МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение высшего образования

«РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ ГИДРОМЕТЕОРОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра «Экономики и управления на предприятии природопользования»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

(бакалаврская работа)
по направлению подготовки 09.03.03 Прикладная информатика
(квалификация – бакалавр)

На тему «Оптимизация ресурсоёмкого браузерного приложения с использованием новых веб-технологий»

Исполнитель Приходченко Игорь Николаевич

Руководитель к.г.н., доцент по кафедре Информатики и прикладной математики, Полупанов Владимир Николаевич

«К защите допускаю»

Руководитель кафедры ____________________

кандидат экономических наук

Майборода Евгений Викторович

«18» 01 2026 г.

Филиал Российского государственного гидрометеорологического университета в г. Туапсе
НОРМОКОНТРОЛЬ ПРОЙДЕН
«16» 01 2026 г.
_______________ / _______________ /
подпись расшифровка подписи

Туапсе
2026

ОГЛАВЛЕНИЕ

Введение……………………………………………………………..................3

1 Вычисления и визуализация в среде браузера………………………………..5

1.1 Анализ предметной области……………………………………………...5

1.2 Стандарт ECMAScript и среда браузера………………………………6

1.3 Визуализация в HTML4 Tables и HTML5 Canvas……………………….10

1.4 Вычислительные возможности с WebWorkers…………………………...13

2 Оптимизация браузерного приложения для имитационного моделиро-вания19

2.1 ES5-вариант приложения и варианты его оптимизации с ES6…………19

2.2 Визуализация цветовых карт в HTML5 Canvas………………………….31

2.3 Распараллеливание вычислений с WebWorkers…………………………34

2.4 Проект оптимизации браузерного приложения…………………………50

3 Оценка эффективности оптимизации браузерного приложения……………...53

3.1 Вычислительная эффективность..……………………………………53

3.2 Экономическая эффективность..……………………………………….53

Заключение..…………………………………………………………………55

Список использованной литературы………………………………………...57

Приложение ..………………………………………………………………...60

Введение

Современные браузеры, GoogleChrome, MozillaFirefox, Safari и другие, всё чаще используются в качестве среды выполнения для ресурсоёмких алгоритмов. Это во многом происходит за счёт увеличения быстродействия интерпретаторов языка Javascript и технологий манипулирования данными с использованием этого языка в среде браузера без использования сервера. Эти возможности, наряду с преимуществами языка высокого уровня, каким является Javascript, и его средствами создания пользовательского веб-интерфейса, позволяют в короткие сроки и небольшими силами создавать прикладные программы, обладающие достаточной вычислительной эффективностью.

Однако невероятно быстро развивающиеся веб-технологии постоянно создают новые возможности для повышения эффективности Javascript-программ.

Актуальность настоящей работы обусловлена малочисленностью исследований, рассматривающих возможности оптимизации ресурсоёмких вычислений за счёт использования новых браузерных технологий. В данной работе рассматриваются такие возможности на примере приложения, предназначенного для имитационного моделирования распространения загрязнения. Исследование возможностей использования новых браузерных технологий для оптимизации имитационного моделирования представляют научный и практический интерес.

Объект исследования: имитационное моделирование в среде браузера.

Предмет исследования: возможности повышения вычислительной эффективности имитационного моделирования за счёт новых браузерных технологий.

Цель исследования: разработать проект оптимизации браузерного приложения, позволяющего более эффективно осуществлять имитационное моделирование.

Для достижения поставленной цели необходимо решить задачи:

– исследовать средства клиента (браузера), которые можно использовать для реализации ресурсоёмких вычислений;
– рассмотреть проблемы ресурсоёмкого приложения для имитационного моделирования, использующего возможности стандарта ECMAScript 5 и более ранних;
– провести эксперименты с новыми технологиями ECMAScript 6 на предмет повышения вычислительной эффективности имитационного моделирования;
– разработать проект преобразования приложения для имитационного моделирования на основе новых браузерных технологий;
– оценить вычислительную и экономическую эффективность приложения при реализации предложенного проекта оптимизации.

1 Вычисления и визуализация в среде браузера

1.1 Анализ предметной области

Задача оптимизации для настоящего исследования сформировалась при рассмотрении браузерного приложения для моделирования загрязнения водных объектов (pvn.org.ru, [14]), в котором для моделирования установившихся течений и последующей транспортировки загрязнения, поступающего в воду, использован механизм дискретных клеточных автоматов (КА) [15]. Преимуществом данного приложения является возможность работы на бюджетных компьютерах, которые могут себе позволить региональные экологические организации. Однако в некоторых случаях моделирование может длиться десятки часов, что приводит к необходимости ускорения вычислений. Напрашиваются два подхода решения проблемы длительности вычислений: первый, затратный, когда приобретаются всё более мощные компьютеры, и второй, по сути - без затрат, с использованием новых веб-технологий. Понятно, что настоящая работа посвящена исследованию возможностей второго подхода.

Механизм дискретных клеточных автоматов [15] используется в качестве альтернативы моделям на основе численных решений непрерывных дифференциальных уравнений, в том числе для моделирования загрязнения воздуха [2] и, как уже упоминалось, водных объектов [14].

Клеточный автомат в последних работах представляет собой дискретную динамическую модель, функционирование которой происходит в двумерном пространстве, представляющим собой множество ячеек (клеток) одинаковой формы и размера, и дискретные промежутки времени, называемые тактами (циклами). Каждая клетка модельного пространства соответствует некоторой области моделируемого физического пространства (атмосферы, моря, водохранилища и т.п.) и описывается набором безразмерных характеристик.

Моделирование при помощи клеточных автоматов заключается в изменении состояний ансамбля клеток автомата по истечении заданного количества тактов работы автомата. Полученное множество состояний описывает динамику исследуемого процесса или объекта (скорость потока жидкости в отдельных точках, концентрацию веществ и т.д.).

Несмотря на вычислительные преимущества перед моделями на основе численных решений непрерывных дифференциальных уравнений, компьютерные реализации КА-моделей всё же относятся к ресурсоёмким и могут занимать достаточно много времени. Возникает необходимость оптимизации для ускорения моделирования. Как представляется, для этого можно использовать новые веб-технологии, рассматриваемые далее в этом разделе.

1.2 Стандарт ECMAScript и среда браузера

Вначале, следуя [1], уточним терминологию:

– ECMA - это стандарт, разрабатываемый ECMAInternational, который имеет право на стандартизацию синтаксиса и функциональности языка JavaScript. Термин ECMAScript был введён для стандартизации языка и является официальной торговой маркой, но не имеет права называться “JavaScript” - торговой маркой, принадлежащей компании Oracle;

- ECMAScript - это встраиваемый, расширяемый, не имеющий средств ввода-вывода, язык программирования общего назначения, используемый в качестве основы для построения скриптовых языков;
- JavaScript - это реализация спецификации ECMAScript, JIT-компилируемый и интерпретируемый скриптовый язык программирования, используемый для выполнения вычислений и управления вычислительными объектами среды выполнения;
- JIT-компиляция - это одна из форм динамической компиляции, компиляция “точно в срок” (Just-In-Time), относящаяся к промежуточному звену между компиляцией и интерпретацией. В отличие от статической компиляции, когда исходный программный код переводится в машинный код перед запуском, и итерпретации, когда исходный код выполняется построчно, при JIT-компиляции исходный код компилируется и выполняется

частями, когда это необходимо и целесообразно для конкретной архитектуры host-среды;

– среда выполнения (или host-среда) - это вычислительное окружение, необходимое для выполнения JavaScript-программы. В качестве host-среды могут выступать:

1) клиентские платформы (браузеры) на основе Chromium: Google Chrome, YandexBrowser, Microsoft Edge, Opera, Vivaldi: а также Mozilla Firefox, Safari и другие;

2) серверные платформы: Node.js, Deno, Bun.js и другие;

3) платформы для программирования микроконтроллеров: Espruino, Tessel, Johnny-Five и другие.

В настоящей работе речь будет идти о Javascript-приложении, работающем в среде клиента (браузера), к которой относятся браузеры с различающимися JavaScript-движками: Blink/v8 (браузеры на основе Chromium), SpiderMonkey (Firefox и его форк Waterfox), JavaScriptCore/WebKit (Safari/Vivaldi) [6]. Совместимость последних версий браузеров и движков такова, что рассмотренные здесь примеры должны работать в современных версиях наиболее популярных браузеров. Для краткости совместимость можно продемонстрировать на основе поддержки технологии WebWorkers [11], так как обычно это соответствует поддержке остальных технологий, используемых в настоящей работе.

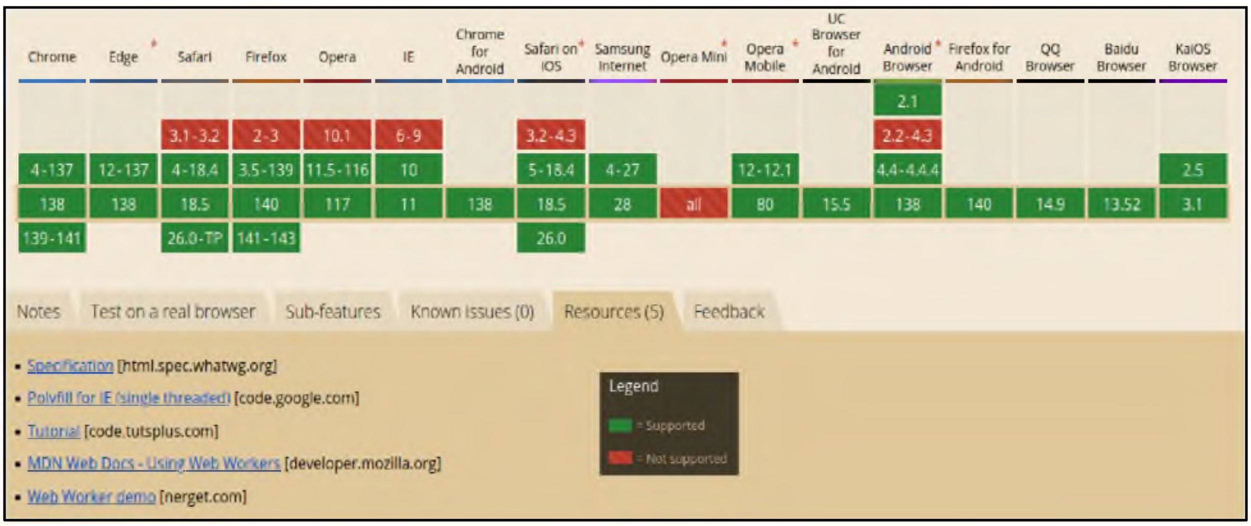

Рисунок 1.1 - Поддержка WebWorkers браузерами

Важно отметить следующее различие архитектур Blink и WebKit/GeCko: Blink отделяет рендеринговый движок от Javascript-движка (v8), тогда как WebKit и GeCko используют интегрированный подход, при котором рендеринг и Javascript (JavaScriptCore/SpiderMonkey) функционируют внутри движков [1, 2].

Язык ECMAScript и его Javascript-реализация используют средства браузера в качестве среды выполнения Javascript-кода (далее - JS-код), посредством API (ApplicationProgrammingInterface) через HTML5, в который встроен этот API [10]. Рассмотрим кратко и лишь на высоком уровне браузерную среду (окружение) применительно к нашей задаче, следуя [5, 8, 12].

Среда выполнения JS-кода (часто называют runtime) включает в себя:

– движок Javascript, в нашем случае: Blink/v8, Gecko/SpiderMonkey, WebKit/JavaScriptCore;
– цикл событий (EventLoop) с использованием стэка вызовов (CallStack);
– очереди задач Task Queue (macrotasks), Microtask Queue;
– WebAPI, предоставляемое JS-коду окружением, в нашем случае посредством HTML5;
– WebAPI для доступа к setTimeout, setInterval, fetch, DOM, console, localStorage, IndexedDB и другим не реализованы внутри JS-движка. Эти API, которые предоставляются средой браузера посредством HTML5.

Движок Javascript (JS-движок) занимается парсингом, компиляцией и выполнением кода. При парсинге код разбивается на токены и строится AST (AbstractSyntaxTree - синтаксическое дерево). При компиляции интерпретатор преобразует AST в байт-код, а JIT-компилятор оптимизирует “горячие” функции. Под “горячими” понимаются функции, которые либо вызываются часто, либо выполняются длительное время, либо обрабатывают большие объёмы данных.

Цикл событий (EventLoop) - это механизм, реализующий бесконечный цикл, в котором JS-движок ждёт задач, выполняет их, а затем снова ждёт новых задач. Этот механизм позволяет Javascript, несмотря на формальную одопоточность, выполнять асинхронные операции без блокировки основного потока за счёт использования стека вызовов (CallStack). Последний работает по принципу LIFO (последним пришёл, первым вышел). При этом поддерживается следующая последовательность выполнения работы:

– первым делом в CallStack помещаются синхронные вызовы (выполнение скрипта, реакция на события), а когда встречается асинхронная операция (например, setTimeout, setInterval, fetch), она передаётся в WebAPI;
– после завершения асинхронной операции её колбэк помещается в соответствующую очередь задач: TasksQueue (macrotasks) для setTimeout, setInterval, fetch, событий DOM; MicrotaskQueue для Promise.then и микропроцессов (queueMicrotask);
– EventLoop проверяет: если CallStack пуст, он берёт задачи из очереди и помещает их в стек для выполнения;
– важно помнить, что все микрозадачи из MicrotaskQueue выполняются сразу после текущего вызова стека вызовов и до следующей макрозадачи, так как это влияет на порядок выполнения кода.

Очереди задач уже упоминались, но не лишне повторить с некоторыми подробностями:

– TasksQueue (macrotasks) - очередь “макрозадач”, в которую попадают setTimeout, setInterval, fetch, события DOM и др. При этом необходимо учитывать две важных подробности:

  1) рендеринг никогда не происходит, когда JS-движок выполняет задачу. Изменения в DOM происходят только после выполнени задачи;

  2) если задача занимает слишком много времени, браузер не может выполнять задачи, такие как обработка событий пользователя. Через некоторое время он поднимает предупреждение типа “Страница не отвечает”, предлагая завершить задачу закрытием страницы.
– MicrotaskQueue (microtasks) - очередь микрозадач, в которую попадают Promise.then/catch/finally, другие задачи, относящиеся к микрозадачам. Например, к таким относится специальная функция queueMicrotask(func),

управляющая очередями микрозадач func. Как уже упоминалось задачи из очереди микрозадач выполняются сразу после текущего стека вызовов, прежде, чем выполнять любые другие макрозадачи или рендеринг или рендеринг или что-либо ещё. Итак, ещё раз, - в нашем случае важно помнить (подробнее - здесь [12]):

1) между микрозадачами нет пользовательского интерфейса, по умолчанию они работают один сразу за другим;

2) если необходимо выполнять функцию асинхронно, то это можно сделать с помощью очереди микрозадач (с использованием queueMicrotask()). Это можно сделать с помощью callback-функций, переданных в промис, поскольку такие функции обрабатываются очередью микрозадач (MicrotaskQueue, microtasks), а не очередью макрозадач (TasksQueue, macrotasks).

1.3 Визуализация в HTML4 Tables и HTML5 Canvas

В исходном приложении для отслеживания пространственных изменений в виде динамичных цветовых карт моделируемой физической храктеристики используется DOM-элемент HTML4 Tables с большим количеством ячеек. Используя атрибут border-collaps можно строить поле для отображения без ограничивающих рамок с наиболее просто изменяемыми размерами ячеек вплоть до одного пикселя. Элементы двумерного массива удобно проецировать на ячейки таблицы. Основными этапами построения (рендеринга) таблицы являются:

1) построение дерева рендера, когда строится финальное дерево с учётом DOM и CSS;
2) компоновка, когда рассчитываются размеры и положение всех ячеек таблицы;
3) отрисовка - заливка цветами пикселей каждой ячейки таблицы.

В нашем случае построение дерева и компоновка происходит один раз при вводе исходных данных программы, а основные затраты на рендеринг приходятся на этап отрисовки. Несмотря на это затраты могут быть всё-же существенными, поскольку каждая ячейка таблицы является объектом DOM-модели, занимающим соотвествующее количество оперативной памяти, хотя все свойства этого объекта не используются. А использование достаточно больших таблиц с 1000 x 1000 ячейками на бюджетных компьютерах становится проблематичным из-за нехватки памяти.

В HTML5 введён элемент Canvas (холст) [10], который, как представляется, может послужить хорошей заменой табличному представлению цветовых карт. Реализация такой замены возможна за счёт использования соответствующего JS-API, введённого в ES6. Рассмотрим далее основные возможности использования отрисовки карт в канвас, следуя руководству [18].

Добавлением на HTML-страницу элемента <canvas></canvas> создаётся невидимая прямоугольная область (холст), по умолчанию имеющая ширину 300px и высоту 150px, но, конечно, можно задавать другие значения атрибутов, или использовать CSS. Вместо добавления вручную можно автоматизировать процесс построения холста, например, следующим образом:

```
canvas = document.createElement('canvas');
canvas.id = "canvas";
n = 800;
canvas.width = n;
canvas.height = n;
img = new Image();
img.src = "./Kerch pen.gif";
ctx = canvas.getContext('2d');
img.onload = function () {
    ctx.drawImage(img, 0, 0, n, n);
    ctx.strokeRect(0, 0, n, n);
    document.getElementById("grid").appendChild(canvas);
}
```

С помощью этого кода создаётся холст с id="canvas", как дочерний элемент div-элемента с id="grid". В холст после загрузки вставляется изображение (карта-подложка) и производится обрамление рамкой. Эти манипуляции с холстом осуществляются посредством объекта контекста (здесь - объект ctx).

В дальнейшем расцветка холста может быть выполнена, например, с использованием кода:

```
imageData = ctx.getImageData(0,0,canvas.width, canvas.height);
data = imageData.data;
for (let i = 1; i < n-1; i++) {
    for (let j = 1; j < n-1; j++) {
        c[i][j] = ... // физический элемент, рассчитываемый по заданному алгоритму
        rgbaValues = colour(c[i][j]); // функция, возвращающая четыре цвета (пишем
    сами)
        const px i = ((j) + (i) * n) * 4;
        data[px i] = rgbaValues.r;
        data[px i+1] = rgbaValues.g;
        data[px i+2] = rgbaValues.b;
    }
}
ctx.putImageData(imageData, 0, 0);
```

В приведённом примере кода массив объектов холста присваивается переменной `imageData` с использованием функции `getImageData()` объекта `ctx`. Цвета пикселей холста содержатся в массиве data. Каждый пиксель представлен в этом массиве четырьмя числами в диапазоне от 0 до 255. Четвёртое число - альфа-компонента, по умолчанию равная 255 (полная непрозрачность, так что в приведённом выше коде по существу использована цветовая модель rgba(). Обновление изображения на холсте осуществляется с помощью `putImageData()`, ещё одной функции контекста.

Из приведённого выше примера видно, что рендеринг с использованием канваса можно выполнять в цикле, без использования DOM. Такая возможность должна привести к ускорению визуализации цветовых карт и уменьшению занимаемой памяти. Конечно, в этом необходимо убедиться экспериментально.

1.4 Вычислительные возможности с WebWorkers

До 2015 года браузеры работали, поддерживая язык ECMAScript, рекомендуемый стандартом ECMA от 2009 года, для которого принято наименование “ECMAScript 5” или, для краткости, “ES5”. До этого момента для реализации длительных вычислений использовали таймеры браузера (setTimeout, setInterval), поскольку, как это понятно из предыдущего подраздела, EventLoop может быть заблокирован.

Но к 2015 году HTML5-спецификацией было предоставлено средство для доступа к отдельным потокам под наименованием WebWorkers (веб-воркеры) [10], а в следующих стандартах ECMAScript (ES6, ES7 и далее) и, соответственно, в современных браузерах, технологии для поддержки многопоточных вычислений на основе веб-воркеров постоянно совершенствовались.

Веб-воркеры позволяют выполнять JS-код в фоновом режиме, не блокируя основной поток пользовательского интерфейса, что повышает отзывчивость и производительность веб-приложений.

При использовании компьютера с многоядерным (многопоточным) процессором в асинхронном режиме с помощью Web Workers можно организовать параллельные вычисления.

В настоящей работе используются лишь DedicatedWorkers, что объясняется спецификой оптимизируемого приложения. В наиболее общем виде работу с DedicatedWorkers можно представить схемой [13]:

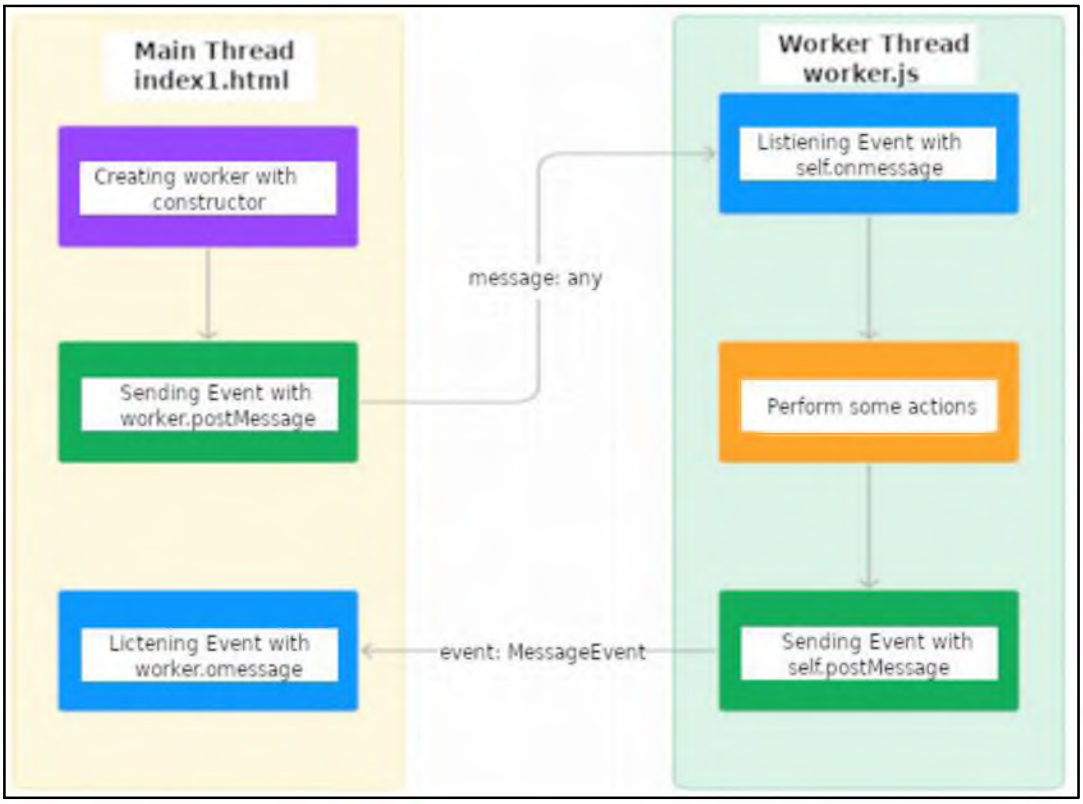

Рисунок 1.2 Схема взаимодействия основного потока cWebWorkers

Согласно спецификации ECMAScriptWebWorkersAPI обеспечивает использование воркеров трёх типов (рисунок 1.2) [22]:

– DedicatedWorkers (выделенные воркеры), привязанные к конкретной странице и недоступные из других;
– SharedWorkers (общие воркеры), доступные из нескольких страниц или вкладок одного сайта (домена);
– ServiceWorkers (служебные воркеры), работающие по типу прокси-сервера между браузером и сетью, перехватывая сетевые запросы для всех страниц, принадлежащих домену.

Используя схему, представленную выше, рабочий процесс с веб-воркерами можно изложить следующим образом:

1) инициализируем воркерс помощью конструктора;
2) вызываем postMessage из главного (родительского) потока;
3) прослушиваем поступление сообщения в воркере и посылаем его здесь же на обработку;
4) воркер выполняет логику нашего приложения;
5) воркер с помощью postMessage отправляет сообщение в главный поток;
6) прослушиваем сообщение воркера и обрабатываем сообщение в в главном потоке.

Таким образом можно создавать столько потоков, сколько позволят ресурсы компьютера. При этом каждый из потоков будет иметь свой контекст.

Инициализация объекта WebWorker сводится к использованию конструктора с передачей ему URL-адреса с указанием абсолютного или относительного пути к нашему файлу с JS-кодом для воркера:

```
if (window.Worker) {

     // newWorker (‘Путь до worker-файла, относиткльно текущего файла’)
     const worker = new Worker(‘worker.js’)

}
```

Рассмотрим далее основные методы для обмена данными между потоками:

1) postMessage(message: any, transfer: Transferabl[]): void - метод для отправки сообщения из одного потока в другой:

– message - любое значение или объект, который может быть обработан алгоритмом структурного клонирования в соответствии со спецификацией HTML5 [23]. Этот новый алгоритм сериализации более функционален, чем JSON. Он поддерживает сериализацию объектов, содержащих циклические графы, может клонировать: Blob, File, ImageData, Buffers. Не работает с: Error, Function и DOMElemets, однако имеет доступ к сетевым запросам и некоторым методам браузера и объекта Window [25], в частности - к таймерам и IndexedDB;
– transfer - массив объектов (ArrayBuffer | MessagePort | ImageBitmap), которые перенесутся в контекст воркера с использованием интерфейса Ttransferable и после этого не будут доступны в главном потоке. Это может пригодиться при копировании большого объёма данных, чтобы выиграть в быстродействии и экономии памяти. Интерфейс Ttransferable реализует несколько стандартных типов данных, в том числе ArrayBuffer, который, в свою очередь, представлен некоторым количеством типов массивов, среди которых имеются подходящие для нашего случая: IInt8, Uint32, Float32 [29].

Пример:

```
// Здесь мы передаем buffer в контекст worker,
//    в этом скрипте он больше недоступен
const buffer = new ArrayBuffer(42);
const data = { text: 'Hello, World!', buffer };
worker.postMessage(data, [buffer]);
```

2) onMessage: ((this: Worker, event: MessageEvent) => any | null): - это слушатель отправки message:

– event - объект события, полученный от worker/mainthread.

Пример:

```
interface MessageEvent <T = any> extends Event {
// Переданные данные
readonly lastEventId: string;
// Origin сообщения. Используется при работе с событиями, связанными
cross-document
```

```
//   messaging. Позволяет определить источник отправителя сообщения.
readonlyorigin: string;
// Порты, по сути - открытые нами страницы. Используются для обмена данными и
//   сообщениями между веб-воркерами и основными потоками
readonlyports: ReadonlyArray<MessagePort>;
// Предоставляет информацию об отправителе сообщения, например, какое окно
//   окно отправило сообщение
readonly source: MessageEventSource | null;
```

3) onError (callback(event) - при возникновении ошибки в webworker срабатывает это событие, содержащее поля:
   - filename - имя файла с кодом скрипта, сгенерировавшего ошибку;
   - lineno - номер строки кода, в которой произошла ошибка;
   - message - строка с описанием ошибки;
4) terminate - прекращение выполнения потоков и освобождение ресурсов, связанных с ним. Потоки WebWokers не могут самостоятельно прекратить работу. Остановить их может лишь страница или поток, которые их запустили, вызвав метод: worker.trminate(). Перезапустить поток можно лишь путём создания нового потока, использовав тот же URL.

Следует учитывать, что, если для передачи данных используется метод postMessage, сериализация/десериализация передаваемых данных может замедлять работу при передаче больших объёмов структурированных данных и вызывать чрезмерную загрузку оперативной памяти.

В нашем случае данные о цвете пикселей изображения (цветовой карты) будут пересылаться между потоками, имея значительный объём. Так при n=600 изображение из 360000 пикселей содержит 360×4 байт или 1,44 Мб RGBA-данных. При этом, для решения некоторых задач, например, фильтрации изображений [3, 10], на каждом цикле повторяется передача такого объёма пиксельных данных изображения из главного потока веб-воркерам, а после модификации возвращение такого же объёма из веб-воркеров в главный поток. Такая пересылка (по значению) затратна по времени и занимает память, поскольку одновременно хранится два экземпляра данных - в главном потоке и в воркерах.

В то же время, как уже упоминалось, технологией WebWorkers для ускорения работы и экономии памяти предусмотрена возможность обмена данными не по значению, а по ссылке через буфер обмена данных с использованием типизированных массивов ArrayBufer (Int8, Uint32, Float32 и т.д.) [21]. В этом случае используется интерфейс Transferable, работающий по концепции передачи владения данными (по ссылке), а не копирования (по значению) [24,29].

В нашем случае описанные выше затраты не должны быть такими большими даже при пересылке данных по значению, поскольку для алгоритмизации нашей задачи используются клеточные автоматы (КА). КА-алгоритм отличается локальностью, когда весь механизм изменения состояния клетки и её ближайшей окрестности зависит лишь от самой этой клетки и окрестности, и не зависит от прошлых состояний [14, 15]. Основные изменения клеточного поля будут происходить в воркерах с передачей в главному потоку. Задача последнего - отображение нового состояния с учётом этих изменений. Поэтому следует ожидать существенного ускорения, даже при использовании передачи данных от воркеров по значению. Будет ли ощущаться преимущество с использованием передачи данных по ссылке (через буфер) можно убедиться лишь экспериментально.

В заключение этого подраздела можно сформулировать следующее обобщение:

– веб-технологии интенсивно развиваются. Постоянно появляются новые возможности, которые делают ECMAScript/JavaScript всё более мощным. Начиная с 2015 года в HTML5-спецификацию вводятся веб-воркеры, которые можно использовать для параллельного выполнения кода;
– DedicatedWorker - разновидность веб-воркера, объект браузера, создающий отдельный изолированный контекст выполнения, который работает параллельно с основным потоком. Он позволяет распределять нагрузку между ядрами процессора и эффективно выполнять вычислительно сложные задачи в среде браузера;
– для передачи данных в воркерах используется postMessage, но обмен данных копированием значений (и сериализацией/десериализацей) может замедлять работу при передаче больших объёмов структурированных данных

и вызывать чрезмерную загрузку оперативной памяти, характерное для некоторых задач, когда данные пересылаются в двух направлениях: между основным потоком и воркерами. В нашем случае основные данные пересылаются в одном направлении от воркеров главному потоку, поэтому всё-таки можно ожидать ускорения;

– если ускорения не произойдёт, то можно опробовать более быстрый и экономичный способ передачи через буфер типизированных данных (ArrayBufer), используя интерфейс Transferable, работающий по концепции передачи владения данными (по ссылке), а не копирования .

2 Оптимизация браузерного приложения для имитационного моделирования

2.1 ES5-вариант приложения и возможности оптимизации с ES6

Для настоящего исследования разработан вариант экспериментального приложения на основе ES5 (далее - ES5-приложение) с архитектурой типа “монолит” (можно отнести к одностраничным), в котором в упрощённом виде используются основные алгоритмы, заложенные в исходное приложение имитационного моделирования [14]. Полностью код экспериментального приложения приведён в Приложении А1.

Рассмотрим концепцию клеточного автомата под кратким наименованиеем D2Q9 (два измерения девять соседей), на основе которой разработаны используемые здесь приложения, воспользовавшись красочным описанием из метеорологической экологии [9], когда под действием ветра распространяется облако консервативной взвешенной примеси из некоторого источника, например, от дымящей трубы, места постоянной разгрузки груза, порождающего облако пыли и т.п.

Процесс распространения облака инертной взвешенной примеси в атмосфере - это процесс рассеивания и смешивания его частиц с частицами воздуха, в котором преобладающую роль играет ветер. Под воздействием ветра частицы примеси попадают из источника в соседние клетки (окрестности). Затем из этих клеток, примыкающих к источнику, проникают в их окрестности, и так далее. Зоны различных концентраций примеси приобретают вид вложенных друг в друга эллипсоидов рассеивания, распространяющихся в направлениях, зависящих от характеристик преобладающих ветров.

Описанный выше процесс рассеивания примеси адекватно моделируется с использованием подхода, основанном на методологии клеточных автоматов [15]. Исследуемый участок представляется в виде области, состоящей из клеток. В области имеются клетки, в которых находятся источники примеси. Концетрацию примеси в клетках-источниках можно задать постоянной в произвольных единицах, например, (от 0 до 1000). Концентрацию примеси в каждой клетке области можно оценить по формуле:

$$c_{i,j} = \sum w_k \times b_k \,, \qquad (2.1)$$

где w_k - это массив из девяти элементов, содержащий веса ветров преобладающих направлений по восьми румбам и штиля, которые в сумме должны давать единицу; b_k - это вектор из девяти элементов, содержащий концентрации примеси в соседних с данной восьми клетках и в ней самой.

Шаг за шагом для всех клеток поля вычисляется сумма произведений концентраций в соседних восьми клетках на соответствующие вероятности ветров. К сумме прибавляется еще и произведение вероятности штиля на концентрацию примеси в самой текущей клетке.

Для нормальной работы программы необходимо иметь две копии клеточного поля. В одной из копий следует хранить предыдущее поколение, а в другой - последующее. При смене поколений содержимое последующего поколения становится предыдущим. Из расчетов исключаются крайние ряды клеток. Это делается для того, чтобы избежать "пограничных" проблем при применении формулы (2.1). Повторяя расчеты многократно, мы можем наблюдать динамику формирования эллипсоидов рассеивания во времени. При разовом вбросе примеси в какую-либо клетку-источник она быстро рассасывается по клеточному массиву. Для получения более выразительной картины рассеивания вброс примеси в клетки-источники осуществляется многократно в начале расчета очередного поколения.

Для восприятия результатов моделирования важна раскраска клеток. В описываемой экспериментальной версии программы использован алгоритм, переводящий значения концентраций примеси в клетках в различные градации яркости цветов клеток. Более высокие значения концентраций изображаются оттенками красного, который, по мере снижения концентраций, переходит в оранжевый, жёлтый и далее в соответствии с цветами радуги. Небольшие (1-2 единицы) концентрации обозначены светло-зелёным цветом. Концетрации менее единицы рассчитываются, но не видны (используется почти полная или полная прозрачность клетки (rgba()).

Изложенный алгоритм в упрощённом виде соответствует алгоритму исходного приложения по расчёту загрязнения и реализован в виде, удобном для экспериментов. В экспериментальном приложении вначале на каждом цикле в три заданные клетки-источники заданное число циклов вбрасывается некоторое количество инертной неосаждающейся прмеси (взвеси). Вбрасываемая взвесь растекается по соседним клеткам в соответствии с выше описанным алгоритмом, который можно реализовать на Javascript следующим образом:

```
for (i = 2; i<=n-1; i++) {
    for ( j = 2; j<=n-1; j++) {
         c[i-1][j-1] = 0;
         k = 0;
         for (l = -1; l<=1; l++) {
                  for (m = -1; m<=1; m++) {
                           k = k + 1;
                           c[i-1][j-1] = c[i-1][j-1] + w[k-1] * b[i -1 + l][j -1 + m];
                  }
         }
         // Перерасцвечиваем клетку по новому значению концентрации
         if (n cycles%n render == 0) {
                  document.all.ins tbl.rows[i-1].cells[j-1].style.backgroundColor=
                           colour(b[i-1][j-1]);
         }
         // Останов при достижении заданной концентрации границы квадрата
         //       (для эксперимента)
         if (i == n-1 &&c[i-1][j-1] >= 2) {
                  clearInterval(var time);
                  break;
         }
    }
}
for (i = 1; i<=n; i++) {
    for (j = 1; j<=n; j=j++) {
          b[i-1][j-1] = c[i-1][j-1];
    }
}
```

Веса (w) здесь равны повторяемостям ветров, сумма которых равна единице. Приведённый выше код содержится в функции mixer(). Для организации циклов использован таймер:

```
var_time=setInterval('mixer(params_in)',0);
```

Расчёт концентрации производится в функции mixer() на каждом цикле, а цветовая карта выводится через регулируемый интервал, что даёт возможность сопоставить временные затраты на вычисления и рендеринг.

В соответствии с HTML4 и ES5, когда ещё не был введён HTML5 Canvas, в качестве “холста” для отображения цветовой карты использован html-элемент <table ... >, в котором количество строк (<tr ...>) и количество столбцов (<td ...>) равно n при высоте строки (ширине столбца) равной 1px. Раскраска каждого пикселя созданного таким образом поля размером n×n выполняется в цикле путём обхода каждой ячейки (cell) таблицы.

В качестве фонового, как и в остальных экспериментальных вариантах, используется файл Kerch-pen.gif с изображением карты Керченского п-ва .

а) б)

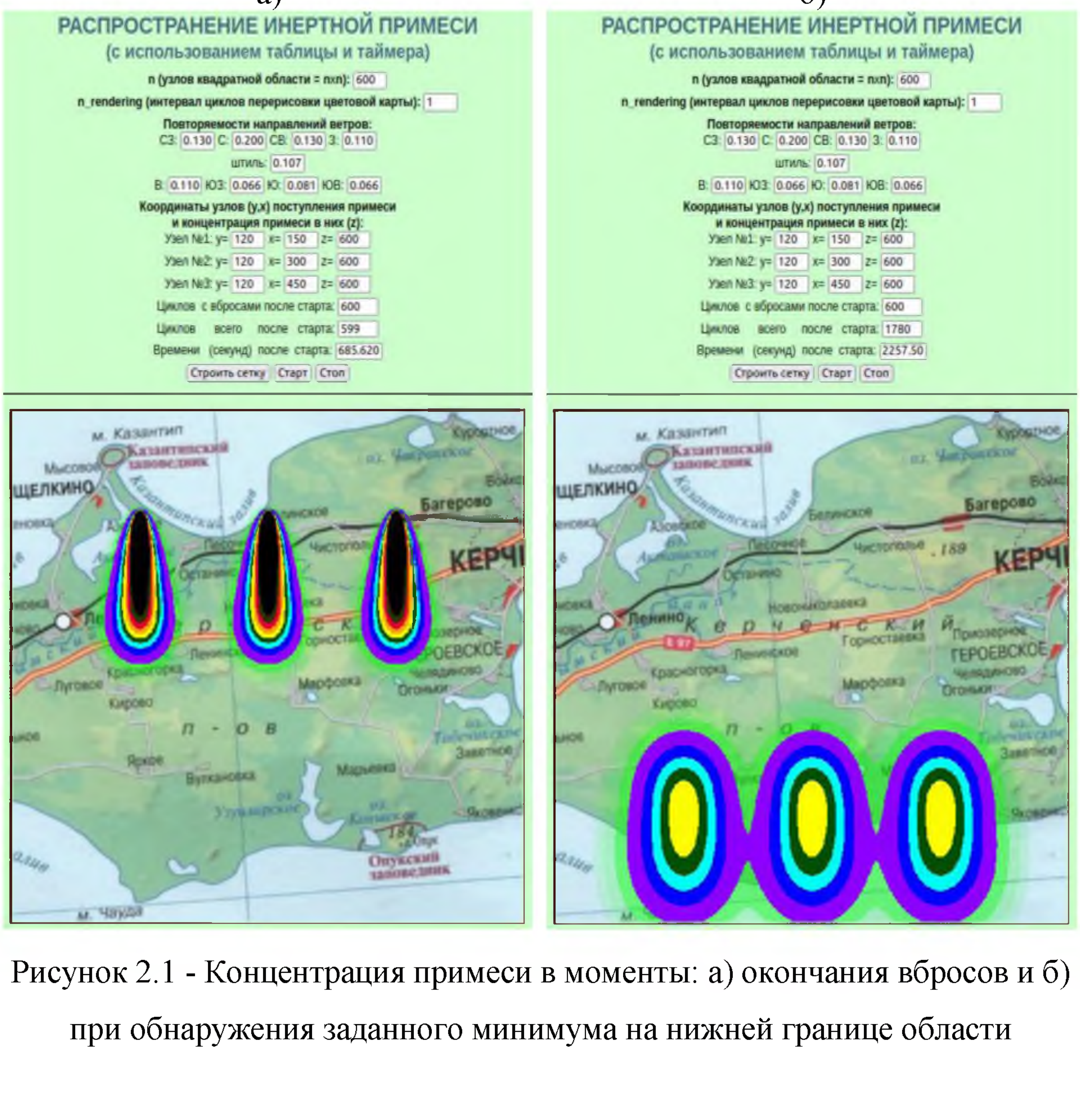

Рисунок 2.1 - Концентрация примеси в моменты: а) окончания вбросов и б) при обнаружения заданного минимума на нижней границе области

На рисунке 2.1 продемонстрирован результат работы экспериментального приложения при n = 600 в различные моменты времени.

С помощью этого приложения выполнена серия экспериментов, позволяющих оценить временные затраты ES5-варианта при различных значениях входных параметров:

– n - количество клеток квадратной клеточной области;
– n_rendering (далее для краткости - n_r) - интервал циклов перерисовки (рендеринга) цветовой карты. Смысл остальных входных параметров нетрудно понять из рисунка 2.1.

В экспериментах задавались следующие значения n: 100, 150, 175, 200, 300, 400, 500 и 600. Такие значения чаще всего соответствуют значениям при практических расчётах загрязнения. При значениях более 600, расчёты на используемых компьютерах часто заканчиваются аварийно из-за нехватки памяти.

При n_r, равном единице, цветовая карта выводится на каждом цикле после вычислений. Затраты времени в этом случае равны совместным затратам на рендеринг и вычисления. Если же n_r задать равным или превышающим максимальное количество циклов, то карта отобразится лишь один раз или не будет отображаться вообще. В этом случае будут измеряться лишь временные затраты на вычисления без затрат на рендеринг. Задавая некоторые промежуточные значения n_r можно уменьшать количество выводимых карт и определять снижение затрат на рендеринг.

При разных значениях n соблюдалось примерное соотношение объёмов вычислений и закраски путём соответствующего изменения значений вбрасываемой концентрации и циклов поступления примеси.

Для измерения затрат времени используется perfomance.now().

Эксперименты проводились на бюджетных компьютерах, примерно соответствующих тем, которые может себе позволить экологическая организация.

Для экспериментов использовались браузеры с движками Blink/v8 (на основе Chromium) и Gecko/SpiderMonkey (FF). Браузер Safari в настоящее время в вариантах для Windows и Linux не поддерживается, а компьютер Apple найти не удалось, поэтому эксперименты с Safari не проводились. Следует заметить, что продолжается поддержка WebKit [webki.org] - движка рендеринга от Apple, который включает в себя JavaScriptCore (WebKitJavaScriptCore). Однако такой вариант также отклонён, так как организация использования WebKitJavaScriptCore вызывает дополнительные накладные расходы, не позволяющие обеспечить сравнимость экспериментов.

Предварительные эксперименты показали, что все браузеры на основе Cromium (одинаковых движках) дают близкие результаты с несущественным преимуществом у Chrome. Результаты у Firefox существенно хуже. Поэтому далее имеет смысл продолжать эксперименты лишь с двумя браузерами, основанными на различных движках: Chrome и Firefox.

При выполнении экспериментов для их сравнимости обеспечивались одинаковые условия:

- экспериментальное приложение в процессе работы должно находиться в активном состоянии;
- блокировка экрана отключается, так как при блокировке частота процессора может уменьшаться;
- ноутбуки с регулируемым разгоном процессора работают от сети, так как при работе от аккамулятора частота процессора может меняться;
- все некритичные службы и приложения отключаются;
- эксперименты для каждого n по возможности выполняются неоднократно (до 10 раз).

Последнее условие вызвано тем, что каждый замер - это случайная величина: относительный разброс вокруг среднего до ±5% (по каждому n).

На рисунке 2.2 показаны графики, демонстрирующие динамику временных затрат при выполнении приложения в браузерах Chrome и Firefox.

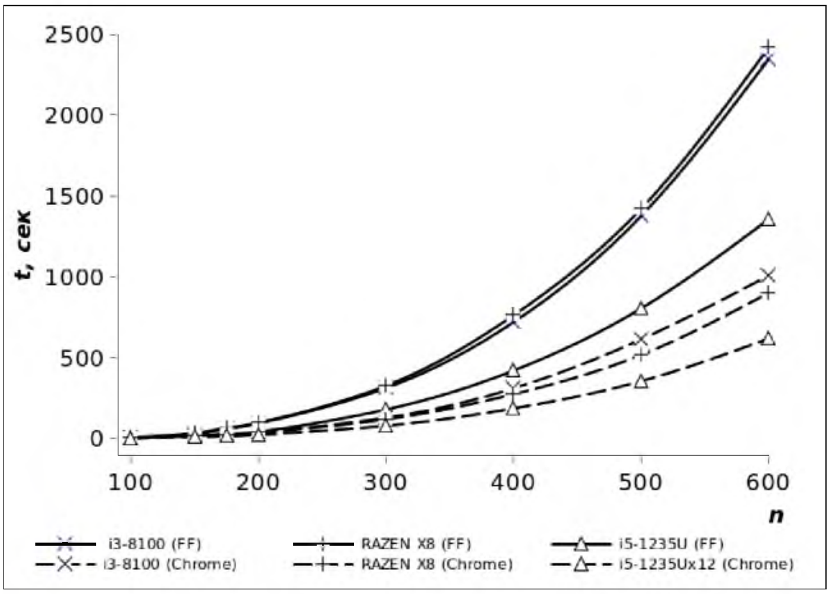

Рисунок 2.2 - Изменение времени выполнения в браузерах Firefox и Chrome с увеличением размеров клеточной области при использовании ES5-приложения

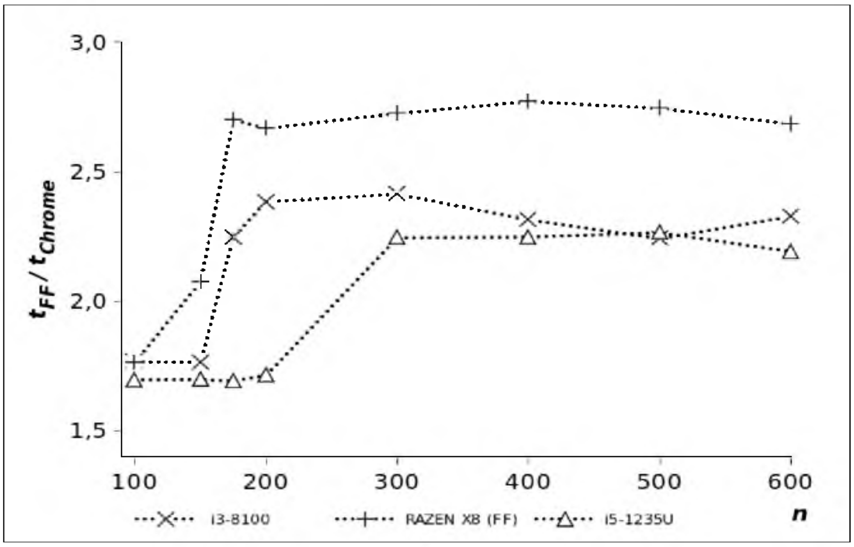

Рисунок 2.3 - Соотношение времён выполнения в браузерах Firefox и Chrome при увеличении размеров клеточной области

Рост временных затрат с увеличением n происходит по степенной функции, но темп роста существенно отличается для различных браузеров. Эти различия отчётливо можно видеть на графиках рисунка 2.3, где показано соотношение времён выполнения.

На этом рисунке на примере экспериментального приложения видно, что расчёты в Firefox выполняется существенно медленнее, чем в Chrome. Для n = 300 и более выполнение в Chrome осуществляется быстрее: в 2,7 раза быстрее на компьютерах с процессором AMDRYZEN и в 2,2-2,3 раза в компьютерах с процессорами Intel (i3-i5).

По результатам экспериментов можно оценить вклад затрат на рендеринг (t_{rend}) в общие временные затраты (t_{all}) на расчёты и рендеринг, вычислив значения t_{all}/t_{rend}. Усреднённые доли вкладов рендеринга в общие затраты составляют 0,92-0,95 для FF и 0,75-0,79 для Chrome. Графики таких долей для различных n показаны на рисунке 2.4.

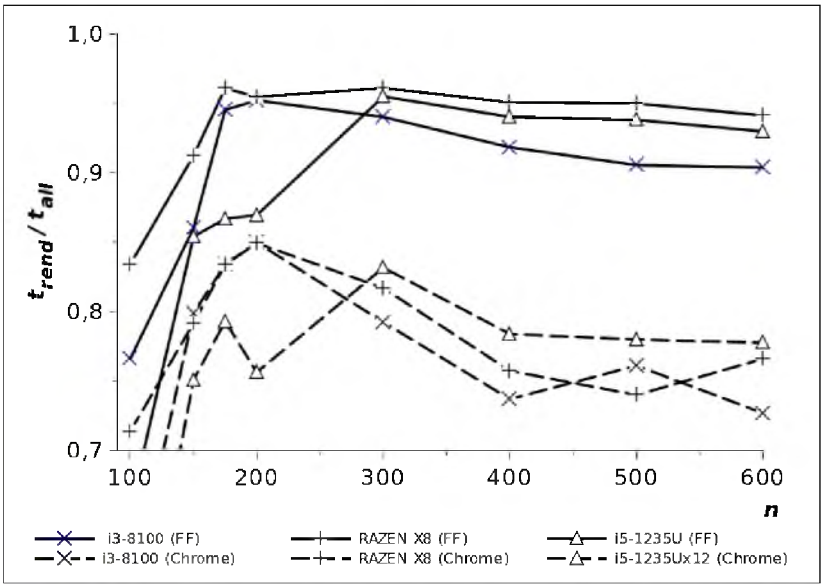

Рисунок 2.4 - Доля на рендеринг в браузерах Firefox и Chrome при увеличении размеров клеточной области

По результатам экспериментов можно также заметить различия в соотношении затрат браузеров на вычисления (t_{calc}) и рендеринг (t_{rend}): в Firefox превышение затрат по сравнению с Chrome происходит за счёт рендеринга. На графиках рисунка 2.5 показаны значения соотношений t_{calc}/t_{rend}.

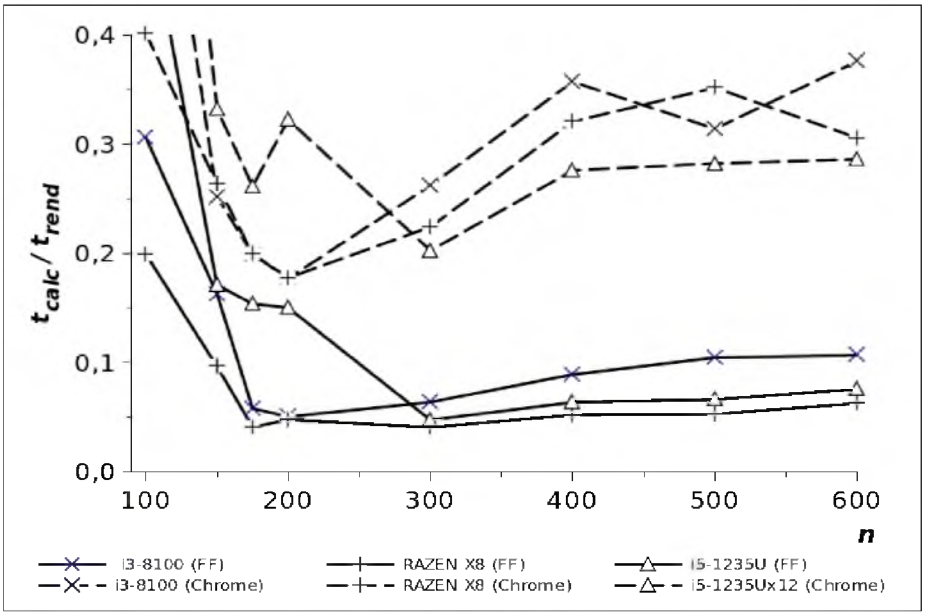

Рисунок 2.5 - Соотношение времён выполнения расчётов и рендеринга в браузерах Firefox и Chrome при увеличении размеров клеточной области

Из графиков на рисунке 2.5 следует, что затраты на рендеринг существенно превышают затраты на вычисления.

Однако следует учитывать, что такое соотношение затрат проявляется в случае обновления цветовых карт после каждого расчётного цикла и на практике такое отслеживание необходимо при чрезвычайно быстром изменении динамики клеточного поля моделируемого элемента.

Тогда как часто изменение состояния поля достаточно отслеживать через несколько интервалов циклов и тогда доля затрат на рендеринг существенно уменьшаются в чём можно убедиться по результатам экспериментов по графикам на рисунке 2.6.

Для краткости приведены лишь данные экспериментов, полученных на компьютере с i3-8100, но результаты для компьютеров с другими процессорами отличаются незначительно.

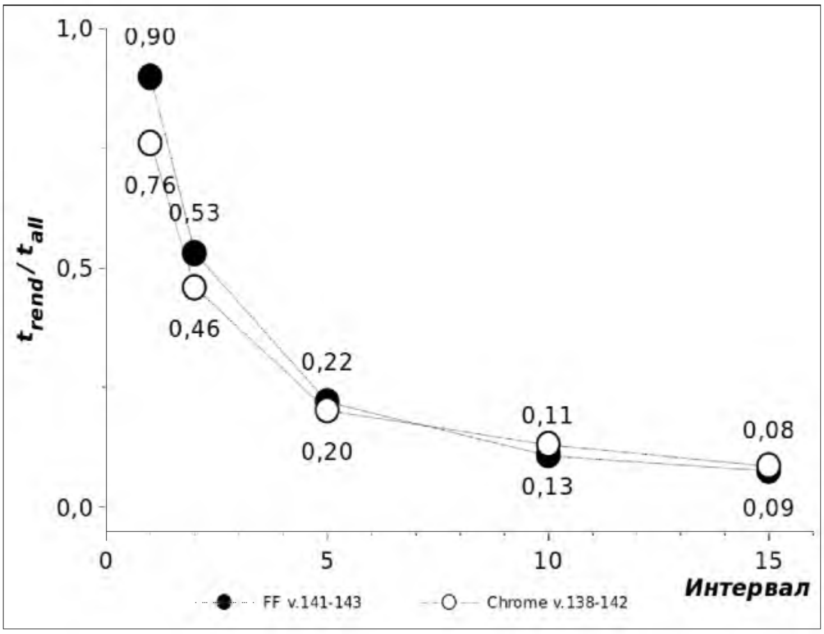

Рисунок 2.6 - Изменение долей затрат на рендеринг в браузерах Firefox и Chrome при изменении интервала обновления цветовой карты

Графики на рисунке 2.6 показывают существенное уменьшение затрат на рендеринг при увеличении интервала между обновлениями изображения цветовой карты.

Получены также соотношения затрат на расчёты и рендеринг в браузерах Firefox и Chrome при изменении интервала обновления цветовой карты.

По этим результатам построены графики, показанные на рисунке 2.7. Из этих графиков видно, что наибольшего эффекта можно достичь за счёт ускорения вычислений в Chrome.

В Firefox эффект за счёт ускорения вычислений следует ожидать менее значительным.

Различие в соотношении затрат на вычисления и рендеринг в разных браузерах можно объяснить различием подходов к рендерингу в движках Blink/v8 (Crome) и SpiderMonkey (FF) (подраздел 1.4).

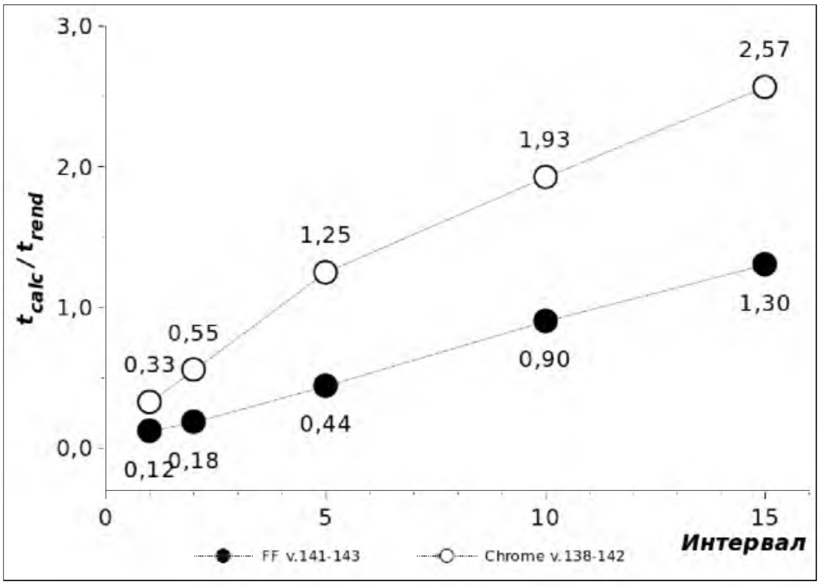

Рисунок 2.7 - Соотношение затрат на расчёты и рендеринг в браузерах Firefox и Chrome при изменении интервала обновления цветовой карты

Таким образом, по результатам экспериментов с ES5-вариантом приложения, описанным в настоящем подразделе, удалось уточнить основные задачи по оптимизации имитационного моделирования:

– сокращение затрат на рендеринг, в худшем случае (при постоянном обновлении цветовой карты) составляющих от 76 до 90 % от общих временных затрат, что можно попытаться осуществить за счёт использования DOM-элемента HTML5 Canvas и технологии WebGL;
– ускорение вычислений (без рендеринга) распараллеливанием с использованием технологии WebWorkers с рассмотрением двух вариантов обмена данными между главным и выделенными потоками: копированием (по значению) и через буфер (по ссылке);
– оптимизация рендеринга и вычислений за счёт совместного использования WebWorkers и WebGL.

Возможность решения перечисленных задач предстоит показать в следующих подразделах этого раздела путём проведения соответствующих экспериментов, показывающих получаемый эффект ускорения за счёт новых технологий при сравнении с ES5-вариантом. Для проведения таких экспериментов разработаны следующие варианты программ с последовательными изменениями ES5-варианта путём поэтапного добавления технологий ES6:

– вариант, в котором оставлено использование таймера, но вместо html-элемента table для рендеринга цветовых карт используется HTML5 Canvas (исходный код - в Приложении А2);
– варианты, в котором вместо таймера используются WebWorkers с двумя версиями обмена данными между главным и выделенными потоками: прямым копированием значений и через буфер по ссылке (исходный код - в приложении А3);

Следует учесть, что при запуске приложений непосредственно в браузере на локальном компьютере (без сервера) при работе с WebWorkers обращение к локальным файлам путём указания URL может привести к CORS-ошибкам [MDN - Cross-OriginResourceSharing (CORS)] из-за запрета такого обращения не из одного и того же домена. Поэтому следует работать в рамках домена, а для этого браузерные приложения с использованием воркеров должны запускаться на веб-сервере. Для экспериментов в настоящей работе будет использоваться веб-сервер Apache 2 с доменом localhost.

И ещё одно замечание: относительно единиц измерений.

Все выше представленные расчёты длительности (t_{calc}) проводились применительно к циклам (n_{cycles}) по всем клеткам клеточной области ($n \times n$). Величина t_{calc} - это длительность прохождения облака по области до момента прохождения облаком цикла n_{cycles}. Можно учесть особенность используемой модели клеточного автомата, в которой численные манипуляции с каждой клеткой области производятся единообразно. Отсюда - одинаковы временные затраты на вычисления, что легко проверить экспериментально. Поэтому можно рассчитывать среднее время, которое затрачивается на вычисления в одной клетке (t_{cl}) при различных n и n_{cycles}:

$$t_{cl} = (t_{calc} / n_{cycles}) / (n \times n) \, . \qquad (2.2)$$

То же самое относится к единицам измерения временных затрат на рендеринг и на сумму рендеринг+вычисления.

2.2 Визуализация цветовых карт в HTML5 Canvas

Здесь изложены результаты экспериментов подтверждающие рассмотренную в подразделе 1.3 целесообразность использования для рендеринга цветовых карт элемента HTML5 Canvas вместо элемента HTML4 Tables. Для выполнения экспериментов создано соответствующее приложение (полный код - в приложении А2).

Использование HTML5 Canvas реализовано в следующей функции:

```
// Определяем canvas в качестве сеточной области
function grider(n) {
    let el canvas = document.getElementById('canvas');
    if (el canvas) {
        el canvas.remove();
        for (let i = 0; i < n; i++) {
             for (let j = 0; j<n; j++) {
                 b[i][j] = 0;
                 c[i][j] = 0;
             }
        }
    }
    canvas = document.createElement('canvas');
    canvas.id = "canvas";
    canvas.width = n;
    canvas.height = n;
    img = new Image();
    img.src = "./Kerch pen.gif";
    ctx = canvas.getContext('2d');
   img.onload = function () {
        document.getElementById("grid").style.cssText ='width:'+n+'px;height:'+
                n+'px; background: URL(./Kerch pen.gif); background-size: auto '+n+
                'px;border-style:solid; border-width:1px; border-color:black;';
        imageData = ctx.getImageData(0,0,n,n);
        data = imageData.data;
        document.getElementById("grid").appendChild(canvas);
}
}
```

Здесь, как и в остальных экспериментальных вариантах, используется файл с изображением карты Керченского п-ва в качестве фонового.

Пример работы приложения при n=600 в браузере Firefox на компьютере с AMDRYZENX8 показан на рисунке 2.8.

а) б)

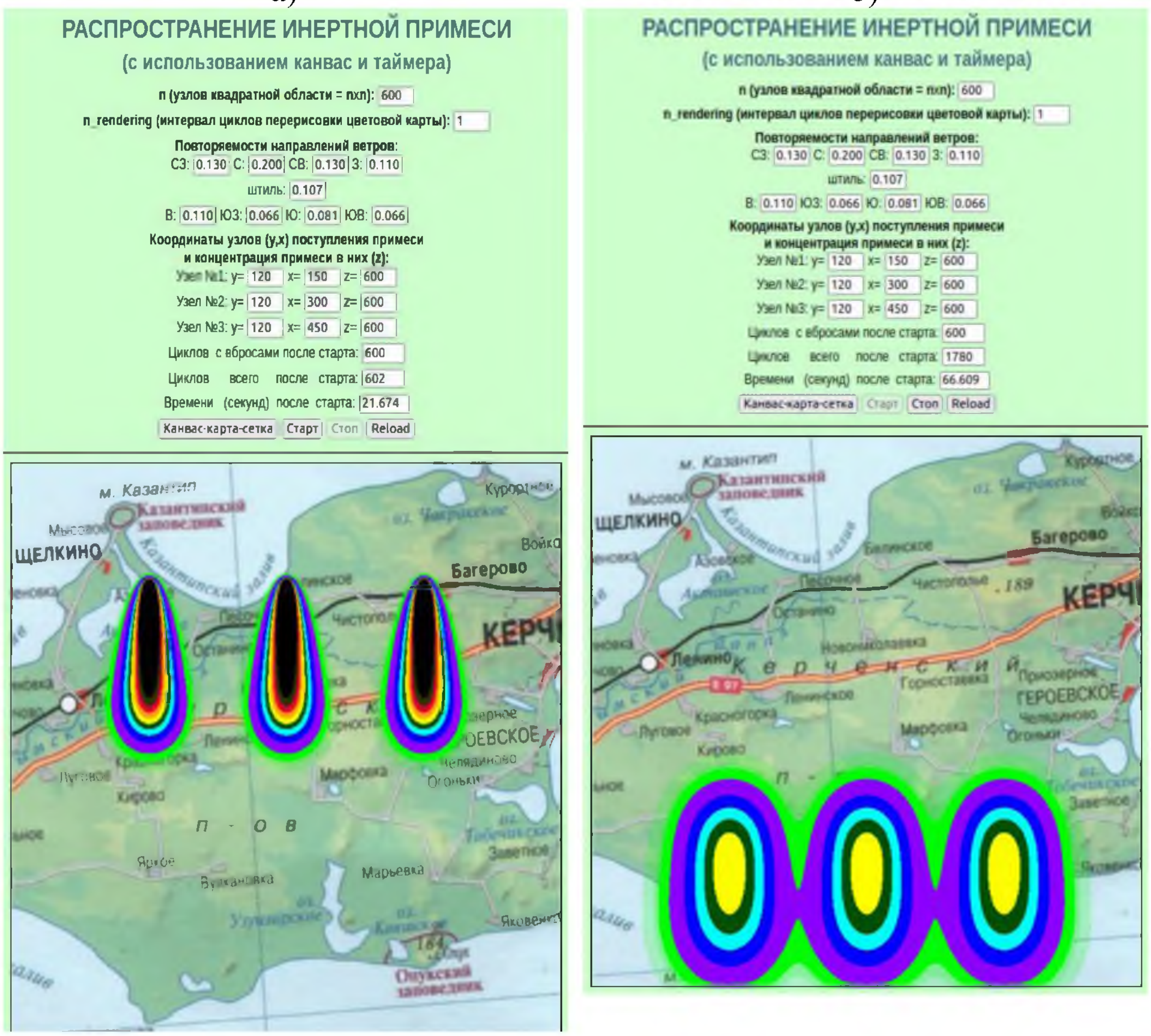

Рисунок 2.8 - Концентрация примеси в моменты: а) окончания вбросов; б) при обнаружения заданного минимума на нижней границе области

Можно сопоставить его с рисунком 2.1, демонстрирующем работу ES5-приложения при таком же n.

Не трудно заметить существенное ускорение за счёт рендеринга в HTML5 Canvas по сравнению с рендерингом в html-таблице.

По результатам таких сравнений построены графики, показанные на рисунках 2.9 ÷ 2.11.

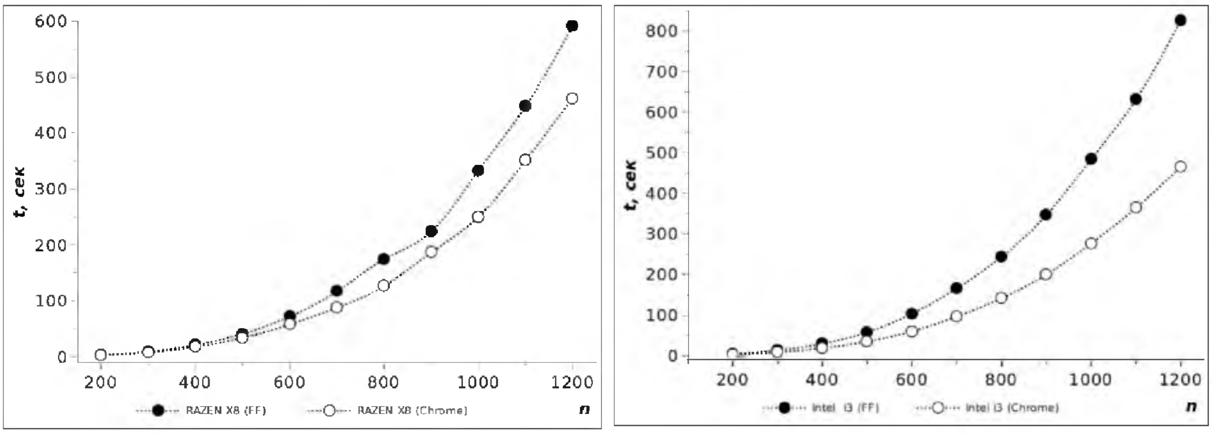

Рисунок 2.9 - Изменение времени моделирования в браузерах Firefox и Chrome с увеличением размеров клеточной области при использовании приложения с HTML5 Canvas вместо HTML4 Tables

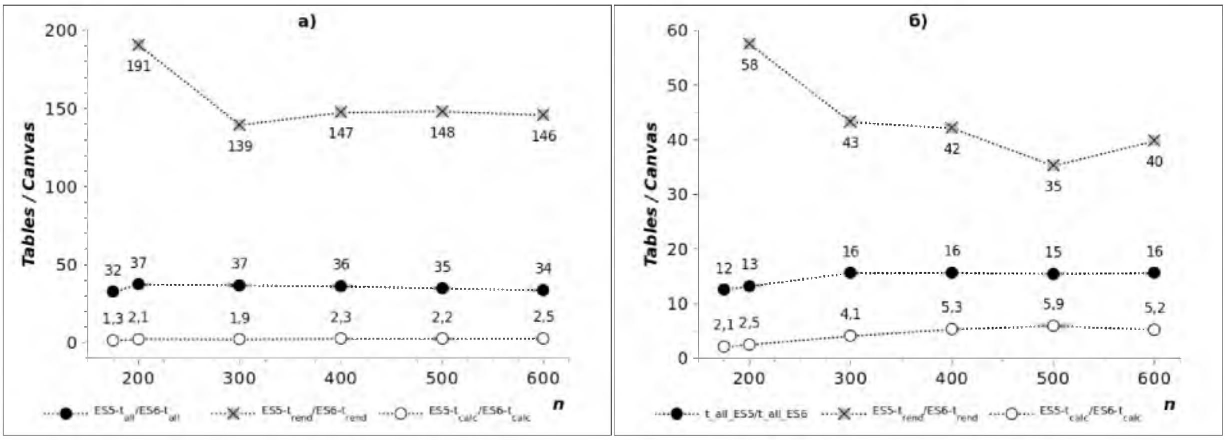

Рисунок 2.10 - Ускорение моделирования при замене HTML4 Tables на HTML5 Canvas в браузерах: а) Firefox и б) Chrome

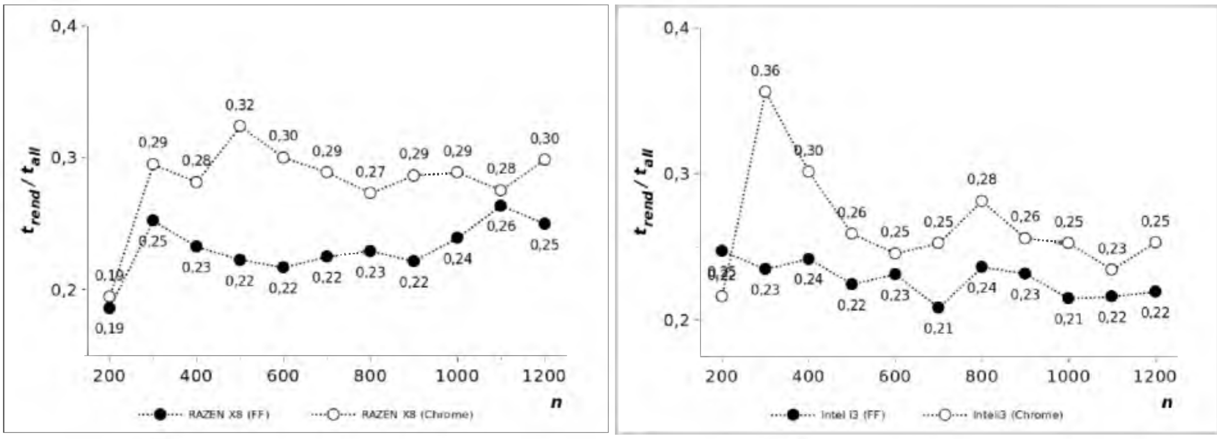

Рисунок 2.11 - Доля времени, затрачиваемой на рендеринг в браузерах Firefox и Chrome при увеличении размеров клеточной области

Сопоставляя графики на рисунках 2.9, 2.10, 2.11 с аналогичными графиками на рисунках 2.2, 2.3, 2.4, относящимся к использованию HTML4 Tables, можно убедиться в существенном уменьшении временных затрат на моделирование в целом и в том, что это происходит в основном за счёт рендеринга: около 43-х для Chrome и 147-ми для Firefox.

Количественные оценки ускорения моделирования (рендеринг+вычисления) за счёт замены HTML4 Tables на HTML5 Canvas составляют для компьютера с AMDRYZENX8: около 16-ти для Chrome и около 37-ти для Firefox. Для компьютера с Inteli3 ускорение равно: около 15-ти для Chrome и около 23-х для Firefox.

Соответствующее ускорение за счёт вычислений (без рендеринга) составляет достигает около 5,9 для Chrome и 2,5 для Firefox.

Примерно то же самое показывают эксперименты на компьютерах с другими процессорами: Inteli3, Inteli5.

И ещё один эффект, важный для настоящего исследования: моделирование при n = 600 не завершается аварийно, как это происходит в ES5-приложении (см. подраздел 2.1). Не происходит даже при n = 1200, что очень много для подобного приложения. Причина - более экономное расходование оперативной памяти элементом HTML5 Canvas по сравнению с HTML4 Tables, каждая ячейка которого - элемент DOM. В результате появляется возможность для увеличения масштабов расчётной области.

2.3 Распараллеливание вычислений с Web Workers

В отличие от предыдущего подраздела здесь исследуется возможность оптимизации работы приложения не за счёт рендеринга, а за счёт ускорения вычислений.

Из подраздела 1.2 следует, что такое ускорение можно попытаться осуществить путём распараллеливания за счёт разделения по выделенным потокам. При этом возможны два подварианта.

В первом из них, более затратном по времени и оперативной памяти, обмен данными между воркерами и главным потоком осуществляется непосредственной пересылкой этих данных копированием (по значению).

Второй, менее затратный подвариант, предполагает обмен данных через буфер (по ссылке) с использованием типизированных массивов ArrayBuffer.

Для экспериментов по этим двум подвариантам разработаны соответствующие экспериментальные ES6-приложения. В приложении А3 для краткости приводятся лишь коды приложения по второму подварианту.

Этого достаточно, так как подварианты различаются незначительно, а необходимые пояснения даются ниже по тексту.

Убедиться в почти двукратном ускорении вычислений при использовании обмена данными через буфер можно по рисункам 2.12 и 2.13, где показаны примеры работы приложений с использованием трёх веб-воркеров в браузере Firefox на компьютере с AMDRYZENA8.

Это, конечно, единичный случай, при усреднении неоднократных измерений результат может быть другим (ошибки измерений - около 5 %).

На рисунках в форме можно видеть, что дополнительно к предыдущим приложениям с таймером (рисунки 2.1 и 2.8) введены поле для выбора количества воркеров и флаг для включения/отключения вывода информации о работе воркеров.

Эта информация полезна при отладке программы и для ослеживания происходящего в воркерах.

В экспериментах флаг убирался, чтобы вывод информации не искажал результаты, увеличивая время выполнения. Указывая различные значение для поля «n_rendering» (далее для краткости - n_r) также, как и в предыдущем подразделе, можно проводить эксперименты с рендерингом или без него, только с вычислениями.

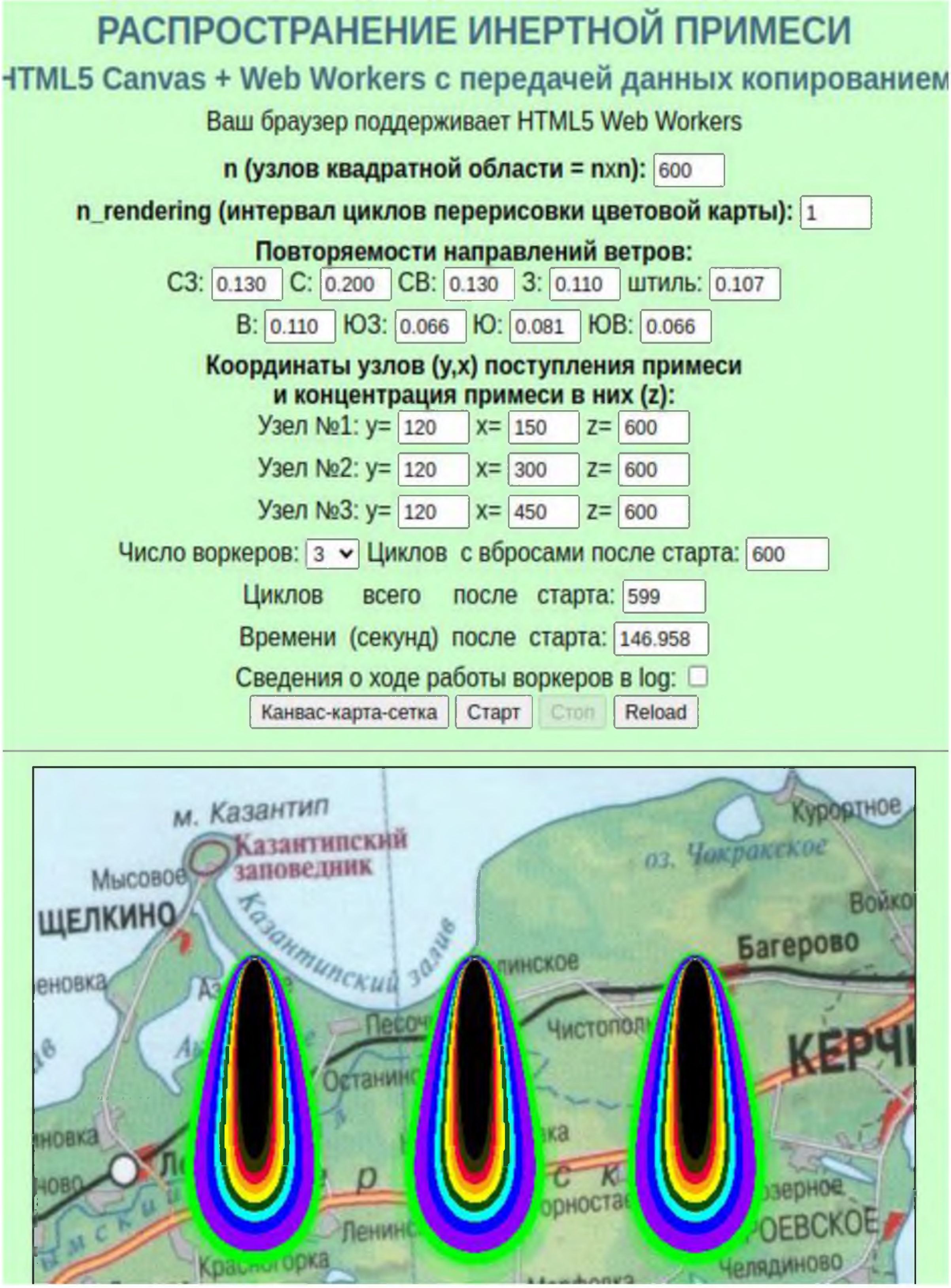

Рисунок 2.12 - Распараллеливание вычислений с обменом данными между потоками по значению (копированием). Концентрация примеси в момент окончания вбросов

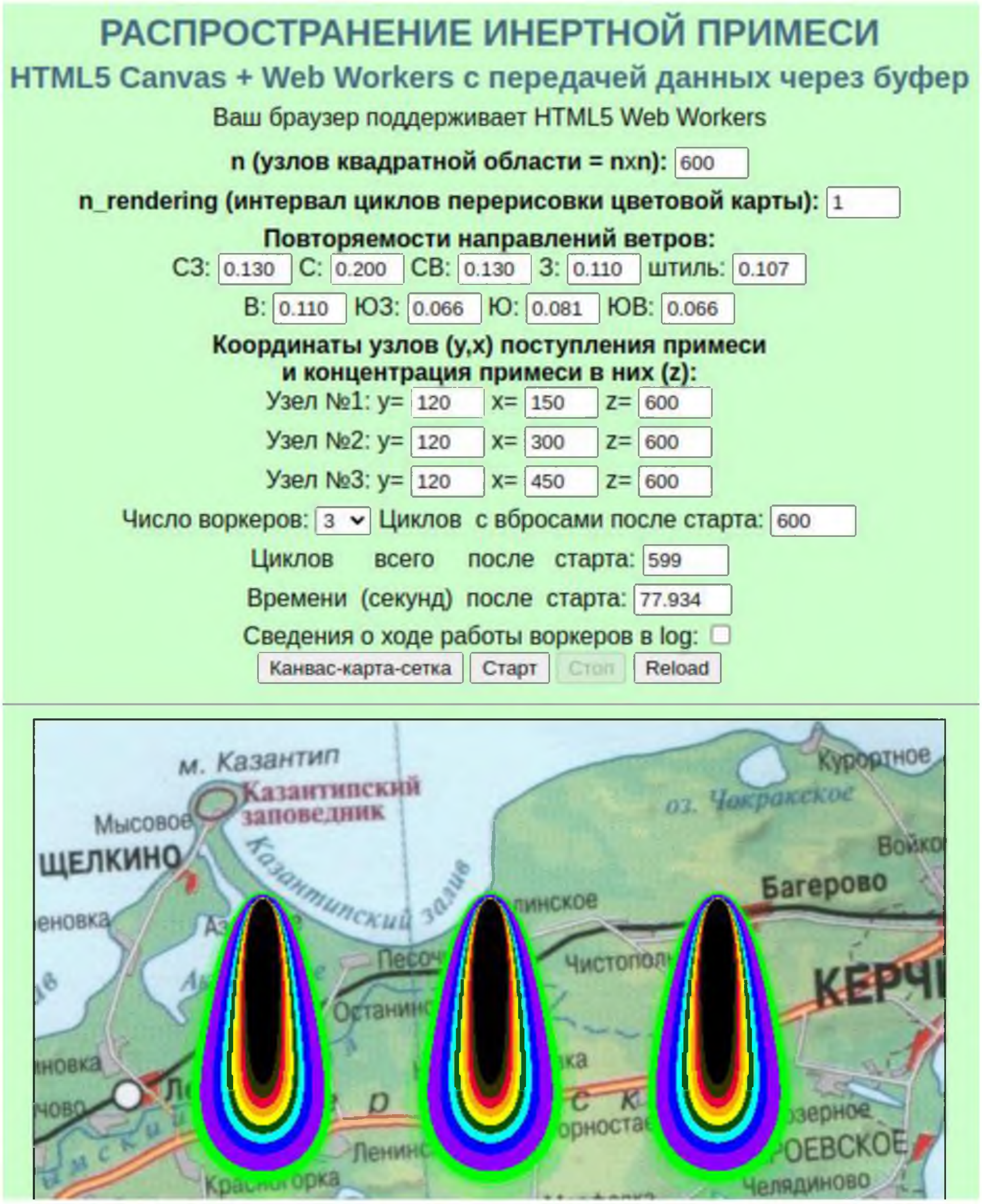

Рисунок 2.13 - Распараллеливание вычислений с обменом данными между потоками по ссылке (через буфер). Концентрация примеси в момент окончания вбросов

Аналогично приложению из предыдущего подраздела создаётся квадратный элемент HTNL5 Canvas - область для отображения перемещения облака концентрации в виде цветовой карты, с незначительным отличием: на этот раз подложка, графическое изображение карты, помещается как фоновое в div, родительский элемент канваса, что позволяет снизить затраты на перерисовку данных, получаемых из воркеров (см. файл index.html).

```
// Определяем canvas в качестве сеточной области
function grider(n) {
    let el canvas = document.getElementById('canvas');
    if (el canvas) {
        el canvas.remove();
    }
    canvas = document.createElement('canvas');
    canvas.id = "canvas";
    canvas.width = n;
    canvas.height = n;

    img = new Image();
    img.src = "./Kerch pen.gif";
    ctx = canvas.getContext('2d');

    img.onload = function () {
        document.getElementById("grid").style.cssText ='width:'+n+'px;height:'+
                n+'px; background: URL(./Kerch pen.gif); background-size: auto '+n+
                'px;border-style:solid; border-width:1px; border-color:black;';
         imageData = ctx.getImageData(0,0,n,n);
         data = imageData.data;
         document.getElementById("grid").appendChild(canvas);
}
}
```

Создаётся требуемое количество веб-воркеров с необходимыми свойствами:

```
// Создаётся экземпляр веб-воркера
functioninitWorker(src) {
var worker = new Worker(src);
    worker.addEventListener("message", messageHandler, true);
    worker.addEventListener("error", errorHandler, true);
    returnworker;
}
...
   // Создаём веб-воркер с нужными свойствами, заносим в массив веб-воркеров.
   // Каждому веб-воркеру – одинаковое задание, и они работают параллельно,
   //  каждый в своей области
     for (let i worker=0; i worker<workerCount; i worker++) {
         let worker = initWorker("moveWorker.js");
         worker.index worker = i worker;
         worker.width = width;
         worker.c = c;
         worker.w = w;
         worker.dump1 = dump1;
         worker.dump2 = dump2;
         worker.dump3 = dump3;
         worker.n render = n render;
         worker.n sycles add = n sycles add;
         worker.n sycles = n sycles;
         worker.no yes log = no yes log;
         worker.n br = n br;
         workers[i worker] = worker;
         sendMoveTask(worker, i worker, width, c, w,dump1, dump2,dump3,
        n render, n sycles add, n sycles, n br, no yes log);
}
}
```

В каждом потоке состояние клеточного поля изменяется в соответствии с описанным выше КА-механизмом с помощью кода, помещённого в файле (модуле) move.js:

```
function in range(j, l j, r j) return (j >= l j && j <= r j);
function mixer(b,w,i,j) {
        let k = 0;      let val = 0;
        for (let l = -1; l<=1; l=l+1) {
            for (let m = -1; m<=1; m=m+1) {
                        k = k + 1;
                        val = val + w[k-1] * b[i + l][j + m];
        }
        }
        returnval;
}
/***** Перемещение с рассеиванием за заданное кол-во циклов   *****/
/***** Кол-во циклов задано таким же, как у варианта с таймером *****/
/***** Два варианта: 1) с рендерингом и 2) без рендеринга        *****/
function boxMove(i worker,width,startX,c,w,dump1,dump2,dump3,n render,
        n sycles add,n sycles,n br) {
        var n = Math.sqrt(c.length);
        vardata = {};
        // Инициализируем массив для клеточного автомата, если его ещё нет
        if (typeof b === 'undefined') {
                b=new Array (n);
                for (let i=0; i<n; i++) {b[i]=new Array(n)};
                for (let i = 0; i<n; i++) {
                        for (let j = 0; j<n; j++) {
                                b[i][j] = 0;
                        }
                }
        }
        // Ещё раз о вариантах эксперимента - их лишь два:
        //      1) с рендерингом на каждом цикле при n render==1 и
        //      2) без рендеринга при n render!=1 (карта - на последнем цикле)
        if (n render == 1) { // 1) вариант - клеточный автомат с рендерингом
        if (n sycles<= n sycles add) {//вброс взвеси заданное кол-во раз
                //Добавляем взвеси в точку если она попадает в интервал воркера
                b[dump1[0]][dump1[1]]=in range(dump1[1],startX,startX+width) ?
                    b[dump1[0]][dump1[1]] + dump1[2] : b[dump1[0]][dump1[1]];
                b[dump2[0]][dump2[1]]=in range(dump2[1],startX,startX+width) ?
                    b[dump2[0]][dump2[1]] + dump2[2] : b[dump2[0]][dump2[1]];
                b[dump3[0]][dump3[1]]=in range(dump3[1],startX,startX+width) ?
                    b[dump3[0]][dump3[1]] + dump3[2] : b[dump3[0]][dump3[1]];
            }
            for (let i = 1; i < n-1; i++) {
                for (let j = startX+1; j < startX+width-1; j++) {
                        const index = j + i * n;
                        c[index] = mixer(b,w,i,j);
                }
            }
            for (let i = 0; i < n; i++) {
                for (let j = startX; j < startX+width+1; j++) {
                        const index = j + i * n;
                        b[i][j] = c[index];
                }
            }
            n sycles = n sycles + 1;
            data = {c,n sycles};
            // Отправляем во внешний поток на рендеринг на каждом цикле
        returndata;
        } else { // 2) вариант - клеточный автомат без рендеринга
            ...
```

В коде, приведённом выше, новое состояние поля концентрации (c) вместе с новым значением номера цикла (n_sycles) отправляется основному потоку для рендеринга. Отправка, также как и получение данных воркером, осуществляется с помощью кода, содержащегося в файле moveWorker.js:

```
importScripts("move.js");
function sendStatus(statusText) {
    postMessage({"type" : "status", "statusText" : statusText});
}
function messageHandler(e) {
    var messageType = e.data.type;
    switch (messageType) {
        case ("move"):
            if (e.data.no yes log) {
                        sendStatus("- - -> Воркер №" + (e.data.index worker+1) +
                        " приступает к работе на интервале X: " +
                        e.data.startX + "-" + (e.data.startX+e.data.width) +
                        ", циклов: " + e.data.n sycles);
            }
            let n = Math.sqrt(e.data.c.length);

            const buffer = new ArrayBuffer(n*n*4);
            let c = new Float32Array(buffer);
            for (let i=0; i < e.data.c.length; i++) c[i]=e.data.c[i];

            var data =
                boxMove(e.data.index worker,e.data.width,e.data.startX,
                        c,e.data.w,e.data.dump1,e.data.dump2,e.data.dump3,

        e.data.n render,e.data.n sycles add,e.data.n sycles,e.data.n br
                );
            postMessage({
        "type" : "progress",
                "index worker" : e.data.index worker,
                "width" : e.data.width,
                "startX" : e.data.startX,
                "c" : data.c,
                "w" : e.data.w,
                "dump1" : e.data.dump1,
                "dump2" : e.data.dump2,
                "dump3" : e.data.dump3,
                "n render" : e.data.n render,
                "n sycles add" : e.data.n sycles add,
                "n sycles" : data.n sycles,
                "n br" : e.data.n br
            }, [buffer]);

            if (e.data.no yes log) {
                sendStatus("<=== Воркер №" + (e.data.index worker+1) +
                " отработал на интервале X: " +
                        e.data.startX + "-" + (e.data.width+e.data.startX) +
                ", циклов: " + e.data.n sycles  +" !!! Отладка !!! "
                 );
            }
            break;
        default: sendStatus("Worker получил сообщение: " + e.data);
}
}

addEventListener("message", messageHandler, true);
```

В приведённом коде функции messageHandler(e) полученный массив значений концентрации предварительно подвергается буферизации: помещается в типизированный массив вещественных чисел Float32, привязанный к буферу двоичных чисел соответствующего размера, и только после этого передаётся на обработку в функцию boxMove(). (см. выше код модуля move.js).

После обработки данные, в том числе и буферизированный масив концентрации отправляется в главный поток по ссылке через буфер, массив которого указывается вторым параметром в postMessage(). Если этого второго параметра не будет, то данные будут отправлены копированием.

Отправленный таким образом буферизированный массив исчезает из области видимости выделенного потока, проявляясь в контексте основного потока. Данные не дублируются.

Изменённое в воркере состояние участка клеточного поля (в виде изменившейся концентрации) отображается основным потоком, составляя вместе с другими воркерами единое изображение поля концентрации в виде цветовой карты (см. файл index.html):

```
function messageHandler(e) {
    let n = Math.sqrt(e.target.c.length);
     let messageType = e.data.type;
     switch (messageType) {
         case ("status"):
             log(e.data.statusText);
             break;
         case ("progress"):
            for (let i = 0; i<n; i++) {
                 for (let j = e.data.startX; j<e.data.startX+e.data.width; j++) {
                         const index = (j + i * n);
                         let val = e.data.c[index];
                         if (val === null || val< 0) {
                         log("Недопустимая концентрация в клетке: i = " + i +
                                 ", j = " + j);
                         return;
                         }
                         // Расцвечиваем клетку по значению концентрации из воркера
                         rgbaValues = colour(val);
                         const px i = (j + i * n) * 4;
                         data[px i] = rgbaValues.r;
                         data[px i+1] = rgbaValues.g;
                         data[px i+2] = rgbaValues.b;
                         data[px i+3] = rgbaValues.a;
                 }
            }
            ctx.putImageData(imageData, 0, 0);
            document.getElementById("n sycles").value = e.data.n sycles;
// Останов по достижению заданного кол-ва циклов
             // (для сравнимости экспериментов с таймером)
```

```
let workerCount =
        parseInt(document.getElementById('workerCount').value);
              if (e.data.n sycles == e.data.n br) {
        rab width = rab width + e.target.width;
        //если область по ширине точно разделена между воркерами
        if (rab width/e.target.width === workerCount) {
                stopMove();
        };
              } else if (e.target.n render == 1) {
        if (e.data.n sycles > e.data.n br+1) {
                stopMove();
        }
              };
            // Отправляем данные для следующего цикла расчёта концентрации
        sendMoveTask(
                e.target, e.target.index worker,e.target.width,
                e.data.c, e.target.w,
                e.target.dump1, e.target.dump2, e.target.dump3,
                e.target.n render,e.target.n sycles add,
                e.data.n sycles,e.target.n br,e.target.no yes log
        );
        break;
default:
break;
    }
}
```

В последних строках кода функция sendMoveTask() используется для отправки потокам данных и аргументов (инструкций), необходимых для обновления состояния клеточного поля (концентрации). Данные потокам отправляются с использованием метода postMessage() (см. файл index.html):

```
function sendMoveTask(worker, i worker, chunkWidth, cc, w, dump1,dump2,dump3,
   n render, n sycles add, n sycles, n br, no yes log) {
    let chunkStartX = i worker * chunkWidth;
    let chunkStartY = 0;
    let n = Math.sqrt(cc.length);
   const buffer = new ArrayBuffer(n*n*4);
   let c = new Float32Array(buffer);
   for (let i=0; i < c.length; i++) c[i]=cc[i];
    worker.postMessage({
        'type' : 'move',
        'index worker' : i worker,
        'width' : chunkWidth,
        'startX' : chunkStartX,
        'c' : c,
        'w' : w,
        'dump1' : dump1,
        'dump2' : dump2,
        'dump3' : dump3,
        'n render' : n render,
        'n sycles add' : n sycles add,
        'n sycles' : n sycles,
        'n br' : n br,
        'no yes log' : no yes log
}, [buffer]);
}
```

В коде этой функции, выполняемой в основном потоке, отправляемый воркеру массив концентрации предварительно загружается в буфер. Буфер указывается вторым параметром в функции postMessage(). Если этого параметра не будет, то данные будут отправлены копированием.

Отправляемый таким образом буферизированный массив исчезает из области видимости основного потока, проявляясь в контексте воркера и, таким образом, данные не дублируются. Похоже на то, что делается в выделенном потоке (см. выше).

Рассмотрим подробнее результаты экспериментов с использованием воркеров.

Ниже приводятся рисунки 2.14÷2.18 с графиками, дающими представление о характере ускорения за счёт веб-воркеров по сравнению с исходным ES5-приложением.

На графиках рисунка 2.14 и 2,15 приведены результаты, позволяющие оценить возможности ускорения вычислений за счёт использования различного количества воркеров в браузерах Chrome и Firefox на компьютере с AMDRYZENX8: на рисунке 2.14 - воркеры без буферизации, а на рисунке 2.15 - воркеры с буферизацией.

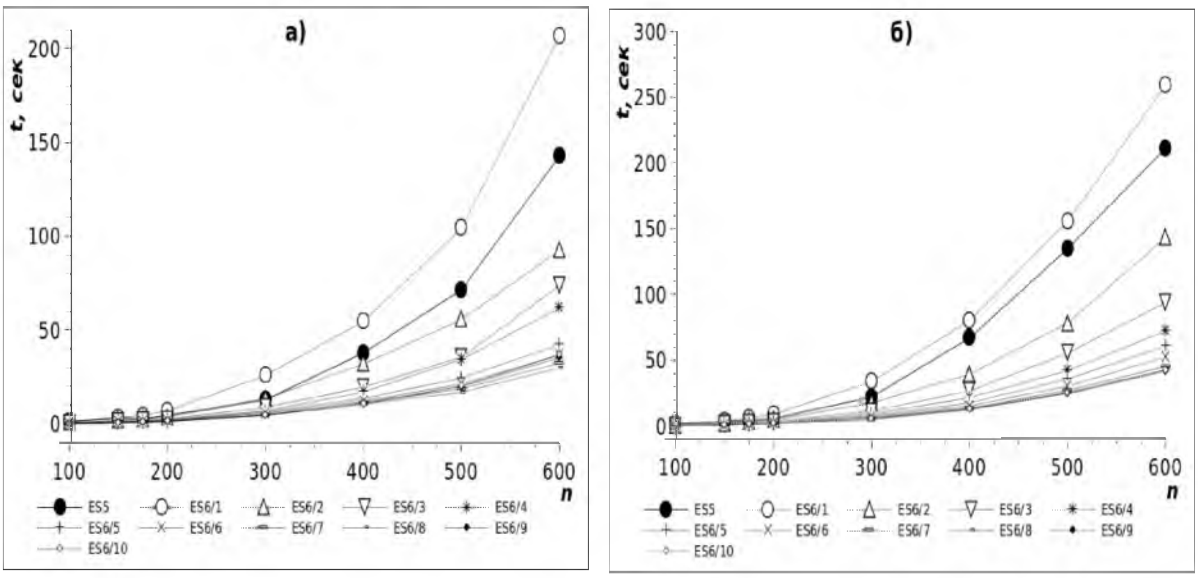

Рисунок 2.14 -Компьютер с AMDRYZENX8, веб-воркеры без буферизации. Длительности вычислений с использованием ES5-варианта приложения и ES6-варианта с различным количеством веб-воркеров в браузерах:

а) Firefox; б) Chrome.

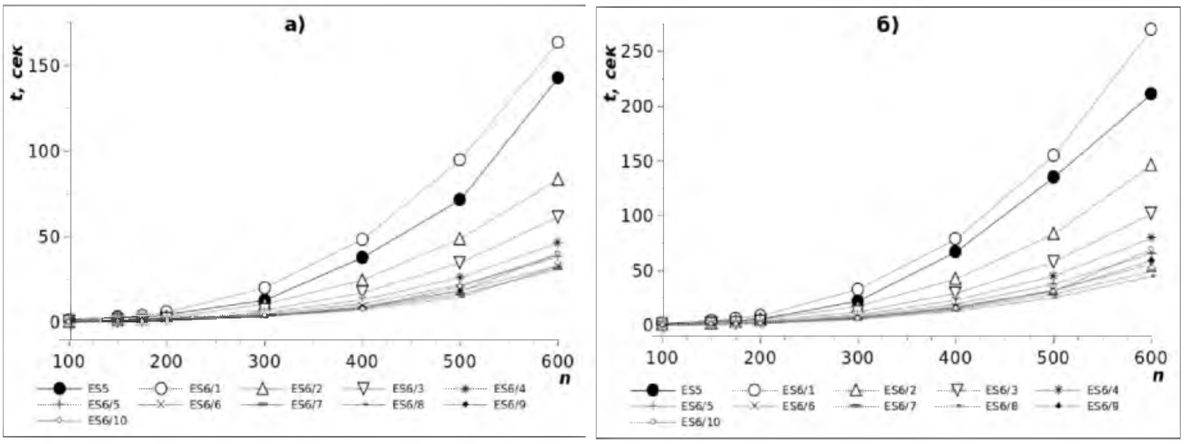

Рисунок 2.15 - Компьютер с AMDRYZENX8, веб-воркеры с буферизацией. Длительности вычислений с использованием ES5-варианта приложения и ES6-варианта с различным количеством веб-воркеров в браузерах: а) Firefox; б) Chrome.

Представленные графики демонстрируют уменьшение временных затрат на вычисления при использовании двух и более веб-воркеров. Использование одного веб-воркера, наоборот, приводит к увеличению затрат, что, как следует из подраздела 1.2, вызвано затратами на создание веб-воркера.

Для количественной оценки преимуществ использования воркеров рассчитаны относительные величины, представленные в виде графиков на рисунках 2.16 и 2.17.

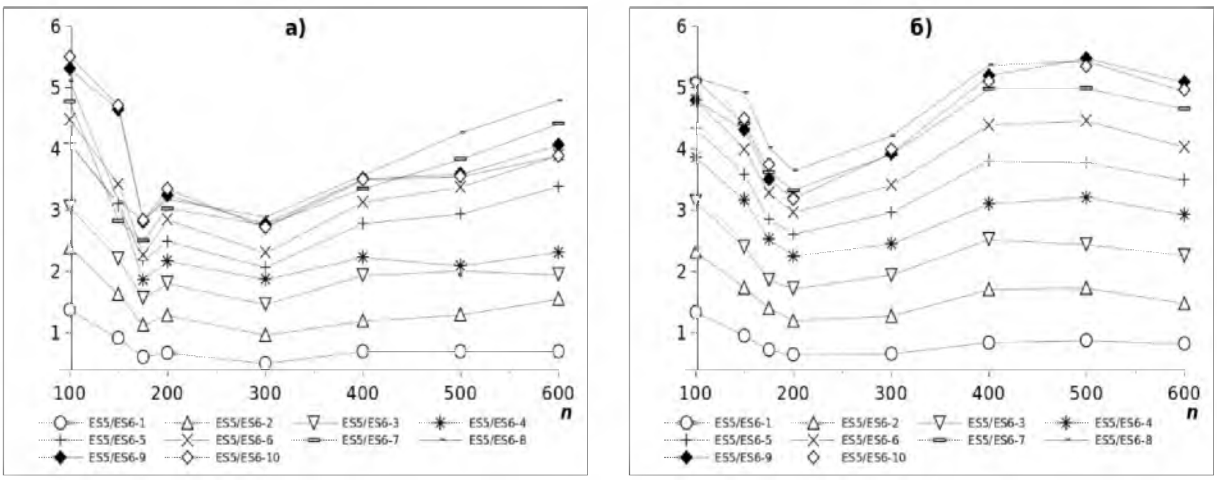

Рисунок 2.16 - Компьютер с AMDRYZENX8, веб-воркеры без буферизации Ускорение вычислений по мере добавления веб-воркеров в браузерах: а) Firefox; б) Chrome.

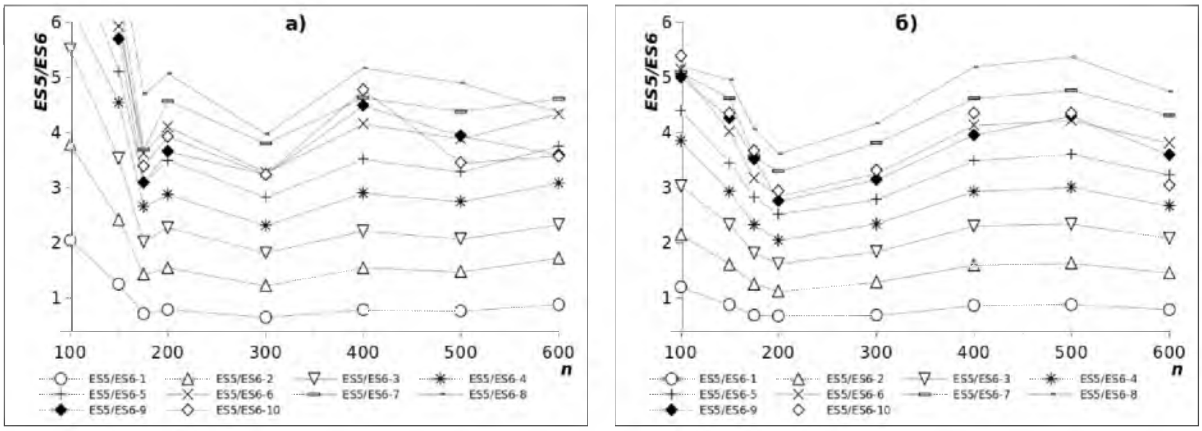

Рисунок 2.17 - Компьютер с AMDRYZENX8, веб-воркеры с буферизацией. Ускорение вычислений по мере добавления веб-воркеров в браузерах: а) Firefox; б) Chrome.

Графики демонстрируют изменчивость ускорения при использовании веб-воркеров для различных масштабов (n) по сравнению с ES5-вариантом, в котором используется элемент HTML4 Table с таймером. Преимущество распараллеливания по выделенным потокам проявляется при двух и более воркерах.

Примерно то же самое показывают эксперименты на компьютерах с другими процессорами: Inteli3, Inteli5.

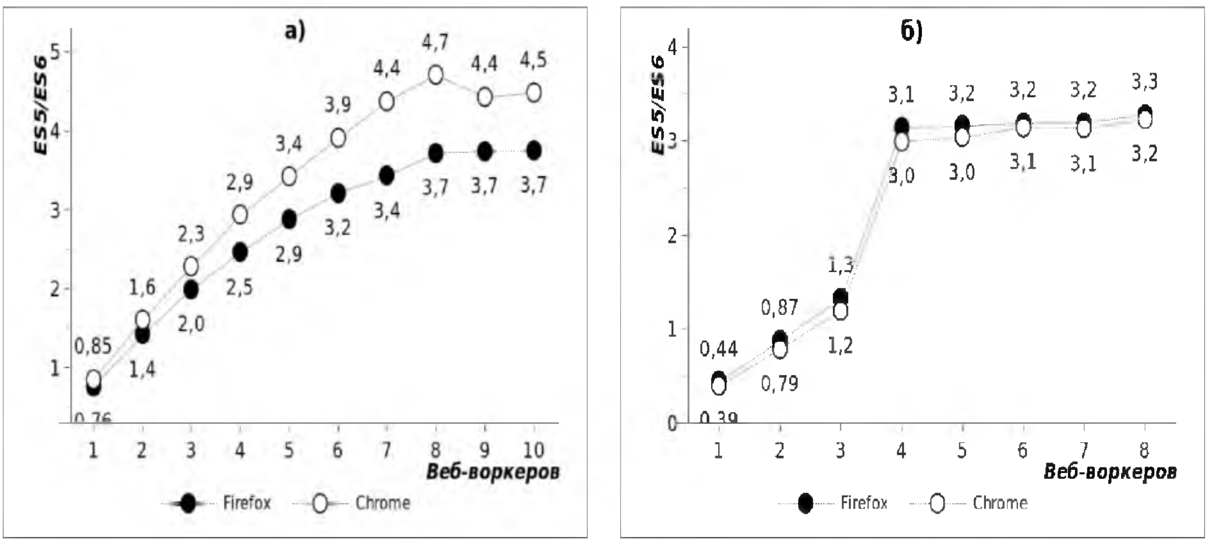

Рисунок 2.18 - Веб-воркеры без буферизации. Ускорение вычислений при добавлении веб-воркеров в компьютерах: а) с AMDRYZENX8; б) с Inteli3

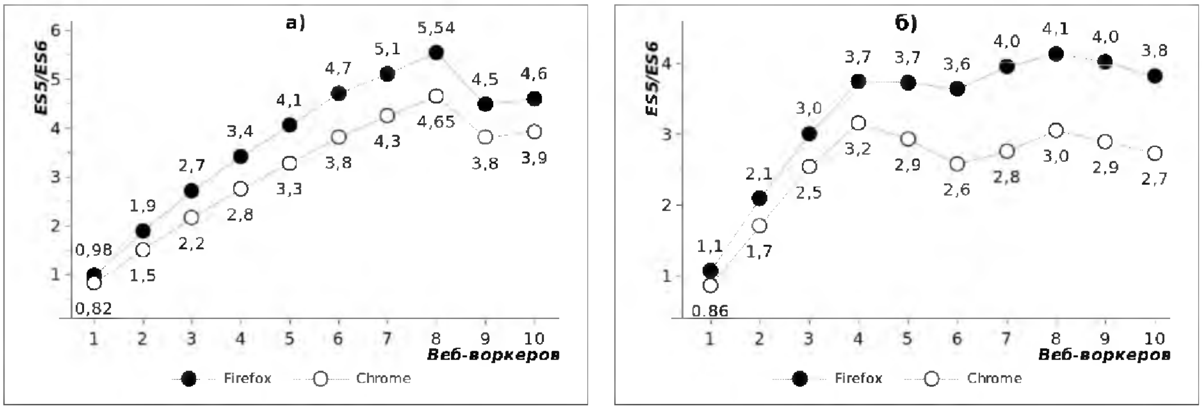

Рисунок 2.19 - Веб-воркеры с буферизацией. Ускорение вычислений при добавлении веб-воркеров в компьютерах: а) с AMDRYZENX8; б) с Inteli3

На рисунке 2.18 и 2.19 показаны графики изменения усреднённых по n отношений времён вычислений в ES5-приложении и рассматриваемом здесь ES6-приложении, показывающие особенности динамики ускорения при добавлении веб-воркеров.

На графиках 2.18, 2.19 можно видеть, что ускорения не происходит, если число воркеров превышает число ядер используемого процессора: AMDRYZENX8 - 8 ядер (двухпоточных), и Inteli3-8100 - 4 ядра (однопоточных).

Ниже, на рисунках 2.20÷2.23 представлены рисунки с графиками, демонстрирующие ускорение рендеринга и суммарное ускорение моделирования (рендеринг + вычисления).

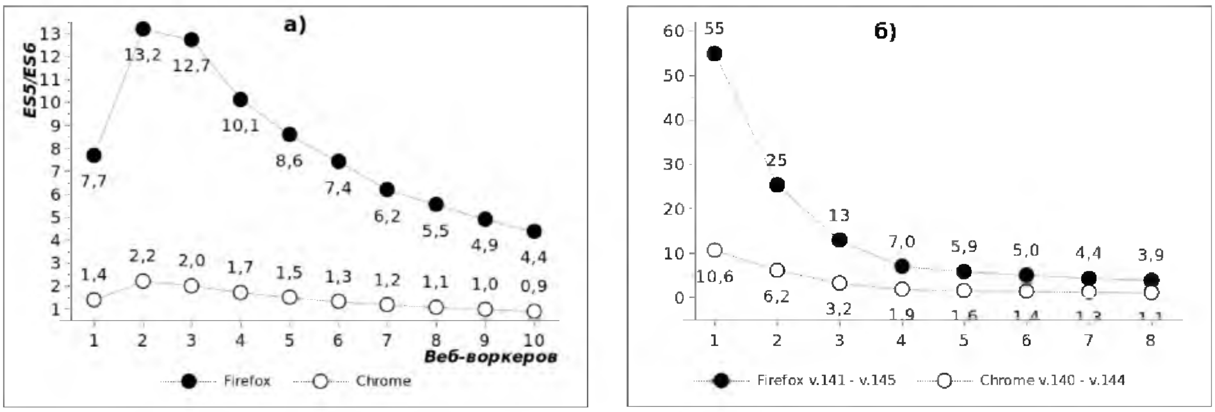

Рисунок 2.20 - Веб-воркеры без буферизации. Ускорение рендеринга при добавлении веб-воркеров в компьютерах:
а) с AMD RYZEN X8; б) с Intel i3

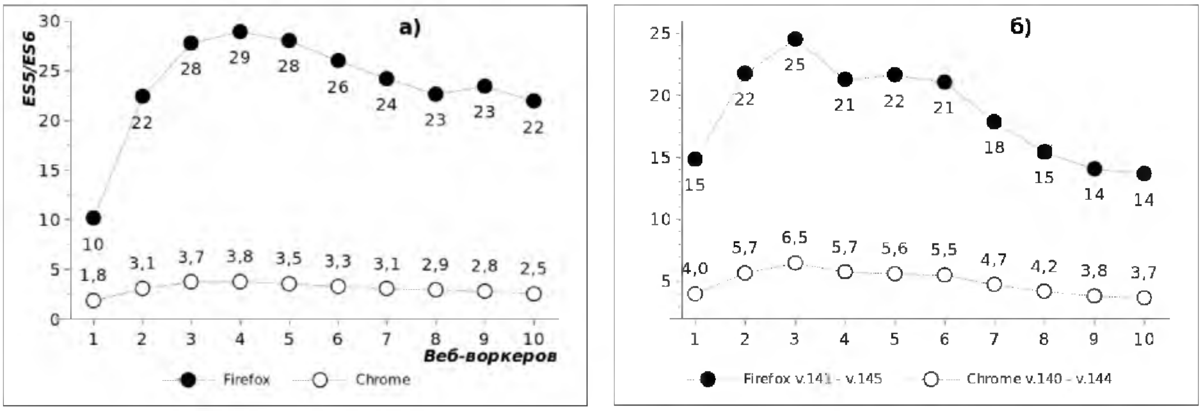

Рисунок 2.21 - Веб-воркеры с буферизацией. Ускорение рендеринга при добавлении веб-воркеров в компьютерах:
а) с AMD RYZEN X8; б) с Intel i3

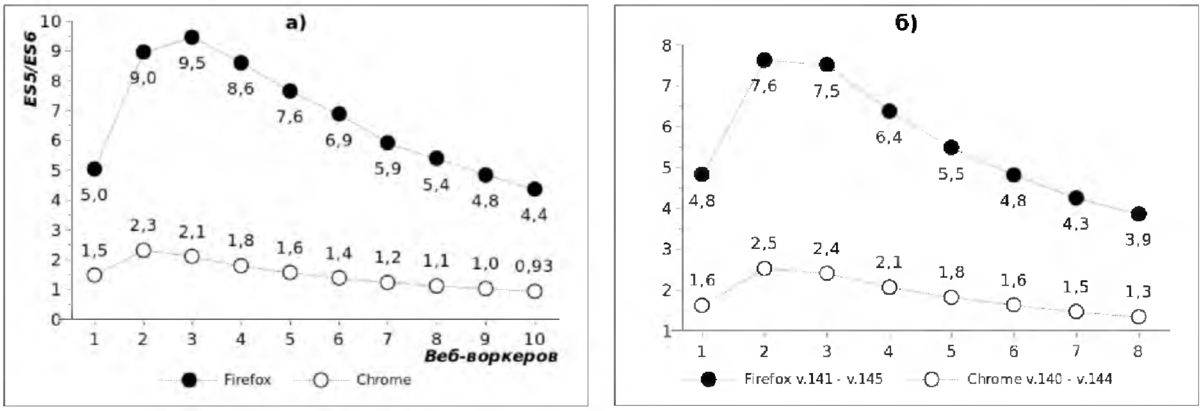

Рисунок 2.22 - Веб-воркеры без буферизации. Суммарное ускорение (рендеринг+вычисления) при добавлении веб-воркеров в компьютерах:
а) с AMD RYZEN X8; б) с Intel i3

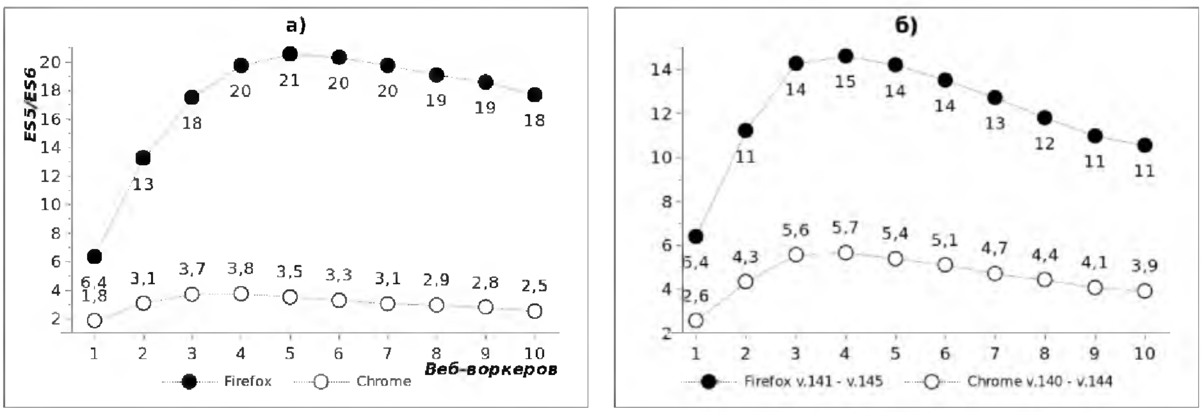

Рисунок 2.23 - Веб-воркеры с буферизацией. Суммарное ускорение (рендеринг+вычисления) при добавлении веб-воркеров в компьютерах:
а) с AMD RYZEN X8; б) с Intel i3

Далее попытаемся более аргументированно, чем только лишь по примеру с изображениями на рисунках 2.12, 2.13, убедиться в ускорении за счёт буферизации обмена данными в воркерах. Тем более, что результаты экспериментов имеют значительную долю случайной составляющей (до 5 %).

Ниже помещены рисунки 2.24÷2.25, построенные по результатам экспериментов. Графики рисунке 2.24 подтверждают особенность, отмеченную выше для случая передачи данных копированием: наиболее значительное уменьшение временных затрат при передаче данных по ссылке происходит при числе веб-воркеров, не превышающем количество ядер процессора используемого компьютера.

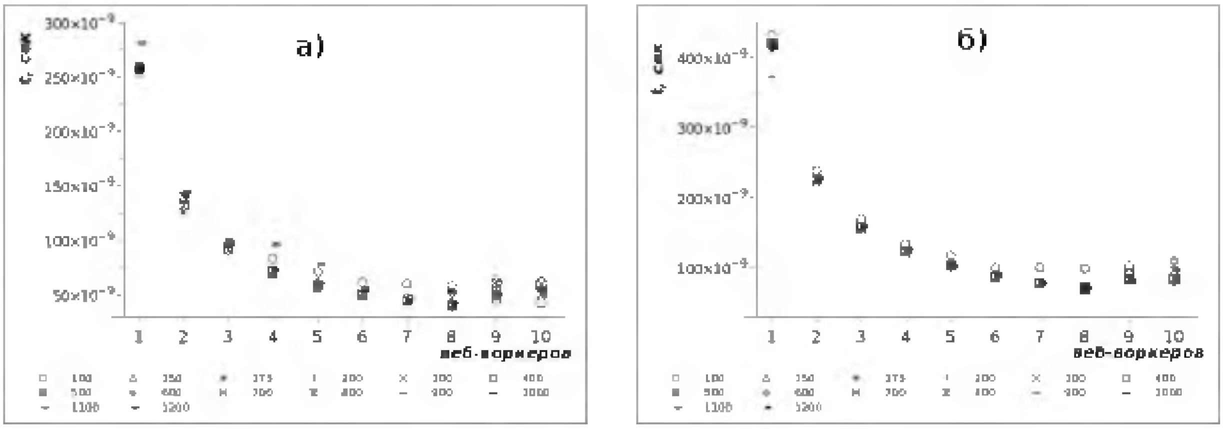

Рисунок 2.24 - Компьютер с AMDRYZENX8. Длительности вычислений при добавлении веб-воркеров с обменом данными по ссылке в браузерах: а) Firefox v.141-145; б) Chrome v.140-144

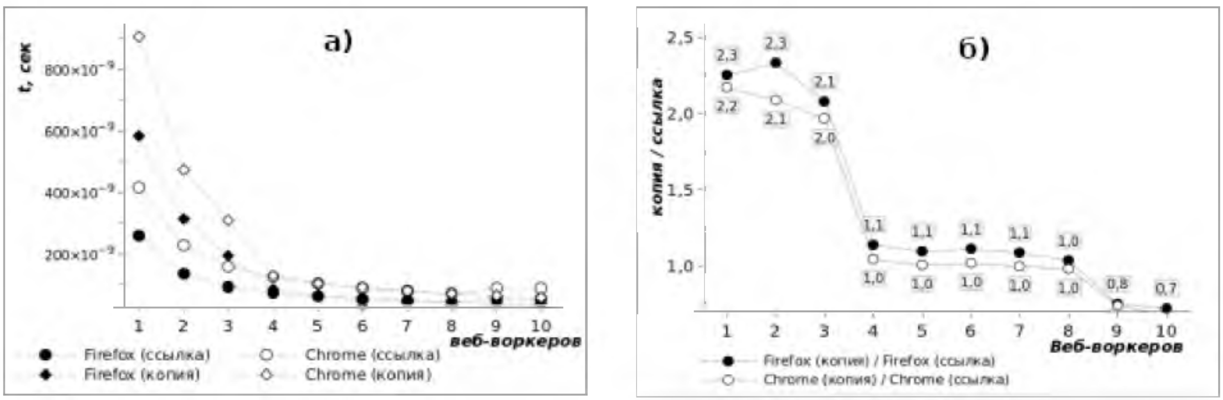

Рисунок 2.25 - Компьютер с AMDRYZENX8, браузеры Firefoxv.141-145 и Chromev.140-144: а) длительности вычислений в клетке при добавлении воркеров с обменом данными по значению и по ссылке; б) отношение длительности выполнения с воркерами с обменом данными по значению к длительности выполнения с воркерами с обменом данными по ссылке

Графики на рисунке 2.25 показывают особенность преимущества веб-воркеров с передачей данных по ссылке через буфер перед веб-воркерами с копированием через память при обмене данными: ускорение немногим более чем в два раза проявляется лишь при введении не более чем трёх веб-воркеров.

Интересно также сопоставить результаты с предыдущим вариантом, в котором используется элемент HTML5 Canvas с таймером (без веб-воркеров). Графики такого сравнения показаны на рисунке 2.26. Можно видеть, что для браузера Firefox ускорение в два раза происходит для всех n, задействованных в экспериментах, при существенном ускорении для небольших n (200 и менее). Для Chrome ускорения за счёт использования веб-воркеров можно добиться лишь для небольших n.

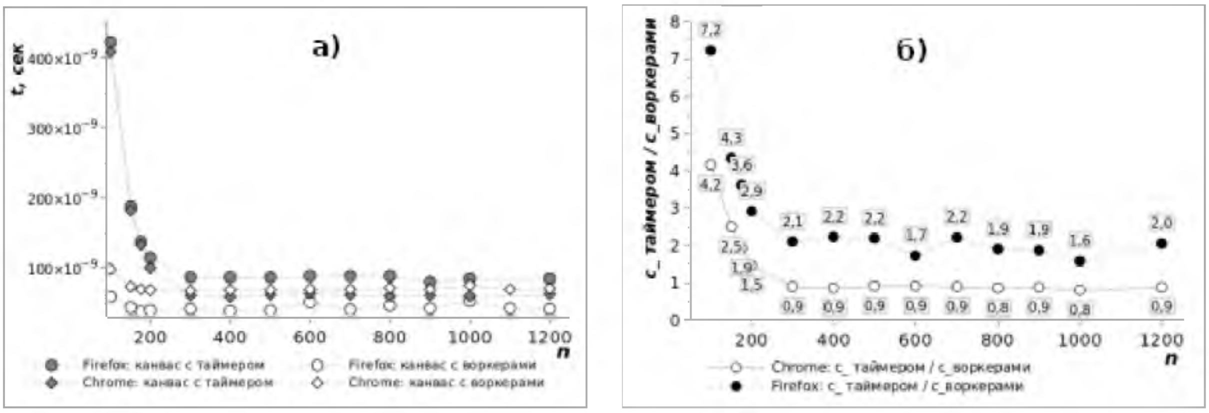

Рисунок 2.26 - Компьютер с AMDRYZENX8, браузеры Firefoxv.141-145 и Chromev.140-144: а) длительности вычислений в клетке с использованием Canvas с таймером и Canvas с воркерами (8 воркеров, обмен по ссылке); б) отношение длительности выполнения при Canvas с таймером к длительности выполнения при Canvas с воркерами (8 воркеров, обмен по ссылке)

Кроме изложенного выше следует добавить, что в экспериментах с использованием веб-воркеров с буфером не происходит аварийного завершения из-за нехватки памяти при достаточно больших n (1200 и более), тогда как в экспериментах с веб-воркерами без буфера крэши случаются уже при n = 1000.

Результаты экспериментов, изложенные в настоящем подразделе, позволяют сделать следующие выводы:

– использование двух и более веб-воркеров позволяет существенно увеличить скорость вычислений;

- наибольшего увеличения скорости вычислений можно достичь в случае, если число веб-воркеров равно числу ядер (физических) процессора используемого компьютера;
- при превышении числа веб-воркеров над числом ядер дальнейшего увеличения скорости вычислений не происходит;
- увеличение скорости вычислений при обмене данных копированием между потоками составляет 3,2÷4,7 раз, а при обмене через буфер - 4,6÷5,5 раз;
- использование варианта с обменом данными по ссылке более предпочтительно при небольших n: в настоящих экспериментах, при n = 200 и менее ускорение от 1,5 (FF) до 2,9 (Chrome) раз). Для Chrome, в отличие от FF, ускорение не менее чем в два раза наблюдается для достаточно больших n;
- использование веб-воркеров с буферизацией пересылаемых данных позволяет экономить оперативную память, что важно при крупномасштабном моделировании.

2.4 Проект оптимизации браузерного приложения

В таблице 2.1 приводятся результаты экспериментов, изложенные в предыдущих подразделах настоящего раздела, дающие обобщённое представление о возможностях ускорения имитационного моделирования за счёт использования новых веб-технологий.

Следует отметить, что при использовании веб-воркеров без буферизации ускорение происходит лишь для небольших масштабов (n = 175 и менее, в таблице 2.1 не отражено, см. графики на рисунках подраздела 2.3). В то же время, к преимуществам использования такого вида веб-воркеров следует отнести отсутствие задержки при обработке событий пользователя за счёт того, что вычисления выполняются параллельно в выделенных потоках и не препятствуют обработке событий, происходящих в главном потоке.

Таблица 2.1 – Возможности браузерного приложения за счёт применения ES6-технологий

№ п /п	Технология	Ускорение, во сколько раз			Примечание
		Ренде-ринг	Вычисле-ния	Рендеринг + вычисления	
1	HTML5 Canvas вместо HTML4 Tables	43÷147	2,5÷5,5	16÷37	Увеличение масштаба до n×2 за счёт экономии памяти
2	HTML5 Canvas+ Web Workers без буферизации	2,2÷55	3,1÷4,7	2,3÷9,5	Уменьшение масштаба до n за счёт потребности в памяти
3	HTML5 Canvas+ Web Workers с буферизацией	3,8÷29	3,2÷5,5	3,8÷21	Увеличение масштаба до n×3 за счёт экономии памяти

Дополнительный эффект от применения ES6-технологий - более экономное расходование оперативной памяти. В первую очередь это происходит при переходе от HTML4 Tables к HTML5 Canvas, а затем - при использовании WebWorkers с буферизацией.

В результате экономии памяти существенно увеличивается возможность масштабирования: по результатам экспериментов максимально допустимый размер клеточной области (n) для имитационного моделирования можно увеличить не менее, чем в два раза. Так, если ранее максимальное значение n = 600 (число клеток - 600×600), то с использованием новых технологий допустимо n = 1200 (число клеток - 1200×1200). Другими словами, без потери производительности площадь обрабатываемой акватории можно увеличить в два раза. Или разрешение модели, что иногда требуется (сложная береговая линия, неравномерность поля течений и прочее). Необходимо помнить, что это относится к бюджетным компьютерам с 8 Гб оперативной памяти.

3 Оценка эффективности оптимизации приложения

3.1 Вычислительная эффективность

На основе материалов настоящего исследования можно утверждать, что использование рассмотренных технологий ES6-стандарта приведёт к значимому увеличению скорости имитационного моделирования, как за счёт вычислений, так и за счёт рендеринга (обновления цветовых карт).

Наибольший эффект от ускорения вычислений достигается за счёт крупномасштабного распараллеливания с использованием выделенных потоков (DedicatedWorkers) с обменом данными по ссылке (через буфер) между главным и выделенными потоками. При этом можно ограничиться небольшим числом выделенных потоков (не более трёх). В любом случае не следует ожидать эффекта ускорения, если число потоков будет превышать число ядер процессора используемого компьютера.

Эффект от ускорения рендеринга обеспечивается путём замены цветовых карт, построенных с использованием элемента HTML4 Canvas, картами с использованием элемента HTML5 Canvas.

В целом следует ожидать, что даже при использовании бюджетных компьютеров с четырёхядерными процессорами при объёме ОЗУ 8Гб оптимизация позволит сократить длительность имитационного моделирования не менее, чем в 3 раза.

Дополнительный эффект от оптимизации состоит в экономии оперативной памяти, что позволяет увеличить возможности масштабирования за счёт двукратного увеличения расчётной области, либо такого же увеличения разрешения.

3.2 Экономическая эффективность

Экономическую эффективность результатов работы, выполненной в рамках настоящего исследования, можно оценить путём сравнения показателей до оптимизации и после оптимизации браузерного приложения.

По результатам таблицы 2.1 можно предположить, что оптимизация приводит как минимум к троекратному ускорению выполнения работы.

Стоимость работ имитационного (математического) моделирования по одному водному объекту можно взять из прейскуранта ФГБНУ ВНИРО (Всероссийский научно-исследовательский институт рыбного зозяйства и океанографии): https://azniirkh.vniro.ru/upload/files/service/2024/pril_1.pdf, которая составляет 56877,5 руб. (без НДС 20 %).

В настоящее время услуги по моделированию с использованием приложения на основе КА-модели и по ценам прейскуранта выполняет АзНИИРХ (филиал ВНИРО): https://sudact.ru/arbitral/doc/ZEYv7ZBpteJ8/, а также ряд коммерческих организаций экологического профиля в Краснодаре: ООО «Аква-Биоресурсы», ООО «Чистая планета», ООО «Экомониторинг» и др. Средний срок моделирования – 1 месяц.

Таким образом, организация, пользуясь приложением до оптимизации выполнит 12 заказов по моделированию в течение одного года.

Используя оптимизированную версию приложения организация имеет возможность выполнять в три раза больше, то есть 36 заказов в течение года.

Показателем экономической эффективности оптимизированной версии приложения может служить выгода в денежном выражении для одной экологической организации за один год:

Выгода/год, руб. = выручка после оптимизации – выручка до оптимизации = (56877,5 × 36) руб. - (56877,5 × 12) руб. = 2047590 руб. – 682530 руб. = 1365069 руб.   или: выгода за месяц (за один заказ) = 1365069/12 = 113755 руб.

Заключение

Web-технологии в настоящее время используются в самых разных областях повседневной деятельности, в том числе для ресурсоёмких вычислений.

В настоящей работе рассматриваются возможности повышения вычислительной эффективности браузерного приложения, предназначенного для численного имитационного моделирования загрязнения мелководных объектов. Это приложение разработано с использованием языка Javascript на основе спецификаций ES5, действовавших до 2009 года. В настоящее время браузеры поддерживают Javascriptc использованием спецификаций ES6 (и более), использование технологий которых позволяет достичь желаемого повышения вычислительной эффективности.

В первом разделе работы рассмотрены основные технологии современных браузерных движков и API для языка JavaScript, которые можно было бы применить при оптимизации нового варианта приложения для имитационного моделирования. К таким относятся технологии APIHTML5: Canvas и WebWorkers. В то же время следует ожидать, что применение рассмотренных технологий может приводить к различным результатам для конкретных приложений.

Во втором разделе работы освоение новых технологий закреплено разработкой программ, позволяющих убедиться в возможности ожидаемой оптимизации путём проведения экспериментов, соответствующих поставленным цели и задачам. Экспериментально показано, что основное ускорение моделирования можно достичь за счёт крупноблочного распараллеливания вычислений с использованием выделенных потоков (DedicatedWorkers). В то же время, оптимальное количество выделенных потоков зависит от количества ядер процессора, используемого компьютера, и архитектуры движка браузера.

Выполненные исследования позволили выявить основные возможности для оптимизации имитационного моделирования загрязнения водных объектов, позволяющие как минимум троекратно ускорить имитационные эксперименты в условиях региональной экологической организации без дорогостоящих затрат на модернизацию имеющихся компьютеров.

Увеличение скорости работы браузерного приложения в три раза даёт возможность региональным экологическим организациями пропорционально увеличить количество выполняемых заказов на моделирование от заинтересованных предприятий. Расчёт по действующему прейскуранту показывает, что экологическая организация может получать дополнительно около 114 тыс. руб. в месяц или 1365 тыс. руб. за год.

Выполненные эксперименты также позволили определить дополнительный эффект от предложенной оптимизации, состоящий в экономии ОЗУ используемых компьютеров. За счёт такой экономии площадь обсчитываемых водных объектов можно увеличить в два раза. Другая возможность - во столько же раз увеличить разрешение модели.

Список литературы

1. Балиев, Б. “JavaScript однопоточный или многопоточный. Ставим точку”. - URL: https://habr.com/ru/articles/786330/(дата обращения: 30.06.2025).
2. Бондарев, И.Н., Матерухин, А.В., Гвоздев, О.Г. Использование клеточных автоматов для имитационного моделирования распространения загрязнения атмосферного воздуха в условиях мегаполиса // Тр. Азиатской школы-семинара «Проблемы оптимизации сложных систем». -2020.- С. 10-15.
3. Бородин, О.В. Многопоточная обработка изображений с использованием APIWebWorkers / О.В. Бородин, В.А. Егунов // Прикаспийский журнал: управление и высокие технологии. -2021. -№3 (55). - С 33-46.
4. Волков, В.Ю., Мухин, А.А. Выбор упрощённой модели для системы поиска максимума концентрации загрязняющего вещества. Известия ТулГУ. Технические науки. 2021, вып.9. С. 218-220.
5. ИДЕНТИФИК-JS. Цикл событий: микрозадачи и макрозадачи. - URL: https://id.javascript.info/event-loop(дата обращения: 30.06.2025).
6. Как работают современные браузеры. Часть 1. - URL: https://habr.com/ru/companies/timeweb/articles/969508/(дата обращения: 07.12.2025).
7. Как работают современные браузеры. Часть 2. - URL: https://habr.com/ru/companies/timeweb/articles/974470/(дата обращения: 13.12.2025).
8. Клешнёв, Дмитрий. “Среда выполнения JavaScript простым языком: движок, Event Loop и очереди задач”. - URL: https://habr.com/ru companies/gnivc/articles/910918/(дата обращения: 30.06.2025).
9. Колесников, А. Клеточные автоматы и компьютерная экология. Журнал "Компьютерные вести" (КВ),- №10, 2002.
10. Лабберс, П. HTML5 для профессионалов / П. Лабберс, Б. Олберс., Ф. Салим // - М.: ООО «И.Д. Вильямс», 2011. - 272 с.
11. Can I use? Web Workers / Baseline Widely available across major browsers. - URL: https://caniuse.com/webworkers (датаобращения: 23.07.2025).
12. Макс-@AloneCoder. Javascript-движки: как они работают? От стэка вызо-

вов до промисов - (почти) всё, что вам нужно знать. - URL: https://habr.com/ru/companies/vk/articles/452906/ (дата обращения: 10.08.2025).

13. Парфёнов, Никита (@Dragonec). WebWorkers в Javascript: Параллельные вычисления и улучшение производительности. - URL: https://habr.com/ru/articles/767494/ (дата обращения: 25.08.2025).

14. Полупанов, В.Н. Имитационная модель загрязнения взвешенными веществами мелководных водных объектов. // Сб. статей конференции, т. 11 (95). - М., ООО «Интернаука», -2021.-С. 14-56.

15. Тоффоли Т. Машины клеточных автоматов: пер. с англ. / Т. Тоффоли, Н. Марголус. — М.: Мир, 1991. — 280 с.

16. AnyLogicCloud - веб-платформа для запуска имитационных моделей. URL: https://www.anylogic.ru/features/cloud/(дата обращения: 25.07.2025).

17. Keul Blog / How to execute CPU intensive tasks in your JavaScript application using Worker. - URL: https://blog.keul.it/javascript-execute-cpu-intensive-tasks-with-workers/. (дата обращения: 25.08.2025).

18. MDN - Руководство по Canvas. - URL: https://developer.mozilla.org/ru/docs /Web/API/Canvas_API/Tutorial(дата обращения: 10.08.2025).

19. MDN - Cross-Origin Resource Sharing (CORS). - URL: https:// developer.mozilla.org/ru/docs/Web/HTTP/Guides/CORS (датаобращения: 25.07.2025).

20. MDN - OffscreenCanvas. URL: https://developer.mozilla.org/en-US /docs/Web/API/OffscreenCanvas (датаобращения: 9.08.2025).

21. MDN - типизированные массивы Javascript. - URL: https:// developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Typed_arrays(дата обращения: 25.08.2025).

22. MDN - Интерфейсы WebAPI - WebWorkersAPI. - URL: https:// developer.mozilla.org/ru/docs/Web/API/Web_Workers_API(дата обращения: 25.08.2025).

23. MDN - WebWorkersAPI. Алгоритм структурного клонирования. - URL: https://developer.mozilla.org/ru/docs/Web/API/Web_Workers_API/Structured_clone _algorithm(дата обращения: 25.08.2025).

24. MDN - Transferable objects. - URL: https://developer.mozilla.org/en-US/docs/

Web/API/Web_Workers_API/Transferable_objects(датаобращения: 25.08.2025).
METANIT.COM. Web Worker API. Определение и выполнение веб-воркера. - URL: https://metanit.com/web/javascript/26.1.php (датаобращения: 25.08.2025).
25. Optimizing Javascript Applications Perfomance with Web Workers. - URL: https://www.twilio.com/en-us/blog/developers/community/optimize-javascript-application-performance-web-workers(датаобращения: 25.07.2025).
26. Peretz, Bnaya. Objectbuffer: object-like API, backed by a [shared]arraybuffer. - URL: https://github.com/Bnaya/objectbuffer?tab=readme-ov-file(датаобращения: 25.07.2025).
27. Verdu, Javier, Pajuelo, Alex. Performance Scalability Analysis of JavaScript Applications with Web Workers. Department of Computer Architecture, BarcelonaTECH (UPC), DOI 10.1109/LCA.2015.2494585.
28. RU_VDS. Веб-воркеры в Javasvript: безопасный параллелизм. - URL: https://habr.com/ru/companies/ru vds/articles/352828/(дата обращения: 25.08.2025).

Приложение А

Исходные коды экспериментальных программ

А1 Использование технологий HTML4 и ES5

Файл index.html

```
<html>
<head>
<title>Клеточные автоматы: расчёт облака примеси (таблица с таймером)</title>
<META http-equiv=Content-Type content="text/html; charset=utf-8">
<META http-equiv=Content-Language content=ru>
<meta http-equiv="Cache-Control" content="no-cache, no-store, must-revalidate">
<meta http-equiv="Pragma" content="no-cache">
<meta http-equiv="Expires" content="0">
<link rel="stylesheet" href = "styles.css">
<script language=Javascript>
/* Построение таблицы-сетки  */
function grider(n) {
    var el tables = document.getElementById('ins tbl');
    if (el tables) {
        el tables.remove();
        for (var i = 0; i < n; i++) {
             for (var j = 0; j<n; j++) {
                 b[i][j] = 0;
                 c[i][j] = 0;
             }
                 }
    }
        tbl="<table id='ins tbl' style='background-image:URL(Kerch pen.gif);" +
                "background-size: auto "+n+"px'>";
        for (var i str=0; i str<n; i str++) {
                tbl+="<tr>";
                for (var i col=0; i col<n; i col++) {
                        tbl+="<td></td>";
                }
                tbl+="</tr>";
        }
        tbl+="</table>";
        document.all.grid.innerHTML=tbl;
        for (var i=0; i < n; i++) {
    document.all.ins tbl.rows[i].cells[0].style.backgroundColor="#000000";
    document.all.ins tbl.rows[i].cells[n-1].style.backgroundColor="#000000";
    document.all.ins tbl.rows[0].cells[i].style.backgroundColor="#000000";
    document.all.ins tbl.rows[n-1].cells[i].style.backgroundColor="#000000";
        }
}
/* Перерасчёт значений примеси в узлах квадратной сетки
        с перерисовкой цветовой карты */
functionmixer(frm all) {
        if (f===0){
                n=1*frm all.elements[0].value;
                n render = 1*frm all.elements[1].value;
                y = n/5, x = n/4, z, N = 1000;
                frm all.elements[11].value = y;
                frm all.elements[12].value = Math.round(x);
                frm all.elements[14].value = y;
                frm all.elements[15].value = Math.round(x+x);
                frm all.elements[17].value = y;
                frm all.elements[18].value = Math.round(x+x+x);
                n cycles add = 1000*n/N;
                frm all.elements[20].value = n cycles add.toFixed(0);
                frm all.elements[13].value = 1000*n/N;
                frm all.elements[16].value = 1000*n/N;
                frm_all.elements[19].value = 1000*n/N;
```

продолжение приложения А1

```
                w=newArray(9);
                z=new Array(3);
                b=new Array (n);
                for (var i=0; i<n; i++) {b[i]=new Array(n)};
                c=new Array (n);
                for (var i=0; i<n; i++) {c[i]=new Array(n)};

                for (var i = 0; i <= 8; i++) {
                        w[i]=1.00*frm all.elements[i+2].value;
                };
                for (var i = 0; i < n; i++) {
            for (var j = 0; j < n; j++) {
                b[i][j] = 0;
c[i][j] = 0;
            }
                }
                f=f+1;
        }

    /* Определяем три точки - источники вытекающей примеси на каждом цикле
        и вбрасываем в этих точках взвесь заданной концентрации
        заданное количество циклов */
if (n cycles<n cycles add) {
                if (1*frm all.elements[13].value!=0) {
                        //x = x;
                        y = 1*frm all.elements[11].value;
                        x = 1*frm all.elements[12].value;
                        z = 1*frm all.elements[13].value;
                        b[y][x] = b[y][x]+z;  // 1-я точка вброса примеси
                };
                if (1*frm all.elements[16].value!=0) {
                        //x = x+x;
                        y = 1*frm all.elements[14].value;
                        x = 1*frm all.elements[15].value;
                        z = 1*frm all.elements[16].value;
                        b[y][x] = b[y][x]+z;  // 2-я точка
                };
                if (1*frm all.elements[19].value!=0) {
                        //x = x+x/2;
                        y = 1*frm all.elements[17].value;
                        x = 1*frm all.elements[18].value;
                        z = 1*frm all.elements[19].value;
                        b[y][x] = b[y][x]+z;  // 3-я точка
                };
        }
        /* Продолжаем моделировать, но без вбросов, если они прекратились */
        for (i = 1; i < n-1; i++) {
                for ( j = 1; j < n-1; j++) {
                    c[i][j] = 0;
                    k = 0;
                    for (l = -1; l<=1; l++) {
                        for (m = -1; m<=1; m++) {
                            k = k + 1;
                            c[i][j] = c[i][j] + w[k-1] * b[i + l][j + m];
                        }
                    }
                    // расцвечиваем клетку по новому значению концентрации
                    if (n cycles%n render == 0) {
                document.all.ins tbl.rows[i].cells[j].style.backgroundColor=
                                                       colour(c[i][j]);
                    }
```

```
                        // Останов при заданной концентрации на границе квадрата
                        //      (для эксперимента)
                        if (i == n-2 &&c[i][j] >= 2) {
                                clearInterval(var time);
                            break;
                        }
            }
        }
    for (var i = 1; i < n-1; i++) {
            for (var j = 1; j < n-1; j++) {
                b[i][j] = c[i][j];
        }
    }

        n cycles = n cycles +1;
        frm all.elements[21].value = n cycles;
        t last = performance.now();
        t all = ((t last - t start)/1e3);
        frm all.elements[22].value = t all.toFixed(3);
}
/* Определение цвета по значению концентрации примеси  */
function colour(zz) {
        var zz min = 2;
        var zz max = 52;
        var zz delta = zz max - zz min;
        var zz grdnt = zz delta / 8;
        var zz1 =zz min + zz grdnt;
        var zz2 = zz1 + zz grdnt;
        var zz3 = zz2 + zz grdnt;
        var zz4 = zz3 + zz grdnt;
        var zz5 = zz4 + zz grdnt;
        var zz6 = zz5 + zz grdnt;
        var zz7 = zz6 + zz grdnt;
        // Добавляем градаций, чтобы увеличить нагрузку на рендеринг
        if (zz>= 0 &&zz<zz min-1.9) {clr="rgba(0,255,0,0.0)"}; //Прозрачный#1
        if (zz>=zz min-1.9
&&zz<zz min-1.7){clr="rgba(0,255,0,0.1)"};//Прозрачный#2
        if (zz>=zz min-1.7 &&zz<zz min-1){clr="rgba(0,255,0,0.2)"};//Прозрачный#3
        //
        if (zz>=(zz min-1) &&zz<(zz+1)){clr = "rgba(0,255,0,0.5)"};//Зелёный
        if (zz>=(zz min+1) &&zz<zz1) {clr  = "rgb(139,0,255)"}; //Фиолетовый
        if (zz>=zz1 &&zz<zz2) {clr  = "rgb(0,0,255)"}; //Синий
        if (zz>=zz2 &&zz<zz3) {clr  = "rgb(0,255,255)"}; //Голубой
        if (zz>=zz3 &&zz<zz4) {clr  = "rgb(0,80,0)"}; // Зелёный
        if (zz>=zz4 &&zz<zz5) {clr  = "rgb(255,255,0)"}; // Жёлтый
        if (zz>=zz5 &&zz<zz6) {clr  = "rgb(255,165,0)"}; // Оранжевый
        if (zz>=zz6 &&zz<zz7) {clr  = "rgb(228,0,51)"}; // Красный
        if (zz>=zz7 &&zz<zz max){clr= "rgb(51,51,0)"};//Тёмно-грязно-зелёный
        if (zz>zz max) {clr = "rgb(0,0,0)"}; // Чёрный
        return clr;
}
</script>
</head>

<body onUnload="if (null != var time) clearInterval(var time);">
<center><nobr>
<h1>РАСПРОСТРАНЕНИЕ ИНЕРТНОЙ ПРИМЕСИ<br>
<small>(с использованием таблицы и таймера)</small></h1>
<script>
var var time,f,x,y,z,n;
varclr;
f=0;
</script>
```

продолжение приложения А1

```
<form id="params in">
        <b>n (узлов квадратной области = n</b>x<b>n):</b>
        <input type="text" id="n cells" size=3 value=600><br>
        <b>n rendering (интервал циклов перерисовки цветовой карты):</b>
        <input type="text" id="n rendering" size=3 value=20><br>
        <b>Повторяемости направлений ветров:</b><br>
        СЗ: <input type="text" size=3 value=0.130>
        С: <input type="text"  size=3 value=0.200>
        СВ: <input type="text" size=3 value=0.130>
        З: <input type="text"  size=3 value=0.110><br>
        штиль: <input type="text" size=3 value=0.107><br>
        В: <input type="text"  size=3 value=0.110>
        ЮЗ: <input type="text"  size=3 value=0.066>
        Ю: <input type="text"  size=3 value=0.081>
        ЮВ: <input type="text"  size=3 value=0.066 ><br>

        <b>Координаты узлов (y,x) поступления примеси<br>
                и концентрация примеси в них (z):</b><br>
        Узел №1: y= <input type="text" size=3>
        x= <input type="text" size=3>
        z= <input type="text" size=3 value=0><br>
        Узел №2: y= <input type="text" size=3>
        x= <input type="text" size=3>
        z= <input type="text" size=3 value=0><br>
        Узел №3: y= <input type="text" size=3>
        x= <input type="text" size=3>
        z= <input type="text" size=3 value=0><br>

        Циклов  с вбросами после старта: <input type="text" size=4 value=0><br>
        Циклов      всего    
            после   старта: <input type="text" size=4 value=0><br>
        Времени   (секунд)  после  старта:
        <input type="text" size=5 value=0><br>

        <input type="button" onClick="grider(params in.n cells.value);" val-
ue="Строить сетку">

        <input type="button" onClick="if (document.all.ins tbl) {
                        var time=setInterval('mixer(params in)',0);
                        t start = performance.now();
                        n cycles = 0; t cycles = 0;
                } else {
                        alert('Вначале надо построить сетку.')
                }" value="Старт">

        <input type="button" onClick="clearInterval(var time)" value="Стоп">
        <hr>
</form>
<div id="grid"></div>
</nobr></center>
</body>
</html>
```

продолжение приложения А1
Файл styles.css

```
body {
	font-family: Arial Narrow, Helvetica, Arial, sans-serif;
	color:#000000;
	background: #ccffcc
}
h1 {
	color: #44687d;
	padding-bottom: 5px;
	margin: 5px 0 5px 0;
	font-size: 175%;
}
input {
	margin-top: 0px;
	margin-bottom: 5px;
}
table {
	border-style: solid; border-width:1px; border-color:#800000;
	margin:0; padding:0px; background: #FFFFFF URL(./Kerch pen.gif);
	background-repeat:no-repeat; border-collapse: collapse
}
td {border-style: none; margin:0px; padding:0px; height:1px; width:1px}
```

Приложение А2
Использование HTML5 Canvas вместо HTML4 Tables

Файл index.html

```
<!DOCTYPE html>
<head>
<title>Клеточные автоматы: расчёт облака примеси (канвас с таймером)</title>
<META http-equiv=Content-Type content="text/html; charset=utf-8">
<META http-equiv=Content-Language content=ru>
<meta http-equiv="Cache-Control" content="no-cache, no-store, must-revalidate">
<meta http-equiv="Pragma" content="no-cache">
<meta http-equiv="Expires" content="0">
<link rel="stylesheet" href = "styles.css">
<scriptlanguage=Javascript>

// Определяем canvas в качестве сеточной области
function grider(n) {
    let el canvas = document.getElementById('canvas');
    if (el canvas) {
        el canvas.remove();
        for (let i = 0; i < n; i++) {
            for (let j = 0; j<n; j++) {
                b[i][j] = 0;
                c[i][j] = 0;
            }
        }

    }
    canvas = document.createElement('canvas');
    canvas.id = "canvas";
    canvas.width = n;
    canvas.height = n;
    img = new Image();
    img.src = "./Kerch pen.gif";
    ctx = canvas.getContext('2d');
    img.onload = function () {
        document.getElementById("grid").style.cssText ='width:'+n+'px;height:'+
                n+'px; background: URL(./Kerch pen.gif); background-size: auto '+n+
                'px;border-style:solid; border-width:1px; border-color:black;';
        imageData = ctx.getImageData(0,0,n,n);
        data = imageData.data;
        document.getElementById("grid").appendChild(canvas);
}
    setRunningState(false);
}
```

продолжение приложения А2

```
/* Перерасчёт значений примеси в узлах квадратной сетки
        с перерисовкой цветовой карты в канвас-элементе */
function mixer(frm all) {
        n=1*frm all.elements[0].value;
        n render = 1*frm all.elements[1].value;
        if (f===0){
                y = n/5, x = n/4, z, N = 1000;
                frm all.elements[11].value = y;
                frm all.elements[12].value = Math.round(x);
                frm all.elements[14].value = y;
                frm all.elements[15].value = Math.round(x+x);
                frm all.elements[17].value = y;
                frm all.elements[18].value = Math.round(x+x+x);
                n cycles add = 1000*n/N;
                frm all.elements[20].value = n cycles add.toFixed(0);
                frm all.elements[13].value = 1000*n/N;
                frm all.elements[16].value = 1000*n/N;
                frm all.elements[19].value = 1000*n/N;
                w=new Array(9);
                z=new Array(3);
                b=new Array (n);
                for (i=0; i<n; i++) {b[i]=new Array(n)};
                c=new Array (n);
                for (let i=0; i<n; i++) {c[i]=new Array(n)};

                for (let i = 0; i<=8; i++) {
                        w[i]=1.00*frm all.elements[i+2].value;
                };
                for (let i = 0; i < n; i++) {
                for (let j = 0; j<n; j++) {
                        b[i][j] = 0;
                        c[i][j] = 0;
                }
                }
                f=f+1;
                setRunningState(false);
        }
    /* Определяем три точки - источники вытекающей примеси, на каждом цикле
        и вбрасываем в этих точках взвесь заданной концентрации
        заданное количество циклов */
if (n cycles<n cycles add) {
                if (1*frm all.elements[13].value!=0) {
                        y = 1*frm all.elements[11].value;
                        x = 1*frm all.elements[12].value;
                        z = 1*frm all.elements[13].value;
                        b[y][x] = b[y][x]+z;  // 1-я точка вброса примеси
                };
                if (1*frm all.elements[16].value!=0) {
                        y = 1*frm all.elements[14].value;
                        x = 1*frm all.elements[15].value;
                        z = 1*frm all.elements[16].value;
                        b[y][x] = b[y][x]+z;  // 2-я точка
                };
                if (1*frm all.elements[19].value!=0) {
                        y = 1*frm all.elements[17].value;
                        x = 1*frm all.elements[18].value;
                        z = 1*frm all.elements[19].value;
                        b[y][x] = b[y][x]+z;  // 3-я точка
                };
        }
```

```
        /* Продолжаем моделировать, но без вбросов, если они прекратились */
        for (let i = 1; i < n-1; i++) {
                for (let j = 1; j < n-1; j++) {
                        c[i][j] = 0;
                        k = 0;
                        for (let l = -1; l<=1; l++) {
                            for (let m = -1; m<=1; m++) {
                                        k = k + 1;
                                        c[i][j] = c[i][j] + w[k-1] * b[i + l][j + m];
                        }
                        }

                        // Перерасцвечиваем клетку по новому значению концентрации
                        if (n cycles%n render == 0) {
                                rgbaValues = colour(c[i][j]);
                                const px i = ((j) + (i) * n) * 4;
                                data[px i] = rgbaValues.r;
                                data[px i+1] = rgbaValues.g;
                                data[px i+2] = rgbaValues.b;
                                data[px i+3] = rgbaValues.b;
                        }

                        // Останов при достижении заданной концентрации границы
                        //      квадрата (для эксперимента)
                        if (i == n-2 &&c[i][j] >= 2) {
                        clearInterval(var time);
                            break;
                        }
            }
        }
        ctx.putImageData(imageData, 0, 0);
        for (let i = 0; i < n; i++) {
            for (let j = 0; j < n; j++) {
                b[i][j] = c[i][j];
            }
        }
        n cycles = n cycles +1;
        frm all.elements[21].value = n cycles;
        t last = performance.now();
        t all = ((t last - t start)/1e3);
        frm all.elements[22].value = t all.toFixed(3);
        setRunningState(true);
}

function stopMove() {
        setRunningState(false);
}

function setRunningState(p) {
    // При выполнении кнопка stop доступна, а кнопка start - нет
    document.getElementById("start move").disabled = p;
    document.getElementById("stop move").disabled = !p;
}
```

```
/* Определение компонент rgba-цвета по значению концентрации примеси  */
function colour(zz) {
        const zz min = 2;
        const zz max = 52;
        const zz delta = zz max - zz min;
        const zz grdnt = zz delta / 8;
        const zz1 =zz min + zz grdnt;
        const zz2 = zz1 + zz grdnt;
        const zz3 = zz2 + zz grdnt;
        const zz4 = zz3 + zz grdnt;
        const zz5 = zz4 + zz grdnt;
        const zz6 = zz5 + zz grdnt;
        const zz7 = zz6 + zz grdnt;
        letr, g, b, a;
        // Добавляем градаций, чтобы увеличить нагрузку на рендеринг, как в вар.ES5
        if (zz>= 0 &&zz<zz min-1.75) {r=0; g=255; b=0; a=0;}; //Прозрачный#1
        if (zz>=zz min-1.75 &&zz<zz min-1.25){r=0; g=255; b=0;
a=50;};//Прозрачный#2
        if (zz>=zz min-1.25 &&zz<zz min-0.75){r=0; g=255; b=0;
a=100;};//Прозрачный#3
        //
        if (zz>=(zz min-0.75) && zz<(zz+1)) {r=0; g=255; b=0; a=255;}; // Зелёный

        if (zz>=(zz min+1) && zz<zz1) {r=139; g=0; b=255; a=255;}; // Фиолетовый
        if (zz>=zz1 && zz<zz2) {r = 0; g = 0; b = 255; a = 255;}; //Синий
        if (zz>=zz2 && zz<zz3) {r = 0; g = 255; b = 255; a = 255;}; //Голубой
        if (zz>=zz3 && zz<zz4) {r = 0; g = 80; b = 0; a = 255;}; // Тёмно-зелёный
        if (zz>=zz4 && zz<zz5) {r = 255; g = 255; b = 0; a = 255;}; // Жёлтый
        if (zz>=zz5 && zz<zz6) {r = 255; g = 165; b = 0; a = 255;}; // Оранжевый
        if (zz>=zz6 && zz<zz7) {r = 228; g = 0; b = 51; a = 255;}; // Красный
        if (zz>=zz7 && zz<zz max) {r=51; g=51; b=0; a=255;}; // Тёмно-грязно-зелёный
        if (zz>zz max) {r = 0; g = 0; b = 0; a = 255;}; // Чёрный
        return {r, g, b, a};
}</script>
</head>
```

продолжение приложения А2

```
<body onUnload="if (null != var time) clearInterval(var time);">
<center><nobr>
<h1>РАСПРОСТРАНЕНИЕ ИНЕРТНОЙ ПРИМЕСИ<br>
<small>(с использованием канвас и таймера)</small></h1>
<script>
var var time,f,x,y,z,n;
var clr;
f=0;
</script>
<form id="params in">
        <b>n (узлов квадратной области = n</b>x<b>n):</b>
        <input type="text" id="n cells" size=3 value=600><br>
        <b>n rendering (интервал циклов перерисовки цветовой карты):</b>
        <input type="text" id="n rendering" size=3 value=20><br>
        <b>Повторяемости направлений ветров:</b><br>
        СЗ: <input type="text" size=3 value=0.130>
        С: <input type="text"  size=3 value=0.200>
        СВ: <input type="text" size=3 value=0.130>
        З: <input type="text"  size=3 value=0.110><br>
        штиль: <input type="text" size=3 value=0.107><br>
        В: <input type="text"  size=3 value=0.110>
        ЮЗ: <input type="text"  size=3 value=0.066>
        Ю: <input type="text"  size=3 value=0.081>
        ЮВ: <input type="text"  size=3 value=0.066 ><br>

        <b>Координаты узлов (y,x) поступления примеси<br>
                и концентрация примеси в них (z):</b><br>
        Узел №1: y= <input type="text" size=3>
        x= <input type="text" size=3>
        z= <input type="text" size=3 value=0><br>
        Узел №2: y= <input type="text" size=3>
        x= <input type="text" size=3>
        z= <input type="text" size=3 value=0><br>
        Узел №3: y= <input type="text" size=3>
        x= <input type="text" size=3>
        z= <input type="text" size=3 value=0><br>

        Циклов  с вбросами после старта: <input type="text" size=4 value=0><br>
        Циклов      всего    
            после   старта: <input type="text" size=4 value=0><br>
        Времени   (секунд)  после  старта:
        <input type="text" size=5 value=0><br>

        <input type="button" onClick="grider(params in.n cells.value);"
            value="Канвас-карта-сетка">

        <input type="button" id="start move" onClick="
                var time=setInterval('mixer(params in)',0);
                t start = performance.now();
                n cycles = 0; t cycles = 0;" value="Старт" disabled>

        <input type="button" id="stop move"
            onClick="clearInterval(var time); stopMove()" value="Стоп">

        <input type="button"
            onclick="document.location = document.location;" value="Reload">

        <hr>
</form>
<div id="grid"></div>
</nobr></center>
</body>
</html>
```

Приложение А3

Использование WebWorkers вместо таймера.Файл index.html

```
<!DOCTYPE html>
<html lang="ru-RU">
<head>
<meta charset="utf-8">
<meta http-equiv="Cache-Control" content="no-cache, no-store, must-revalidate">
<meta http-equiv="Pragma" content="no-cache">
<meta http-equiv="Expires" content="0">
<title>Клеточные автоматы - Canvas+Web Workers - через буфер</title>
<link rel="stylesheet" href = "styles.css">
</head>
<body onload = "init dumping(params in)" id="content">
<center><nobr>
<h1>РАСПРОСТРАНЕНИЕ ИНЕРТНОЙ ПРИМЕСИ<br>
        <small>HTML5 Canvas + WebWorkers с передачей данных через буфер</small></h1>
<pid="status">Ваш браузер не поддерживает HTML5 WebWorkers.</p>
<script>
var f,x,y,z,n;
f=0;
</script>
<form id="params in">
        <b>n (узлов квадратной области = n</b>x<b>n):</b>
        <input type="text" id="n cells" size=3 value=600
                onchange="init dumping(params in)"><br>
        <b>n rendering (интервал циклов перерисовки цветовой карты):</b>
        <input type="text" id="n rendering" size=3 value=20><br>
        <b>Повторяемости направлений ветров:</b><br>
        СЗ: <input type="text" size=3 value=0.130>
        С: <input type="text"  size=3 value=0.200>
        СВ: <input type="text" size=3 value=0.130>
        З: <input type="text"  size=3 value=0.110>
        штиль: <input type="text" size=3 value=0.107><br>
        В: <input type="text"  size=3 value=0.110>
        ЮЗ: <input type="text"  size=3 value=0.066>
        Ю: <input type="text"  size=3 value=0.081>
        ЮВ: <input type="text"  size=3 value=0.066
                onchange="rose(params in)"><br>

        <b>Координаты узлов (y,x) поступления примеси<br>
                и концентрация примеси в них (z):</b><br>
        Узел №1: y= <input type="text" size=3>
        x= <input type="text" size=3>
        z= <input type="text" size=3 value=0><br>
        Узел №2: y= <input type="text" size=3>
        x= <input type="text" size=3>
        z= <input type="text" size=3 value=0><br>
        Узел №3: y= <input type="text" size=3>
        x= <input type="text" size=3>
        z= <input type="text" size=3 value=0><br>

        <label for="workerCount">Число воркеров:</label>
        <select id="workerCount">
                <option>1</option>
                <option selected>2</option>
                <option>3</option>
                <option>4</option>
                <option>5</option>
                <option>6</option>
                <option>7</option>
                <option>8</option>
                <option>9</option>
                <option>10</option>
        </select>
```

```
        Циклов  с вбросами после старта: <inputtype="text"size=4
                value=0><br>
        Циклов      всего    
        после   старта: <input type="text" id="n sycles" size=4
            value=0><br>
        Времени  (секунд)  после  старта:
        <input type="text" id="t all" size=5 value=0><br>
        Сведения о ходе работы воркеров в log: <inputtype="checkbox"
                id="no yes log"><br>
        <input type="button" onClick = "grider(params in.n cells.value)"
                value="Канвас-карта-сетка">
        <input type="button" id="start move" value="Старт" disabled>
        <input type="button" id="stop move" value="Стоп" disabled>
        <input type="button" onclick="document.location = document.location;"
                value="Reload">
        <hr>
</form>
<div id="grid"></div>
<div id="logOutput"></div>
<script>
/* Автоматизируем заполнение значений в полях формы для трёх точек вброса */
function  init dumping(frm all) {
                var n = 1*frm all.elements[0].value;
                var n render = 1*frm all.elements[1].value;
                var y = Math.floor(n/5), x = Math.floor(n/4), z = n;
                frm all.elements[11].value = y;
                frm all.elements[12].value = x;
                frm all.elements[14].value = y;
                frm all.elements[15].value = x+x;
                frm all.elements[17].value = y;
                frm all.elements[18].value = x+x+x;
                n sycles add = n;
                frm all.elements[21].value = n sycles add.toFixed(0);
                frm all.elements[13].value = z;
                frm all.elements[16].value = z;
                frm all.elements[19].value = z;
}
/* Определяем canvas в качестве сеточной области */
function grider(n) {
    let el canvas = document.getElementById('canvas');
    if (el canvas) el canvas.remove();
    canvas = document.createElement('canvas');
    canvas.id = "canvas";
    canvas.width = n;
    canvas.height = n;

    img = new Image();
    img.src = "./Kerch pen.gif";
    ctx = canvas.getContext('2d');

    img.onload = function () {
        document.getElementById("grid").style.cssText ='width:'+n+'px;height:'+
                n+'px; background: URL(./Kerch pen.gif); background-size: auto '+n+
                'px;border-style:solid; border-width:1px; border-color:black;';
        imageData = ctx.getImageData(0,0,n,n);
        data = imageData.data;
        document.getElementById("grid").appendChild(canvas);
    }
        setRunningState(false);
}
```

продолжение приложения А3

```
var workers = [];

function log(s) {
    var logOutput = document.getElementById("logOutput");
    logOutput.innerHTML = s + "<br>" + logOutput.innerHTML;
}

function setRunningState(p) {
    // При выполнении кнопка stop доступна, а кнопка start - нет
    document.getElementById("start move").disabled = p;
    document.getElementById("stop move").disabled = !p;
}

/* Создаётся экземпляр веб-воркера */
function initWorker(src) {
    var worker = new Worker(src);
    worker.addEventListener("message", messageHandler, true);
    worker.addEventListener("error", errorHandler, true);
    returnworker;
}

/* Начинаем моделирование с использованием веб-воркеров */
function startMove() {
        t start = performance.now(); t cycles = 0; rab width = 0;
        n = canvas.width;
        var n sycles = 0;
        let n br;
        var d = document.getElementById("params in");
        n render = 1*d[1].value; // Интервал циклов обновления карты
                // здесь используется лишь в двух вариантах:
                // 1 - обновление на каждом цикле; не 1 - лишь на последнем цикле
        varworkerCount = 1*d[20].value; // Воркеров
        varn sycles add = 1*d[21].value; // Циклов вброса примеси
        d[22].value = d[23].value = 0; // Обнуляем предыдущие результаты
        var no yes log = d[24].checked;
    var width = Math.floor(n/workerCount);
// Массив весов. Упрощаем: веса = повторяемости ветров
    //  (сумма должна быть равна 1)
letw=newArray(9);
for (let i = 0; i<=8; i++) w[i]=d[i+2].value;
        // Координаты и концентрация в точках вброса
        let dump1 = [1*d[11].value,1*d[12].value,1*d[13].value];
        let dump2 = [1*d[14].value,1*d[15].value,1*d[16].value];
        let dump3 = [1*d[17].value,1*d[18].value,1*d[19].value];
        // Инициализируем массив для клеточного автомата
        letc = newArray(n*n);
        for (let i=0; i < c.length; i++) c[i]=0;

        // Для останова при достижении заданного количества циклов
        //      (для сравнимости с экспериментами варианта с таймером)
        switch (n) {
                case (100):
                        n br = 274;    break;
                case (150):
                        n br = 424;    break;
                ...
        }
```

продолжение приложения А3

```
   // Создаём веб-воркер с нужными свойствами, заносим в массив веб-воркеров.
   // Каждому веб-воркеру - одинаковое задание, и они работают параллельно,
   //   каждый в своей области
    for (let i worker=0; i worker<workerCount; i worker++) {
        let worker = initWorker("moveWorker.js");
        worker.index worker = i worker;
        worker.width = width;
        worker.c = c;
        worker.w = w;
        worker.dump1 = dump1;
        worker.dump2 = dump2;
        worker.dump3 = dump3;
        worker.n render = n render;
        worker.n sycles add = n sycles add;
        worker.n sycles = n sycles;
        worker.no yes log = no yes log;
        worker.n br = n br;
        workers[i worker] = worker;
        sendMoveTask(worker, i worker, width, c, w,dump1, dump2,dump3,
       n render, n sycles add, n sycles, n br, no yes log);
    }
    setRunningState(true);
}

function sendMoveTask(worker, i worker, chunkWidth, cc, w, dump1,dump2,dump3,
        n render, n sycles add, n sycles, n br, no yes log) {
    let chunkStartX = i worker * chunkWidth;
    let chunkStartY = 0;
    var n = Math.sqrt(cc.length);

        const buffer = new ArrayBuffer(n*n*4);
        let c = new Float32Array(buffer);
        for (let i=0; i < c.length; i++) c[i]=cc[i];
    worker.postMessage({
        'type' : 'move',
            'index worker' : i worker,
            'width' : chunkWidth,
            'startX' : chunkStartX,
            'c' : c,
            'w' : w,
            'dump1' : dump1,
            'dump2' : dump2,
            'dump3' : dump3,
            'n render' : n render,
            'n sycles add' : n sycles add,
            'n sycles' : n sycles,
            'n br' : n br,
            'no yes log' : no yes log
    }, [buffer]);
}

function stopMove() {
   for (let i worker=0; i worker<workers.length; i worker++) {
        workers[i worker].terminate();
    }
   t last = performance.now();
   t all = ((t last - t start)/1e3);
   document.getElementById("t all").value = t all.toFixed(3);
   setRunningState(false);
}
```

продолжение приложения А3

```
function messageHandler(e) {
        let n = Math.sqrt(e.target.c.length);
    let messageType = e.data.type;
    switch (messageType) {
        case ("status"):
            log(e.data.statusText);
            break;
        case ("progress"):
        for (let i = 0; i<n; i++) {
                for (let j = e.data.startX; j<e.data.startX+e.data.width; j++) {
                        const index = (j + i * n);
                        let val = e.data.c[index];
                        if (val === null || val< 0) {
                        log("Недопустимая концентрация в клетке: i = " + i +
                                ", j = " + j);
                        return;
                        }
                        // Расцвечиваем клетку по значению концентрации из воркера
                                        rgbaValues = colour(val);
                                        const px i = (j + i * n) * 4;
                                        data[px i] = rgbaValues.r;
                                        data[px i+1] = rgbaValues.g;
                                        data[px i+2] = rgbaValues.b;
                                        data[px i+3] = rgbaValues.a;
                }
           }
           ctx.putImageData(imageData, 0, 0);
            document.getElementById("n sycles").value = e.data.n sycles;
// Останов по достижению заданного кол-ва циклов
            // (для сравнимости экспериментов с таймером)
let workerCount =
        parseInt(document.getElementById('workerCount').value);
            if (e.data.n sycles == e.data.n br) {
        rab width = rab width + e.target.width;
        //если область по ширине почти точно разделена между воркерами
        if (rab width/e.target.width === workerCount) {
                stopMove();
        };
            } else if (e.target.n render == 1) {
        if (e.data.n sycles > e.data.n br+1) {
                stopMove();
        }
            };
           // Отправляем данные для следующего цикла расчёта концентрации
        sendMoveTask(
                        e.target, e.target.index worker,e.target.width,
                        e.data.c, e.target.w,
                        e.target.dump1, e.target.dump2, e.target.dump3,
                        e.target.n render,e.target.n sycles add,
                        e.data.n sycles,e.target.n br,e.target.no yes log
                    );
                break;
        default:
            break;
    }
}

function errorHandler(e) {
    log("error: " + e.message);
}
```

продолжение приложения А3

```
functionloadDemo() {
log("Исходные данные загружены!");
if (typeof(Worker) !== "undefined") {
        document.getElementById("status").innerHTML =
        "Ваш браузер поддерживает HTML5 WebWorkers";
document.getElementById("stop move").onclick = stopMove;
        document.getElementById("start move").onclick = startMove;
        document.getElementById("start move").disabled = true;
        document.getElementById("stop move").disabled = true;
}
}

/* Определение компонент rgba-цвета по значению концентрации примеси  */
function colour(zz) {
        const zz min = 2;
        const zz max = 52;
        const zz delta = zz max - zz min;
        const zz grdnt = zz delta / 8;
        const zz1 =zz min + zz grdnt;
        const zz2 = zz1 + zz grdnt;
        const zz3 = zz2 + zz grdnt;
        const zz4 = zz3 + zz grdnt;
        const zz5 = zz4 + zz grdnt;
        const zz6 = zz5 + zz grdnt;
        const zz7 = zz6 + zz grdnt;
        letr, g, b, a;
        // Добавляем градаций, чтобы увеличить нагрузку на рендеринг, как в вар.ES5
        if (zz>= 0 &&zz<zz min-1.75) {r=0; g=255; b=0; a=0;}; //Прозрачный#1
        if (zz>=zz min-1.75 &&zz<zz min-1.25){r=0; g=255; b=0;
a=50;};//Прозрачный#2
        if (zz>=zz min-1.25 &&zz<zz min-0.75){r=0; g=255; b=0;
a=100;};//Прозрачный#3
        //
        if (zz>=(zz min-0.75) && zz<(zz+1)) {r=0; g=255; b=0; a=255;}; // Зелёный
        if (zz>=(zz min+1) && zz<zz1) {r=139; g=0; b=255; a=255;}; // Фиолетовый
        if (zz>=zz1 && zz<zz2) {r = 0; g = 0; b = 255; a = 255;}; //Синий
        if (zz>=zz2 && zz<zz3) {r = 0; g = 255; b = 255; a = 255;}; //Голубой
        if (zz>=zz3 && zz<zz4) {r = 0; g = 80; b = 0; a = 255;}; // Тёмно-зелёный
        if (zz>=zz4 && zz<zz5) {r = 255; g = 255; b = 0; a = 255;}; // Жёлтый
        if (zz>=zz5 && zz<zz6) {r = 255; g = 165; b = 0; a = 255;}; // Оранжевый
        if (zz>=zz6 && zz<zz7) {r = 228; g = 0; b = 51; a = 255;}; // Красный
        if (zz>=zz7 && zz<zz max) {r=51; g=51; b=0; a=255;}; // Тёмно-грязно-зелёный
        if (zz>zz max) {r = 0; g = 0; b = 0; a = 255;}; // Чёрный
        return {r, g, b, a};
}

window.addEventListener("load", loadDemo, true);

</script>
</nobr></center>
</body>
</html>
```

продолжение приложения А3
Файл move.js

```
function in range(j, l j, r j) return (j >= l j && j <= r j);

function mixer(b,w,i,j) {
        let k = 0; let val = 0;
        for (let l = -1; l<=1; l=l+1) {
            for (let m = -1; m<=1; m=m+1) {
                        k = k + 1;
                        val = val + w[k-1] * b[i + l][j + m];
        }
        }
        returnval;
}
/***** Перемещение с рассеиванием за заданное кол-во циклов     *****/
/***** Кол-во циклов задано таким же, как у варианта с таймером *****/
/***** Два варианта: 1) с рендерингом и 2) без рендеринга       *****/
function boxMove(i worker,width,startX,c,w,dump1,dump2,dump3,n render,
        n sycles add,n sycles,n br) {
        var n = Math.sqrt(c.length);
        //var n = c.length;
        var data = {};
        // Инициализируем массив для клеточного автомата, если его ещё нет
        if (typeof b === 'undefined') {
                b=new Array (n);
                for (let i=0; i<n; i++) {b[i]=new Array(n)};
                for (let i = 0; i<n; i++) {
                        for (let j = 0; j<n; j++) {
                                b[i][j] = 0;
                        }
                }
        }
        // Ещё раз о вариантах эксперимента - их лишь два:
        //      1) с рендерингом на каждом цикле при n render==1 и
        //      2) без рендеринга при n render!=1 (карта - на последнем цикле)
        if (n render == 1) { // 1) вариант - клеточный автомат с рендерингом
                //      (общее кол-во циклов задано в основном потоке)
                if (n sycles<= n sycles add) {// вброс взвеси заданное кол-во раз
                  //Добавляем взвеси в точку если она попадает в интервал воркера
                b[dump1[0]][dump1[1]]=in range(dump1[1],startX,startX+width) ?
                    b[dump1[0]][dump1[1]] + dump1[2] : b[dump1[0]][dump1[1]];
                b[dump2[0]][dump2[1]]=in range(dump2[1],startX,startX+width) ?
                    b[dump2[0]][dump2[1]] + dump2[2] : b[dump2[0]][dump2[1]];
                b[dump3[0]][dump3[1]]=in range(dump3[1],startX,startX+width) ?
                    b[dump3[0]][dump3[1]] + dump3[2] : b[dump3[0]][dump3[1]];
                }
                for (let i = 1; i < n-1; i++) {
                        for (let j = startX+1; j < startX+width-1; j++) {
                                const index = j + i * n;
                                c[index] = mixer(b,w,i,j);
                        }
                }
                for (let i = 0; i < n; i++) {
                        for (let j = startX; j < startX+width+1; j++) {
                                const index = j + i * n;
                                b[i][j] = c[index];
                        }
                }
                n sycles = n sycles + 1;
                data = {c,n sycles};
                // Отправляем во внешний поток на рендеринг на каждом цикле
                returndata;
```

продолжение приложения А3

```
        } else { // 2) вариант - клеточный автомат без рендеринга
            //  (общее кол-во циклов задано в основном потоке)
            for (let i rab=1; i rab <= n br; i rab++) {
                if (i rab <= n sycles add) {
                //Вброс взвеси в точку если она попадает в интервал воркера
                b[dump1[0]][dump1[1]]=in range(dump1[1],startX,startX+width) ?
                        b[dump1[0]][dump1[1]] + dump1[2] : [dump1[0]][dump1[1]];
                b[dump2[0]][dump2[1]]=in range(dump2[1],startX,startX+width) ?
                        b[dump2[0]][dump2[1]] + dump2[2] : [dump2[0]][dump2[1]];
                b[dump3[0]][dump3[1]]=in range(dump3[1],startX,startX+width) ?
                        b[dump3[0]][dump3[1]] + dump3[2]: b[dump3[0]][dump3[1]];
                }
                for (let i = 1; i < n-1; i++) {
                        for (let j = startX+1; j < startX+width-1; j++) {
                                        const index = (j + i * n);
                                        c[index] = mixer(b,w,i,j);
                        }
                }
                for (let i = 0; i < n; i++) {
                        for (let j = startX; j < startX+width+1; j++) {
                                        const index = (j + i * n);
                                        b[i][j] = c[index];
                        }
                }
        }
        n sycles = n br;
        data = {c,n sycles};
            // Отправляем во внешний поток на рендеринг лишь на последнем цикле
        returndata;
        }
}
```

продолжение приложения А3

Файл moveWorker.js

```
importScripts("move.js");
function sendStatus(statusText) {
    postMessage({"type" : "status",
                "statusText" : statusText}
);
}
function messageHandler(e) {
    var messageType = e.data.type;
    switch (messageType) {
        case ("move"):
               if (e.data.no yes log) {
                       sendStatus("---> Воркер №" + (e.data.index worker+1)+
               " приступает к работе на интервале X: " +
               e.data.startX + "-"+(e.data.startX+e.data.width)+
               ", циклов: " + e.data.n sycles);
               }

               let n = Math.sqrt(e.data.c.length);
               const buffer = new ArrayBuffer(n*n*4);
               let c = new Float32Array(buffer);
               for (let i=0; i < e.data.c.length; i++) c[i]=e.data.c[i];
               var data =
               boxMove(e.data.index worker,e.data.width,e.data.startX,
               c,e.data.w,e.data.dump1,e.data.dump2,e.data.dump3,
            e.data.n render,e.data.n sycles add,e.data.n sycles,e.data.n br
               );
               postMessage({
               "type" : "progress",
                               "index worker" : e.data.index worker,
                               //"data" : data,
                               "width" : e.data.width,
                               "startX" : e.data.startX,
                               "c" : data.c,
                               "w" : e.data.w,
                               "dump1" : e.data.dump1,
                               "dump2" : e.data.dump2,
                               "dump3" : e.data.dump3,
                               "n render" : e.data.n render,
                               "n sycles add" : e.data.n sycles add,
                               "n sycles" : data.n sycles,
                               "n br" : e.data.n br
               }, [buffer]);
        if (e.data.no yes log) {
                       sendStatus("<=== Воркер №" + (e.data.index worker+1)+
               " отработал на интервале X: " +
                       e.data.startX + "-" + (e.data.width+e.data.startX) +
               ", циклов: " + e.data.n sycles  +" !!! Отладка !!! " );
        }
        break;
        default: sendStatus("Worker получил сообщение: " + e.data);
    }
}

addEventListener("message", messageHandler, true);
```