Examining Performance of Conway's Game of Life in a Web Browser

Student Name: Robbie Jakob-Whitworth

Supervisor Name: Dr Steven Bradley

Submitted as part of the degree of BSc Natural Sciences to the
Board of Examiners in the Department of Computer Science, Durham University.

Abstract –

Context/Background: Conway's Game of Life is a well-known cellular automaton. By applying a set of rules to simple configurations, players create intricate patterns in this "zero-player game".

Aims: This project examines different visualisation techniques in the web browser, primarily (1) using vanilla JavaScript, and (2) React. This project also has the objective of examining whether offloading generation computation to a server is efficient.

Method: I built custom implementations for each of the defined aims. Rendering and generation were implemented client-side in the browser. Comparison of each of these methods took place to determine which was more performant. A simple server-client environment was also partially implemented.

Results: In general, React was more performant than using vanilla JavaScript. A combination of React's diffing algorithms with custom optimisation and diffing techniques led to fast and efficient rendering and updating of the board for each generation. The most effective change, however, is having a dedicated server perform the generation computation rather than the web browser.

Conclusions: React and vanilla JavaScript both provide suitable, relatively performant frameworks to visualise Conway's Game of Life. However, work in this field must continue, as web technology changes so rapidly that any performant visualisation implemented today will be outdated in fewer than five years.

Keywords – Game of Life, John Conway, React, JavaScript, reconciliation, diffing, web profiling.

I. Introduction

A. Context

First popularised after appearing in a Scientific American article (Gardner 1970), Conway's Game of Life is a zero-player game. Its evolution is determined by its initial state, with no further input required. A player creates an initial configuration and allows the simulation to iterate repeatedly, observing how the state evolves.

The Game starts with a two-dimensional infinite grid of cells, where each cell can be alive or dead. The living cells comprise the current population. On each iteration of the Game, the following simple genetic laws (McIntosh 2010) apply:

1. Rules

a. *Survival* – every live cell with exactly two or three neighbouring live cells survives for the next iteration.
b. *Death* – every live cell with four or more alive neighbours dies due to overpopulation. Every live cell with one or zero live neighbours dies from isolation.
c. *Birth* – every dead cell with exactly three live neighbours becomes alive.

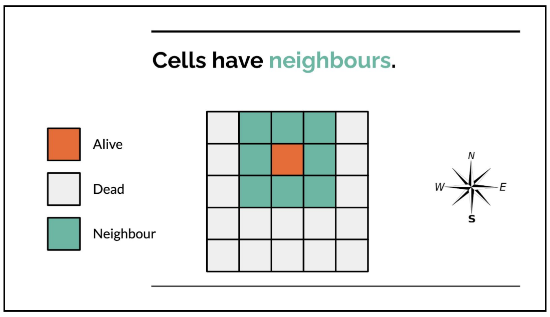


Figure 1 (above). A cell can be alive or dead. A cell has eight neighbours.

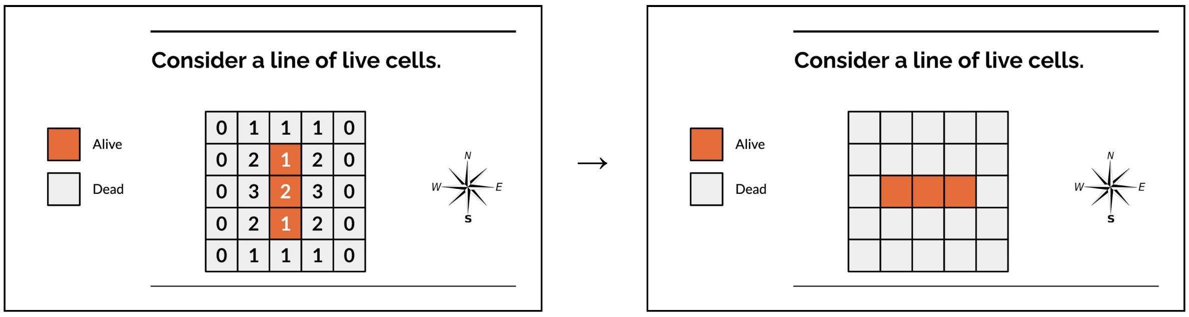


Figure 2 (above). In each iteration, some cells die due to under- or overpopulation, some cells are born, and some survive.

B. Research Question

What is the most performant technique for visualising Conway's Game of Life in a web browser?

This project's goal was to implement and visualise Conway's Game of Life. To determine the most optimal technique, I intend to build a selection of different technical approaches for simulating and visualising the Game.

The intention was also to use advanced rendering techniques, such as only rendering the changes on each iteration rather than re-rendering the entire visualisation. The best way to implement this approach is by using React, which uses diffing algorithms to determine which elements on a webpage to update in response to a change of state (Facebook Inc. 2019).

C. Objectives and Deliverables

1. Basic	*2. Intermediate*	*3. Advanced*
● Use different techniques to implement a frontend simulation of Conway's Game of Life such that it is performant on a **range of browsers**, e.g. ○ `<div>`s in native HTML DOM using "vanilla" JavaScript ○ React virtual DOM	Basic, and: ● Use a **client/server architecture** where the server is responsible for computing each iteration of the Game ● Use different techniques to vary **what** we send: ○ Do we send the whole state on each	Intermediate, and: ● **Intelligently** compute each iteration of the Game: ○ Avoid calculating the next iteration for **completely stable regions** (as these will not change on that iteration), taking

● **Analyse** and **evaluate** the performance of each implementation, **suggesting possible improvements for each**	iteration? ○ Do we just send the changes between iterations? ● Use different techniques to change **how** we send iterations: ○ Ajax with long polling ○ WebSockets	care to respect possible changes at the boundaries of these regions ○ Try to **identify periodic regions of the Game using hashing** and record these to **avoid having to recompute periodically**

II. Related Work

A. Existing Game of Life Visualisations

There exist already many visualisations of the Game of Life. Having been popularised for close to 50 years, these range from implementations using modern web technology to Python visualisations or even older ones.

Many existing web-based visualisations utilise the DOM heavily. The Document Object Model (DOM) refers to the structure of a document (usually the HTML representing a webpage) in the browser and how it is stored in memory (Mozilla Contributors 2020). The DOM represents a document with a logical tree. Each branch of the tree ends in a node (also called an *element*, for example a `<h1>` or `<div>` tag) and each node contains objects (such as plain text or more nodes). JavaScript can be used to interact with the DOM programmatically, by selecting elements and changing their style, structure, or content. This forms the basis of most modern web applications.

A simple, elegant visualisation (Weldon & Laventure 2018) of Conway's Game of Life was submitted as a project to a hackathon, HackUVic 2018. This web-based visualisation uses a 100x100-cell grid and is rather rudimentary in its approach (which is to be expected, given that it is a hackathon project). All computation happens on the client-side, with 10,000 nodes in the DOM comprising the grid, which is not particularly performant when the iteration interval is set to the minimum option, 100 milliseconds.

A more complete and polished visualisation (Bettilyon 2018) is also a client-side, web-based visualisation. However, it utilises the modern HTML5 canvas and, as such, can achieve a vastly more responsive simulation than that of Weldon and Laventure.

Another accessible visualisation (Vanderplas 2013) uses Python and NumPy to compute iterations for the Game of Life. It then passes the calculated results to the author's package, JSAnimation, which is responsible for visualising the calculated results on a webpage. This approach renders complex populations smoothly and promptly as no computation takes place during render-time.

Arguably the most well-known and respected visualisation is the application Golly (Trevorrow & Rokicki 2006), which has native apps for Windows, macOS, Linux, Android and iOS. It contains support for many different types of cellular automaton rule configurations, as well as implementations of many different algorithms for simulating with those rules. Moreover, Golly provides scripting support for Lua or Python, with some standard scripts enabling it to produce animated GIFs of patterns.

While there are several web-based implementations of Game of Life visualisations, these tend to be reasonably simple implementations with little consideration given to the performance or suitability of a particular visualisation. Typically, all computation occurs client-side, leaving the browser to do complex calculations for large grids. Conway's Game of Life is in fact Turing-complete (Durand & Róka 1999), and so the potential for large simulations in the Game is interesting as a result.

This project addresses some of these concerns by using more advanced rendering techniques to reduce the load on the web browser and produce a more performant implementation.

B. Computing New Iterations

While the rules of the Game of Life are simple, it is shown that small, finite patterns can produce emergent structures of immense complexity, given enough iterations (Gotts 2010). The famous "glider" structure uses just five live cells and a bounding box of 3x3 (Sapin 2010) but translates itself across the board indefinitely. Moreover, the "glider gun" (36 live cells, contained within a 36x9 bounding box) produces gliders, which as they themselves move, have the effect of growing the non-empty part of the board with no upper bound.

It follows that, over time, each iteration of the simulation could increase in complexity. Golly implements an algorithm called Hashlife (Trevorrow & Rokicki 2006). Hashlife is a memoised[1] algorithm which computes the long-term outcome of a given starting configuration in Conway's Game of Life (and related cellular automata). It is known for doing so with impressive speed and comparatively small computational load.

Memoisation (Abelson, Sussman & Sussman 1997), also called tabulation, refers to storing previously computed values in a lookup table. When a memoised procedure calculates a result, it first checks the table to see if the result has already been calculated. If the computed result is not found in the lookup table, the result is calculated ordinarily and stored in the lookup table for future use.

This approach is used in Hashlife by splitting the board into subpatterns. For example, a commonly reused pattern (like the aforementioned "glider gun", or indeed merely large regions of space) can be hashed and mapped to the lookup table. Many copies of the same subpattern can be stored just once using the corresponding hash table entry (Gosper 1984). Hashing values in this way avoids having to recompute the same value over and over, instead essentially caching the result.

C. Web-Based Client-Server Communication

The most basic architecture for retrieving data from a server using the web is through a single page load. The user requests a web page, and the server responds with some (static) contents (Berners-Lee & Cailliau 1990). After the server responds, additional data must be requested by requesting a new web page. In this architecture, the client must request some data from the server for the server to respond; the server is unable to "push" data directly to the client without the client having requested it.

While it is technically possible to have a web page periodically self-reload, the delay is

[1] Most dictionaries only list the US-variant spelling *memoization*. However, Wikipedia lists *memoisation* as an acceptable spelling, and as this paper is written in British English, the *-isation* variant shall be adopted throughout.

too high for many applications, and decreasing the delay would lead to unnecessarily many network requests. In the context of a Game of Life visualisation, this approach would lead to abysmal user experience as the page would have to reload for every iteration!

This "stateless" architecture is dated and much of the modern web now heavily utilises Ajax (Zepeda & Chapa 2007). Ajax enables a client to request additional data from a server without having to reload the entire web page. Google Maps, for example, heavily relies on Ajax to dynamically load new regions of the map as the user pans around the area, without having to reload the page. Such a technique could be used for a Game of Life visualisation if the server were to, say, send blocks of computed iterations in response to a client making successive requests. However, Ajax still faces the limitation that the client must request data from the server for the server to respond.

One previously popular model that attempts to circumvent this limitation is Comet, also known as Ajax 2.0 (Crane & McCarthy 2008). Comet utilises a "long-polling" technique. Because the client must initiate a connection to the server, the server is unable to send data directly to the client without the client having first requested it. "Comet" itself is an umbrella term (Russell 2006), encompassing several techniques in which a long-held HTTP(S) request allows a web server to push data to a browser (the client), without the browser explicitly having requested it. The client must make the first initial Ajax request, but the server does not respond until it wishes to. Long-polling allows a server to effectively push data to a client only when it needs to rather than needing to wait for an incoming request from the client. For a long time, this was considered the best way to send data asynchronously to a browser (Kachhwaha & Patni 2012). Indeed, Facebook Messenger used a variant of this approach when it first launched (Letuchy 2008). Implementing Comet, however, is non-trivial and can present scalability issues (Mesbah & van Deursen 2008).

WebSocket (Puranik, Feiock & Hill 2013) is a modern standard that seeks to address these issues. Long-polling itself can be memory and network intensive and is an anti-pattern to how Ajax was initially designed. Unlike HTTP, WebSocket provides full-duplex communication, enabling a server to send a stream of data to the client with significantly reduced overhead and significant performance improvements (Lubbers & Greco n.d.). It is ideal for situations where a web server needs to continuously send data to a browser – perfect for repeatedly sending new states and iterations of a Game of Life implementation for a browser to then visualise on the client-side.

III. Solution

In this project, I successfully implemented two distinct Game of Life visualisations. Both relied entirely on client-side computation and rendering. No server was used, beyond hosting a static server on the user's machine to serve the files from the local filesystem to the browser over HTTP.

The two visualisation techniques successfully implemented from the basic objectives are as follows:

- `<div>`s in native HTML DOM using "vanilla" JavaScript
- React virtual DOM

A. Architecture

As the vanilla JavaScript and React approaches had a lot of shared elements, the two applications were built using a modular approach. By isolating the simulation and computation into a separate JavaScript module, this module was able to be reused between both implementations with very few changes.

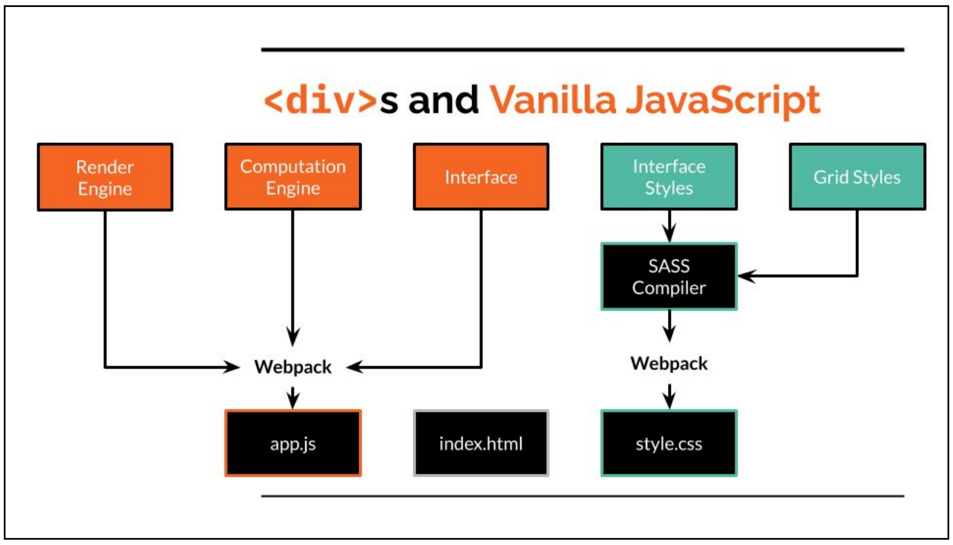


Figure 3 (above). Architecture overview.

Webpack is used as a module bundler to compile the separate JavaScript modules into one single JavaScript file app.js. This bundling has the benefit of a web browser only having to make one request for the JavaScript, and it can cache this response, rather than having to make several requests for separate scripts. Moreover, Webpack creates a dependency graph and only bundles the minimum number of required JavaScript modules, reducing overall file size and also the number of files (Singh 2019). Webpack has numerous other benefits, in addition to improved speed and dependency management, with one key example being scoping, or name collisions. Webpack gives each module and global variable a unique name, eliminating the possibility of name collisions or scoping issues, without having to resort to immediately-invoked function expressions (O'Brien, Montoya, Menichelli, *et al.* 2020).

Stylesheets are compiled similarly from Sass, which is a CSS preprocessor scripting language that gets compiled to standard CSS. Sass provides powerful features such as variables and mixins (Nesbitt 2020), making development more straightforward. Using webpack to compile JavaScript and Sass is standard in modern web development.

B. Vanilla JavaScript Implementation

The first implementation used vanilla (plain) JavaScript without any frameworks like React.js, jQuery, Vue.js, or any number of numerous JavaScript libraries.

Because the intention was to compare this approach with a React.js-based implementation, the two implementations were kept as structurally and architecturally similar as possible to compare React's rendering approach to a vanilla JavaScript one.

1. Interface

Modernism was embraced, and a clean and straightforward interface for visualising Conway's Game of Life was developed. This interface is shared by both the vanilla-JavaScript-based and the React-based visualisations.

Cells are illuminated upon mouseover to indicate over which cell the user is hovering. To ease the drawing and erasing of live or dead cells, the user can hold the `Alt` or `Control` keys to switch between Drawing and Erasing mode, respectively (see Figure 4 and Figure 5).

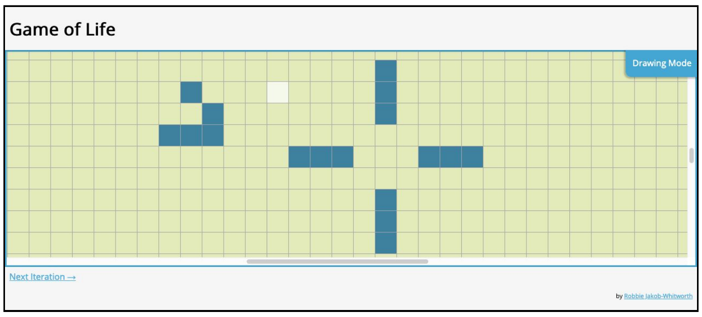


Figure 4 (above). Interface with `Alt` key pressed to enable Drawing Mode. Observe the highlighted cell, over which the user's cursor is currently hovering.

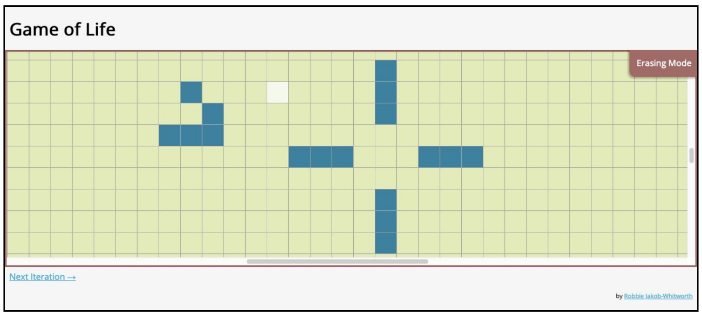


Figure 5 (above). Interface with `Control` key pressed to enable Erasing Mode.

After drawing the user's desired initial state, the user can click the `Next Iteration →` button to iterate the simulation (Figure 6).

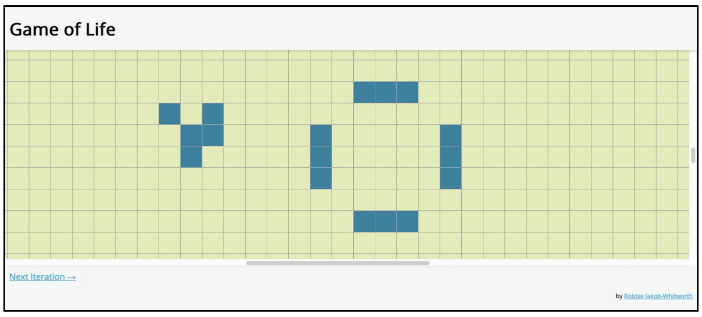


Figure 6 (above). Board after the user has progressed to the next iteration.

2. *Initial Render*

The original `index.html` file encodes the 'external' interface elements (the header and footer) as static elements. However, the grid and the cells are created dynamically by JavaScript when the page loads. As the grid is generated dynamically, we (as the developer) can specify the dimensions of the grid as $m \times n$. Each row is its own DOM element to improve source code readability and maintainability. Within each row, each cell exists as a separate node.

If there are m rows and n columns, then the number of DOM elements to display the grid is as follows:

1. Each row requires $n + 1$ elements to be stored in the DOM (n cells and 1 node for the row).
2. There are m rows.
3. There is 1 container node.
4. Therefore there are $m \times (n + 1) + 1$ nodes in the DOM for the grid.

This is important because it shows that the number of nodes that must be parsed or rewritten on each iteration tends to varying linearly with the number of rows or columns.

3. State Computation and Final Render

When the user clicks `Next Iteration →`, the application computes the new board and renders the application with the new state. The process for this is as follows:

Step 1: Get Current Board

The application begins determining the current state of the board. The board state is not held anywhere in the application memory apart from the existing state of the DOM. Cells are classed as alive or dead, depending on whether or not they have the `alive` class.

This process makes heavy use of `for` and `forEach` loops.

1. Select all row nodes.
2. For each row, get the child nodes (cells).
3. For each cell, determine whether the cell is alive (i.e. has the `alive` class) and push this Boolean value onto an array which stores the row's cells' alive statuses.
4. This procedure gives us a two-dimensional array of values, for example:

```
board = [
  [true, true, false, ...]
  ...
  [false, true, false, ...]
]
```

Step 2: Compute New Cell Statuses

Taking the two-dimensional `board` array from step 1, the application loops through every row and cell and determines whether the cell should be alive on the next iteration.

1. Create a two-dimensional array for the statuses of the cells on the new iteration.
2. Count the number of neighbours the cell has, making sure to consider boundary cases (a cell in a corner of the board only has three neighbours, for example).
3. Look at the current status of the cell, then apply the rules (I.A.1) to determine whether the cell should be alive in the new generation, and store the result.

Step 3: Render the Updated Board

This step is rudimentarily implemented with vanilla JavaScript and would be far better achieved using React (see React Implementation).

1. Get each row and cell DOM node.
2. For each cell, check against the two-dimensional array whether the new cell status is alive or dead, and add or remove the `alive` class accordingly.

This process is a little convoluted and relies on storing the board state in the classes of DOM nodes, which is not ideal. It would be better to store the state separate from the DOM to avoid having to read the state from the DOM. It would also be better to use a more structured way of re-rendering cells rather than using nested for loops.

C. React Implementation

The second implementation is entirely a React app. The `index.html` file contains just two tags within the body.

With this approach, the board state is stored in the `<GameOfLife />` React component, and React is responsible for rendering the changed components rather than us having to render those changes manually.

1. Introduction to React[2]

React is a JavaScript framework used heavily by the modern web. Its popularity among developers is due to how it embraces the fact that, for dynamic web applications, rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

Instead of artificially separating *technologies* by putting markup and logic in separate files, React instead separates *concerns* through loosely coupled units called "components" that contain both markup and logic.

Every React component can be represented as a JavaScript class, and so the "state" for that component is stored in either the class instance or its parent. If the state is stored in a parent component, then relevant values are passed down to the child component as "props" (properties). A component cannot change its own props, but it can change its own state (and potentially the states of its parents, if callback functions to achieve this are passed down to the child component as props).

For example, a React component can be written as the following:

```
import React from "react";

class Cell extends React.Component {
    render() {
        return (
            <div
                onMouseEnter={(e) => this.props.handleCellMouseEnter(e)}
                className="cell"
            />
        );
    }
}

export default Cell;
```

This JavaScript class uses JSX, a syntax extension to JavaScript used to describe React "elements" in JavaScript files. It is a hybrid between plain HTML and JavaScript that allows components to directly refer to their state or props through `this.state` and `this.props`. In the example above, the component returns a `<div>` element with an event listener attached that calls a function `handleCellMouseEnter()`, which is passed to the component through its props.

[2] This description borrows heavily from the official React documentation (Facebook Inc. 2020).

JSX cannot itself be parsed by the browser, and so it is transpiled to “vanilla” JavaScript using babel, as part of the build process in Webpack (see Figure 3).

React performs advanced diffing algorithms (Facebook Inc. 2019) to determine whether an element should update based on its state and props. If the component should update, the `render()` function of the component is called to determine the output based on the state and props. This approach allows developers to define a component functionally and have it automatically re-render on the page where necessary, and is consequently very powerful.

2. *Interface*

The React-based visualisation has visually the same appearance as the vanilla JavaScript implementation. This is because its HTML output is identical, except the React implementation separates the page into React components whereas the first implementation is simply a mixture of plain HTML and some vanilla JavaScript.

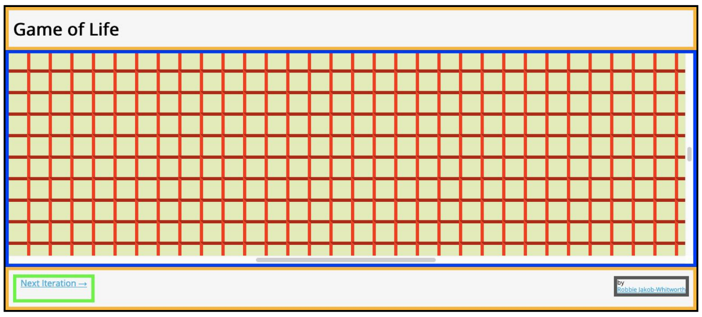


Figure 7 (right). Interface of React app, with React components highlighted.

3. *Initial Render*

The HTML markup produced by React for the board is identical to the markup produced by the vanilla JavaScript implementation (see III.B.2). Rows are represented with distinct DOM nodes, and each cell exists as a separate node within that row.

```
<GameOfLife>
 ...
 <Grid>
  <Row>
   <Cell />
   ...
   <Cell />
  </Row>
  ...
  <Row>...</Row>
 </Grid>
</GameOfLife>
```

The key differences start when we examine how the rows are initially painted in the browser. As the whole page is a single React app, components are provided with “props” to define how they should render. In this implementation, the hierarchy of the React app is illustrated to the right.

We must initialise each component to define how the components should render on page load. The key initial props used are the height and width of the grid (number of rows and cells within rows to create, respectively) and the initial board state.

As per the first implementation, the board is represented as a two-dimensional array. However, because we are using React components, the board is stored in the “state” (this term has a special meaning in React; `this.state` is an object on the React component that represents the parts of an app that can change).

We generate a two-dimensional array and fill it with the value `false` to represent the board being empty (all cells are dead) on page load.

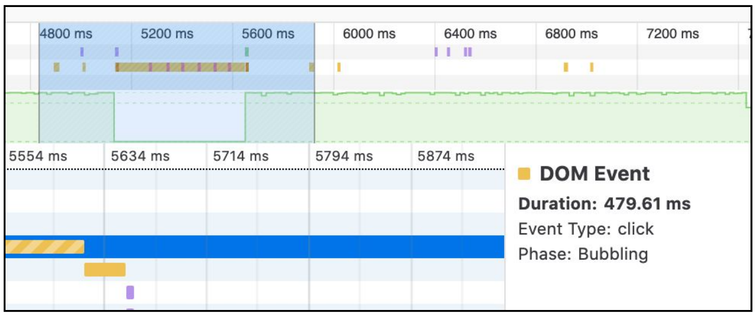
However, storing the board as a two-dimensional array causes us some issues, which we will examine in more detail below.

4. *State Computation and Final Render*

One of React’s key strengths is that it only re-renders a component when its state or props have *changed*. This means that, in theory, if the board is in a stable position (say, all cells are dead) and a re-render is triggered (if the user clicks `Next Iteration →`), none of the cells will be rendered on the page again.

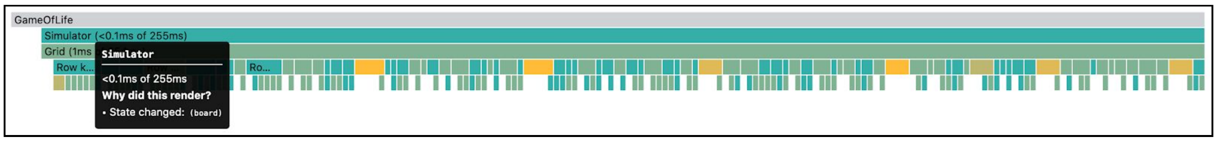
In practice, however, this is not the case using our current two-dimensional array implementation. When determining whether state or props for a component have changed, React does *shallow comparison* on the old vs new state and props objects. This means that if we store the board in the state in a nested format, like a two-dimensional array, React does not do a deep comparison and so it may not detect a change. However, because JavaScript is pass-by-reference for objects and arrays, we find that in fact, because our iteration computation engine returns a new board array regardless of whether its contents have changed, **React always re-renders all children of the `<GameOfLife />` component**.

This led to very poor performance in the React implementation. In some cases, it would take almost 500 ms to render a cell as alive/dead after clicking on it to toggle its state. This is obviously unacceptable.



DOM Event
Duration: 479.61 ms
Event Type: click
Phase: Bubbling

Figure 8 (left). Screenshot from the Performance tab of the Firefox Developer Tools. When the click event is fired after clicking on a cell to toggle its alive state, it took hundreds of milliseconds before the DOM was updated and the new cells painted on the webpage.



Simulator
<0.1ms of 255ms
Why did this render?
• State changed: (board)

Figure 9 (above). Screenshot from the Profiler tab of the React DevTools in Firefox. On every iteration, every single component re-rendered. In this graph, the bottom blocks each represent a cell that has re-rendered because its parent has "changed". This leads to serious performance issues.

The issue of poor performance was first addressed by changing the `<Cell />` component from a `React.Component` to a `React.PureComponent`. The `<Cell />` component is purely functional in that it always gives exactly the same rendered output, given the same set of props as an input, and so declaring it as a `React.PureComponent` had the effect of slightly increasing performance.

Moreover, memoisation was also implemented (see II.B) – which is trivial in React, simply export the component as `export default React.memo(Cell)` rather than `export default Cell` – and enabling this significantly increased performance. Memoisation stores the result of a computation to avoid having to recompute it, so enabling it unsurprisingly made a notable difference to performance.

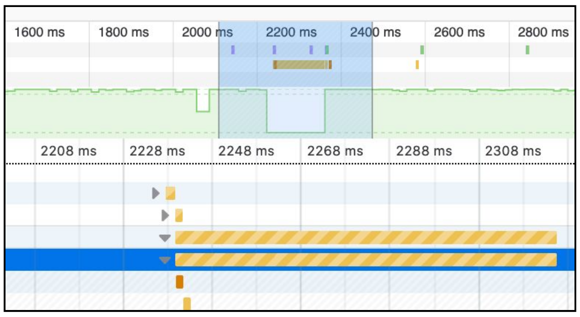


Figure 10 (left). After enabling memoisation and changing the `<Cell />` component from a `React.Component` to a `React.PureComponent`, performance improved significantly. This Event Handler only took 85.89 ms compared to the 479.61 ms of Figure 8.

The other key change made to improve performance was to manually override React's diffing algorithms. As mentioned earlier, React's diffing algorithm does shallow comparison for state and props, meaning that it does not detect deep changes inside a nested state or props object. Alternatively, due to objects and arrays being pass-by-reference in JavaScript, React can mistakenly detect changes when there are no semantic changes.

On each render, each row received a copied (and possibly modified) version of the previous state's cell statuses (stored as an array of `true` or `false`). This meant React would always "detect" a change due to doing shallow comparison on a pass-by-reference object. To prevent this, a custom deep comparison algorithm was manually implemented by overloading the `shouldComponentUpdate()` React method.

We pass the new "row board" (a subset of the two-dimensional `board` array containing only this row's cell statuses) and the old row board (the row board from the previous iteration) and check whether any of the cell statuses have changed. If none of them have changed then the row is the same as it was in the previous iteration, and therefore no changes need to be made. This approach means React does not have to render the component to the shadow DOM and therefore reduces the number of components that need to be re-rendered significantly.

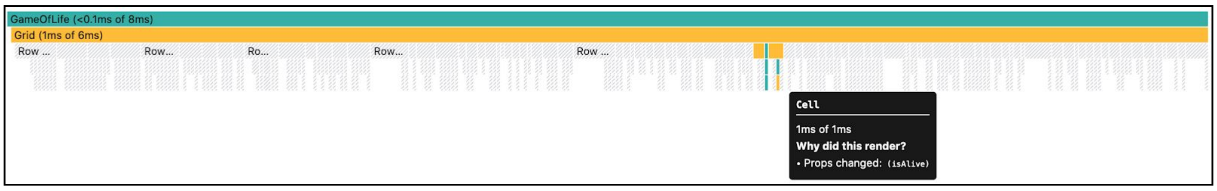


Figure 11. Observe how only affected Row and Cell components render rather than all of them when a change to the board is made.

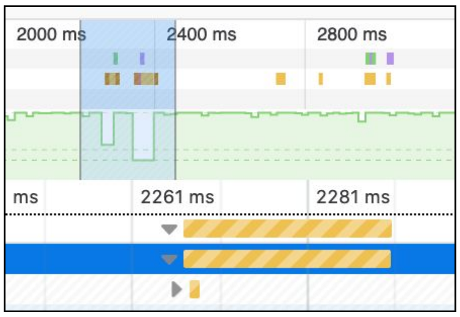


Figure 12 (left). Performance is now greatly improved. In this screenshot, the Event Handler takes just 23.60 ms when clicking on a cell toggle it.

IV. Results

A. Testing Conditions

In order to compare the React and vanilla JavaScript implementations, a glider was drawn on the board in each implementation and 20 iterations were made successively. The glider graphic is used because it translates itself diagonally across the board, moving (1, 1) after four iterations. All tests were performed under the following conditions:

- Google Chrome 84.0.4147.105 (Official Build) (64-bit)
- macOS Catalina 10.15.6
- Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz
- 32 GB 2400 MHz DDR RAM
- Intel UHD Graphics 630 1536 MB
- 100x100 grid

B. Vanilla JavaScript Implementation

Table 1 (below). Vanilla JavaScript, 20 iterations of a glider.

Iteration No.	Time to Get Current Board (ms)	Time to Calculate New State (ms)	Time to Render Updated Board (ms)	Total
1	11.0	62.7	19.8	93.5
2	7.4	57.8	15.0	80.2
3	9.6	55.5	16.1	81.2
4	6.6	56.4	15.3	78.3
5	10.0	55.5	16.0	81.5
6	6.3	55.3	15.4	77.0
7	11.4	55.2	14.9	81.5
8	5.9	55.8	15.3	77.0
9	6.0	55.5	15.8	77.3
10	6.2	55.3	15.1	76.6
11	10.1	53.8	15.2	79.1
12	5.9	55.4	15.4	76.7
13	5.8	55.7	16.5	78.0
14	6.8	55.8	15.1	77.7
15	10.7	54.2	15.2	80.1
16	6.3	55.2	14.9	76.4
17	6.0	54.9	17.7	78.6
18	7.1	55.8	15.4	78.3
19	6.1	55.9	15.4	77.4
20	6.0	56.6	15.5	78.1

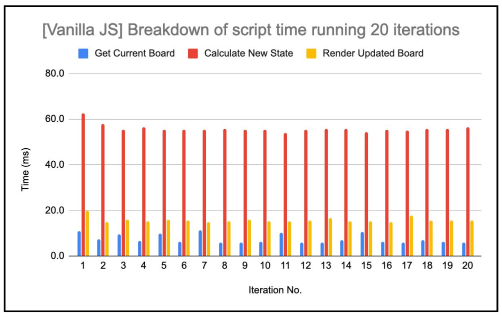


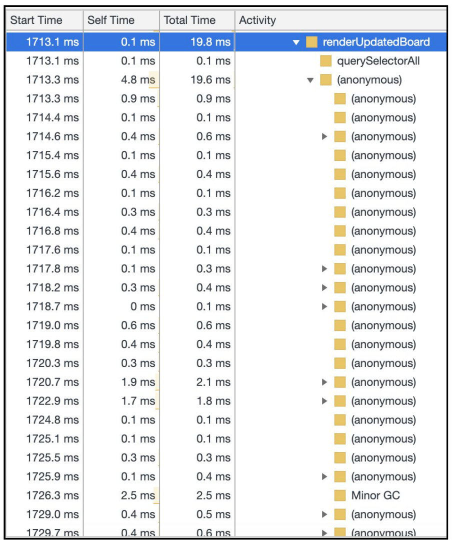
Figure 13 (above). Vanilla JavaScript, 20 iterations of a glider.

From Table 1, we can observe that 9.54% of the time spent per iteration, on average, was spent on calculating the current board state from the DOM. Recall that the single source of truth for whether a given cell is alive or dead is whether that cell DOM node has the `alive` class. On average, the browser spends 7.6 ms computing the current board state.

It is also clear that the vast majority of the time spent at each iteration is on calculating the new state and applying the Game of Life rules (see I.A.1). This step takes 55.9 ms on average per iteration and is the longest part of the computation process because intense array manipulation occurs. For all of

Mean (ms):	**7.6**	**55.9**	**15.8**	**79.2**
Percentage of time:	9.54%	70.58%	19.88%	100%
Standard Deviation (ms):	1.97	1.752	1.126	3.641

the mn cells on the board, each of each cell's eight neighbours are examined, and the rules applied based on the number of live neighbours and the current status of that cell. Therefore roughly $9mn$ cells are examined. This means that, for every cell on the board, up to nine cells (including the target cell) are parsed by reading several arrays within the board. This stage is unavoidably computationally expensive, and might be better served if offloaded to a high performance server.



Start Time	Self Time	Total Time	Activity
1713.1 ms	0.1 ms	19.8 ms	▾ renderUpdatedBoard
1713.1 ms	0.1 ms	0.1 ms	querySelectorAll
1713.3 ms	4.8 ms	19.6 ms	▾ (anonymous)
1713.3 ms	0.9 ms	0.9 ms	(anonymous)
1714.4 ms	0.1 ms	0.1 ms	(anonymous)
1714.6 ms	0.4 ms	0.6 ms	▸ (anonymous)
1715.4 ms	0.1 ms	0.1 ms	(anonymous)
1715.6 ms	0.4 ms	0.4 ms	(anonymous)
1716.2 ms	0.1 ms	0.1 ms	(anonymous)
1716.4 ms	0.3 ms	0.3 ms	(anonymous)
1716.8 ms	0.4 ms	0.4 ms	(anonymous)
1717.6 ms	0.1 ms	0.1 ms	(anonymous)
1717.8 ms	0.1 ms	0.3 ms	▸ (anonymous)
1718.2 ms	0.3 ms	0.4 ms	▸ (anonymous)
1718.7 ms	0 ms	0.1 ms	▸ (anonymous)
1719.0 ms	0.6 ms	0.6 ms	(anonymous)
1719.8 ms	0.4 ms	0.4 ms	(anonymous)
1720.3 ms	0.3 ms	0.3 ms	(anonymous)
1720.7 ms	1.9 ms	2.1 ms	▸ (anonymous)
1722.9 ms	1.7 ms	1.8 ms	▸ (anonymous)
1724.8 ms	0.1 ms	0.1 ms	(anonymous)
1725.1 ms	0.1 ms	0.1 ms	(anonymous)
1725.5 ms	0.3 ms	0.3 ms	(anonymous)
1725.9 ms	0.1 ms	0.4 ms	▸ (anonymous)
1726.3 ms	2.5 ms	2.5 ms	Minor GC
1729.0 ms	0.4 ms	0.5 ms	▸ (anonymous)
1729.7 ms	0.4 ms	0.6 ms	▸ (anonymous)

Figure 14 (left). Each of these quick anonymous functions is run for a cell within a row. Functions which contain further nested functions are those where a class was added or removed, i.e. the cell's alive status was changed.

C. *React Implementation*

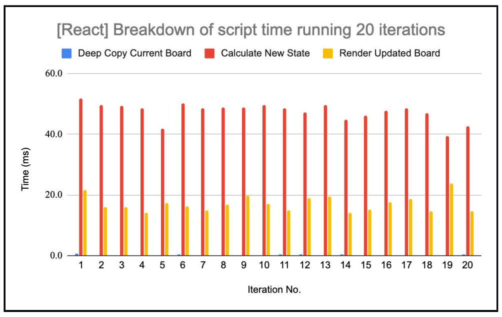


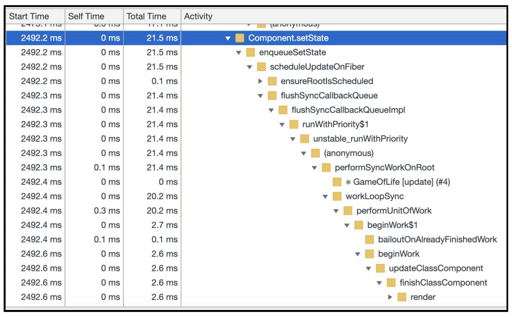
Figure 15. React, 20 iterations of a glider.

Table 2 shows the benefit of storing the board in a React component rather than encoded in the DOM. The mean time for retrieving the current board in this algorithm is just 0.18 ms, only 0.28% of the time required to visualise the next iteration.

Interestingly, the state calculation was approximately 15% faster on average than the same task in the vanilla JavaScript implementation. It is possible that less garbage collection takes place because of the reduced overhead in the previous task, therefore lowering the overhead of this task.

Table 2. React, 20 iterations of a glider.

Iteration No.	Time to Get Current Board (ms)	Time to Calculate New State (ms)	Time to Render Updated Board (ms)	Total
1	0.6	51.8	21.5	73.9
2	0.1	49.6	16.1	65.8
3	0.1	49.2	15.9	65.2
4	0.1	48.6	14.0	62.7
5	0.1	41.7	17.4	59.2
6	0.3	50.2	16.3	66.8
7	0.1	48.6	14.9	63.6
8	0.1	48.7	16.9	65.7
9	0.1	48.9	19.8	68.8
10	0.2	49.6	17.1	66.9
11	0.3	48.5	15.0	63.8
12	0.3	47.1	19.0	66.4
13	0.1	49.7	19.4	69.2
14	0.3	44.8	14.1	59.2
15	0.1	46.2	15.2	61.5
16	0.1	47.8	17.7	65.6
17	0.1	48.5	18.6	67.2
18	0.1	46.9	14.6	61.6
19	0.1	39.5	23.9	63.5
20	0.3	42.7	14.7	57.7
Mean (ms):	**0.18**	**47.43**	**17.11**	**64.72**



```
Start Time   Self Time   Total Time   Activity
2492.2 ms    0 ms        21.5 ms      ▼ Component.setState
2492.2 ms    0 ms        21.5 ms        ▼ enqueueSetState
2492.2 ms    0 ms        21.5 ms          ▼ scheduleUpdateOnFiber
2492.2 ms    0 ms         0.1 ms            ▶ ensureRootIsScheduled
2492.3 ms    0 ms        21.4 ms            ▼ flushSyncCallbackQueue
2492.3 ms    0 ms        21.4 ms              ▼ flushSyncCallbackQueueImpl
2492.3 ms    0 ms        21.4 ms                ▼ runWithPriority$1
2492.3 ms    0 ms        21.4 ms                  ▼ unstable_runWithPriority
2492.3 ms    0 ms        21.4 ms                    ▼ (anonymous)
2492.3 ms    0.1 ms      21.4 ms                      ▼ performSyncWorkOnRoot
2492.4 ms    0 ms         0 ms                            GameOfLife [update] (#4)
2492.4 ms    0 ms        20.2 ms                        ▼ workLoopSync
2492.4 ms    0.3 ms      20.2 ms                          ▼ performUnitOfWork
2492.4 ms    0 ms         2.7 ms                            ▼ beginWork$1
2492.4 ms    0.1 ms       0.1 ms                                bailoutOnAlreadyFinishedWork
2492.6 ms    0 ms         2.6 ms                              ▼ beginWork
2492.6 ms    0 ms         2.6 ms                                ▼ updateClassComponent
2492.6 ms    0 ms         2.6 ms                                  ▼ finishClassComponent
2492.6 ms    0 ms         2.6 ms                                    ▶ render
```

Percentage of time:	0.28%	73.29%	26.43%	100%
Standard Deviation (ms):	0.13	3.003	2.560	3.743

Figure 16 (left). Observe how many nested functions are called as part of the `setState()` function.

The final task, rendering the updated board, is interestly a little slower on average than in the vanilla JavaScript implementation. Its times, however, are less consistent, with a standard deviation of 2.560 ms compared to the first implementation's 1.126 ms. Taking a look at the call stack of the `Component.setState()` function (determining whether or not to re-render a component) using the performance tools in the Chrome Developer Tools (Figure 16), it is apparent that the `render()` function is part of the bottleneck. The `render()` function, as part of `beginWork$1`, is called when the component's state or props have changed. However, the `render()` function is only called four times within this call stack, and much of the delay is from the fixed overhead of React. While React is known for being fast, the framework overhead is not insignificant here.

However, the call stack for the `renderUpdatedBoard()` function in the vanilla JavaScript implementation (Figure 14) shows a different story. Every cell within a row has an anonymous function operated against it, usually taking a fraction of a millisecond, but these add up quickly when there are hundreds of rows and columns. It is clear that React is faster at re-rendering cells, but has a fixed overhead that hurts performance. It is thus concluded that as the dimensions of the grid (and thus the number of cells) increases, React becomes the clear winner, as its fixed overhead becomes a negligible cost and its faster per-component re-rendering times lead to it being the more performant implementation overall.

D. node.js Implementation

It is clear that, on the whole, React is faster than vanilla JavaScript when rendering large amounts of DOM elements due to the optimisations it makes. Moreover, it is well suited to this form of visualisation, where data is stored in a single multidimensional array and DOM nodes represent each of the array's values.

However, in order to further investigate, and to implement some of the intermediate objectives of this project, the iteration computation was offloaded to a node.js server. While this server did not yet interface with the frontend of the application, it did demonstrate significantly improved performance when compared to having the web browser perform the state calculation.

Table 3 (below). node.js v14.7.0, 20 iterations of a glider.

Iteration No.	Time to Calculate New State (ms)
1	37.9
2	27.4
3	31.1
4	32.4
5	27.9
6	30.8
7	28.3
8	33.7
9	32.8
10	27.9
11	27.6
12	31.1
13	34.2
14	31.8
15	30.0
16	33.8
17	31.0
18	28.4
19	28.3
20	30.1
Mean (ms):	**30.83**
Standard Deviation (ms):	**2.696**

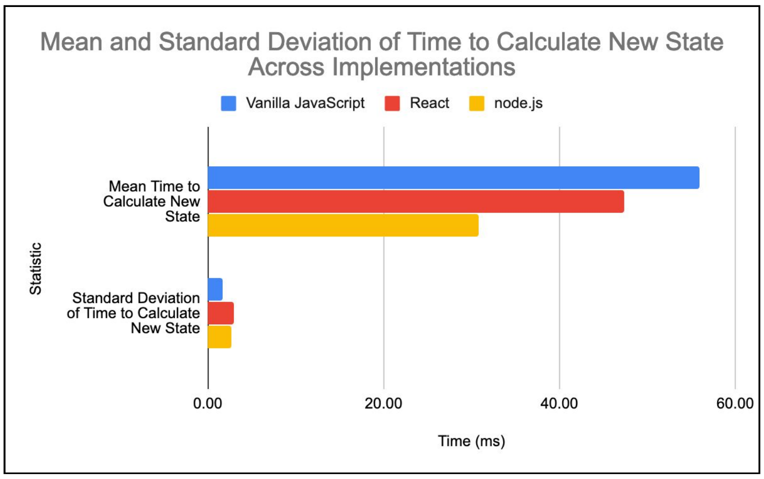


Figure 17. node.js v14.7.0, 20 iterations of a glider.

Figure 17 shows us how much of a performance increase we get using node.js compared to vanilla JavaScript in the browser. The same test was performed – iterating a 100x100 grid with a single glider on it 20 times. This performance increase suggests that it would be worthwhile to offload the iteration computation to a server to enable the client to be responsible solely for rendering the output of the computation.

It is worth noting that there would be additional overhead from the network connection used to communicate between the client and the server, whether this be Ajax, WebSockets or indeed another technique, which could potentially negate any improvement brought about by implementing this approach.

V. Evaluation

A. Future Work

It is clear that there is a lot of benefit to offloading the iteration computation to a server. In the node.js implementation (see IV.D) the mean iteration computation time was 30.83 ms, nearly half the 55.92 ms when using client-side vanilla JavaScript. This suggests that there is definitely scope for building very performant applications by having a dedicated server perform the state computation. Network overhead is something to consider here, however, as a slow or high-latency connection has the potential to bottleneck performance significantly. Experimenting with different connection protocols, such as Ajax or WebSockets, would likely lead to finding an optimal solution.

There remains potential for further optimisation in the actual state computation itself. Rather than just relying on a more optimised or powerful platform, there exist approaches which reduce the computational complexity of the iteration calculation. One such example, discussed in section II.B, is to use memoisation. Hashlife (Trevorrow & Rokicki 2006) is the de facto standard memoisation algorithm used in Game of Life simulations and implementing it on a server-side simulator would likely yield significant performance improvements. Many configurations of the board in the Game of Life lead to periodic patterns, which would benefit immensely from being cached using memoisation.

B. Vanilla JavaScript Implementation

1. Achievements

The vanilla JavaScript implementation is clean and effective in its approach. Able to handle a large number of cells whilst remaining responsive and performant, its lightweight footprint means it has a very small file size. Moreover, debugging and maintaining the application is simple as it does not require any prerequisite knowledge of a particular framework.

2. Improvements

There is significant scope for improvement in the most simple implementation. Storing the current board state in the physical DOM is not efficient. When the application calculates a new iteration, 10% of the time per iteration is just retrieving the current status of the board from the DOM by parsing all the existing nodes.

It would also be wise to decouple the rendering logic from the computation logic. In this specific implementation, the `computation-engine.js` file is tied closely to the DOM and would benefit from being extracted to its own separate module to allow easy reuse.

Moreover, continuously parsing and updating the DOM is memory- and CPU-intensive, and this implementation is not scalable. Therefore, the vanilla JavaScript implementation is not a particularly performant technique for visualising Conway's Game of Life.

C. React Implementation

1. Achievements

The React implementation is appropriately architected and is more performant than the vanilla JavaScript implementation. Indeed, the React framework provides several optimisation techniques such as memoisation and custom diffing (see III.C.4), which this implementation makes heavy use of in order to increase performance. While performance was poor with the default React configuration, making some of the described changes increased performance significantly to the point at which it was faster than the vanilla JavaScript implementation, despite the overhead of using a ~40 kB framework.

From a development perspective, React allows the application to have a sensible architecture which lends itself to good maintainability and future improvement.

With this implementation, the `computation-engine.js` file was simplified such that it only contained the core logic for calculating the next iteration of the board, given an existing board. This had the benefit that this module was able to be included directly in the node.js implementation with no changes needing to be made to the file at all.

2. Improvements

As noted, the React framework is approximately 40 kB in size. This is a large framework whose presence in an application dwarfs the application's own native code. One alternative could be to use a framework called Preact (Miller & Hagemeister 2020), a lightweight alternative to React that uses the same syntax. For a relatively simple use case such as this one, Preact may suffice. It is only 3 kB, so it is a far lighter library to include with the application build.

Further optimisations could certainly be made to the React application. Parts of the rendering process could be improved by further implementing custom diffing to override

React's diffing, which does not support deep comparison. Alternatively, to avoid the shallow/deep comparison issue in the first place, values could be stored in the state using a flat hierarchy rather than multidimensional arrays. While this would be more difficult to develop and maintain, it would enable React to perform its own diffing directly on the state and props of each component between iterations, which would likely be significantly faster than the custom diffing algorithms currently implemented.

D. node.js Implementation

The node.js implementation is not complete. The core logic for calculating new iterations of the board state is developed and a server-side script exists for processing this. However, it would need to be connected to the frontend interface, and significant changes would need to be made to the frontend in order to accommodate retrieving the state from a server.

This is, however, definitely a route worth pursuing as the statistics show that it brings about a significant performance increase. Furthermore, this would provide the opportunity to implement parallelisation to compute iterations, and would also provide a suitable environment for implementing the memoisation algorithm Hashlife.

There already exist node.js packages on npm such as *memoizee* and *node-gol-hashlife* which implement memoisation and the Hashlife algorithm respectively. These would be suitable for usage in a client-server environment such as the one developed here.

E. Project Management

This project was not developed in quite as structured or a planned way as intended. However, a simplified git flow model (Driessen 2010) was adopted, enabling each implementation to be distinct from the others and for a clear commit history to exist in the git repository.

Netlify was used for static hosting. Each implementation was hosted on its own static server during development, facilitating rapid testing and evaluation of each implementation. This also enabled detection of build errors much faster, as with each commit each application was rebuilt from scratch as part of a continuous integration development workflow.

F. Further Comments

The various algorithms implemented in this project are sufficient for visualising the Game of Life and also provide interesting opportunities for testing and experimentation. In particular, using custom diffing for React provides us with a scientific opportunity to compare our own custom-built algorithm with the existing general-purpose React shallow comparison.

The existing implementation of the rules of the Game of Life in the computation engine provides an opportunity to contrast against more specific algorithms such as the Hashlife algorithm. It is worth noting that the *node-gol-hashlife* algorithm (Harel 2015) explicitly states that it should be used for "far-future" evolutionary computation of the simulation, and not for "evolving the simulation one generation at a time".

VI. Conclusion

From our implementations, it is evident that JavaScript can be used effectively and efficiently to visualise Conway's Game of Life in the browser. However, the slowest part of the generation process by far is the iteration computation. For both the vanilla JavaScript and React implementations, over 70% of the time spent on each iteration is computing the new generation of the board.

Preliminary experimentation with a node.js server yielded results significantly faster than having the browser perform this computation. With additional optimisations such as memoisation and the Hashlife algorithm, the time spent calculating the next state of the board could be significantly reduced. While network overhead is a consideration, with an appropriate architecture and implementation, this could probably be reduced such that offloading the computation to a separate server is still beneficial.

Research in this field must continue, with Conway's Game of Life being over 50 years old at the time of writing. Web technology adapts so quickly that any visualisation implemented today will be out-of-date within five years. More work can be done to speed up rendering in the browser and this developing field leaves ample scope for future optimisation.

References

Abelson, H., Sussman, G.J. and Sussman, J. (1997) Structure and interpretation of computer programs, (second edition). *Computers & Mathematics with Applications*. [Online] 33 (4), 133. Available at: doi:10.1016/S0898-1221(97)90051-1.

Berners-Lee, T. and Cailliau, R. (1990) *WorldWideWeb: Proposal for a HyperText Project*. [Online]. 12 November 1990. Available at: https://www.w3.org/Proposal.html (Accessed: 31 October 2019).

Bettilyon, T.E. (2018) *The Game Of Life (Simulation)*. [Online]. 2018. Available at: https://tebs-game-of-life.com/single-large/single-large.html (Accessed: 21 October 2019).

Crane, D. and McCarthy, P. (2008) *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Berkely, CA, USA, Apress.

Driessen, V. (2010) *A successful Git branching model*. [Online]. 5 January 2010. nvie.com. Available at: http://nvie.com/posts/a-successful-git-branching-model/ (Accessed: 9 August 2020).

Durand, B. and Róka, Z. (1999) The Game of Life: Universality Revisited. In: M. Delorme and J. Mazoyer (eds.). *Cellular Automata: A Parallel Model*. Mathematics and Its Applications. [Online]. Dordrecht, Springer Netherlands. pp. 51–74. Available at: doi:10.1007/978-94-015-9153-9_2 (Accessed: 20 October 2019).

Facebook Inc. (2020) *Introducing JSX – React*. [Online]. 9 March 2020. Available at: https://reactjs.org/docs/introducing-jsx.html (Accessed: 8 August 2020).

Facebook Inc. (2019) *Reconciliation*. [Online]. 21 February 2019. React. Available at: https://reactjs.org/docs/reconciliation.html (Accessed: 14 October 2019).

Gardner, M. (1970) *Conway's Game of Life: Scientific American, October 1970*. [Online]. October 1970. The fantastic combinations of John Conway's new solitaire game 'life'. Available at: https://www.ibiblio.org/lifepatterns/october1970.html (Accessed: 21 October 2019).

Gosper, R.Wm. (1984) Exploiting regularities in large cellular spaces. *Physica D: Nonlinear Phenomena*. [Online] 10 (1), 75–80. Available at: doi:10.1016/0167-2789(84)90251-3.

Gotts, N. (2010) Emergent Complexity in Conway's Game of Life. In: Andrew Adamatzky (ed.). *Game of Life Cellular Automata*. [Online]. London, Springer London. pp. 389–436. Available at: doi:10.1007/978-1-84996-217-9_20 (Accessed: 20 October 2019).

Harel, R. (2015) *rharel/node-gol-hashlife*. [Online]. Available at: https://github.com/rharel/node-gol-hashlife (Accessed: 9 August 2020).

Kachhwaha, R. and Patni, P. (2012) Ajax Enabled Web Application Model with Comet Programming. *International Journal of Engineering and Technology*. 2 (7), 7.

Letuchy, E. (2008) *Facebook Chat*. [Online]. 14 May 2008. Facebook. Available at:

https://www.facebook.com/note.php?note_id=14218138919 (Accessed: 31 October 2019).
Lubbers, P. and Greco, F. (n.d.) *HTML5 WebSocket - A Quantum Leap in Scalability for the Web*. [Online]. Available at: http://www.websocket.org/quantum.html (Accessed: 31 October 2019).
McIntosh, H.V. (2010) Conway's Life. In: Andrew Adamatzky (ed.). *Game of Life Cellular Automata*. [Online]. London, Springer London. pp. 17–33. Available at: doi:10.1007/978-1-84996-217-9_3 (Accessed: 20 October 2019).
Mesbah, A. and van Deursen, A. (2008) A component- and push-based architectural style for ajax applications. *Journal of Systems and Software*. [Online] 81 (12), 2194–2209. Available at: doi:10.1016/j.jss.2008.04.005.
Miller, J. and Hagemeister, M. (2020) *Preact*. [Online]. 23 April 2020. Available at: https://preactjs.com/ (Accessed: 9 August 2020).
Mozilla Contributors (2020) *Document Object Model (DOM)*. [Online]. 24 June 2020. MDN Web Docs. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model (Accessed: 13 August 2020).
Nesbitt, A. (2020) *Sass: Syntactically Awesome Style Sheets*. [Online]. 4 August 2020. Available at: https://sass-lang.com/ (Accessed: 8 August 2020).
O'Brien, D., Montoya, F., Menichelli, J. and Hlushko, E. (2020) *Why webpack*. [Online]. 12 May 2020. Why webpack | webpack. Available at: https://v4.webpack.js.org/concepts/why-webpack/#iifes---immediately-invoked-function-expressions (Accessed: 8 August 2020).
Puranik, D.G., Feiock, D.C. and Hill, J.H. (2013) Real-Time Monitoring using AJAX and WebSockets. In: *2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*. [Online]. April 2013 pp. 110–118. Available at: doi:10.1109/ECBS.2013.10.
Russell, A. (2006) Comet: Low Latency Data for the Browser *Infrequently Noted*. [Online]. Available at: https://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/ (Accessed: 31 October 2019).
Sapin, E. (2010) Gliders and Glider Guns Discovery in Cellular Automata. In: Andrew Adamatzky (ed.). *Game of Life Cellular Automata*. [Online]. London, Springer London. pp. 135–165. Available at: doi:10.1007/978-1-84996-217-9_9 (Accessed: 20 October 2019).
Singh, A.N. (2019) *An intro to Webpack: what it is and how to use it*. [Online]. 15 January 2019. freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/an-intro-to-webpack-what-it-is-and-how-to-use-it-8304ecdc3c60/ (Accessed: 8 August 2020).
Trevorrow, A. and Rokicki, T. (2006) *Golly Game of Life Home Page*. [Online]. 18 June 2006. Available at: http://golly.sourceforge.net/ (Accessed: 31 October 2019).
Vanderplas, J. (2013) *Conway's Game of Life in Python | Pythonic Perambulations*. [Online]. 7 August 2013. Available at: https://jakevdp.github.io/blog/2013/08/07/conways-game-of-life/ (Accessed: 31 October 2019).
Weldon, J. and Laventure, A. (2018) *Conway's Game of Life Visualization*. [Online]. 11 February 2018. Devpost. Available at: be (Accessed: 31 October 2019).
Zepeda, J.S. and Chapa, S.V. (2007) From Desktop Applications Towards Ajax Web Applications. In: *2007 4th International Conference on Electrical and Electronics Engineering*. [Online]. September 2007 pp. 193–196. Available at: doi:10.1109/ICEEE.2007.4345005.