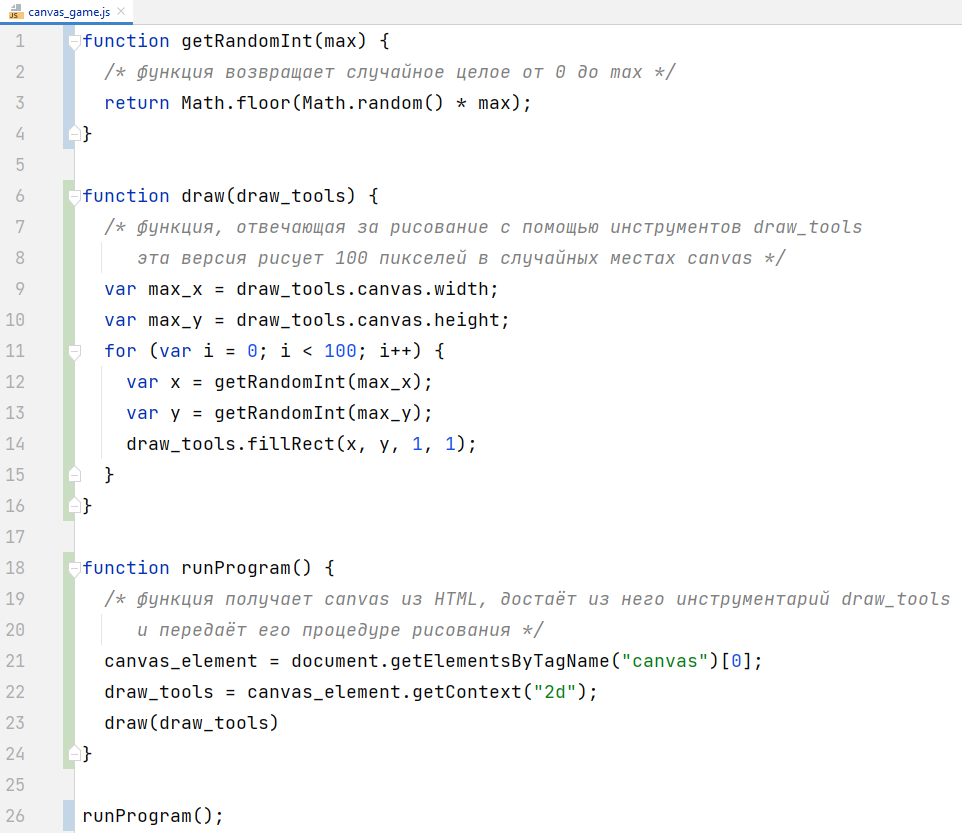
Рис.7: пример отчёта об ошибке в скрипте  


Логи можно использовать для распечаток при нормальной работе программы. Пока Вы не уверены до конца в том, что скрипт нормально загружается при запуске страницы, полезно оставить в начале скрипта лог о запуске скрипта. Задача **alert** – остановить программу при серьёзной ошибке, чтобы можно было эту ошибку обработать. Поэтому **alert** пользуются реже (не говоря уж о том, что в итоговой программе их оставлять нельзя – подобные всплывающие окошки считаются крайне плохим тоном с точки зрения дизайна интерфейсов).

Таким образом, мы можем с помощью распечаток удостоверится в работоспособности нашего скрипта. Если распечатки почему-то не работают, то надо обязательно разобраться, что именно пошло не так! Если же с распечатками всё в порядке, то перейдём к следующему шагу – нарисуем что-то на нашем Canvas!

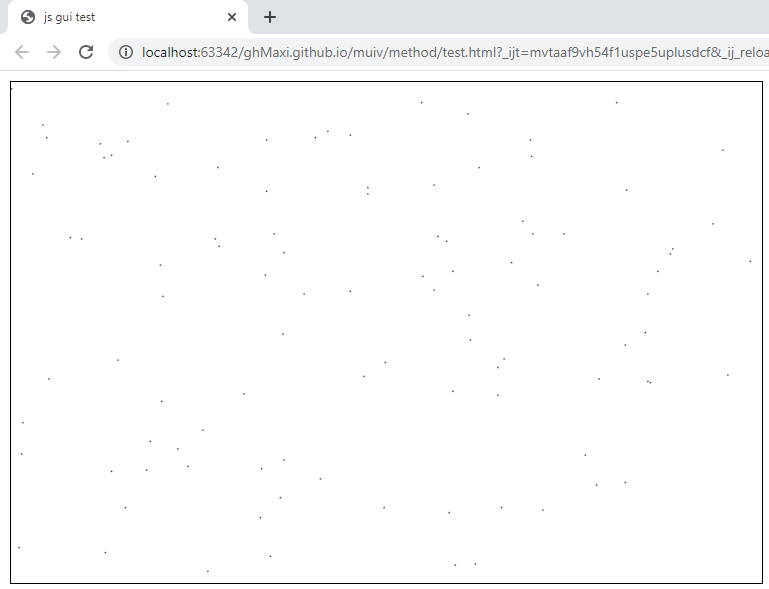
Уберём из нашего скрипта старые распечатки и напишем программу, которая будет рисовать 100 точек в случайных местах на **<canvas>**. На картинке ниже приведён код такой программы. Обратите внимание, что программа значительно выросла в размере – теперь в ней почти 30 строк! Так же обратите внимание, что программа теперь разделена на смысловые блоки, называемые функциями и обозначаемые в JS как **function**.

Рис.8: программа в функциональном стиле, рисующая 100 пикселей



Ожидаемым результатом работы программы является 100 серых точек, расставленных в случайных местах canvas (пример на картинке ниже). Если у Вас это почему-то не так, то найдите и устраните ошибку!

Рис.9: ожидаемый результат работы программы



Давайте посмотрим внимательно на работу этой программы.

Во-первых, программа написана таким образом, что весь её код находится внутри функций. Сама же программа всего лишь запускает стартовую функцию **runProgram** на последней строчке кода.

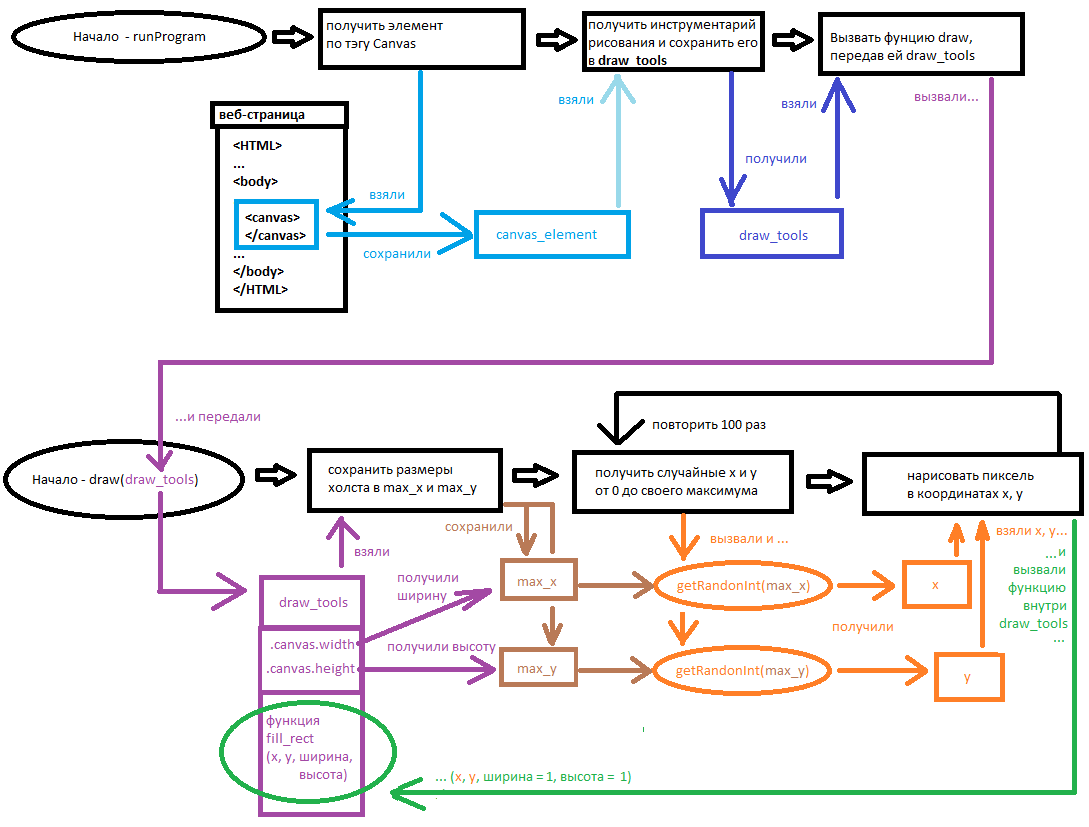


Такое разделение позволяет лучше управлять работой большой программы.

Будучи разделённой на функции **runProgram**, **draw** и **getRandomInt**, работа программы может быть представлена визуально на приведённой ниже блок-схеме. Обратите внимание на следующие моменты:

* **canvas** достаётся непосредственно изсодержимого web-страницы;
* **draw\_tools** создаётся на основе **canvas** и даёт доступ ко всем его параметрам, включая ширину и высоту, через вызов **draw\_tools.canvas**;
* название **draw\_tools** это просто выбранное нами в коде название переменной. В Cети Вы чаще увидите объекты такого типа с названием **ctx** (от английского context);
* **draw\_tools** передаётся как параметр другой функции, так же как **max\_x,** **max\_y**, **x** и **у**
* все параметры **canvas** доступны через **draw\_tools.canvas**;
* **getRandomInt** вызывается дважды с разными значениями максимумов;
* непосредственно рисующая функция **fill\_rect** так же есть часть **draw\_tools**;
* функция **fill\_rect** базово приспособлена под рисование прямоугольников, но мы заставляем её рисовать пиксели, требуя прямоугольники с шириной и высотой 1

Рис.10: блок-схема программы (чёрный цвет – что написано в коде, цветное – что хранится ипередаётся в памяти при выполнении)



Поздравляю, мы создали первый рисующий что-то на экране скрипт на JS! Мы так же написали программу таким образом, что дальше её будет удобно усложнять и расширять, просто добавляя новые функции.

**Упражнения:**

1. Сделайте 1000 точек вместо 100.
2. Сделайте так, чтобы программа рисовала 10 квадратиков размера 20х20, а не точки.
3. Сделайте так, чтобы программа нарисовала 4 больших прямоугольника разных размеров в разных местах **canvas**.

**Задачи для самостоятельной работы:**

1. Найдите в Интернет документацию по CanvasRenderingContext2D (это и есть наш draw\_tools) и разберитесь, как изменять цвет рисования.
2. Нарисуйте 100 квадратиков случайного цвета и размера.
3. Замените код функции **draw** на следующий:  
     
   function draw(draw\_tools) {  
    var max\_x = draw\_tools.canvas.width;  
    var max\_y = draw\_tools.canvas.height;  
    for (var x = 0; x < 10000; x += 0.01) {  
    var x2 = x - max\_x / 2;  
    var y = -x2 \* x2 \* 0.005 + max\_y;  
    draw\_tools.fillRect(x, y, 1, 1);  
    }  
   }  
     
   Объясните результат

**Общее и различное с другими архитектурами GUI.**

**Общее** состоит в том, что все архитектуры должны так или иначе упаковывать информацию об экране и функции рисования в какой-то объект контекста, аналогичный нашему draw\_tools. Так же базовые функции рисования обычно идентичны и включают в себя рисование геометрических примитивов (прямоугольников, овалов итд.), а также картинок из файлов (до этого мы дойдём чуть позже).

То есть разобранная нами организация объектов для «сырого» рисования на экране пикселей, на самом деле, достаточна стандартная.

**Различие** состоит в том, что часто через такой аналог draw\_tools не предоставляют возможность получать базовую информацию о настройках экрана (такую как, например, ширина и высота), ограничивая сферу «ответственности» такого объекта только функциями рисования. В этом случае информацию о параметрах экрана можно получать другими способами.

### Самодельная кнопка – рисуем текст и отслеживаем мышку.