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.

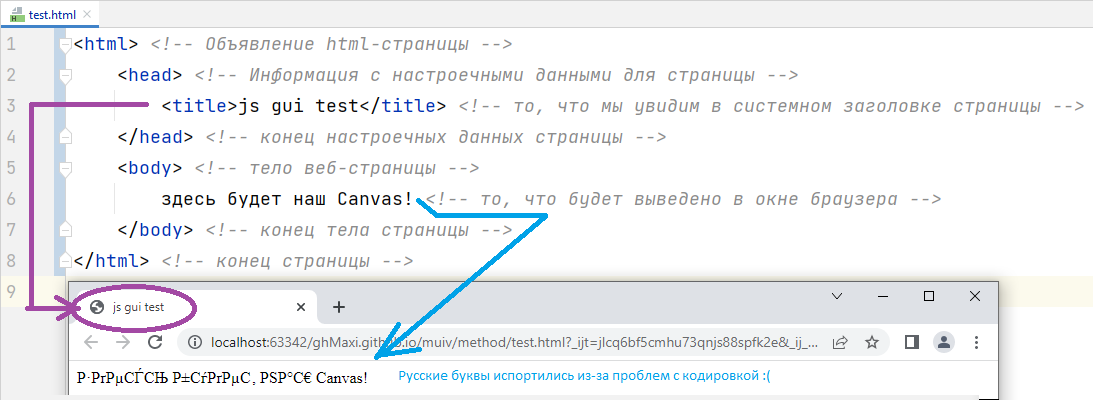
* Язык HTML состоит из **тэгов**, которые пишутся в **треугольных скобках**. При этом подавляющее большинство тэгов должны сначала открываться, а потом закрываться и всё, что находится между открытием и закрытием считается внутренностью этого тэга.  
  Например, вся наша исходная страница будет находится 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!

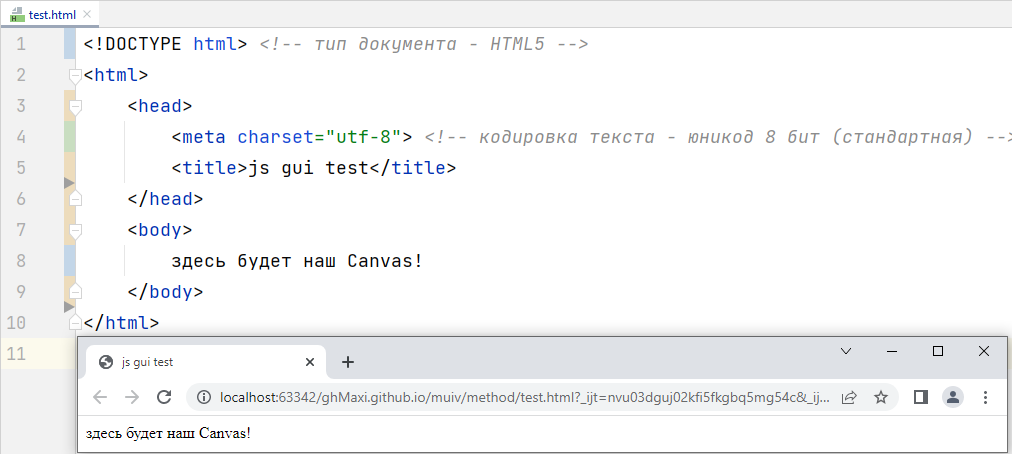
*Рис.1: файл test.html, версия 1*

**

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*

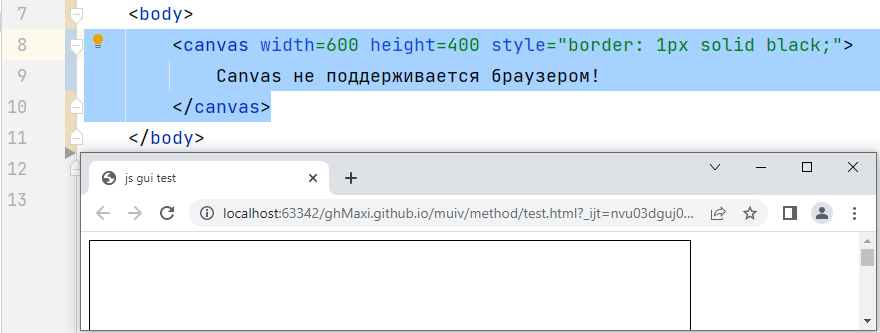
**

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>**.

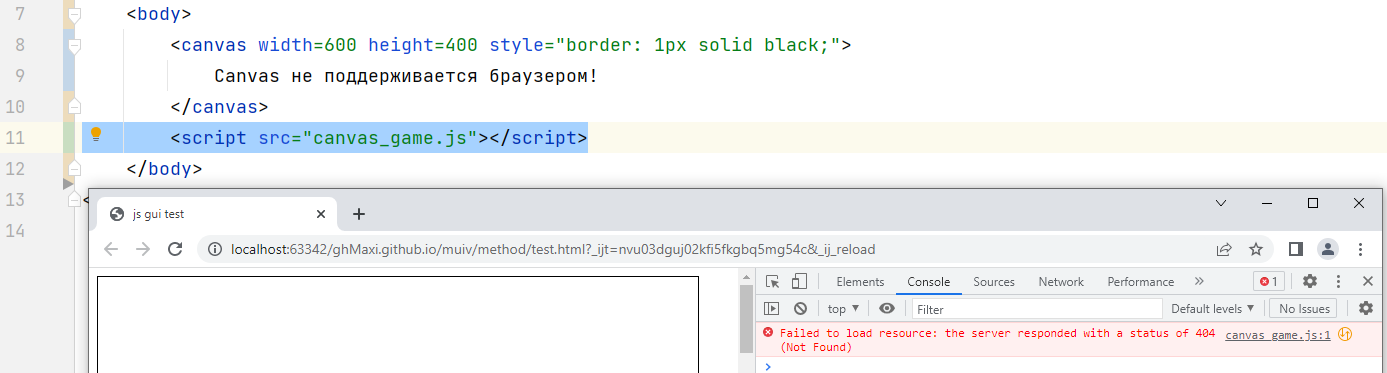
Кое-какие особенности использования тэга:

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*

**

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.

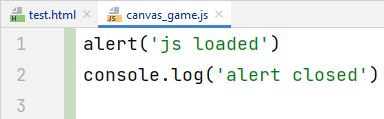
1.1.2. Первый JS файл. Рисуем пиксели на Canvas.

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!

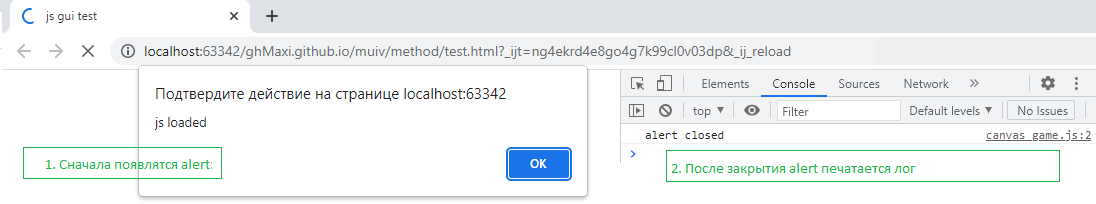
Создайте в одной папке с html-файлом файл **canvas\_game.js** и напишите в нём следующие строки:

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

**

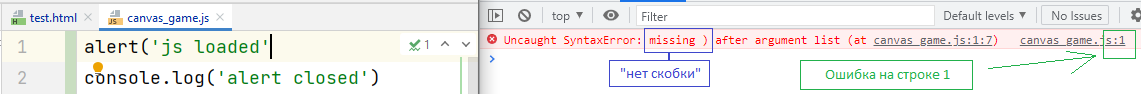
Первая строчка выводит окно с данным в скобках текстовым сообщением и останавливает работу скрипта, пока это окно активно. Вторая строчка печатает лог в уже знакомую нам по концу прошлой части консоль. Увидеть работу скрипта можно так же, как и раньше, запустив html-файл в браузере. Обратите внимание,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

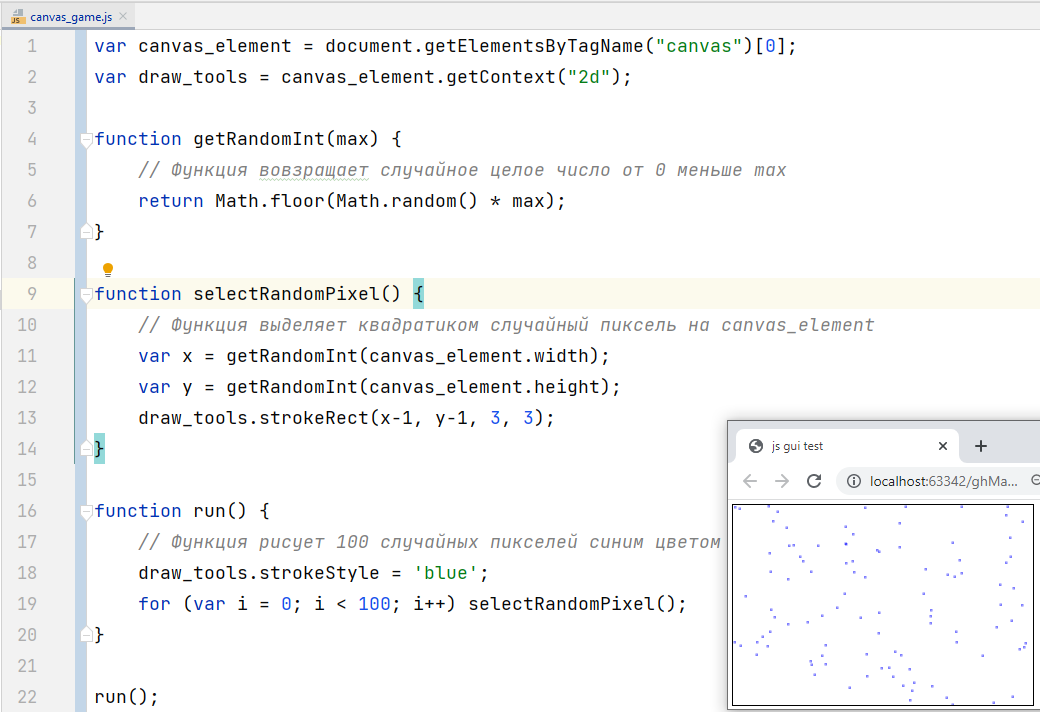
**

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 выбирать 100 точек в случайных местах на **<canvas>** и рисовать вокруг них рамочки. На картинке ниже приведён код такой программы и результат её работы в браузере. Обратите внимание, что программа значительно выросла в размере – теперь в ней более 20 строк!

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

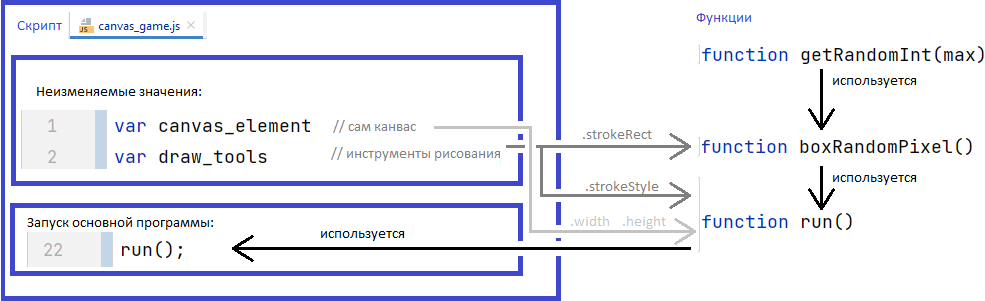


Воспроизведите у себя эту программу. В процессе воспроизведения, обратите внимание на кое-какие важные особенности:

* в вехней части определяются глобальные значения, которые назначаются один раз и после этого не изменяются в течение всей работы программы. Для нас такими значениями на данном этапе являются **canvas\_element** и получаемый из него **draw\_tools**. Оба эти значения используются в нашей программе в самых разных местах (поэтому объявляются в её начале);
* сама программа разделена на функции, каждая из которых имеет свою задачу. Причём чем ближе функция к началу скрипта, тем более узкую и конкретную задачу она выполняет;
* функция **get\_random\_int** выполняет самую элементарную задачу - выдать случайное целое. Она вызывает функцию встроенной библиотеки Math, подробнее о которых Вы можете почитать в документации библиотеки по этой ссылке: ­<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math>
* функции **box\_random\_pixel** и **run** выполняют более сложные задачи, которые описаны в комментариях после названия функций. Эти функции опираются на наш **draw\_tools**, и конкретно на функционал рисования прямоугольника (**strokeRect**) и функционал назначения стиля штриха (**strokeStyle**), в том числе его цвета;
* функция **run** является самой главной функцией этой программы, которая определяет порядок вызовов всех остальных функций. Она определяется самой последней и отдельно вызывается в конце скрипта на 22й строчке командой **run();**

Таким образом, мы с Вами подготовились к созданию нашей собственной интерфейсной архитектуры на основе **Canvas-JS**. На картинке ниже можно увидеть, из каких элементов сейчас состоит программа, и какие элементы где используются:

*Рис.9: текущая архитектура*



**Упражнения:**

1. Посмотрите внимательно на строчку 13 этой программы:  
     
   Попробуйте объяснить, что значат эти параметры. Посмотрите что будет, если Вы, например, поставите вместо 3 число 10, или поставите 0 вместо **x**-1.
2. Попробуйте заменить функцию **strokeRect** на **fillRect**. Что получилось?
3. Посмотрите на строку 18:  
     
   замените **'blue'** на **'green'**. Что получилось?замените **'blue'** на **'#FF0'**. Что получилось? Попробуйте объяснить, как это получилось (подсказка: RGB, 16-ричный код);
4. Сделайте 1000 точек вместо 100.
5. Сделайте так, чтобы программа рисовала 10 полностью закрашенных квадратиков размера 20х20.
6. Сделайте так, чтобы программа нарисовала 4 больших прямоугольника разных размеров в разных местах **canvas**.

**Задачи для самостоятельной работы:**

1. Нарисуйте 100 квадратиков случайного цвета и размера.
2. Замените код функции **draw** на следующий (код внизу можно копировать):  
     
   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);  
    }  
   }  
     
   Проверьте работу и 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.

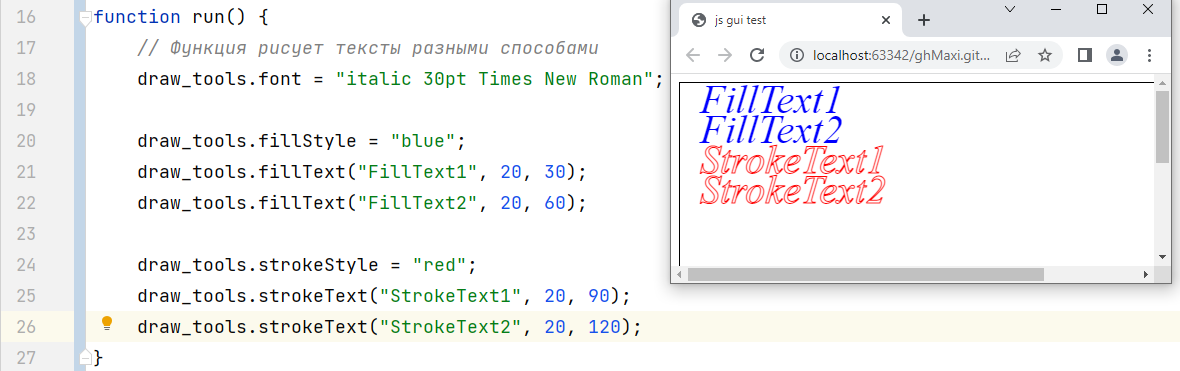
**Различие** состоит в том, что часто через такой аналог draw\_tools не предоставляют возможность получать базовую информацию о настройках экрана (такую как, например, ширина и высота), ограничивая сферу «ответственности» такого объекта только функциями рисования. В этом случае информацию о параметрах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

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

**

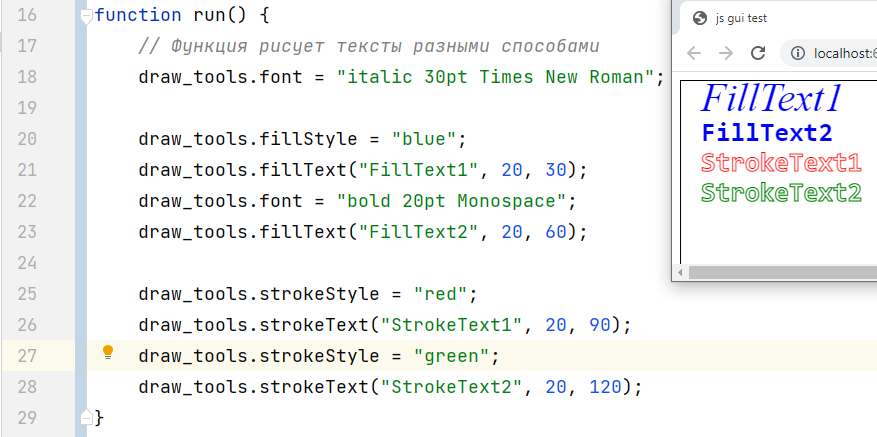
Обратите внимание на несколько особенностей этого кода:

* 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 **run** с изменёнными цветами и шрифтами ниже.

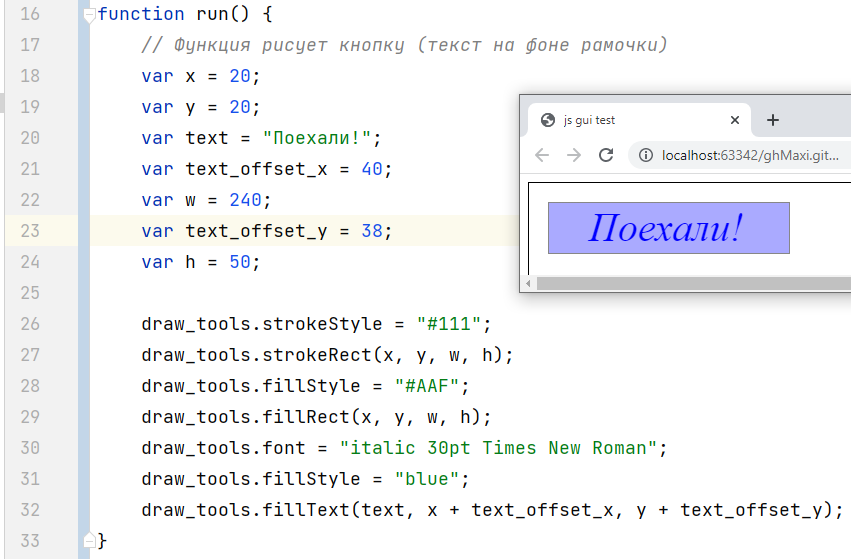
*Рис.12: функция отрисовки текстав с изменениями шрифта и цвета*

**

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 **run** таким образом, чтобы она выводила на экран одну кнопку.

*Рис.13: функция отрисовки одной кнопки*

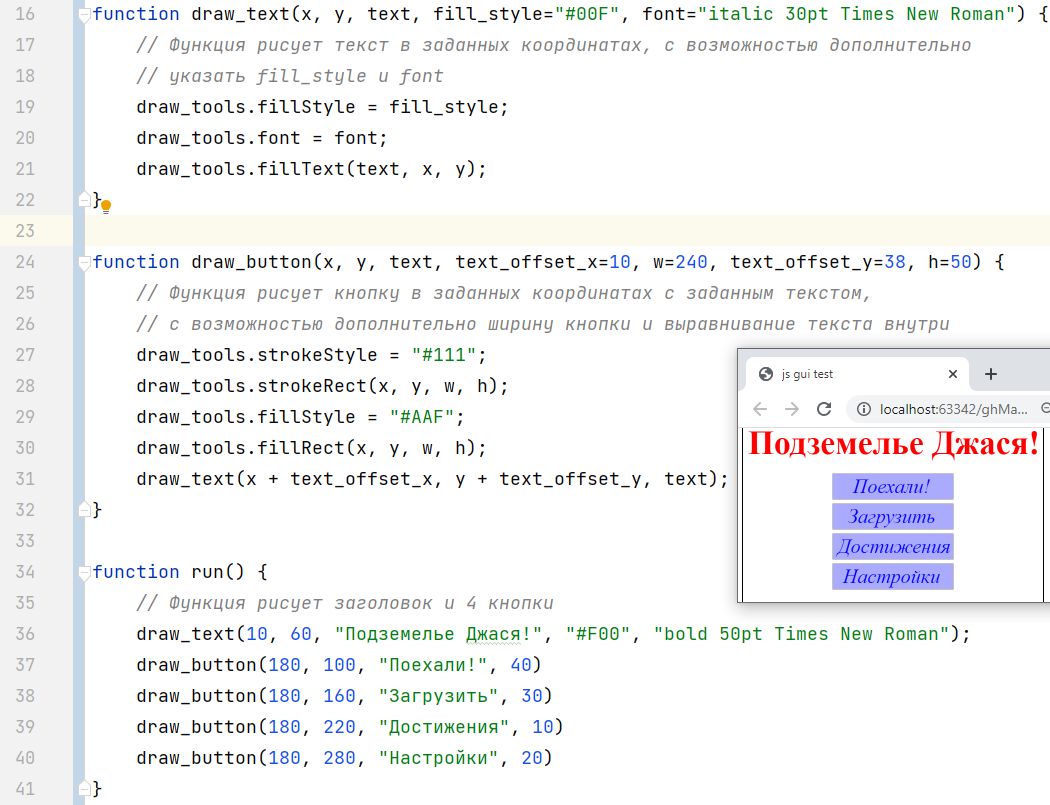


Обратите внимание на следующие моменты:

* цвета можно задавать не только словами, но и принятыми в Web-разработке краткими RGB-кодами (например, #111 - тёмно-серый, а #AAF - светло-синий);
* 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**;
* сначала рисуется прямоугольник с обводкой, потом текст. Это нужно чтобы текст отрисовался поверх прямоугольника, а не оказался им закрыт. Так же обратите внимание, как при необходимости один **fillStyle** заменяется на другой;
* 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). Функция **draw\_text** была веделена из той части старого run, где рисовался текст. Функция **draw\_button** так же была выделена из старого **run**, но так же теперь сама вызывает **draw\_text**, когда нужно рисовать текст кнопки;
* переменные **x**, **y**, **text**, **text\_offset\_x**, **w, text\_offset\_y** и **h**, которые мы в старом **run** задавали вручную, теперь стали параметрами функции **draw\_button**. Те, без которых сама идея рисовать кнопку бессмысленна (координаты и текст внутри) стали обязательными параметрами, а остальные получили значения по умолчанию. Аналогично с **draw\_text**;
* параметры со значением по умолчанию можно при вызове не указывать, но можно и указывать, если надо назначиь свои. Например, когда функция **draw** вызывает на 36й 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 новые значения **fill\_style** и **font**. Параметры при вызове пишутся и передаются в том же порядке, что и при объявлении функции (подробнее на картинке ниже);  
    
  <<!!!>>
* с другой стороны, параметры со здначениями по умолчанию можно и не передавать при вызове. Например, когда **draw\_text** вызывают на 31й строчке, ему даются только первые четыре параметра - **draw\_tools**, **text** и перерассчитанные с учётом параметров **text\_offset** значения **x** и **y**. **fill\_style** и **font** не передаются и поэтмоу функция использует значения по умолчанию. Аналогично с рисованием кнопок, которые используют стандартные значения ширины (**w**=240), выравнивания текста по вертикали (**text\_offset\_y**=38) и высоты (**h**=50);
* хотя количество элементов GUI увеличилось в 5 раз, обьём кода вырос меньше, чем на треть! Более того, каждую новую кнопки или текст мы сможем добавлять одной строчкой! В этом сила правильно написанных функций;

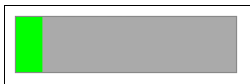
В конечном итоге, обратите внимание, что наша функция **draw**, с одной стороны, повысила уровень своей ответственности (теперь она работает с целым набором интерфейсных элементов, а не с одной кнопкой), а с другой - стала проще (все сложные взаимодействия с **draw\_tools** ушли внутрь двух служебных функций, которые **draw** может просто вызывать). 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(x, y, w, h, percent)**, которая будет выводить на экран полоску выполнения, заполненную на заданное число процетов. Например, вызов **draw\_progress\_bar(10, 10, 200, 50, 12);** должен давать заполненную на 12% полоску:  
   
2. Напишите функцию **draw\_storage(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(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. 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). Так же возникающая у нас функция **run**, отвечающая за запуск всей программы, так же в том или ином виде присутствует во всех GUI-решениях (хотя до реального наполнения этой функции мы ещё не дошли);

**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 (то есть, что на неё можно кликнуть, что клик должен что-то вызвать 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!

1.1.3 Рисование картинок

Мы почти целиком разобрали вопросы отрисовки, и даже научились рисовать простейшие элементы графического интерфейса в виде текстов, рамочек и комбинирующих их кнопок.

Последний оставшийся у нас технический вопрос рендеринга - вопрос о том, как в интерфейсе рисовать картинки из файлов. Действительно, при определённой сноровке можно, конечно, весь интерфейс сделать на уже разобранных нами прямоугольниках и текстах. Но сделанный таким образом интерфейс будет крайне ограничен в плане визуальной стилистики.

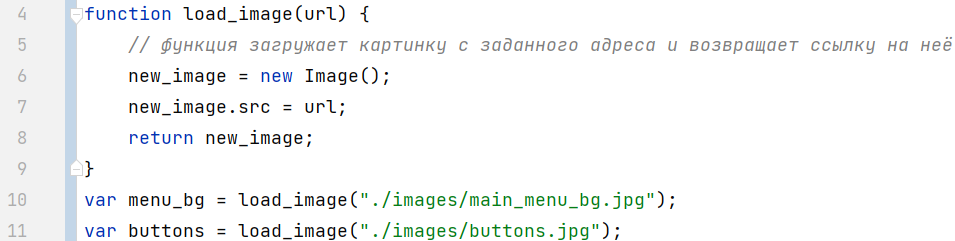
Давайте попробуем реализовать что-то поинтереснее! Например, давайте сделаем так чтобы в нашем титульном листе приложения "пещера Джася" на фоне была картинка пещеры, а сами кнопки были оформлены так, будто они высечены на камне. Для этого воспользуемся следующими картинками:

*Рис.15: фон для меню (ссылка:* [*https://cdnb.artstation.com/p/assets/images/images/001/421/529/large/galina-levchenko-112-stonehallway.jpg*](https://cdnb.artstation.com/p/assets/images/images/001/421/529/large/galina-levchenko-112-stonehallway.jpg)*)*

*Рис.16: каменные кнопки (ссылка:* [*https://cdn3.vectorstock.com/i/1000x1000/14/82/cartoon-stone-buttons-set-vector-6511482.jpg*](https://cdn3.vectorstock.com/i/1000x1000/14/82/cartoon-stone-buttons-set-vector-6511482.jpg)*)*

Первое, что надо сделать, - чтобы обе картинки загружались в нашу страницу. Добавим функцию загрузки картинок и запустим эту самую загрузку в начале нашего скрипта, сразу после создания draw\_tools (на скришоте ниже вставляемый в начало код выделен синим).

*Рис.17: загрузка картинок в переменные в JS*

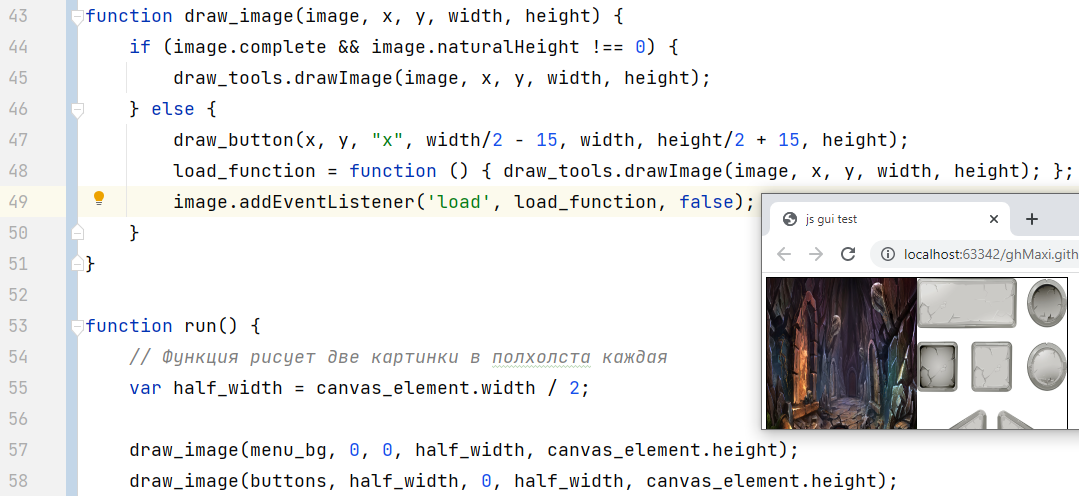


Здесь надо понимать следующее: так как **Canvas-JS** изначально архитектура для Web-приложений, существующая в браузере, любая картинка (и любой другой файл) в этой архитектуре загружается в по ссылке так, будто это происходит из Интернета. Конкретно, когда мы пишет **new Image()**, мы говорим что у нас в принципе будет картинка, а когда мы даём созданной таким образом картинке настройку **.src**, начинается процесс загрузки этой картинки по заданному адресу.

То есть после выполнения кода выше картинки только начинают загружаться, но ещё не загрузились до конца!

Запустив таким образом загрузку всех нужных нам картинок в верхней части скрипта, мы теперь можем организвать отрисовку этих картинок через функцию **run** (с соответствующей вспомогательной функцией). Напишем программу таким образом, чтобы она рисовала картинку если та уже загружена, а если нет - то рисовала под эту картинку прямоугольник-заглушку и ждала загрузки.

*Рис.18: отрисовка картинок или заглушек под них*



Разберём этот код:

* функция **run** очень простая - она дважды просит нарисовать картинку, в одном случаю из переменной **menu-bg** в левой части экрана, а в другом - из переменной **buttons** в правой части экрана. Основные сложности внутри вызываемый функции **draw\_image**;
* **draw\_image** начинает с проверки наличия картинку. Если загрузка картинки не завершена, то **image.complete** будет **False**. Если же в процессе загрузки что-то пошло не так, что **image.complete** будет **True** (потому что процедура загрузки всё равно завершена), но высота картинки будет 0. Если обе эти проверки проходят, то можно рисовать картинку!
* если же одна из этих проверок не проходит, то функция делает две вещи: рисует большую кнопку-заглушку с текстом "х" и готовиться рисовать саму картинку. Вы увидите эту заглушку если, например, испортите адрес картинки (и тогда она не сможет загрузиться);
* подготовка к моменту загрузке картинки начинается на строке 48 и содержит в себе несколько необычных действий. Сначала создаётся "слушающая функция" **load\_function**. Эта функция необычна, прежде всего, тем, как она объявлена - прямо в тексте кода! Такое объявление повзоляет каждой ожидающей загрузки картинке получить свою копию этой функции, с тем чтобы выполнить её когда загрузка произойдёт. Фактически каждый вызов draw\_image для незагруженной картинки "испекает" свою новую и отдельную **load\_function**;
* вторая необычность этой функции - то, как она вызывается. Мы никогда не вызываем её в явном виде. Вместо этого, на 49й строчке мы добавляем эту свежеиспечённую функцию **load\_function** к слушателям события **'load'** нашей кнопки. С точки зрения браузера - событие **'load'** будет объявлено в тот момент, когда картинка будет загружена. Это означает, что когда наш **image** скажет, что у него произошёл **'load'**, все слушающие это событие функции, включая наш **load\_function**, выполнятся автоматически. В конечном итоге, это означает автоматический вызов **load\_function** в момент успешной загрузки картинки;
* наконец, сама **load\_function** просто рисует картинку;

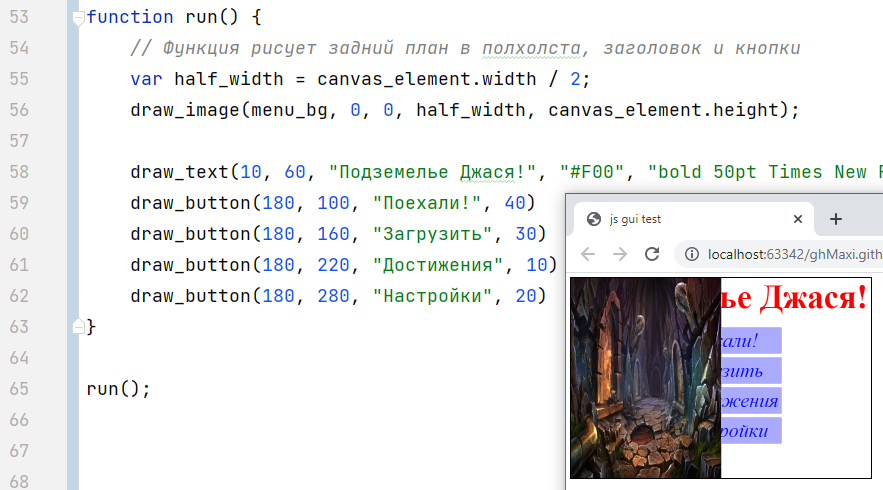
Итак, нам вроде удалось выполнить загрузку картинок. Но, на самом деле, в этой реализации кроется одна проблема, которую мы увидим на следующем шаге, когда попытаемся наложить на наши картинки сверху кнопки.

Модифицируем функцию run следующим образом:

* убрав из неё отрисовку кнопочного атласа;
* вернув в неё старые отрисовки кнопок;
* пока не будем делать каменные плашки и пока не будем растягивать фон на всю ширину экрана;

Мы ожидаем, что увидим кнопки поверх задника, но на самом деле будет наоборот (код и результат на картинке ниже).

*Рис.19 фон на фоне кнопок*



Почему так происходит? Очень просто. Вызывая draw\_image, мы заказываем отрисовку картинки, но не получаем её сразу! Вместо этого, компьютер начинает её загружать. Пока он её загружает, он успевает отрисовать заголовок и кнопки. А дальше завершается загрузка картинки, и компьютер честно рисует её поверх всего.

Мы в рамках этого учебного пособия применим самый простой и прямолинейный способ решения задачи - когда у нас загрузиться картинка, мы просто заново вызовем отрисовке всего в нужном нам порядке. Тогда уже загруженная картинка встанет туда, куда надо. Добьёмся мы этого следующими изменениями в коде:

* переименуем старую функцию **run** в **draw**, тем самым ограничивая её зону ответственности исключительно вопросами рендеринга (строка 53 в коде внизу). Заодно растянем картинку на фоне на весь холст (строка 55);
* создадим новую ответственную за всё функцию **run**, которая пока будет просто вызывать **draw** (строки 64-67);
* выкинем из **load\_image** создание **load\_function** и вместо неё просто подпишем на событие '**load'** функцию **draw** (строка 49);

После этих изменений в коде, мы наконец-то увидим наше меню с кнопками и заголовком на фоне задника из картинки.



На этом, мы подводим черту под вопросами рендеринга. Мы разобрались, что такое порядок отрисовки и какие существуют подводные камни в процессе низкоуровневой работы с пиксельным экраном. Нас ждут упражнения и следующий раздел!

**Упражнения:**

1. Попробуйте загрузить картинку напрямую из Интернета, дав ей соответствующую ссылку как **src** (Картинка в каких-то случаях загрузится, в каких-то нет);

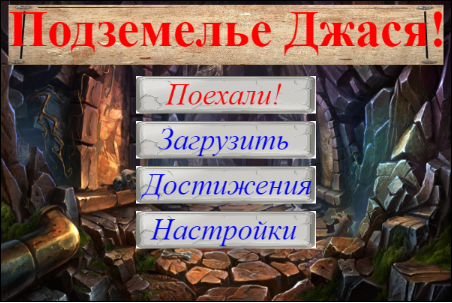
2. Найдите в Интернете, скачайте и вставьте классный фон для заголовка;

3. Изспользуя любой инструмент редактирования картинок, вырежьте из атласа кнопок задник и сделайте так, чтобы кнопки были на фоне камней. Для этого надо будет:

* добавить в **draw\_button** вызов соответствующего **draw\_image**;
* убрать из **draw\_image** вызов **draw\_button** для рисования заглушки, чтобы не было цикличноо вызова, заменив его чем-то другим (или просто отказавшись от заглушек);

4. Сделайте кнопку "Поехали" отличающейся от других, чтобы на неё хотелось кликнуть;

Результатом упражнений может быть что-то такое:



**Задачи для самостоятельной работы:**

1. Сделайте так, чтобы у Вас в меню на переднем фоне была картинка с прозрачностью (это может быть куст, дерево, персонаж или что угодно ещё). Для этого надо будет разобраться, как выводить картинки с прозрачностью на **canvas**;

2. Существует способ выведения в canvas только части картинки, что позволяет, например, не нарезать атлас кнопок на кучу отдельных файлов, а использовать один файл, выставяляя на экран только нужные элементы. Сделайте так, чтобы задники кнопок в Вашем меню рисовались из такого вот атласа, а не из собственного файла;