
Building Conway's
'Game Of Life'
in React and Vue
@Paul_Hadfield

Tonight's Presentation

- About me:
  - Senior Innovation Developer at AutomationSquared
    - A digital prototype laboratory, based out of Horsham
  - Has had a love/hate relationship with Javascript since its inception

- I'll introduce:
  - Conway's "Game Of Life"
  - Some basic concepts of React and Vue
  - A step by step approach to building a working website

What is Conway's "Game Of Life"?

- The ***Game of Life***, also known simply as ***Life***, is a cellular automaton devised by the British mathematician John Horton Conway in 1970

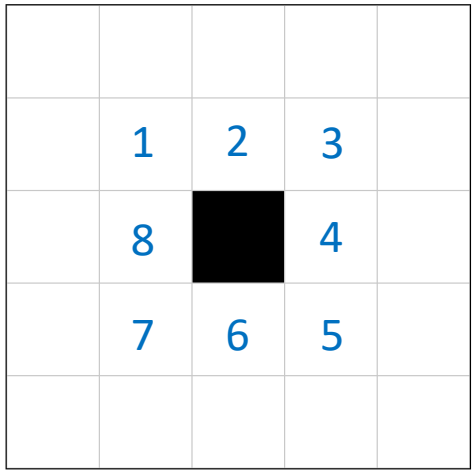
- The game is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input
  - One interacts with the Game of Life by creating an initial configuration and observing how it evolves, or, for advanced players, by creating patterns with particular properties

- Four basic rules govern all game play

Source: Wikipedia

What are the rules?

- Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent.



Source: Wikipedia

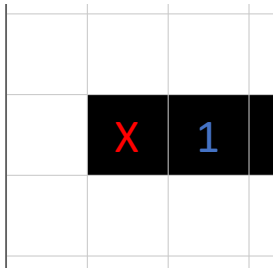
What are the rules?

- Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

Source: Wikipedia

What are the rules?

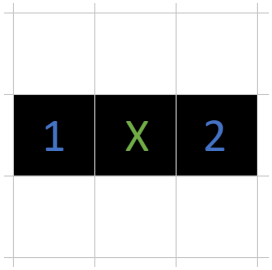
- Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:
  - Any live cell with fewer than two live neighbours dies, as if by underpopulation



Source: Wikipedia

What are the rules?

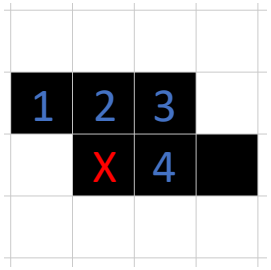
- Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:
  - Any live cell with fewer than two live neighbours dies, as if by underpopulation
  - Any live cell with two or three live neighbours lives on to the next generation



Source: Wikipedia

What are the rules?

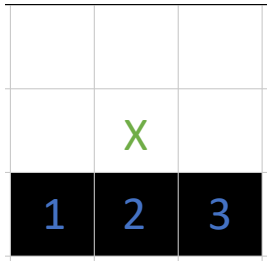
- Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:
  - Any live cell with fewer than two live neighbours dies, as if by underpopulation
  - Any live cell with two or three live neighbours lives on to the next generation
  - Any live cell with more than three live neighbours dies, as if by overpopulation



Source: Wikipedia

What are the rules?

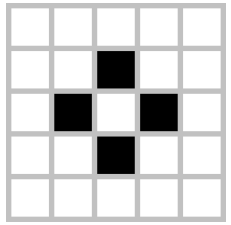
- Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:
  - Any live cell with fewer than two live neighbours dies, as if by underpopulation
  - Any live cell with two or three live neighbours lives on to the next generation
  - Any live cell with more than three live neighbours dies, as if by overpopulation
  - Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction



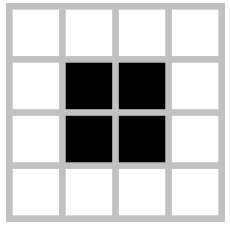
Source: Wikipedia

Examples of Patterns

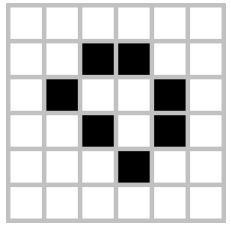
Still Lifes



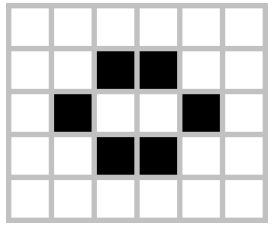
Tub



Block

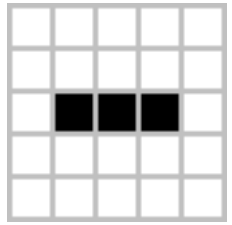


Loaf

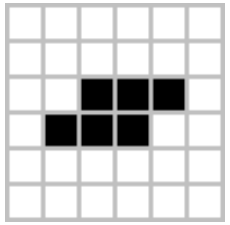


Beehive

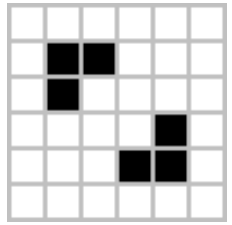
Oscillators



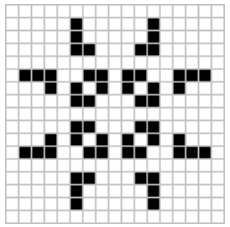
Blinker
(period 2)



Toad
(period 2)



Beacon
(period 2)



Pulsar
(period 3)

Source: [Wikipedia]

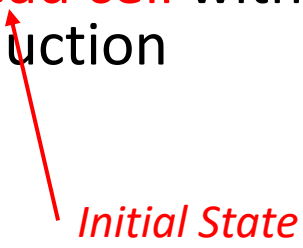
Developing the Game Engine

- Any live cell with fewer than two live neighbours dies, as if by underpopulation
- Any live cell with two or three live neighbours lives on to the next generation
- Any live cell with more than three live neighbours dies, as if by overpopulation
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction

Source: Wikipedia

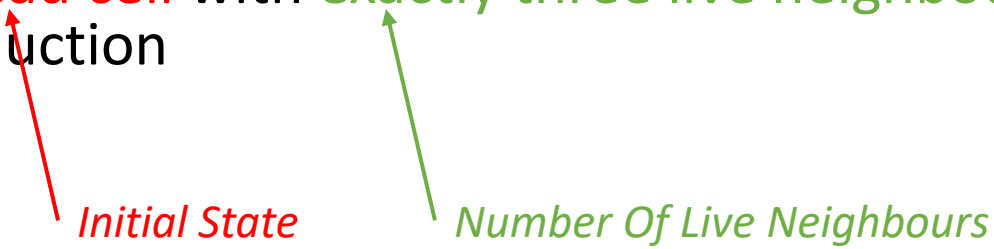
Developing the Game Engine

- Any live cell with fewer than two live neighbours dies, as if by underpopulation
- Any live cell with two or three live neighbours lives on to the next generation
- Any live cell with more than three live neighbours dies, as if by overpopulation
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction



Developing the Game Engine

- Any live cell with fewer than two live neighbours dies, as if by underpopulation
- Any live cell with two or three live neighbours lives on to the next generation
- Any live cell with more than three live neighbours dies, as if by overpopulation
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction



Developing the Game Engine

- Any live cell with fewer than two live neighbours dies, as if by underpopulation
- Any live cell with two or three live neighbours lives on to the next generation
- Any live cell with more than three live neighbours dies, as if by overpopulation
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction

Developing the Game Engine

- Any live cell with fewer than two live neighbours dies, as if by underpopulation
- Any live cell with two or three live neighbours lives on to the next generation
- Any live cell with more than three live neighbours dies, as if by overpopulation
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction

```
newState = applyRules(currentState, liveNeighbours)
```

```
const applyRules = (isLive, liveNeighbours) => {
  if (isLive) {
    return (liveNeighbours == 2) | (liveNeighbours == 3);
  } else {
    return liveNeighbours == 3;
  }
};
```

JavaScript Concepts

- Arrow Functions
  - *An arrow function expression is a syntactically compact alternative to a regular function expression, although **without its own bindings to the this, arguments, super, or new.target keywords**. Arrow function expressions are ill suited as methods, and they cannot be used as constructors.*

```
const applyRules = (isLive, liveNeighbours) => {
  if (isLive) {
    return (liveNeighbours == 2) | (liveNeighbours == 3);
  } else {
    return liveNeighbours == 3;
  }
};
```

Source: Mozilla

JavaScript Concepts

- Array.Filter()
  - *The filter() method* ***creates a new array with all elements that pass the test*** *implemented by the provided function.*
  - *Test are often written as arrow functions*

Source: [Mozilla]

Determining Live Neighbours

```
const getNumberOfLiveNeighbours = (cell, currentGrid) => {
  return currentGrid
    .filter(nc => includeNeighbours(nc, cell))
    .filter(nc => excludeSelf(nc, cell))
    .filter(isLive).length;
};
```

Determining Live Neighbours

```
const getNumberOfLiveNeighbours = (cell, currentGrid) => {
  return currentGrid
    .filter(nc => includeNeighbours(nc, cell))
    .filter(nc => excludeSelf(nc, cell))
    .filter(isLive).length;
};
```

```
const excludeSelf = (cell, currentCell) => {
  return !(cell.x == currentCell.x && cell.y == currentCell.y);
};

const isLive = cell => {
  return cell.live;
};

const includeNeighbours = (cell, currentCell) => {
  return (
    (cell.x >= currentCell.x - 1) & (cell.x <= currentCell.x + 1) &&
    (cell.y >= currentCell.y - 1) & (cell.y <= currentCell.y + 1)
  );
};
```

Templates for Starting Projects

- Options:
  - Use the CLI *(and wait whilst it downloads half the internet as packages)*
    - "`vue create gof-vue`" and select desired inputs
      - https://cli.vuejs.org/guide/creating-a-project.html#vue-create
    - "`npm init react-app gof-react`"
      - https://github.com/facebook/create-react-app
  - Use a prebuilt project template and change settings
    - For more control/include particular libraries/enforce company standards

Architecture: Common Approach

- Four Components:
  - App
    - Top level component, responsible for calling Grid and Options Components
    - Interacts with “Game Of Life” engine and passes data to child component
  - Grid
    - Draws the grid and loops through each cell, passing to the Cell component
  - Cell
    - Draws the cell
  - Options
    - Handles displaying FORM for user interaction (introduced in final demo)

Architecture: Common Approach

- Use minimal feature set to implement
  - Use ES6 features (arrow functions)
  - Keep things as simple as possible
  - Try not to use Redux and/or Vuex

React Approach

- Functional Components
  - New way to create React components
  - ‘React Hooks’ give comparable feature set to class life cycle
  - ES6 feature, requires Babel to compile

```
// CLASS COMPONENT
class ReactHeader extends React.Component {
  render() {
    return <p>{this.props.greeting} World!</p>;
  }
}

// FUNCTIONAL COMPONENT
function ReactHeader(props) {
  return <p>{props.greeting} World!</p>;
}
```

Vue Approach

- Single File Components
  - Template
  - Component
  - Style

```
<template>
  <p>{{ greeting }} World!</p>
</template>

<script>
module.exports = {
  data: function () {
    return {
      greeting: 'Hello'
    }
  }
}
</script>

<style scoped>
p {
  font-size: 2em;
  text-align: center;
}
</style>
```

Implementation Steps

1. Display the “Game Engine” initial state
2. Display the initial state in a dynamically sized grid
3. Apply the “Game Engine” rules, displaying the evolving game state
4. Add the UI to change the initial state, and restart

Displaying Initial State

- Create the default projects
  - Deleting the default components / assets
- Create the required components
  - Grid and Cell
- Add a copy of the “Game Engine” into project
- Integrate the “Game Engine” into the App component
  - Passing data to the Grid component

Demo 1 – Display Initial State

Demo Time

Implementation Steps

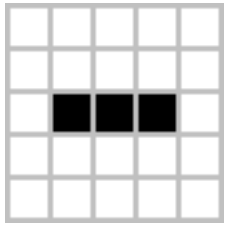
- Display the “Game Engine” initial state 
- Display the initial state in a grid
- Apply the “Game Engine” rules, displaying the evolving game state
- Add the UI to change the initial state, and restart

Implementation Steps

- Display the “Game Engine” initial state ✓
- Display the initial state in a grid
- Apply the “Game Engine” rules, displaying the evolving game state
- Add the UI to change the initial state, and restart

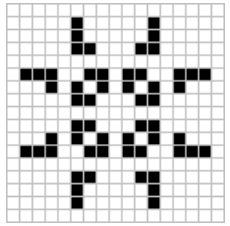
Styling Components

- Apply ‘CSS Grid’ to existing Grid/Cell HTML Div’s
  - Great Resource: https://gridbyexample.com/examples/
- Dynamically calculate grid dimensions
  - “Game Of Life” examples have different number of rows / columns



5x5

vs



17x17

Demo 2 – CSS Grid

Demo Time

Implementation Steps

- Display the “Game Engine” initial state
- Display the initial state in a grid
- Apply the “Game Engine” rules, displaying the evolving game state
- Add the UI to change the initial state, and restart

Implementation Steps

- Display the “Game Engine” initial state ✓
- Display the initial state in a grid ✓
- Apply the “Game Engine” rules, displaying the evolving game state
- Add the UI to change the initial state, and restart

State Change 101

1. Create an instance of a component
2. Populate the required data and add to local state
3. Bind the state tracked data to the template
4. Render the template
5. Respond to an event, updating the data held in state
6. React/Vue re-renders the template, with the updated data
   - *Using a shadow DOM and tracked ‘keys’ to reduce changes*

React: Updating State

- React Hooks
  - Introduced in React 16.8
  - Uses the naming convention “use….”
  - Functional Components are now as powerful as Class Components

Source: React Documentation

React: Updating State

- “useState” React Hook
  - Provide an initial state
  - Returns instance of value and function to set new value (state)

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Source: [React Documentation]

React: Updating State

- “useEffect” React Hook
  - Use for ‘Side Effects’ (i.e. HTTP Calls, scheduling calls)
  - Similar to class component life cycle events “componentDidMount”, “componentDidUpdate” and “componentWillUnmount” combined

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Source: [React Documentation]

Demo 3 – React: Updating State

Demo Time

Vue: Updating State

- Uses life cycle events
  - Created
  - beforeDestroy

```js
new Vue({
  data: {
    a: 1
  },
  created: function () {
    // `this` points to the vm instance
    console.log('a is: ' + this.a)
  }
})
// => "a is: 1"
```

Source: Vue Documentation

Demo 4 – Vue: Updating State

Demo Time

Implementation Steps

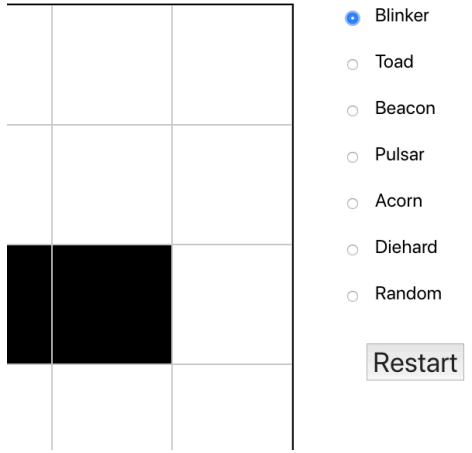
- Display the “Game Engine” initial state ✓
- Display the initial state in a grid ✓
- Apply the “Game Engine” rules, displaying the evolving game state ✓
- Add the UI to change the initial state, and restart

Implementation Steps

- Display the “Game Engine” initial state ✓
- Display the initial state in a grid ✓
- Apply the “Game Engine” rules, displaying the evolving game state ✓
- Add the UI to change the initial state, and restart

FORM data / User Interaction

- Add new component:
  - Options
- Inter-component Communication
  - Parent to sibling or sibling to parent
    - Local state
  - Sibling to sibling
    - Redux or Vuex



Design skills optional

FORM data / User Interaction

- React
  - Functions passed in from parent as props

```
class Foo extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

Arrow Function in Render

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={() => this.handleClick()}>Click Me</button>;
  }
}
```

Note:
Using an arrow function in render creates a new function each time the component renders, which may break optimizations based on strict identity comparison.

Source: [React Documentation]

Demo 5 – React: Select “Initial State” UI

Demo Time

FORM data / User Interaction

- Vue:
  - Initial state passed in as props
  - Events emitted to parent

```
export default {
  methods: {
    onClickButton (event) {
      this.$emit('clicked', 'someValue')
    }
  }
}
```

Parent component receive `clicked` event:

```
<div>
  <child @clicked="onClickChild"></child>
</div>
```

```
export default {
  methods: {
    onClickChild (value) {
      console.log(value) // someValue
    }
  }
}
```

Source: [Vue Forums]

Demo 6 – Vue: Select “Initial State” UI

Demo Time

Implementation Steps

- Display the “Game Engine” initial state
- Display the initial state in a grid
- Apply the “Game Engine” rules, displaying the evolving game state
- Add the UI to change the initial state, and restart

Conclusion

- Vue and React are very similar
  - Both were easy to pick up
- React Event hooks work really well
- Feels easier passing functions into react
  - Compared to Vue emitting events
- I think I actually prefer React (a surprise)
  - I still have a love/hate relationship with Javascript

Thank you – Any questions?

- GitHub: Slides and source code can be found here
  - https://github.com/Paul-Hadfield/GameOfLife-Talk

- Contact me on Twitter: @Paul_Hadfield