ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΕΠΙΤΑΧΥΝΣΗ 3D ΓΡΑΦΙΚΩΝ ΣΕ WEBGL ΚΑΙ THREE.JS ΜΕ ΤΗΝ ΧΡΗΣΗ WEB WORKERS

Διπλωματική Εργασία

του

Γκουτζαμάνη Βασίλη

Θεσσαλονίκη, Μάιος 2023

ΕΠΙΤΑΧΥΝΣΗ 3D ΓΡΑΦΙΚΩΝ ΣΕ WEBGL ΚΑΙ THREE.JS ΜΕ ΤΗΝ ΧΡΗΣΗ WEB WORKERS

Γκουτζαμάνης Βασίλειος

Πτυχίο Εφαρμοσμένης Πληροφορικής, Πανεπιστήμιο Μακεδονίας, 2019

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής
Κασκάλης Θεόδωρος

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την ηη/μμ/εεεε

Ονοματεπώνυμο 1	Ονοματεπώνυμο 2	Ονοματεπώνυμο 3
.....	.....	.....

Γκουτζαμάνης Βασίλειος

.....

Περίληψη

Στην παρούσα εργασία θα μελετηθούν οι τρόποι με τους οποίους μπορούμε να επιταχύνουμε την απεικόνιση 3D γραφικών με την χρήση web workers.

Αρχικά, θα παρουσιαστούν τα βήματα τα οποία εκτελεί ένας browser για την απεικόνιση περιεχομένου σε μια σελίδα, καθώς και ο βέλτιστος αριθμός FPS, στα οποία θα πρέπει να εκτελείται μια σελίδα πολυμεσικού περιεχομένου. Έπειτα, θα παρουσιαστούν οι τεχνολογίες WebGL και η βιβλιοθήκη three.js, που χρησιμοποιούνται για την απεικόνιση γραφικών σε web εφαρμογές. Θα γίνει επίσης, μια εισαγωγή στους web workers, στον τρόπο με τον οποίο λειτουργούν και επικοινωνούν, και θα απαριθμηθούν τα πλεονεκτήματα και μειονεκτήματά τους.

Στη συνέχεια, θα παρουσιαστούν τρία παραδείγματα εφαρμογών, με τις περιπτώσεις χρήσης και μη web workers, εντοπίζοντας τα σημεία εκείνα στα οποία μπορούν να αξιοποιηθούν. Σε κάθε παρουσίαση, θα προβληθούν τα αντίστοιχα τμήματα κώδικα που αλλάζουν για να υποστηρίξουν την χρήση των web workers. Ακολουθώντας τις υλοποιήσεις, θα προβληθούν τα αποτελέσματα των εκτελέσεων των εφαρμογών, με την παρουσίαση πινάκων και διαγραμμάτων.

Τέλος, θα εξαχθούν συμπεράσματα αναφορικά με την χρήση των web workers σε εφαρμογές απεικόνισης 3D γραφικών και θα προταθούν περιπτώσεις επέκτασης της εργασίας.

Λέξεις-κλειδιά: web worker, 3D graphics, webGL, three.js, FPS, επιτάχυνση γραφικών, GPU

Abstract

The subject of this thesis focuses on the study of web workers and the ways that can help in accelerating 3D graphics rendering in a web application.

First, the browser rendering pipeline will be presented, along with the optimal number of FPS that a multimedia web application should have. Then, WebGL and the three.js library will be presented, as the graphics APIs that will be used for rendering. Furthermore, a brief introduction of web workers and the way the work and communicate will be shown, along with their advantages and drawbacks.

Moving on, there will be a demonstration of three cases of web applications with 3D graphics, each of them having a serial and a parallel implementation using web workers. The purpose of those applications is to identify those parts which could be transferred and executed inside a worker. Next, the benchmark results of each application will be presented with the use of tables and diagrams.

Finally, the conclusions about the use of web workers in accelerating 3D graphics are presented and a few suggestions are provided to extend the current thesis.

Keywords: web worker, 3D graphics, webGL, three.js, FPS, επιτάχυνση γραφικών, GPU

Πρόλογος - Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον κ. Κασκάλη, με τον οποίο συνεργαστήκαμε για την εκπόνηση της διπλωματικής εργασίας, και ο οποίος με βοήθησε σε ότι ζήτημα αντιμετώπισα και μου έλυσε όποιες απορίες και προβληματισμούς μου δημιουργούνταν κατά την διάρκειά της. Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου, η οποία μου στάθηκε και με στήριξε στην προσπάθεια αυτή.

Περιεχόμενα

Κατάλογος Εικόνων

Κατάλογος Πινάκων

1. Εισαγωγή

1.1 Πρόβλημα – Σημαντικότητα του θέματος

Από την εφεύρεσή του και την εισαγωγή του στον κόσμο της πληροφορικής, ο παγκόσμιος ιστός αποτέλεσε μια τεχνολογία ορόσημο στην επικοινωνία και μεταφορά δεδομένων μεταξύ υπολογιστών. Παρόλο που το υλισμικό (hardware) περιόριζε τις δυνατότητες των web εφαρμογών που αναπτύσσονταν, η σταδιακή αύξηση της υπολογιστικής ισχύος σε συνδυασμό με τις δυνατότητες προσβασιμότητας (accessibility) και φορητότητας (portability) που εγγενώς παρέχουν οι web εφαρμογές, ώθησαν τους προγραμματιστές να προσθέσουν νέα χαρακτηριστικά και λειτουργίες που μέχρι τότε μόνο native εφαρμογές μπορούσαν να έχουν. Μια από αυτές τις δυνατότητες είναι και η απεικόνιση 3D γραφικών.

Η εμφάνιση 3D γραφικών πραγματοποιείται από τον browser, αξιοποιώντας την μονάδα επεξεργασίας γραφικών (GPU) του Η/Υ, ώστε να επιταχύνεται ο υπολογισμός και η σχεδίασή τους. Παρόλο που η χρήση της GPU πλέον υφίσταται σε web εφαρμογές, η μονονηματική (single threaded) αρχιτεκτονική εκτέλεσης της Javascript στην σελίδα, αποτελεί κύριο πρόβλημα σε πολλές εμπορικές και μη εφαρμογές, επηρεάζοντας αρνητικά την απόδοσή τους και την συνολική εμπειρία χρήστη.

Το παραπάνω πρόβλημα έρχονται να αντιμετωπίσουν οι Web Workers, οι οποίοι επιτρέπουν την παράλληλη (parallel) και συντρέχουσα (concurrent) εκτέλεση τμημάτων κώδικα Javascript, με σκοπό να ελευθερώσουν το κύριο νήμα (main thread) με τον διαμοιρασμό των δεδομένων και εργασιών προς εκτέλεση, κάνοντας αποδοτικότερη χρήση των πυρήνων της CPU, και οδηγώντας σε υψηλότερη απόδοση της web εφαρμογής και βελτίωση της εμπειρίας χρήστη.

1.2 Σκοπός – Στόχοι

Η εργασία αυτή αποσκοπεί στην ανάδειξη των πλεονεκτημάτων που αποκτούν οι web εφαρμογές με απεικόνιση 3D γραφικών από την αξιοποίηση των Web Workers, τόσο σε επίπεδο ταχύτητας και απόδοσης στους υπολογισμούς της επιχειρηματικής λογικής, τα αποτελέσματα της οποίας θα χρησιμοποιηθούν για τα γραφικά που πρόκειται να σχεδιαστούν, όσο και στην καλύτερη απόκριση της εφαρμογής και συνεπώς στην εμπειρία του χρήστη. Ειδικότερα, οι βελτιώσεις πάνω στις οποίες θα εστιάσουμε θα αφορούν:

- Τον καταμερισμό δεδομένων σε έναν αριθμό από Web Workers, με στόχο την ταχύτερη εκτέλεση, συγκέντρωση και απεικόνιση των αποτελεσμάτων, χρησιμοποιώντας την ίδια ακολουθία εντολών σε κάθε Web Worker.
- Τον συγχρονισμό εκτελέσεων της ίδιας ακολουθίας εντολών από Web Workers, με σκοπό την επίτευξη υψηλότερων τιμών **FPS**, σε περίπτωση εφαρμογών στις οποίες οι υπολογισμοί διαδέχονται ο ένας τον άλλον και ενημερώνουν τα γραφικά της εφαρμογής με τα αποτελέσματα του εκάστοτε υπολογισμού.
- Την μεταφορά του υπολογιστικού όγκου της εφαρμογής σε Web Worker, έτσι ώστε το κύριο νήμα (main thread) να μην επηρεάζεται από τυχόν καθυστερήσεις στην εκτέλεσή του, εξασφαλίζοντας έτσι υψηλή απόκριση της εφαρμογής με την αλληλεπίδραση του χρήστη και οδηγώντας σε βελτιωμένη εμπειρία (UX).

1.3 Συνεισφορά

Η συνεισφορά της εργασίας αυτής αποτυπώνεται στην ολοκληρωμένη παρουσίαση των επιμέρους στοιχείων που καθορίζουν την εκτέλεση και απόδοση μιας web εφαρμογής 3D γραφικών, την αξιοποίηση των στοιχείων εκείνων που μπορούν να επιφέρουν καλύτερα αποτελέσματα στην ταχύτητα και απόκριση της εφαρμογής και στην ανασκόπηση και παρουσίαση αποτελεσμάτων προηγούμενων ερευνών πάνω στους Web Workers. Τέλος, το σημαντικότερο τμήμα της εργασίας αυτής είναι η ανάπτυξη εφαρμογών, που ανταποκρίνονται στις περιπτώσεις βελτίωσης που αναφέρθηκαν στην προηγούμενη ενότητα, και στην συγκριτική αξιολόγηση (benchmarking) σε ένα πλήθος φυλλομετρητών (browsers), με περιπτώσεις χρήσης και μη χρήσης Web Workers για την εκάστοτε εφαρμογή.

1.4 Βασική Ορολογία

Thread: Ένα τμήμα μιας διεργασίας που εκτελείται αυτόνομο από το σύνολο της διεργασίας.
Worker: Ένα νήμα το οποίο εκτελεί μια εργασία στο υπόβαθρο μιας εφαρμογής.
Browser: Πρόγραμμα περιήγησης στον παγκόσμιο ιστό, ο φυλλομετρητής.
FPS (frames per second): Ο ρυθμός με τον οποίο ανανεώνονται οι εικόνες ενός πολυμεσικού περιεχομένου (βίντεο).

API (application programming interface): Είναι το σύνολο ιδιοτήτων, συναρτήσεων και μεθόδων που καθιστά διαθέσιμο ο πάροχος μιας υπηρεσίας, ώστε να είναι δυνατόν να χρησιμοποιηθεί από προγραμματιστές.

Pipeline: Στο πλαίσιο της απεικόνισης γραφικών, είναι το σύνολο των βημάτων που ακολουθείται για την απεικόνιση και ενημέρωσή τους.

Script: Αρχείο ή τμήμα κώδικα που μπορεί να εκτελεστεί από έναν browser.

1.5 Διάρθρωση της μελέτης

Στο 1ο κεφάλαιο έγινε μια εισαγωγή αναφορικά με το πρόβλημα και τις λύσεις που θα αναπτύξουμε χρησιμοποιώντας τους Web Workers.

Στο 2ο κεφάλαιο θα παρουσιαστούν ο τρόπος με τον οποίο εκτελεί ένας browser μια web εφαρμογή, καθώς τις τεχνολογίες που χρησιμοποιούνται για την απεικόνιση 3D γραφικών. Έπειτα θα γίνει αναφορά στους Web Workers, στις κατηγορίες τις οποίες χωρίζονται ανάλογα με τον σκοπό που εξυπηρετούν και στον τρόπο με τον οποίο λειτουργούν και επικοινωνούν τόσο με το κύριο νήμα της εφαρμογής όσο και μεταξύ τους. Στο τέλος του 2ου κεφαλαίου, θα παρουσιαστούν οι τρόποι και τα αποτελέσματα προηγούμενων εργασιών που χρησιμοποίησαν Web Worker σε εφαρμογές με 3D γραφικά.

Στο 3ο κεφάλαιο θα προχωρήσουμε στην παρουσίαση τριών εφαρμογών, κάθε μια από τις οποίες επηρεάζεται από τα θέματα της προηγούμενης ενότητας τα οποία καλούμαστε να αντιμετωπίσουμε. Θα δείξουμε λεπτομερώς τις υλοποιήσεις και τα σημεία στα οποία μπορούν να τροποποιηθούν ώστε να χρησιμοποιηθούν Web Workers, παρουσιάζοντας τμήματα κώδικα και βελτιώσεις. Στο τέλος της κάθε εφαρμογής θα παρουσιάσουμε τα αποτελέσματα των παραπάνω υλοποιήσεων σε ένα πλήθος browsers, συγκρίνοντας τους χρόνους της σειριακής υλοποίησης της εκάστοτε εφαρμογής, με τις υλοποιήσεις που χρησιμοποιούν Web Workers.

Στο 4ο κεφάλαιο θα συνοψίσουμε τα αποτελέσματα της ακόλουθης εργασίας και θα προτείνουμε τρόπους βελτίωσης.

2. Βιβλιογραφική Επισκόπηση – Θεωρητικό Υπόβαθρο

Παρόλο που η χρησιμότητα των Web Workers μπορεί να γίνει αντιληπτή παραθέτοντας αποτελέσματα μετρήσεων συγκριτικής αξιολόγησης με τις αντίστοιχες σειριακές web εφαρμογές από την παρούσα εργασία και από τις βιβλιογραφικές αναφορές, η ολοκληρωμένη παρουσίασή τους μαζί με τους τρόπους επικοινωνίας τους, η κατανόηση της διαδικασίας απεικόνισης και ενημέρωσης των στοιχείων μιας εφαρμογής από έναν browser, αλλά και η αναφορά στις υπάρχουσες τεχνολογίες αναπαράστασης γραφικών και τις δομές που παρέχουν ώστε να μπορούν να αξιοποιηθούν από έναν Web Worker, συμβάλλουν σημαντικά στην ικανότητα ανίχνευσης των σημείων εκείνων που μπορούν να βελτιωθούν.

2.1 Ροή απεικόνισης στοιχείων στον browser (Browser Rendering Pipeline)

Με την συνεχή προσθήκη νέων χαρακτηριστικών στους browsers και με την αυξανόμενη χρήση των κινητών συσκευών για πλοήγηση στο διαδίκτυο, η ροή απεικόνισης στοιχείων (rendering pipeline) είναι μια διαδικασία η οποία συνεχώς εξελίσσεται προκειμένου να μπορεί να διαχειρίζεται όσον το δυνατόν καλύτερα τους πόρους της εκάστοτε συσκευής και να μπορεί να προσφέρει όλο και καλύτερη εμπειρία στον χρήστη.

Η επίτευξη καλύτερης διαχείρισης απεικόνισης στοιχείων, τόσο από άποψη χρήσης υπολογιστικών πόρων όσο και ταχύτητας, έχει οδηγήσει στην κατασκευή ξεχωριστών rendering engines για κάθε ευρέως χρησιμοποιούμενο browser, από την εταιρεία ή τον οργανισμό που το αναπτύσσει. Παρόλο που δημοφιλείς engines όπως ο Webkit της Apple [1], ο Gecko της Mozilla [2] και ο Blink της Google [3], έχουν τις δικές τους παραλλαγές στον τρόπο φόρτωσης και απεικόνισης περιεχομένου σε μια σελίδα, τα βασικά βήματα τα οποία ακολουθούν είναι κοινά για όλους.

Εικόνα 1: Τα βήματα εκτέλεσης ενός rendering engine για την απεικόνιση μιας σελίδας

Στην παραπάνω εικόνα φαίνεται η σειρά των βημάτων [4] με τα οποία ένας browser εμφανίζει και ενημερώνει τα στοιχεία μιας σελίδας. Αναλυτικότερα, για το κάθε βήμα έχουμε:

1. **Javascript**: Όπως υποδεικνύει και το όνομα, αφορά την εκτέλεση του κώδικα Javascript της σελίδας, από το αποτέλεσμα του οποίου εξαρτάται η μετάβαση ή μη, στο επόμενο βήμα.

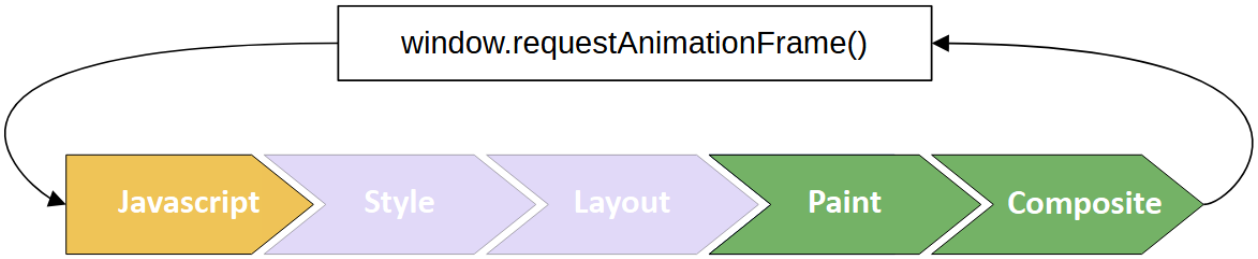
2. **Style**: Έχει να κάνει με την ενημέρωση του styling των στοιχείων, βάσει των κανόνων που έχουν οριστεί με CSS. Παραδείγματος χάρη, εάν μέσω κώδικα αλλάξει η κλάση ενός στοιχείου της σελίδας, τότε σε αυτό το βήμα θα γίνει αναζήτηση της κλάσης αυτής στα styles της σελίδας, και εφόσον υπάρχει, θα ενημερώσει το στοιχείο με τους κανόνες της συγκεκριμένης κλάσης.

3. **Layout**: Έρχεται μετά το styling των στοιχείων και αφορά την αναδιάταξη της σελίδας μετά από αλλαγές σε ιδιότητες του Box model και των width και height.

4. **Paint**: Γνωρίζοντας τους κανόνες που πρόκειται να εφαρμοστούν σε κάθε στοιχείο της σελίδας, στο βήμα αυτό πραγματοποιείται ο χρωματισμός όλων των pixel του κάθε στοιχείου, λαμβάνοντας υπόψη όλους τους κανόνες στους οποίους υπάρχει πληροφορία αναφορικά με χρώμα, πχ. color, background-color, text-shadow, κτλ. Ο χρωματισμός γίνεται σε πολλαπλές στρώσεις, που ονομάζονται layers.

5. **Composite**: Το τελευταίο στάδιο της απεικόνισης, στο οποίο γίνεται ταξινόμηση των στοιχείων σχετικά με την σειρά εμφάνισής τους.

Στις περιπτώσεις εφαρμογών που θα εξετάσουμε, δεν θα μας απασχολήσει το styling των στοιχείων, καθώς το rendering θα γίνεται σε canvas, συνεπώς τα στάδια της απεικόνισης που μας ενδιαφέρουν είναι τα ακόλουθα:



Εικόνα 2: Κύκλος ζωής (lifecycle) μιας εφαρμογής με 3D γραφικά.

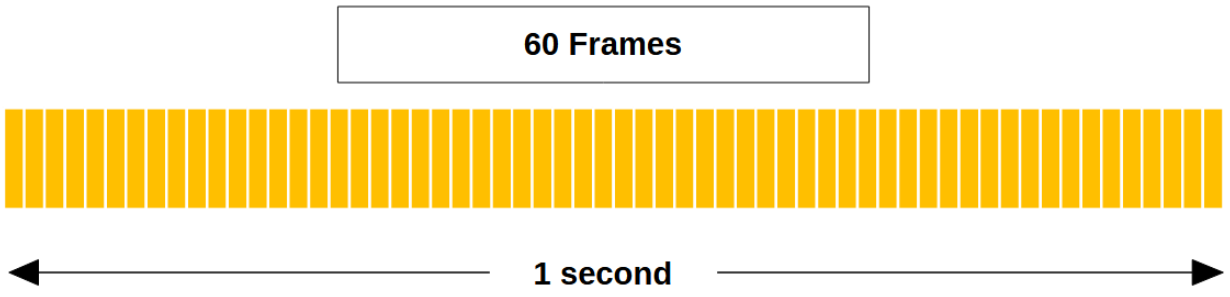
Η συνάρτηση **requestAnimationFrame(callback)** [5], εκτελεί την συνάρτηση **callback** που δέχεται ως παράμετρο, κάθε φορά που καλείται. Μόλις ολοκληρώσει την εκτέλεσή της, ο

browser θα εκτελέσει εκ νέου τα στάδια Paint και Composite για να ανανεωθούν τα pixels της οθόνης. Η χρήση της συνάρτησης συνίσταται στο γεγονός ότι στις εφαρμογές που αφορούν απεικόνιση γραφικών, χρειαζόμαστε έναν μηχανισμό ο οποίος θα εκτελείται και θα ενημερώνει συνέχεια τα pixels της οθόνης. Το ακόλουθο τμήμα κώδικα αποτελεί χαρακτηριστικό παράδειγμα της χρήσης της requestAnimationFrame(), μέσα στην συνάρτηση render(), οδηγώντας έτσι την τελευταία να εκτελείται 60 φορές το δευτερόλεπτο (60 FPS) .

```
/* Rendering loop to keep updating circles positions. */
function render() {
      /*
       * Code that keeps updating the canvas.
       */
        window.requestAnimationFrame(render);
}
 render();
```

2.2 Βέλτιστο Framerate

Ένα επίσης πολυσυζητημένο θέμα που αφορά την προβολή πολυμεσικού περιεχομένου και παίζει καθοριστικό ρόλο στο UX μιας εφαρμογής, είναι η βέλτιστη τιμή FPS που θα πρέπει να αναπαράγεται. Έχει παρατηρηθεί [6] πως η επίτευξη της βέλτιστης εμπειρίας χρήστη επιτυγχάνεται σε τιμές κοντά στα 60 FPS. Αναλυτικότερα, περιεχόμενο του οποίου ο ρυθμός ανανέωσης είναι μεγαλύτερος ή ίσος από 30 FPS, είναι αποδεκτό σαν μια ικανοποιητική εμπειρία στον τελικό χρήστη.



Εικόνα 3: Αναπαράσταση 60 frames σε διάρκεια ενός δευτερολέπτου.

Όπως αναφέραμε στην προηγούμενη ενότητα, η συνάρτηση requestAnimationFrame() εκτελεί την callback συνάρτηση που δέχεται σαν παράμετρο 60 φορές το δευτερόλεπτο. Γνωρίζοντας ότι:

1 second = 1000 ms

ο μέγιστος χρόνος προς εκτέλεση της callback συνάρτησης πριν γίνει η επόμενη ανανέωση των pixels στην οθόνη είναι:

1000 ms / 60 fps ~= 16.7 ms

Αυτό πρακτικά σημαίνει ότι για να μπορέσουμε να πετύχουμε την υψηλότερη τιμή FPS που είναι το 60, θα πρέπει ο κώδικας της callback να εκτελεστεί το πολύ σε **16.7 ms**. Σε περίπτωση που ο κώδικας χρειαστεί περισσότερο χρόνο για να εκτελεστεί, η ανανέωση των pixels της οθόνης θα πραγματοποιηθεί χωρίς να ληφθούν υπόψη οι καινούργιες αλλαγές την σωστή στιγμή, μειώνοντας έτσι τα FPS και οδηγώντας σε λιγότερο ομαλό rendering (jank).

2.3 Διεπαφές γραφικών για web εφαρμογές

Μέχρι το 2007, η σχεδίαση και απεικόνιση γραφικών σε μια web εφαρμογή μπορούσε να γίνει είτε με την χρήση του canvas [7], ενός HTML element και του API που παρέχει για σχεδίαση γραφικών, είτε με την χρήση SVG (Scalable Vector Graphics) [8], ένα διαφορετικό format απεικόνισης 2D γραφικών, μέσω κανόνων σε XML μορφή. Παρόλο που και οι δυο τεχνολογίες είχαν ένα πλήθος λειτουργιών, παρουσίαζαν 2 βασικά μειονεκτήματα:

- Δεν υποστήριζαν την σχεδίαση 3D γραφικών.
- Δεν αξιοποιούσαν την GPU του συστήματος για την ταχύτερη σχεδίαση γραφικών.

Αν και η σχεδίαση 3D γραφικών ήταν δυνατή, χρησιμοποιώντας εξωτερικές διεπαφές όπως το Stage3D της Adobe [9] και το Silverlight της Microsoft [10], έγινε επιτακτική η ανάγκη για την ενσωμάτωση ενός ενιαίου API που θα μπορούσε να δώσει την δυνατότητα στους browsers να χρησιμοποιούν την GPU χωρίς την παρέμβαση third party εφαρμογών.

Στις επόμενες δυο ενότητες, θα παρουσιάσουμε την WebGL, ένα ενιαίο API που αναπτύχθηκε ώστε να είναι δυνατή και αποδοτική η σχεδίαση 3D γραφικών σε μια web εφαρμογή

με την χρήση της GPU, καθώς και την Three.js, μια βιβλιοθήκη η οποία στηρίχθηκε στην WebGL ώστε να διευκολύνει την ανάπτυξη εφαρμογών με 3D γραφικά.

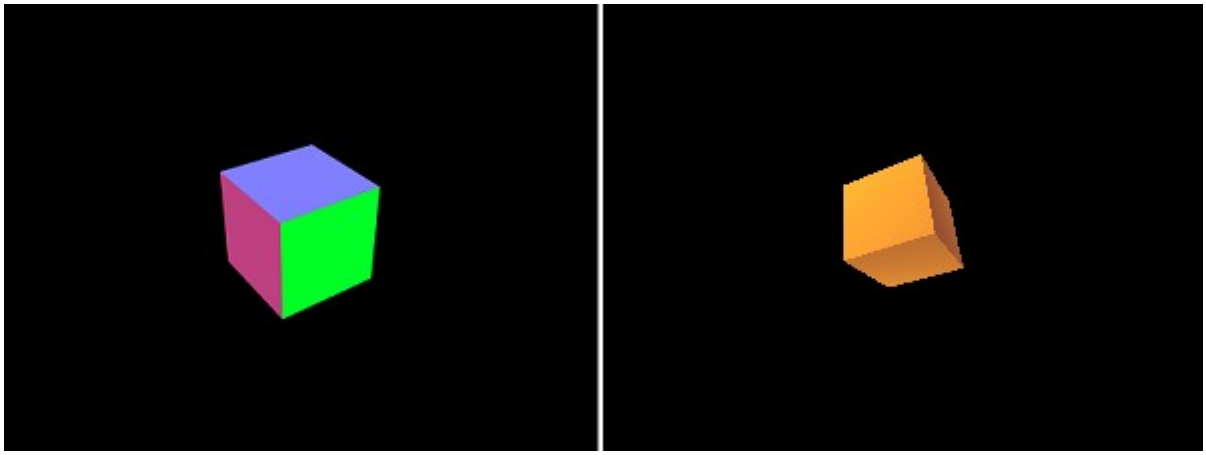
2.3.1 WebGL

Η WebGL είναι ένα Javascript API το οποίο δημιουργήθηκε για την σχεδίαση 2D και 3D γραφικών. Βασίζεται στο διεπαφή OpenGL ES 2.0, ένα υποσύνολο εντολών της OpenGL[11] για mobile και ενσωματωμένα (embedded) συστήματα, έτσι ώστε να είναι μπορεί να υποστηριχθεί από όλους τους browsers και να αξιοποιεί την GPU του εκάστοτε συστήματος, χρησιμοποιώντας το HTML στοιχείο του canvas για την αποτύπωση των γραφικών στην σελίδα.

Εικόνα 4: Λογότυπο της WebGL

Η ιδέα εισαγωγής ενός ενιαίου API για την χρήση σχεδίασης 3D γραφικών ξεκίνησε το 2006 με δοκιμές από τον οργανισμό της Mozilla. 3 χρόνια αργότερα, το 2009, η κοινοπραξία Khronos Group, που απαρτίζεται από μεγάλες εταιρείες όπως η Apple και η Google, ίδρυσε το WebGL Working Group με την συμμετοχή εταιρειών και οργανισμών κατασκευής λογισμικού φυλλομετρητών, όπως η Mozilla και η Opera, για να αναπτύξουν την νέα τεχνολογία. Στις 11 Μαρτίου του 2011 υπήρξε το πρώτο release της WebGL, για να ακολουθήσει 6 χρόνια αργότερα το επόμενο stable release της WebGL (WebGL 2.0).

Με την εισαγωγή της WebGL, έγινε εφικτό το rendering 3D γραφικών στην μεριά του πελάτη (client – side rendering) μεταφέροντας το workload στην GPU του συστήματος. Αυτό, σε συνδυασμό με την υποστήριξη της OpenGL, έδωσε στους προγραμματιστές ένα ολοκληρωμένο εργαλείο, του οποίου η ευελιξία και η υποστήριξη από τους περισσότερους browsers, τους επέτρεπε να μπορούν να μεταφέρουν σε μια web εφαρμογή, περιεχόμενο το οποίο μπορούσε να εκτελεστεί μόνο σε native εφαρμογές [12].



Εικόνα 5: Δύο κύβοι σχεδιασμένοι με την WebGL, με τον κύβο στα αριστερά να φέρει διαφορετικά χρώματα στις επιφάνειες του και τον κύβο στα δεξιά να έχει σκίαση.

Μπορούμε να αντιληφθούμε ότι μια τεχνολογία όπως η WebGL έρχεται με αρκετά πλεονεκτήματα έναντι των υπολοίπων λύσεων πριν την εισαγωγή της. Τα πλεονεκτήματα αυτά είναι:

- Η χρήση του API μπορεί να γίνει χωρίς την χρήση εξωτερικών πρόσθετων (plugins), καθώς είναι πλέον ενσωματωμένο και υποστηρίζεται από όλους τους browsers.
- Μεγαλύτερη απόδοση των εφαρμογών λόγω της χρήσης της GPU του συστήματος που εκτελεί την web εφαρμογή.
- Οι εφαρμογές που είναι γραμμένες σε WebGL δεν απαιτούν την εγκατάστασή τους στο εκάστοτε σύστημα, καθώς το ανέβασμα τους σε διακομιστή (server) τις καθιστά προσβάσιμες από όλους τους χρήστες του διαδικτύου.

Παρόλα αυτά, κάθε τεχνολογία συνοδεύεται και από τα μειονεκτήματά της. Συγκεκριμένα, οι θέσεις στις οποίοι οι WebGL υστερεί είναι:

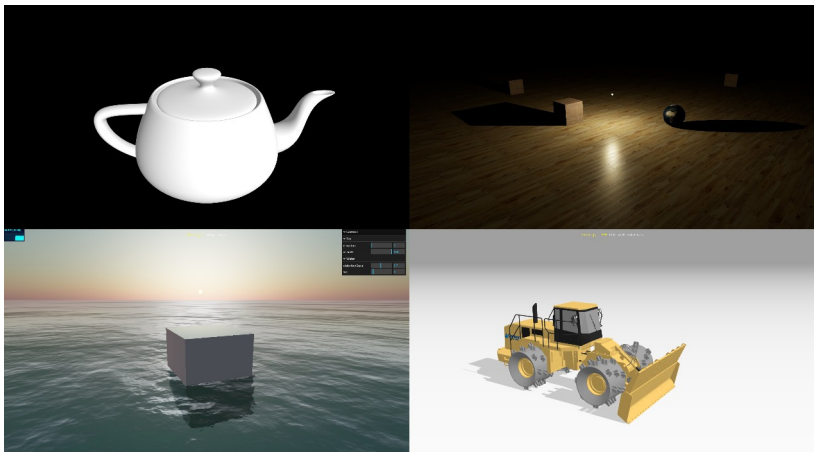
- Η φλυαρία (verbosity) που την συνοδεύει, το οποίο είναι λογικό αν αναλογιστεί κανείς πως σκοπός της είναι να προσφέρει στον προγραμματιστή μια πλούσια και ευέλικτη διεπαφή για την υλοποίηση οποιουδήποτε χαρακτηριστικού απαιτείται από την εφαρμογή. Αυτό σημαίνει ότι το μέγεθος των προγραμμάτων WebGL είναι αρκετά μεγάλο ακόμη και για απλά προγράμματα, καθώς απαιτούνται αρκετές εντολές για την υλοποίησή τους.
- Η δυσκολία στην εκμάθησή της, ειδικά από προγραμματιστές ιστού που προσπαθούν να χρησιμοποιήσουν το API της, χωρίς να έχουν προηγούμενη εμπειρία με προγραμματισμό γραφικών. Αυτό είναι απόρροια του προηγούμενου μειονεκτήματος, καθώς ο όγκος των

εντολών και των ιδεών (concepts) πάνω στις οποίες βασίζεται η WebGL, απαιτούν χρόνο για την εξοικείωση με αυτές.

Προκειμένου να περιοριστεί ο αριθμός των εντολών και να γίνει ομαλότερη η καμπύλη μάθησης της WebGL, έχουν αναπτυχθεί αρκετές βιβλιοθήκες, οι οποίες αποκρύπτουν όλες τις λεπτομέρειες και εστιάζουν στα χαρακτηριστικά και στις λειτουργικότητες που απαιτούνται για τις 3D εφαρμογές στο διαδίκτυο. Εμείς θα εστιάσουμε στην βιβλιοθήκη **three.js**, την οποία και θα χρησιμοποιήσουμε για τις εφαρμογές που πρόκειται να παρουσιάσουμε στο 3ο κεφάλαιο.

2.3.2 three.js

Η **three.js** είναι μια βιβλιοθήκη της Javascript που έχει σκοπό την ευκολότερη και ταχύτερη δημιουργία και απεικόνιση 3D γραφικών και animations σε web εφαρμογές. Βασίζεται στην WebGL έτσι ώστε να αξιοποιείται η GPU για την σχεδίαση των γραφικών και να αποφευχθεί η εξάρτηση από εξωτερικά πρόσθετα (plugins) για την λειτουργία της. Σύμφωνα με τον δημιουργό της, η βιβλιοθήκη αναπτύχθηκε πρώτα στην γλώσσα προγραμματισμού Actionscript, προϊόν της Adobe, καθώς ήταν αρκετά δημοφιλής γλώσσα για προγραμματισμού πολυμεσικού περιεχομένου για web εφαρμογές. Με την εξέλιξη του Google Chrome όμως και την βελτίωση στην ταχύτητα εκτέλεσης της Javascript, σε συνδυασμό με την ευκολία που παρέχει στο να μπορεί να εκτελείται εγγενώς (out of the box) από όλους τους browsers, η εξέλιξη της three.js συνεχίστηκε στην Javascript μέχρι και σήμερα [13].

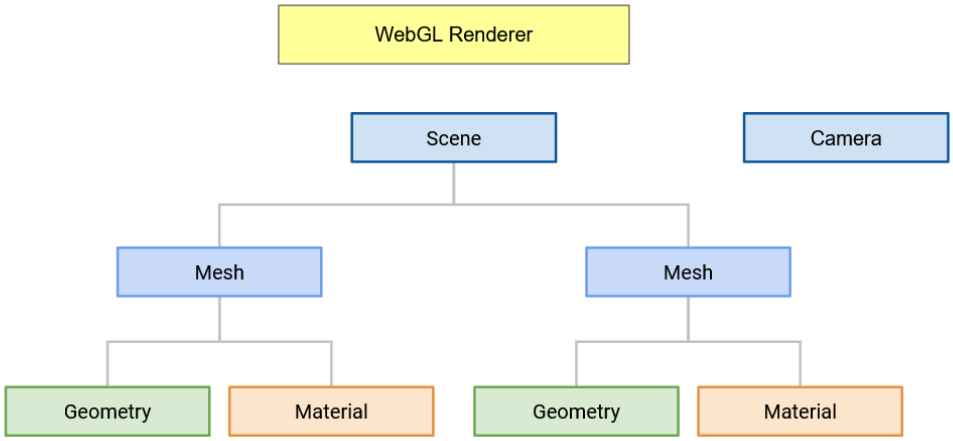


Εικόνα 6: Προβολή μερικών χαρακτηριστικών της Three.js.

Για να είναι δυνατή η εύκολη ανάπτυξη εφαρμογών με την βιβλιοθήκη αυτή, η three.js προσφέρει αρκετές έτοιμες δομές και λειτουργικότητες τις οποίες μπορούν να χρησιμοποιήσουν οι προγραμματιστές μέσω του API τους. Ορισμένες από αυτές είναι [14]:

- **Skeletal animation**: Λειτουργίες για την φόρτωση και διαχείριση 3D animation.
- **Camera**: Μια έτοιμη δομή η οποία μας απαλλάσσει από τους πολύπλοκους μετασχηματισμούς που απαιτούνται για αλλαγή θέσης και γωνίας απεικόνισης των αντικειμένων από την πλευρά του χρήστη, παρέχοντάς μας έτοιμες μεθόδους για τις εργασίες αυτές.
- **Geometries**: Έτοιμες γεωμετρίες σχημάτων (κύβος, σφαίρα, κτλ).
- **Materials**: Δομές που συνδυάζονται με τα geometries και προσθέτουν υφή και δυναμική συμπεριφορά στην επιφάνειά τους (σκίαση με την ύπαρξη φωτός, λεία επιφάνεια, κτλ).
- **FontLoaders**: Μας επιτρέπουν την φόρτωση γραμματοσειρών και την 3D απεικόνιση κειμένου.
- **Instancing**: Λειτουργικότητα με την οποία μπορεί να γίνει render ένας μεγάλος αριθμός ίδιων αντικειμένων με μικρή επιβάρυνση της GPU.
- **Lighting**: Έτοιμες δομές για τον φωτισμό μιας 3D σκηνής.
- **Loaders**: Μας επιτρέπουν να κάνουμε import 3D μοντέλα στην εφαρμογή μας.
- **Shaders**: Είναι περισσότερο για προχωρημένους χρήστες και μας δίνουν την δυνατότητα να χρησιμοποιήσουμε το API της **GLSL** [15], μιας γλώσσας προγραμματισμού για shaders, για την δημιουργία διαφόρων εφφέ στην εφαρμογή μας.
- Λειτουργίες **import** και **export** για διάφορα προγράμματα επεξεργασίας 3D μοντέλων όπως το Blender.

Στο παρακάτω σχήμα, μπορούμε να δούμε τα βασικά αντικείμενα μιας εφαρμογής της three.js και τον τρόπο που συνδυάζονται και αλληλεπιδρούν μεταξύ τους:



Εικόνα 7: Διάγραμμα με την μορφή και την ιεραρχία αντικειμένων μιας εφαρμογής σε three.js.

Μια εφαρμογή της three.js αποτελείται από σκηνές (**Scenes**), μέσα στις οποίες μπορούμε να προσθέσουμε Meshes. Το **Mesh** είναι μια αφηρημένη έννοια ενός αντικειμένου με ορισμένη γεωμετρία και εμφάνιση, ιδιότητες οι οποίες μοντελοποιούνται με τις κλάσεις **Geometry** και **Material** αντίστοιχα. Συνεπώς η διαχείριση ενός Mesh, γίνεται έμμεσα από την αλλαγή των τιμών των ιδιοτήτων των αντικειμένων Geometry και Material.

Ο έλεγχος του πεδίου προβολής (projection) του εκάστοτε Scene προς τον χρήστη, γίνεται με την βοήθεια της κάμερας που μοντελοποιείται από την κλάση **Camera**, με τις δύο βασικές υποκλάσεις της να είναι:

- **Perspective camera**, η οποία χρησιμοποιείται για την απεικόνιση 3D σκηνών, καθώς έχει σχεδιαστεί έτσι ώστε να αποτυπώνει το βάθος και την προβολή ενός αντικειμένου με παρόμοιο τρόπο που αντιλαμβάνεται το ανθρώπινο μάτι.
- **Orthographic camera**, για την απεικόνιση 2D σκηνών, λόγω της ιδιότητάς της να αποτυπώνει το κάθε αντικείμενο με σταθερό μέγεθος, χωρίς να λαμβάνει υπόψη την απόστασή του από την κάμερα.

Τέλος, η κάθε 3D σκηνή θα σχεδιαστεί μέσω του αντικειμένου **WebGL Renderer**, το οποίο αποτελεί το πλαίσιο της 3D εφαρμογής και χρησιμοποιεί την WebGL για την επεξεργασία και απεικόνιση των γραφικών.

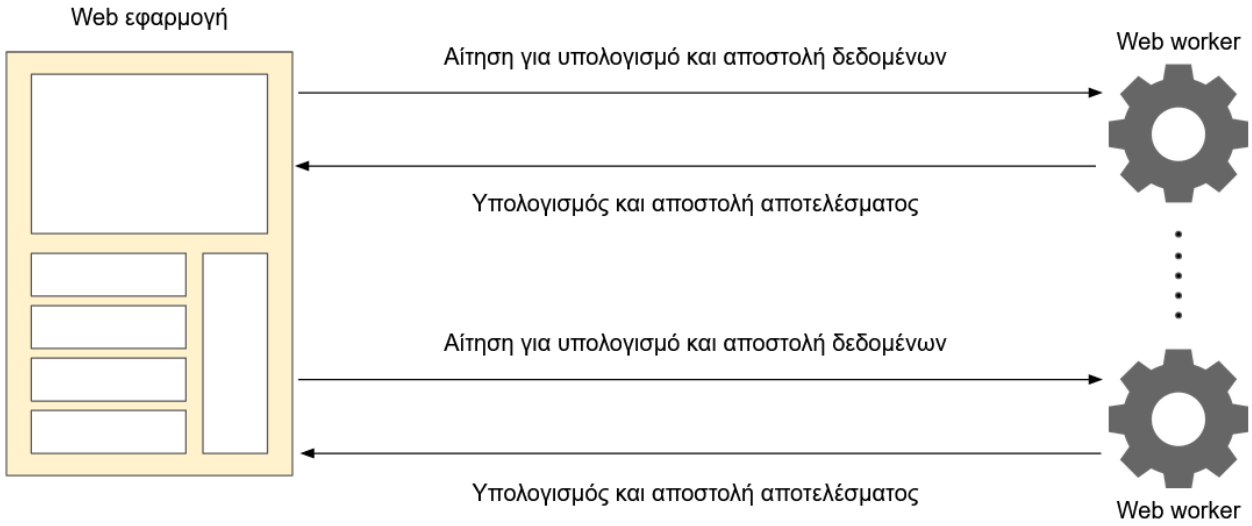
Όπως αναφέραμε παραπάνω, οι κλάσεις Geometry και Material είναι εκείνες που αποθηκεύουν τον μεγαλύτερο όγκο πληροφορίας αναφορικά με το 3D αντικείμενο. Υπάρχουν περιπτώσεις, όπως θα δούμε στις υλοποιήσεις του 3ου κεφαλαίου, όπου η γεωμετρία ενός 3D αντικειμένου απαρτίζεται από χιλιάδες έως και εκατομμύρια σημεία. Συνεπώς, θα πρέπει η επεξεργασία, αποθήκευση και ανάκτηση αυτών των σημείων να γίνεται όσο το δυνατόν πιο αποδοτικά. Για τον λόγο αυτό αποφασίστηκε κατά την ανάπτυξη της βιβλιοθήκης, να χρησιμοποιηθεί ο τύπος δεδομένων **TypedArray**, ο οποίος, όπως θα δούμε και σε επόμενη ενότητα που αφορά τους τρόπους αποστολής δεδομένων μεταξύ των Web Workers, υποστηρίζεται από τις μεθόδους επικοινωνίας τους και μειώνει σημαντικά την καθυστέρηση στην ανταλλαγή δεδομένων.

2.4 Web Workers

Όπως έχουμε αναφέρει και στο προηγούμενο κεφάλαιο, οι λειτουργίες πολλών από τις σημερινές web εφαρμογές, απαιτούν την παράλληλη ή συντρέχουσα εκτέλεση πολλών λειτουργιών, έτσι ώστε να αυξάνεται αφενός η απόδοσή τους και αφετέρου να μπορούν να παρέχουν την βέλτιστη εμπειρία στον τελικό χρήστη. Παρόλα αυτά, το τμήμα της εφαρμογής που εκτελείται στον browser του χρήστη (client side), περιορίζεται στην single thread αρχιτεκτονική της Javascript, η οποία δεν έχει την δυνατότητα να εκτελέσει πολλαπλά scripts ταυτόχρονα. Το ζήτημα αυτό καλούνται να επιλύσουν οι Web Workers. Στην ενότητα που ακολουθεί, θα παρουσιαστούν οι Web Workers και ο τρόπος με τον οποίο λειτουργούν, τα πλεονεκτήματα και μειονεκτήματά τους, καθώς και οι κατηγορίες στις οποίες χωρίζονται και οι μηχανισμοί επικοινωνίας με το κύριο thread της web εφαρμογής, που είναι και ένα από τα βασικότερα πράγματα που επηρεάζουν την απόδοσή τους.

Οι Web Workers είναι ένα API το οποίο επιτρέπει σε μια web εφαρμογή να μπορεί να εκτελεί διεργασίες (task) στο παρασκήνιο χωρίς να επηρεάζει την λειτουργία του κύριου νήματός της [16]. Χρησιμοποιούνται σε εφαρμογές ώστε να αναλαμβάνουν απαιτητικές σε υπολογισμούς διεργασίες, χωρίς να επηρεάζουν την απόκριση της εφαρμογής, διατηρώντας ενεργό το UI ώστε να μην διακόπτεται η αλληλεπίδραση με τον χρήστη.

Στο παρακάτω σχήμα απεικονίζεται η χρήση web workers σε μια εφαρμογή:



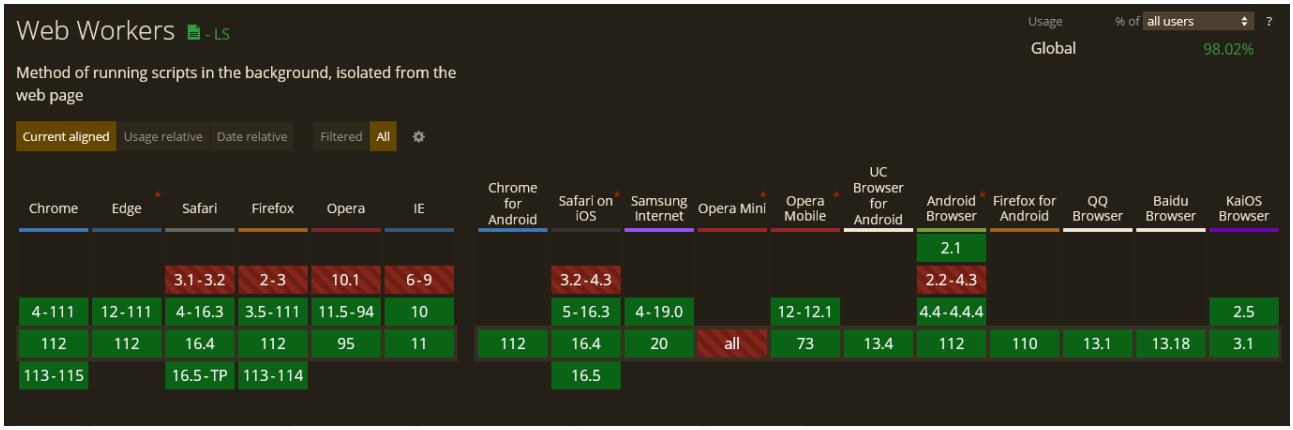
Εικόνα 8: Χρήση web workers σε εφαρμογή.

Προκειμένου να μπορέσει να επικοινωνήσει η εφαρμογή με τους workers, θα πρέπει τα δεδομένα του μηνύματος που θα αποστείλει να έχουν συγκεκριμένη μορφή. Όπως θα δούμε παρακάτω, στην ενότητα της επικοινωνίας, υπάρχουν ορισμένοι τύποι δεδομένων για τους οποίους η μεταφορά από και προς έναν web worker γίνεται αποδοτικά. Με την αποστολή των δεδομένων στον web worker, ο τελευταίος ειδοποιείται και ξεκινά την εκτέλεση των υπολογισμών του. Εφόσον τελειώσει επιτυχώς, αποστέλλει τα αποτελέσματα πίσω στο κύριο νήμα. Να σημειωθεί ότι με την ίδια λογική, μπορούμε να έχουμε επικοινωνία και μεταξύ web workers, χωρίς να αλλάζει κάτι στην διαδικασία επικοινωνίας.

Ο μέγιστος αριθμός των web workers που μπορεί να δημιουργήσει μια εφαρμογή εξαρτάται από τις ρυθμίσεις του browser. Για παράδειγμα, στην έκδοση **112.0.2** του **Firefox**, ο αριθμός αυτός είναι **512**. Αν και πρόκειται για αρκετά μεγάλο νούμερο, στην πράξη θα πρέπει να λάβουμε υπόψη και τα τεχνικά χαρακτηριστικά της συσκευής στην οποία τρέχουμε την εφαρμογή, όπως ο αριθμός των πυρήνων της CPU και η διαθέσιμη μνήμη. Ο βέλτιστος αριθμός των web workers θα πρέπει να είναι ίσος με τον αριθμό των μονάδων επεξεργασίας της CPU. Το πεδίο **hardwareConcurrency** του αντικειμένου **navigator** [17], μας επιστρέφει τον αριθμό των λογικών επεξεργαστών του συστήματος [18], με την ακόλουθη εντολή:

navigator.hardwareConcurrency

Αναφορικά με την ενσωμάτωση των web workers από τους browsers, η παρακάτω εικόνα δείχνει το επίπεδο υποστήριξης με βάση τα σημερινά δεδομένα:



Εικόνα 9: Στιγμιότυπο πίνακα που δείχνει την υποστήριξη των web workers από ένα πλήθος browsers (Πηγή: https://caniuse.com/webworkers).

Συγκεκριμένα, βλέπουμε ότι η καθολική χρήση τους από όλους τους browsers σε όλες τις συσκευές (desktop και mobile), φτάνει στο **98,02%**, με κάποιες παλαιότερες εκδόσεις να μην τους υποστηρίζουν (παλαιότερες εκδόσεις των Safari, Firefox, Opera και Internet Explorer, καθώς και mobile browsers όπως ο Safari για iOS και ο Android Browser) και άλλοι browsers να μην τους έχουν ενσωματώσει ακόμη (Opera Mini).

Πλεονεκτήματα

Η χρήση των web workers συνοδεύεται με αρκετά πλεονεκτήματα όπως:

- Η γρηγορότερη και αποδοτικότερη εκτέλεση εργασιών, η φύση των οποίων επιτρέπει στα δεδομένα τους να μπορούν να κατανεμηθούν σε web workers, με κάθε worker να εκτελεί τις ίδιες εντολές για ένα τμήμα του συνόλου των δεδομένων προς επεξεργασία.
- Η εξάλειψη της καθυστέρησης ή της αδυναμίας απόκρισης της εφαρμογής, λόγω της εκτέλεσης κάποιας υπολογιστικά απαιτητικής εργασίας, καθώς πλέον μπορεί να μεταφερθεί σε έναν worker, χωρίς να απασχολεί το κύριο νήμα.
- Η κατανομή πολλών διαφορετικών εργασιών, ανεξάρτητων μεταξύ τους, σε διαφορετικούς workers, βελτιώνοντας την εμπειρία χρήστη από άποψη ταχύτητας και απόκρισης της εφαρμογής.

Μειονεκτήματα

Προτού απαριθμήσουμε τα μειονεκτήματα από την χρήση των web workers, είναι σημαντικό να αναφέρουμε πως κάθε worker εκτελείται σε δικό του απομονωμένο περιβάλλον (browsing context)

[19], χωρίς να έχει άμεση πρόσβαση στην κατάσταση της κύριας εφαρμογής (μεταβλητές, συναρτήσεις). Λαμβάνοντας υπόψη το παραπάνω, μπορούμε να πούμε ότι τα μειονεκτήματά τους είναι:

- Σημαντική καθυστέρηση λόγω του overhead της επικοινωνίας μεταξύ των workers και του κύριου thread. Υπάρχει τρόπος να επικοινωνούν απευθείας, χωρίς ανταλλαγή μηνυμάτων, όπως θα δούμε σε επόμενη ενότητα, αλλά τίθενται ζητήματα ασφάλειας της εφαρμογής.
- Καθυστέρηση κατά την δημιουργία ενός worker, καθώς πρέπει να δημιουργηθεί και το σχετικό browsing context, μέσα στο οποίο θα εκτελέσει τον κώδικά του.
- Αδυναμία πρόσβασης και διαχείρισης του DOM tree της σελίδας, το οποίο είναι απόρροια του διαφορετικού browsing context στο οποίο δημιουργούνται οι workers, καθώς τα αντικείμενα window και document με τα οποία γίνεται η διαχείριση του DOM, δεν είναι προσβάσιμα μέσα σε αυτούς.

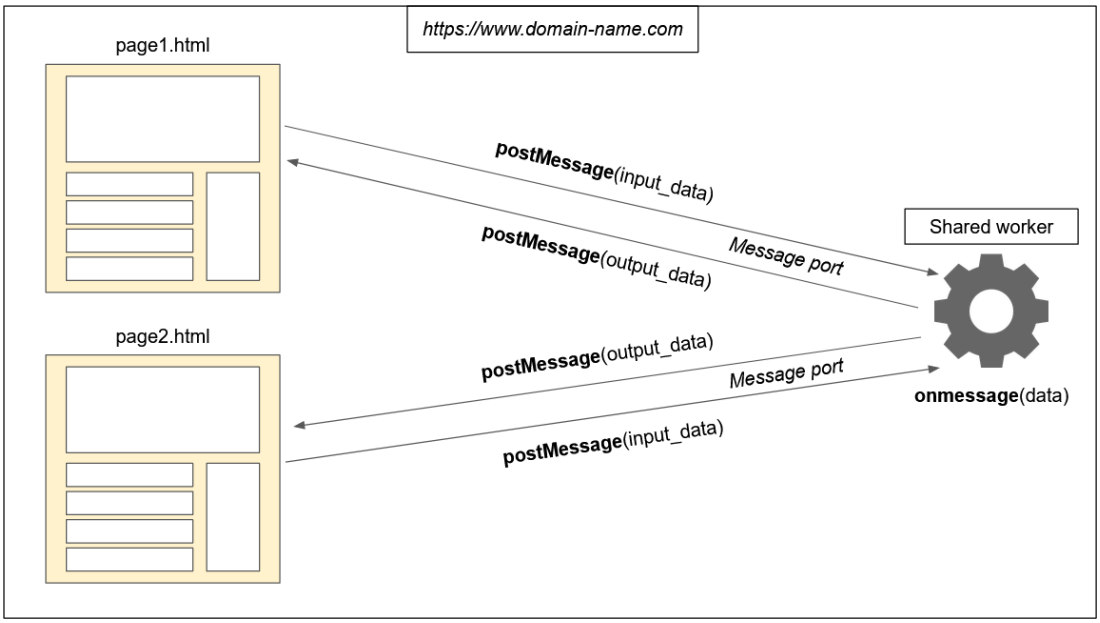
Το API των Web Workers μας παρέχει 2 διαφορετικά είδη workers, με βάση το browsing context και τις δυνατότητες πρόσβασης (permissions) από τα υπόλοιπα browsing contexts της εφαρμογής (window, υπόλοιποι web workers). Αυτοί είναι οι **shared workers** και οι **dedicated workers**.

2.4.1 Shared Workers

Ο shared worker είναι ένα είδος web worker ο οποίος μπορεί να είναι προσβάσιμος από πολλαπλά browsing contexts, όπως windows (πολλαπλές σελίδες μιας εφαρμογής) ή και από πολλαπλούς web workers [20]. Μόλις δημιουργηθεί ένας shared worker από κάποιο script της εφαρμογής, μπορεί να είναι προσβάσιμος και να επικοινωνεί με οποιοδήποτε άλλο script αυτής της εφαρμογής. Ο τερματισμός του και η εκκαθάρισή του από το σύστημα γίνεται μόνο σε περίπτωση που δεν υπάρχει κάποια ενεργή αναφορά (reference) στο αντικείμενο του worker σε κάποιο από τα script της εφαρμογής. Να σημειωθεί επίσης πως για να έχουν δικαίωμα πρόσβασης και επικοινωνίας με τον shared worker τα διάφορα browsing contexts, θα πρέπει να έχουν το ίδιο origin, δηλαδή:

- Κοινό πρωτόκολλο επικοινωνίας (http, https).
- Κοινό host (domain name).
- Κοινή θύρα (port number).

Στο παρακάτω σχήμα απεικονίζεται η επικοινωνία μεταξύ διάφορων browsing contexts (σελίδων της εφαρμογής στην συγκεκριμένη περίπτωση) και ενός shared worker:



Εικόνα 10: Απεικόνιση επικοινωνίας πολλαπλών browsing contexts με έναν shared worker.

Κατά την δημιουργία του, ο shared worker κατασκευάζει ένα message port το οποίο μπορούν να χρησιμοποιούν τα διάφορα browsing contexts με σκοπό να επικοινωνούν μαζί του. Όπως θα δούμε στην ενότητα της επικοινωνίας, η αποστολή και παραλαβή δεδομένων από αμφότερα μέρη της εφαρμογής, γίνεται με την χρήση της συνάρτησης **postMessage()**. Κάθε φορά που αποστέλλεται ένα μήνυμα στον worker, εκτελείται το **onmessage()** event πάνω στο port που έχει ανοίξει ο ίδιος, για να μπορεί να ενημερωθεί πως υπάρχουν δεδομένα προς παραλαβή. Έπειτα, αφού γίνει η όποια επεξεργασία τους, τα αποτελέσματα επιστρέφονται με τον ίδιο τρόπο στα εκάστοτε browsing contexts [21].

Αν και η ιδέα ενός μηχανισμού που να μπορεί να εκτελεί tasks από διάφορες πηγές εντός της ίδιας εφαρμογής και να δίνει από κοινού πρόσβαση σε αυτές ακούγεται εξαιρετικά χρήσιμη, η υποστήριξή του από τους browsers είναι σε ικανοποιητικό επίπεδο μόνο στις desktop εκδόσεις, με μεγάλο μέρος των mobile browsers να μην παρέχουν υποστήριξη για τον συγκεκριμένο τύπο worker. Αυτό μπορούμε να το δούμε και στο παρακάτω σχήμα:

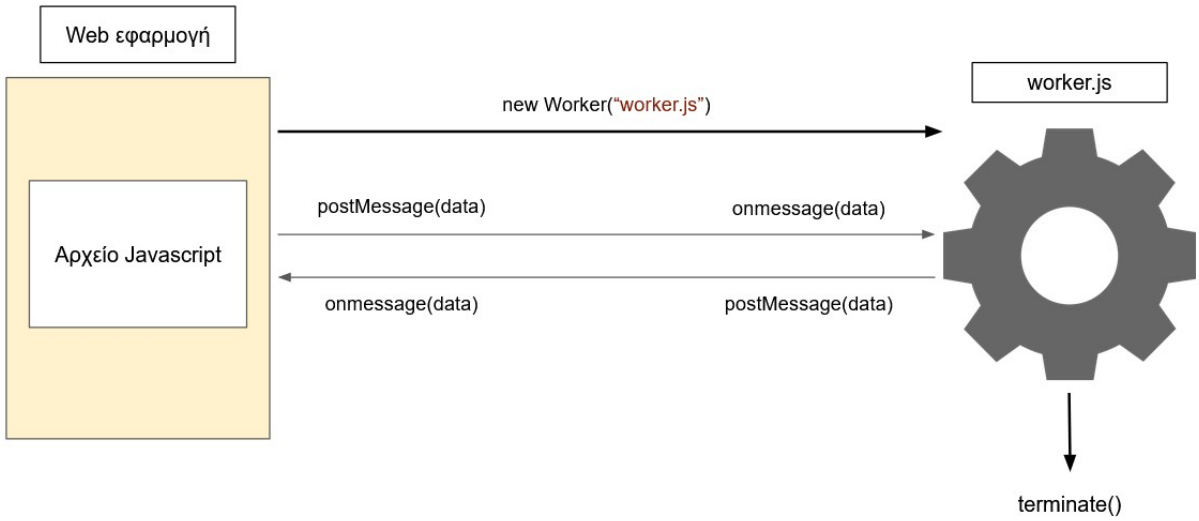
Εικόνα 11: Στιγμιότυπο πίνακα που δείχνει την υποστήριξη του shared worker από τους browsers.

Όπως μπορούμε να δούμε, η συνολική κάλυψη που έχει από το σύνολο των browsers, φτάνει μέχρι το **47,38%**, με τον Internet Explorer να μην παρέχει υποστήριξη για τον συγκεκριμένο τύπο worker, καθώς πλέον ο συγκεκριμένος browser έχει σταματήσει να αναπτύσσεται, και με την συντριπτική πλειοψηφία των mobile browsers να μην έχουν την αντίστοιχη υλοποίηση. Αν και δεν είναι γνωστοί οι ακριβείς λόγοι για τους οποίους δεν υφίσταται ακόμη υποστήριξη, μπορούμε να υποθέσουμε ότι οφείλεται σε λόγους απόδοσης που προκύπτουν από τα χαρακτηριστικά ενός shared worker που πρέπει να υλοποιηθούν, σε αντίθεση με την αρχιτεκτονική των μηχανών Javascript που έχουν αναπτυχθεί για κινητές συσκευές [22].

2.4.2 Dedicated Workers

Ο dedicated worker είναι ένα άλλο είδους worker, ο οποίος, σε αντίθεση με τον shared worker, είναι προσπελάσιμος μόνο από το script που τον δημιούργησε [23]. Είναι ο πιο σημαντικός τύπος worker, και στα πλαίσια αυτής της εργασίας αλλά και αναφορικά με την ευρύτερη χρήση του, καθώς έχει μεγαλύτερη υποστήριξη από τους browsers, απ’ ότι έχουν οι shared workers (Εικόνα 2.x: Υποστήριξη Web Workers).

Στο σχήμα που ακολουθεί, μπορούμε να δούμε τον κύκλο ζωής ενός dedicated web worker σε μια εφαρμογή:



Εικόνα 12: Κύκλος ζωής ενός dedicated web worker.

Παρόλο που δεν απεικονίζεται στο σχήμα, για να ακολουθήσουμε μια πιο ασφαλή λογική προγραμματισμού με περιορισμένα σφάλματα κατά την εκτέλεση της εφαρμογής (runtime), καθώς και να εξασφαλίσουμε την βέλτιστη αξιοποίηση των δυνατοτήτων ενός browser [24], οφείλουμε να ελέγξουμε εάν οι workers υποστηρίζονται με την συνθήκη:

```
if ( window.Worker )
    // Δημιουργία και επικοινωνία worker με το κύριο thread.

}
```

Εφόσον οι workers υποστηρίζονται, μπορούμε να δημιουργήσουμε έναν, δημιουργώντας ένα νέο στιγμιότυπο Worker με την χρήση του API:

```
const worker = new Worker("worker.js");
```

Στις σύγχρονες web εφαρμογές, υπάρχουν αρκετά scripts τα οποία εξαρτώνται από συναρτήσεις και μεθόδους άλλων scripts. Πριν την εισαγωγή του ES6 [25], η σειρά με την οποία φορτώνονταν τα JS αρχεία, επηρέαζε τον αριθμό των συναρτήσεων που ήταν υλοποιημένες στα υπόλοιπα αρχεία. Παραδείγματος χάρη, ένα αρχείο **alpha.js** το οποίο φορτώνεται πριν το αρχείο **beta.js**, δεν θα

μπορεί να χρησιμοποιήσει συναρτήσεις του τελευταίου, καθώς δεν έχει πρόσβαση σε πόρο (resource) που δεν έχει φορτωθεί, ενώ δεν ισχύει το ίδιο για το beta.js. Για να εξαλειφθεί το παραπάνω πρόβλημα εξαρτήσεων, εισήχθη η έννοια του module [26]. Κάνοντας ένα script να είναι module, μπορούμε να χρησιμοποιήσουμε την εντολή import() και να εισάγουμε μεθόδους και συναρτήσεις άλλων scripts, ανεξαρτήτου σειράς φόρτωσης. Το ίδιο ισχύει και για τους web workers. Εάν θέλουμε να χρησιμοποιήσουμε την λογική της ES6 και τα imports, θα πρέπει να το δηλώσουμε στον κατασκευαστή του worker, προσθέτοντας και δεύτερη παράμετρο όπως φαίνεται παρακάτω:

```
const worker = new Worker("worker.js", {type: "module"});
```

Σε διαφορετική περίπτωση, μπορούμε να παραλείψουμε την παράμετρο αυτή και να κάνουμε import με την εντολή **importScripts()** [27] στην αρχή του κώδικα, δίνοντας σαν παράμετρο το όνομα του JS αρχείου. Εφόσον έχει δημιουργηθεί ο worker, μπορεί να πραγματοποιηθεί επικοινωνία με την μέθοδο **postMessage()** και το **onmessage** event. Τέλος, μπορούμε να αποδεσμεύσουμε τον worker, εφόσον δεν χρειάζεται στην εφαρμογή πλέον, είτε καλώντας την μέθοδο **terminate()** μέσα από τον κώδικα του web worker είτε καλώντας την μέθοδο **close()** από το script που έχει τον worker σαν αναφορά.

2.5 Επικοινωνία μεταξύ worker και κύριου νήματος εφαρμογής

Όταν αναφερόμαστε σε web workers, έμμεσα αναφερόμαστε στην επικοινωνία τους με το κύριο thread, καθώς η δημιουργία ξεχωριστού browsing context σε κάθε worker, αποκλείει την δυνατότητα πρόσβασης και χρήσης κοινών μεταβλητών (αν και όπως θα δούμε παρακάτω, υπάρχει τρόπος το κύριο thread και οι workers να έχουν πρόσβαση σε κοινές μεταβλητές, υπό ορισμένες προϋποθέσεις). Η επικοινωνία αυτή αποτελεί το πιο σημαντικό κομμάτι κατά την σχεδίαση μιας εφαρμογής με web workers καθώς, όπως θα δούμε, ο τύπος και η μορφή των δεδομένων που στέλνουμε μπορούν να επηρεάσουν την απόδοσή της. Στις παρακάτω ενότητες θα δούμε την μέθοδο postMessage(), τους τύπους δεδομένων που μπορεί να αξιοποιήσει για γρηγορότερες μεταφορές από και προς το κύριο thread, και τα αντικείμενα SharedArrayBuffer και Offscreencanvas, που διευκολύνουν στην επικοινωνία και στην διαχείριση των canvas στοιχείων μιας σελίδας, που έχει μεγάλη σημασία σε εφαρμογές γραφικών.

2.5.1 Μέθοδος postMessage()

Η postMessage() είναι η μέθοδος που χρησιμοποιείται για την επικοινωνία μεταξύ του κύριου thread και του web worker. Η γενική μορφή της είναι:

postMessage(data, transfer)

Η βασική λειτουργία της είναι να αντιγράφει και να μεταφέρει δεδομένα (data) από ένα browsing context σε ένα άλλο. Ειδικότερα, βλέποντας τις παραμέτρους, έχουμε:

- **data**: Αφορά τα δεδομένα που πρόκειται να σταλούν κατά την επικοινωνία. Μπορούν να είναι στοιχεία οποιουδήποτε τύπου, εκτός από function objects και στοιχεία του DOM, καθώς αυτά δεν υποστηρίζονται από τον αλγόριθμο που χρησιμοποιείται για το structure cloning [28], μια διαδικασία που δημιουργεί αντίγραφα δεδομένων, απαραίτητη για την postMessage προκειμένου να τα στείλει.
- **transfer**: Αφορά τα transferable objects [29], τα οποία μπορούν να μεταφερθούν από ένα browsing context σε ένα άλλο, χωρίς να χρειάζεται να αντιγραφούν, με την προϋπόθεση ότι τα αντικείμενα αυτά θα είναι διαθέσιμα σε ένα browsing context τη φορά. Από τους τύπους δεδομένων που είναι transferable, μας ενδιαφέρουν τα **ArrayBuffer** αντικείμενα. Η παράμετρος αυτή είναι ένα array το οποίο δέχεται ως στοιχεία όλα τα αντικείμενα που είναι transferable και δεν χρειάζεται να αντιγραφούν για την μεταφορά τους.

Στην παρακάτω εικόνα φαίνονται 2 περιπτώσεις χρήσης της postMessage(), καθώς και το συντακτικό που χρησιμοποιούμε προκειμένου να δηλώσουμε ότι ένα αντικείμενο μπορεί να αντιμετωπιστεί σαν transferable από την postMessage().

Απλή αντιγραφή

```
let arr = new ArrayBuffer(1000000 * Int32Array.BYTES_PER_ELEMENT);
let worker = new Worker('worker.js');
worker.postMessage(arr);
```

Χρήση transferable object

```
let arr = new ArrayBuffer(1000000 * Int32Array.BYTES_PER_ELEMENT);
let worker = new Worker('worker.js');
worker.postMessage(arr, [arr]);
```

Εικόνα 13: Τμήματα κώδικα postMessage με αντιγραφή και με χρήση transferables.

Βλέπουμε πως όταν χρησιμοποιούμε transferable objects, χρειαζόμαστε ένα δεύτερο όρισμα το οποίο είναι array και δέχεται τα αντικείμενα που είναι transferables. Χρησιμοποιώντας transferable objects, μπορούμε να μειώσουμε σημαντικά τον συνολικό χρόνο ανταλλαγής δεδομένων. Σε ένα benchmark που πραγματοποιήθηκε από τους προγραμματιστές της Google στον Chrome browser, διαπιστώθηκε ότι η ταχύτητα μετάδοσης δεδομένων με χρήση transferables ήταν **45 φορές** πιο γρήγορη από την απλή αντιγραφή και αποστολή δεδομένων [30]. Ειδικότερα, η αποστολή ενός αντικειμένου ArrayBuffer συνολικού μεγέθους **32 MB**, εκτελέστηκε σε **302 ms** με απλή αντιγραφή, ενώ με μεταφορά, λόγω της ιδιότητάς του ως transferable object, εκτελέστηκε σε **6.6 ms**.

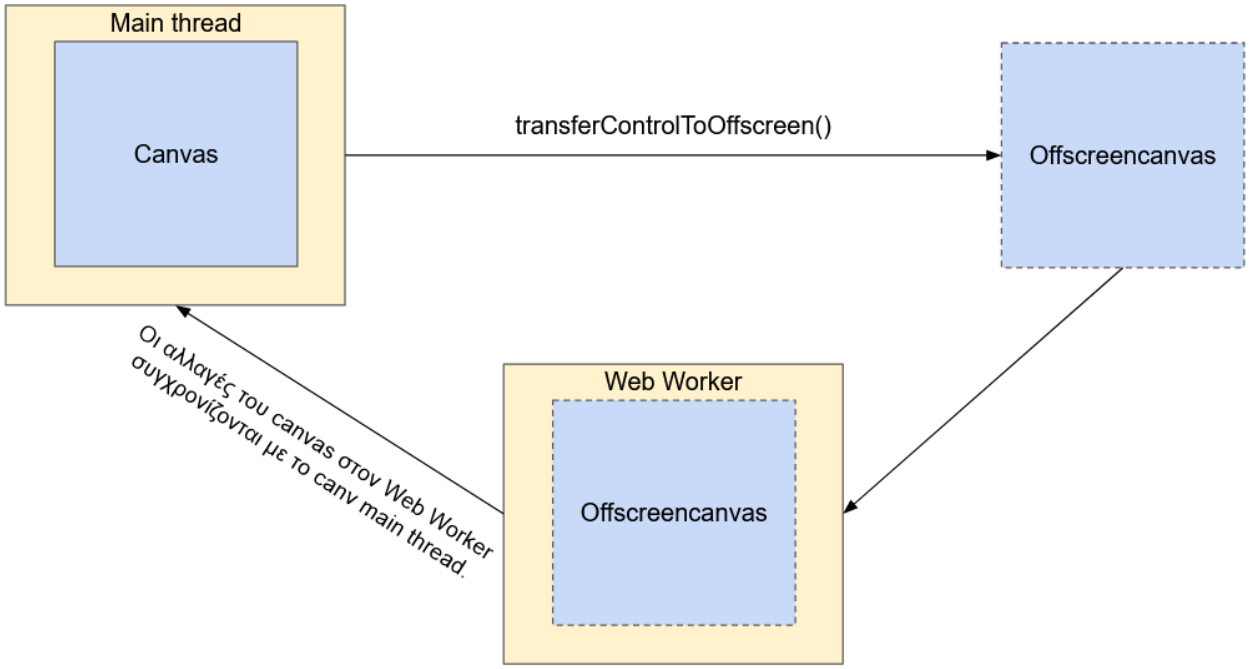
Όπως είδαμε και στην ενότητα της βιβλιοθήκης three.js, όλα τα δεδομένα που αφορούν το geometry ενός mesh, αποθηκεύονται σε ένα ArrayBuffer. Αυτό θα μας βοηθήσει πολύ, όπως θα δούμε στο 3ο κεφάλαιο με τις υλοποιήσεις, καθώς η ιδιότητα του transferable object θα κάνει πιο γρήγορη την ανταλλαγή δεδομένων.

2.5.2 Offscreencanvas

Το Offscreencanvas είναι ένα interface το οποίο επιτρέπει την εκτέλεση λειτουργιών σχεδίασης γραφικών σε ένα <canvas> εκτός του κυρίου thread, χωρίς να επηρεάζεται η απόδοση της εφαρμογής [31]. Μέχρι πρότινος, η εκτέλεση ενεργειών σε ένα <canvas> μπορούσε να γίνει μόνο από το κύριο thread μιας εφαρμογής, μαζί με την αλληλεπίδραση του χρήστη. Αυτό οδηγούσε σε προβλήματα απόδοσης και απόκρισης της εφαρμογής, καθώς ένα thread καλούνταν να εκτελέσει

υπολογιστικά εντατικές ενέργειες του <canvas>, αλλά και να παρακολουθεί τα events των υπολοίπων στοιχείων του DOM που αποτελούσαν το UI. Επίσης, όλες οι ενέργειες απεικόνισης γραφικών πραγματοποιούνταν πάνω σε ένα <canvas>, το οποίο είναι επίσης στοιχείο του DOM. Με την εισαγωγή του Offscreencanvas, έγινε δυνατός ο διαχωρισμός του <canvas> και των λειτουργιών του από το DOM. Αυτό είναι ένα μεγάλο πλεονέκτημα καθώς, όπως έχουμε αναφέρει σε προηγούμενη ενότητα, οι web workers δεν μπορούν να έχουν πρόσβαση σε DOM στοιχεία. Συνεπώς, το Offscreencanvas ξεπερνά τον περιορισμό αυτό και μπορεί να εκτελείται μέσα σε web worker.

Στο παρακάτω σχήμα, απεικονίζεται η ενημέρωση ενός αντικειμένου Offscreencanvas από έναν web worker, και ο συγχρονισμός του με το <canvas> του main thread:



Εικόνα 14: Σχήμα εκτέλεσης offscreencanvas σε web worker και συγχρονισμός με το main thread.

Για να δημιουργήσουμε ένα αντικείμενο Offscreencanvas από ένα <canvas> στοιχείο, πρέπει να καλέσουμε την μέθοδο **transferControlToOffscreen()** [32]. Να σημειωθεί επίσης ότι ένα αντικείμενο Offscreencanvas είναι transferable, που σημαίνει ότι μπορεί να μεταφερθεί απευθείας στον web worker χωρίς να χρειάζεται αντιγραφή. Στην συνέχεια, μπορούμε να ανακτήσουμε το Offscreencanvas μέσα στον web worker και να το διαχειριστούμε όπως ένα συμβατικό <canvas> στοιχείο.

2.6 Βιβλιογραφική επισκόπηση

Η δυνατότητα των web workers να εκτελούνται παράλληλα με το main thread της εφαρμογής, αποτέλεσε εφαλτήριο για να μελετηθούν οι δυνατότητές τους σε real time εφαρμογές, αξιοποιώντας τους για την υλοποίησή τους και συγκρίνοντας την απόδοσή τους με την απόδοση του single threaded μοντέλου της ίδιας εφαρμογής. Στην ενότητα που ακολουθεί, θα δούμε έρευνες και αξιολογήσεις εφαρμογών που χρησιμοποιούν τους web workers, καθώς και τα πλεονεκτήματα που προσφέρουν στην ταχύτητα και στην συνολική απόδοσή τους.

Στην εργασία τους οι Hyung Woo Kim και Yang-Won Lee [33], χρησιμοποίησαν web workers σε συνδυασμό με WebGL για την οπτικοποίηση 3D γεωγραφικών δεδομένων (geo-visualization). Μελέτησαν τις δυνατότητες διαχωρισμού της λογικής της εφαρμογής που αφορά την επεξεργασία και απεικόνιση raster δεδομένων, μοιράζοντας τον όγκο των δεδομένων προς επεξεργασία στους web workers, και στέλνοντας τα αποτελέσματά τους στην GPU μέσω του WebGL API ώστε να γίνει η απεικόνιση. Χρησιμοποιώντας έναν σταθερό (desktop) υπολογιστή και μια συσκευή tablet, με τον αριθμό των web workers να είναι ίδιος με τον αριθμό των πυρήνων του επεξεργαστή της κάθε συσκευής, και ένα σύνολο περιπτώσεων ελέγχου με raster δεδομένα προς απεικόνιση (100x100, 500x500, 1000x1000, 2000x2000, 3000x3000), βλέποντας τα αποτελέσματα των εκτελέσεων της εφαρμογής με single threaded και multi threaded υλοποίηση, παρατήρησαν ότι:

- Οι διαφορές στους χρόνους εκτέλεσης σε σταθερό υπολογιστή είναι μιδαμινές για όλα τις περιπτώσεις ελέγχου, με την χρήση των web workers να προσφέρει μικρότερο χρόνο εκτέλεσης.
- Οι χρόνοι εκτέλεσης της εφαρμογής στην συσκευή tablet ανεβαίνουν εκθετικά και στις 2 υλοποιήσεις της εφαρμογής, για τις απεικονίσεις των 2000x2000 και 3000x3000 δεδομένων, διατηρώντας σταθερή βελτίωση, ανάλογη της βελτίωσης στον σταθερό υπολογιστή, με την χρήση των web workers.

Ο υπολογισμός του βέλτιστου αριθμού web workers και η δυναμική ανάθεσή τους είναι ένα θέμα που απασχολεί τους προγραμματιστές εφαρμογών. Στην εργασία των J. Verd´u et al. [34], προτείνεται ένας αλγόριθμος δυναμικής ανάθεσης web workers, ώστε να εξυπηρετούνται με τον καλύτερο τρόπο οι ανάγκες της εφαρμογής και να μεγιστοποιείται η απόδοσή της. Ο αλγόριθμος αυτός αποφασίζει εάν χρειάζεται να αυξήσει ή να μειώσει τον αριθμό των web workers που χρησιμοποιούνται, συγκρίνοντας τον μέσο όρο απόδοσης της εφαρμογής από την τελευταία φορά

που άλλαξε ο αριθμός των web workers, με τον μέσο όρο της υφιστάμενης απόδοσης της εφαρμογής.

Για την πραγματοποίηση μετρήσεων, επιλέχθηκαν οι εφαρμογές HashApp, που αφορά τον υπολογισμό του hash ενός αλφαριθμητικού, και RayApp, που αφορά τον υπολογισμό σκιών και φωτεινότητας των αντικειμένων σε μια εφαρμογή με γραφικά. Η πρώτη εφαρμογή αποτελεί παράδειγμα ασύγχρονης εκτέλεσης, καθώς δεν απαιτείται ο συγχρονισμός των workers κατόπιν υπολογισμού του ενός hash για να προχωρήσουν στο επόμενο, ενώ στην δεύτερη είναι απαραίτητη η επικοινωνία μεταξύ τους, ώστε τα τελικά αποτελέσματα να αποτυπωθούν σωστά από την εφαρμογή.

Για τις δοκιμές έγιναν σε λειτουργικά συστήματα Windows Server 2008 R2 και Linux Ubuntu LTS 14.04, χρησιμοποιώντας τους Chrome, Firefox και Internet Explorer για την εκτέλεση των εφαρμογών. Επίσης, οι δοκιμές έγιναν χρησιμοποιώντας από 1 έως 20 web workers. Τα αποτελέσματα τα οποία προέκυψαν, έδειξαν ότι οι δυναμική ανάθεση web workers μπορεί να προσφέρει καλύτερα αποτελέσματα από την αρχικοποίηση ενός συγκεκριμένου αριθμού web workers χωρίς την δυνατότητα μεταβολής του αριθμού τους, αλλά δεν μπορεί να προσφέρει ακριβώς το αποτέλεσμα που θα προέκυπτε από την χρήση του βέλτιστου αριθμού από web workers για κάθε εργασία.

Το Offscreencanvas API ήταν μια άλλη τεχνολογία που συνδυάζεται με web workers και μελετήθηκε. Συγκεκριμένα, οι Michail Schwab et al [35] υλοποίησαν και παρουσίασαν ένα καινούργιο τρόπο φόρτωσης και απεικόνισης SVG γραφικών, το SSVG (Scalable Scalable Vector Graphics). Η ιδέα ξεκίνησε από το γεγονός ότι τα SVG παρουσιάζουν προβλήματα στην απόδοση της εφαρμογής, όταν πρόκειται να απεικονιστεί μεγάλος αριθμός γραφικών στοιχείων, αναγκάζοντας τους προγραμματιστές να αναζητήσουν λύσεις σε τεχνολογίες όπως το Canvas και το WebGL API, για να βελτιώσουν την απόδοση της εφαρμογής. Καθώς η μαθησιακή καμπύλη των Canvas και ειδικά του WebGL είναι αρκετά δύσκολη, το SSVG είναι μια τεχνολογία που συνδυάζει το SVG, το SharedArrayBuffer object και το Offscreencanvas. Ειδικότερα:

1. Όταν πρόκειται να γίνει μια αλλαγή σε ένα SVG στοιχείο του DOM, τότε η αλλαγή αυτή μετατρέπεται σε κατάλληλη μορφή για μεταφορά μέσω του SharedArrayBuffer και μεταφέρεται στο worker thread, στο οποίο υπάρχει το VDOM (Virtual DOM). Το VDOM είναι μια δομή στην οποία αποθηκεύεται η κατάσταση της τελευταίας αλλαγής των SVG στοιχείων που είναι έτοιμη προς απεικόνιση.

2. Στην συνέχεια, η ενημερωμένη κατάσταση του VDOM απεικονίζεται σε ένα Offscreencanvas. Μόλις οι αλλαγές ολοκληρωθούν, η νέα κατάσταση του VDOM μεταφέρεται από το worker thread στο main thread, ώστε να γίνει η ενημέρωση.

Στόχοι του SSVG ήταν η επίτευξη χαμηλότερου φόρτου εργασίας στο main thread, υψηλότερη απόδοση και διαδραστικότητα. Οι μετρήσεις σε περιπτώσεις εφαρμογών που έγιναν, διαπιστώθηκε ότι το SSVG μπορεί να απεικονίσει 9 φορές γρηγορότερα από το SVG, στα **15.000** στοιχεία. Επιπλέον, το SVG μπορούσε διατηρήσει ρυθμό ανανέωσης στα **30 FPS** μέχρι **2.500** στοιχεία, ενώ το SSVG έως **35.000** στον ίδιο ρυθμό ανανέωσης. Όσον αφορά το scaling της CPU, παρέχοντας την διπλάσια επεξεργαστική ισχύ, παρατηρήθηκε ότι η απόδοση του SVG αυξήθηκε κατά **50%**, ενώ η απόδοση του SSVG κατά **100%**, καθιστώντας το πιο scalable από το SVG.

3. Βελτιστοποίηση εφαρμογών 3D γραφικών με web workers

Στο κεφάλαιο αυτό θα παρουσιαστούν τρεις περιπτώσεις εφαρμογών με σχεδίαση και απεικόνιση 3D γραφικών, κάθε μια από τις οποίες περιλαμβάνει ένα στοιχείο που μπορεί να βελτιωθεί με την χρήση web workers, βάσει των στόχων που τέθηκαν στο 1ο κεφάλαιο. Οι εφαρμογές αυτές είναι:

1. **Terrain Generation**: Εφαρμογή που αφορά μεγάλο όγκο υπολογισμών οι οποίοι μπορούν να παραλληλοποιηθούν και να εκτελεστούν ταχύτερα.
2. **Boid Simulation**: Αφορά υπολογισμούς που γίνονται συνεχόμενα σε βρόχο επανάληψης και καθορίζουν τα FPS της εφαρμογής, η παραλληλοποίηση των οποίων μπορεί να οδηγήσει σε ταχύτερες επαναλήψεις και υψηλότερες τιμές FPS.
3. **Graph Visualization**: Αφορά την φόρτωση, επεξεργασία και απεικόνιση δεδομένων σε 3D γράφο, με τις ενέργειες της φόρτωσης και επεξεργασίας να εκτελούνται σε web worker, έτσι ώστε να μην επηρεάζεται η απόκριση της εφαρμογής και η τελική εμπειρία χρήστη.

Για κάθε μια από τις παραπάνω εφαρμογές, θα παρουσιαστεί η σειριακή υλοποίησή τους χωρίς την χρήση web workers, καθώς και η υλοποίησή τους με την χρήση των web workers, ερευνώντας τα σημεία τα οποία μπορούν να μεταφερθούν εκτός του main thread, και στην συνέχεια, θα παρουσιαστούν οι συγκριτικές αξιολογήσεις και τα αποτελέσματα αυτών, για κάθε εφαρμογή ξεχωριστά.

Σε όλες τις εφαρμογές, οι μετρήσεις έγιναν με την χρήση της μεθόδου *performance.now()* [36], με τα αποτελέσματα να είναι σε λεπτά του δευτερολέπτου (milliseconds). Επίσης, για την εφαρμογή Boid Simulation, η μέτρηση των FPS έγινε με την βιβλιοθήκη *stats.js* [37], η οποία αποτελεί ξεχωριστό πρόσθετο (addon) της three.js. Επιπλέον, για την ευκολότερη διαχείριση των παραμέτρων πάνω στις οποίες γίνονται οι μετρήσεις για κάθε εφαρμογή, χρησιμοποιήθηκε η βιβλιοθήκη *lil-gui* [38], η οποία μας βοηθά στην γρήγορα δημιουργία GUI στοιχείων και την σύνδεσή τους με τις τιμές των παραμέτρων της εφαρμογής.

Όλες οι μετρήσεις έγιναν σε σταθερό (desktop) υπολογιστή με τα ακόλουθα χαρακτηριστικά:

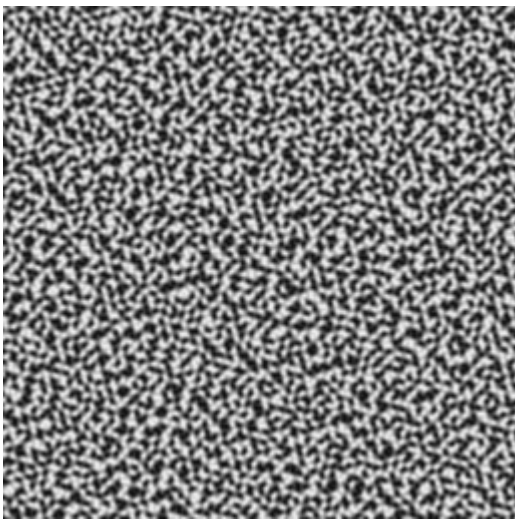
- **Επεξεργαστής**: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz (4 CPUs)
- **Κάρτα γραφικών**: Intel(R) HD Graphics 4600
- **Λετουργικό σύστημα**: Windows 10 Home 64-bit

Επιπλέον, οι ακόλουθοι browsers χρησιμοποιήθηκαν για τις μετρήσεις των εφαρμογών:

- **Mozilla Firefox**, έκδοσης 113.0.1
- **Google Chrome**, έκδοσης 113.0.5672.94
- **Opera**, έκδοσης 98.0.4759.39
- **Microsoft Edge**, έκδοσης 113.0.1774.42

3.1 Εφαρμογής παραγωγής τρισδιάστατου τοπίου (3D terrain generation)

Η πρώτη εφαρμογή που θα παρουσιάσουμε είναι μια εφαρμογή παραγωγής τυχαίου 3D τοπίου (procedural terrain generation) [39] με την χρήση του αλγορίθμου Simplex Noise [40]. Ο αλγόριθμος Simplex Noise δημιουργήθηκε από το Ken Perlin ως εξέλιξη του Perlin Noise [41], του πρώτου αλγορίθμου που κατασκευάστηκε επίσης από τον ίδιο και χρησιμοποιήθηκε για την παραγωγή procedural γραφικών, βρίσκοντας ευρεία χρήση σε βιντεοπαιχνίδια και ταινίες.

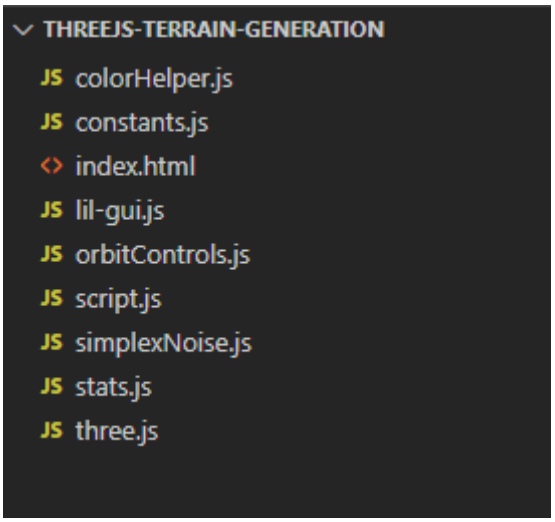


Εικόνα 15: Ένα heightmap κατασκευασμένο με τον Simplex Noise (Πηγή: https://en.wikipedia.org/wiki/Simplex_noise)

Ένα χαρακτηριστικό αυτού του αλγορίθμου είναι ότι ο υπολογισμός της τιμής του noise για ένα σημείο είναι ανεξάρτητος από τις τιμές προηγούμενων ή και γειτονικών σημείων, παρέχοντάς μας την δυνατότητα παραλληλοποίησης των υπολογισμών του για κάθε σημείο, στην περίπτωσή μας με την χρήση Web Workers. Πριν φτάσουμε όμως στους Web Workers, θα παρουσιάσουμε την υλοποίηση της σειριακής έκδοσής του.

3.1.1 Σειριακή υλοποίηση

Στην παρακάτω εικόνα φαίνεται η δομή με τα αρχεία της εφαρμογής:



Εικόνα 16: Δομή αρχείων της εφαρμογής για την σειριακή υλοποίηση.

Τα αρχεία τα οποία αποτελούν βιβλιοθήκες και παρέχονται σαν πρόσθετα από την three.js είναι τα ακόλουθα:

- **three.js**: Πρόκειται για την βιβλιοθήκη που χρησιμοποιούμε για την σχεδίαση 3D γραφικών στην εφαρμογή μας.
- **stats.js**: Είναι το module που μας παρέχει με πληροφορίες αναφορικά με τα FPS σε πραγματικό χρόνο κατά την διάρκεια εκτέλεσης της εφαρμογής.
- **orbitControls.js**: Είναι ένα βοηθητικό module το οποίο μας βοηθάει στην πλοήγηση στον 3D χώρο της εφαρμογής, παρέχοντας λειτουργίες μεταφοράς, περιστροφής και μεγέθυνσης / σμίκρυνσης της 3D σκηνής, με την βοήθεια του ποντικιού.
- **lil-gui.js**: Είναι η βιβλιοθήκη που μας βοηθά να φτιάξουμε εύκολα UI controls, όπως text fields, dropdown, checkboxes και buttons ώστε να μπορούμε να μεταβάλλουμε τις τιμές των παραμέτρων της εφαρμογής, αλλάζοντας απλά τις τιμές των πεδίων αυτών, αλλά και να εκτελούμε λειτουργίες με το πάτημα ενός κουμπιού.

Τα αρχεία που απαρτίζουν το κύριο μέρος της εφαρμογής είναι:

- **colorHelper.js**: Παρέχει την βοηθητική συνάρτηση *setColorValue()* οι οποίες χρησιμοποιούνται για την επιλογή χρώματος σε ένα συγκεκριμένο σημείο, με βάση την τιμή που επέστρεφε ο Simplex Noise γι' αυτό το σημείο.
- **constants.js**: Περιέχει σταθερές τιμές που αφορούν ρυθμίσεις της εφαρμογής, όπως θα δούμε παρακάτω.
- **simplexNoise.js**: Είναι η υλοποίηση του Simplex Noise, μετατρέποντας τον κώδικα Java από την εργασία του Stefan Gustavson σε Javascript [42].
- **script.js**: Είναι το κύριο μέρος της εφαρμογής, όπου γίνεται η αλληλεπίδραση με τον χρήστη και η απεικόνιση των τοπίων που παράγονται με τον Simplex Noise.

Αναλυτικότερα, για το αρχείο colorHelper.js, έχουμε την υλοποίηση της συνάρτησης setColorValue(), όπως φαίνεται στην παρακάτω εικόνα:

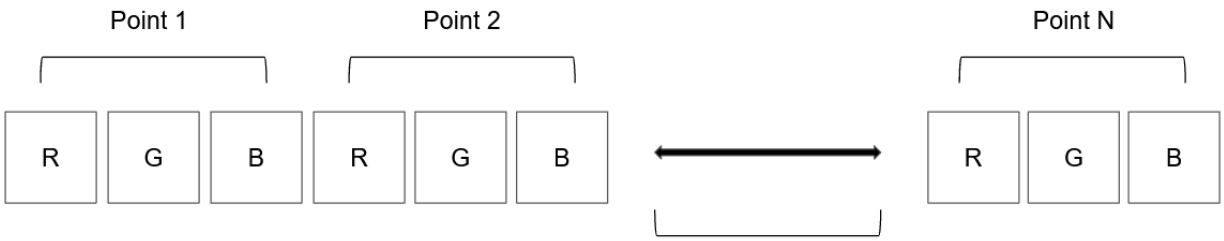
```
function setColorToVertice(colorsAttr, i, r, g, b) {
    colorsAttr.array[i] = r / 255;
    colorsAttr.array[i+1] = g / 255;
    colorsAttr.array[i+2] = b / 255;
}

export function setColorValue(colorsAttr, i, color) {
    if(color < -50) {
        setColorToVertice(colorsAttr, i, 0, 140, 180);
    }
    else if(color >= -50 && color <= 0){
        setColorToVertice(colorsAttr, i, 0, 168, 210);
    }
    else if(color > 0 && color < 20) {
        setColorToVertice(colorsAttr, i, 252, 255, 129);
    }
    else if(color >= 20 && color < 100) {
        setColorToVertice(colorsAttr, i, 38, 189, 15);
    }
    else if(color >= 100 && color < 180) {
        setColorToVertice(colorsAttr, i, 20, 108, 7);
    }
    else if(color >= 180 && color < 255) {
        setColorToVertice(colorsAttr, i, 105, 53, 1);
    }
    else {
        setColorToVertice(colorsAttr, i, 54, 28, 1);
    }
}
```

Εικόνα 17: Κώδικας αρχείου colorHelper.js.

Η συνάρτηση setColorValue() δέχεται 3 παραμέτρους, οι οποίες είναι:

- **colorsAttr**: Μια μεταβλητή τύπου TypedArray που περιέχει την χρωματική πληροφορία για κάθε σημείο του τοπίου.
- **i**: Είναι η θέση του εκάστοτε σημείου μέσα στο colorsAttr.
- **color**: Είναι η τιμή που επιστρέφεται από τον Simplex Noise, ο οποίος στην εφαρμογή αυτή επιστρέφει τιμές από το -1 μέχρι το 1, πολλαπλασιασμένη με το 255.

Βάσει της τιμής του color, εκτελείται η συνάρτηση setColorToVertice(), με διαφορετικές τιμές r, g, b. Να σημειωθεί ότι για κάθε σημείο, αποθηκεύονται 3 τιμές στην μεταβλητή colorsAttr, με την δομή της να είναι όπως παρακάτω:



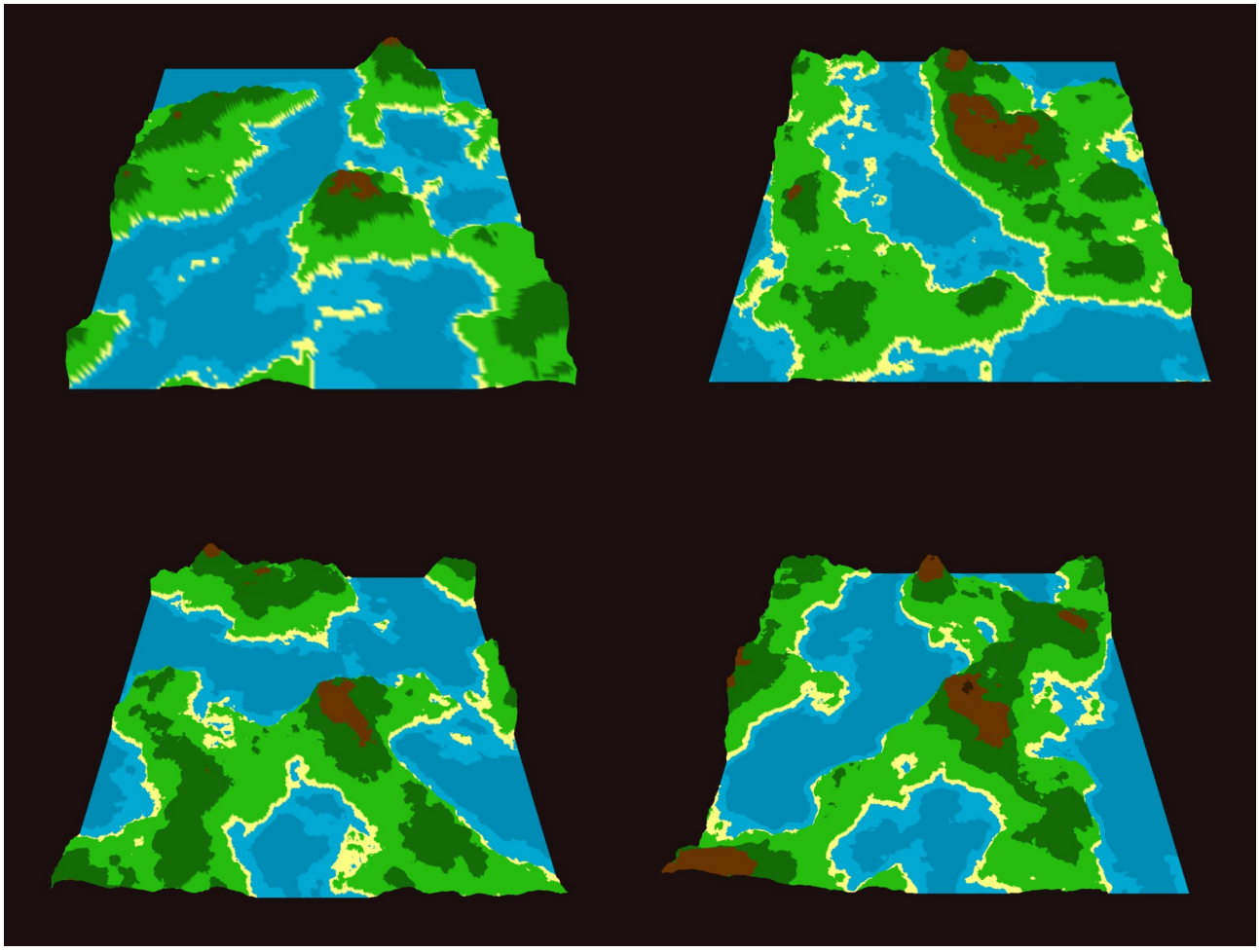
Εικόνα 18: Απεικόνιση δομής στοιχείων της μεταβλητής colorsAttr.

Στην συνέχεια, έχουμε το αρχείο constants.js, με τις σταθερές που φαίνονται παρακάτω:

```
export const SCENE_BACKGROUND = 0x1b0e0e;
export const AMBIENT_LIGHT_COLOR = 0xffffff;
export const PLANE_SIZE_OPTIONS = [1024];
export const SEGMENTS_OPTIONS = [128, 256, 512, 1024];
export const OCTAVES = [16, 32, 64, 128, 256];

export const SMOOTHNESS = 150;
export const FACTOR = 600;

export const CAMERA_PARAMS = {
  fov: 60,
  aspect: window.innerWidth / window.innerHeight,
  near: 0.1,
  far: 10000,
};
```

Εικόνα 19: Κώδικας αρχείου constants.js.

Οι δύο πρώτες τιμές **SCENE_BACKGROUND** και **AMBIENT_LIGHT_COLOR** αφορούν το φόντο της εφαρμογής και το χρώμα του περιβάλλοντος φωτισμού. Η μεταβλητή **PLANE_SIZE_OPTIONS** έχει να κάνει με το μέγεθος του τοπίου. Η τιμή 1024 σημαίνει ότι το τοπίο που θα παραχθεί θα είναι 1024x1024. Εδώ χρησιμοποιείται μόνο η τιμή 1024, καθώς εξυπηρετεί καλύτερα τον σκοπό της εργασίας. Τα **SEGMENT_OPTIONS** έχουν να κάνουν με το επίπεδο λεπτομέρειας του τοπίου, όπως φαίνεται στην παρακάτω εικόνα:



Εικόνα 20: Παραδείγματα τοπίων για 128, 256, 512, 1024 segments.

Τα segments είναι χαρακτηριστικό των Geometry αντικειμένων της three.js και ορίζουν των αριθμό των τμημάτων στα οποία θα χωριστεί ένα Geometry [43]. Στην παραπάνω εικόνα βλέπουμε πάνω αριστερά ότι το τοπίο με τα 128 segments φαίνεται θολό και με λιγότερες λεπτομέρειες, σε αντίθεση με το τοπίο που βρίσκεται κάτω δεξιά, το οποίο με 1024 segments έχει ένα υψηλό επίπεδο λεπτομέρειας.

Οι μεταβλητές **OCTAVES**, **SMOOTHNESS** και **FACTOR** είναι παράμετροι του Simplex Noise και αφορούν το εικαστικό αποτέλεσμα του τοπίου. Τέλος, η μεταβλητή **CAMERA_PARAMS** είναι οι ρυθμίσεις της κάμερας της εφαρμογής σχετικά με το τμήμα της εφαρμογής που μπορεί να είναι ορατό στον χρήστη.

Προχωρώντας στο αρχείο script.js, θα δούμε τον τρόπο που έχει υλοποιηθεί η εφαρμογή, αναφορικά με την αρχικοποίηση των αντικειμένων της three.js και την ενημέρωση των γραφικών, ο οποίος είναι παρόμοιος με τον τρόπο που έχουν στηθεί και οι υπόλοιπες εφαρμογές στην συνέχεια του κεφαλαίου. Η παρακάτω εικόνα απεικονίζει την δομή που ακολουθεί η εφαρμογή:

```
1  const stats = initStats();
2  const renderer = initWebGLRenderer();
3  const camera = initCamera(CONSTANTS.CAMERA_PARAMS);
4  const controls = new OrbitControls(camera, renderer.domElement);
5  const axesHelper = new THREE.AxesHelper(200);
6  const scene = new THREE.Scene();
7  const planeMesh = new THREE.Mesh();
8  const light = new THREE.AmbientLight(CONSTANTS.AMBIENT_LIGHT_COLOR);
9  const gui = new GUI();
10
11 /*
12  * Κώδικας για την δημιουργία των UI controls που παραλείπεται.
13  */
14
15 scene.background = new THREE.Color(CONSTANTS.SCENE_BACKGROUND);
16
17 camera.lookAt(scene.position);
18
19 generateTerrain(planeMesh, planeSize, segments, wireFrameOn, octaves);
20
21 planeMesh.rotateX(Math.PI / 2);
22
23 scene.add(light);
24 scene.add(planeMesh);
25 scene.add(axesHelper);
26
27 function render() {
28   requestAnimationFrame(render);
29   renderer.render(scene, camera);
30   stats.update();
31   controls.update();
32 }
33
34 render();
```

Εικόνα 21: Κώδικας αρχείου script.js

Οι πρώτες 9 γραμμές είναι η αρχικοποίηση των αντικειμένων της εφαρμογής. Η συνάρτηση initStats() προσθέτει στην εφαρμογή μας ένα στοιχείο το οποίο μας ενημερώνει για τον αριθμό των FPS στα οποία εκτελείται η εφαρμογή μας. Η initWebGLRenderer έχει να κάνει με την δημιουργία πλαισίου (context) το οποίο θα χρησιμοποιήσει η three.js για την απεικόνιση των γραφικών. Στην συνέχεια αρχικοποιείται η κάμερα με την συνάρτηση initCamera(), δέχοντας ως παραμέτρους τις ρυθμίσεις από το αρχείο constants.js. Οι υλοποιήσεις των συναρτήσεων αυτών φαίνονται παρακάτω:

```
function initCamera(params) {
  const camera = new THREE.PerspectiveCamera(
    params.fov,
    params.aspect,
    params.near,
    params.far
  );
  camera.position.set(0, 2048, 1600);
  camera.lookAt(512, 0, 512);
  return camera;
}

function initStats() {
  const fpsStats = Stats();
  document.body.appendChild(fpsStats.dom);
  return fpsStats;
}

function initWebGLRenderer() {
  const webGLRenderer = new THREE.WebGLRenderer({ antialias: true });
  webGLRenderer.setPixelRatio(window.devicePixelRatio);
  webGLRenderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(webGLRenderer.domElement);
  return webGLRenderer;
}
```

Εικόνα 22: Υλοποιήσεις συναρτήσεων initCamera(), initStats() και initWebGLRenderer().

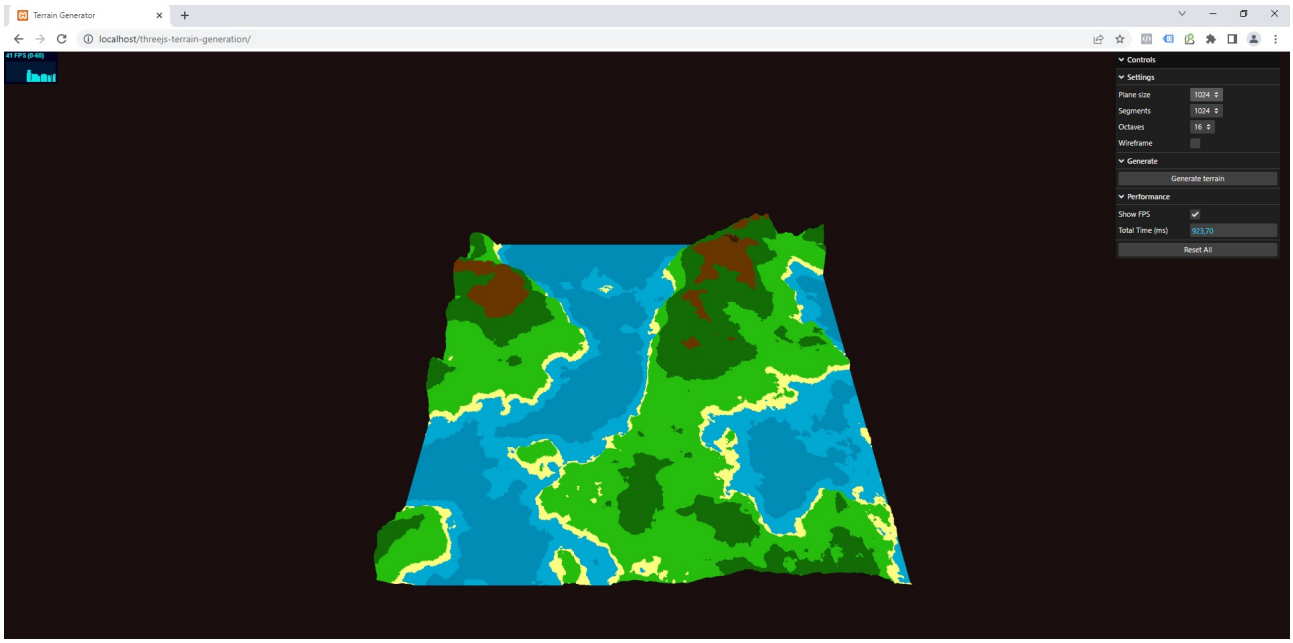
Η συνάρτηση η οποία παράγει το 3D τοπίο είναι η generateTerrain(). Όπως βλέπουμε και στην εικόνα παρακάτω, δέχεται σαν παραμέτρους το planeMesh, που είναι το Mesh αντικείμενο στο οποίο θα σχηματιστεί το τοπίο, το planeSize που αφορά το μέγεθος του τοπίου (στην προκειμένη περίπτωση είναι 1024), το αριθμό των segments, και οι επιλογές wireframeOn και octaves είναι για δοκιμαστικούς σκοπούς και αφορούν αν το τοπίο σχεδιαστεί με πλέγμα (wireframe) για λόγους debugging και το πόσες εσωτερικές επαναλήψεις θα κάνει ο Simplex Noise για κάθε στοιχείο, με την τιμή της παραμέτρου octaves.

```
1  function generateTerrain(mesh, planeSize, segments, wireFrameOn, octaves) {
2
3    mesh.geometry.dispose();
4
5    const geometry = new THREE.PlaneGeometry(planeSize, planeSize, segments, segments);
6    const material = new THREE.MeshLambertMaterial({
7      side: THREE.DoubleSide,
8      wireframe: wireFrameOn,
9      vertexColors: !wireFrameOn,
10     flatShading: true,
11   });
12
13   mesh.geometry = geometry;
14   mesh.material = material;
15
16   const vertices = mesh.geometry.attributes.position.array;
17   const colorsAttr = mesh.geometry.attributes.position.clone();
18   const simplexNoise = new SimplexNoise();
19
20   let noise = 0, color = 0;
21
22   const timeStart = performance.now();
23
24   for (let i = 0; i < vertices.length; i += 3) {
25     noise = simplexNoise.calcNoise(vertices[i] / CONSTANTS.FACTOR, vertices[i + 1] / CONSTANTS.FACTOR, octaves);
26     color = 255 * noise;
27     if (noise < 0) noise = 0;
28     vertices[i + 2] = -CONSTANTS.SMOOTHNESS * noise;
29     setColorValue(colorsAttr, i, color);
30   }
31
32   const timeEnd = performance.now();
33
34   let totalTime = timeEnd - timeStart;
35
36   mesh.geometry.attributes.position.needsUpdate = true;
37   mesh.geometry.computeVertexNormals();
38   mesh.geometry.setAttribute("color", colorsAttr);
39   mesh.material.needsUpdate = true;
40
41   return totalTime.toFixed(2);
42 }
```

Εικόνα 23: Συνάρτηση generateTerrain().

Ξεκινώντας από την γραμμή 3 και με την εντολή mesh.geometry.dispose(), διαγράφουμε όλες τις πληροφορίες των σημείων του Geometry αντικειμένου για το συγκεκριμένο mesh, προκειμένου να το ενημερώσουμε με τις καινούργιες τιμές. Στις γραμμές 5 με 14, δημιουργούμε νέο Geometry και Material, τα οποία θα περιέχουν την μορφολογία και την χρωματική πληροφορία του 3D τοπίου. Στην συνέχεια, στις γραμμές 16 με 18, παίρνουμε το array με τα σημεία του νέου Geometry (vertices) και δημιουργούμε ένα αντίγραφο από το array αυτό για να αρχικοποιήσουμε και το array με την χρωματική πληροφορία του κάθε σημείου. Το σημαντικότερο τμήμα αυτής της συνάρτησης βρίσκεται στις γραμμές 24 με 30, όπου και γίνεται ο υπολογισμός και η ανάθεση τιμών σε κάθε σημείο. Αν παρατηρήσουμε τον βρόχο for, θα δούμε ότι το step είναι 3. Αυτό συμβαίνει επειδή το array με τα σημεία είναι μονοδιάστατο (1D) και εσωτερικά κάθε σημείο στο array έχει 3 συντεταγμένες (x, y, z), ακριβώς όπως το colorsAttr που είδαμε παραπάνω (βλέπε Εικόνα 3.4).

Επειδή σκοπός είναι να δώσουμε ύψος σε κάθε σημείο, ενημερώνουμε μόνο την συντεταγμένη z, όπως φαίνεται και στην γραμμή 28. Τέλος, στις γραμμές 36 με 39, γίνεται η ενημέρωση των σημείων με τα νέα ύψη και τις νέες χρωματικές πληροφορίες.



Εικόνα 24: Στιγμιότυπο της εφαρμογής Terrain Generator.

3.1.2 Υλοποίηση με Web Workers

Όπως μπορούμε να δούμε από την προηγούμενη ενότητα, η συνάρτηση generateTerrain() υπολογίζει την τιμή του noise για κάθε σημείο του Geometry μέσα σε ένα βρόχο for. Σε περιπτώσεις μεγάλων διαστάσεων, όπως 1024x1024 που εξετάζουμε, και των διαφόρων τιμών των segments, το πλήθος των σημείων για τα οποία πρέπει να υπολογιστεί η τιμή του noise, ολοένα και αυξάνεται. Ο παρακάτω πίνακας αποτυπώνει τον αριθμό των σημείων για κάθε τιμή του segment:

Πλήθος Segments	**Αριθμός στοιχείων**
128	16641
256	66049
512	263169
1024	1050625

Πίνακας 1: Αριθμός σημείων για κάθε πλήθος segments.

Από τον παραπάνω πίνακα παρατηρούμε ότι για κάθε διαφορετική τιμή της μεταβλητής segments, ο αριθμός των σημείων σχεδόν τετραπλασιάζεται. Συνεπώς και η αύξηση του χρόνου που απαιτείται για τους υπολογισμούς θα είναι αντίστοιχη. Μπορούμε να επιταχύνουμε την διαδικασία αυτή, αλλάζοντας την υλοποίηση ώστε να αξιοποιούνται οι web workers. Για να το κάνουμε αυτό, θα χρειαστεί να δημιουργήσουμε ένα νέο αρχείο με όνομα **worker.js**, το οποίο θα περιέχει την λογική υπολογισμού του noise για κάθε σημείο. Πριν προχωρήσουμε στην δημιουργία του αρχείου, θα πάμε στο constants.js και θα το αλλάξουμε όπως στην παρακάτω εικόνα:

```
1  export const SCENE_BACKGROUND = 0x1b0e0e;
2  export const AMBIENT_LIGHT_COLOR = 0xffffff;
3  export const PLANE_SIZE_OPTIONS = [1024];
4  export const SEGMENTS_OPTIONS = [128, 256, 512, 1024];
5  export const WORKERS_OPTIONS = [1, 2, 4, 8, 16];
6  export const OCTAVES = [16, 32, 64, 128, 256];
7
8  export const SMOOTHNESS = 150;
9  export const FACTOR = 600;
```

Εικόνα 25: Ανανεωμένος κώδικας αρχείου constants.js.

Αυτό που έχουμε προσθέσει είναι η μεταβλητή **WORKER_OPTIONS**, η οποία προστίθεται και στα UI controls ώστε να μπορούμε να αλλάζουμε το πλήθος των web workers για να γίνονται ευκολότερα οι εκτελέσεις του αλγορίθμου και οι μετρήσεις.

Στην συνέχεια, δημιουργούμε το αρχείο worker.js, ο κώδικας του οποίου είναι ο ακόλουθος:

```
import SimplexNoise from "./simplexNoise.js";
import { setColorValue } from "./colorHelper.js";
import * as CONSTANTS from "./constants.js";

onmessage = (message) => {
  let vertices = message.data[0];
  let colors = message.data[1];
  const simplexNoiseParams = message.data[2];
  const octaves = message.data[3];
  const simplexNoise = new SimplexNoise(simplexNoiseParams);

  let newVertices = new Float32Array(vertices.length);
  let noise = 0,
    color = 0;

  for (let i = 0; i < vertices.length; i += 3) {
    noise = simplexNoise.calcNoise(
      vertices[i] / CONSTANTS.FACTOR,
      vertices[i + 1] / CONSTANTS.FACTOR,
      octaves
    );

    color = 255 * noise;
    if (noise < 0) noise = 0;

    vertices[i + 2] = -CONSTANTS.SMOOTHNESS * noise;

    setColorValue(colors, i, color);

    newVertices[i] = vertices[i];
    newVertices[i + 1] = vertices[i + 1];
    newVertices[i + 2] = vertices[i + 2];
  }

  postMessage([newVertices, colors]);
};
```

Εικόνα 26: Κώδικας αρχείου worker.js.

Αφού κάνουμε import όλες τις απαραίτητες κλάσεις για τους υπολογισμούς, καλούμε την onmessage(), έτσι ώστε να μπορούμε να λαμβάνουμε μηνύματα από το main thread. Η onmessage δέχεται την παράμετρο message, η οποία έχει το χαρακτηριστικό (property) data, στο οποίο είναι αποθηκευμένα σε μορφή array όλα τα δεδομένα που θέλουμε να στείλουμε. Συγκεκριμένα, έχουμε:

- **vertices**: Είναι τα σημεία του Geometry αντικειμένου.
- **colors**: Αφορά την χρωματική πληροφορία του κάθε αντικειμένου.
- **simplexNoiseParams**: Είναι κάποιες επιπλέον παράμετροι που απαιτούνται για την σωστή λειτουργία του αλγορίθμου Simplex Noise με web workers.

- **octaves**: Παράμετρος του Simplex Noise που αφορά τον αριθμό των εσωτερικών επαναλήψεών του για κάθε σημείο.

Στην γραμμή 12 αρχικοποιούμε ένα νέο TypedArray αντικείμενο, το **newVertices**, το οποίο είναι τύπου Float32Array [44] και έχει μήκος όσο το μήκος του array των vertices που έχει λάβει ο worker. Σε αυτή την μεταβλητή θα αποθηκεύσουμε για κάθε σημείο του Geometry αντικειμένου, τις τιμές noise που έχουν παραχθεί. Οι γραμμές 13 έως 33 ακολουθούν την ίδια λογική με την σειριακή υλοποίηση της προηγούμενης ενότητας. Τέλος, εφόσον πραγματοποιηθούν οι υπολογισμοί και ενημερωθούν οι νέες τιμές, χρησιμοποιούμε την μέθοδο postMessage() ώστε να στείλουμε τα νέα δεδομένα στο main thread. Σαν παράμετρο δέχεται ένα array 2 στοιχείων, όπου το πρώτο είναι οι νέες τιμές noise των σημείων και το δεύτερο είναι η νέα χρωματική πληροφορία για κάθε σημείο.

Έχοντας τελειώσει με την λογική του web worker, επιστρέφουμε στο αρχείο script.js και προσθέτουμε μια νέα βοηθητική συνάρτηση, την initWorkers().

```
1  function initWorkers(numOfWorkers) {
2    let workers = [];
3    for (let i = 0; i < numOfWorkers; i++) {
4      workers.push(new Worker("worker.js", { type: "module" }));
5    }
6    return workers;
7  }
```

Εικόνα 27: Κώδικας συνάρτησης initWorkers().

Σκοπός της παραπάνω συνάρτησης είναι να δημιουργήσει ένα pool από web workers οι οποίοι θα υπάρχουν πριν την εκτέλεση του παράλληλου κώδικα, και δεν θα δημιουργούνται εκείνη την στιγμή. Ο λόγος που το κάνουμε αυτό είναι το διαφορετικό browsing context που πρέπει να δημιουργηθεί για κάθε web worker (βλέπε κεφάλαιο 2), το οποίο επιβαρύνει τον συνολικό χρόνο εκτέλεσης, προσθέτοντας overhead.

Με την παράμετρο **numOfWorkers**, ορίζουμε τον αριθμό των web workers που θέλουμε να προστεθούν στο pool. Να σημειωθεί ότι κάθε φορά που καλείται αυτή η συνάρτηση, το pool με τους web workers αδειάζει και δημιουργείται ένα καινούργιο.

Στην γραμμή 4 προστίθεται ο νέος worker, με τις παραμέτρους {type: “module”}, ώστε να μπορούμε να χρησιμοποιούμε, όπως αναφέραμε στην ενότητα των web workers στο κεφάλαιο 2, τις

εντολές για import κλάσεων. Στην συνέχεια, αλλάζουμε το τμήμα του κώδικα που παράγει το νέο τοπίο όπως στην παρακάτω εικόνα:

```
1  planeMesh.geometry.dispose();
2  const geometry = new THREE.PlaneGeometry(planeSize, planeSize,  segments, segments);
3  const material = new THREE.MeshLambertMaterial({
4    side: THREE.DoubleSide,
5    wireframe: wireFrameOn,
6    vertexColors: !wireFrameOn,
7    flatShading: true,
8  });
9  planeMesh.geometry = geometry;
10 planeMesh.material = material;
11
12 const simplexNoiseParams = getSimplexNoiseParams();
13
14 const timeStart = performance.now();
15
16 let workersFinished = 0;
17 let vertices = planeMesh.geometry.attributes.position.array.slice();
18 let colorsAttr = planeMesh.geometry.attributes.position.clone();
19 let colors = colorsAttr.array.slice();
20 let chunk = (Math.floor(vertices.length / (3 * numOfWorkers)) + 1) * 3;
21
22 for (let i = 0; i < numOfWorkers; i++) {
23   let currentOffset = i * chunk;
24   let currentChunk = Math.min(currentOffset + chunk, vertices.length);
25
26   workers[i].postMessage([
27     vertices.slice(currentOffset, currentChunk),
28     colors.slice(currentOffset, currentChunk),
29     simplexNoiseParams,
30     octaves,
31   ]);
32
33   workers[i].onmessage = (message) => {
34     planeMesh.geometry.attributes.position.array.set(message.data[0], currentOffset);
35     planeMesh.geometry.attributes.position.needsUpdate = true;
36
37     colorsAttr.array.set(message.data[1], currentOffset);
38
39     workersFinished++;
40     if (workersFinished == numOfWorkers) {
41       const timeEnd = performance.now();
42       let totalTime = timeEnd - timeStart;
43       totalTime = totalTime.toFixed(2);
44
45       totalTimeController.setValue(totalTime);
46
47       planeMesh.geometry.setAttribute("color", colorsAttr);
48       planeMesh.material.needsUpdate = true;
49     }
50   };
51 }
```

Εικόνα 28: Ενημερωμένος κώδικας για την παραγωγή τοπίου με την χρήση web workers.

Οι πρώτες 10 γραμμές είναι οι ίδιες γραμμές που χρησιμοποιούνται και στην σειριακή υλοποίηση, για την αρχικοποίηση των Geometry και Material αντικειμένων. Στις γραμμές 16 έως 20 ξεκινάει η

αρχικοποίηση των μεταβλητών που αφορούν τους web workers. Συγκεκριμένα, είναι οι μεταβλητές:

- **workersFinished**, η οποία μετράει τον αριθμό των web workers που έχουν τελειώσει με την εκτέλεση της διαδικασίας για το τμήμα των σημείων που τους αναλογεί.
- **chunk**, είναι η μεταβλητή η οποία υπολογίζει το μέγεθος του τμήματος των vertices και colors που θα ανατεθεί σε κάθε web worker, ώστε να ενημερωθεί από αυτόν. Να θυμίσουμε ότι τα arrays των vertices και colors είναι μονοδιάστατα και για κάθε σημείο, έχουμε 3 τιμές για τις συντεταγμένες (x, y, z) και 3 τιμές για την χρωματική πληροφορία (r, g, b).

Στην συνέχεια, στην γραμμή 22, περνάμε στον βρόχο for, όπου αναθέτουμε και στέλνουμε το τμήμα των σημείων σε κάθε web worker και συλλέγουμε τα αποτελέσματα. Αυτό φαίνεται στις γραμμές 23 και 24, στις οποίες γίνεται ο υπολογισμός του δείκτη (index) του σημείου από το οποίο αρχίζει το τμήμα που θα σταλεί στον web worker, με την μεταβλητή **currentOffset**, καθώς και το μέγεθος του κάθε τμήματος με την μεταβλητή **currentChunk**. Ο λόγος που χρησιμοποιούμε την συνάρτηση Math.min(), είναι για την περίπτωση που το μήκος των vertices και colors δεν είναι πολλαπλάσιο του αριθμού των web workers και πρέπει να διαχειριστούμε το υπόλοιπο των στοιχείων, των οποίων το μήκος θα είναι μικρότερο του currentChunk, για να σταλούν στον τελευταίο web worker.

Έχοντας υπολογίσει τα παραπάνω μεγέθη, κάνουμε κλήση στην συνάρτηση postMessage() στην γραμμή 26, στέλνοντας το αντίστοιχο τμήμα των vertices, colors και κάποιων επιπλέον μεταβλητών που χρειάζονται για τον Simplex Noise. Οι web workers θα εκτελέσουν τον Simplex Noise και θα επιστρέψουν με την σειρά τους τα αποτελέσματα. Το main thread μπορεί να λάβει τα αποτελέσματα ενός web worker με το onmessage() event στην γραμμή 33. Κάθε φορά που ολοκληρώνεται η εκτέλεση ενός web worker, η μεταβλητή workersFinished αυξάνεται κατά 1. Όταν γίνει ίση με τον αριθμό των workers, σημαίνει ότι έχει ολοκληρωθεί η εκτέλεση και μπορούν να ενημερωθούν τα γραφικά.

3.1.3 Σύγκριση αποτελεσμάτων

Για να φανεί η χρησιμότητα των web workers στην επιτάχυνση των υπολογισμών, πραγματοποιήθηκαν μετρήσεις στους τέσσερις browsers που αναφέραμε στην αρχή του κεφαλαίου. Για κάθε browser, οι δύο εφαρμογές έτρεξαν από 100 επαναλήψεις, εξετάζοντας την εκτέλεση με 2,

4, 8 και 16 web workers. Τα αποτελέσματα των μετρήσεων του συνολικού χρόνου παρουσιάζονται για κάθε browser στους παρακάτω πίνακες:

	128	**256**	**512**	**1024**
Single thread	18,60	64,75	237,37	913,78
2	17,01	58,16	191,81	704,85
4	15,37	47,80	148,84	583,32
8	15,45	47,15	178,50	616,63
16	17,02	49,09	183,88	646,47

Πίνακας 2: Χρόνοι εκτέλεσης στον Chrome.

	128	**256**	**512**	**1024**
Single thread	19,57	69,88	238,52	917,47
2	18,22	53,02	176,98	664,76
4	19,94	48,01	150,93	600,84
8	17,04	48,81	166,87	669,2
16	18,17	53,22	177,36	683,33

Πίνακας 3: Χρόνοι εκτέλεσης στον Edge.

	128	**256**	**512**	**1024**
Single thread	26,67	106,14	400,38	1598,49
2	20,95	71,36	240,01	913,42
4	19,81	69,62	211,71	935,34
8	21,6	69,53	227,43	1131,1
16	23,18	68,78	261,77	1250,74

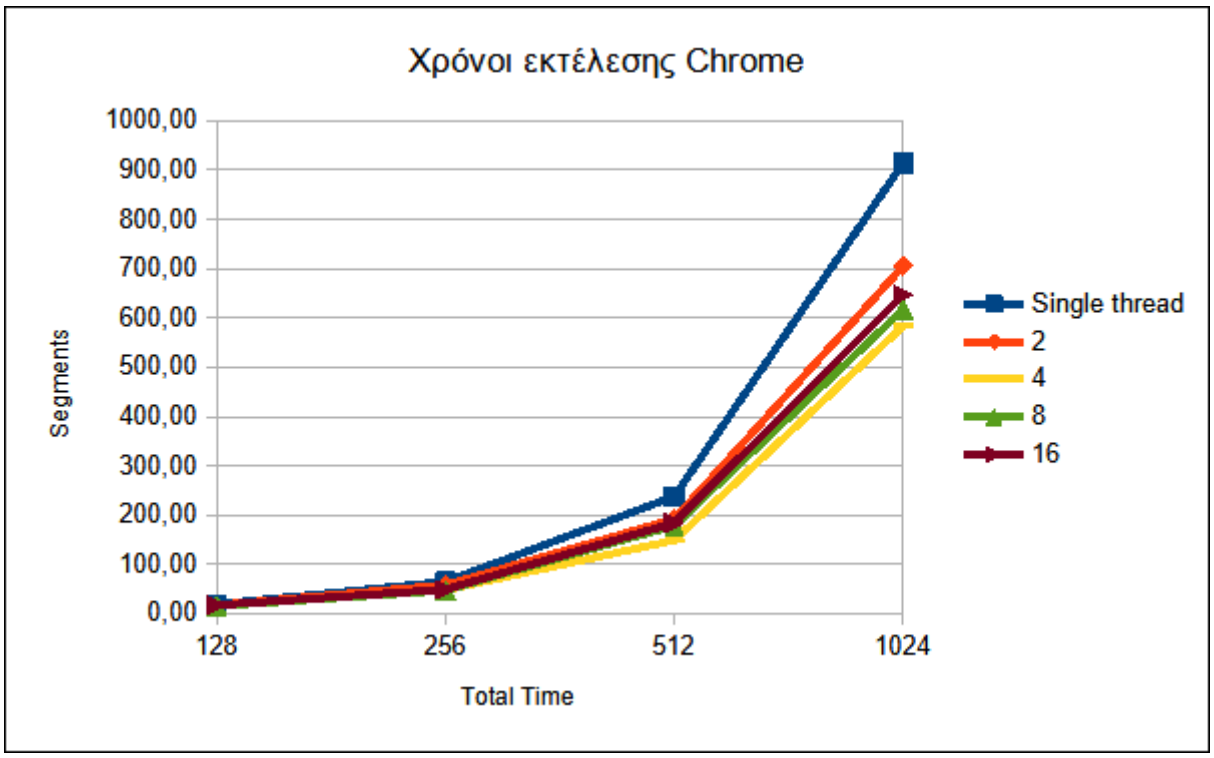
Πίνακας 4: Χρόνοι εκτέλεσης στον Firefox.

	128	**256**	**512**	**1024**
Single thread	19,12	64,74	236,42	918,87
2	19,98	52,64	183,24	656,23
4	16,47	45,88	154,23	597,69
8	16,33	48,2	172,32	637,45
16	17,92	51,04	176,47	669,07

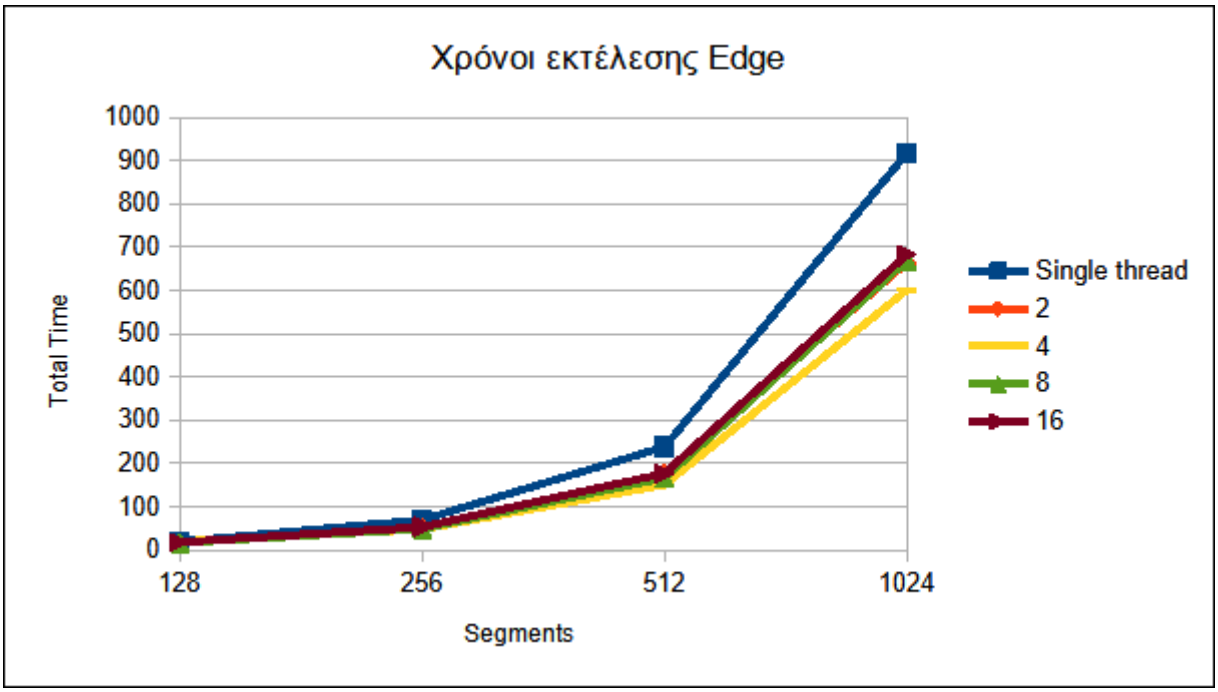
Πίνακας 5: Χρόνοι εκτέλεσης στον Opera.

Οι δοκιμές έγιναν σε τοπίο διαστάσεων 1024x1024 και σε κάθε πίνακα η πρώτη γραμμή δηλώνει τον αριθμό των segments και η πρώτη στήλη δηλώνει το main thread και τον αριθμό των web workers για κάθε περίπτωση.

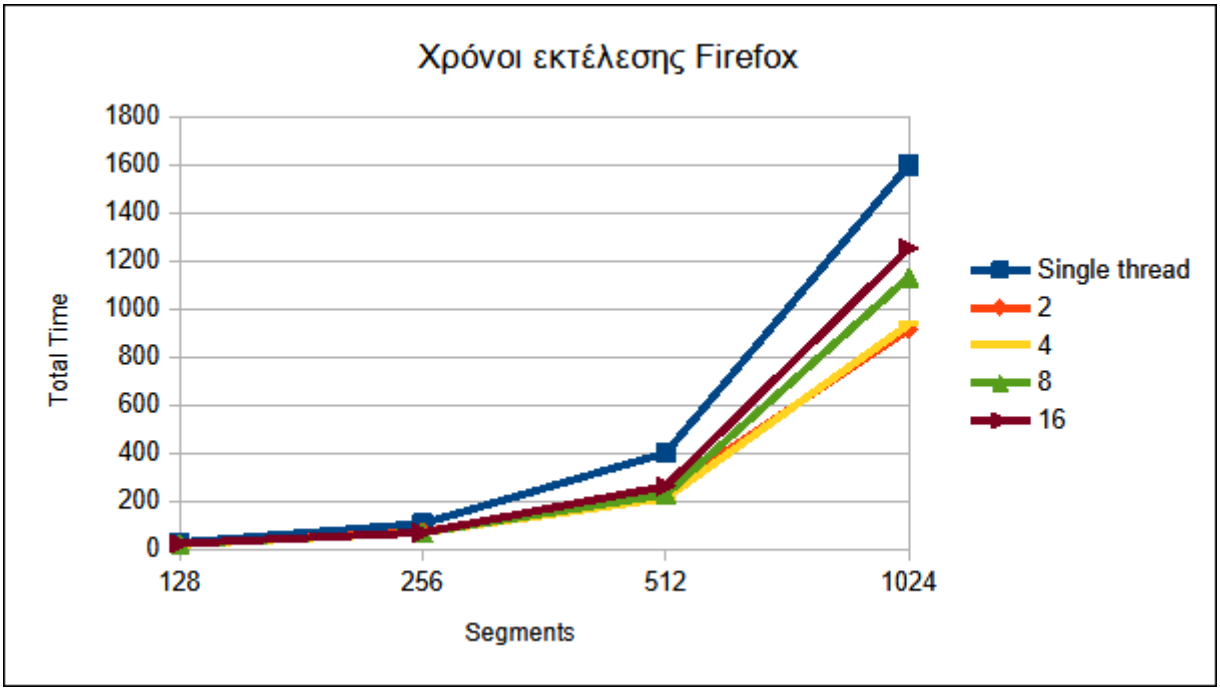
Ξεκινώντας από τα 128 segments, παρατηρείται ελάχιστη βελτίωση στην πλειοψηφία των browsers, με λίγη μεγαλύτερη διαφορά στην περίπτωση του Firefox. Συνεχίζοντας στα 256 segments, βλέπουμε μια μικρή βελτίωση στους χρόνους, με τον Firefox πάλι να αξιοποιεί καλύτερα τους web workers όπως θα δούμε και παρακάτω στα διαγράμματα με τις επιταχύνσεις. Στα 512 segments φαίνεται περισσότερο η χρησιμότητα των web workers, με την διαφορά ανάμεσα στην single thread εκτέλεση και τις υπόλοιπες να αυξάνεται, φτάνοντας τέλος στα 1024 segments όπου και πετυχαίνουμε τις μεγαλύτερες διαφορές από όλες τις περιπτώσεις εκτέλεσης. Στις περισσότερες περιπτώσεις παρατηρούμε ότι όσο ο αριθμός των web workers αυξάνεται και πλησιάζει τον αριθμό των πυρήνων της CPU, έχουμε καλύτερα αποτελέσματα, με τα βέλτιστα αποτελέσματα να επιτυγχάνονται όταν ο αριθμός των web workers είναι ίσος με τον αριθμό των πυρήνων του επεξεργαστή, δηλαδή 4. Από αυτό το σημείο και έπειτα, η αύξηση του αριθμού των web workers εξακολουθεί μεν να δίνει καλύτερα αποτελέσματα από την single threaded εκτέλεση, αλλά είναι χειρότερα από τα αποτελέσματα με μικρότερο πλήθος web workers. Θα μπορούσαμε να συνεχίσουμε και σε μεγαλύτερα μεγέθη segments, ωστόσο δεδομένων των τεχνικών χαρακτηριστικών του σταθερού υπολογιστή στον οποίο εκτελούνται οι μετρήσεις, υπάρχουν περιορισμοί στους υπολογιστικούς πόρους που καθυστερούν τις εκτελέσεις και δυσκολεύουν τις μετρήσεις. Τα παραπάνω αποτελέσματα μπορούμε να τα δούμε και με την μορφή διαγραμμάτων, όπως παρακάτω:



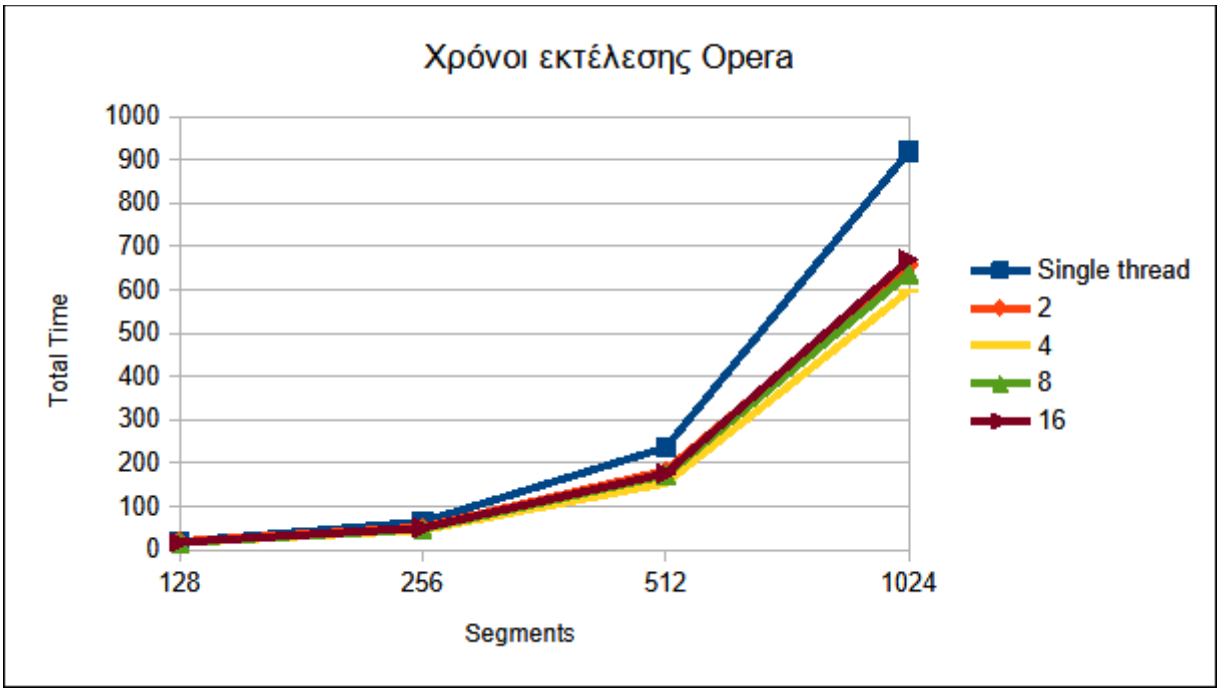
Εικόνα 29: Διάγραμμα χρόνων εκτέλεσης του Chrome.



Εικόνα 30: Διάγραμμα χρόνων εκτέλεσης του Edge.



Εικόνα 31: Διάγραμμα χρόνων εκτέλεσης του Firefox.



Εικόνα 32: Διάγραμμα χρόνων εκτέλεσης του Opera.

Στους παρακάτω πίνακες, προς συμπλήρωση των χρόνων εκτέλεσης, μπορούμε να δούμε την επιτάχυνση που έχουμε σε κάθε περίπτωση εκτέλεσης με την χρήση web workers, ως προς την single threaded εφαρμογή.

	128	**256**	**512**	**1024**
Single thread	1,00	1,00	1,00	1,00
2	1,09	1,11	1,24	1,30
4	1,21	1,35	1,59	1,57
8	1,20	1,37	1,33	1,48
16	1,09	1,32	1,29	1,41

Πίνακας 2: Πίνακας τιμών επιτάχυνσης του Chrome.

	128	**256**	**512**	**1024**
Single thread	1,00	1,00	1,00	1,00
2	1,07	1,32	1,35	1,38
4	0,98	1,46	1,58	1,53
8	1,15	1,43	1,43	1,37
16	1,08	1,31	1,34	1,34

Πίνακας 3: Πίνακας τιμών επιτάχυνσης του Edge.

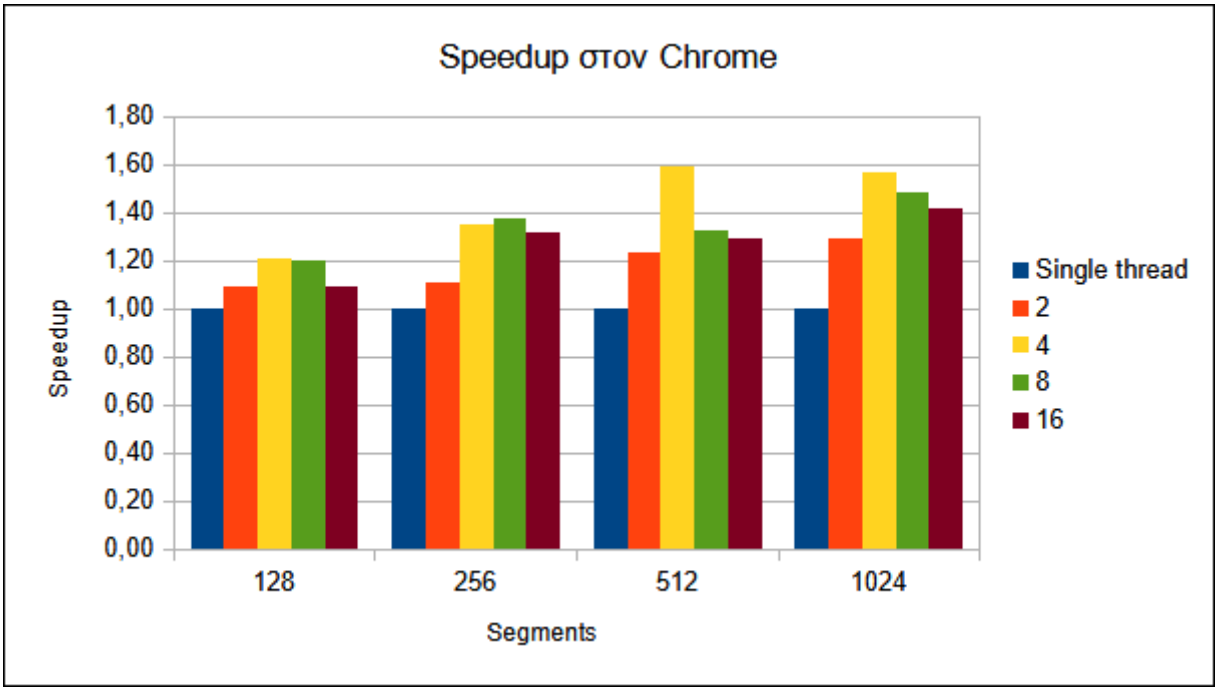
	128	**256**	**512**	**1024**
Single thread	1,00	1,00	1,00	1,00
2	1,27	1,49	1,67	1,75
4	1,35	1,52	1,89	1,71
8	1,23	1,53	1,76	1,41
16	1,15	1,54	1,53	1,28

Πίνακας 4: Πίνακας τιμών επιτάχυνσης του Firefox.

	128	256	512	1024
Single thread	1,00	1,00	1,00	1,00
2	0,96	1,23	1,29	1,40
4	1,16	1,41	1,53	1,54
8	1,17	1,34	1,37	1,44
16	1,07	1,27	1,34	1,37

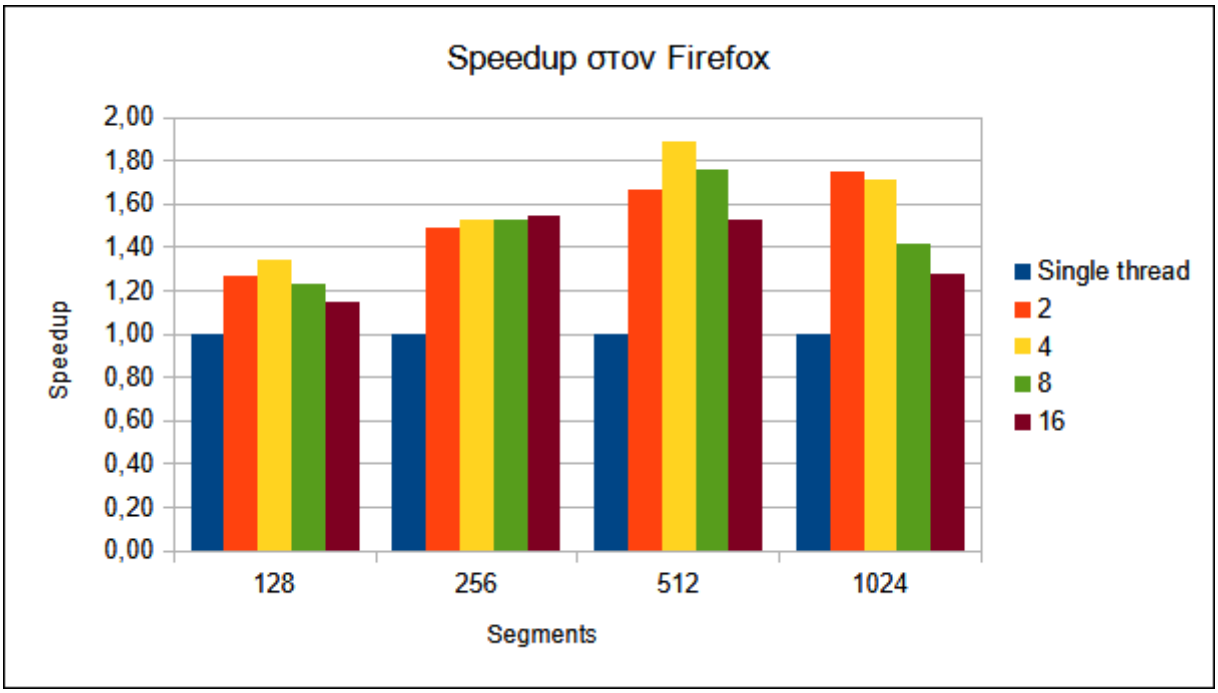
Πίνακας 5: Πίνακας τιμών επιτάχυνσης του Opera.

Παρατηρώντας τους πίνακες, μπορούμε να δούμε ότι στην συντριπτική πλειοψηφία των περιπτώσεων, η επιτάχυνση είναι μεγαλύτερη όταν ο αριθμός των web workers που εκτελούνται είναι ίσος με τον αριθμό των πυρήνων της CPU του συστήματος. Αυτό φαίνεται από την περίπτωση των 256 segments και έπειτα, καθώς η εκτέλεση της εφαρμογής είναι **1,5** έως **2** φορές ταχύτερη από την single threaded εκτέλεση, με τον Firefox να πετυχαίνει την μεγαλύτερη τιμή που είναι **1,89** στην περίπτωση των 512 segments, χρησιμοποιώντας 4 web workers. Σε γενικές γραμμές βλέπουμε οι επιταχύνσεις εκτέλεσης να είναι πολύ κοντά μεταξύ τους, με τον Firefox να πετυχαίνει τις καλύτερες τιμές. Χαρακτηριστική είναι, όπως είπαμε και στους χρόνους εκτέλεσης, η μείωση που παρουσιάζεται όταν ο αριθμός των web workers αυξάνεται και γίνεται μεγαλύτερος από τον αριθμό των πυρήνων της CPU. Για την καλύτερη κατανόηση, οι παραπάνω πίνακες μπορούν να απεικονιστούν και με διαγράμματα, όπως παρακάτω:

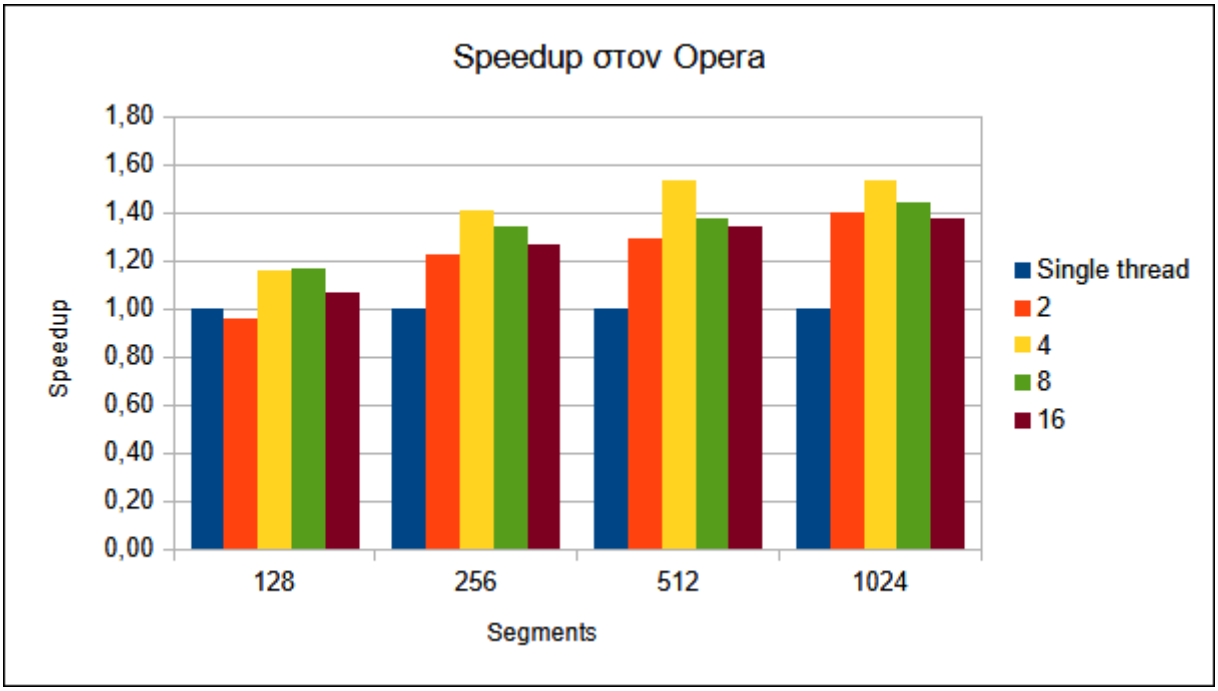


Εικόνα 33: Διάγραμμα επιτάχυνσης για τον Chrome.

Εικόνα 34: Διάγραμμα επιτάχυνσης για τον Edge.



Εικόνα 35: Διάγραμμα επιτάχυνσης για τον Firefox.



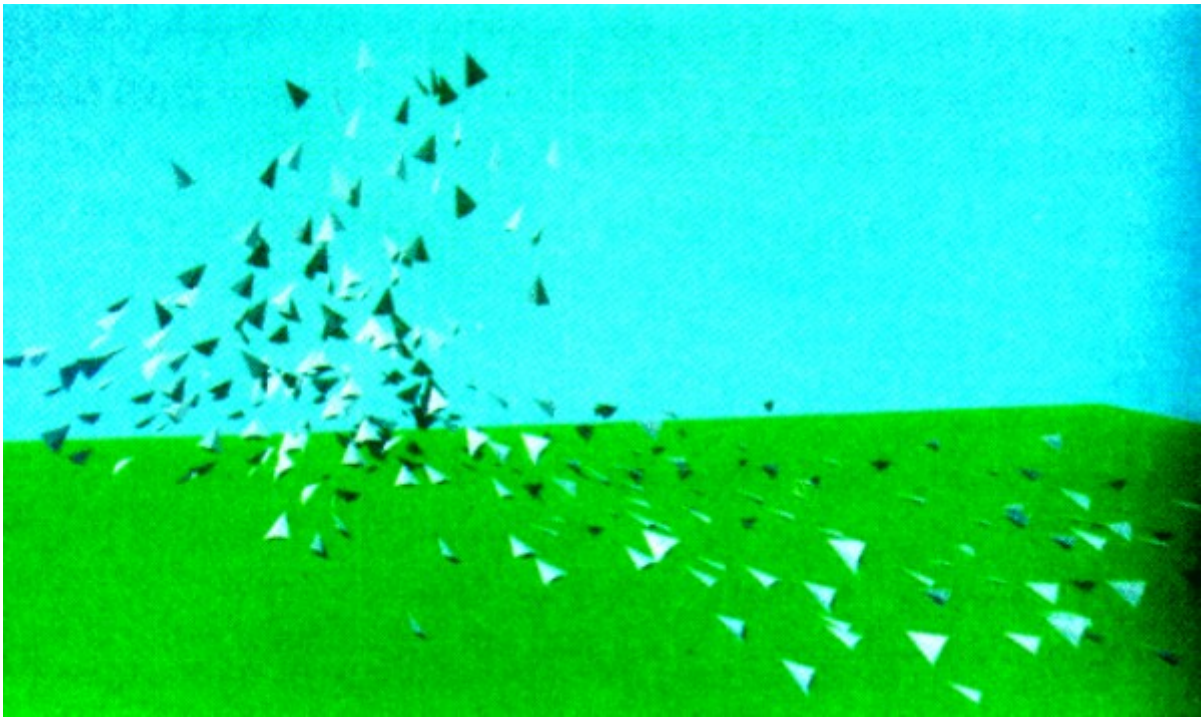
Εικόνα 36: Διάγραμμα επιτάχυνσης για τον Opera.

3.2 Εφαρμογή προσομοίωσης σμήνους (boid simulation)

Η δεύτερη εφαρμογή που θα παρουσιάσουμε αφορά την προσομοίωση σμήνους ή αλλιώς boid simulation. Πρόκειται για ένα μοντέλο το οποίο εστιάζει γύρω από την συμπεριφορά και την αλληλεπίδραση ενός συνόλου οντοτήτων (boids), μέσα από ορισμένους κανόνες [45]. Ανήκει στην γενικότερη κατηγορία των προσομοιώσεων σμήνους (flock simulation) και εισήχθη από τον προγραμματιστή και ειδικό σε θέματα γραφικών υπολογιστή Craig Reynolds το 1986, ως ένας εναλλακτικός τρόπος απεικόνισης γραφικών με πολλές οντότητες που αλληλεπιδρούν μεταξύ τους [46]. Ο όρος boid είναι συντόμευση του όρου bird-oid object.

Στην πιο απλή μορφή της, μια προσομοίωση με boids ορίζει την συμπεριφορά των οντοτήτων της με τρεις κανόνες:

- **Αποφυγή σύγκρουσης**: Κάθε boid της προσομοίωσης θα πρέπει να αποφύγει να συγκρουστεί με τα υπόλοιπα boids.
- **Αρμονία ταχύτητας**: Κάθε boid θα προσπαθήσει να αποκτήσει την ίδια ταχύτητα με την μέση ταχύτητα που έχει το υπόλοιπο σμήνος.
- **Κεντράρισμα**: Αφορά τις προσπάθειες του κάθε boid να παραμείνει κοντά με γειτονικά boids.



Εικόνα 37: Προσομοίωση boids. (Πηγή: https://team.inria.fr/imagine/files/2014/10/flocks-hers-and-schools.pdf)

Η πρόκληση την οποία καλούμαστε να αντιμετωπίσουμε στην συγκεκριμένη εφαρμογή, είναι η ελαχιστοποίηση του χρόνου υπολογισμού των νέων θέσεων και ταχυτήτων του κάθε boid σε σχέση με τα υπόλοιπα και η μεγιστοποίηση του ρυθμού ανανέωσης, που αποτυπώνεται στην τιμή των FPS. Ο τρόπος με τον οποίο λειτουργεί η εφαρμογή είναι να τρέχει πολλαπλές επαναλήψεις, σε κάθε μια από τις οποίες υπολογίζονται όλες οι θέσεις και οι ταχύτητες των boids, και αφού ολοκληρωθεί ο υπολογισμός, πραγματοποιείται η αντίστοιχη ενημέρωση των γραφικών, ξεκινώντας μετά μια νέα επανάληψη με την ίδια διαδικασία.

Θα παρουσιάσουμε πρώτα την σειριακή υλοποίηση, περιγράφοντας την λογική μέσα από τον κώδικα, και στην συνέχεια θα παρουσιαστεί και η υλοποίηση με web workers, καθώς και η χρήση του Offscreencanvas, για την μεταφορά της προσομοίωσης σε web worker ώστε να μην επηρεάζει την απόκριση της σελίδας. Τέλος, θα παρουσιάσουμε τα αποτελέσματα των μετρήσεων από την συγκριτική αξιολόγηση και θα ερμηνεύσουμε τα αποτελέσματα.

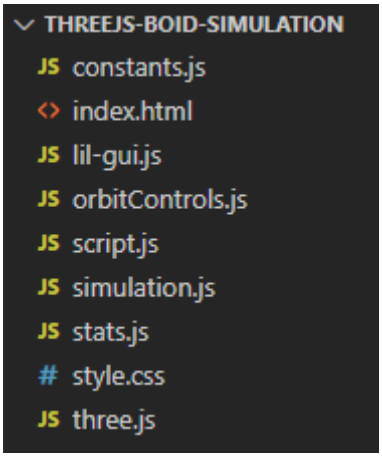
3.2.1 Σειριακή υλοποίηση

Η εκτέλεση της προσομοίωσης για το σύνολο των boids αποτελείται από δυο βρόχους for, όπου ο ένας είναι εμφωλευμένος στον άλλο. Για κάθε boid του συνόλου, πρέπει να γίνουν οι υπολογισμοί που αφορούν την αλληλεπίδρασή του με τα υπόλοιπα boids και βάσει των τιμών αυτών, να υπολογιστούν η νέα θέση και ταχύτητά του.

```
1  for boid in boids:
2    for every_other_boid in boids:
3      # Εκτέλεσε τους κανόνες αλληλεπίδρασης
4      # και υπολόγισε τις νέες τιμές θέσης και ταχύτητας.
5      pass
```

Εικόνα 38: Ψευδοκώδικας του διπλού βρόχου της προσομοίωσης.

Για την υλοποίηση της εφαρμογής, και της σειριακής και της παράλληλης, χρησιμοποιήθηκε η δομή των αρχείων που φαίνεται στην παρακάτω εικόνα:



Εικόνα 39: Δομή αρχείων της εφαρμογής.

Όπως βλέπουμε, χρησιμοποιούμε τις ίδιες βιβλιοθήκες με την προηγούμενη εφαρμογή που παρουσιάσαμε, με τα αρχεία lil-gui.js, orbitControls.js, stats.js και three.js να είναι τα ίδια. Επίσης πάλι έχουμε τα αρχεία constants.js και script.js τα οποία χειρίζονται τις παραμέτρους και την λογική

της εφαρμογής αντίστοιχα. Ξεκινώντας από το αρχείο constants.js έχουμε το περιεχόμενο της παρακάτω εικόνας:

```
1  export const CAMERA_PARAMS = {
2    fov: 60,
3    aspect: window.innerWidth / window.innerHeight,
4    near: 0.1,
5    far: 10000,
6  };
7
8  export const SCENE_BACKGROUND = 0x1b0e0e;
9
10 export const BOUND_SIZE_X = 5000;
11 export const BOUND_SIZE_Y = 5000;
12 export const BOUND_SIZE_Z = 5000;
13
14 export const TOTAL_BOIDS = 5000;
15 export const TOTAL_PROPERTIES = 6;
```

Εικόνα 40: Κώδικας αρχείου constants.js.

Οι μεταβλητές **CAMERA_PARAMS** και **SCENE_BACKGROUND** είναι ήδη γνωστές από την πρώτη εφαρμογή και αφορούν τις ρυθμίσεις της κάμερας του Scene στο οποίο θα γίνει η προσομοίωση και το φόντο του Scene της προσομοίωσης. Οι νέες μεταβλητές για την εφαρμογή είναι:

- **BOUND_SIZE_X, BOUND_SIZE_Y, BOUND_SIZE_Z**: Αφορά το μέγεθος του 3D χώρου, που στην ουσία πρόκειται για έναν κύβο. Αλλάζοντας τις μεταβλητές αυτές, παίρνουμε διαφορετική διαμόρφωση του χώρου και διαφορετική συμπεριφορά της προσομοίωσης.
- **TOTAL_BOIDS**: Αφορά τον συνολικό αριθμό των boids που θα συμμετέχουν στην προσομοίωση.
- **TOTAL_PROPERTIES**: Είναι μια μεταβλητή που αφορά τον αριθμό των θέσεων και ταχυτήτων που πρέπει να υπολογιστούν. Καθώς μιλάμε για τρισδιάστατο χώρο, έχουμε για

κάθε boid τις συντεταγμένες **(x, y, z)** της θέσης και τις ταχύτητες για κάθε διάσταση **(vx, vy, vz)**.

Στην συνέχεια έχουμε το αρχείο simulation.js το οποίο είναι και η υλοποίηση της προσομοίωσης που θα εκτελεστεί. Η συγκεκριμένη υλοποίηση βασίζεται στον ψευδοκώδικα του V. Hunter Adams [47].

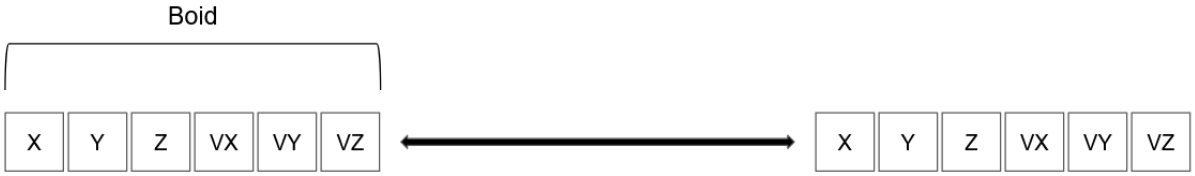
Ο κώδικας περιλαμβάνει την κλάση Simulation, η οποία με την σειρά της αποτελείται από τον constructor και περιλαμβάνει τις μεθόδους **initBoids()** και **updateBoids()**. Στην παρακάτω εικόνα φαίνεται ο constructor της κλάσης Simulation:

```
1  constructor(boidData) {
2
3    this.boidData = boidData;
4
5    this.turnfactor = 0.2;
6    this.visualRange = 200;
7    this.protectedRange = 100;
8    this.centeringfactor = 0.005;
9    this.avoidfactor = 0.05;
10   this.matchingfactor = 0.05;
11   this.xlower = 0;
12   this.ylower = 0;
13   this.zlower = 0;
14   this.xupper = BOUND_SIZE_X;
15   this.yupper = BOUND_SIZE_Y;
16   this.zupper = BOUND_SIZE_Z;
17   this.maxspeed = 6;
18   this.minspeed = 3;
19 }
```

Εικόνα 41: Κώδικας του constructor της κλάσης Simulation.

Ο constructor δέχεται σαν παράμετρο την μεταβλητή boidData, η οποία είναι ένα array τύπου Float32Array. Ορίστηκε να είναι ο ακόλουθος τύπος καθώς έχει την ιδιότητα του transferable και

όπως αναφέραμε στο κεφάλαιο 2 στην ενότητα της επικοινωνίας, μπορεί να επιταχύνει την μεταφορά δεδομένων από και προς έναν web worker. Κάθε boid αναπαρίσταται με 6 διαδοχικές θέσεις μέσα στο array αυτό, 3 θέσεις για τις συντεταγμένες και 3 θέσεις για τις ταχύτητες. Το παρακάτω σχήμα δείχνει εσωτερικά τις θέσεις και ποιές τιμές αντιπροσωπεύουν:



Εικόνα 42: Εσωτερική αναπαράσταση ενός boid στα boidData.

Στις γραμμές 5 με 10 αρχικοποιούνται όλες οι παράμετροι της προσομοίωσης, η μεταβολή των οποίων μπορεί να οδηγήσει σε διαφορετική συμπεριφορά των boids. Στις γραμμές 11 με 16 ορίζονται τα ελάχιστα και μέγιστα όριο του νοητού κύβου της προσομοίωσης. Τέλος στις γραμμές 17 και 18 ορίζεται η ελάχιστη και η μέγιστη ταχύτητα με την οποία μπορεί να κινείται ένα boid.

Στην συνέχεια, έχουμε την μέθοδο initBoids(), όπου γίνεται η αρχικοποίηση των θέσεων για κάθε boid εντός των ορίων της προσομοίωσης, όπως φαίνεται στην εικόνα παρακάτω:

```
1  initBoids() {
2    for(let i=0; i<this.boidData.length; i+=TOTAL_PROPERTIES) {
3      this.boidData[i] = Math.random() * BOUND_SIZE_X;
4      this.boidData[i + 1] = Math.random() * BOUND_SIZE_Y;
5      this.boidData[i + 2] = Math.random() * BOUND_SIZE_Z;
6    }
7  }
```

Εικόνα 43: Κώδικας μεθόδου initBoids.

Τέλος, έχουμε την μέθοδο updateBoids(), όπου είναι υλοποιημένος ο αλγόριθμος που ενημερώνει τις θέσεις και τις κινήσεις όλων των boids (βλέπε Παράρτημα Α). Πηγαίνοντας στο script.js, το παρακάτω τμήμα κώδικα είναι εκείνο που χειρίζεται την εκτέλεση της προσομοίωσης:

```
1  const boidData = new Float32Array(CONSTANTS.TOTAL_BOIDS * CONSTANTS.TOTAL_PROPERTIES);
2  const simulation = new Simulation(boidData);
3
4  simulation.initBoids();
5  createBoidMeshes(scene, boidData);
6
7  function render() {
8    requestAnimationFrame(render);
9    renderer.render(scene, camera);
10   stats.update();
11   controls.update();
12
13   startTime = performance.now();
14   simulation.updateBoids(0, boidData.length);
15   endTime = performance.now();
16   totalTime = endTime - startTime;
17   totalTime = totalTime.toFixed(2);
18
19   totalTimeController.setValue(totalTime);
20
21   updateBoidMeshes(boidMeshes, boidData);
22
23 }
24
25 render();
```

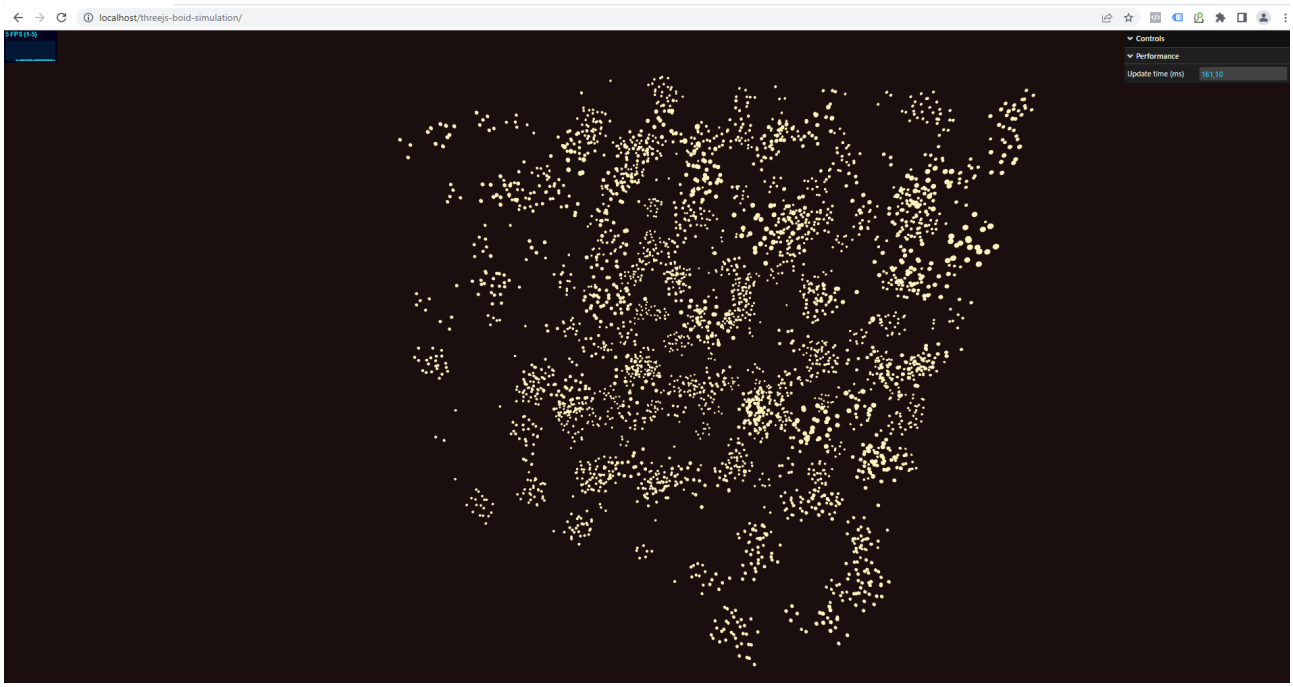
Εικόνα 44: Τμήμα κώδικα αρχείου script.js.

Στις δύο πρώτες γραμμές έχουμε την αρχικοποίηση της μεταβλητής boidData και του αντικειμένου simulation, δίνοντας στο array boidData τον συνολικό αριθμό στοιχείων για όλα τα boids. Στις επόμενες γραμμές 4 και 5, το αντικείμενο simulation δίνει τυχαίες θέσεις σε κάθε boid και έπειτα δημιουργούνται τα Mesh αντικείμενα στο Scene. Μέσα στην συνάρτηση render() έχουμε τις γραμμές 13 με 21 όπου γίνεται η εκτέλεση της προσομοίωσης και η παρακολούθηση των χρόνων εκτέλεσης. Συγκεκριμένα, η εντολή simulation.updateBoids(0, boidData.length) ενημερώνει τις θέσεις των boids και στην γραμμή 21 καλείται η συνάρτηση updateBoidMeshes(boidMeshes, boidData). Οι υλοποιήσεις των συναρτήσεων αυτών, φαίνονται παρακάτω:

```
function createBoidMesh(x, y, z) {
  const boidGeometry = new THREE.SphereGeometry(16, 32, 16);
  const boidMaterial = new THREE.MeshBasicMaterial({
    side: THREE.DoubleSide,
    color: 0xFFF3BF
  });
  const boidMesh = new THREE.Mesh(boidGeometry, boidMaterial);
  boidMesh.position.set(x, y, z);
  return boidMesh;
}

function createBoidMeshes(scene, boidData) {
  for(let i=0; i<boidData.length; i+=CONSTANTS.TOTAL_PROPERTIES){
    let boidMesh = createBoidMesh(boidData[i], boidData[i + 1], boidData[i + 2]);
    boidMeshes.push(boidMesh);
    scene.add(boidMesh);
  }
}

function updateBoidMeshes(boidMeshes, boidData){
  boidMeshes.forEach((boidMesh, index) => {
    boidMesh.position.x = boidData[index * CONSTANTS.TOTAL_PROPERTIES];
    boidMesh.position.y = boidData[index * CONSTANTS.TOTAL_PROPERTIES + 1];
    boidMesh.position.z = boidData[index * CONSTANTS.TOTAL_PROPERTIES + 2];
  });
}
```

Εικόνα 45: Συναρτήσεις createBoidMeshes() και updateBoidMeshes().

Η βοηθητική συνάρτηση createBoidMesh δέχεται σαν παραμέτρους τις συντεταγμένες του boid και δημιουργεί ένα αντικείμενο Mesh για να προστεθεί στο Scene. Κάθε boid αναπαρίσταται από μια σφαίρα με ακτίνα 16 pixels και κίτρινο χρώμα. Έπειτα η createBoidMeshes καλεί την createBoidMesh για κάθε boid. Τέλος η updateBoidMeshes δέχεται ως παράμετρο όλα τα Mesh αντικείμενα των boids και τα δεδομένα που αφορούν τις θέσεις τους και τα ενημερώνει. Η παρακάτω εικόνα δείχνει ένα στιγμιότυπο της εκτέλεσης μιας προσομοίωσης με 5000 boids σε τρισδιάστατο χώρο 5000x5000x5000.



Εικόνα 46: Στιγμιότυπο εκτέλεσης της εφαρμογής Boid Simulation.

3.2.2 Υλοποίηση με χρήση web workers

Όπως είδαμε στην προηγούμενη ενότητα, η ενημέρωση των boids γίνεται μέσα από διπλά εμφωλευμένο βρόχο for. Για μικρό αριθμό boids δεν θα υπάρξει κάποιο ζήτημα αναφορικά με την ταχύτητα ενημέρωσης, όμως όσο αυξάνεται ο αριθμός τους, η ταχύτητα ενημέρωσης και ο ρυθμός ανανέωσης της οθόνης (FPS) θα συνεχίζουν να μειώνονται. Η χρήση των web workers μπορεί να μας βοηθήσει σε βελτίωση των δύο αυτών στοιχείων.

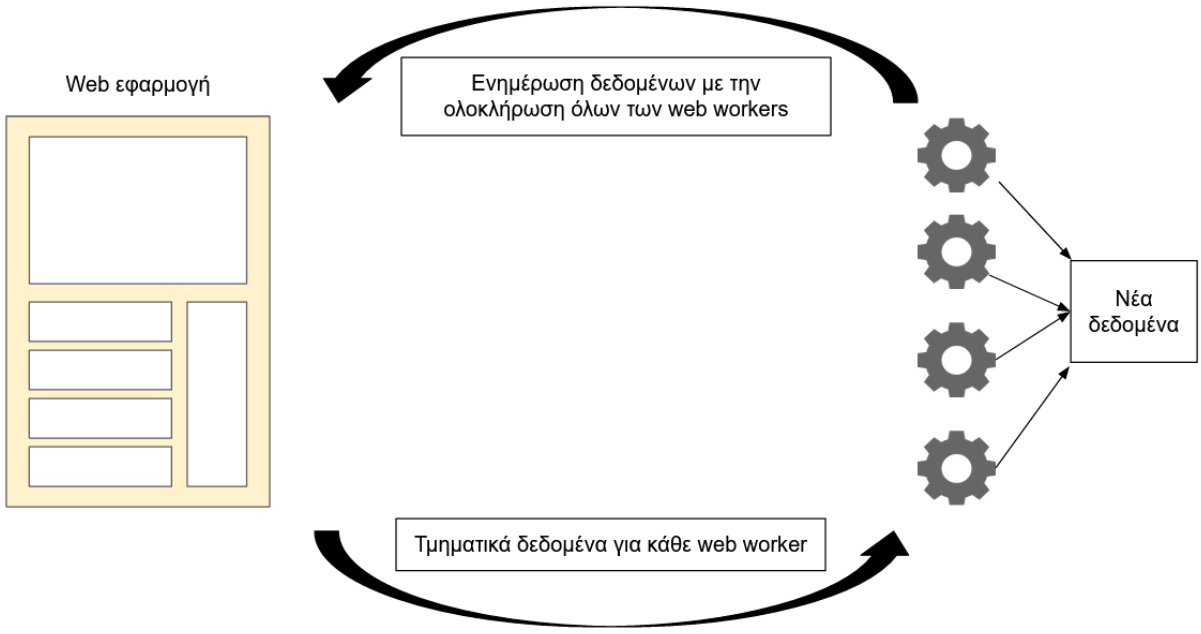
Στην δομή των αρχείων της σειριακής εφαρμογής, θα προσθέσουμε ένα αρχείο worker.js και θα αλλάξουμε την λογική εκτέλεσης της προσομοίωσης στο αρχείο script.js. Ο κώδικας του αρχείου worker.js φαίνεται στην παρακάτω εικόνα:

```
1  onmessage = (message) => {
2    const boidData = message.data[0];
3    const start = message.data[1];
4    const end = message.data[2];
5
6    const simulation = new Simulation(boidData);
7    const newBoidData = simulation.updateBoids(start, end);
8    postMessage({newBoidData, start});
9  };
```

Εικόνα 47: Κώδικας αρχείου worker.js.

Ο κώδικας του αρχείου είναι αρκετά απλός. Αυτό που κάνει είναι να περιμένει να λάβει μήνυμα από το main thread με τα δεδομένα που αφορούν το array με τις θέσεις των boids, καθώς και τα όρια του τμήματος του array για το οποίο θα εκτελέσει την προσομοίωση. Αυτό φαίνεται στις γραμμές 4 με 6, όπου λαμβάνονται το array boidData και τα οι θέσεις start και end του τμήματος που πρόκειται να υπολογιστεί. Στις γραμμές 8 και 9, αρχικοποιείται ένα αντικείμενο της κλάσης Simulation που είδαμε στην προηγούμενη ενότητα και εκτελείται η προσομοίωση, αποθηκεύοντας το τμήμα του ενημερωμένου array σε μια νέα μεταβλητή newBoidData. Τέλος, η newBoidData στέλνεται πίσω στο main thread μαζί με την μεταβλητή start, η οποία χρειάζεται για να γνωρίζουμε την αρχή της θέσης του τελικού array που θα ενημερώσουμε με τις τιμές της newBoidData, καλώντας την μέθοδο postMessage().

Πριν περάσουμε στο αρχείο script.js και παρουσιάσουμε τις αλλαγές στον κώδικα, μπορούμε να δούμε στο παρακάτω σχήμα την λογική την οποία ακολουθούμε για την εκτέλεση της προσομοίωσης χρησιμοποιώντας web workers.



Εικόνα 48: Ροή εκτέλεσης της εφαρμογής Boid Simulation.

Όπως βλέπουμε, η εκτέλεση ξεκινά με τον καταμερισμό των τμημάτων και την αποστολή τους σε κάθε web worker. Στην συνέχεια, ο κάθε web worker εκτελεί την προσομοίωση στο τμήμα που του

αναλογεί και επιστρέφει ένα νέο τμήμα array, το οποίο ενημερώνει το τελικό array με τις θέσεις των boids. Αφού ολοκληρωθούν οι εκτελέσεις όλων των web workers και ενημερωθούν τα δεδομένα, το main thread της εφαρμογής στέλνει ξανά τα τμήματα του ενημερωμένου array για τον εκ νέου υπολογισμό των θέσεων.

Η πρώτη αλλαγή που έγινε αφορά την αρχικοποίηση των web workers και τον υπολογισμό των τμημάτων για κάθε έναν από αυτούς. Οι αλλαγές αυτές φαίνονται στην παρακάτω εικόνα:

```
1  const boidData = new Float32Array(CONSTANTS.TOTAL_BOIDS * CONSTANTS.TOTAL_PROPERTIES);
2  const simulation = new Simulation(boidData);
3
4  simulation.initBoids();
5
6  createBoidMeshes(scene, boidData);
7
8  initWorkers(numWorkers);
9
10 let chunk = boidData.length / numWorkers;
11 let workersFinished = 0;
12 let workerChunk = [];
13
14 startTime = performance.now();
15
16 for(let index=0; index<numWorkers; index++){
17   let start = index * chunk;
18   let end = Math.min(start + chunk, boidData.length);
19   workerChunk.push([start, end]);
20 }
```

Εικόνα 49: Κώδικας αρχικοποίησης των web workers.

Όπως βλέπουμε, αφού αρχικοποιηθούν οι μεταβλητές boidData και simulation, δημιουργούμε τα Mesh αντικείμενα για την γραφική αναπαράσταση των boids, και δημιουργούμε τους web workers με την συνάρτηση initWorkers(). Έπειτα, για κάθε worker, πρέπει να υπολογίσουμε το τμήμα το οποίο θα ενημερώσει κατά την εκτέλεσή του. Αυτό γίνεται στις γραμμές 10 με 19, όπου χρησιμοποιούμε την μεταβλητή workerChunk, μέσα στην οποία αποθηκεύουμε ζεύγη δεικτών που αναπαριστούν τις θέσεις αρχής και τέλους του τμήματος που θα λάβει ο κάθε web worker. Έχοντας υπολογίσει το μέγεθος τους τμήματος που αντιστοιχεί σε κάθε web worker με την μεταβλητή chunk, ορίζουμε με τις μεταβλητές start και end την θέση αυτού του τμήματος μέσα στο array boidData. Λαμβάνουμε υπόψη και την περίπτωση του τελευταίου worker να έχει μικρότερο τμήμα από τους υπόλοιπους, στην γραμμή 18. Στην συνέχεια, προχωράμε στην εκτέλεση της προσομοίωσης όπως φαίνεται στην εικόνα που ακολουθεί:

```
1  for(let index=0; index<numWorkers; index++){
2
3    workerPool[index].postMessage([boidData, workerChunk[index][0], workerChunk[index][1]]);
4    workerPool[index].onmessage = (message) => {
5
6      let data = message.data;
7      boidData.set(data.newBoidData, data.start);
8      workersFinished++;
9
10     if(workersFinished === numWorkers){
11
12       endTime = performance.now();
13       totalTime = endTime - startTime;
14       totalTime = totalTime.toFixed(2);
15       totalTimeController.setValue(totalTime);
16
17       updateBoidMeshes(boidMeshes, boidData);
18
19       workersFinished = 0;
20       startTime = performance.now();
21       for(let j=0; j<numWorkers; j++){
22         workerPool[j].postMessage([boidData, workerChunk[j][0], workerChunk[j][1]]);
23       }
24     }
25   }
26 }
```

Εικόνα 50: Κώδικας εκτέλεσης με web workers.

Στέλνουμε σε κάθε web worker τα δεδομένα προς υπολογισμό, στην γραμμή 3, με την μέθοδο postMessage(). Στην συνέχεια περιμένουμε αν ειδοποιηθεί το main thread με την ολοκλήρωση της εκτέλεσης του κάθε worker και της επιστροφής των ενημερωμένων δεδομένων. Η ενημέρωση των νέων δεδομένων γίνεται στις γραμμές 6 και 7 με την ανάκτηση των δεδομένων που στέλνει ο worker όπως είδαμε παραπάνω (βλέπε Εικόνα 3.26) και με την μέθοδο set(), η οποία παίρνει ως παράμετρο ένα array και έναν δείκτη που είναι το index του array από το οποίο θα γίνει η ενημέρωση. Στην συνέχεια, αυξάνουμε κατά ένα την μεταβλητή workersFinished, η οποία δηλώνει το πλήθος των web workers οι οποίοι έχουν ολοκληρώσει. Όταν η workersFinished γίνει ίση με τον αριθμό των workers που υπάρχουν, όπως φαίνεται στον έλεγχο της γραμμής 10, αυτό σημαίνει ότι όλοι οι workers έχουν ολοκληρώσει και ότι μπορούμε να ενημερώσουμε τα γραφικά της εφαρμογής μας με τις νέες θέσεις. Αυτό γίνεται με την συνάρτηση updateBoidMeshes(). Τέλος, μηδενίζουμε την μεταβλητή workersFinished και ξεκινάμε νέα εκτέλεση της προσομοίωσης, στέλνοντας τα νέα δεδομένα σε κάθε worker για τους νέους υπολογισμούς.

3.2.2.1 Προσθήκη offscreencanvas

Στην προηγούμενη ενότητα χρησιμοποιήσαμε web workers και παραλληλοποιήσουμε τις εκτελέσεις της προσομοίωσης, μοιράζοντας τον όγκο δεδομένων και τον υπολογισμό σε κάθε ένα από αυτούς. Το κύριο thread της εφαρμογής, εκτός από την επικοινωνία με τους workers, είναι υπεύθυνο και για την ενημέρωση και απεικόνιση των γραφικών, καθώς η three.js ενημερώνει την σκηνή απεικόνισης μέσα από αυτό. Στην παρακάτω εικόνα φαίνεται το αποτέλεσμα από την καταγραφή εκτέλεσης της εφαρμογής στον Chrome browser.

Εικόνα 51: Γράφημα ροής λειτουργιών εκτέλεσης και απεικόνισης στον Chrome.

Στην εικόνα υπάρχουν δύο περιγράμματα με κόκκινο χρώμα. Το ένα αφορά το main thread και δείχνει την στοίβα των κλήσεων που πραγματοποιεί για την ενημέρωση και την απεικόνιση των γραφικών. Το δεύτερο περίγραμμα αφορά τους workers και τους υπολογισμούς που εκτελούν, όπως φαίνονται από τα σχήματα σε κάθε γραμμή. Όπως βλέπουμε, το main thread, αν και δεν είναι αυτό το οποίο εκτελεί τις υπολογιστικά εντατικές εργασίες, είναι συνεχώς απασχολημένο με το να τρέχει τις συναρτήσεις που ελέγχουν και ενημερώνουν τα γραφικά και τα στατιστικά της εφαρμογής. Αυτό έχει το μειονέκτημα ότι οι συνεχείς έλεγχοι που γίνονται, δεσμεύουν την CPU στο μεγαλύτερο μέρος της, αφήνοντας μικρά περιθώρια για τις υπόλοιπες λειτουργίες, όπως η αλληλεπίδραση με τον χρήστη, μειώνοντας έτσι την απόκριση της εφαρμογής σε συμβάντα (events) του χρήστη.

Για να λύσουμε το παραπάνω ζήτημα και να αποδεσμεύσουμε το main thread από τους εντατικούς ελέγχους και ενημερώσεις, θα χρησιμοποιήσουμε το Offscreencanvas και θα

μεταφέρουμε το rendering σε έναν web worker. Όπως είπαμε και στο κεφάλαιο 2, το Offscreencanvas μας δίνει την δυνατότητα να μεταφέρουμε και να ενημερώνουμε ένα canvas στοιχείο μέσα σε web worker και αυτομάτως να συγχρονίζεται στο main thread, διατηρώντας συνέπεια ανάμεσα στα δύο στοιχεία.

Η σελίδα της εφαρμογής μας έχει ένα στοιχείο canvas με id=**"three-canvas",** στο οποίο εκτελείται ο renderer της three.js.

<canvas id="three-canvas"></canvas>

Η λογική η οποία θα ακολουθήσουμε είναι η εξής:

- Θα μεταφέρουμε το canvas σε web worker, μέσα στο οποίο θα αρχικοποιήσουμε την three.js και τα γραφικά της εφαρμογής.
- Οι εκτελέσεις της προσομοίωσης θα γίνονται με την χρήση web workers, μεταφέροντας δεδομένα από το main thread στους workers.
- Μόλις ολοκληρωθούν οι υπολογισμοί της επανάληψης της προσομοίωσης και συγκεντρωθούν τα αποτελέσματα, θα μεταφερθούν από το main thread στον worker που χειρίζεται το canvas, θα ενημερωθούν τα γραφικά και οι αλλαγές θα φανούν στην σελίδα.

Για να υλοποιήσουμε την παραπάνω λογική, θα τροποποιήσουμε το αρχείο index.html, προσθέτοντας ένα script μέσα στην σελίδα. Μέσα σε αυτό το script θα δημιουργήσουμε μια αναφορά στο canvas στοιχείο που μας ενδιαφέρει και έναν web worker που θα χειρίζεται τις ενημερώσεις του στοιχείου αυτού, όπως φαίνεται στον κώδικα της εικόνας παρακάτω:

```
1  const canvas = document.getElementById('three-canvas');
2  canvas.width = window.innerWidth;
3  canvas.height = window.innerHeight;
4
5  if(canvas.transferControlToOffscreen) {
6
7    const offscreen = canvas.transferControlToOffscreen();
8    const offscreenWorker = new Worker('offscreen.js', { type: 'module' });
9    /*
10    *
11    */
12 }
```

Εικόνα 52: Κώδικας αρχικοποίησης offscreencanvas.

Αφού πάρουμε την αναφορά του canvas στις γραμμές 1 με 3, στην συνέχεια εξετάζουμε αν υποστηρίζεται η μέθοδος transferControlToOffscreen() από τον browser, έτσι ώστε να δημιουργήσουμε ένα αντίγραφο του canvas το οποίο θα τρέχει μέσα σε web worker. Αυτό γίνεται στις γραμμές 7 με 8, όπου δημιουργείται το αντίγραφο offscreen και ο offscreenWorker ο οποίος θα εκτελεί το αρχείο offscreen.js. Μέσα στο αρχείο offscreen.js βρίσκεται ο κώδικας της three.js όπως είδαμε και στην προηγούμενη εφαρμογή. Η μόνη διαφορά είναι το onmessage event το οποίο λαμβάνει μηνύματα από το main thread που αφορούν την ενημέρωση γραφικών.

```
1  onmessage = (message) => {
2
3    const data = message.data;
4
5    if(data.command === 'init') {
6      initData(data);
7    }
8
9    if(data.command === 'update') {
10     updateBoidMeshes(boidMeshes, data.boidData);
11   }
12 }
```

Εικόνα 53: onmessage event στο αρχείο offscreen.js.

Όπως βλέπουμε, μέσα στην μεταβλητή data που λαμβάνει, υπάρχει το property **command**, το οποίο μπορεί να είναι είτε init για να ξεκινήσει την 3D σκηνή, είτε update για να ενημερώσει τα γραφικά. Η συνάρτηση initData() φαίνεται στην παρακάτω εικόνα:

```
function initData(data) {

  const CAMERA_PARAMS = {
    fov: 60,
    aspect: data.width / data.height,
    near: 0.1,
    far: 10000,
  };

  const renderer = initWebGLRenderer(data);

  const camera = initCamera(CAMERA_PARAMS);

  const axesHelper = new THREE.AxesHelper(200);

  scene = new THREE.Scene();

  scene.background = new THREE.Color(CONSTANTS.SCENE_BACKGROUND);

  scene.add(axesHelper);

  createBoidMeshes(scene, data.boidData);

  function render() {
    renderer.render(scene, camera);
    requestAnimationFrame(render);
  }

  render();
}
```

Εικόνα 54: Κώδικας συνάρτησης initData().

Η συνάρτηση updateBoidMeshes() φαίνεται στην παρακάτω εικόνα:

```
function updateBoidMeshes(boidMeshes, boidData){
  boidMeshes.forEach((boidMesh, index) => {
    boidMesh.position.x = boidData[index * CONSTANTS.TOTAL_PROPERTIES];
    boidMesh.position.y = boidData[index * CONSTANTS.TOTAL_PROPERTIES + 1];
    boidMesh.position.z = boidData[index * CONSTANTS.TOTAL_PROPERTIES + 2];
  });
}
```

Εικόνα 55: Κώδικας συνάρτησης updateBoidMeshes().

Συνεχίζοντας τώρα από το σημείο που αρχικοποιήσαμε το νέο Offscreencanvas και τον web worker που θα το ενημερώνει, στέλνουμε στον offscreen worker το μήνυμα αρχικοποίησης των γραφικών, όπως φαίνεται παρακάτω:

```
const offscreen = canvas.transferControlToOffscreen();
const offscreenWorker = new Worker('offscreen.js', { type: 'module' });

offscreenWorker.postMessage({
  command: 'init',
  drawingSurface: offscreen,
  width: canvas.width,
  height: canvas.height,
  pixelRatio: window.devicePixelRatio,
  boidData: boidData
}, [offscreen]);
```

Εικόνα 56: Κώδικας αποστολής μηνύματος στον offscreen worker.

Τα δεδομένα που στέλνουμε είναι τα εξής:

- **command**: Είναι η εντολή βάσει της οποίας θα αποφασίσει ο offscreenWorker ποιό τμήμα κώδικας θα εκτελέσει (αρχικοποίηση / ενημέρωση γραφικών).
- **drawingSurface**: Είναι το Offscreencanvas αντικείμενο που στέλνουμε.
- **width**: Το πλάτος του canvas.
- **height**: Το ύψος του canvas.
- **pixelRatio**: Μια μεταβλητή η οποία καθορίζει τις αναλογίες με τις οποίες θα απεικονίζονται τα γραφικά στο canvas.
- **boidData**: Τα δεδομένα που αφορούν τις θέσεις των boids.

Παρατηρούμε ότι εκτός του object με τα παραπάνω δεδομένα, στέλνεται και δεύτερο όρισμα, ένα array με το offscreen σαν μοναδικό στοιχείο. Αυτό γίνεται επειδή το offscreen είναι transferable και μπορεί να μεταφερθεί χωρίς αντιγραφή, για γρηγορότερες ενημερώσεις.

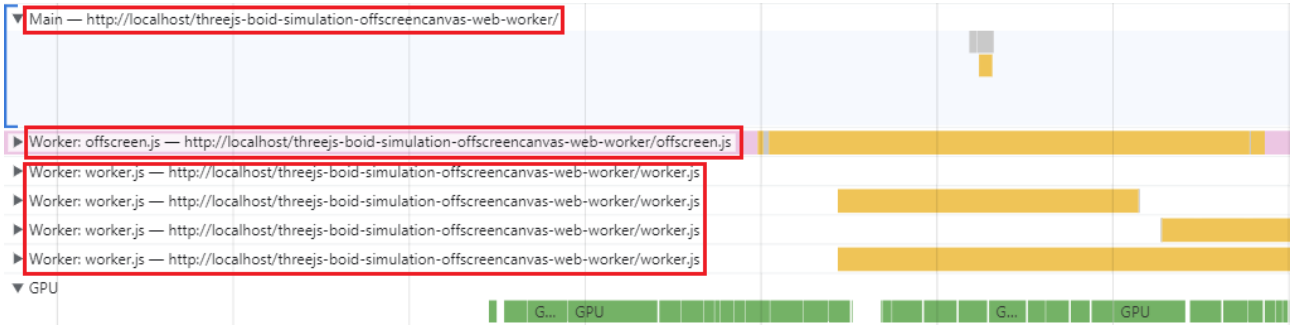
Στην συνέχεια θα εκτελεστεί η προσομοίωση χρησιμοποιώντας ένα pool με web workers, όπως και σε προηγούμενες υλοποιήσεις, χωρίζοντας τα δεδομένα σε τμήματα και κατανέμοντάς τα στους workers και ενημερώνοντας τα γραφικά, όταν ο κύκλος της εκτέλεσης έχει ολοκληρωθεί από το πλήθος των workers. Αυτό μπορούμε να το δούμε στον κώδικα της παρακάτω εικόνας:

```
    let chunk = boidData.length / TOTAL_WORKERS;
    for(let index=0; index<TOTAL_WORKERS; index++) {
      let start = index * chunk;
      let end = Math.min(start + chunk, boidData.length);
      let workerBoidData = boidData.slice();

      workerPool[index].postMessage({
        boidData: workerBoidData.buffer,
        start: start,
        end: end,
      }, [workerBoidData.buffer]);

      workerPool[index].onmessage = (message) => {
        boidData.set(message.data.boidData, message.data.start);
        workersFinished++;
        if(workersFinished === TOTAL_WORKERS){
          offscreenWorker.postMessage({
            command: 'update',
            boidData: boidData
          });
          workersFinished = 0;
          count++;
          for(let j=0; j<TOTAL_WORKERS; j++){
            start = j * chunk;
            end = Math.min(start + chunk, boidData.length);
            workerBoidData = boidData.slice();
            workerPool[j].postMessage({
              boidData: workerBoidData.buffer,
              start: start,
              end: end,
            }, [workerBoidData.buffer]);
          }
        }
      }
    }
}
```

Εικόνα 57: Κώδικας εκτέλεσης της προσομοίωσης με web workers

Αφού υπολογίσουμε το μέγεθος τμήματος των δεδομένων (chunk) που αντιστοιχεί σε κάθε worker, στις γραμμές 3 με 5 ορίζουμε τα δεδομένα για κάθε worker. Στην συνέχεια, στις γραμμές 7 με 11, τα στέλνουμε με την μέθοδο postMessage(), χρησιμοποιώντας το property buffer του TypedArray boidData σαν transferable. Στην γραμμή 13, το main thread ειδοποιείται για την ολοκλήρωση κάθε web worker, ενημερώνοντας το array boidData με τα νέα δεδομένα, και αυξάνοντας την μεταβλητή workersFinished κατά ένα. Μόλις τελειώσουν όλοι οι web workers, στέλνεται μήνυμα στον offscreen worker να ενημερώσει τα γραφικά των boids με τις νέες θέσεις, στην γραμμή 17, και έπειτα ξεκινάει ένας νέος κύκλος υπολογισμού από τις γραμμές 21 και 31.

Είχαμε δει στην αρχή της ενότητας ένα στιγμιότυπο από την εκτέλεση της παραπάνω εφαρμογής μέσω των εργαλείων αποσφαλμάτωσης του Chrome, αναφορικά με τον χρόνο εκτέλεσης των διάφορων συναρτήσεων και του όγκου των υπολογισμών. Είδαμε ότι, κατά την εκτέλεση της προσομοίωσης, το main thread είναι μονίμως απασχολημένο με τους ελέγχους για την ενημέρωση των γραφικών που πραγματοποιεί η three.js. Με την χρήση Offscreencanvas, η εκτέλεση της προσομοίωσης αποτυπώνεται στο παρακάτω στιγμιότυπο:



Εικόνα 58: Στιγμιότυπο χρήσης υπολογιστικών πόρων με offscreencanvas.

Όπως μπορούμε να διαπιστώσουμε, το main thread, το οποίο είναι το πρώτο με κόκκινο περίγραμμα, δεν απασχολείται καθόλου, καθώς ο Worker offscreen.js, που βρίσκεται στο δεύτερο κόκκινο περίγραμμα, έχει αναλάβει όλη την ενημέρωση και απεικόνιση των γραφικών. Οι web workers συνεχίζουν να απασχολούνται σταθερά με τον υπολογισμό των νέων θέσεων των boids. Στην επόμενη ενότητα, θα παρουσιάσουμε τα αποτελέσματα των μετρήσεων αναφορικά με τους χρόνους εκτέλεσης και τις τιμές FPS που πετυχαίνουμε με την σειριακή και παράλληλη υλοποίηση.

3.2.3 Σύγκριση αποτελεσμάτων

Για την λήψη δεδομένων αναφορικά με τους χρόνους εκτέλεσης και τους ρυθμούς ανανέωσης στην σειριακή και παράλληλη υλοποίηση της εφαρμογής, εκτελέστηκαν **1000** επαναλήψεις της προσομοίωσης σε κάθε εφαρμογή για τις περιπτώσεις των **1000**, **2000**, **3000**, **4000** και **5000** boids.

Ξεκινώντας από τους χρόνους εκτέλεσης, τα αποτελέσματα φαίνονται στους παρακάτω πίνακες για κάθε browser:

	1000	**2000**	**3000**	**4000**	**5000**
Single Thread	7,66	27,51	62,24	112,65	175,1
2 web workers	16,3	36,59	73,63	111,45	163,09
4 web workers	9,65	34,91	54,22	84,43	127,72
8 web workers	10,27	34,77	49,08	72,1	100,44
16 web workers	9,29	34,44	50,61	64,1	79,39

Πίνακας 6: Μέσοι χρόνοι εκτέλεσης στον Chrome.

	1000	**2000**	**3000**	**4000**	**5000**
Single thread	8,42	29,53	63,98	113,91	173,95
2 web workers	10,98	28,78	46,62	86,52	124,52
4 web workers	12,44	28,75	43,54	69,53	102,82
8 web workers	12,91	27,16	41,06	61,5	85,45
16 web workers	10,02	25,77	41,06	56,79	72,2

Πίνακας 7: Μέσοι χρόνοι εκτέλεσης του Edge.

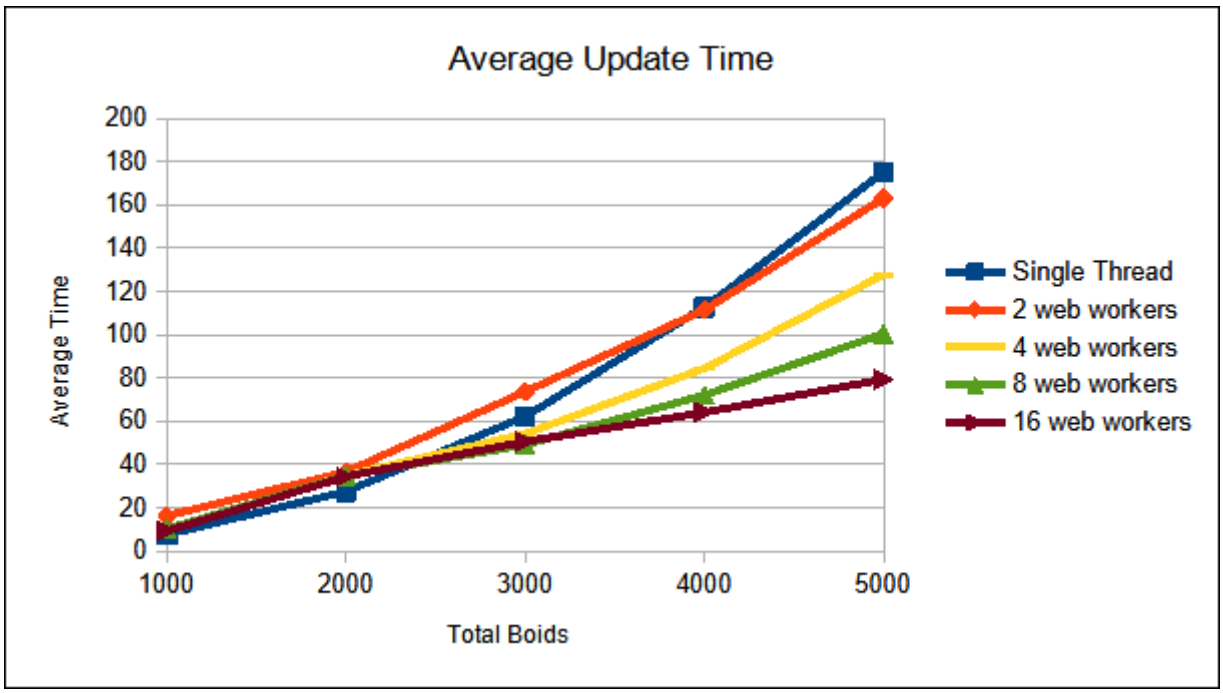
	1000	**2000**	**3000**	**4000**	**5000**
Single thread	8,02	19,25	185,42	325,84	506,37
2 web workers	31,58	81,1	91,99	231,84	345,66
4 web workers	33,24	82,2	105,34	145	213,81
8 web workers	38,27	77,48	105,11	128,57	186,46
16 web workers	34,8	73,82	98,82	135,6	151,55

Πίνακας 8: Μέσοι χρόνοι εκτέλεσης στον Firefox.

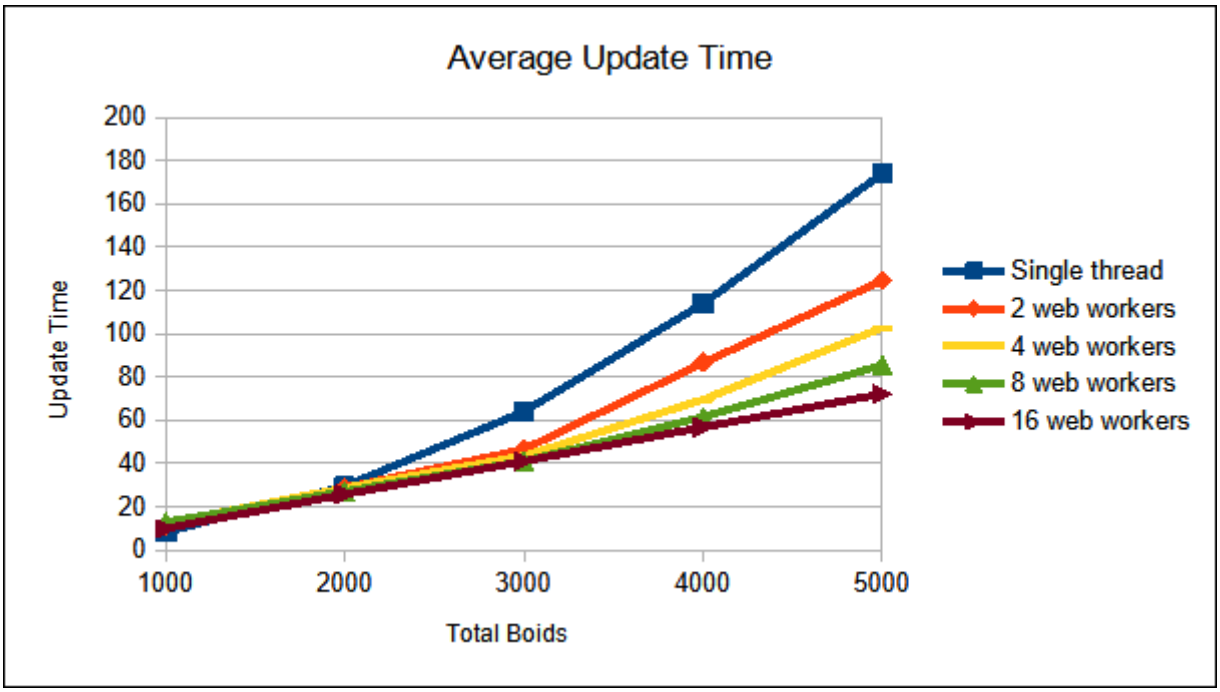
	1000	2000	3000	4000	5000
Single thread	9,45	33,25	72,39	121,81	187,5
2 web workers	14,9	33,66	57,65	94,63	130,62
4 web workers	16,82	33,25	47,57	75,24	109,93
8 web workers	17,21	35,32	74,93	67,73	92,82
16 web workers	17,99	33	44,82	62,68	77,4

Πίνακας 9: Μέσοι χρόνοι εκτέλεσης στον Opera.

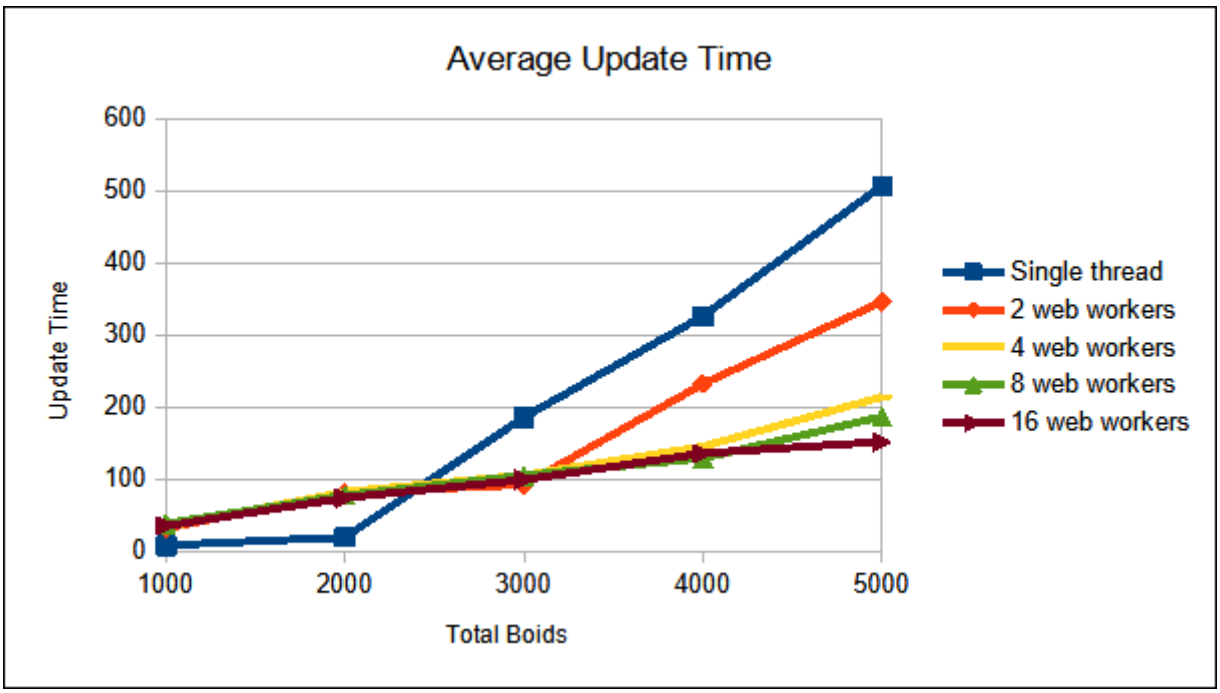
Ξεκινώντας από την περίπτωση των 1000 boids, μπορούμε να δούμε ότι δεν έχουμε κάποιο όφελος από την χρήση των web workers καθώς όλοι οι χρόνοι εμφανίζονται να είναι χειρότεροι από αυτούς της σειριακής υλοποίησης. Το ίδιο συμβαίνει και στην περίπτωση των 2000 boids, όπου οι χρόνοι εκτέλεσης είναι ίδιοι, εκτός από την περίπτωση του Firefox, όπου οι χρόνοι είναι κατά πολύ μεγαλύτεροι χρησιμοποιώντας web workers. Από το πλήθος των 3000 boids, αρχίζουμε να βλέπουμε κάποιες βελτιώσεις στους χρόνους, με κάθε browser να αξιοποιεί διαφορετικά το πλήθος των web workers, καθώς σε κάθε browser ο καλύτερος χρόνος βρίσκεται σε διαφορετικό πλήθος workers, με τον Chrome να έχει την μέγιστη απόδοση στους 8, τον Edge στους 8 και 16, τον Firefox στους 2 και τον Opera στους 16. Περνώντας στα 4000 boids βλέπουμε ότι έχουμε αισθητές διαφορές στους χρόνους, φθάνοντας σχεδόν στον μισό χρόνο εκτέλεσης, με όλους τους browsers, εκτός του Firefox, να αξιοποιούν καλύτερα τους 16 workers για να πετύχουν το βέλτιστο αποτέλεσμα, με τον Firefox να πετυχαίνει καλύτερα αποτελέσματα στους 8 workers. Τέλος, για τα 5000 boids, η απόδοση των εφαρμογών καταφέρνει να υπερδιπλασιαστεί, με τους χρόνους να πέφτουν κάτω από το μισό του αντίστοιχων χρόνων της σειριακής εφαρμογής, με τον Firefox να κάνει την καλύτερη διαχείριση από τους 4 workers και πάνω, ο Edge με τον Opera από τους 8 και ο Chrome από τους 16. Οι αυξήσεις των αποδόσεων της εφαρμογής σε κάθε browser, αποτυπώνονται πιο ξεκάθαρα στα διαγράμματα που ακολουθούν:



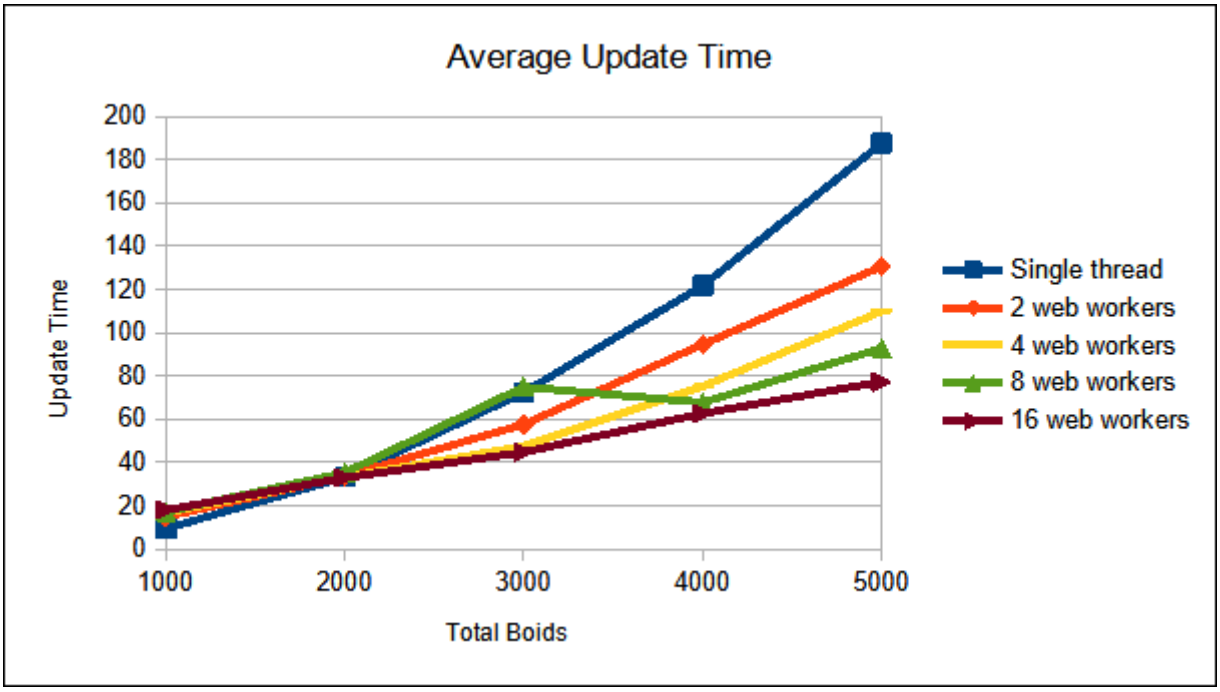
Εικόνα 59: Διάγραμμα μέσου χρόνου εκτέλεσης στον Chrome.



Εικόνα 60: Διάγραμμα μέσου χρόνου εκτέλεσης στον Edge.



Εικόνα 61: Διάγραμμα μέσου χρόνου εκτέλεσης στον Firefox.



Εικόνα 62: Διάγραμμα μέσου χρόνου εκτέλεσης στον Opera.

Περνώντας στην μέτρηση των FPS, στους παρακάτω πίνακες παρουσιάζονται τα αποτελέσματα των μετρήσεων:

	1000	**2000**	**3000**	**4000**	**5000**
Single Thread	58,7	24,23	12,29	7,32	4,9
2 web workers	60	57,38	42,04	34,88	29,48
4 web workers	60	55,62	39,07	28,09	24,81
8 web workers	60	54,76	35,56	24,63	18,43
16 web workers	60	52,53	36,22	22,21	16,1

Πίνακας 10: Τιμές FPS στον Chrome.

	1000	**2000**	**3000**	**4000**	**5000**
Single thread	56,41	23	11,59	7,1	4,84
2 web workers	59,92	53,29	43,16	32,59	28,9
4 web workers	59,83	53,58	37,33	29,76	23,22
8 web workers	57,36	51,04	36,05	25,17	18,61
16 web workers	59,88	48,19	34,95	21,48	15,04

Πίνακας 11: Τιμές FPS στον Edge.

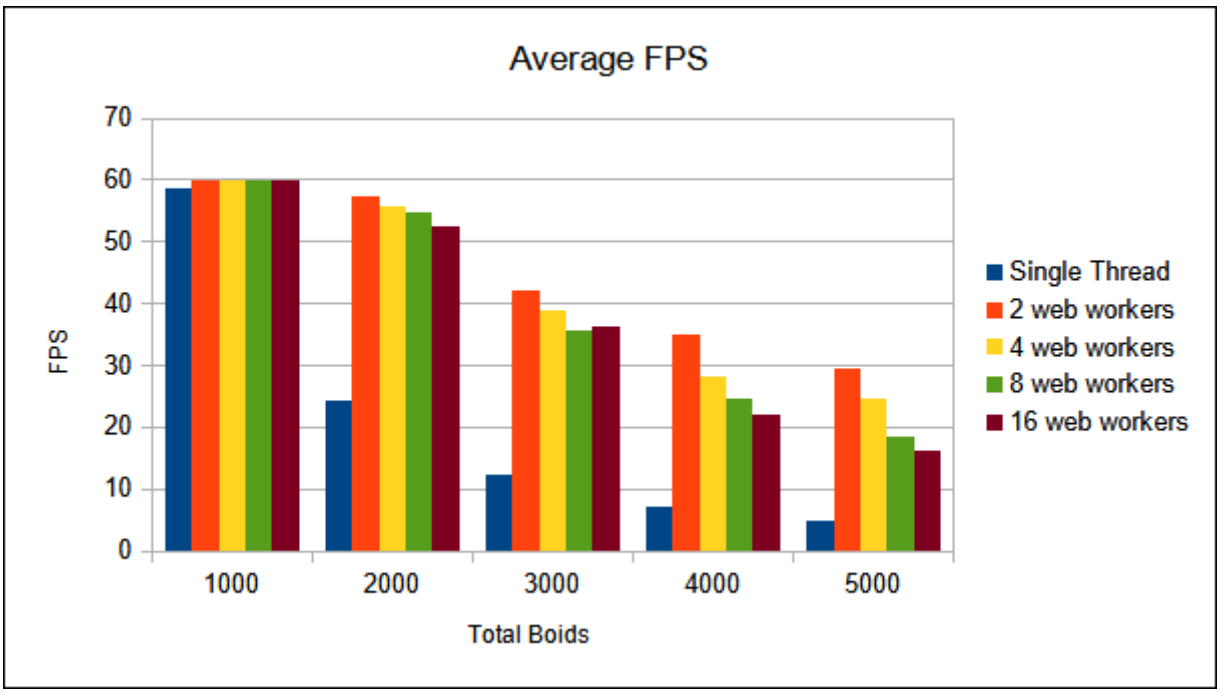
	1000	**2000**	**3000**	**4000**	**5000**
Single thread	40,69	27,05	4,47	2,68	1,78
2 web workers	52,07	33,56	24,7	18,82	14,9
4 web workers	54,34	34,92	23,67	16,68	13,58
8 web workers	53,41	33,19	23	16,7	12,92
16 web workers	51,88	26,35	20,16	14,99	12,76

Πίνακας 12: Τιμές FPS στον Firefox.

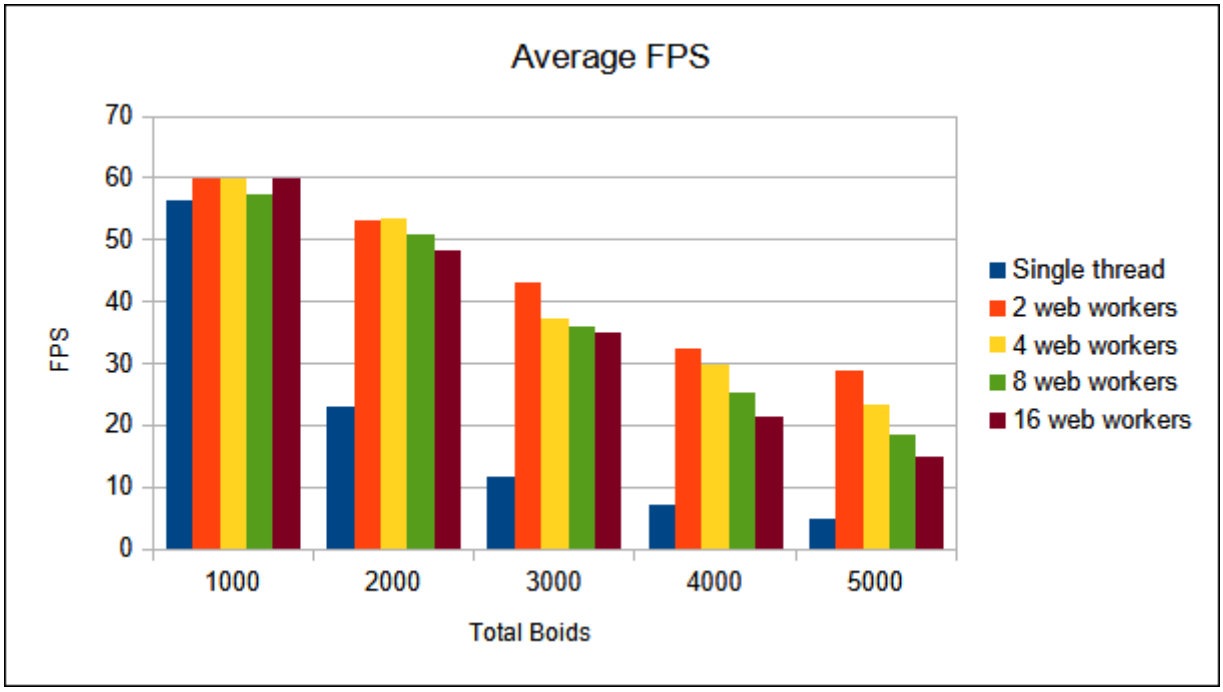
	1000	2000	3000	4000	5000
Single thread	48,7	20,03	10,77	6,59	4,46
2 web workers	60	54,71	39,04	34,29	28,72
4 web workers	60	50,11	35,41	27,42	23,01
8 web workers	59,33	51,26	33,63	22,03	17,16
16 web workers	59,67	47,2	33,93	20,31	14,53

Πίνακας 13: Τιμές FPS στον Opera.

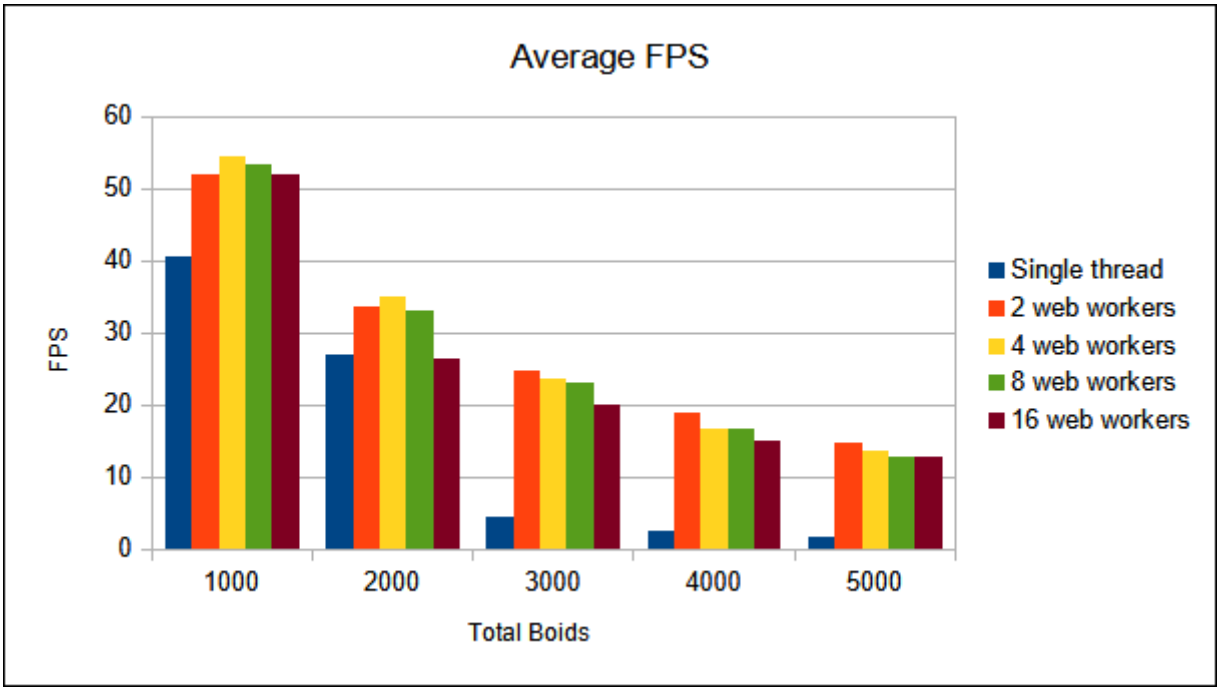
Μπορούμε αμέσως να δούμε τις διαφορές που υπάρχουν στις τιμές FPS σε όλους τους browsers. Ξεκινώντας από το πλήθος των 1000 boids, βλέπουμε ότι για όλες τις τιμές των web workers, ο αριθμός των FPS είναι πάνω από 50, με τους Chrome και Opera να πετυχαίνουν ακόμη και 60 FPS. Συνεχίζοντας στα 2000 boids, βλέπουμε τα FPS να παραμένουν σε υψηλά επίπεδα για τους workers, σε αντίθεση με την single threaded υλοποίηση όπου ο αριθμός των FPS υποδιπλασιάζεται. Μόνο ο Firefox φαίνεται να έχει χαμηλότερες τιμές, κοντά στις τιμές της single threaded υλοποίησης. Πηγαίνοντας στα 3000 boids, βλέπουμε ότι πάλι η πλειοψηφία των browsers, εκτός του Firefox, εξακολουθεί να έχει τιμές FPS μεγαλύτερες των 30, οι οποίες είναι ικανοποιητικές για την εμπειρία του χρήστη. Στα 4000 boid πλέον οι τιμές έχουν πέσει κάτω από 30 FPS, εκτός από τις περιπτώσεις που χρησιμοποιούνται 2 web workers οι οποίοι παρουσιάζουν σε όλες τις περιπτώσεις τον υψηλότερο αριθμό FPS. Επίσης παρατηρείται ότι όσο αυξάνεται ο αριθμός των web workers, τόσο χαμηλότερες τιμές παίρνουμε, βάσει της μεθόδου και της βιβλιοθήκης stats.js με την οποία έγιναν οι μετρήσεις. Αυτό μπορεί να οφείλεται στο γεγονός ότι μεγαλύτερο πλήθος workers σημαίνει περισσότερες ανταλλαγές μηνυμάτων, το οποίο σημαίνει περισσότερες κλήσεις στην μέθοδο postMessage() και περισσότερα onmessage events, τα οποία μπλοκάρουν για αμελητέο χρονικό διάστημα, καθιστώντας την εφαρμογή μη αποκρίσιμη για αυτό το διάστημα. Η αύξηση στον αριθμό των FPS φαίνεται πιο ξεκάθαρα στα διαγράμματα που ακολουθούν:



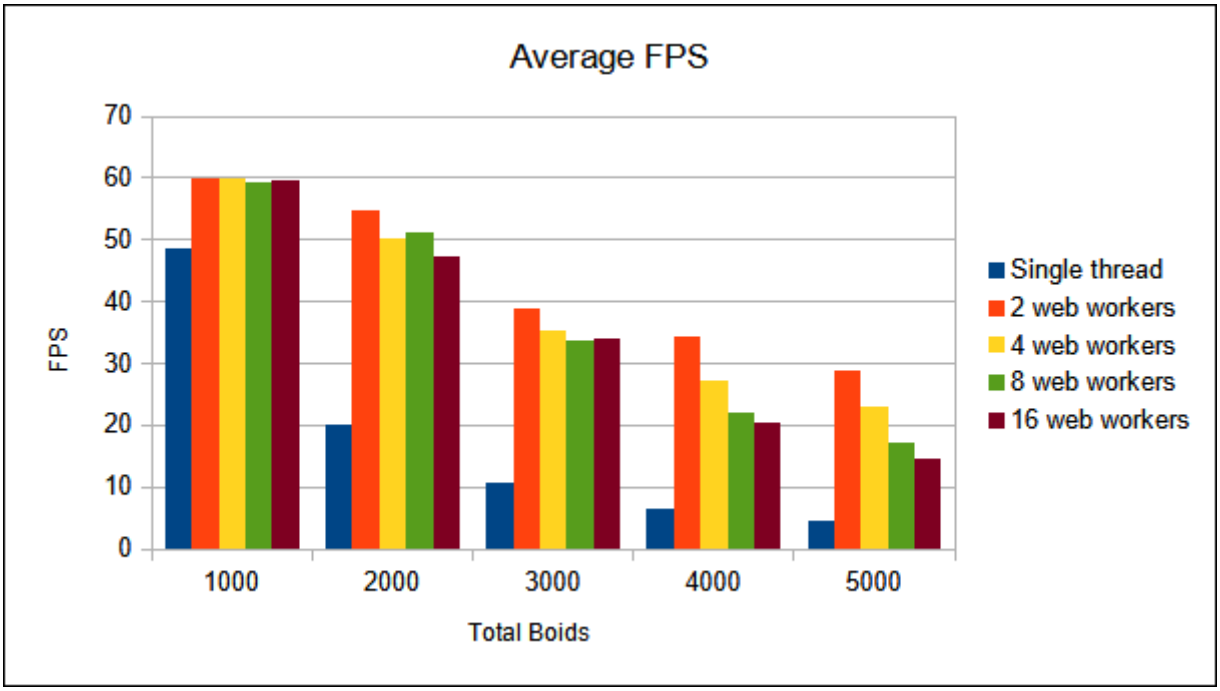
Εικόνα 63: Μέση τιμή FPS για τον Chrome.



Εικόνα 64: Μέση τιμή FPS για τον Edge.



Εικόνα 65: Μέση τιμή FPS για τον Firefox.



Εικόνα 66: Μέση τιμή FPS για τον Opera.

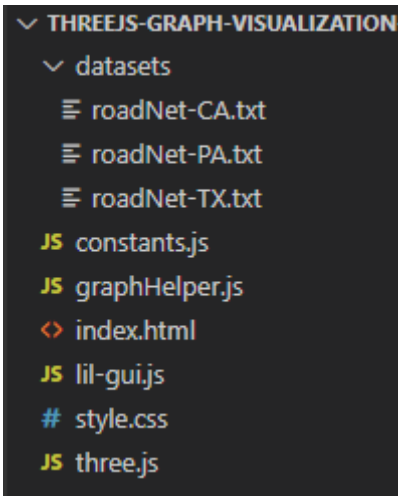
3.3 Εφαρμογή οπτικοποίησης 3D γράφου

Η τρίτη εφαρμογή που θα εξετάσουμε αφορά την οπτικοποίηση γράφων. Συγκεκριμένα, το ζήτημα της συγκεκριμένης περίπτωσης εφαρμογής είναι η καθυστέρηση από την φόρτωση και επεξεργασία των δεδομένων που πρόκειται να απεικονιστούν στον γράφο, η οποία περιορίζει αρκετά την απόκριση της εφαρμογής στις αλληλεπιδράσεις του χρήστη, οδηγώντας σε κακό UX.

Για την συγκεκριμένη εφαρμογή χρησιμοποιήθηκαν η 3d-force-graph [48], η οποία είναι βιβλιοθήκη που αναπτύχθηκε χρησιμοποιώντας την three.js, και τα datasets του πανεπιστημίου του Stanford που αφορούν τους κόμβους των οδικών συστημάτων της Καλιφόρνια, της Πενσυλβάνια και του Τέξας [49].

3.3.1 Σειριακή υλοποίηση

Στην παρακάτω εικόνα μπορούμε να δούμε την δομή της εφαρμογής:



Εικόνα 67: Δομή εφαρμογής graph visualization.

Εκτός από τις βιβλιοθήκες lil-gui.js και three.js που χρησιμοποιούνται σε κάθε project, έχουμε τα αρχεία constants.js και graphHelper.js, στα οποία ορίζονται οι παράμετροι της εφαρμογής και οι βοηθητικές συναρτήσεις αντίστοιχα. Συγκεκριμένα, στο constants.js, βρίσκονται τα ονόματα των αρχείων των οποίων τα δεδομένα θα φορτώσουμε και επεξεργαστούμε, καθώς και ο αριθμός των κόμβων που θα απεικονιστούν στον γράφο. Τα περιεχόμενα του αρχείου φαίνονται στην παρακάτω εικόνα:

```
1  export const BASE_PATH = './datasets/';
2
3  export const FILENAMES = [
4    'roadNet-CA.txt',
5    'roadNet-PA.txt',
6    'roadNet-TX.txt'
7  ];
8
9  export const TOTAL_NODES = [
10   1000,
11   2000,
12   3000,
13   4000,
14   5000
15 ];
```

Εικόνα 68: Κώδικας αρχείου constants.js.

Έχουμε τις σταθερές:

- **BASE_PATH**: Είναι το path της εφαρμογής στο οποίο βρίσκονται τα datasets.
- **FILENAMES**: Είναι τα ονόματα των αρχείων τα οποία μπορούμε να φορτώσουμε και να επεξεργαστούμε.
- **TOTAL_NODES**: Είναι ο συνολικός αριθμός των κόμβων που θα απεικονιστούν στον γράφο.

Στην συνέχεια έχουμε το αρχείο graphHelper.js, με την συνάρτηση getGraphData(), όπως φαίνεται στην παρακάτω εικόνα:

```
1  export async function getGraphData(filePath, totalNodes) {
2    let links = [];
3    let nodes = [...Array(totalNodes).keys()].map(i => ({ id: i }));
4    let response = await fetch(filePath);
5    let responseText = await response.text();
6    let pairs = responseText.split(/\r?\n/);
7    pairs = pairs.slice(4, pairs.length).map(pair => pair.split('\t'));
8    pairs.forEach(pair => {
9      if(parseInt(pair[0]) < totalNodes && parseInt(pair[1]) < totalNodes){
10       links.push({source: parseInt(pair[0]), target: parseInt(pair[1])});
11     }
12   });
13   return {nodes, links};
14 }
```

Εικόνα 69: Κώδικας αρχείου graphHelper.js.

Βλέπουμε πως πρόκειται για μια async συνάρτηση, καθώς περιλαμβάνει τμήματα κώδικα τα οποία εκτελούνται ασύγχρονα και για τα οποία θα χρειαστεί να περιμένουμε να ολοκληρωθούν για να λάβουμε τα αποτελέσματα, χρησιμοποιώντας await. Δέχεται σαν παραμέτρους τις μεταβλητές filePath και totalNodes, δηλώνοντας την διαδρομή του αρχείου που θα φορτωθεί και τον μέγιστο αριθμό κόμβων που θα επεξεργαστούν. Στην γραμμή 3, αρχικοποιούμε την δομή την οποία θα στείλουμε σαν όρισμα στην βιβλιοθήκη 3d-force-graph για την αναπαράσταση του γράφου. Στις γραμμές 4 και 5, κάνουμε fetch το αρχείο με τους κόμβους του οδικού δίκτυου και το επιστρέφουμε σαν απλό κείμενο, προκειμένου να το επεξεργαστούμε. Η μορφή του εκάστοτε αρχείου που διαβάζουμε είναι όπως παρακάτω:

 # Directed graph (each unordered pair of nodes is saved once): roadNet-PA.txt

 # Pennsylvania road network

 # Nodes: 1088092 Edges: 3083796

 # FromNodeIdToNodeId

0 1

0 6309

0 6353

1 0

6353 0

6353 6354

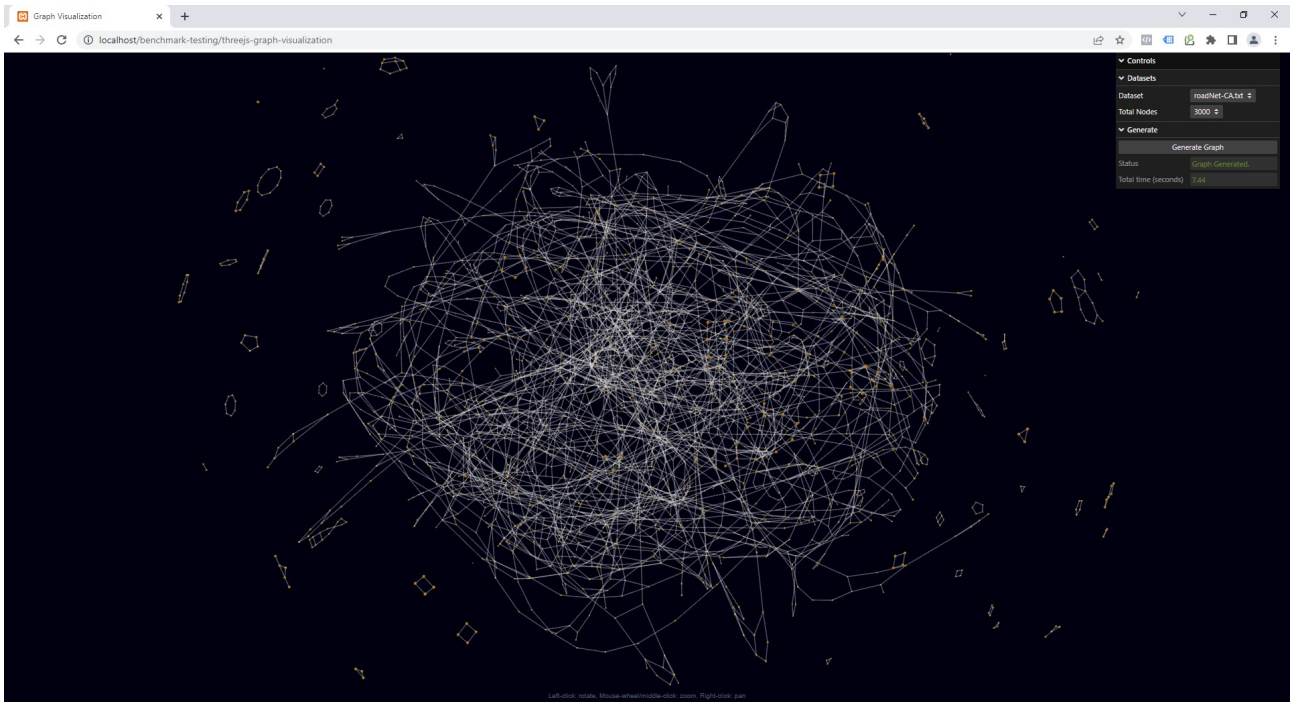
Οι τέσσερις πρώτες γραμμές μας δίνουν πληροφορίες αναφορικά με το dataset, όπως τον τίτλο του και τον αριθμό των κόμβων και ακμών. Στις γραμμή 6, μετατρέπουμε το κείμενο σε λίστα, χρησιμοποιώντας την μέθοδο split() για να δημιουργήσουμε ξεχωριστά στοιχεία στην λίστα για κάθε γραμμή του κειμένου. Στην συνέχεια απομακρύνουμε τις τέσσερις πρώτες γραμμές για να επεξεργαστούμε μόνο τα ζευγάρια κόμβων. Τέλος στις γραμμές 8 με 12, δημιουργούμε τις ακμές και επιστρέφουμε τα δεδομένα σε ένα object.

Μέσα στο index.html, έχουμε το script το οποίο φορτώνει, επεξεργάζεται και οπτικοποιεί τον γράφο. Ο κώδικας του βρίσκεται στην παρακάτω εικόνα:

```
1  async function generateGraph(filePath, n) {
2
3    let gData = await getGraphData(filePath, n);
4
5    const Graph = ForceGraph3D()
6      (document.getElementById('3d-graph'))
7      .nodeLabel('id')
8      .nodeColor(() => '#ffb51f')
9      .graphData(gData);
10
11 }
```

Εικόνα 70: Κώδικας οπτικοποίησης γράφου.

Πρόκειται για μια ακόμη async συνάρτηση η οποία δέχεται τις ίδιες παραμέτρους filePath και n με την getGraphData(). Βλέπουμε στην γραμμή 3 να φορτώνει και να επεξεργάζεται τα δεδομένα και στις γραμμές 5 με 9 να αρχικοποιείται ένα αντικείμενο ForceGraph3D, στο οποίο στέλνουμε τα δεδομένα με την μέθοδο graphData(). Στην παρακάτω εικόνα φαίνεται ένα στιγμιότυπο εκτέλεσης της εφαρμογής.



Εικόνα 71: Στιγμιότυπο εκτέλεσης της εφαρμογής Graph Visualization.

3.3.2 Υλοποίηση με χρήση web worker

Θα χρησιμοποιήσουμε έναν web worker προκειμένου να μεταφέρουμε την φόρτωση και την επεξεργασία των αρχείων στο παρασκήνιο, στέλνοντας στο main thread τα δεδομένα μόλις ολοκληρωθεί η διαδικασία. Θα προσθέσουμε στην εφαρμογή ένα νέο αρχείο worker.js, του οποίου τα περιεχόμενα φαίνονται στην παρακάτω εικόνα:

```
1  import { getGraphData } from './graphHelper.js';
2
3  onmessage = (message) => {
4    let data = message.data;
5    let filePath = data.filePath;
6    let totalNodes = data.totalNodes;
7
8    getGraphData(filePath, totalNodes)
9    .then((gData) =>
10     postMessage({gData})
11   );
12 }
```

Εικόνα 72: Κώδικας αρχείου worker.js.

Στο αρχείο γίνεται import η συνάρτηση getGraphData() του graphHelper.js και μόλις ληφθεί η διαδρομή του αρχείου και ο αριθμός των κόμβων προς απεικόνιση με τις μεταβλητές filePath και totalNodes από το property data του αντικειμένου message, καλείται η συνάρτηση getGraphData(), η οποία εκτελείται σύγχρονα και επιστρέφει ένα Promise [50]. Για να ανακτήσουμε την τιμή από το Promise καλούμε την μέθοδο then(), η οποία λαμβάνει σαν παράμετρο το αποτέλεσμα του ασύγχρονου υπολογισμού της getGraphData(). Μόλις εκτελεστεί η then() και λάβουμε τα δεδομένα, τότε τα στέλνουμε στο main thread με την postMessage(). Έπειτα, ενημερώνουμε το script στο αρχείο index.html, ώστε να προσθέσουμε τον web worker και την επικοινωνία με το main thread:

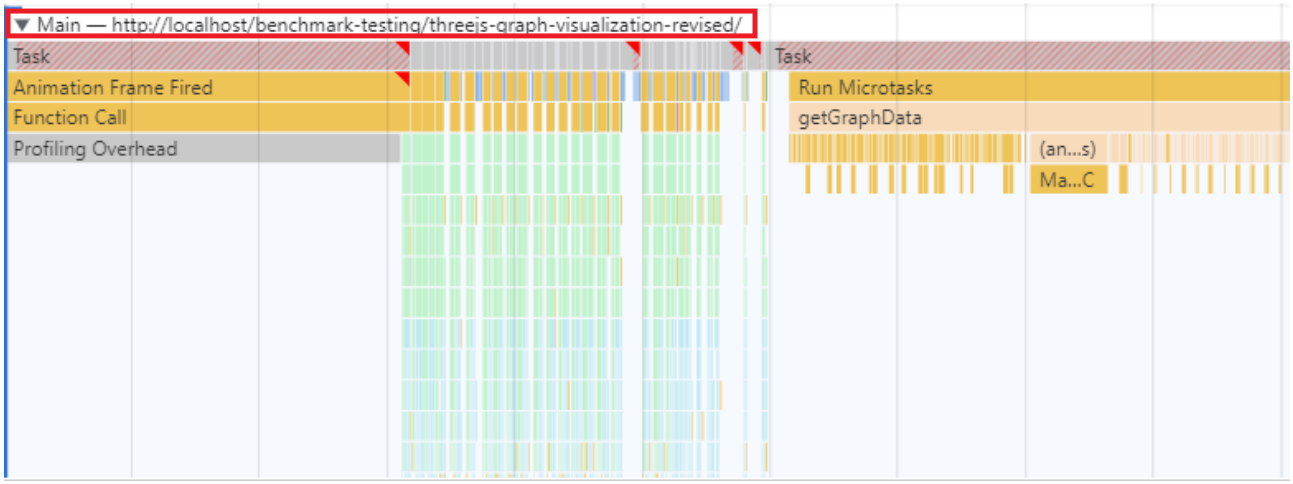
```
1  /* Init app variables. */
2  let filePath = BASE_PATH + FILENAMES[0];
3  let totalNodes = TOTAL_NODES[0];
4
5  let worker = new Worker('worker.js', { type: 'module' });
6  let startTime, totalTime;
7
8  worker.postMessage({filePath, totalNodes});
9
10 worker.onmessage = (message) => {
11   generateGraph(message.data.gData);
12   totalTime = ((performance.now() - startTime) / 1000).toFixed(2);
13   statusController.setValue('Graph Generated.');
14   totalTimeController.setValue(totalTime);
15   generateController.enable();
16 }
```

Εικόνα 73: Κώδικας οπτικοποίησης γράφου με web worker.

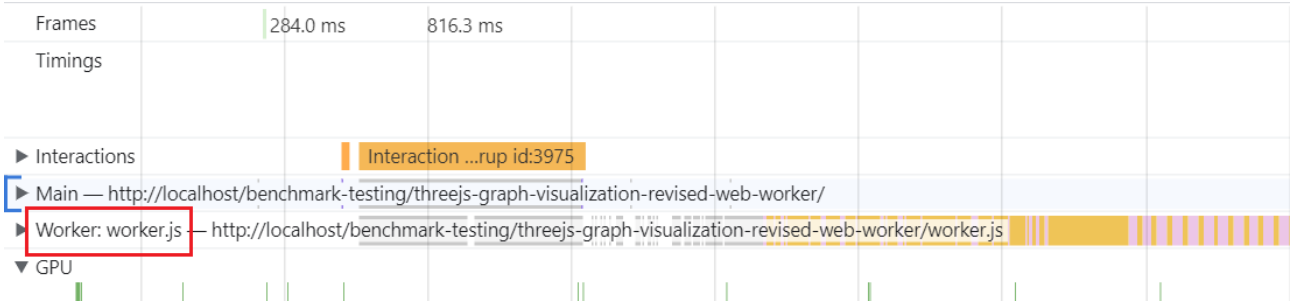
Στις γραμμές 2 με 6 έχουμε την αρχικοποίηση των μεταβλητών και του web worker και στην συνέχεια την υλοποίηση της επικοινωνίας με το main thread, στέλνοντας τις μεταβλητές filePath και totalNodes στον worker, και εκτελώντας την συνάρτηση generateGraph() της προηγούμενης ενότητας, μόλις λάβουμε τα δεδομένα.

3.3.3 Σύγκριση αποτελεσμάτων

Οι αλλαγές στην δυνατότητα απόκρισης και αλληλεπίδρασης της εφαρμογής με τον χρήστη, φαίνονται στις δύο εικόνες που ακολουθούν. Πρόκειται για δύο στιγμιότυπα καταγραφής της απόδοσης της εφαρμογής στον Chrome, χρησιμοποιώντας το εργαλείο της καταγραφής του Performance που μας παρέχει:



Εικόνα 74: Στιγμιότυπο καταγραφής απόδοσης της εφαρμογής χωρίς worker.



Εικόνα 75: Στιγμιότυπο καταγραφής εκτέλεσης της εφαρμογής με worker.

Στην πρώτη περίπτωση (Εικόνα 3.47) βλέπουμε ότι το main thread είναι εκείνο που εκτελεί, εκτός από τις ενημερώσεις των γραφικών, την φόρτωση και τους υπολογισμούς του αρχείου με τους κόμβους. Αυτό φαίνεται από την συνάρτηση getGraphData(), η οποία εκτελείται στο ενδιάμεσο των ενημερώσεων των γραφικών, και καθιστά αδύνατη την απόκριση της εφαρμογής σε τυχόν αλληλεπιδράσεις του χρήστη. Στην δεύτερη περίπτωση (Εικόνα 3.48), όπου ο υπολογισμός έχει μεταφερθεί στον web worker, το main thread δεν απασχολείται καθόλου μου τους εντατικούς

υπολογισμούς και μπορεί και αποκρίνεται στα events που προκαλεί ο χρήστης, χρησιμοποιώντας την εφαρμογή.

4. Επίλογος

4.1 Σύνοψη και συμπεράσματα

Στην εργασία αυτή παρουσιάστηκαν στο 2ο κεφάλαιο η ροή με την οποία οι browsers ακολουθούν για να εκτελούν τις ενέργειες που απαιτούνται για να ενημερωθούν τα διάφορα τμήματα μιας web εφαρμογής και ο βέλτιστος αριθμός FPS για την καλύτερη εμπειρία χρήστη. Πηγαίνοντας στην σχεδίαση και απεικόνιση 3D γραφικών παρουσιάστηκαν η διεπαφή WebGL και η βιβλιοθήκη three.js η οποία βασίζεται πάνω σε αυτή την διεπαφή. Έγινε μια εισαγωγή στους web workers, στον τρόπο με τον οποίο λειτουργούν και επικοινωνούν με το main thread, καθώς και τα προτερήματα και τους περιορισμούς στους οποίους υπάγονται. Στον 3ο κεφάλαιο παρουσιάστηκαν τρεις περιπτώσεις web εφαρμογών με 3D γραφικά, η εφαρμογή παραγωγής 3D τοπίου όπου χρησιμοποιήθηκαν web workers για την επιτάχυνση των υπολογισμών του ύψους του κάθε σημείου στο τοπίο, η εφαρμογή προσομοίωσης των boids σε 3D χώρο, όπου η χρήση των web workers έγινε με σκοπό την επίτευξη υψηλότερων τιμών FPS, και η εφαρμογή οπτικοποίηση 3D γράφου, με την μεταφορά της φόρτωσης και επεξεργασίας των δεδομένων προς απεικόνιση σε worker, με σκοπό την απελευθέρωση του main thread από εντατικούς υπολογισμούς και την αύξηση της απόκρισης με την αλληλεπίδραση του χρήστη. Λαμβάνοντας υπόψη τα αποτελέσματα των μετρήσεων για κάθε εφαρμογή, μπορούμε να πούμε ότι:

- Η χρήση των web workers για την κατανομή του όγκου των δεδομένων προς υπολογισμό οδήγησε σε μείωση του χρόνου εκτέλεσης της διαδικασίας παραγωγής του 3D τοπίου, με τον αλγόριθμο να εκτελείται, κατά το μέγιστο, 1,89 φορές γρηγορότερα απ’ ότι στην σειριακή υλοποίηση. Επίσης, η μέγιστη επιτάχυνση εκτέλεσης για κάθε browsers σημειώθηκε στις περιπτώσεις όπου ο αριθμός των web workers ισούται με τον αριθμό των πυρήνων της CPU.
- Στην εφαρμογή προσομοίωσης, η μεταφορά των υπολογισμών στους web workers είχε ως αποτέλεσμα ο συνολικός αριθμός των FPS να αυξηθεί έως και 8,5 φορές περισσότερο από την σειριακή υλοποίηση, με όλους τους browsers να φτάνουν σε αύξηση των FPS κατά 6 φορές. Παρατηρήθηκε επίσης ότι όσο μικρότερος είναι ο αριθμός των workers, τόσο

υψηλότερος είναι ο αριθμός των FPS, καθώς υπάρχουν λιγότερες μεταφορές δεδομένων μεταξύ των workers και του main thread, χωρίς να μπλοκάρει πολλές φορές η εφαρμογή σε κάθε αποστολή και λήψη δεδομένων.

- Όσον αφορά την μεταφορά του όγκου των υπολογισμών σε web worker, όπως στην περίπτωση της εφαρμογής οπτικοποίησης 3D γράφου, είδαμε ότι έχει θετικό αποτέλεσμα στην αύξηση της απόκρισης της εφαρμογής, καθώς οι υπολογισμοί πραγματοποιούνται στο παρασκήνιο χωρίς να επηρεάζουν την εφαρμογή. Με αυτό τον τρόπο η εμπειρία χρήστη δεν επηρεάζεται από τους όποιους εντατικούς υπολογισμούς.

4.2 Μελλοντικές επεκτάσεις

Στην παρούσα εργασία είδαμε την χρήση των web workers σε συνδυασμό με την βιβλιοθήκη three.js για την επιτάχυνση της απεικόνισης γραφικών. Παρά την προφανή βελτίωση τόσο σε χρόνους υπολογισμού και απεικόνισης, τόσο και σε επίτευξη υψηλού αριθμού FPS και απόκρισης της εφαρμογής, υπάρχουν ορισμένες δυνατότητες επέκτασης της εργασίας, τόσο στο κομμάτι των web workers, όσο και στο κομμάτι απεικόνισης των γραφικών. Συγκεκριμένα:

- Όσον αφορά την επικοινωνία των web workers με το main thread, θα μπορούσε να χρησιμοποιηθούν τύποι δεδομένων, όπως το SharedArrayBuffer, με σκοπό την άμεση ενημέρωση των δεδομένων, χωρίς την αποστολή μηνυμάτων μεταξύ των δυο μερών, με στόχο την περαιτέρω μείωση του χρόνου ενημέρωσης των γραφικών και την παραγωγή καλύτερων αποτελεσμάτων.
- Στο κομμάτι απεικόνισης των γραφικών, θα μπορούσαν να εξεταστούν εναλλακτικά API της WebGL, όπως η WebGPU, η οποία μπορεί να αξιοποιήσει τις νεότερες τεχνολογίες που έχουν αναπτυχθεί και υποστηρίζονται από τις σύγχρονες κάρτες γραφικών.

Με τις παραπάνω προτάσεις για επέκταση, υπάρχουν οι δυνατότητες για την εύρεση μεθόδων που θα οδηγήσουν σε ακόμη καλύτερα αποτελέσματα τόσο στο κομμάτι υπολογισμού και μεταφορά δεδομένων από και προς τους web workers, όσο και στην απεικόνιση των γραφικών.

Βιβλιογραφία

[1] WebKit, (n.d.), WebKit, [Online], Available at: https://webkit.org/ (Accessed: April 2023).

[2] Mozilla Wiki, (n.d.), Gecko:Overview, [Online], Available at: https://wiki.mozilla.org/Gecko:Overview#Document_rendering_pipeline (Accessed: April 2023).

[3] Chromium Project, (n.d.), Blink, [Online], Available at: https://www.chromium.org/blink/ (Accessed: 28 May 2023).

[4] Liccine, Joe, An Introduction to the Browser Rendering Pipeline, [Online], Available at: https://www.webperf.tips/tip/browser-rendering-pipeline/, (Accessed: April 2023).

[5] MDN Web Docs, (n.d.), window.requestAnimationFrame(), [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame (Accessed: April 2023).

[6] Schwab, Michael, et al, Scalable Scalable Vector Graphics: Automatic Translation of Interactive SVGs to a Multithread VDOM for Fast Rendering, 2021, p. 2-3.

[7] WHATWG, (n.d.), The canvas element, [Online], Available at: https://html.spec.whatwg.org/multipage/scripting.html#the-canvas-element (Accessed: April 2023).

[8] Wikipedia contributors, (2023), SVG, In Wikipedia, The Free Encyclopedia, [Online], Available at: https://en.wikipedia.org/wiki/SVG (Accessed: April 2023).

[9] Wikipedia contributors, (2023), Stage3D, In Wikipedia, The Free Encyclopedia, [Online] Available at: https://en.wikipedia.org/wiki/Stage3D (Accessed: April 2023).

[10] Microsoft Corporation, (n.d.), Perspective 3D Graphics, [Online], Available at: https://www.microsoft.com/silverlight/perspective-3d-graphics/ (Accessed: April 2023).

[11] Khronos Group, (n.d.), OpenGL ES Overview, [Online], Available at: https://www.khronos.org/opengles/ (Accessed: April 2023).

[12] Wikipedia contributors. (2023). WebGL. In Wikipedia, The Free Encyclopedia., [Online], Available at: https://en.wikipedia.org/wiki/WebGL (Accessed: April 2023).

[13] Three.js White Paper, Github.com. 2012-05-21, [Online], Available at: https://github.com/mrdoob/three.js/issues/1960 (Accessed: April 2023)

[14] Wikipedia contributors, (2023), Three.js – Features, In Wikipedia, The Free Encyclopedia., [Online], Available at: https://en.wikipedia.org/wiki/Three.js#Features (Accessed: April 2023).

[15] Khronos Group, (2021), OpenGL Shading Language, [Online], Available at: https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language (Accessed: April 2023).

[16] WHATWG, (n.d.), Introduction to web workers, [Online], Available at: https://html.spec.whatwg.org/multipage/workers.html#introduction-14 (Last updated: 26 May 2023) (Accessed: April 2023).

[17] MDN Web Docs, (n.d.). Navigator, [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/API/Navigator (Accessed: April 2023).

[18] MDN Web Docs, (n.d.), Navigator.hardwareConcurrency property, [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/API/Navigator/hardwareConcurrency (Accessed: April 2023).

[19] MDN Web Docs, - Glossary (n.d.), Browsing context, [Online], Available at: https://developer.mozilla.org/en-US/docs/Glossary/Browsing_context (Accessed: April 2023).

[20] MDN Web Docs, (n.d.), SharedWorker, [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/API/SharedWorker (Accessed: April 2023).

[21] MDN Web Docs, (n.d.), Sending messages to and from a dedicated worker, [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers#sending_messages_to_and_from_a_dedicated_worker (Accessed: April 2023).

[22] caniuse.com, (n.d.), SharedWorkers, [Online], Available at: https://caniuse.com/sharedworkers (Accessed: April 2023).

[23] MDN Web Docs, (n.d.). Web Workers API: Worker types, [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API#worker_types (Accessed: April 2023).

[24] Github.com, (n.d.), dom-examples: Simple Web Worker – main.js, [Online], Available at: https://github.com/mdn/dom-examples/blob/main/web-workers/simple-web-worker/main.js (Accessed: April 2023).

[25] ecma International, 2015, ECMAScript® 2015 Language Specification, Available at: https://www.ecma-international.org/wp-content/uploads/ECMA-262_6th_edition_june_2015.pdf

[26] MDN Web Docs, (n.d.), JavaScript modules, [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules (Accessed: April 2023).

[27] MDN Web Docs, (n.d.), WorkerGlobalScope.importScripts(), [Online] Available at: https://developer.mozilla.org/en-US/docs/Web/API/WorkerGlobalScope/importScripts (Accessed: April 2023).

[28] MDN Web Docs, (n.d.). Web Workers API: Structured clone algorithm, [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm (Accessed: April 2023).

[29] MDN Web Docs, (n.d.), Web Workers API: Transferable objects, [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Transferable_objects (Accessed: April 2023).

[30] Bidelman, Eric, (2011), Transferable objects - Lightning fast, (Last Updated: 29 January, 2019), [Online], Available at: https://developer.chrome.com/blog/transferable-objects-lightning-fast/ (Accessed April 2023)

[31] MDN Web Docs, (n.d.), OffscreenCanvas, [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/API/OffscreenCanvas (Accessed: April 2023).
[32] MDN Web Docs, (n.d.), HTMLCanvasElement.transferControlToOffscreen(), [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement/transferControlToOffscreen (Accessed: April 2023).

[33] Hyung Woo Kim, Yang-Won Lee, (2013), Single and Multiple Thread Programming for Geo-visualization by Using WebGL with Web Workers, Proceedings of the World Congress on Engineering and Computer Science 2013 Vol I WCECS 2013, 23-25 October, 2013, San Francisco, USA, Retrieved from: https://www.iaeng.org/publication/WCECS2013/WCECS2013_pp468-473.pdf

[34] Verd´u J., Costa J. J., Pajuelo A., (2010), Dynamic Web Worker Pool Management for Highly Parallel JavaScript Web Applications, Concurrency and computation. Practice and experience, 10 September 2016, vol. 28, num. 13, p. 3525-3539.

[35] Schwab, Michael, et al, Scalable Scalable Vector Graphics: Automatic Translation of Interactive SVGs to a Multithread VDOM for Fast Rendering, 2021, p. 2-3.

[36] MDN Web Docs, (n.d.), Performance.now(), [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.

[37] stats.js, Javascript Performance Monitor, [Online], Available at: https://github.com/mrdoob/stats.js/.

[38] lil-gui, [Online], Available at: https://lil-gui.georgealways.com/.

[39] Wikipedia contributors, (2023), Procedural generation. In Wikipedia, The Free Encyclopedia., [Online], Available at: https://en.wikipedia.org/wiki/Procedural_generation.

[40] Gustavson S., (2005), Simplex noise demystified, Retrieved from: https://cgvr.cs.uni-bremen.de/teaching/cg_literatur/simplexnoise.pdf.

[41] Perlin K., (1985), An Image Synthesizer, Courant Institute of Mathematical Sciences, Volume 19, Number 3, Retrieved from: https://dl.acm.org/doi/pdf/10.1145/325165.325247 (Accessed: May 2023).
[42] Gustavson S., (2005), Simplex noise demystified, Retrieved from: https://cgvr.cs.uni-bremen.de/teaching/cg_literatur/simplexnoise.pdf, p. 8-16 (Accessed: May 2023).

[43] three.js. (n.d.), BufferGeometry, [Online], Available at: https://threejs.org/docs/#api/en/core/BufferGeometry (Accessed: May 2023).

[44] MDN Web Docs, (n.d.), Float32Array, [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Float32Array (Accessed: May 2023).

[45] Wikipedia contributors, (n.d.), Boids, In Wikipedia, The Free Encyclopedia., [Online], Available at: https://en.wikipedia.org/wiki/Boids (Accessed: May 2023).

[46] Reynolds, Craig (1987). Flocks, herds and schools: A distributed behavioral model. SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques. Association for Computing Machinery. pp. 25–34. CiteSeerX 10.1.1.103.7187. doi:10.1145/37401.37406. ISBN 978-0-89791-227-3. S2CID 546350, Retrieved from: https://team.inria.fr/imagine/files/2014/10/flocks-hers-and-schools.pdf (Accessed: May 2023).

[47] Adams, Hunter V. (n.d.), Boids algorithm – augmented for distributed consensus, [Online], Available at: https://vanhunteradams.com/Pico/Animal_Movement/Boids-algorithm.html#Pseudocode (Accessed: May 2023).

[48] vasturiano/3d-force-graph GitHub repository, Available at: https://github.com/vasturiano/3d-force-graph (Accessed: May 2023).

[49] Jure Leskovec and Andrej Krevl, (2014), SNAP Datasets: Stanford Large Network Dataset Collection, [Online], Available at: https://snap.stanford.edu/data/#road (Accessed: May 2023).

[50] MDN Web Docs, (n.d.), Promise, [Online], Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise (Accessed: May 2023).

Παράρτημα Α

```
import { TOTAL_PROPERTIES, TOTAL_BOIDS, BOUND_SIZE_X, BOUND_SIZE_Y, BOUND_SIZE_Z } from "./constants.js";

class Simulation {
  constructor(boidData) {
    this.boidData = boidData;
    this.turnfactor = 0.2;
    this.visualRange = 200;
    this.protectedRange = 100;
    this.centeringfactor = 0.005;
    this.avoidfactor = 0.05;
    this.matchingfactor = 0.05;
    this.xlower = 0;
    this.ylower = 0;
    this.zlower = 0;
    this.xupper = BOUND_SIZE_X;
    this.yupper = BOUND_SIZE_Y;
    this.zupper = BOUND_SIZE_Z;
    this.maxspeed = 6;
    this.minspeed = 3;
  }
  initBoids() {
    for(let i=0; i<this.boidData.length; i+=TOTAL_PROPERTIES) {
      this.boidData[i] = 100 + parseInt(i / (TOTAL_BOIDS / 100 * 6)) * 32;
      this.boidData[i + 1] = parseInt(i / (TOTAL_BOIDS / 10 * 6)) * 32;
      this.boidData[i + 2] = 100 + parseInt(i % (TOTAL_BOIDS / 100 * 6)) * 8;
    }
  }
  updateBoids(start, end) {
    let otherBoidData = this.boidData.slice();
    for(let i=start; i<end; i+=TOTAL_PROPERTIES) {
      let xpos_avg = 0,
          ypos_avg = 0,
          zpos_avg = 0,
          xvel_avg = 0,
          yvel_avg = 0,
          zvel_avg = 0,
          neighbors = 0,
          close_dx = 0,
          close_dy = 0,
          close_dz = 0,
          speed = 0;

      for(let j=0; j<otherBoidData.length; j+=TOTAL_PROPERTIES) {
        if(i == j) continue;
        let dx = this.boidData[i] - otherBoidData[j];
        let dy = this.boidData[i + 1] - otherBoidData[j + 1];
        let dz = this.boidData[i + 2] - otherBoidData[j + 2];
        if(Math.abs(dx) < this.visualRange && Math.abs(dy) < this.visualRange && Math.abs(dz) < this.visualRange) {
          let squared_distance = Math.pow(dx, 2) + Math.pow(dy, 2) + Math.pow(dz, 2);
          if(squared_distance < Math.pow(this.protectedRange, 2)) {
            close_dx += this.boidData[i] - otherBoidData[j];
            close_dy += this.boidData[i + 1] - otherBoidData[j + 1];
            close_dz += this.boidData[i + 2] - otherBoidData[j + 2];
          }
          else if(squared_distance < Math.pow(this.visualRange, 2)) {
            xpos_avg += otherBoidData[j];
            ypos_avg += otherBoidData[j + 1];
            zpos_avg += otherBoidData[j + 2];
            xvel_avg += otherBoidData[j + 3];
            yvel_avg += otherBoidData[j + 4];
            zvel_avg += otherBoidData[j + 5];
            neighbors++;
          }
        }
      }
      if(neighbors > 0) {
        xpos_avg = xpos_avg / neighbors;
        ypos_avg = ypos_avg / neighbors;
        zpos_avg = zpos_avg / neighbors;
        xvel_avg = xvel_avg / neighbors;
        yvel_avg = yvel_avg / neighbors;
        zvel_avg = zvel_avg / neighbors;
        this.boidData[i + 3] = this.boidData[i + 3] + (xpos_avg - this.boidData[i]) * this.centeringfactor + (xvel_avg - this.boidData[i + 3]) * this.matchingfactor;
        this.boidData[i + 4] = this.boidData[i + 4] + (ypos_avg - this.boidData[i + 1]) * this.centeringfactor + (yvel_avg - this.boidData[i + 4]) * this.matchingfactor;
        this.boidData[i + 5] = this.boidData[i + 5] + (zpos_avg - this.boidData[i + 2]) * this.centeringfactor + (zvel_avg - this.boidData[i + 5]) * this.matchingfactor;;
      }
      this.boidData[i + 3] = this.boidData[i + 3] + (close_dx * this.avoidfactor);
      this.boidData[i + 4] = this.boidData[i + 4] + (close_dy * this.avoidfactor);
      this.boidData[i + 5] = this.boidData[i + 5] + (close_dz * this.avoidfactor);
      if(this.boidData[i] > this.xupper) this.boidData[i + 3] -= this.turnfactor;
      if(this.boidData[i] < this.xlower) this.boidData[i + 3] += this.turnfactor;
      if(this.boidData[i + 1] > this.yupper) this.boidData[i + 4] -= this.turnfactor;
      if(this.boidData[i + 1] < this.ylower) this.boidData[i + 4] += this.turnfactor;
      if(this.boidData[i + 2] > this.zupper) this.boidData[i + 5] -= this.turnfactor;
      if(this.boidData[i + 2] < this.zlower) this.boidData[i + 5] += this.turnfactor;
      speed = Math.sqrt(Math.pow(this.boidData[i + 3], 2) + Math.pow(this.boidData[i + 4], 2) + Math.pow(this.boidData[i + 5], 2));
      if(speed < this.minspeed) {
        this.boidData[i + 3] = (this.boidData[i + 3] / speed) * this.minspeed;
        this.boidData[i + 4] = (this.boidData[i + 4] / speed) * this.minspeed;
        this.boidData[i + 5] = (this.boidData[i + 5] / speed) * this.minspeed;
      }
      if(speed > this.maxspeed) {
        this.boidData[i + 3] = (this.boidData[i + 3] / speed) * this.maxspeed;
        this.boidData[i + 4] = (this.boidData[i + 4] / speed) * this.maxspeed;
        this.boidData[i + 5] = (this.boidData[i + 5] / speed) * this.maxspeed;
      }
      this.boidData[i] += this.boidData[i + 3];
      this.boidData[i + 1] += this.boidData[i + 4];
      this.boidData[i + 2] += this.boidData[i + 5];
    }
    return this.boidData.slice(start, end);
  }
}
export default Simulation;
```