1. 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.

1.1. 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**.

1.1.1. 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.

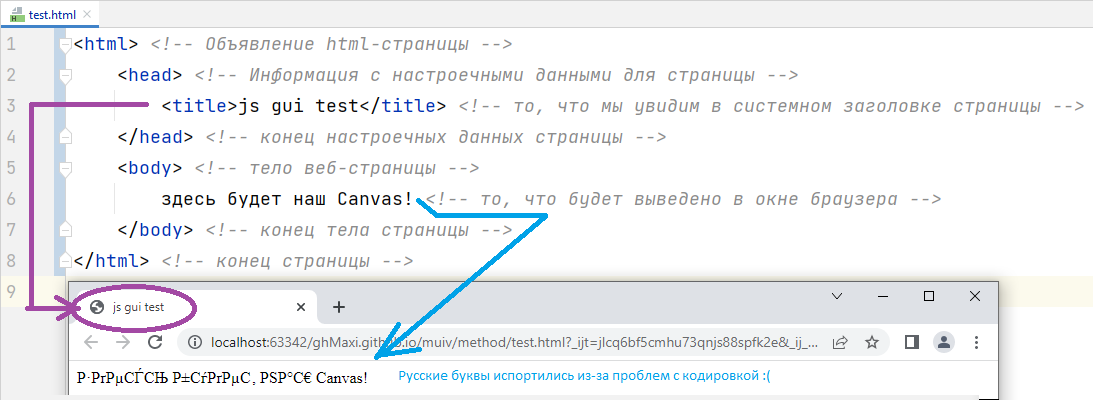
Перед тем, как мы перейдём к разбору нашего первого HTML файла, пара слов о том, как устроен язык HTML.

* 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:

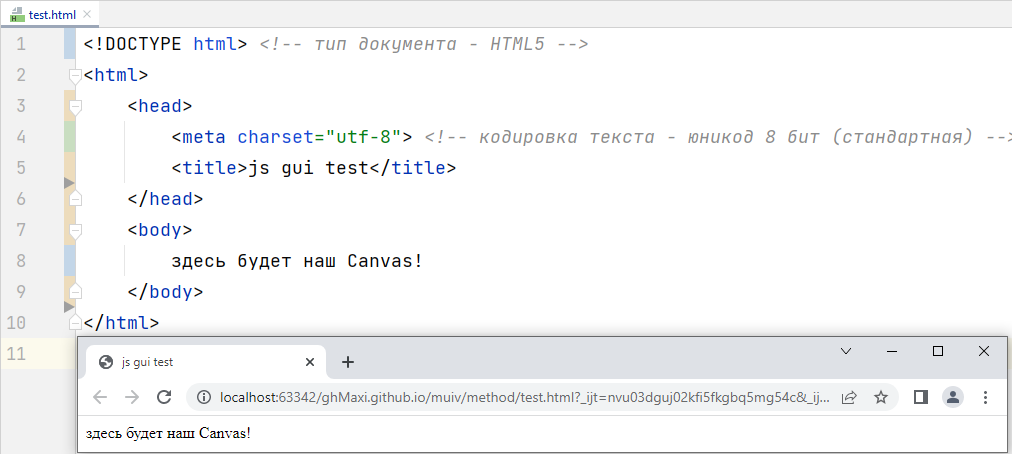
* В файле пока нет нужного нам **сanvas**
* В файле не подключен **JavaScript**
* В зависимости от того, как именно Вы откроете этот файл, могут возникнуть проблемы с отображением русских букв (как на скриншоте)
* Файл снабжён избыточными комментариями, описывающими работу каждой строки. Комментарии 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.

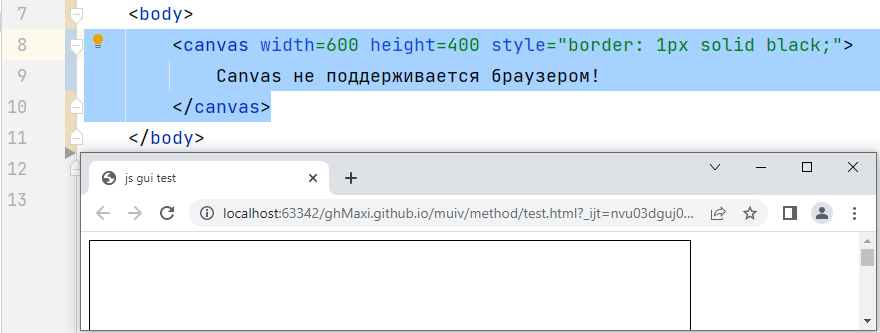
Так как на скриншоте мы работает в **pyCharm**, который по умолчанию сохраняет файлы в кодировке **utf-8**, то именно эту информацию мы и укажем в настройках тэга **<meta>**. Эта кодировка обычно настроена во всех редакторах по умолчанию, и её выставление решает проблему с отображением букв. Но, теоретически, 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.

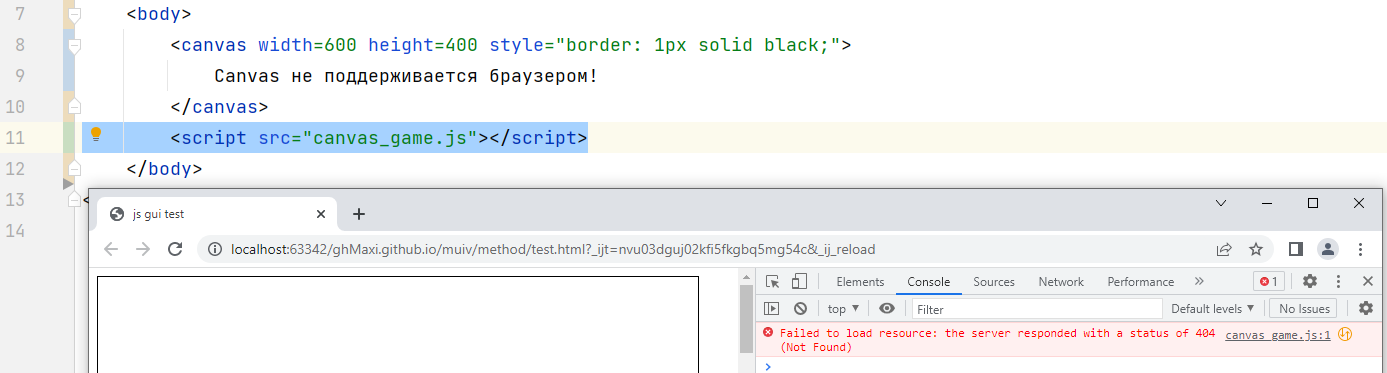
В результате в браузере не должно быть видно текста, но должна быть видна чёрная рамочка заданого размера! (Код и пример на скриншоте ниже)

*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**.

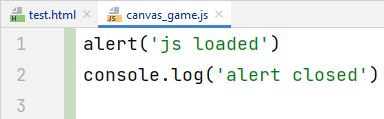
**Различие** состоит в том, что в большинстве других gui-решений этот самый gui автоматически занимает всё окно, в то время как наш canvas занимает только строго отведённое ему место в браузере. Это является следствием более фундаментального отличия, которое состоит в том, что там, где другие архитектуры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

1.1.2. Первый JS файл. Рисуем пиксели на Canvas.

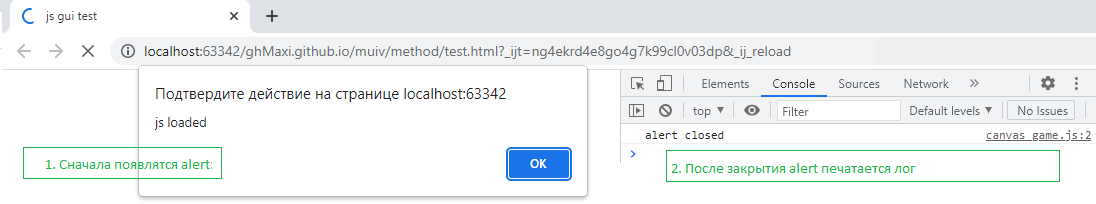
В прошлом разделе мы создали HTML файл, в котором есть элемент **<canvas>** и в которой загружается скрипт из файла **canvas\_game.js**.

Сейчас нам предстоит создать этот скрипт!

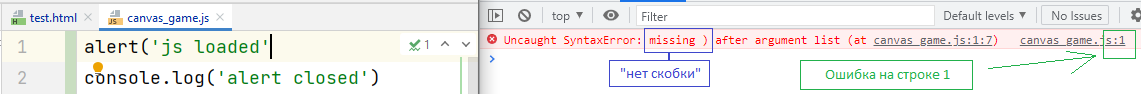
Создайте в одной папке с html-файлом файл **canvas\_game.js** и напишите в нём следующие строки:

*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  
*

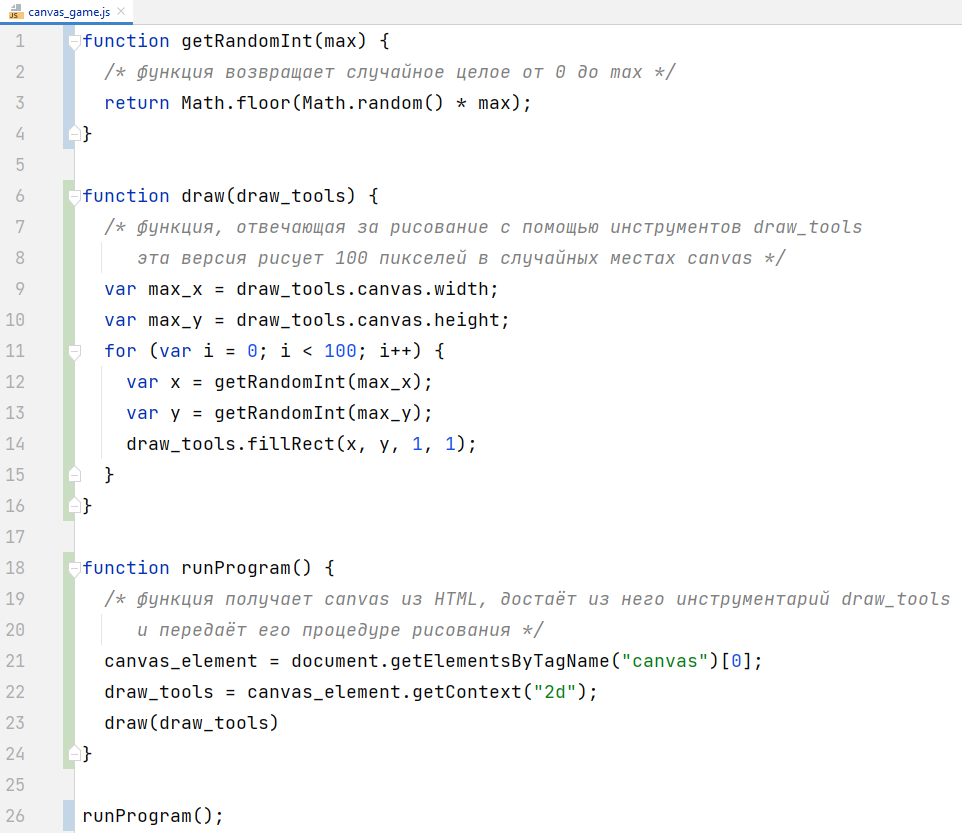
Если в скрипте будут допущены ошибки, то отчёт об ошибках будет в той же консоли. Например, если потерять закрывающую скобку на первой строке, то отчёт об ошибке будет такой:

*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*

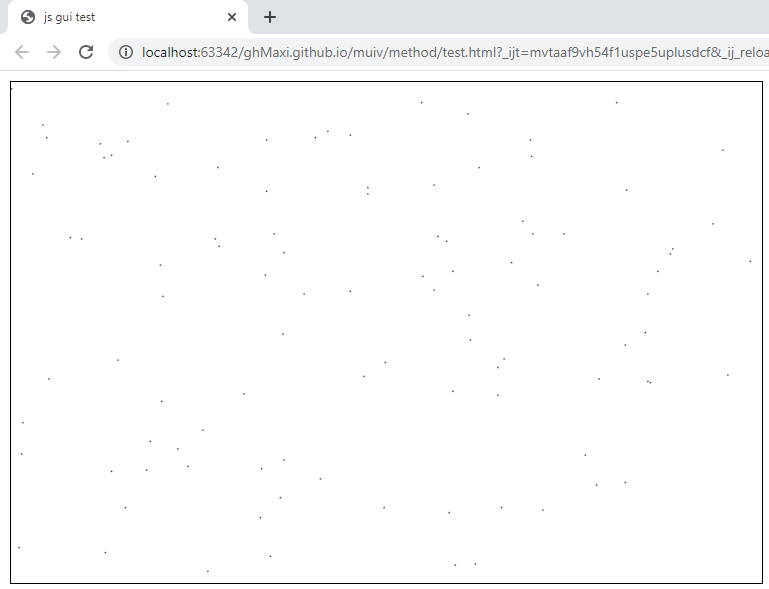
Ожидаемым результатом работы программы является 100 серых точек, расставленных в случайных местах canvas (пример на картинке ниже). Если у Вас это почему-то не так, то найдите и устраните ошибку!

Рис.9: ожидаемый результат работы программы (100 серых пикселей на белом фоне)

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.

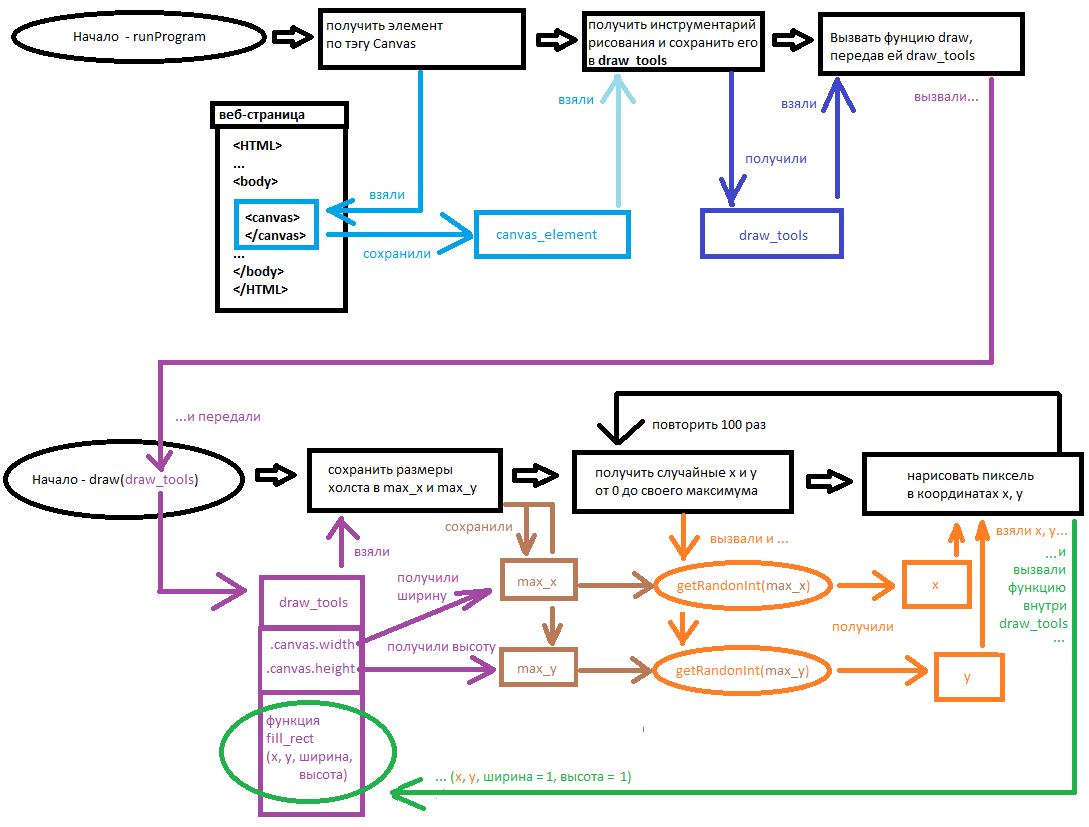
Image

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).

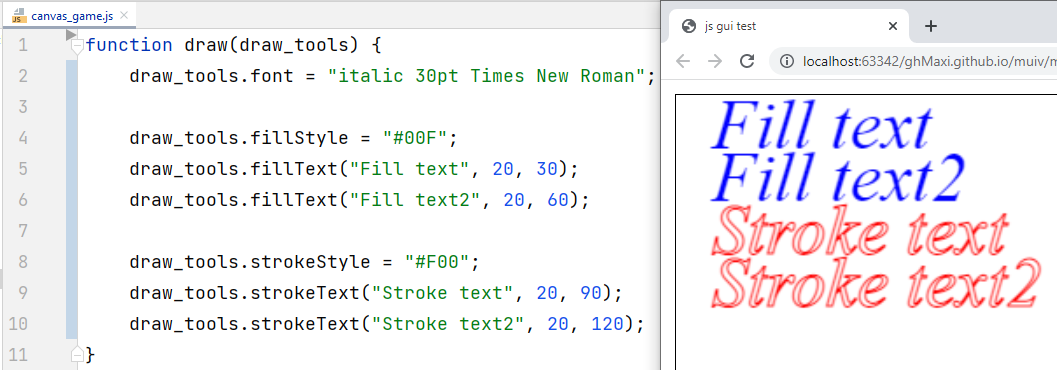
То есть разобранная нами организация объектов для «сырого» рисования на экране пикселей, на самом деле, достаточна стандартная.

**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.

1.1.2 Рисование текстов и кнопок

Научившись рисовать в **canvas** отдельные пиксели, перейдём к рисованию более сложных объектов! Для нашего учебного прототипа нам понадобятся кнопки, тексты, картинки и рамочки (в которые помещать всё остальное).

Разберёмся с каждым таким элементов по очереди. В этой части разберёмся с текстами и кнопками. Для этого уберём старый функционал функции **draw** и напишем новый, отрисовывающие разные тексты.

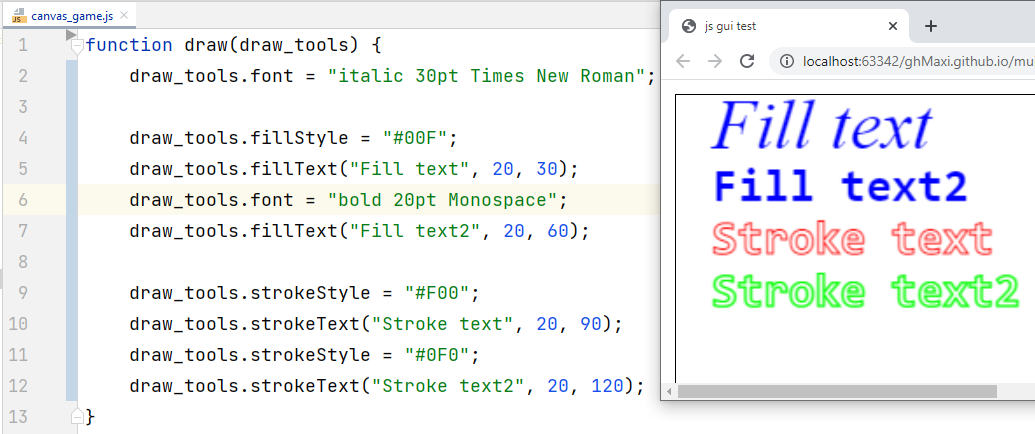
*Рис.11: функция draw, отрисовывающая разнообразные тексты*

Обратите внимание на несколько особенностей этого кода:

* непосредственно отрисовка текста делается двумя способами - **fillText** или **strokeText**. Оба способа - функции внутри **draw\_tools**. Оба способа берут в качестве параметров сам текст и его положение на экране (пиксельную координату основания первой буквы);
* внутри draw\_tools можно назначать параметры **fillStyle**, **strokeStyle** и **font**, позволяющие изменять внешний вид текста. Будучи назначенными, эти параметры влияют на все вызовы **fillText** и **strokeText**;

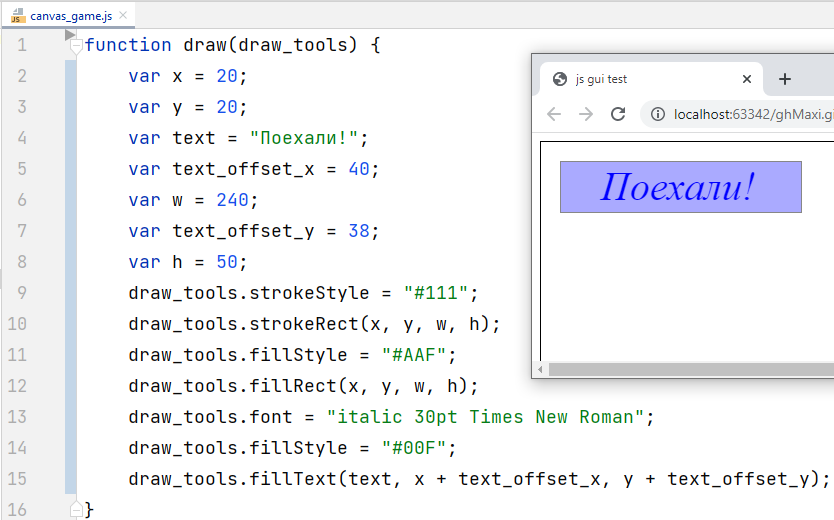
Представьте себе draw\_tools как большую печатную машину. Параметры **fillStyle** и **strokeStyle** - это цвет и вид чернил. Параметр **font** - это конкретный набор оттисков для печатания букв (и если его заменить, то буквы будут печататься другие). Тексты можно печатать (**fillText**) или обводить (**strokeText**), но в любом случае форма букв и цвет будут те, которые назначили.

Но в отличие от реальной печатной машинки, мы можем достаточно легко менять шрифты и цвета. Пример новой функции **draw** с изменёнными цветами и шрифтами ниже.

*Рис.12: функция отрисовки текстав с изменениями шрифта и цвета*

Обратите внимание на новые строчки - 6 и 11, где сначала меняется тип шрифта, а потом меняется цвет обводки.

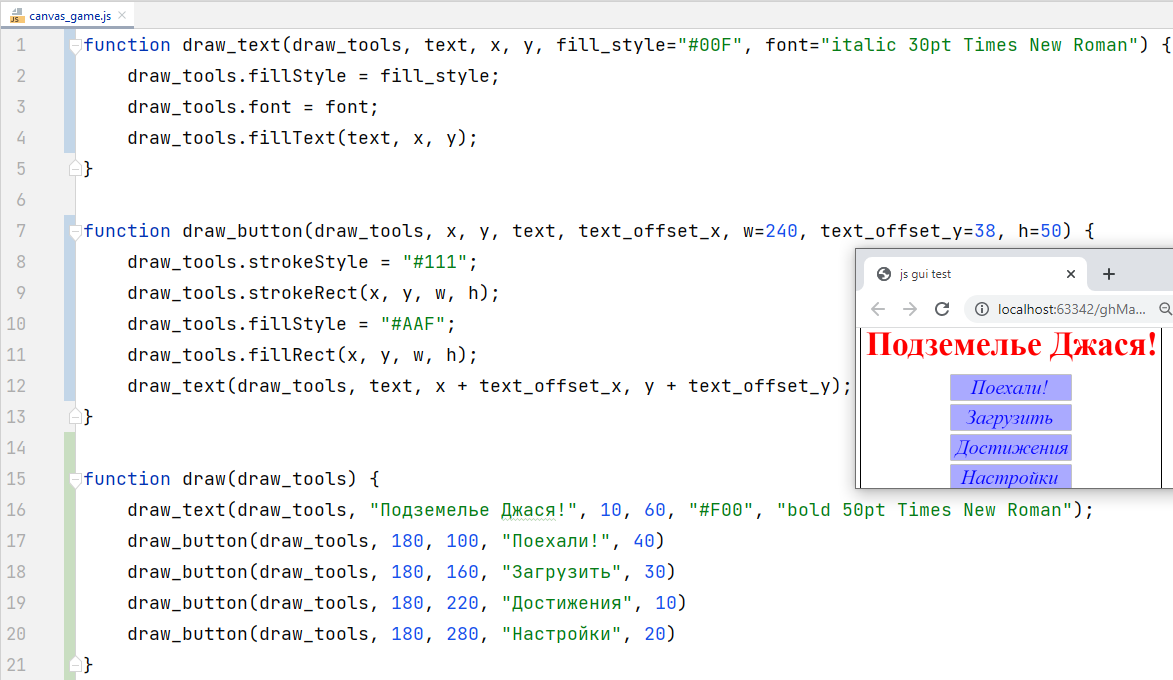
Умея выводить на экран сырой текст, давайте теперь выведем на экран кнопку! Кнопка это просто текст на фоне какой-то графики. Возьмём для начала самый простой вариант - текст на фоне рамочки. Изменим функцию **draw** таким образом, чтобы она выводила на экран одну кнопку.

*Рис.13: функция отрисовки одной кнопки*

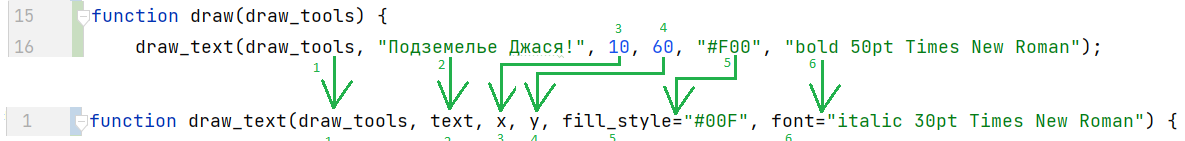
Обратите внимание на следующие моменты:

* сначала рисуется прямоугольник с обводкой, потом текст. Это нужно чтобы текст отрисовался поверх прямоугольника, а не оказался им закрыт. Так же обратите внимание, что цвет заливки прямоугольника и цвет заливки текста берутся из одной и той же переменной **fillStyle**, поэтому ей приходится "на лету" давать другое значение.
* координаты кнопки и её ширина / высота задаются изначально в переменных **x**, **y**, **w** и **h**, после чего используются при всех рассчётах. Это позволяет легко сохранять одинаковый размер обводки и заливки. Это же позволяет позже выровнять текст по высоте и ширине, с помощью дополнительных переменных **text\_offset\_x** и **text\_offset\_y**;
* как не сложно понять, все эти значения подогнаны под конкретно эту кнопку с конкретно этим текстом. Для другой кнопки их придётся подгонять заново. Подробнее об этом ниже;

Таким образом, мы теперь можем вывести на экран одну кнопку. Но даже это уже 15 строчек кода! А если нам нужно, например, 4 кнопи и заголовок? Это же огромная программа! Чтобы не писать почём зря одинаковый код, используют уже знакомый Вам способ разделения программы на функции. Ниже - пример функций отрисовки текста и кнопок, вызы

*Рис.14: отрисовка 4 кнопок и заголовка с помощью служебных функций*

Обратите внимание на следующие моменты:

* у нас две новые функции - **draw\_text** и **draw\_button**, каждая со своей понятной сферой ответственности (одна рисует тексты, другая кнопки);
* у обеих функций есть достаточно много параметров. Обе получают для работы draw\_tools, обе получают **x**, **y** и **text** (хотя тексту важнее **text**, а кнопке важнее **x** и **у**, поэтому они идут раньше);
* настроечные переменные из предыдущего примера стали в этом примере параметрами новых функций. Например, когда функция **draw** вызывает на 16й строке функцию **draw\_text**, она передаёт ей, по порядку: свой **draw\_tools** в параметр **draw\_tools**, текст "Подземелье Джася" в **text**, значения 10 и 60 в **x** и **у**, а так же новое значение **fill\_style** и **font**. Параметры при вызове пишутся и передаются в том же порядке, что и при объявлении функции. Подробнее можно посмотреть на картинке ниже:  
    
  
* некоторые параметры меняются редко, поэтому им присвоены свойства "по умолчанию". Так, если не указывать функции **draw\_text** параметр **fill\_style**, то он будет по умолчанию синим. А если не указывать **draw\_button** ширину **w**, она будет по умолчанию 240;
* значения по умолчанию не обязательно передавать при вызове. Например, когда **draw\_text** вызывают на 12й строчке, ему даются только первые четыре параметра - **draw\_tools**, **text** и перерассчитанные с учётом параметров **text\_offset** значния **x** и **y**. Если же всё равно надо изменить значения по умолчанию (например, как в случае с заголовком нужно измениь шрифт и цвет текста), то их можно передать в обычном порядке. Пример этого - на 16й строчке, где **draw\_text** получает в **fill\_style** красный цвет и жирный большой шрифт в **font**;
* функция **draw\_button** сама пользуется функцией **draw\_text** чтобы нарисовать текст внутри кнопки;
* хотя смысловая плотность кода сильно выросла, его обьём вырос всего на 4 строчки!

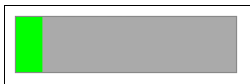
В конечном итоге, обратите внимание, насколько проще стала наша функция **draw**. Нам теперь не нужно подробно прописывать код создания каждой отдельной кнопки, вместо этого мы просто отдаём команды с параметрами, а созданые нами служебные функции по этим параметрам отрабатывают.

**Упражнения:**

Для выполнения упражнений достаточно просто модифицировать функцию **draw** из последнего примера, вызывая необходимые функции отрисовки.

1. Сделайте макет интерфейса простейшего калькулятора - 9 кнопок с цифрами, 4 кнопки с действиями, кнопка "равно" и пустой белый квадратик под вывод цифр;
2. Сделайте интерфейс новостной ленты. Пусть на экране будет 4 новости с заголовками, краткими аннотациями и кнопками "читать" и "поделиться";

**Задачи для самостоятельной работы:**

1. Напишите функцию **draw\_progress\_bar(draw\_tools, x, y, w, h, percent)**, которая будет выводить на экран полоску выполнения, заполненную на заданное число процетов. Например, вызов **draw\_progress\_bar(draw\_tools, 10, 10, 200, 50, 12);** должен давать полоску, заполненную примерно на 1/8:  
   
2. Напишите функцию **draw\_storage(draw\_tools, x, y, w, h, max, current)**, которая будет, вызывая **draw\_progress\_bar**, рисовать полоску заполнения какого-то абстрактного хранилища, указывая при этом сколько есть и сколько максимум. Например, при вызове **draw\_storage(draw\_tools, 10, 10, 200, 50, 800, 120)**результат должен быть примерно таким:  
     
   (так же обратите внимание на то, что текст здесь одновременно и с заливкой, и с обводкой);

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

**Общее** состоит в том, что во всех архитектурах определяются свои способы создания стандартных объектов - кнопок, текстов, рамочек и прочих. И у каждого объекта есть какой-то свой набор параметров. Способ пиксельного позиционирования объектов на экране так же применяется везде, без исключения (хотя так же почти везде применяются и другие способы позиционирования).

**Различие** состоит в том, что рассматриваемые нами здесь способы являются очень низкоуровневыми способами прямой отрисовки отдельных элементов. Мы, на самом деле, ещё не дошли до уровня, на котором наша программа понимает, что кнопка это кнопка (а значит, что на неё можно кликнуть, что она должна себя как-то вести итд.), мы всё ещё работаем на уровне простейших инструментов рисования. Во всех развитых архитектурах эти действия происходят "под капотом", скрыто от любопытных глаз и неопытных рук. Многие просто не дают прямой доступ к такой вот отрисовке, вместо этого сразу предлагая людям работать с готовыми классами объектов.  
Как создавать такие классы объектов - мы обсудим в одной из более поздних глав!