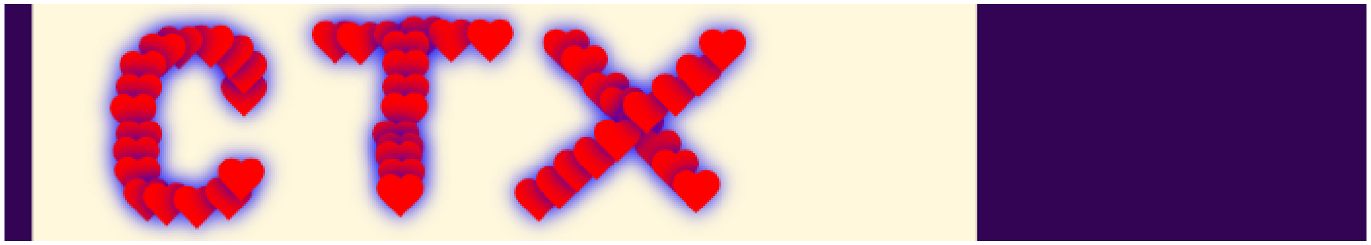
React: Creating an Interactive Canvas Component

Martin Crabtree Follow
Feb 28 · 3 min read



The HTML5 canvas element behaves differently than standard HTML elements, especially when embedded in a React.js application. In this guide, we will take a look at how to set up a basic React.js application that incorporates an interactive HTML5 canvas element, while also compartmentalizing the logic using a custom useCanvas() hook.

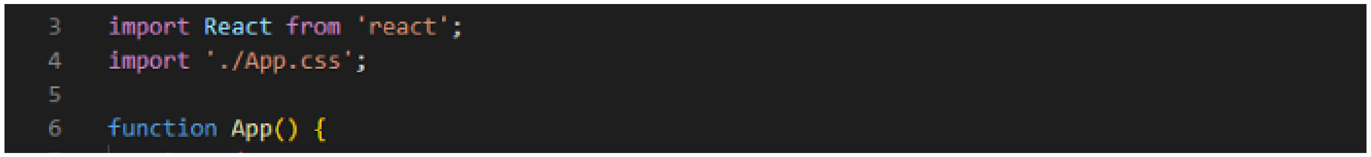
The Context Element of HTML5 Canvas:

Have you ever wondered why you can't just insert a canvas tag into some HTML code and pass it properties just like most other HTML elements? The reason is because the canvas element is somewhat unique, so when it is invoked, the browser creates a JavaScript object — known as context — that can interact with the HTML5 Canvas API. A context is unique to its parent element, and can be accessed by using the useContext() method.

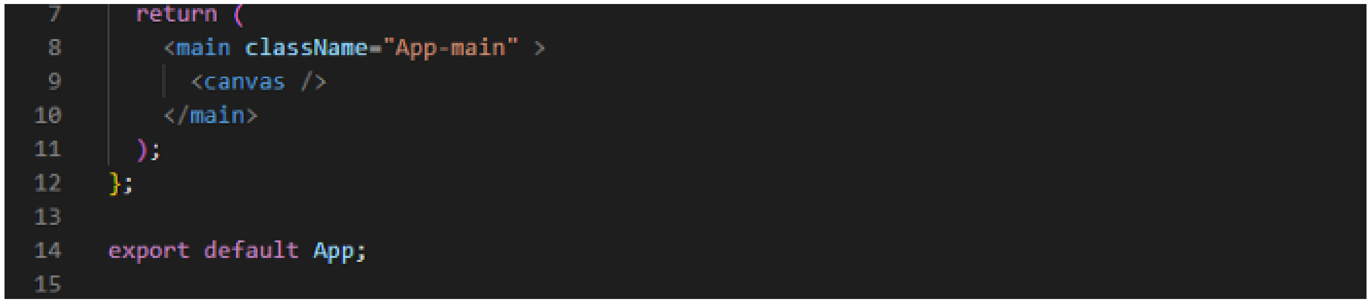
When invoking the useContext() method, you have to pass an argument that specifies if it will be used for 2d or 3d (webgl) rendering. The 3d context is used for cases that require hardware accelerated graphics. While, the standard 2d context — which is used for most animations, data visualizations, and image manipulations — will be used in this demo.

Giving Canvas a Home:

For the purposes of this demo we will be rendering everything in the main App.js component, although the canvas element can be housed in any standard React.js component. The application will render the canvas element, and will draw a heart at the coordinates specified by an onClick() event listener.



```
import React from 'react';
import './App.css';

function App() {
```



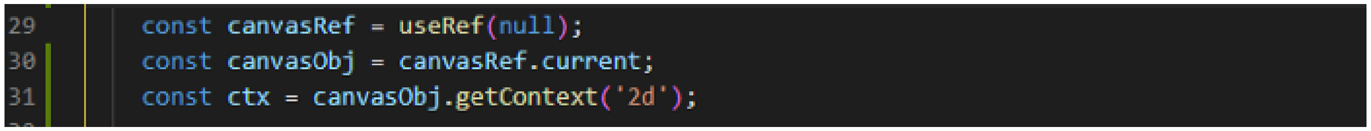
```
  return (
    <main className="App-main" >
      <canvas />
    </main>
  );
};

export default App;
```

Creating a useCanvas() Hook:

React.js 16.8.0 introduced hooks which allows us to develop reusable functions that interact with state and lifecycle features within a function component, and also without relying on a class. In our case we will be creating a custom hook, that will also utilize some common hooks that are now offered natively in React.js (useState, useEffect, useRef).

The useCanvas() hook needs to: reference the context JavaScript object created by the canvas element; use the context reference to update properties in the Canvas API and then render the updated canvas; and update the context — in this case on mouse events.

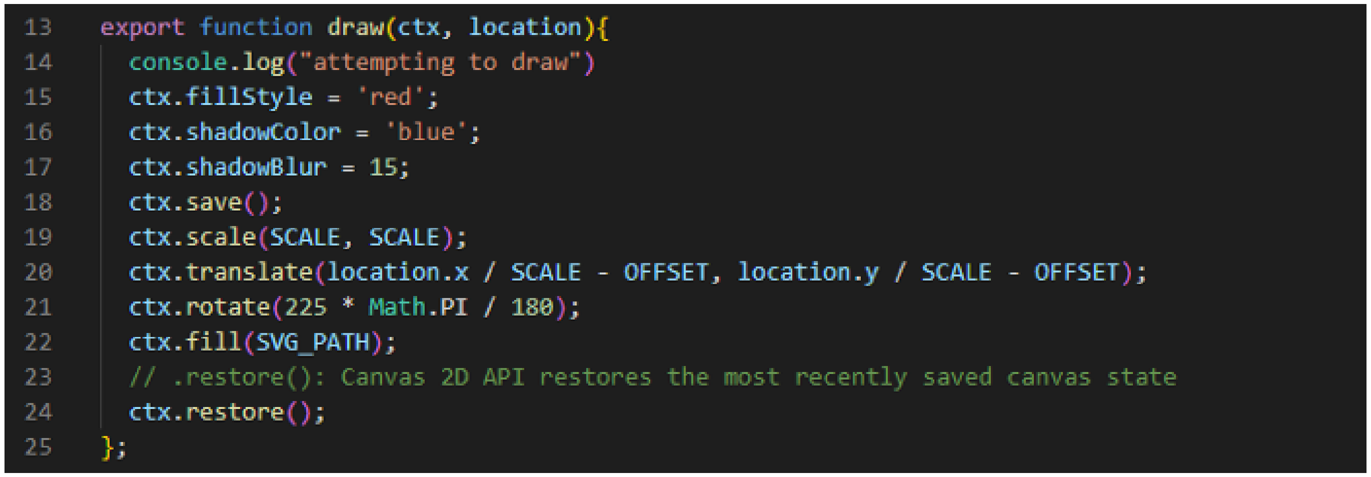
Referencing Context 2D:



```
    const canvasRef = useRef(null);
    const canvasObj = canvasRef.current;
    const ctx = canvasObj.getContext('2d');
```

A reference to the canvas context(2d) object is assigned to the **ctx** variable that allows us to interact with properties in the HTML5 Canvas API.

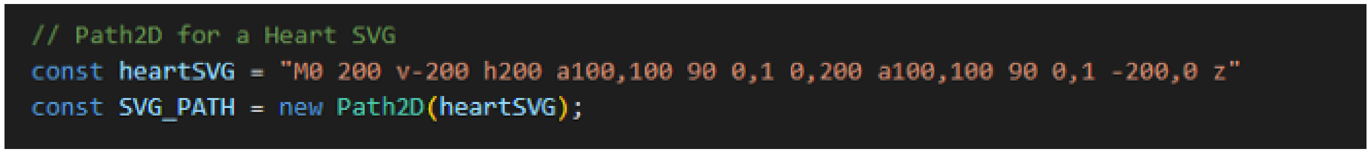
Updating Canvas API & Rendering Canvas:



```
export function draw(ctx, location){
  console.log("attempting to draw")
  ctx.fillStyle = 'red';
  ctx.shadowColor = 'blue';
  ctx.shadowBlur = 15;
  ctx.save();
  ctx.scale(SCALE, SCALE);
  ctx.translate(location.x / SCALE - OFFSET, location.y / SCALE - OFFSET);
  ctx.rotate(225 * Math.PI / 180);
  ctx.fill(SVG_PATH);
  // .restore(): Canvas 2D API restores the most recently saved canvas state
  ctx.restore();
};
```

Using the ctx reference variable, we are able to set properties needed to render our canvas background and heart-shaped SVG in this specific instance of the canvas context.

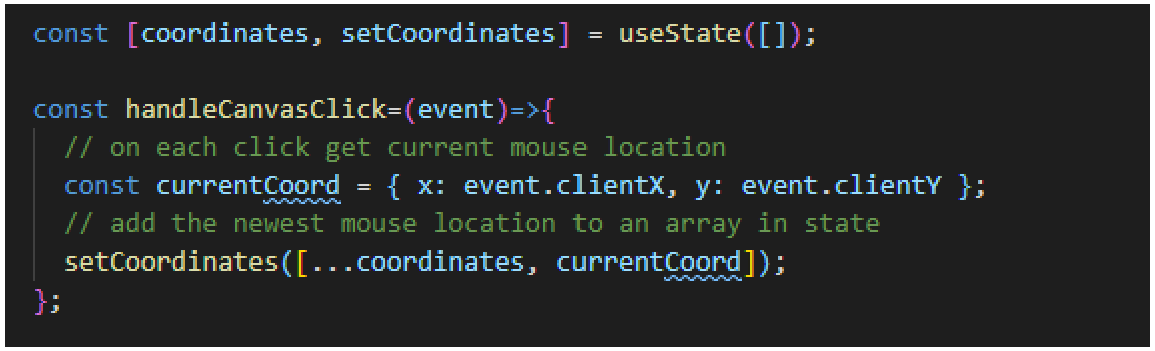
What to Draw:



```
// Path2D for a Heart SVG
const heartSVG = "M0 200 v-200 h200 a100,100 90 0,1 0,200 a100,100 90 0,1 -200,0 z"
const SVG_PATH = new Path2D(heartSVG);
```

A path to our heart… SVG. Which is then saved to a SVG_PATH variable.

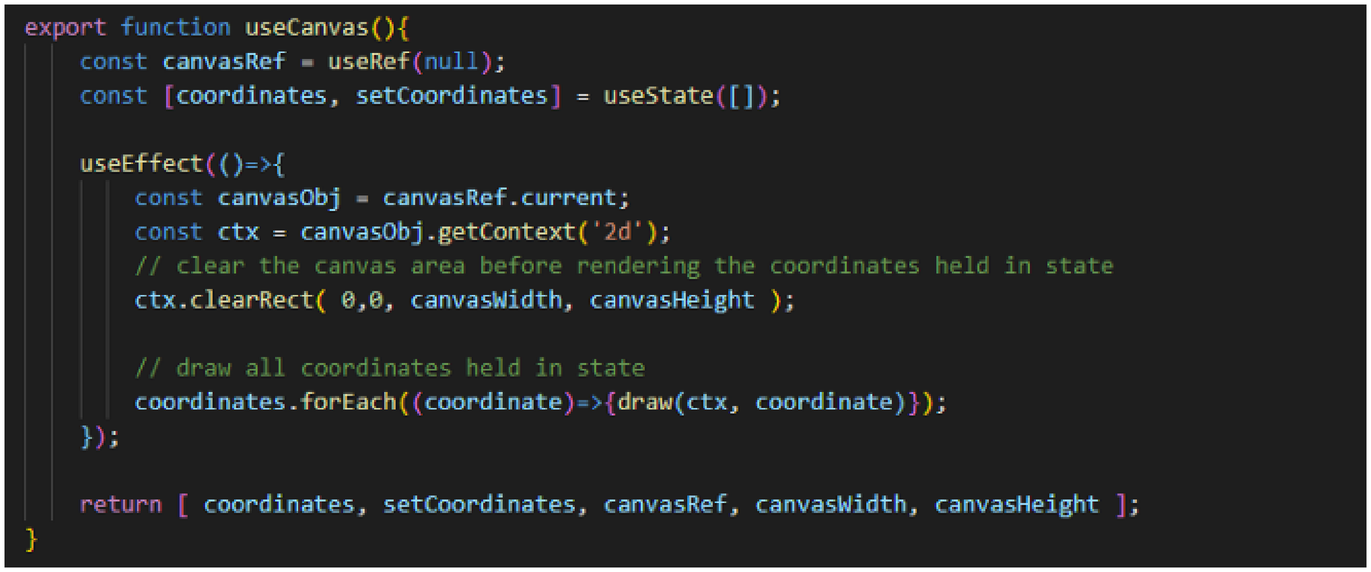
Stateful Event Handling:



```
const [coordinates, setCoordinates] = useState([]);

const handleCanvasClick=(event)=>{
  // on each click get current mouse location
  const currentCoord = { x: event.clientX, y: event.clientY };
  // add the newest mouse location to an array in state
  setCoordinates([...coordinates, currentCoord]);
};
```

Utilizing the useState() hook to create a coordinates array in React.js state, that will add new coordinate pairs to the array with the setCoordinates() update function.

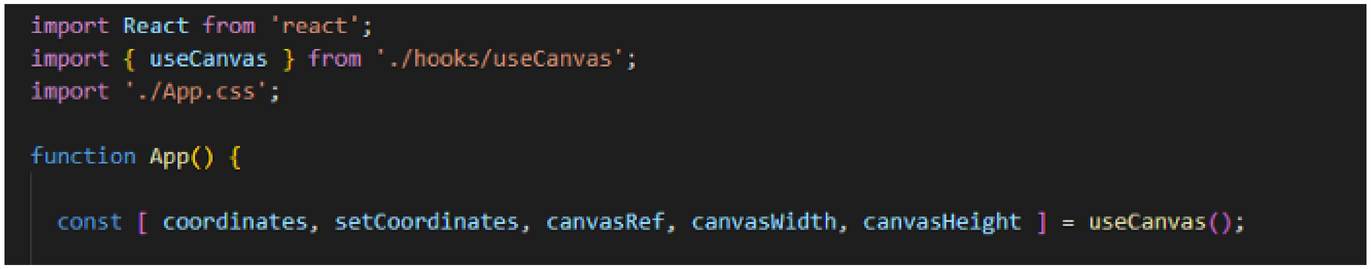
The useCanvas() Function:



```
export function useCanvas(){
    const canvasRef = useRef(null);
    const [coordinates, setCoordinates] = useState([]);

    useEffect(()=>{
        const canvasObj = canvasRef.current;
        const ctx = canvasObj.getContext('2d');
        // clear the canvas area before rendering the coordinates held in state
        ctx.clearRect( 0,0, canvasWidth, canvasHeight );

        // draw all coordinates held in state
        coordinates.forEach((coordinate)=>{draw(ctx, coordinate)});
    });

    return [ coordinates, setCoordinates, canvasRef, canvasWidth, canvasHeight ];
}
```

The useCanvas() hook is ready for export! The useEffect() hook is used because we only want the reference to the canvas context to be initialized after the first rerender. Also we only want to trigger a rerender if the value of the coordinates array changes.

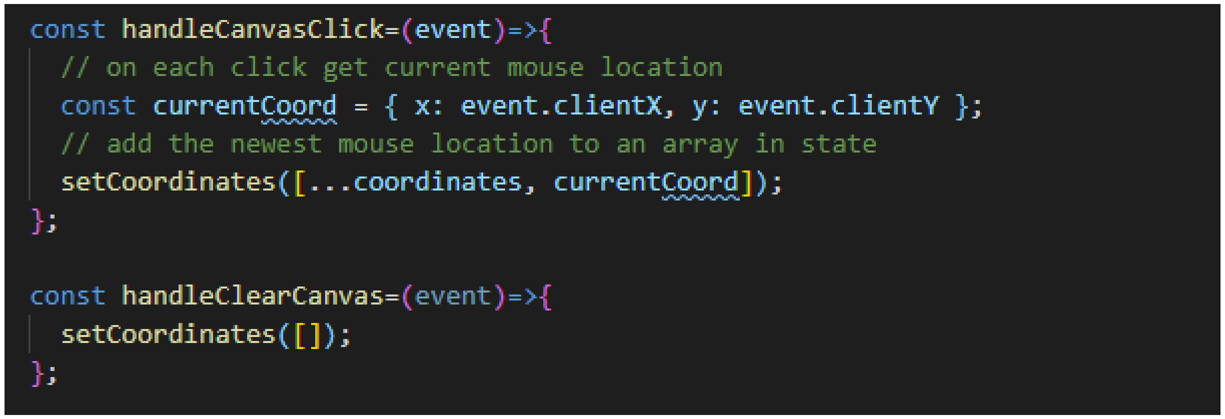
Importing and Using the useCanvas() Hook:



```
import React from 'react';
import { useCanvas } from './hooks/useCanvas';
import './App.css';

function App() {

  const [ coordinates, setCoordinates, canvasRef, canvasWidth, canvasHeight ] = useCanvas();
```

Adding the useCanvas() hook to our React.js application.

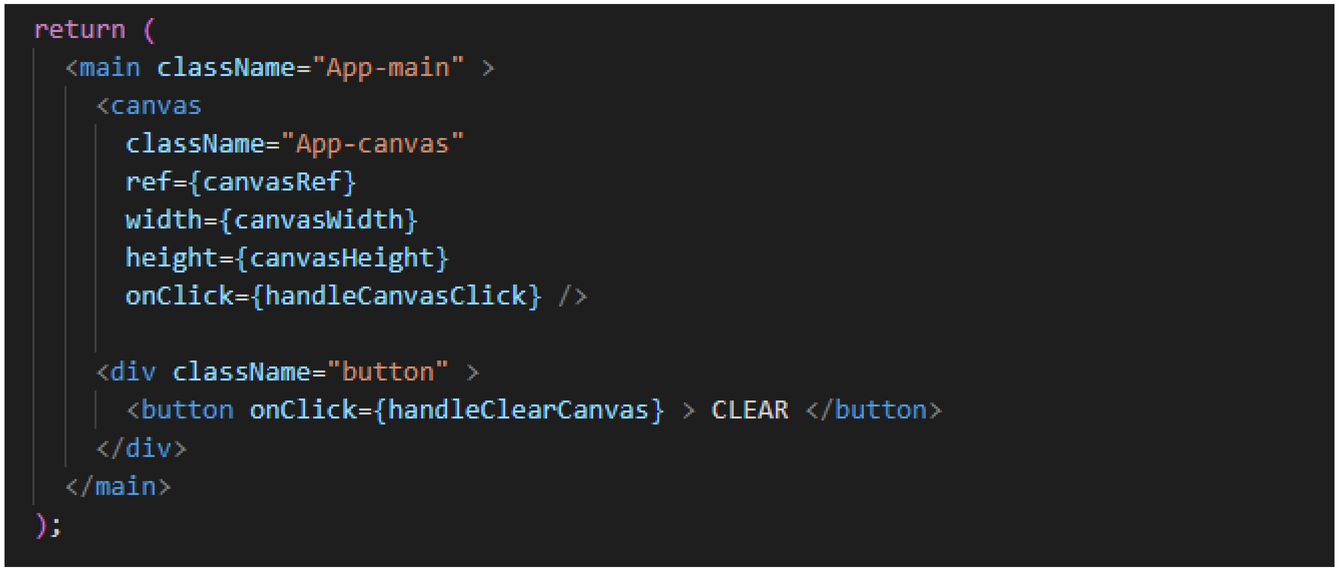
Event Handlers:



```
const handleCanvasClick=(event)=>{
  // on each click get current mouse location
  const currentCoord = { x: event.clientX, y: event.clientY };
  // add the newest mouse location to an array in state
  setCoordinates([...coordinates, currentCoord]);
};

const handleClearCanvas=(event)=>{
  setCoordinates([]);
};
```

We have two event handlers: handleCanvasClick() will trigger with an onClick() event handler, and handleClearCanvas() will be tied to a button that will wipe the canvas clean by overwriting the coordinates array.

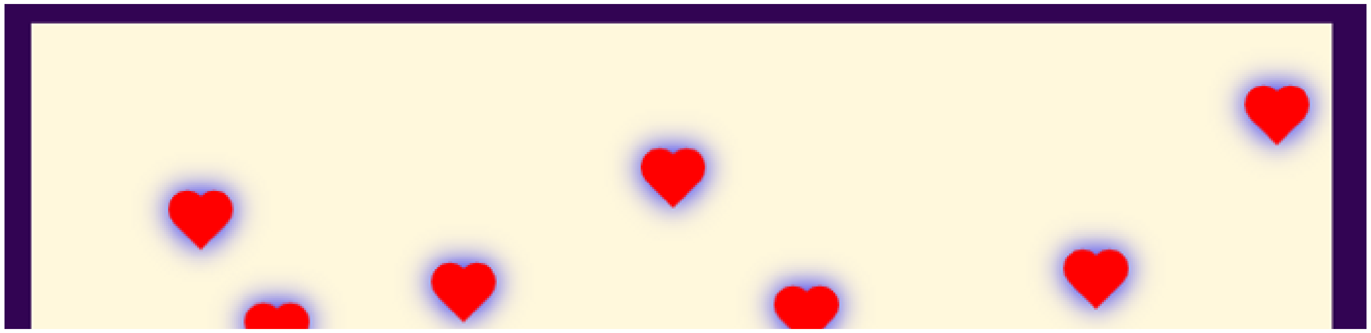
Back to the App.js return():



```
return (
  <main className="App-main" >
    <canvas
      className="App-canvas"
      ref={canvasRef}
      width={canvasWidth}
      height={canvasHeight}
      onClick={handleCanvasClick} />

    <div className="button" >
      <button onClick={handleClearCanvas} > CLEAR </button>
    </div>
  </main>
);
```

Adding some styling, event handlers, and properties derived from the useCanvas() custom hook.

The Result:



It's a bit late for Valentine's Day, but I guess it's the thought that counts?

To see and/or run the whole project, check out my react-canvas-demo repository on GitHub.

References:

https://itnext.io/using-react-hooks-with-canvas-f188d6e416c0

https://www.folio3.com/mobile/blog/how-to-make-use-of-html-5-canvas-with-react/

https://www.html5canvastutorials.com/tutorials/html5-canvas-element/