Optimizing canvas

← Previous Next →

The `<canvas>` element is one of the most widely used tools for rendering 2D graphics on the web. However, when websites and apps push the Canvas API to its limits, performance begins to suffer. This article provides suggestions for optimizing your use of the canvas element to ensure that your graphics perform well.

Performance tips

The following is a collection of tips to improve canvas performance.

Pre-render similar primitives or repeating objects on an offscreen canvas

If you find yourself repeating some of the same drawing operations on each animation frame, consider offloading them to an offscreen canvas. You can then render the offscreen image to your primary canvas as often as needed, without unnecessarily repeating the steps needed to generate it in the first place.

JS

```
myCanvas.offscreenCanvas = document.createElement("canvas");
myCanvas.offscreenCanvas.width = myCanvas.width;
myCanvas.offscreenCanvas.height = myCanvas.height;

myCanvas.getContext("2d").drawImage(myCanvas.offScreenCanvas, 0, 0);
```

Avoid floating-point coordinates and use integers instead

Sub-pixel rendering occurs when you render objects on a canvas without whole values.

JS

```
ctx.drawImage(myImage, 0.3, 0.5);
```

This forces the browser to do extra calculations to create the anti-aliasing effect. To avoid this, make sure to round all co-ordinates used in calls to `drawImage()` using `Math.floor()`, for example.

Don't scale images in `drawImage`

Cache various sizes of your images on an offscreen canvas when loading as opposed to constantly scaling them in `drawImage()`.

Use multiple layered canvases for complex scenes

In your application, you may find that some objects need to move or change frequently, while others remain relatively static. A possible optimization in this situation is to layer your items using multiple `<canvas>` elements.

For example, let's say you have a game with a UI on top, the gameplay action in the middle, and a static background on the bottom. In this case, you could split your game into three `<canvas>` layers. The UI would change only upon user input, the gameplay layer would change with every new frame, and the background would remain generally unchanged.

HTML

```
<div id="stage">
  <canvas id="ui-layer" width="480" height="320"></canvas>
  <canvas id="game-layer" width="480" height="320"></canvas>
  <canvas id="background-layer" width="480" height="320"></canvas>
</div>
```

CSS

```
#stage {
  width: 480px;
  height: 320px;
  position: relative;
  border: 2px solid black;
}

canvas {
  position: absolute;
}
#ui-layer {
  z-index: 3;
}
#game-layer {
  z-index: 2;
}
#background-layer {
  z-index: 1;
}
```

Use plain CSS for large background images

If you have a static background image, you can draw it onto a plain `<div>` element using the CSS `background` property and position it under the canvas. This will negate the need to render the background to the canvas on every tick.

Scaling canvas using CSS transforms

CSS transforms are faster since they use the GPU. The best case is to not scale the canvas, or have a smaller canvas and scale up rather than a bigger canvas and scale down.

JS

```
const scaleX = window.innerWidth / canvas.width;
const scaleY = window.innerHeight / canvas.height;

const scaleToFit = Math.min(scaleX, scaleY);
const scaleToCover = Math.max(scaleX, scaleY);

stage.style.transformOrigin = "0 0"; // Scale from top left
stage.style.transform = `scale(${scaleToFit})`;
```

Turn off transparency

If your application uses canvas and doesn't need a transparent backdrop, set the `alpha` option to `false` when creating a drawing context with `HTMLCanvasElement.getContext()`. This information can be used internally by the browser to optimize rendering.

JS

```
const ctx = canvas.getContext("2d", { alpha: false });
```

Scaling for high resolution displays

You may find that canvas items appear blurry on higher-resolution displays. While many solutions may exist, a simple first step is to scale the canvas size up and down simultaneously, using its attributes, styling, and its context's scale.

JS

```
// Get the DPR and size of the canvas
const dpr = window.devicePixelRatio;
const rect = canvas.getBoundingClientRect();

// Set the "actual" size of the canvas
```

```
canvas.width = rect.width * dpr;
canvas.height = rect.height * dpr;

// Scale the context to ensure correct drawing operations
ctx.scale(dpr, dpr);

// Set the "drawn" size of the canvas
canvas.style.width = `${rect.width}px`;
canvas.style.height = `${rect.height}px`;
```

More tips

- Batch canvas calls together. For example, draw a polyline instead of multiple separate lines.
- Avoid unnecessary canvas state changes.
- Render screen differences only, not the whole new state.
- Avoid the `shadowBlur` property whenever possible.
- Avoid text rendering whenever possible.
- Try different ways to clear the canvas (`clearRect()` vs. `fillRect()` vs. resizing the canvas).
- With animations, use `Window.requestAnimationFrame()` instead of `setInterval()`.
- Be careful with heavy physics libraries.

← Previous Next →