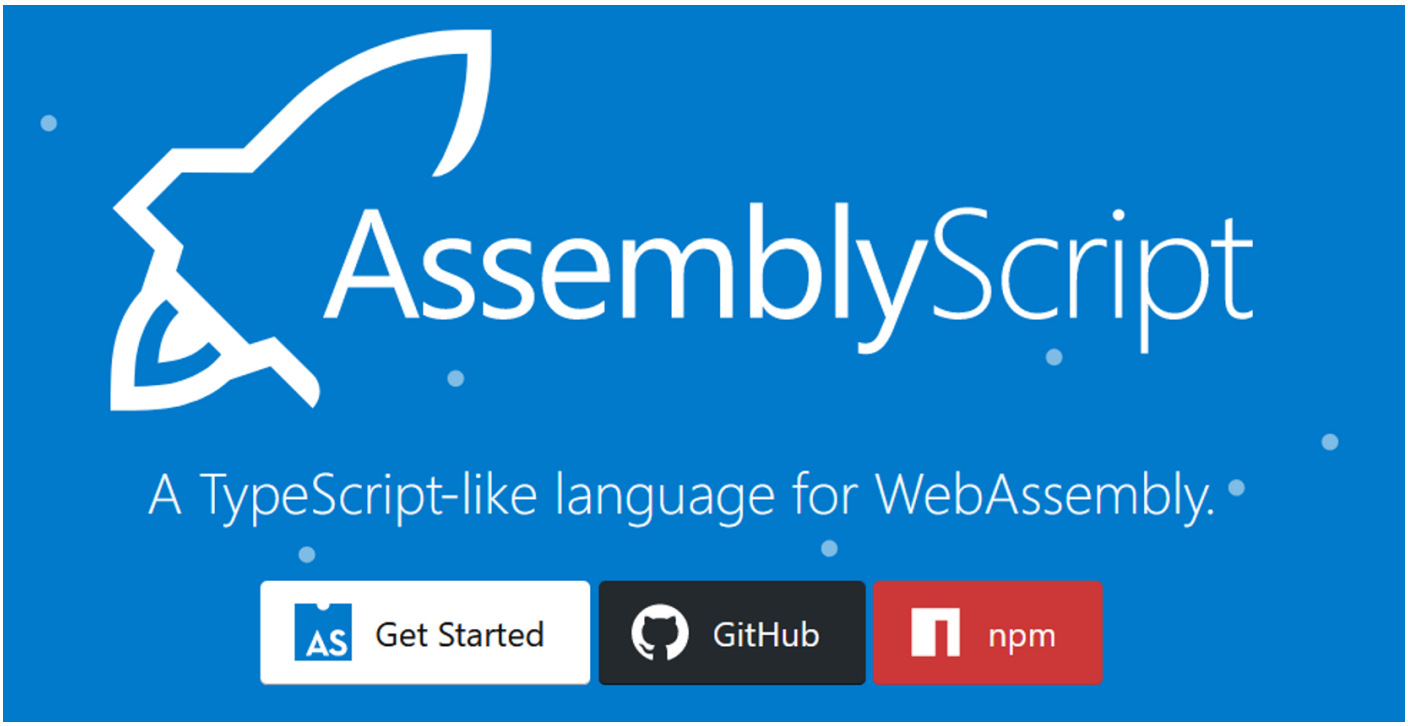
Web Assembly avec JavaScript et AssemblyScript

Il est possible aujourd'hui de faire du code qui sera compilé en WebAssembly de plusieurs manières.

Les deux manières les plus évidentes sont :

- Coder dans un langage bas niveau existant (Go, Rust, C++, etc.) pour ensuite générer un binaire (`wasm`) à l'aide d'outils spécifiques à chaque langage.
- Utiliser le langage « texte » du WebAssembly (`wat`) pour le compiler directement en binaire (`wasm`).

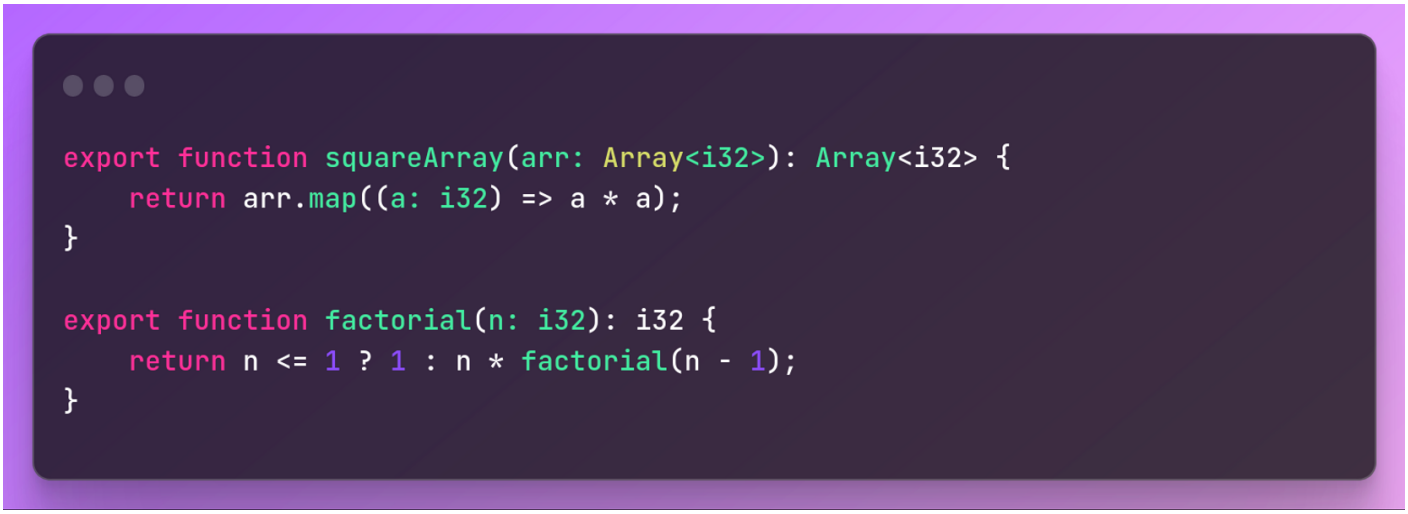
Nous allons aborder une approche intermédiaire, car nous allons utiliser AssemblyScript, un langage basé sur TypeScript conçu pour être compilé en WebAssembly.



Présentation

L'AssemblyScript est un langage de programmation open source et communautaire qui offre une optimisation pour WebAssembly. Il est de typage fort et présente de nombreuses similitudes avec

le TypeScript, ce qui en fait un outil très intuitif sans avoir besoin de maîtriser un nouveau langage de programmation.



```
export function squareArray(arr: Array<i32>): Array<i32> {
    return arr.map((a: i32) => a * a);
}

export function factorial(n: i32): i32 {
    return n <= 1 ? 1 : n * factorial(n - 1);
}
```

AssemblyScript respecte les spécifications du WebAssembly et se base sur binaryen, un compilateur rapide et performant en WebAssembly codé en C++.

Il ne faut pas oublier que JavaScript demeure indispensable dans un navigateur, car même le WebAssembly a besoin de lui pour interagir.

Compatibilité

L'AssemblyScript est compatible avec la plupart des navigateurs, mais aussi avec Node.js, Wasmtime et Wasmer.

Premier projet

Compilation

Il faut bien comprendre que lors de la compilation, l'AssemblyScript ne se compile pas directement en `wasm`. Il faut plutôt le voir comme une transpilation en code `wat` (similaire à la commande `tsc`, qui transpile le TypeScript en JavaScript), qui est ensuite compilé en `wasm`.

Options de compilations

Les options de compilation sont variées et peuvent se passer soit en argument de la commande de compilation (`asc`), soit dans un fichier `asconfig.json` à la racine du projet.

Les principales options de compilation sont les suivantes :

Optimisation :

- `--optimizeLevel`
  - Niveau d'optimisation du code (**0-3**), indique le niveau d'optimisation du code (au détriment de la taille du binaire), **3** étant le niveau d'optimisation le plus haut.
- `--shrinkLevel`
  - Indique à quel point le compilateur va tenter de réduire la taille du binaire (**0-2**).
  - ⚠ Il et déconseillé de mettre une valeur haute à ce paramètre si vous avez également mis des options d'optimisation du code.

Sortie :

- `--outFile`
  - Chemin de sortie du fichier `wasm`
- `--textFile`
  - Chemin de sortie du fichier `wat`

Débogage :

- `--debug`
  - Ajoute des informations de debug dans la version compilée (peut entrainer une réduction des performances)

Fichier de configuration

Les options mentionnées ci-dessus peuvent être spécifiées directement dans le fichier `asconfig.json` de la façon suivante :

```
{
  "targets": {
    "debug": {
      "outFile": "build/debug.wasm",
      "textFile": "build/debug.wat",
      "sourceMap": true,
      "debug": true
    },
    "release": {
      "outFile": "build/release.wasm",
      "textFile": "build/release.wat",
      "sourceMap": true,
      "optimizeLevel": 3,
      "shrinkLevel": 0,
      "converge": false,
      "noAssert": false
    }
  },
  "options": {
    "bindings": "esm"
  }
}
```

Binding & typage

La commande `asc` non seulement compile notre code, mais génère aussi du typage pour permettre d'utiliser le code AssemblyScript depuis du code JavaScript ou TypeScript avec de l'autocomplétion et une certaine sécurité. Cela facilite considérablement le transfert entre les deux mondes.

```
/**
 * assembly/index/add
 * @param a `i32`
 * @param b `i32`
 * @returns `i32`
 */
export declare function add(a: number, b: number): number;
/**
 * assembly/index/sort
 * @param arr `~lib/array/Array<i32>`
 * @returns `~lib/array/Array<i32>`
 */
export declare function sort(arr: Array<number>): Array<number>;
/**
 * assembly/index/squareArray
 * @param arr `~lib/array/Array<i32>`
 * @returns `~lib/array/Array<i32>`
 */
...
```

Limitations

Typage

Il n'y a pas de cast "intelligent", les types entiers et flottants sont donc tous convertis en `number` pour le TypeScript ou le JavaScript. Même si la plupart des types du TypeScript sont supportés, certains ne le sont pas.

Effets de bords

Il n'est pas possible d'accéder au DOM ou au réseau.

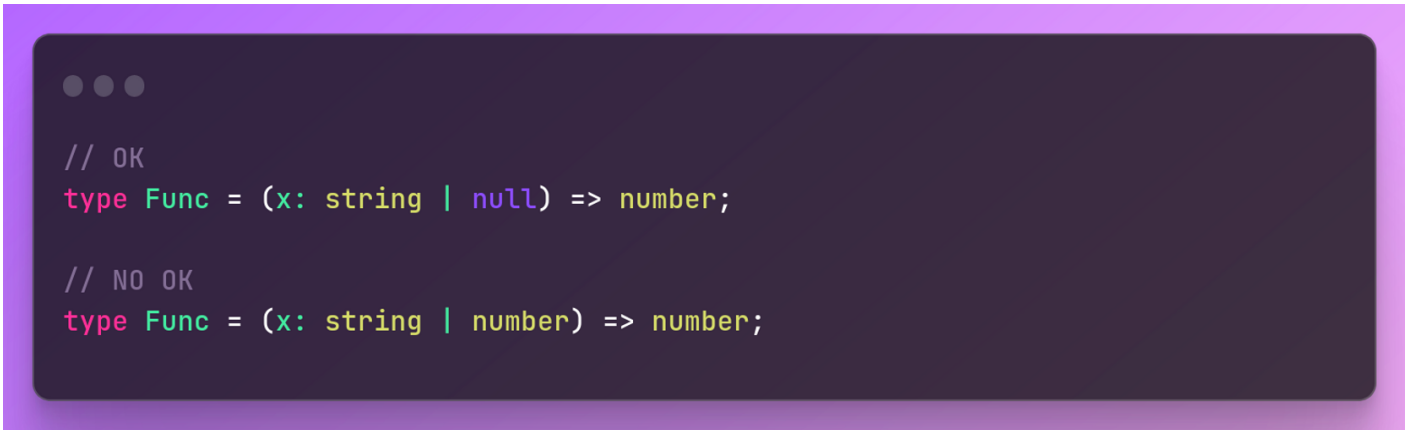
Code synchrone

Les fonctions asynchrones, intervalles, timer, timeout et observables sont à oublier : l'AssemblyScript est un langage synchrone.

Langage statique

Ce n’est pas un langage dynamique comme le TypeScript, on veut, pour que le WebAssembly soit efficace, que tout, soit explicite (les type, les retours). ~~`any, undefined, unknown`~~

On peut avoir du typage alternatif, dans la limite du raisonnable :



```
// OK
type Func = (x: string | null) => number;

// NO OK
type Func = (x: string | number) => number;
```

Performances

Nous ne faisons pas le choix d'utiliser le WebAssembly à la légère.

En général, nous recherchons des performances accrues ou une sécurisation de notre code pour notre application web.

Nous allons tenter de déterminer dans quels cas l'utilisation du WebAssembly pourrait s'avérer être judicieuse, ou pas.

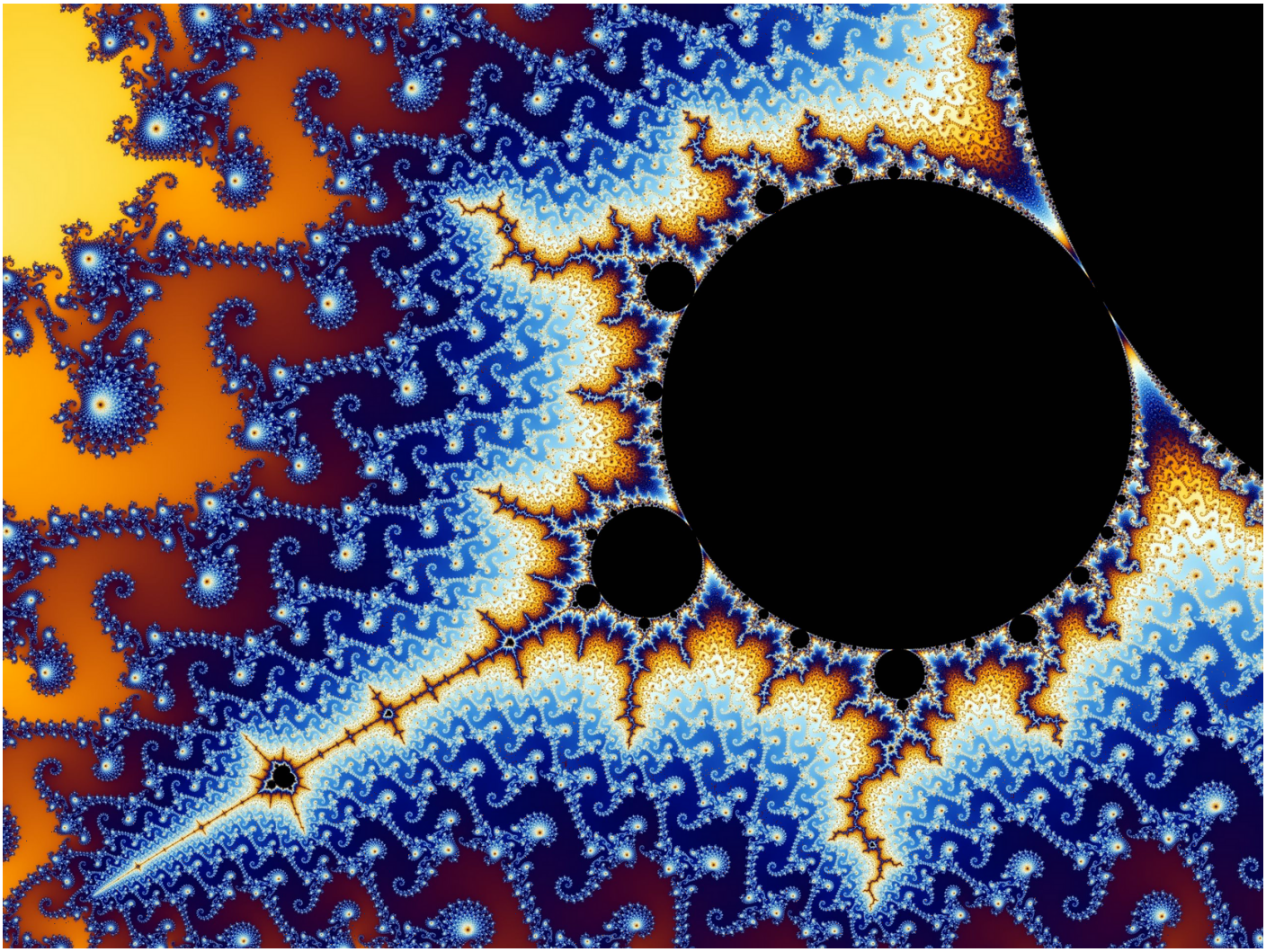
Benchmark

Ensemble de Mandelbrot

L’ensemble de Mandelbrot est une fractale composée d’un ensemble de points du plan complexe qui a pour formule :

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$$

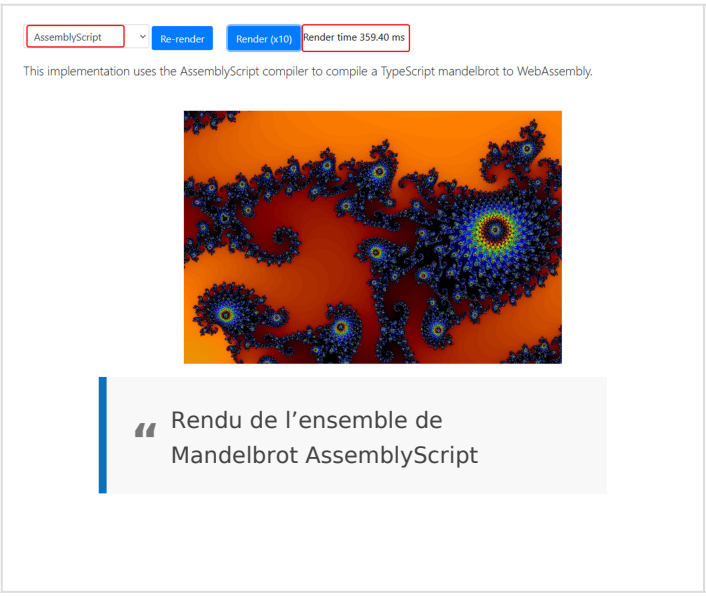
Cette formule permet de générer des images représentant le parcours des nombres complexes sur une région carrée de son plan. Ces images sont très élaborées et constituent un terrain de jeu parfait pour tester le calcul et le rendu de certains programmes.



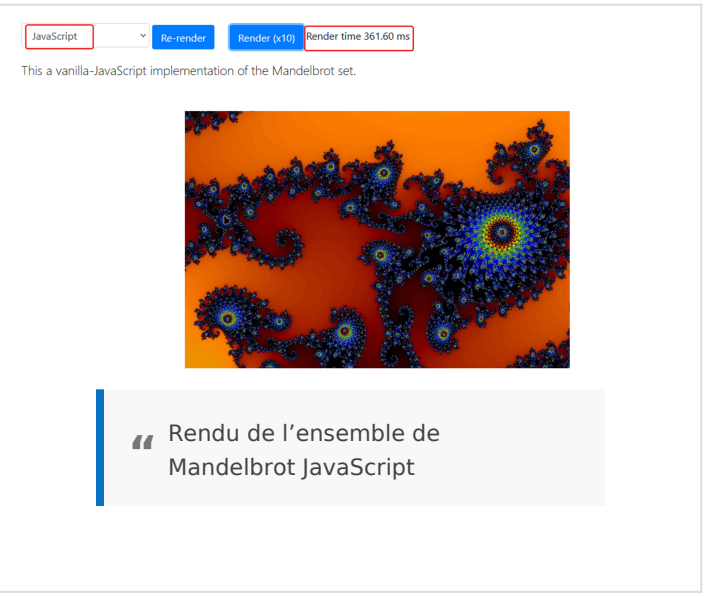
> Exemple d’un ensemble de Mandlebrot

Grâce au projet disponible sur “https://github.com/ColinEberhardt/wasm-mandelbrot”, nous avons différentes implémentations de ce rendu, dont une en **JavaScript** pur et une en **AssemblyScript**, avec le minimum d'adaptation.

Si nous testons le calcul de rendu x10 (pour mettre à l'épreuve notre PC Patate), nous observons les résultats suivants :



> Rendu de l’ensemble de Mandelbrot AssemblyScript



> Rendu de l’ensemble de Mandelbrot JavaScript

Et là, surprise, aucune différence notable n'apparaît.

L'explication vient du fait que l'ensemble de Mandelbrot se génère par des calculs avec essentiellement des nombres à virgule flottante. Il se trouve qu'en JavaScript, le type `number` est un flottant et le JavaScript n'a aucune peine à faire des calculs sur ce type, car il est optimisé pour. Dans le cas d'AssemblyScript, comme le code utilisé est le même que celui en JavaScript avec le minimum d'adaptation, il n'y a pas de miracle : les opérations sur les nombres flottants en WebAssembly ne sont pas aussi optimisées.

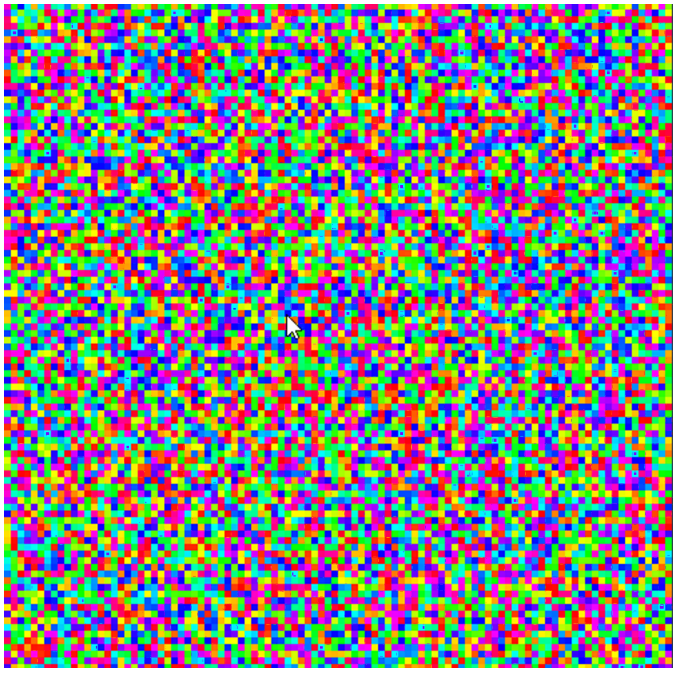
Résultat : le JavaScript se débrouille mieux dans cette situation.

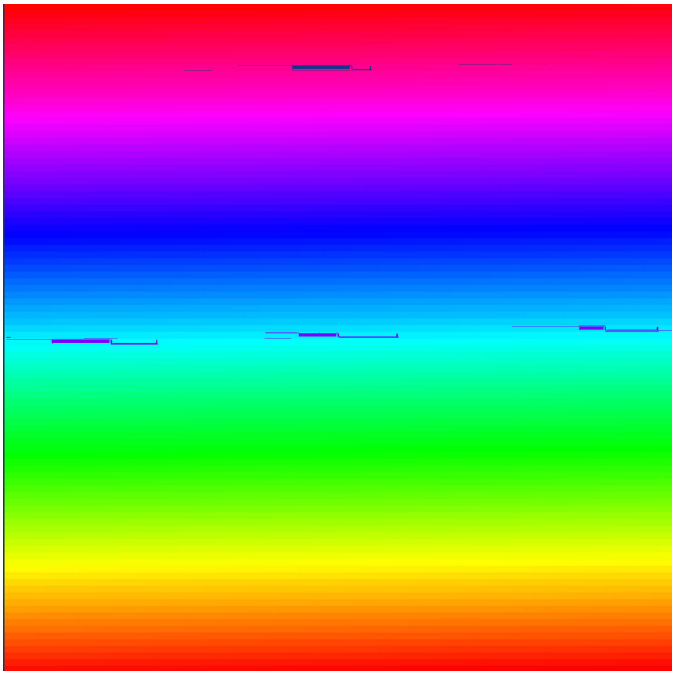
> Il est possible de trouver des versions de l'ensemble de Mandelbrot plus performantes en AssemblyScript, si le code de ce dernier a été pensé pour le langage et n'est pas une simple "adaptation" d'un code existant dans un langage tel que JavaScript.

Tri des couleurs sur une image

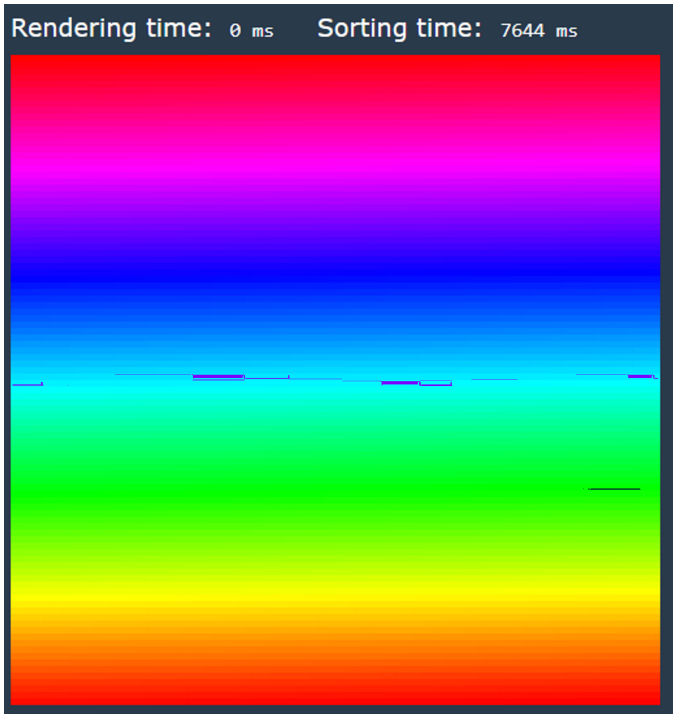
Tournons-nous vers un autre projet, celui-ci : “https://github.com/manueldois/WebAssembly”.

Ce projet consiste à générer une image contenant des couleurs aléatoires, déterminées par une `seed` donnée. Le test de performance va consister à observer le temps que prend le programme pour trier chaque pixel de l’image afin d'obtenir un rendu comme celui ci-dessous :

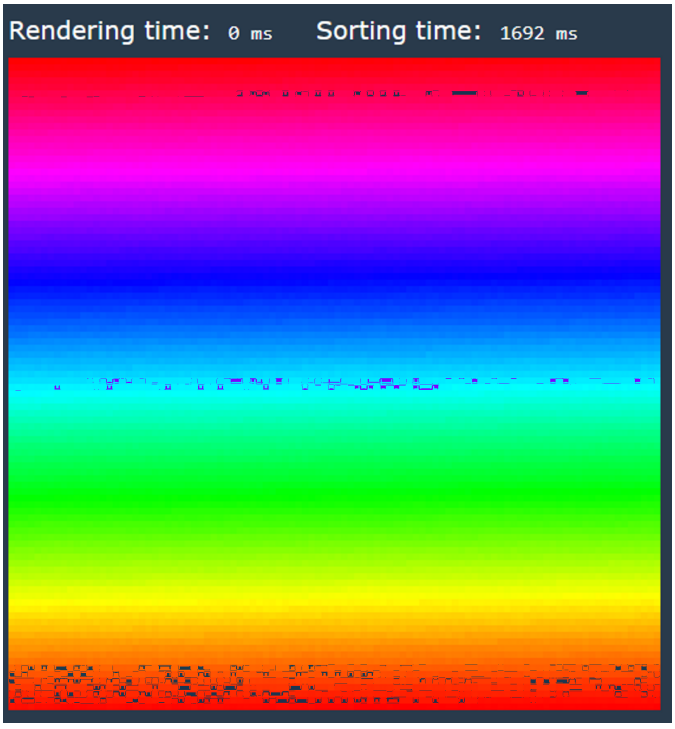




La `seed` utilisée sera le mot “**ESGI**”, observons le temps que prend chaque implémentation pour trier les pixels de notre image :



> Tri des couleurs en JavaScript



> Tri des couleurs en AssemblyScript

Ici, le résultat est sans appel. L'implémentation en AssemblyScript est 4,5 fois plus rapide que celle en JavaScript.

Une image n'est, en fin de compte, qu'un tableau de pixels ayant une valeur représentant sa couleur. Le WebAssembly est tout à fait adapté à ce type d'utilisation.

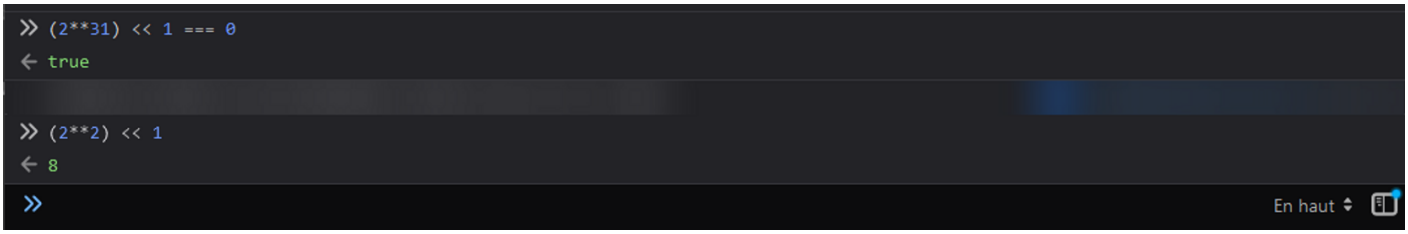
Gestion des nombres et décalage de bits

Là où l'**AssemblyScript** se démarque, c'est dans la manipulation des nombres. En JavaScript, on utilise le type `number`, qui est un nombre à virgule flottante sur 64 bits, ou bien `bigint` (très rare).

En **AssemblyScript**, on a des types qui se calquent sur le WebAssembly, qui permettent de manipuler des nombres entiers sur 32, 64 bits, signés, non signés... (`i32`, `u32`, `i64`, `u64`...). Ces types sont bien gérés par le langage. On peut également utiliser des opérateurs logiques (**bitwise operator**).

Ce n'est pas quelque chose que l'on retrouve souvent dans du code métier, mais c'est un moyen d'optimiser son code. En JavaScript, cela fonctionne très mal (et c'est lent !)... La limitation vient du fait que cela ne fonctionne que sur les 32 premiers bits d'un nombre.

```
(2**31) << 1
// Soit 2^31, décallé de 1 vers la gauche (x2), on devrait retrouver alors 2^32.
// SAUF QUE, essayez cela dans votre navigateur et vous verrez ...
(2**31) << 1 === 0
```



C'est juste inutilisable... À l'opposé de l'**AssemblyScript** qui gère ça très bien !

De nombreux algorithmes peuvent bénéficier de cette optimisation. Prenez l'exemple d'une IA.

Dans le cas d'un apprentissage d'un jeu d'échecs, il est possible de représenter en mémoire de manière très optimisée un échiquier, en utilisant les décalages de bits. On peut ainsi représenter l'état d'un échiquier avec seulement une douzaine d'entiers sur 64 bits.

Cette méthode s'appelle **Bitboards**. Vous pouvez trouver plus d'informations à ce sujet sur ce site : “https://www.chessprogramming.org/Bitboards”

En utilisant ce principe, nous pouvons effectuer un benchmark pour calculer le nombre de coups possibles sur un échiquier après **N** tours.

Si nous jouons 5 tours, le nombre de possibilités s'élève à **4 865 609** ! Pour calculer cela, nous obtenons les résultats suivants :

- Version JavaScript optimisée au mieux : **14 680 ms**
- Version AssemblyScript utilisant les **Bitboards** : **1220 ms**

CQFD.

Conclusion

AssemblyScript est super pour se mettre au WebAssembly, mais exige de revoir sa façon de coder pour obtenir des performances convaincantes et ne pas se contenter de « transposer » son code TypeScript en AssemblyScript. On y gagne rarement.

Il faut aussi savoir être pertinent ! Ce n'est pas parce qu'on va adapter son code (même bien) que cela sera forcément plus performant. Il faut bien étudier le besoin et ne pas hésiter à faire des benchmarks pour déterminer si le passage par le WebAssembly n'est pas une fausse bonne idée.

Ressources

assemblyscript.zip

https://www.assemblyscript.org/introduction.html

https://manueldois.github.io/WebAssembly/Sort Colors Benchmark/dist/index.html

youtube.com/watch?v=3KuDtqFxvRg

youtube.com/watch?v=gCT9ebtTzqw

youtube.com/watch?v=E0Z5oRq8APs

https://manueldois.github.io/WebAssembly/Sort Colors Benchmark/dist/index.html

https://colineberhardt.github.io/wasm-mandelbrot/#AssemblyScript

https://nischayv.github.io/as-benchmarks/graphics/fire/dist/index.html

https://www.chessprogramming.org/Bitboards

Revision #5
Created 2023-05-05 12:50:28 UTC by Noé Larrieu-Lacoste
Updated 2023-05-05 13:11:34 UTC by Noé Larrieu-Lacoste