Canvas with React.js

Lucas Miranda Follow
Mar 21 · 8 min read

Photo by Logan Weaver on Unsplash

In this article, we will see how to create a Canvas React component and a custom hook for extracting its logic, so we can just draw inside it like we usually draw in a regular canvas html element.

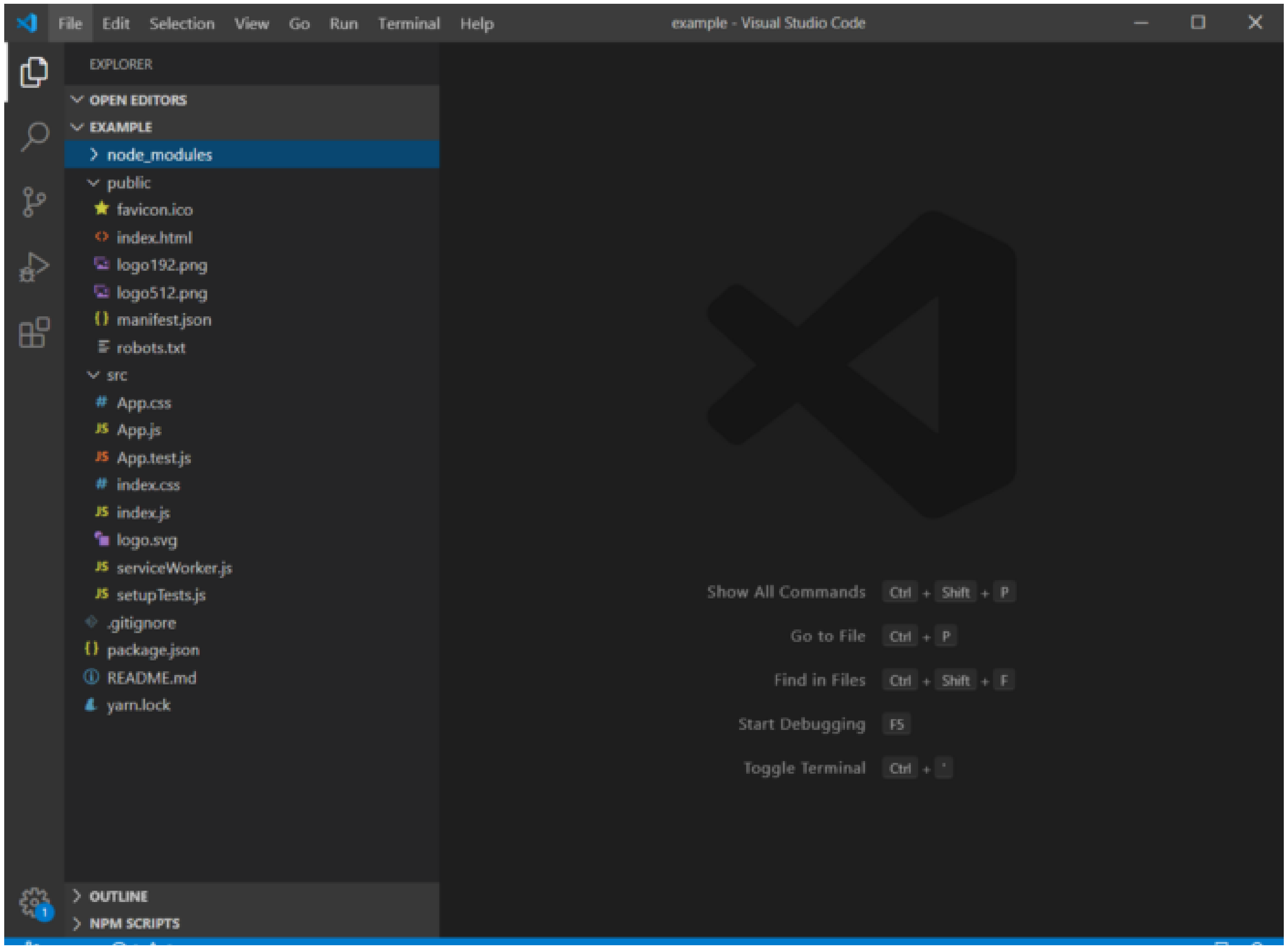
This article is based on Corey's article "Animating a Canvas with React Hooks". Any other sources and related contents are linked throughout this article.

I am assuming that you already know canvas, but if you don't know yet, I recommend this tutorial from MDN to you.

. . .

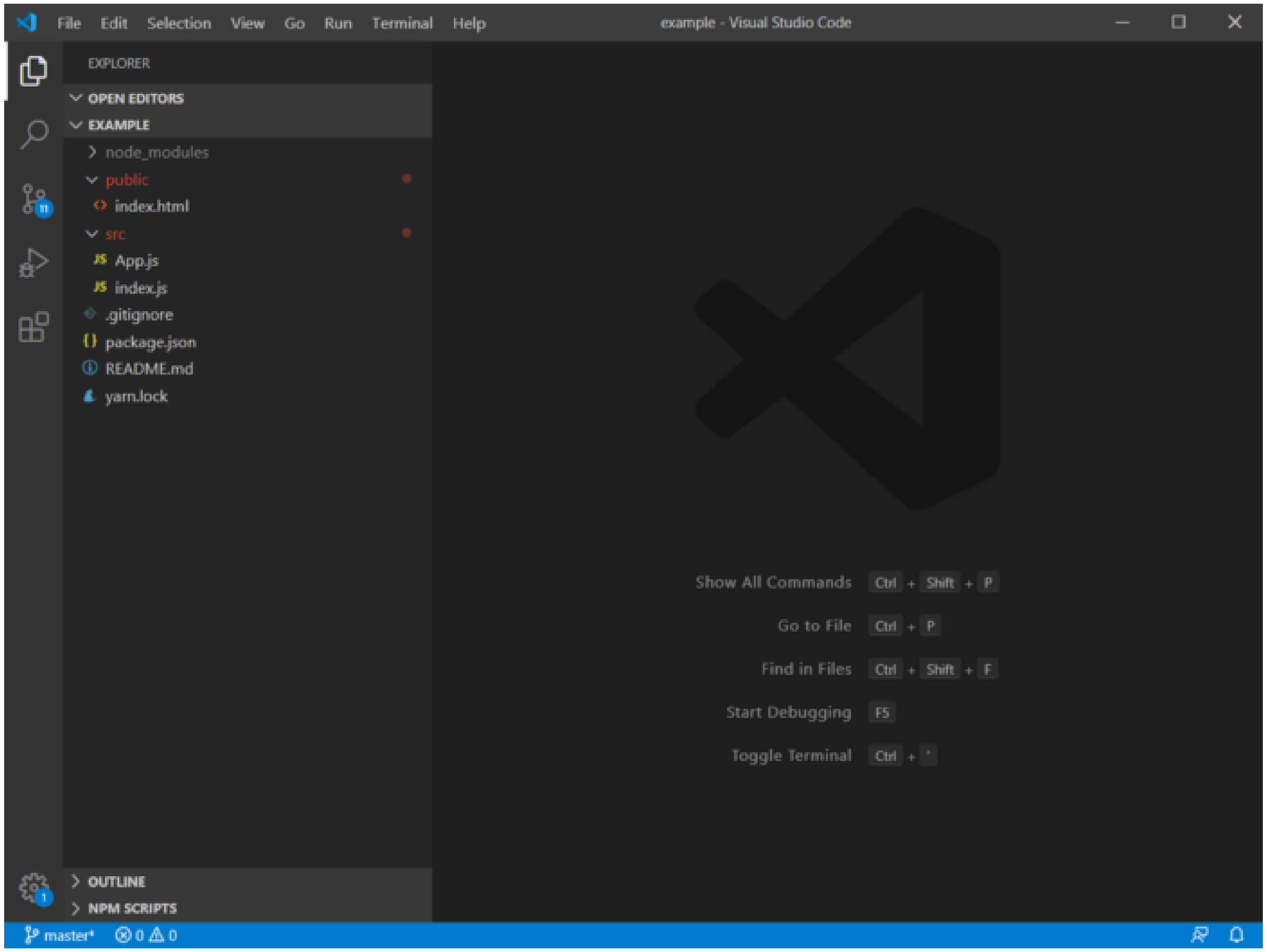
Starting a new project

In order to see what we are doing, let's create a new react app with create-react-app (feel free to skip this step if you are already familiar with React and create-react-app). You can start a new project by running `npx create-react-app example` or `yarn create react-app example` if you prefer yarn. If you open the project folder (example) in your code editor, you must get something like this:



The react app boilerplate

We don’t need all these files, so we will delete almost everything from src folder and public folder. We just need the index.html, App.js and index.js like you can see there:



Keep only the essential in the project boilerplate.

We will also keep only the essential in the index.js script and index.html markup file.

```jsx
import React from 'react'
import Canvas from './Canvas'

function App() {

  return <h1>Hello world!</h1>
}

export default App
```

App.jsx hosted with ❤ by GitHub view raw

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
```

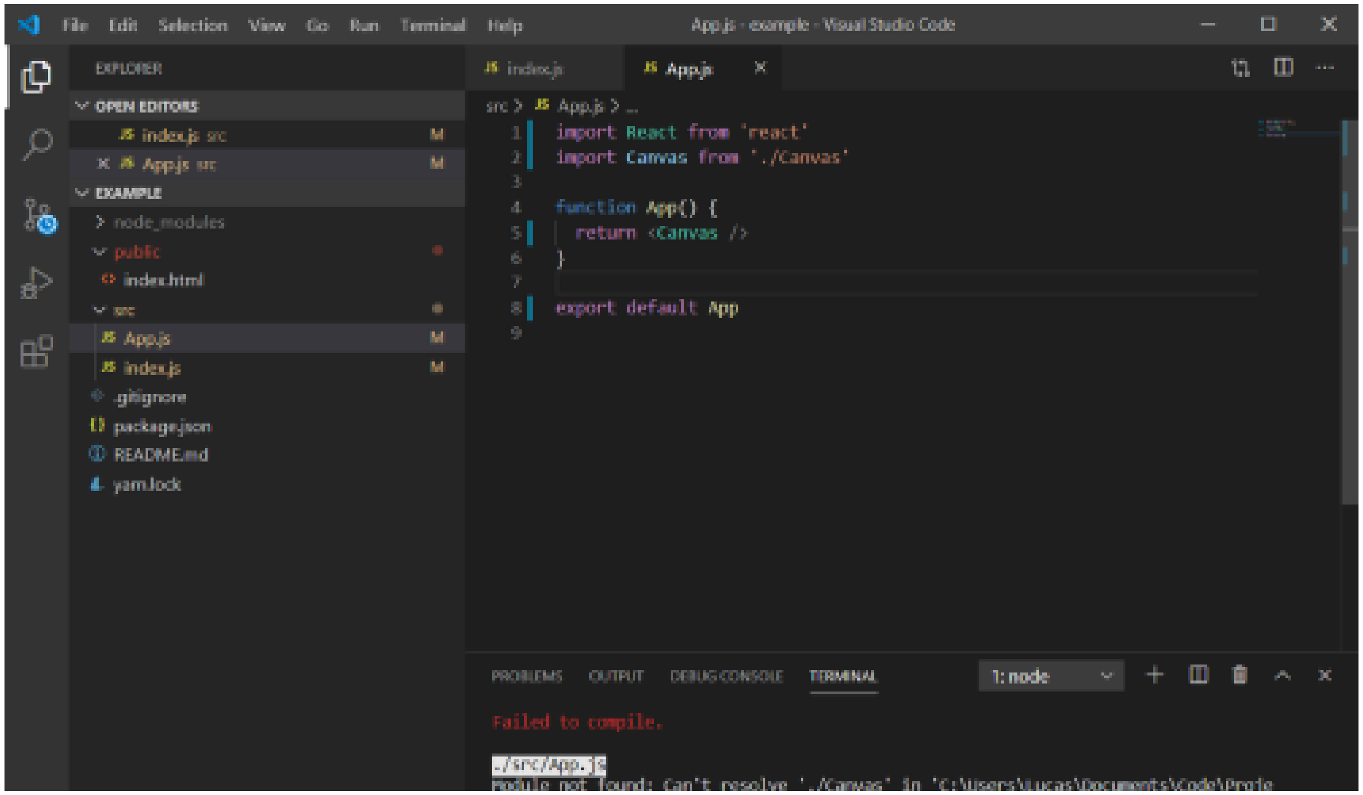
```
</html>
```

index.html hosted with ❤ by GitHub view raw

Minimal app structure for testing the Canvas component that is being built

If we run `yarn start` or `npm start` we can see the “hello world” in the page. So, it is working!

Now that we see it is all working, we can replace the “hello world” with the Canvas component that we will create.



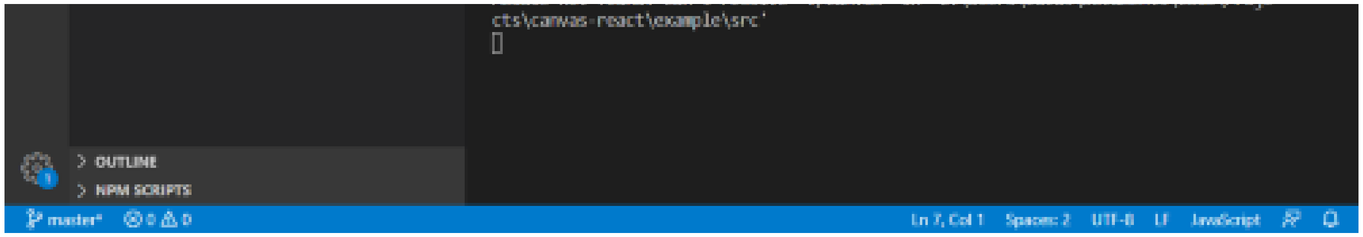
```
import React from 'react'
import Canvas from './Canvas'

function App() {
  return <Canvas />
}

export default App
```

```
Failed to compile.

./src/App.js
Module not found: Can't resolve './Canvas' in 'C:\Users\Lucas\Documents\Code\Proje
```



```
cts\canvas-react\example\src'
```

Ready to start creating the Canvas component

Our app crash, but it will get fixed when we create the Canvas component.

. . .

The Canvas Component

We need a canvas element to draw inside it, so we must create a component that is basically a canvas element:

```jsx
import React from 'react'

const Canvas = props => <canvas {...props}/>

export default Canvas
```

CanvasPart1.jsx hosted with ❤ by GitHub view raw

Now we have a canvas element wrapped in a react component called Canvas. Great! However, how can we draw in it? Well… we will need to get the DOM canvas element itself to get its context object, right? The React way to get a dom element is by giving it a **ref** prop.

Getting Canvas Context

To get the canvas element, we will create a ref and give it to the canvas element:

```jsx
import React, { useRef } from 'react'

const Canvas = props => {

  const canvasRef = useRef(null)

  return <canvas ref={canvasRef} {...props}/>
}

export default Canvas
```

CanvasPart2.jsx hosted with ❤ by GitHub view raw

We can access the canvas element through the canvasRef now. Now we just need to get the context and start drawing! 🙌

```jsx
import React, { useRef } from 'react'
import React, { useRef } from 'react'

const Canvas = props => {

  const canvasRef = useRef(null)
  const canvas = canvasRef.current
  const context = canvas.getContext('2d')

  return <canvas ref={canvasRef} {...props}/>
}

export default Canvas
```

CanvasPart3Broken.js hosted with ❤ by GitHub view raw

What? Is it broken? Cannot read property getContext of null? getContext is not a function? 😓

The component is not mounted yet when we tried to get the canvas through the ref, so its value is, naturally, the initial value that we gave for it (which is null in my case). We must wait the component did mount properly before get the real canvas. Fortunately, there is a hook to handle that problem!

The useEffect hook allow us to perform side effects in function components. It means that we can call functions right after the component did mount, component update or change of some variable, and some other stuff. (Learn more about useEffect hook here and life cycle of components here)

We are interested in the first case right now: the component did mount. Right after the canvas element is available in the dom for us, we want to get it on javascript to take its context and make some draw. To do that, we pass a function to be executed as the first argument of the useEffect, and an empty array as the second. The empty array says to useEffect that we want execute that function only once, after the component did mount (we will discuss more about this array later). If we pass only the first argument (the function), useEffect will call the function after every single update of the component.

```
import React, { useRef, useEffect } from 'react'

const Canvas = props => {

  const canvasRef = useRef(null)

  useEffect(() => {
    const canvas = canvasRef.current
    const context = canvas.getContext('2d')
    //Our first draw
    context.fillStyle = '#000000'
    context.fillRect(0, 0, context.canvas.width, context.canvas.height)
  }, [])

  return <canvas ref={canvasRef} {...props}/>
}

export default Canvas
```

CanvasPart3.jsx hosted with ❤ by GitHub view raw

Now we are able to draw in the canvas! 🎉🎊🎉🎊🎉

. . .

The Draw

We have a black rectangle in our canvas right now, but we don't wanna draw a black rectangle every single time we use this component, right? Then, we can take a callback function that make our draw, so our component can be more dynamic. Let's do it!

```
import React, { useRef, useEffect } from 'react'

const Canvas = props => {

  const canvasRef = useRef(null)

  const draw = ctx => {
    ctx.fillStyle = '#000000'
    ctx.beginPath()
    ctx.arc(50, 100, 20, 0, 2*Math.PI)
    ctx.fill()
  }

  useEffect(() => {

    const canvas = canvasRef.current
    const context = canvas.getContext('2d')

    //Our draw come here
    draw(context)
  }, [draw])

  return <canvas ref={canvasRef} {...props}/>
}

export default Canvas
```

CanvasStep4.jsx hosted with ❤ by GitHub view raw

You may notice that now we have changed the array as the second argument of the useEffect, right? Now it is no longer empty. We have put the draw function inside it. Do you remember that I said that useEffect could call functions after a change of some variable? This is the case. That array is known as the dependencies array, and everything we put in it, is watched by the useEffect. When anything that is inside of the dependencies array changes, the function will be called again with its updated values. Thus, every time we change the draw, the function of the useEffect will be called again for the new draw.

We have a draw function instead of a bunch of code spread in our useEffect callback now. Nevertheless, we still have a screen with a static draw on it. How about adding some animation to it?

```
import React, { useRef, useEffect } from 'react'

const Canvas = props => {

  const canvasRef = useRef(null)

  const draw = (ctx, frameCount) => {
    ctx.clearRect(0, 0, ctx.canvas.width, ctx.canvas.height)
    ctx.fillStyle = '#000000'
    ctx.beginPath()
    ctx.arc(50, 100, 20*Math.sin(frameCount*0.05)**2, 0, 2*Math.PI)
    ctx.fill()
  }

  useEffect(() => {

    const canvas = canvasRef.current
    const context = canvas.getContext('2d')
    let frameCount = 0
    let animationFrameId

    //Our draw came here
    const render = () => {
      frameCount++
      draw(context, frameCount)
      animationFrameId = window.requestAnimationFrame(render)
    }
    render()

    return () => {
      window.cancelAnimationFrame(animationFrameId)
    }
  }, [draw])

  return <canvas ref={canvasRef} {...props}/>
}

export default Canvas
```

CanvasStep5.jsx hosted with ❤ by GitHub view raw

Now we have a simple animation!

Animation running in the Canvas component

Let me explain what happened here:

- **Function render:** All the steps that will be repeated in the animation were wrapped in a function called render which will be called recursively by the requestAnimationFrame method.

- **The frameCount variable:** This is a control variable that counts frames. If you prefer, you can use a counter for time instead. The goal of this variable is provide a clock to our draw function since the animation is time dependent.
- **Draw function:** The draw function now takes the frame counter as argument and the radius of the circle changes over time. We also clear the canvas with clearRect function, otherwise it would draw over the previous draw every iteration.
- **Cancel animation frame:** The function returned in the useEffect callback (aka clean up function) is called right before the component unmount. That way we can ensure that our animation frame is cancelled after our canvas component unmount.

Looks good, don't you think? But defining the draw function inside the component doesn't sounds good… I mean, we still have the same draw for every canvas! 🤔

I can see you yelling to me: "TAKE THE DRAW CALLBACK FROM PROPS!!"

Keep calm young one, that is exactly what we're gonna do right now 😎

```
import React, { useRef, useEffect } from 'react'
```

```jsx

const Canvas = props => {

  const { draw, ...rest } = props
  const canvasRef = useRef(null)

  useEffect(() => {

    const canvas = canvasRef.current
    const context = canvas.getContext('2d')
    let frameCount = 0
    let animationFrameId

    const render = () => {
      frameCount++
      draw(context, frameCount)
      animationFrameId = window.requestAnimationFrame(render)
    }
    render()

    return () => {
      window.cancelAnimationFrame(animationFrameId)
    }
  }, [draw])

  return <canvas ref={canvasRef} {...rest}/>
}

export default Canvas
```

CanvasStep6.jsx hosted with ❤ by **GitHub** view raw

We should insert the draw function in the Canvas component right now:

```jsx
import React from 'react'
import Canvas from './Canvas'

function App() {

  const draw = (ctx, frameCount) => {
    ctx.clearRect(0, 0, ctx.canvas.width, ctx.canvas.height)
    ctx.fillStyle = '#000000'
    ctx.beginPath()
    ctx.arc(50, 100, 20*Math.sin(frameCount*0.05)**2, 0, 2*Math.PI)
    ctx.fill()
  }

  return <Canvas draw={draw} />
}

export default App
```

AppForCanvas.jsx hosted with ❤ by GitHub view raw

It looks great now 🧐. All we need is to use the Canvas component giving the draw function as a prop to it and we have a complete canvas animated.

But… What if we want to make a different component, with some little changes, but keeping the same logic for the canvas? To achieve that, we should create a custom hook for our canvas, then we can use the logic in our canvas, and if we want to create some different component, but keeping the logic, we can just call the hook for handle the logic for us.

. . .

The Canvas Hook

Note that the only thing we need to give to our logic is the draw callback, and the only thing we have to pass to our component is the canvasRef. Therefore, our component must be something like that in the end:

```jsx
import React from 'react'
import useCanvas from './useCanvas'

const Canvas = props => {

  const { draw, ...rest } = props
  const canvasRef = useCanvas(draw)

  return <canvas ref={canvasRef} {...rest}/>
}

export default Canvas
```

CanvasFinalStep.jsx hosted with ❤ by GitHub view raw

The draw callback is given to our hook and a ref for our canvas element is returned. Now we just have to copy our logic to the useCanvas hook and transform it in a function that receives the draw callback and return the canvasRef:

```
import { useRef, useEffect } from 'react'

const useCanvas = draw => {

  const canvasRef = useRef(null)

  useEffect(() => {

    const canvas = canvasRef.current
    const context = canvas.getContext('2d')
    let frameCount = 0
    let animationFrameId

    const render = () => {
      frameCount++
      draw(context, frameCount)
      animationFrameId = window.requestAnimationFrame(render)
    }
    render()

    return () => {
      window.cancelAnimationFrame(animationFrameId)
    }
  }, [draw])

  return canvasRef
}

export default useCanvas
```

CanvasHook.js hosted with ❤ by GitHub view raw

Now we are (almost) done!

. . .

Extra features

Setup canvas

What if you want to use another context instead of 2d in the canvas? If you have some other configurations that you want for your canvas? Then you might want make the Canvas component more flexible by giving it one more prop: options.

The options prop might be an object which contains all your custom setup for the canvas and whatever you want to your Canvas component. Your canvas hook also must have a little modification ir order to accept new arguments.

```
import React from 'react'
import useCanvas from './useCanvas'

const Canvas = props => {

  const { draw, options, ...rest } = props
  const { context, ...moreConfig } = options
  const canvasRef = useCanvas(draw, {context})

  return <canvas ref={canvasRef} {...rest}/>
}

export default Canvas
```

CanvasWithOptions.js hosted with ❤ by GitHub view raw

```
import { useRef, useEffect } from 'react'

const useCanvas = (draw, options={}) => {

  const canvasRef = useRef(null)

  useEffect(() => {

    const canvas = canvasRef.current
    const context = canvas.getContext(options.context || '2d')
    let frameCount = 0
    let animationFrameId
    const render = () => {
      frameCount++
      draw(context, frameCount)
      animationFrameId = window.requestAnimationFrame(render)
    }
    render()
    return () => {
      window.cancelAnimationFrame(animationFrameId)
    }
  }, [draw])
  return canvasRef
}
export default useCanvas
```

useCanvasWithOptions.js hosted with ❤ by GitHub view raw

Making the code falls back to the tradicional setup will provide a better usability. With that you can setup a webgl context.

Resizing function

At this moment, we don't have control over the canvas size. What happen if we want a bigger or a smaller canvas? Or even a canvas with dynamic adjustable height and width? We should have a function to handle that.

Here I will show how you could track the canvas size, but you can come with a different approach that you prefer for this function.

```
function resizeCanvasToDisplaySize(canvas) {

  const { width, height } = canvas.getBoundingClientRect()

  if (canvas.width !== width || canvas.height !== height) {
    canvas.width = width
```

```
        canvas.height = height
        return true // here you can return some usefull information like delta width and d
        // this information can be used in the next redraw...
      }

      return false
    }
```

resizeCanvas.js hosted with ❤ by GitHub view raw

This way you are able to set the size of your canvas using only CSS, and your draw will not look distorted. You can call this function before the draw function or even inside it.

You may think that the resize function isn't doing anything at all, but without it your canvas will keep the same initial logical size, therefore, the space used to draw (canvas logical size) can be bigger or smaller than the size being displayed (canvas style size) and you will have trouble because canvas will scale the draw to fit the whole displayed canvas. You can learn more about logical size and CSS size of the canvas here and here.

High pixel density devices

When your canvas is running in some device with high pixel density like modern smartphones, the draw can look blurry. To avoid that, you must define the size of your canvas according on the device pixel ratio.

```
function resizeCanvas(canvas) {
  const { width, height } = canvas.getBoundingClientRect()

  if (canvas.width !== width || canvas.height !== height) {
    const { devicePixelRatio:ratio=1 } = window
    const context = canvas.getContext('2d')
    canvas.width = width*ratio
    canvas.height = height*ratio
    context.scale(ratio, ratio)
    return true
  }

  return false
}
```

resizeCanvasWithDevicePixelRatio.js hosted with ❤ by GitHub view raw

Here we rewrite the resize canvas function to take into account the device pixel ratio. It is necessary to scale to the ratio to draw with the actual CSS pixels. Learn more about correcting resolution in a canvas here.

Pre-draw and post-draw

In the draw function, there is some procedures that we might want execute for every animation, like clear the canvas or increment the frame counter. We could abstract this procedures to special functions that will be executed before and after the draw: the predraw and postdraw functions. This way we can write less code every time we want to create a different animation.

These functions must be called (obviously) inside of the render function, before and after the draw function.

```
const postdraw = () => {
 index++
 ctx.restore()
}
```

postdraw.js hosted with ❤ by GitHub view raw

```
const predraw = (context, canvas) => {
  context.save()
  resizeCanvasToDisplaySize(context, canvas)
  const { width, height } = context.canvas
  context.clearRect(0, 0, width, height)
}
```

predraw.js hosted with ❤ by GitHub view raw

We could just drop the code of these functions in the render function, but writing in functions will allow us to replace them by taking them from props in the component like that:

```
import ...

const _predraw = (context) => { ... }
const _postdraw = () => { ... }
const Canvas = props => {

  const { draw, predraw=_predraw, postdraw=_postdraw } = props
  const canvasRef = useCanvas(draw, {predraw, postdraw})

  return <canvas ref={canvasRef} {...rest}/>

}

export default Canvas
```

Note that you still should make some modification in the useCanvas hook in order to accept the predraw and postdraw function. You can modularize it in the way that attends you!

Well, that is it! Now you have what is needed to make a complete Canvas component with custom hooks for making animations, static draws or even mini-games. 🎉🎊🎊🎉✨🏆

Reactjs React Hook Html5 Canvas Animation JavaScript