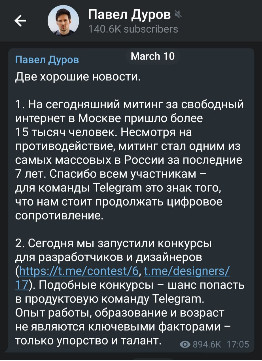
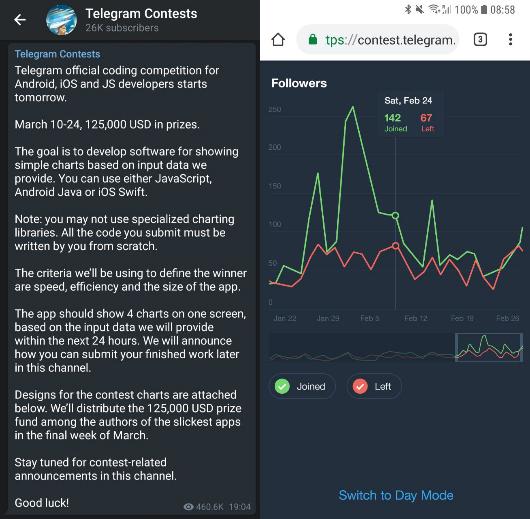
Как я не занял первое место в конкурсе для JavaScript-разработчиков от Telegram

* [Блог компании ГК ЛАНИТ](https://habr.com/ru/company/lanit/" \o "Вы не подписаны на этот хаб),
* [JavaScript](https://habr.com/ru/hub/javascript/),
* [Программирование](https://habr.com/ru/hub/programming/),
* [Canvas](https://habr.com/ru/hub/canvas/),
* [TypeScript](https://habr.com/ru/hub/typescript/)

Активные пользователи Телеграма, особенно те, кто подписан на Павла Дурова, наверняка что-то слышали о том, что Телеграм проводил в этих ваших интернетах конкурс для iOS, Android и JavaScript разработчиков, а также для дизайнеров. Несмотря на то, что это было довольно эпичное событие с раздачей солидных призов (один из участников получил 50к долларов за первое место, написав самое быстрое и лёгкое приложение для Android), о нём как-то слабо писали, во всяком случае в Рунете. Своим дебютным постом попробую исправить ситуацию.



Коль скоро я являюсь фуллстек JavaScript-разработчиком (если совсем точно, то TypeScript-разработчиком), я решил испытать себя. Манил не только призовой фонд, но и сам формат: это не соревнования по программированию, где важны абстрактность и скорость мышления. Здесь было важно всё в комплексе: опыт, скорость разработки в среднесрочной перспективе, вкус в вопросах UI, знание computer science в целом, самокритичность. По условиям конкурса необходимо было разработать библиотеку для отображения графиков для одной из платформ: iOS, Android или Web.



Разработчики для разных платформ не конкурировали между собой, и у каждой платформы победители были свои. Основными критериями были: скорость работы (в том числе и на старых устройствах), соответствие дизайну, плавность анимации и минимальный размер приложения. Уже существующие решения и библиотеки использовать было нельзя, всё должно было быть написано с нуля.  
  
До этого я участвовал в конкурсах для разработчиков, где на все задачи выделялось не более 5 часов, эти часы приходилось проводить в огромном напряжении. Несмотря на то, что выполнение задачи в конкурсе от Телеграма не требовало такого напряжения, это один из самых сложных конкурсов, в которых мне приходилось участвовать. С виду несложная задача оказалась настолько ёмкой, что если бы мне за это платили, я бы мог пилить эти «графики» месяцами, пытаясь найти компромисс между производительностью кода и архитектурной его стройностью. Выручало то, что на решение выделялось три (**upd:** две, спасибо [vlad2711](https://habr.com/ru/users/vlad2711/) за поправку) недели. Некоторые из соперников специально брали отпуск, чтобы уделить конкурсу больше времени, а я решил совмещать разработку для конкурса по вечерам и выходным с работой в "[Онланте](https://www.onlanta.ru/)" в обычном режиме.

CANVAS versus SVG

Самый главный архитектурный вопрос, вставший перед всеми нами, был в выборе инструмента отрисовки графики. На текущий момент веб-стандарты предлагают нам два подхода: через генерацию «на лету» svg-графики и старый добрый canvas. Вот плюсы и минусы каждого из них.

Canvas

**+** Абсолютная универсальность — имея возможность изменить цвет любого пикселя на полотне, можно нарисовать всё, что угодно.  
**+** [Потенциально] Высокая производительность — если уметь готовить canvas, он может показывать неплохую производительность. Было бы замечательно использовать webgl, но его поддержка на смартфонах оставляет желать лучшего.  
  
**-** Все расчёты и вся отрисовка вручную — в отличие от SVG, где промежуточные точки ломаной можно задать единожды, а далее можно манипулировать viewbox-ом для перемещения «камеры» по участкам ломаной, с canvas всё сложнее: никаких «камер» тут нет, есть только координаты от левого верхнего угла; если нужно «переместить» текущую область просмотра графика, необходимо заново рассчитать все координаты всех его точек относительно новой позиции области просмотра. Другими словами viewbox, который в svg есть из коробки, в canvas нужно реализовывать вручную.  
**-** Вся анимация вручную — исходя из предыдущего пункта, все возможные анимации реализуются посредством пересчёта координат, значений цвета и прозрачности и перерисовке всей сцены N-е количество раз в секунду, и чем большее количество раз удалось пересчитать и перерисовать сцену, тем плавнее анимация.

SVG

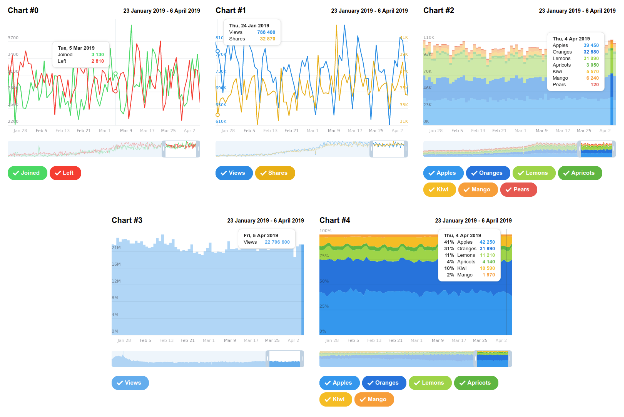
**+** Простая отрисовка — достаточно один раз добавить в SVG необходимые линии, фигуры и далее можно, манипулируя viewport, параметрами цвета и прозрачности, обеспечить навигацию по графикам.  
**+** Простая реализация анимаций — опять же, исходя из предыдущего пункта, достаточно N-e количество раз в секунду указать новые значения для viewbox, цвета и прозрачности, а изображение перерисуется само, об этом позаботится браузер. Кроме того, не стоит забывать, что фигуры и примитивы в SVG можно стилизовать в CSS, поэтому их можно анимировать с помощью CSS3-анимаций, что открывает широчайшие возможности для получения крутых анимаций с минимальными усилиями.  
**+** Неплохая производительность по умолчанию — если с canvas можно легко, что называется «в лоб», накодить что-то медленное и жрущее сотни ресурсов, то результат, основанный на SVG всегда будет выглядеть вполне легковесным, приличным и плавным.   
  
Но есть и обратная сторона медали.  
  
**-** Скромные возможности для оптимизации — поскольку svg рисуем не мы, а браузер, то и контролировать этот процесс невозможно — если хочется увеличить производительность, например, за счёт кэширования уже отдельных отрисованных элементов, сделать это нельзя никак. Скорее всего это уже делает браузер, но мы не можем быть уверены до конца.  
**-** Ограниченность инструментария — в SVG мы уже не контролируем каждый пиксель полотна, а думаем и кодим в рамках векторных примитивов. Впрочем, для этой задачи это несущественный минус, накладывающий некоторые, опять же несущественные ограничения в контексте задачи конкурса.  
  
Выбором инструмента мучиться мне не приходилось никогда, поскольку у меня есть отвратительная черта характера — я максималист и привык использовать в работе только любимый инструмент. Так получилось, что еще со студенческих времён, когда я забавлялся с DirectDraw, любимым моим инструментом всегда было полотно, на котором «делай что хочешь». И canvas для решения конкурсной задачи действительно оказался хорош, но по-настоящему сыграл мне на руку лишь один его плюс: широчайшие возможности для оптимизаций, поскольку основным критерием была всё-таки производительность приложения.

Хороший код нехороший

Задача ясна: нужно рисовать точки на полотне в нужном месте и в нужное время. Осталось написать код. Снова нужно было выбирать, на этот раз между написанием производительного компактного кода одной «портянкой» в процедурном стиле или не очень производительного и уж тем более некомпактного в моём любимом объектно-ориентированном. Наверное, вы уже догадались, что я выбрал второй вариант, приправив его ещё одним моим любимцем — TypeScript.  
  
И этот выбор оказался не очень правильным. Из-за использования абстракций и инкапсулирования не везде получается сохранять, передавать и повторно использовать промежуточные результаты вычислений, что плохо сказывается на производительности. А из-за повсеместного использования this, без которого ООП в JS невозможен, код плохо минифицируется, тогда как размер тоже имел значение.  
  
Настало время дать ссылку на гитхаб: [github.com/native-elements/telechart](https://github.com/native-elements/telechart). Если интересно, рекомендую обратить внимание на историю коммитов, она хранит память об оптимизационных мытарствах и небезуспешных попытках выжать пару лишних кадров отрисовки в секунду.  
  
Ну а в конкурсе я не занял призового места. И проблема, как это часто с нами-программистами бывает, оказалась не в недостаточном опыте, сообразительности или скорости, а в недостаточной самокритичности: сам факт того, что у меня получилось сделать, оно работает и выглядит как на картинке, меня порадовал, а по поводу тормозов отрисовки я думал, что я сделал всё, что мог, у остальных наверняка так же. Стыдно об этом говорить, но я был уверен, что займу первое-второе место. На деле же оказалось, что я написал тормозную и глючную программу, не самую плохую, но и далеко не самую хорошую. Когда я увидел работы других разработчиков, понял что у меня нет шансов и оставалось только кусать локти. Будь я беспристрастен к своему труду, я бы занялся производительностью, самой важной частью конкурсного задания.  
  
Один из ценнейших уроков в моей профессиональной жизни, который я не устаю получать, заключается в том, что хороший инженер в отличие, например, от художника, обязан объективно оценивать качество своей работы, отбросив самоуверенность, потому что результат его труда должен не только глаз радовать, но должен правильно и хорошо работать.  
  
Это был первый этап конкурса. Победители были щедро вознаграждены. К моей неописуемой радости на этом история не закончилась, потому что был анонсирован второй этап:



Необходимо было доработать свою поделку, всего лишь за неделю реализовав дополнительные типы графиков. Покажу сразу, что получилось, а ниже расскажу как это получилось.



В моём случае, прежде чем добавлять новую функциональность, нужно было разобраться с производительностью старой. Первая проблема, которую я решил — это  
  
**Дёрганая анимация**  
  
Даже если вам хватает мощностей, чтобы выдавать 60 кадров в секунду, анимация не будет плавной, если положение элемента или его прозрачность не детерминированы временем, прошедшим с начала анимации. Это обусловлено неравными промежутками времени между тиками: например, один тик сработал через 10 мс, а второй — через 40, в то время как и за первый, и за второй тики объект переместился влево на 1 пиксель — то есть скорость его перемещения постоянно плавает, визуально это выглядит как «подёргивание». Иными словами, нужно делать не так:

let left = 10, interval = setInterval(() => {

left += 1

if (left >= 90) {

clearInterval(interval)

}

}, 10)

А так:

let left = 10, startLeft = 10, targetLeft = 90, startTime = Date.now(), duration = 1000, interval = setInterval(() => {

left = startLeft + (targetLeft - startLeft) \* (Date.now() - startTime) / duration

if (left >= targetLeft) {

left = targetLeft

clearInterval(interval)

}

})

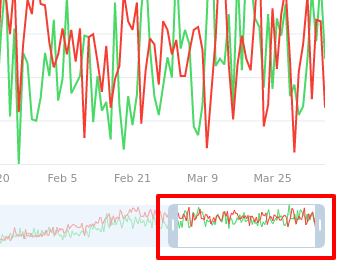
Поскольку анимируемых параметров в коде много, я запилил [универсальный класс](https://github.com/native-elements/telechart/blob/master/src/Telemation.ts), который облегчает задачу, да ещё и добавляет изинг к анимации. Он достаточно прост в использовании:

let left = Telemation.create(10, 90, 1000)

…

drawVerticalLine(left.value) *// В любое время здесь будет нужное, детерминированное значение.*

Дальше в игру вступает правило 60 fps. ПК-геймеры меня поймут: чтобы анимация выглядела идеально, она должна отрисовываться со скоростью не менее 60 fps. Соответственно, каждая отрисовка кадра должна занимать не более 1/60 секунды. Для этого нужно мощное железо и хороший код.  
  
Дальнейшие изыскания показали, что  
  
**Прорисовка canvas тормозит, если над canvas есть html-элементы**.  
  
Изначально я использовал «пустые» html-элементы для того, чтобы реализовать управление текущей областью просмотра:



Эти элементы располагались поверх canvas, и несмотря на то, что у них не было никакого контента, они использовались только для отслеживания событий мыши, в результате экспериментов выяснилось, что само их наличие снижает производительность отрисовки. Убрав их и немного усложнив логику определения событий управления областью просмотра, я увеличил скорость отрисовки кадра.  
  
Оставалось выдернуть последний гвоздь из крышки гроба производительности: я сделал  
  
**Кэширование миникарты**  
  
До этого для миникарты линии отрисовывались каждый кадр заново. Это дорогая операция, потому что на ней отображался весь график за год (365 точек на каждую линию). Очевидным решением, которое я просто поленился реализовать с самого начала, было однократное отрисовывание линий графика для миникарты, сохранение результата в кэш и использование этого кэша в дальнейшем. После этой оптимизации за производительность приложения перестало быть стыдно.

Дальше что?

Было ещё много успешных и не очень драк за производительность: попытки кэшировать результаты вычислений координат, эксперименты с параметрами lineJoin у CanvasRenderingContext2D (miter быстрее), но они не так интересны, так как не давали заметного выигрыша в производительности либо не давали его вообще.  
  
Из восьми дней пять я потратил на ускорение кода и только три — на допиливание новой функциональности. Да, мне хватило всего три дня, чтобы добавить новые типы графиков, и тут весьма кстати оказался ООП, с ним кодовая база увеличилась незначительно. Мне не хватило времени, чтобы выполнить бонусное задание (ещё +5 дополнительных графиков). Полагаю, что те пять дней, которые я потратил на устранение последствий моей уверенности в себе, я мог потратить на решение бонусной задачи.  
  
Тем не менее мои труды дали результат: 4-е место и «утешительный» приз в одну тысячу долларов:



Кстати, конкурс продолжился дальше, но уже без меня.  
  
Я доволен участием: кроме того, что это просто интересно и является интересным приключением, я получил хороший профессиональный опыт и жизненный урок.  
  
Кроме того, эту библиотеку я использовал в разработке нашего корпоративного таймтрекера, о котором тоже планирую рассказать на Хабре в ближайшее время.  
  
Для обсуждения предлагаю такой вопрос: зачем Телеграму это всё нужно? Я считаю, что за адекватные деньги Телеграм получит самую лучшую в мире библиотеку для отображения графиков: лучший результат из сотен попыток сделать лучше, чем у других. Соревновательный принцип позволяет получить настолько высокий уровень качества, который на заказ не способен сделать никто и ни за какие деньги.