JavaScript Canvas Programming

readbytes

JavaScript Canvas Programming

From Basics to Advanced Graphics

readbytes.github.io

2025-07-10

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to Canvas and Setup

1. What Is the HTML5 Canvas?
2. Setting Up the Canvas Element
3. The Canvas 2D Context Explained
4. Drawing Your First Shape: A Simple Rectangle
5. Clearing and Resizing the Canvas
6. Practical Example: Drawing a Basic Scene

1 Introduction to Canvas and Setup

1.1 What Is the HTML5 Canvas?

The <canvas> element is a powerful addition to the HTML5 specification that allows developers to draw graphics directly within a web page using JavaScript. Think of it as a blank, rectangular "drawing board" on which you can paint 2D shapes, text, images, and even create complex animations or interactive visual experiences. It can also be used with WebGL for 3D rendering, though this book focuses on 2D graphics.

A Blank Slate for Dynamic Graphics

When included in HTML, the <canvas> element doesn't display anything by default—it's just an empty area. You must access it through JavaScript and use its drawing API to create visual output. Here's what the basic HTML looks like:

```
<canvas id="myCanvas" width="400" height="300"></canvas>
```

Then in JavaScript:

```
const canvas = document.getElementById("myCanvas");
const ctx = canvas.getContext("2d");

// Draw a filled rectangle
ctx.fillStyle = "blue";
ctx.fillRect(50, 50, 100, 75);
```

This code draws a solid blue rectangle 100 pixels wide and 75 pixels high at coordinates (50, 50).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Canvas Rectangle</title>
</head>
<body>

<canvas id="myCanvas" width="400" height="300" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

  // Draw a filled rectangle
  ctx.fillStyle = "blue";
  ctx.fillRect(50, 50, 100, 75);
</script>

</body>
</html>
```

1.1.1 Canvas vs. SVG and Traditional HTML

To understand what makes canvas special, it helps to compare it with SVG (Scalable Vector Graphics) and traditional HTML elements.

Feature	`<canvas>` (Immediate Mode)	SVG / HTML (Retained Mode)
Rendering Model	Immediate: Draw commands are executed directly and don't retain state	Retained: DOM-like structure stores visual elements
Interactivity	Manual: You must track shapes and respond to events yourself	Built-in: Elements are part of the DOM and can receive events
Performance	High for dynamic, pixel-heavy scenes	Better for static or DOM-integrated graphics
Best Use Cases	Games, visualizations, animations, image editing	Diagrams, charts, UI components

What Is Immediate Mode?

Canvas uses what's called an **immediate mode rendering model**. Once you draw something on the canvas, it's rendered to pixels and forgotten by the system. If you need to change it, you must redraw the entire scene. This contrasts with SVG or HTML, which retain elements in memory and update them individually when needed.

1.1.2 Why Use Canvas?

Canvas is used in many modern web applications where performance and pixel-level control are essential:

- **Game Development**: Real-time rendering of sprites, backgrounds, and effects.
- **Data Visualization**: Dynamic charts and graphs that scale to large datasets.
- **Image and Video Editing**: Tools like cropping, filtering, or drawing.
- **Custom Animations**: Animating shapes, particles, and physics-based effects.

By controlling each pixel, developers can push the limits of what's possible inside a web browser.

> **Example Use Case** A 2D racing game updates the screen 60 times per second using canvas to draw the background, cars, road, and effects like smoke—all in real-time.

1.2 Setting Up the Canvas Element

Before we can draw anything with the HTML5 Canvas API, we need to include the `<canvas>` element in our HTML and connect to it using JavaScript. This section covers how to properly create, configure, and access the canvas in a way that works reliably across modern browsers.

1.2.1 Declaring the `canvas` Element

The `<canvas>` element is a standard HTML tag introduced in HTML5. It acts as a drawing surface and must be given explicit dimensions—otherwise, it will default to **300px wide** and **150px high**, which might not match your layout needs.

Here's the minimal setup:

```
<canvas id="myCanvas" width="400" height="300"></canvas>
```

Attributes:

- `id`: Used to uniquely identify the canvas so JavaScript can find it.
- `width` and `height`: Define the actual resolution of the canvas drawing surface.

  WARNING **Important**: The width and height attributes should be set **in the HTML tag**, not via CSS. Setting dimensions via CSS only scales the canvas visually and can blur graphics.

1.2.2 Accessing the Canvas with JavaScript

To draw on the canvas, you need to access it from JavaScript and get its **2D rendering context**—an object that contains all the drawing methods.

There are two common ways to grab the canvas element:

Using `getElementById()`:

```
const canvas = document.getElementById("myCanvas");
```

Using `querySelector()`:

```
const canvas = document.querySelector("#myCanvas");
```

Then, obtain the 2D rendering context:

```
const ctx = canvas.getContext("2d");
```

Now, `ctx` gives you access to all drawing functions like `fillRect`, `stroke`, `beginPath`, and more.

1.2.3 Fallback for Older Browsers

Nearly all modern browsers support `<canvas>`, but in rare cases where it's unsupported, you can provide fallback content between the opening and closing `<canvas>` tags:

```
<canvas id="myCanvas" width="400" height="300">
  Your browser does not support the HTML5 canvas tag.
</canvas>
```

Browsers that understand `<canvas>` will ignore the fallback text, but others will show it as a message.

1.2.4 Complete Example: Basic Canvas Setup

Here's a full example that sets up a canvas and draws a rectangle using the 2D context:

```
  <script>
    // Access the canvas element
    const canvas = document.getElementById("myCanvas");

    // Get the 2D rendering context
    const ctx = canvas.getContext("2d");

    // Draw a red rectangle
    ctx.fillStyle = "red";
    ctx.fillRect(50, 50, 150, 100);
  </script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Canvas Setup Example</title>
  <style>
    canvas {
      border: 1px solid #000;
    }
  </style>
</head>
<body>
  <canvas id="myCanvas" width="400" height="300">
```

```
    Your browser does not support the HTML5 canvas tag.
  </canvas>

  <script>
    // Access the canvas element
    const canvas = document.getElementById("myCanvas");

    // Get the 2D rendering context
    const ctx = canvas.getContext("2d");

    // Draw a red rectangle
    ctx.fillStyle = "red";
    ctx.fillRect(50, 50, 150, 100);
  </script>
</body>
</html>
```

YES When you open this in a browser, you should see a red rectangle drawn inside a bordered canvas area.

1.2.5 Summary

Setting up a canvas in HTML is simple, but doing it correctly is essential for clean and responsive graphics. Always specify the canvas's width and height directly in the HTML tag, and use JavaScript to access the element and acquire the drawing context. In the next section, we'll explore what the 2D rendering context is and how it powers everything you draw on the canvas.

1.3 The Canvas 2D Context Explained

Once you've set up a `<canvas>` element in your HTML, the next step is accessing its **2D rendering context**—an object that provides all the methods and properties needed to draw on the canvas.

This section introduces the `getContext('2d')` method, explains the powerful drawing capabilities it unlocks, and walks you through the key categories of canvas API methods. We'll also demonstrate a small working example and explore the concept of **drawing state**.

1.3.1 What Is `getContext('2d')`?

The `getContext('2d')` method is used to retrieve the **2D drawing context** from a canvas element. It looks like this:

```
const canvas = document.getElementById("myCanvas");
const ctx = canvas.getContext("2d");
```

Once you have the `ctx` object (short for **context**), you can use its methods to draw shapes, render text, apply transformations, draw images, and more.

1.3.2 What Can the 2D Context Do?

The 2D context provides a rich set of methods and properties that fall into five major categories:

Shapes

Draw rectangles, lines, circles, paths, and curves.

- `fillRect(x, y, width, height)`
- `strokeRect(x, y, width, height)`
- `beginPath()`, `moveTo(x, y)`, `lineTo(x, y)`, `arc()`, `closePath()`, `stroke()`, `fill()`

Styles and Colors

Customize how shapes are filled or outlined.

- `fillStyle`, `strokeStyle`
- `lineWidth`, `lineCap`, `lineJoin`
- `createLinearGradient()`, `createPattern()`

Text

Draw and style text directly on the canvas.

- `fillText(text, x, y)`
- `strokeText(text, x, y)`
- `font`, `textAlign`, `textBaseline`

Transformations

Move, scale, or rotate the canvas drawing coordinate system.

- `translate(x, y)`
- `rotate(angle)`
- `scale(x, y)`
- `save()`, `restore()`

Images and Pixel Manipulation

Draw images or manipulate pixels directly.

- `drawImage(image, x, y)`

- `getImageData()`, `putImageData()`
- `createImageData()`

1.3.3 Simple Demo: Drawing Shapes and Text

Here's a complete example that uses a few 2D context methods:

```
  <style>
    canvas {
      border: 1px solid #ccc;
    }
  </style>

<canvas id="myCanvas" width="400" height="300"></canvas>

  <script>
    const canvas = document.getElementById("myCanvas");
    const ctx = canvas.getContext("2d");

    // Draw a filled rectangle
    ctx.fillStyle = "skyblue";
    ctx.fillRect(50, 50, 200, 100);

    // Draw a border around the rectangle
    ctx.strokeStyle = "navy";
    ctx.lineWidth = 4;
    ctx.strokeRect(50, 50, 200, 100);

    // Draw text
    ctx.font = "20px Arial";
    ctx.fillStyle = "black";
    ctx.fillText("Hello Canvas!", 70, 120);
  </script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>2D Context Demo</title>
  <style>
    canvas {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>
  <canvas id="myCanvas" width="400" height="300"></canvas>

  <script>
    const canvas = document.getElementById("myCanvas");
    const ctx = canvas.getContext("2d");

    // Draw a filled rectangle
```

```
    ctx.fillStyle = "skyblue";
    ctx.fillRect(50, 50, 200, 100);

    // Draw a border around the rectangle
    ctx.strokeStyle = "navy";
    ctx.lineWidth = 4;
    ctx.strokeRect(50, 50, 200, 100);

    // Draw text
    ctx.font = "20px Arial";
    ctx.fillStyle = "black";
    ctx.fillText("Hello Canvas!", 70, 120);
  </script>
</body>
</html>
```

This example draws a sky-blue rectangle with a navy border and places some styled text inside it. You can try changing the color, font, or position to experiment.

1.3.4 Understanding the Drawing State

The **2D context maintains a drawing state**, which includes things like:

- Current `fillStyle` and `strokeStyle`
- Line width and font
- Transformation matrix
- Clipping region

This means that once you set a property (like `ctx.fillStyle = "red"`), it remains in effect until you change it again. This makes the context **stateful**.

To manage complex scenes, you can use:

- `ctx.save()` — Saves the current state on a stack.
- `ctx.restore()` — Pops the last saved state off the stack and restores it.

Example:

```
ctx.fillStyle = "green";
ctx.fillRect(10, 10, 100, 100);  // Green square

ctx.save(); // Save green fillStyle

ctx.fillStyle = "orange";
ctx.fillRect(120, 10, 100, 100); // Orange square

ctx.restore(); // Restore green

ctx.fillRect(230, 10, 100, 100); // Green square again
```

1.3.5 Summary

The `getContext('2d')` method opens the door to a robust API for drawing 2D graphics on the canvas. Whether you're drawing rectangles, styling text, applying transformations, or animating sprites, the 2D context is your toolbox. In the next section, you'll use some of these tools to draw your very first shape on the canvas.

1.4 Drawing Your First Shape: A Simple Rectangle

Now that you've set up the canvas and learned about the 2D context, it's time to draw your first shape. In this section, you'll create a **filled rectangle** and a **stroked (outlined) rectangle** using the simplest drawing methods provided by the 2D context: `fillRect()` and `strokeRect()`.

We'll also explain how the canvas coordinate system works so you know exactly where and how shapes appear on the screen.

1.4.1 The Canvas Coordinate System

The canvas uses a **pixel-based coordinate system** with the origin `(0, 0)` at the **top-left corner** of the canvas. All positions and dimensions are measured in pixels.

```
(0, 0)  → Origin (top-left)

+------------------------------→ X (horizontal)
|
|
|
V
Y (vertical)
```

So, a point at `(100, 50)` is 100 pixels to the right and 50 pixels down from the top-left corner.

1.4.2 Rectangle Drawing Methods

There are two basic methods to draw rectangles:

`fillRect(x, y, width, height)`

Draws a solid rectangle filled with the current `fillStyle`.

`strokeRect(x, y, width, height)`

Draws only the border of a rectangle using the current `strokeStyle` and `lineWidth`.

1.4.3 Hands-On Example: Drawing Two Rectangles

Let's put this into action with a complete example:

```
  <script>
    // Step 1: Get the canvas and context
    const canvas = document.getElementById("myCanvas");
    const ctx = canvas.getContext("2d");

    // Step 2: Set the fill color and draw a filled rectangle
    ctx.fillStyle = "steelblue";           // Set fill color
    ctx.fillRect(50, 40, 120, 80);         // Draw filled rectangle at (50, 40)

    // Step 3: Set the stroke color and draw a rectangle outline
    ctx.strokeStyle = "black";             // Set border color
    ctx.lineWidth = 4;                     // Set border thickness
    ctx.strokeRect(200, 100, 120, 80);     // Draw outline-only rectangle
  </script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Drawing Rectangles</title>
  <style>
    canvas {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>
  <canvas id="myCanvas" width="400" height="300"></canvas>

  <script>
    // Step 1: Get the canvas and context
    const canvas = document.getElementById("myCanvas");
    const ctx = canvas.getContext("2d");

    // Step 2: Set the fill color and draw a filled rectangle
    ctx.fillStyle = "steelblue";           // Set fill color
    ctx.fillRect(50, 40, 120, 80);         // Draw filled rectangle at (50, 40)

    // Step 3: Set the stroke color and draw a rectangle outline
    ctx.strokeStyle = "black";             // Set border color
    ctx.lineWidth = 4;                     // Set border thickness
```

```
    ctx.strokeRect(200, 100, 120, 80);     // Draw outline-only rectangle
  </script>
</body>
</html>
```

1.4.4 What Each Line Does

```
ctx.fillStyle = "steelblue";    // Sets the fill color for shapes
ctx.fillRect(50, 40, 120, 80);  // Draws a solid rectangle 120x80 at (50, 40)

ctx.strokeStyle = "black";      // Sets the stroke (outline) color
ctx.lineWidth = 4;              // Sets the thickness of the outline
ctx.strokeRect(200, 100, 120, 80); // Draws only the border of a 120x80 rectangle at (200, 100)
```

1.4.5 Try Modifying It Yourself!

Experiment with different values to see how canvas behaves:

- **Change the position**: Try `fillRect(10, 10, 100, 50)` to draw near the top-left.
- **Change the size**: Try `fillRect(100, 50, 200, 200)` for a large square.
- **Change the color**: Use `ctx.fillStyle = "orange"` or any CSS color value.
- **Combine shapes**: Try drawing multiple rectangles in different areas.

1.4.6 Summary

With just a few lines of code, you've drawn your first shapes on the canvas using `fillRect()` and `strokeRect()`. These simple functions form the building blocks of canvas graphics. Up next, you'll learn how to **clear and resize the canvas**, which is especially useful for animation and interaction.

1.5 Clearing and Resizing the Canvas

As you begin to create interactive or animated graphics with the canvas, you'll often need to **erase** what's already been drawn or **resize** the canvas area dynamically. This section explains how to use `clearRect()` to clear specific regions, how to resize the canvas via JavaScript, and what to watch out for when resizing—especially during animations or redrawing.

1.5.1 Clearing the Canvas with `clearRect()`

The primary method to clear part or all of a canvas is:

```
ctx.clearRect(x, y, width, height);
```

This function clears a rectangular region of the canvas, making it fully transparent. It's especially useful when updating or animating the canvas to prevent previous frames from stacking visually.

Example: Clearing the Whole Canvas

```
ctx.clearRect(0, 0, canvas.width, canvas.height);
```

> YES This clears the entire drawing area, effectively erasing everything previously drawn.

1.5.2 When Should You Use `clearRect()`?

- **In animations**: To clear the canvas before drawing each new frame.
- **On user interaction**: When resetting the drawing state or changing the scene.
- **During redrawing**: When updating only part of a scene (like dragging or resizing).

Example: Clearing and Redrawing a Moving Shape

```
  <script>
    const canvas = document.getElementById("myCanvas");
    const ctx = canvas.getContext("2d");

    let x = 0;

    function draw() {
      ctx.clearRect(0, 0, canvas.width, canvas.height); // Clear previous frame
      ctx.fillStyle = "blue";
      ctx.fillRect(x, 50, 50, 50); // Draw rectangle at new position
      x += 2;
      requestAnimationFrame(draw);
    }

    draw();
  </script>
```

```
<!DOCTYPE html>
<html>
<head>
  <title>Clear Canvas Animation</title>
</head>
```

```
<body>
  <canvas id="myCanvas" width="400" height="200" style="border:1px solid #000;"></canvas>
  <script>
    const canvas = document.getElementById("myCanvas");
    const ctx = canvas.getContext("2d");

    let x = 0;

    function draw() {
      ctx.clearRect(0, 0, canvas.width, canvas.height); // Clear previous frame
      ctx.fillStyle = "blue";
      ctx.fillRect(x, 50, 50, 50); // Draw rectangle at new position
      x += 2;
      requestAnimationFrame(draw);
    }

    draw();
  </script>
</body>
</html>
```

This creates a simple animation where a rectangle moves to the right, clearing the canvas on each frame.

1.5.3 Resizing the Canvas Dynamically

Sometimes you'll want to adjust the canvas size based on the browser window, a device's resolution, or user input. You can resize the canvas by setting its `width` and `height` properties **in JavaScript**.

Example: Resize to Window Width

```
canvas.width = window.innerWidth;
canvas.height = window.innerHeight;
```

WARNING **Important:** Resizing the canvas **resets its contents** and **clears the drawing state** (like `fillStyle`, `strokeStyle`, `font`, etc.).

1.5.4 Best Practices for Resizing

1. **Save and restore state if needed:**

   If you're using styles or transformations, you'll need to reapply them after resizing.

   ```
   const oldFill = ctx.fillStyle;

   canvas.width = 800;  // Resize resets canvas
   ```

```
canvas.height = 600;

ctx.fillStyle = oldFill; // Restore style
```

2. **Re-render content after resizing:**

   If you need to preserve or redraw content, keep your drawing logic in a function you can call again after resizing.

```
function drawScene() {
  ctx.fillStyle = "green";
  ctx.fillRect(10, 10, 100, 100);
}

canvas.width = 500;
canvas.height = 400;
drawScene(); // Redraw after resize
```

3. **Use `window.devicePixelRatio` for high-DPI screens (optional advanced tip):**

   For sharper rendering on devices with high-resolution displays:

```
const ratio = window.devicePixelRatio || 1;
canvas.width = canvas.clientWidth * ratio;
canvas.height = canvas.clientHeight * ratio;
ctx.scale(ratio, ratio);
```

1.5.5 Summary

Clearing and resizing the canvas are essential techniques, especially for animations and dynamic interfaces. Use `clearRect()` to erase old content, and be cautious when resizing the canvas via JavaScript—it clears all drawings and resets context settings. To handle this properly, restore your drawing state and re-render your content as needed.

In the next section, we'll use all the concepts you've learned so far to build a **basic canvas scene**—combining shapes, colors, and clearing techniques in a single working example.

1.6 Practical Example: Drawing a Basic Scene

Now that you've learned how to set up the canvas, draw shapes, clear the canvas, and understand the 2D context, it's time to put it all together in a complete, practical example.

In this section, we'll draw a **simple outdoor scene** featuring the sky, sun, grass, and a tree. This will give you hands-on experience with combining shapes, layering elements correctly, and using colors creatively. Along the way, we'll emphasize good code structure and encourage you to experiment with your own variations.

1.6.1 Scene Overview

We'll draw the following components:

- A **blue sky** background using a rectangle.
- A **sun** using a filled circle.
- A **green ground** using a wide rectangle.
- A **tree** composed of a brown rectangle (trunk) and green circle (leaves).

1.6.2 Complete Example: Drawing the Scene

```
<script>
  const canvas = document.getElementById("sceneCanvas");
  const ctx = canvas.getContext("2d");

  // --- Draw Background Sky ---
  ctx.fillStyle = "#87CEEB"; // Sky blue
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  // --- Draw Sun ---
  ctx.beginPath();
  ctx.arc(500, 80, 40, 0, Math.PI * 2, false); // Circle at (500, 80) radius 40
  ctx.fillStyle = "yellow";
  ctx.fill();

  // --- Draw Ground ---
  ctx.fillStyle = "#228B22"; // Forest green
  ctx.fillRect(0, 300, canvas.width, 100); // Ground rectangle

  // --- Draw Tree Trunk ---
  ctx.fillStyle = "#8B4513"; // Saddle brown
  ctx.fillRect(100, 220, 30, 80); // Trunk

  // --- Draw Tree Leaves ---
  ctx.beginPath();
  ctx.arc(115, 200, 40, 0, Math.PI * 2); // Leaves circle
  ctx.fillStyle = "#006400"; // Dark green
  ctx.fill();

  // Optional: Add another layer of leaves for depth
  ctx.beginPath();
  ctx.arc(135, 210, 30, 0, Math.PI * 2);
  ctx.fill();

  ctx.beginPath();
  ctx.arc(95, 210, 30, 0, Math.PI * 2);
  ctx.fill();
</script>
```

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <title>Canvas Scene Example</title>
  <style>
    canvas {
      border: 1px solid #aaa;
      display: block;
      margin: 20px auto;
    }
  </style>
</head>
<body>
  <canvas id="sceneCanvas" width="600" height="400"></canvas>

  <script>
    const canvas = document.getElementById("sceneCanvas");
    const ctx = canvas.getContext("2d");

    // --- Draw Background Sky ---
    ctx.fillStyle = "#87CEEB"; // Sky blue
    ctx.fillRect(0, 0, canvas.width, canvas.height);

    // --- Draw Sun ---
    ctx.beginPath();
    ctx.arc(500, 80, 40, 0, Math.PI * 2, false); // Circle at (500, 80) radius 40
    ctx.fillStyle = "yellow";
    ctx.fill();

    // --- Draw Ground ---
    ctx.fillStyle = "#228B22"; // Forest green
    ctx.fillRect(0, 300, canvas.width, 100); // Ground rectangle

    // --- Draw Tree Trunk ---
    ctx.fillStyle = "#8B4513"; // Saddle brown
    ctx.fillRect(100, 220, 30, 80); // Trunk

    // --- Draw Tree Leaves ---
    ctx.beginPath();
    ctx.arc(115, 200, 40, 0, Math.PI * 2); // Leaves circle
    ctx.fillStyle = "#006400"; // Dark green
    ctx.fill();

    // Optional: Add another layer of leaves for depth
    ctx.beginPath();
    ctx.arc(135, 210, 30, 0, Math.PI * 2);
    ctx.fill();

    ctx.beginPath();
    ctx.arc(95, 210, 30, 0, Math.PI * 2);
    ctx.fill();
  </script>
</body>
</html>
```

1.6.3 Code Organization and Drawing Sequence

Canvas draws elements in the order they are called in code. That means **background elements must be drawn first**, and foreground items later—just like layering paint on a canvas.

Layering order in our example:

1. Sky background
2. Sun (appears behind tree)
3. Ground (covers bottom of the sky)
4. Tree trunk
5. Tree leaves (overlaps trunk and sun)

Organizing your code into **sections** (with comments or functions) makes it easier to manage and update individual parts of the scene.

1.6.4 Encourage Creative Variations

Try modifying the scene to make it your own:

- **Day/Night Theme**:
  - Change the sky to `#001f3f` for night and the sun to a moon using light gray.
- **Add Gradient Background**:

```
const gradient = ctx.createLinearGradient(0, 0, 0, canvas.height);
gradient.addColorStop(0, "#87CEEB");   // Light sky blue
gradient.addColorStop(1, "#ffffff");   // White at the bottom
ctx.fillStyle = gradient;
ctx.fillRect(0, 0, canvas.width, canvas.height);
```

- **Scale Elements**: Try changing the radius of the sun, the size of the tree, or adding multiple trees of different sizes to simulate depth.
- **Add More Detail**: Draw clouds using multiple overlapping circles, or a house using rectangles and triangles.

1.6.5 Summary

You've now created your first fully drawn canvas scene by combining rectangles, circles, and colors. You also practiced proper layering and code organization to keep the drawing clear and maintainable. In future chapters, you'll build on these skills to add interactivity, animation, and even game-like behaviors to your canvas graphics.

Chapter 2.

Basic Drawing Techniques

1. Drawing Rectangles, Paths, and Lines
2. Drawing Circles and Arcs
3. Using Colors, Gradients, and Patterns
4. Working with Stroke and Fill Styles
5. Practical Example: Drawing a Colorful House

2 Basic Drawing Techniques

2.1 Drawing Rectangles, Paths, and Lines

In this section, we'll expand your drawing skills by introducing **rectangles**, **paths**, and **lines**—the building blocks of most 2D canvas graphics.

We'll begin with the canvas's convenient rectangle methods (`fillRect()` and `strokeRect()`), then move on to path-based drawing using `beginPath()`, `moveTo()`, `lineTo()`, and `stroke()` to draw custom lines and shapes.

2.1.1 Drawing Rectangles

The 2D context provides two simple methods for drawing rectangles without needing to define paths:

`fillRect(x, y, width, height)`

Draws a **filled** rectangle using the current `fillStyle`.

`strokeRect(x, y, width, height)`

Draws the **outline** of a rectangle using the current `strokeStyle` and `lineWidth`.

Example:

```
ctx.fillStyle = "skyblue";
ctx.fillRect(30, 30, 100, 60);  // Filled rectangle

ctx.strokeStyle = "black";
ctx.lineWidth = 3;
ctx.strokeRect(160, 30, 100, 60);  // Outlined rectangle
```

2.1.2 Drawing Custom Lines with Paths

While rectangle methods are quick and convenient, they're limited to axis-aligned rectangles. For more flexible shapes—like lines, polygons, or custom figures—you use **paths**.

Basic Path Drawing Steps:

1. `beginPath()` – Starts a new path.
2. `moveTo(x, y)` – Moves the pen to a starting point without drawing.
3. `lineTo(x, y)` – Draws a straight line from the current point to the given point.
4. `stroke()` – Outlines the path with the current stroke style.

Example: Drawing a Diagonal Line

```
ctx.beginPath();
ctx.moveTo(50, 50);      // Start point
ctx.lineTo(200, 100);    // End point
ctx.strokeStyle = "red";
ctx.lineWidth = 2;
ctx.stroke();
```

2.1.3 Rectangles vs. Paths: Whats the Difference?

Feature	Rectangle Methods	Path-Based Drawing
Ease of Use	Very easy (`fillRect`, etc)	More steps, but more flexible
Shape Support	Rectangles only	Any line-based shape
Control Over Paths	No	Yes (can build custom shapes)
Can Combine Multiple Lines	No	Yes (draw polygons, etc.)

Use **rectangle methods** when you just need a quick filled or outlined box. Use **paths** when you need flexibility—like drawing angled lines, complex shapes, or animations.

2.1.4 Example: Drawing Multiple Lines with Varying Angles and Widths

```
<script>
  const canvas = document.getElementById("lineCanvas");
  const ctx = canvas.getContext("2d");

  // Line 1: Thin, horizontal
  ctx.beginPath();
  ctx.moveTo(20, 40);
  ctx.lineTo(200, 40);
  ctx.strokeStyle = "blue";
  ctx.lineWidth = 2;
  ctx.stroke();

  // Line 2: Medium, diagonal
  ctx.beginPath();
  ctx.moveTo(20, 80);
  ctx.lineTo(200, 120);
  ctx.strokeStyle = "green";
  ctx.lineWidth = 4;
  ctx.stroke();
```

```
    // Line 3: Thick, vertical
    ctx.beginPath();
    ctx.moveTo(100, 150);
    ctx.lineTo(100, 250);
    ctx.strokeStyle = "purple";
    ctx.lineWidth = 6;
    ctx.stroke();
  </script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Canvas Lines Demo</title>
  <style>
    canvas {
      border: 1px solid #aaa;
      display: block;
      margin: 20px auto;
    }
  </style>
</head>
<body>
  <canvas id="lineCanvas" width="400" height="300"></canvas>

  <script>
    const canvas = document.getElementById("lineCanvas");
    const ctx = canvas.getContext("2d");

    // Line 1: Thin, horizontal
    ctx.beginPath();
    ctx.moveTo(20, 40);
    ctx.lineTo(200, 40);
    ctx.strokeStyle = "blue";
    ctx.lineWidth = 2;
    ctx.stroke();

    // Line 2: Medium, diagonal
    ctx.beginPath();
    ctx.moveTo(20, 80);
    ctx.lineTo(200, 120);
    ctx.strokeStyle = "green";
    ctx.lineWidth = 4;
    ctx.stroke();

    // Line 3: Thick, vertical
    ctx.beginPath();
    ctx.moveTo(100, 150);
    ctx.lineTo(100, 250);
    ctx.strokeStyle = "purple";
    ctx.lineWidth = 6;
    ctx.stroke();
  </script>
</body>
</html>
```

This example draws three different lines:

- A thin blue horizontal line,
- A medium green diagonal line,
- And a thick vertical purple line.

2.1.5 Mini-Demo: Drawing a Zigzag Pattern

Let's create a connected path using `lineTo()` to draw a zigzag shape:

```
ctx.beginPath();
ctx.moveTo(50, 200);   // Start point
ctx.lineTo(100, 150);
ctx.lineTo(150, 200);
ctx.lineTo(200, 150);
ctx.lineTo(250, 200);
ctx.strokeStyle = "orange";
ctx.lineWidth = 3;
ctx.stroke();
```

This code draws a continuous line in a zigzag pattern across the canvas.

2.1.6 Summary

In this section, you learned how to draw:

- **Filled and stroked rectangles** quickly using `fillRect()` and `strokeRect()`
- **Custom lines and shapes** using path-based methods
- **Lines with different angles and thicknesses**

You now have a foundation for drawing both static and dynamic shapes. In the next section, you'll learn how to bring your shapes to life with **colors, gradients, and patterns** to make your canvas artwork more visually rich and expressive.

2.2 Drawing Circles and Arcs

Circles and curves are essential elements in 2D graphics. In this section, you'll learn how to draw full and partial circles, semicircles, and curves using the `arc()` and `arcTo()` methods. We'll explain how the math works behind the scenes—especially how angles in radians control the shape and direction of arcs. You'll also get hands-on practice drawing useful shapes like clocks, smiley faces, and gauge-style meters.

2.2.1 The `arc()` Method

The most common way to draw circles or curved paths is with the `arc()` method.

Syntax:

```
ctx.arc(x, y, radius, startAngle, endAngle, counterclockwise);
```

Parameter	Description
`x`, `y`	The center of the arc (or circle)
`radius`	Distance from center to edge
`startAngle`	Angle in **radians** where the arc begins
`endAngle`	Angle in **radians** where the arc ends
`counterclockwise`	Optional: `true` = draw counterclockwise; default is `false`

2.2.2 Understanding Radians

Canvas uses **radians**, not degrees, to define angles:

- Full circle = `2 * Math.PI` radians `6.283`
- Half circle = `Math.PI` radians `3.1416`
- Quarter circle = `Math.PI / 2` radians `1.57`

Angle direction starts from the positive x-axis and goes clockwise by default.

> **Conversion formula**: Degrees × Pi ÷ 180 = Radians Example: 90° → `90 * Math.PI / 180` = `Math.PI / 2`

2.2.3 Example: Drawing a Circle and Semicircle

```
<canvas id="myCanvas" width="400" height="200"></canvas>
<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

  // Draw full circle
  ctx.beginPath();
  ctx.arc(100, 100, 50, 0, Math.PI * 2);  // Full circle
  ctx.strokeStyle = "blue";
  ctx.stroke();

  // Draw semicircle
  ctx.beginPath();
```

```
  ctx.arc(250, 100, 50, 0, Math.PI);  // Top half
  ctx.strokeStyle = "red";
  ctx.stroke();
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

  // Draw full circle
  ctx.beginPath();
  ctx.arc(100, 100, 50, 0, Math.PI * 2);  // Full circle
  ctx.strokeStyle = "blue";
  ctx.stroke();

  // Draw semicircle
  ctx.beginPath();
  ctx.arc(250, 100, 50, 0, Math.PI);  // Top half
  ctx.strokeStyle = "red";
  ctx.stroke();
</script>

</body>
</html>
```

2.2.4 Drawing Direction: Clockwise vs Counterclockwise

By default, arcs are drawn **clockwise**. If you set the last parameter of `arc()` to `true`, the arc will draw in the **counterclockwise** direction.

```
ctx.arc(200, 100, 50, 0, Math.PI, true);  // Counterclockwise semicircle
```

Changing direction can affect how shapes are filled or animated, especially when combining multiple arcs in one path.

2.2.5 The `arcTo()` Method

While `arc()` draws circular arcs around a center point, `arcTo()` creates smooth **curves that connect two lines**, often used for rounded corners.

Syntax:

```
ctx.arcTo(x1, y1, x2, y2, radius);
```

Parameter	Description
`x1, y1`	First tangent point (start of curve)
`x2, y2`	Second tangent point (end of curve)
`radius`	Radius of the curve that connects the two lines smoothly

Example: Rounded Corner

```
ctx.beginPath();
ctx.moveTo(50, 50);             // Start at top-left
ctx.arcTo(150, 50, 150, 150, 40);  // Curve to lower-right
ctx.lineTo(150, 150);
ctx.stroke();
```

This draws a smooth corner between two straight lines with a circular arc in between.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.beginPath();
ctx.moveTo(50, 50);             // Start at top-left
ctx.arcTo(150, 50, 150, 150, 40);  // Curve to lower-right
ctx.lineTo(150, 150);
ctx.stroke();
</script>

</body>
</html>
```

2.2.6 Practice Ideas: Using Arcs Creatively

Now that you know how to draw arcs, here are a few fun and practical mini-projects to apply what you've learned.

Draw a Clock Face

```
ctx.beginPath();
ctx.arc(200, 150, 100, 0, 2 * Math.PI); // Clock circle
ctx.stroke();

for (let i = 0; i < 12; i++) {
  const angle = (i * Math.PI) / 6;
  const x1 = 200 + Math.cos(angle) * 90;
  const y1 = 150 + Math.sin(angle) * 90;
  const x2 = 200 + Math.cos(angle) * 100;
  const y2 = 150 + Math.sin(angle) * 100;
  ctx.beginPath();
  ctx.moveTo(x1, y1);
  ctx.lineTo(x2, y2);
  ctx.stroke();
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
ctx.beginPath();
ctx.arc(200, 150, 100, 0, 2 * Math.PI); // Clock circle
ctx.stroke();

for (let i = 0; i < 12; i++) {
  const angle = (i * Math.PI) / 6;
  const x1 = 200 + Math.cos(angle) * 90;
  const y1 = 150 + Math.sin(angle) * 90;
  const x2 = 200 + Math.cos(angle) * 100;
  const y2 = 150 + Math.sin(angle) * 100;
  ctx.beginPath();
  ctx.moveTo(x1, y1);
  ctx.lineTo(x2, y2);
  ctx.stroke();
}

</script>

</body>
```

```
</html>
```

Draw a Smiley Face

```
// Face circle
ctx.beginPath();
ctx.arc(200, 150, 80, 0, 2 * Math.PI);
ctx.stroke();

// Eyes
ctx.beginPath();
ctx.arc(170, 130, 10, 0, 2 * Math.PI);
ctx.arc(230, 130, 10, 0, 2 * Math.PI);
ctx.fill();

// Smile
ctx.beginPath();
ctx.arc(200, 150, 50, 0, Math.PI, false); // Bottom half
ctx.stroke();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
// Face circle
ctx.beginPath();
ctx.arc(200, 150, 80, 0, 2 * Math.PI);
ctx.stroke();

// Eyes
ctx.beginPath();
ctx.arc(170, 130, 10, 0, 2 * Math.PI);
ctx.arc(230, 130, 10, 0, 2 * Math.PI);
ctx.fill();

// Smile
ctx.beginPath();
ctx.arc(200, 150, 50, 0, Math.PI, false); // Bottom half
ctx.stroke();

</script>

</body>
</html>
```

Draw a Radial Gauge

```
function drawGauge(value) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  ctx.beginPath();
  ctx.arc(200, 150, 100, Math.PI, 0, false); // Half-circle gauge
  ctx.stroke();

  // Pointer line
  const angle = Math.PI * (1 - value); // value between 0 and 1
  const x = 200 + Math.cos(angle) * 80;
  const y = 150 - Math.sin(angle) * 80;
  ctx.beginPath();
  ctx.moveTo(200, 150);
  ctx.lineTo(x, y);
  ctx.strokeStyle = "red";
  ctx.lineWidth = 3;
  ctx.stroke();
}

drawGauge(0.7); // 70%
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
function drawGauge(value) {
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

  ctx.clearRect(0, 0, canvas.width, canvas.height);

  ctx.beginPath();
  ctx.arc(200, 150, 100, Math.PI, 0, false); // Half-circle gauge
  ctx.stroke();

  // Pointer line
  const angle = Math.PI * (1 - value); // value between 0 and 1
  const x = 200 + Math.cos(angle) * 80;
  const y = 150 - Math.sin(angle) * 80;
  ctx.beginPath();
  ctx.moveTo(200, 150);
  ctx.lineTo(x, y);
  ctx.strokeStyle = "red";
  ctx.lineWidth = 3;
  ctx.stroke();
}
```

```
drawGauge(0.7); // 70%

</script>

</body>
</html>
```

2.2.7 Summary

With `arc()` and `arcTo()`, you now have the tools to draw circles, semicircles, curves, and rounded corners. These methods allow you to create everything from simple decorative elements to dynamic data visualizations. You've also learned the importance of understanding radians and controlling drawing direction—core skills for advanced canvas work.

In the next section, we'll bring your shapes to life with **colors, gradients, and patterns** to give your graphics style and depth.

2.3 Using Colors, Gradients, and Patterns

Visual richness in canvas drawing comes from using **color**, **gradients**, and **patterns**. In this section, you'll learn how to use solid colors with `fillStyle` and `strokeStyle`, how to create linear and radial gradients, and how to apply image-based patterns. You'll also build an interactive example that switches between these styles dynamically.

2.3.1 Solid Colors with `fillStyle` and `strokeStyle`

Canvas lets you set the fill and stroke colors using CSS-style color values:

```
ctx.fillStyle = "blue";      // Fill color
ctx.strokeStyle = "black";   // Border color
ctx.lineWidth = 3;

ctx.fillRect(50, 50, 100, 100);    // Draw filled square
ctx.strokeRect(50, 50, 100, 100);  // Outline the square
```

You can use:

- Color names (`"red"`, `"green"`)
- Hex codes (`"#FF5733"`)
- RGB or RGBA (`"rgb(255, 0, 0)"`, `"rgba(0, 0, 255, 0.5)"`)
- HSL (`"hsl(200, 50%, 50%)"`)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
ctx.fillStyle = "blue";      // Fill color
ctx.strokeStyle = "black";   // Border color
ctx.lineWidth = 3;

ctx.fillRect(50, 50, 100, 100);    // Draw filled square
ctx.strokeRect(50, 50, 100, 100);  // Outline the square

</script>

</body>
</html>
```

2.3.2 Creating Gradients

Gradients let you blend multiple colors together smoothly. Canvas supports two types:

Linear Gradients

Use `ctx.createLinearGradient(x0, y0, x1, y1)` to create a gradient from point A `(x0, y0)` to point B `(x1, y1)`.

```
const gradient = ctx.createLinearGradient(0, 0, 200, 0);
gradient.addColorStop(0, "blue");
gradient.addColorStop(1, "white");

ctx.fillStyle = gradient;
ctx.fillRect(50, 50, 200, 100);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
```

```
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
const gradient = ctx.createLinearGradient(0, 0, 200, 0);
gradient.addColorStop(0, "blue");
gradient.addColorStop(1, "white");

ctx.fillStyle = gradient;
ctx.fillRect(50, 50, 200, 100);

</script>

</body>
</html>
```

This creates a horizontal gradient from blue to white.

Radial Gradients

Use `ctx.createRadialGradient(x0, y0, r0, x1, y1, r1)` to create a circular gradient.

```
const radial = ctx.createRadialGradient(150, 100, 20, 150, 100, 80);
radial.addColorStop(0, "yellow");
radial.addColorStop(1, "orange");

ctx.fillStyle = radial;
ctx.fillRect(50, 50, 200, 100);
```

This produces a glow effect that radiates outward.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
const radial = ctx.createRadialGradient(150, 100, 20, 150, 100, 80);
radial.addColorStop(0, "yellow");
radial.addColorStop(1, "orange");

ctx.fillStyle = radial;
ctx.fillRect(50, 50, 200, 100);

</script>

</body>
</html>
```

2.3.3 Using Image-Based Patterns

Canvas can fill shapes with repeated images using `createPattern(image, repetition)`.

Steps:

1. Create an `Image` object.
2. Wait for it to load.
3. Use `createPattern()` and assign it to `ctx.fillStyle`.

```
const img = new Image();
img.src = "https://via.placeholder.com/40"; // Example tile

img.onload = () => {
  const pattern = ctx.createPattern(img, "repeat"); // repeat-x, repeat-y, no-repeat
  ctx.fillStyle = pattern;
  ctx.fillRect(0, 0, 300, 150);
};
```

Tip: Use small, tileable images (like textures or icons) for seamless backgrounds.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
const img = new Image();
img.src = "https://via.placeholder.com/40"; // Example tile

img.onload = () => {
  const pattern = ctx.createPattern(img, "repeat"); // repeat-x, repeat-y, no-repeat
  ctx.fillStyle = pattern;
  ctx.fillRect(0, 0, 300, 150);
};

</script>

</body>
</html>
```

2.3.4 Interactive Example: Switching Fill Styles

Let's combine what we've learned into an interactive demo using `<select>` and JavaScript to change the canvas fill style.

HTML:

```
<label>Fill Style:
  <select id="styleSelector">
    <option value="color">Solid Color</option>
    <option value="linear">Linear Gradient</option>
    <option value="radial">Radial Gradient</option>
    <option value="pattern">Pattern</option>
  </select>
</label>
<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>
```

JavaScript:

```
const canvas = document.getElementById("myCanvas");
const ctx = canvas.getContext("2d");
const selector = document.getElementById("styleSelector");

const img = new Image();
img.src = "https://via.placeholder.com/40"; // Example pattern tile

function draw(styleType) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  switch (styleType) {
    case "color":
      ctx.fillStyle = "teal";
      break;
    case "linear":
      const linear = ctx.createLinearGradient(0, 0, canvas.width, 0);
      linear.addColorStop(0, "red");
      linear.addColorStop(1, "yellow");
      ctx.fillStyle = linear;
      break;
    case "radial":
      const radial = ctx.createRadialGradient(200, 100, 20, 200, 100, 100);
      radial.addColorStop(0, "lightblue");
      radial.addColorStop(1, "navy");
      ctx.fillStyle = radial;
      break;
    case "pattern":
      if (img.complete) {
        const pattern = ctx.createPattern(img, "repeat");
        ctx.fillStyle = pattern;
      }
      break;
  }

  ctx.fillRect(0, 0, canvas.width, canvas.height);
```

```
}

selector.addEventListener("change", (e) => draw(e.target.value));
img.onload = () => draw("color"); // Draw default
```

YES Try switching between fill styles to see the effect in real time.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Canvas Fill Style Switcher</title>
  <style>
    body {
      font-family: sans-serif;
      padding: 1rem;
    }
    label {
      display: block;
      margin-bottom: 0.5rem;
    }
  </style>
</head>
<body>

<label>Fill Style:
  <select id="styleSelector">
    <option value="color">Solid Color</option>
    <option value="linear">Linear Gradient</option>
    <option value="radial">Radial Gradient</option>
    <option value="pattern">Pattern</option>
  </select>
</label>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
  const selector = document.getElementById("styleSelector");

  const img = new Image();
  img.src = "https://via.placeholder.com/40"; // Pattern tile

  function draw(styleType) {
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    switch (styleType) {
      case "color":
        ctx.fillStyle = "teal";
        break;
      case "linear":
        const linear = ctx.createLinearGradient(0, 0, canvas.width, 0);
        linear.addColorStop(0, "red");
        linear.addColorStop(1, "yellow");
        ctx.fillStyle = linear;
        break;
```

```
      case "radial":
        const radial = ctx.createRadialGradient(200, 100, 20, 200, 100, 100);
        radial.addColorStop(0, "lightblue");
        radial.addColorStop(1, "navy");
        ctx.fillStyle = radial;
        break;
      case "pattern":
        if (img.complete) {
          const pattern = ctx.createPattern(img, "repeat");
          ctx.fillStyle = pattern;
        } else {
          // wait for image to load
          img.onload = () => draw("pattern");
          return;
        }
        break;
    }

    ctx.fillRect(0, 0, canvas.width, canvas.height);
  }

  selector.addEventListener("change", (e) => draw(e.target.value));
  img.onload = () => draw("color"); // Initial draw
</script>

</body>
</html>
```

2.3.5 Summary

You've now mastered the three primary ways to add visual flair to your canvas drawings:

- **Solid Colors** using `fillStyle` and `strokeStyle`
- **Gradients** (linear and radial) with smooth transitions
- **Patterns** using image tiles and `createPattern()`

You can now bring depth, style, and personality to your canvas artwork. In the next section, we'll take this further by exploring how to **customize stroke and fill styles** in more detail, including transparency, line styles, and advanced composition.

2.4 Working with Stroke and Fill Styles

In canvas drawing, every shape and path can be **filled**, **stroked**, or both. Understanding how these styles work—and how to customize them—lets you add precision and personality to your graphics, from sleek UI components to hand-drawn effects.

This section covers:

- The difference between `fill` and `stroke`
- How to style strokes using width, caps, joins, and dashes
- Real-world examples showing how style changes the character of your drawing

2.4.1 Stroke vs. Fill: Whats the Difference?

- `**fill**`: Applies a color, gradient, or pattern **inside** a shape or path.
- `**stroke**`: Draws an outline **around** the edges of a shape or path.

You can use either **independently** or **together** on the same shape.

Example:

```
ctx.beginPath();
ctx.arc(100, 100, 50, 0, Math.PI * 2); // Circle
ctx.fillStyle = "skyblue";
ctx.fill();    // Fill the circle
ctx.strokeStyle = "navy";
ctx.lineWidth = 5;
ctx.stroke();  // Outline the circle
```

YES This will draw a sky-blue filled circle with a navy border.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.beginPath();
ctx.arc(100, 100, 50, 0, Math.PI * 2); // Circle
ctx.fillStyle = "skyblue";
ctx.fill();    // Fill the circle
ctx.strokeStyle = "navy";
ctx.lineWidth = 5;
ctx.stroke();  // Outline the circle
</script>

</body>
</html>
```

2.4.2 Stroke Styling Options

Canvas provides several properties to control how strokes look. These work together to create everything from smooth curves to sketchy edges.

`lineWidth`

Sets the thickness of the stroke (in pixels):

```
ctx.lineWidth = 8;
```

`lineCap`

Controls how the ends of lines are drawn.

```
ctx.lineCap = "butt";      // (default) square-cut
ctx.lineCap = "round";     // rounded end
ctx.lineCap = "square";    // extends past endpoint
```

Demo:

```
ctx.lineWidth = 10;

["butt", "round", "square"].forEach((cap, i) => {
  ctx.beginPath();
  ctx.moveTo(50, 30 + i * 40);
  ctx.lineTo(250, 30 + i * 40);
  ctx.lineCap = cap;
  ctx.stroke();
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.lineWidth = 10;

["butt", "round", "square"].forEach((cap, i) => {
  ctx.beginPath();
  ctx.moveTo(50, 30 + i * 40);
  ctx.lineTo(250, 30 + i * 40);
  ctx.lineCap = cap;
  ctx.stroke();
```

```
});
</script>

</body>
</html>
```

`lineJoin`

Controls how corners between connected lines are rendered:

```
ctx.lineJoin = "miter";   // (default) sharp corner
ctx.lineJoin = "round";   // rounded corner
ctx.lineJoin = "bevel";   // flattened corner
```

Demo:

```
ctx.lineWidth = 10;

["miter", "round", "bevel"].forEach((join, i) => {
  ctx.beginPath();
  ctx.moveTo(80, 50 + i * 60);
  ctx.lineTo(130, 20 + i * 60);
  ctx.lineTo(180, 50 + i * 60);
  ctx.lineJoin = join;
  ctx.stroke();
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.lineWidth = 10;

["miter", "round", "bevel"].forEach((join, i) => {
  ctx.beginPath();
  ctx.moveTo(80, 50 + i * 60);
  ctx.lineTo(130, 20 + i * 60);
  ctx.lineTo(180, 50 + i * 60);
  ctx.lineJoin = join;
  ctx.stroke();
});
</script>

</body>
```

```
</html>
```

`setLineDash()`

Creates dashed or dotted lines by specifying a pattern of dashes and gaps:

```
ctx.setLineDash([10, 5]);  // 10px dash, 5px gap
ctx.setLineDash([]);       // Reset to solid
```

You can also offset the dashes using `ctx.lineDashOffset`.

Example:

```
ctx.beginPath();
ctx.setLineDash([6, 4]);
ctx.moveTo(50, 200);
ctx.lineTo(250, 200);
ctx.strokeStyle = "darkgreen";
ctx.lineWidth = 4;
ctx.stroke();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.beginPath();
ctx.setLineDash([6, 4]);
ctx.moveTo(50, 200);
ctx.lineTo(250, 200);
ctx.strokeStyle = "darkgreen";
ctx.lineWidth = 4;
ctx.stroke();
</script>

</body>
</html>
```

2.4.3 Combined Stroke Fill

You can apply both `fill()` and `stroke()` to the same shape for enhanced contrast or emphasis.

```
ctx.beginPath();
ctx.rect(50, 50, 150, 100);
ctx.fillStyle = "lightyellow";
ctx.strokeStyle = "orange";
ctx.lineWidth = 5;
ctx.fill();
ctx.stroke();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.beginPath();
ctx.rect(50, 50, 150, 100);
ctx.fillStyle = "lightyellow";
ctx.strokeStyle = "orange";
ctx.lineWidth = 5;
ctx.fill();
ctx.stroke();
</script>

</body>
</html>
```

2.4.4 Style in Practice: From Clean to Sketchy

The way you style strokes can dramatically change the **tone** of your drawing:

Style	Technique Used	Effect
Clean vector shapes	Solid strokes, `lineCap: 'butt'`	UI design, charts
Organic feel	`lineCap: 'round'`, `lineJoin: 'round'`	Friendly, hand-drawn look
Dashed outlines	`setLineDash([5, 5])`	Emphasis, cut-out effect
Emulated sketch lines	Combine dashes with slight offsets	Rough or casual sketches

Mini-Demo: Stylized Trees

```
function drawTree(x, y) {
  ctx.beginPath();
  ctx.moveTo(x, y);
  ctx.lineTo(x, y - 40);
  ctx.lineTo(x - 10, y - 30);
  ctx.lineTo(x + 10, y - 30);
  ctx.lineTo(x, y - 40);
  ctx.closePath();

  ctx.fillStyle = "#228B22";
  ctx.strokeStyle = "#004d00";
  ctx.lineWidth = 3;
  ctx.lineJoin = "round";
  ctx.setLineDash([3, 2]);
  ctx.fill();
  ctx.stroke();
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
function drawTree(x, y) {
  ctx.beginPath();
  ctx.moveTo(x, y);
  ctx.lineTo(x, y - 40);
  ctx.lineTo(x - 10, y - 30);
  ctx.lineTo(x + 10, y - 30);
  ctx.lineTo(x, y - 40);
  ctx.closePath();

  ctx.fillStyle = "#228B22";
  ctx.strokeStyle = "#004d00";
  ctx.lineWidth = 3;
  ctx.lineJoin = "round";
  ctx.setLineDash([3, 2]);
  ctx.fill();
  ctx.stroke();
}

drawTree(100,20);

</script>

</body>
</html>
```

2.4.5 Summary

You've now gained control over how paths look by using:

- `fill()` and `stroke()` for different effects
- `lineWidth`, `lineCap`, and `lineJoin` for stroke shape
- `setLineDash()` for patterns like dashed or dotted lines

These tools help bring clarity and personality to your drawings. Whether you're designing clean UIs or playful illustrations, mastering stroke and fill styling gives your canvas graphics a polished, expressive touch.

Next, we'll put everything together by drawing a complete **colorful house scene** using the techniques from this chapter.

2.5 Practical Example: Drawing a Colorful House

Let's put everything we've learned so far into practice by drawing a **simple cartoon-style house** using the Canvas API. We'll combine **rectangles**, **paths**, **circles**, **gradients**, and even a **pattern** to build a fun, colorful scene. To enhance it further, we'll add **basic interactivity** so the user can change the house color or toggle the door open/closed.

2.5.1 What Well Draw

- **Sky and grass background** using solid and gradient fills.
- **House body** using a rectangle with a customizable color.
- **Roof** using a triangular path.
- **Windows** using rectangles and patterns.
- **Door** with an open/close toggle.
- **Sun** using a filled circle.

We'll also add two buttons:

- **Change Color** – changes the house wall color.
- **Toggle Door** – opens/closes the door.

2.5.2 HTML Canvas Setup

```
<div class="controls">
  <button id="colorBtn">Change House Color</button>
  <button id="doorBtn">Toggle Door</button>
</div>
<canvas id="houseCanvas" width="600" height="400"></canvas>
```

Javascript:

```
  const canvas = document.getElementById("houseCanvas");
  const ctx = canvas.getContext("2d");

  // State variables
  const houseColors = ["#FFB6C1", "#D8BFD8", "#ADD8E6", "#F4A460"];
  let currentColorIndex = 0;
  let doorOpen = false;

  function drawScene() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    // --- Background: sky and grass ---
    const skyGradient = ctx.createLinearGradient(0, 0, 0, canvas.height);
    skyGradient.addColorStop(0, "#87CEFA");
    skyGradient.addColorStop(1, "#ffffff");
    ctx.fillStyle = skyGradient;
    ctx.fillRect(0, 0, canvas.width, canvas.height);

    ctx.fillStyle = "#7CFC00"; // Grass green
    ctx.fillRect(0, 300, canvas.width, 100);

    // --- Sun ---
    ctx.beginPath();
    ctx.arc(520, 80, 40, 0, Math.PI * 2);
    ctx.fillStyle = "yellow";
    ctx.fill();

    // --- House Walls ---
    ctx.fillStyle = houseColors[currentColorIndex];
    ctx.fillRect(200, 180, 200, 120);

    // --- Roof ---
    ctx.beginPath();
    ctx.moveTo(180, 180);  // Left point
    ctx.lineTo(300, 100);  // Top point
    ctx.lineTo(420, 180);  // Right point
    ctx.closePath();
    ctx.fillStyle = "#8B0000";  // Dark red
    ctx.fill();

    // --- Door ---
    ctx.fillStyle = "#654321";  // Brown
    if (doorOpen) {
      ctx.beginPath();
      ctx.moveTo(280, 300);
      ctx.lineTo(280, 240);
```

```
        ctx.lineTo(260, 250);
        ctx.lineTo(260, 300);
        ctx.closePath();
        ctx.fill();
      } else {
        ctx.fillRect(270, 240, 40, 60);  // Closed door
      }

      // Doorknob
      ctx.beginPath();
      ctx.arc(300, 270, 4, 0, 2 * Math.PI);
      ctx.fillStyle = "gold";
      ctx.fill();

      // --- Windows ---
      ctx.fillStyle = "#ffffffcc";  // Semi-transparent white
      ctx.strokeStyle = "#444";
      ctx.lineWidth = 2;
      drawWindow(220, 200);
      drawWindow(340, 200);
    }

    function drawWindow(x, y) {
      ctx.fillRect(x, y, 30, 30);
      ctx.strokeRect(x, y, 30, 30);
      ctx.beginPath();
      ctx.moveTo(x + 15, y);
      ctx.lineTo(x + 15, y + 30);
      ctx.moveTo(x, y + 15);
      ctx.lineTo(x + 30, y + 15);
      ctx.stroke();
    }

    // --- Interactivity ---
    document.getElementById("colorBtn").addEventListener("click", () => {
      currentColorIndex = (currentColorIndex + 1) % houseColors.length;
      drawScene();
    });

    document.getElementById("doorBtn").addEventListener("click", () => {
      doorOpen = !doorOpen;
      drawScene();
    });

    drawScene();  // Initial draw
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Canvas House Example</title>
  <style>
    canvas {
      border: 1px solid #ccc;
      display: block;
      margin: 20px auto;
      background-color: #e0f7ff;
```

```
    }
    .controls {
      text-align: center;
      margin-bottom: 10px;
    }
  </style>
</head>
<body>
  <div class="controls">
    <button id="colorBtn">Change House Color</button>
    <button id="doorBtn">Toggle Door</button>
  </div>
  <canvas id="houseCanvas" width="600" height="400"></canvas>

  <script>
    const canvas = document.getElementById("houseCanvas");
    const ctx = canvas.getContext("2d");

    // State variables
    const houseColors = ["#FFB6C1", "#D8BFD8", "#ADD8E6", "#F4A460"];
    let currentColorIndex = 0;
    let doorOpen = false;

    function drawScene() {
      ctx.clearRect(0, 0, canvas.width, canvas.height);

      // --- Background: sky and grass ---
      const skyGradient = ctx.createLinearGradient(0, 0, 0, canvas.height);
      skyGradient.addColorStop(0, "#87CEFA");
      skyGradient.addColorStop(1, "#ffffff");
      ctx.fillStyle = skyGradient;
      ctx.fillRect(0, 0, canvas.width, canvas.height);

      ctx.fillStyle = "#7CFC00"; // Grass green
      ctx.fillRect(0, 300, canvas.width, 100);

      // --- Sun ---
      ctx.beginPath();
      ctx.arc(520, 80, 40, 0, Math.PI * 2);
      ctx.fillStyle = "yellow";
      ctx.fill();

      // --- House Walls ---
      ctx.fillStyle = houseColors[currentColorIndex];
      ctx.fillRect(200, 180, 200, 120);

      // --- Roof ---
      ctx.beginPath();
      ctx.moveTo(180, 180);  // Left point
      ctx.lineTo(300, 100);  // Top point
      ctx.lineTo(420, 180);  // Right point
      ctx.closePath();
      ctx.fillStyle = "#8B0000";  // Dark red
      ctx.fill();

      // --- Door ---
      ctx.fillStyle = "#654321";  // Brown
      if (doorOpen) {
```

```
        ctx.beginPath();
        ctx.moveTo(280, 300);
        ctx.lineTo(280, 240);
        ctx.lineTo(260, 250);
        ctx.lineTo(260, 300);
        ctx.closePath();
        ctx.fill();
      } else {
        ctx.fillRect(270, 240, 40, 60);  // Closed door
      }

      // Doorknob
      ctx.beginPath();
      ctx.arc(300, 270, 4, 0, 2 * Math.PI);
      ctx.fillStyle = "gold";
      ctx.fill();

      // --- Windows ---
      ctx.fillStyle = "#ffffffcc";  // Semi-transparent white
      ctx.strokeStyle = "#444";
      ctx.lineWidth = 2;
      drawWindow(220, 200);
      drawWindow(340, 200);
    }

    function drawWindow(x, y) {
      ctx.fillRect(x, y, 30, 30);
      ctx.strokeRect(x, y, 30, 30);
      ctx.beginPath();
      ctx.moveTo(x + 15, y);
      ctx.lineTo(x + 15, y + 30);
      ctx.moveTo(x, y + 15);
      ctx.lineTo(x + 30, y + 15);
      ctx.stroke();
    }

    // --- Interactivity ---
    document.getElementById("colorBtn").addEventListener("click", () => {
      currentColorIndex = (currentColorIndex + 1) % houseColors.length;
      drawScene();
    });

    document.getElementById("doorBtn").addEventListener("click", () => {
      doorOpen = !doorOpen;
      drawScene();
    });

    drawScene();  // Initial draw
  </script>
</body>
</html>
```

2.5.3 Code Breakdown

Part	Description
Background	Uses a vertical linear gradient for the sky and a solid green base for grass.
House Walls	A colored rectangle with dynamic color.
Roof	A triangle path connecting three points.
Door	Changes shape depending on open/closed state.
Windows	Drawn with helper `drawWindow()` for reuse, including crossbars.
Sun	A bright yellow circle in the top-right.
Controls	Two buttons for interactivity (change color and toggle door).

2.5.4 Creative Extensions

Try expanding this demo with your own ideas:

- Add **a chimney** using a smaller rectangle on the roof.
- Draw **bushes or clouds** using overlapping circles.
- Let users choose colors with an `<input type="color">`.
- Animate the **sun moving across the sky** over time.
- Add a **night mode toggle** (switch sky and sun to stars/moon).

2.5.5 Summary

You've now drawn a complete cartoon-style house by combining shapes, colors, gradients, and interactivity. This exercise ties together the skills from the entire chapter and gives you a solid foundation for more complex scenes in future chapters—including animation and interactive graphics.

Up next, we'll step into **intermediate canvas techniques** and begin exploring text rendering, transformations, and layering strategies to add depth and motion to your visuals.

Chapter 3.

Working with Text on Canvas

1. Drawing Text with `fillText()` and `strokeText()`
2. Setting Font Styles and Alignments
3. Measuring Text Metrics
4. Practical Example: Dynamic Text Rendering and Animations

3 Working with Text on Canvas

3.1 Drawing Text with `fillText()` and `strokeText()`

Canvas lets you render text in two main ways: **filled text** and **outlined text**. These are handled by the methods `fillText()` and `strokeText()` respectively. Understanding how to position and style text on the canvas will help you add clear labels, headings, or even artistic typography to your graphics.

3.1.1 Using `fillText()` to Draw Filled Text

The `fillText()` method draws **solid, filled** text at the specified position.

Syntax:

```
ctx.fillText(text, x, y [, maxWidth]);
```

- `text`: The string to render.
- `x`, `y`: Coordinates where the text baseline starts.
- `maxWidth` (optional): Max width to scale text if necessary.

3.1.2 Using `strokeText()` to Draw Outlined Text

The `strokeText()` method draws text outlines—useful for creating hollow or outlined letters.

Syntax:

```
ctx.strokeText(text, x, y [, maxWidth]);
```

Parameters are the same as `fillText()`.

3.1.3 Text Positioning: Baseline and Alignment

Text is drawn relative to a **baseline** and horizontally aligned according to these properties:

- `ctx.textBaseline` (default is `"alphabetic"`) controls vertical alignment.
  - Options include: `"top"`, `"middle"`, `"bottom"`, `"alphabetic"`, `"hanging"`.

- `ctx.textAlign` (default is `"start"`) controls horizontal alignment.
  - Options include: `"start"`, `"end"`, `"left"`, `"right"`, `"center"`.

Example: Positioning Text

```
ctx.font = "24px Arial";
ctx.textBaseline = "middle";
ctx.textAlign = "center";

ctx.fillText("Hello Canvas", canvas.width / 2, canvas.height / 2);
```

This draws the text centered both horizontally and vertically at the canvas center.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.font = "24px Arial";
ctx.textBaseline = "middle";
ctx.textAlign = "center";

ctx.fillText("Hello Canvas", canvas.width / 2, canvas.height / 2);
</script>

</body>
</html>
```

3.1.4 Visual Differences and Use Cases

Method	Appearance	Use Case
`fillText()`	Solid, filled letters	Body text, labels, headings
`strokeText()`	Outlined letters	Stylish effects, text shadows, emphasis

You can combine both to create text with colored fills and contrasting outlines:

```
ctx.fillStyle = "yellow";
ctx.strokeStyle = "black";
```

```
ctx.lineWidth = 2;

ctx.fillText("Canvas", 150, 100);
ctx.strokeText("Canvas", 150, 100);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
ctx.fillStyle = "yellow";
ctx.strokeStyle = "black";
ctx.lineWidth = 2;

ctx.fillText("Canvas", 150, 100);
ctx.strokeText("Canvas", 150, 100);

</script>

</body>
</html>
```

3.1.5 Example: Short Label and Multiline Text

Canvas does **not** support automatic multiline text. To draw multiple lines, you'll need to split text and position each line manually.

```
const lines = [
  "This is line 1",
  "Here is line 2",
  "And finally line 3"
];

ctx.font = "18px Verdana";
ctx.fillStyle = "navy";
ctx.textBaseline = "top";
ctx.textAlign = "left";

const startX = 50;
const startY = 50;
const lineHeight = 24;

lines.forEach((line, i) => {
  ctx.fillText(line, startX, startY + i * lineHeight);
```

```
});
```

3.1.6 Complete Example

```
<script>
  const canvas = document.getElementById("textCanvas");
  const ctx = canvas.getContext("2d");

  // Filled text
  ctx.font = "30px Arial";
  ctx.fillStyle = "blue";
  ctx.textBaseline = "top";
  ctx.textAlign = "left";
  ctx.fillText("Filled Text", 20, 20);

  // Outlined text
  ctx.font = "40px Georgia";
  ctx.strokeStyle = "red";
  ctx.lineWidth = 2;
  ctx.textBaseline = "alphabetic";
  ctx.textAlign = "center";
  ctx.strokeText("Outlined Text", canvas.width / 2, 150);

  // Combined fill + stroke
  ctx.font = "50px Verdana";
  ctx.fillStyle = "yellow";
  ctx.strokeStyle = "black";
  ctx.lineWidth = 3;
  ctx.textBaseline = "bottom";
  ctx.textAlign = "right";
  ctx.fillText("Fill & Stroke", canvas.width - 20, canvas.height - 10);
  ctx.strokeText("Fill & Stroke", canvas.width - 20, canvas.height - 10);
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
  // Filled text
  ctx.font = "30px Arial";
  ctx.fillStyle = "blue";
  ctx.textBaseline = "top";
  ctx.textAlign = "left";
```

```
  ctx.fillText("Filled Text", 20, 20);

  // Outlined text
  ctx.font = "40px Georgia";
  ctx.strokeStyle = "red";
  ctx.lineWidth = 2;
  ctx.textBaseline = "alphabetic";
  ctx.textAlign = "center";
  ctx.strokeText("Outlined Text", canvas.width / 2, 150);

  // Combined fill + stroke
  ctx.font = "50px Verdana";
  ctx.fillStyle = "yellow";
  ctx.strokeStyle = "black";
  ctx.lineWidth = 3;
  ctx.textBaseline = "bottom";
  ctx.textAlign = "right";
  ctx.fillText("Fill & Stroke", canvas.width - 20, canvas.height - 10);
  ctx.strokeText("Fill & Stroke", canvas.width - 20, canvas.height - 10);

</script>

</body>
</html>
```

3.1.7 Summary

- Use **`fillText()`** for solid, readable text and **`strokeText()`** for outlines and decorative effects.
- Position text precisely with `textBaseline` and `textAlign`.
- To draw multiline text, manually position each line with a consistent vertical offset.
- Combine fill and stroke for impactful typography.

Next, we'll explore how to customize font styles, sizes, and alignments to make your canvas text look exactly how you want it.

3.2 Setting Font Styles and Alignments

Styling text on canvas involves choosing the right **font**, **size**, **weight**, and **alignment**. The Canvas API gives you full control over these properties so you can position your text precisely and match your design's look and feel.

3.2.1 Setting Fonts with the `font` Property

The `font` property works similarly to CSS `font` shorthand. You can set:

- Font style (optional): `normal`, `italic`, `oblique`
- Font weight (optional): `normal`, `bold`, `lighter`, `100`–`900`
- Font size (required): with units like `px`, `em`, `%`
- Font family (required): such as `"Arial"`, `"Verdana"`, `"Times New Roman"`, or generic families like `"serif"` or `"sans-serif"`

3.2.2 Syntax:

```
ctx.font = "[font-style] [font-weight] [font-size] [font-family]";
```

3.2.3 Examples:

```
ctx.font = "normal normal 24px Arial";  // Regular 24px Arial
ctx.font = "italic bold 30px 'Times New Roman'";  // Italic, bold 30px Times
ctx.font = "normal 18px Verdana";  // 18px Verdana, default weight/style
```

You can omit optional parts, as long as the font size and family are included.

3.2.4 Text Alignment: `textAlign` and `textBaseline`

The way text is **positioned** on canvas depends on two properties:

3.2.5 `textAlign`

Controls horizontal alignment relative to the `x` coordinate where text is drawn.

Value	Description	Effect on drawing position
`"left"`	Text starts at the x-coordinate	Left edge aligns with x
`"right"`	Text ends at the x-coordinate	Right edge aligns with x
`"center"`	Text is centered on the x-coordinate	Center aligns with x
`"start"`	Same as left for LTR, right for RTL	Depends on text direction

Value	Description	Effect on drawing position
`"end"`	Opposite of start	Depends on text direction

3.2.6 `textBaseline`

Controls vertical alignment relative to the `y` coordinate.

Value	Description	Visual position relative to y
`"top"`	Text top aligns with y	Top of text is at y
`"hanging"`	Slightly below top	Based on language script metrics
`"middle"`	Vertical center aligns with y	Middle of text is at y
`"alphabetic"`	Baseline of text aligns with y	Most common, baseline of characters
`"ideographic"`	Bottom of ideographs aligns with y	East Asian scripts
`"bottom"`	Bottom of text aligns with y	Bottom edge of text at y

3.2.7 Visual Examples

Let's see how different alignments affect the placement of text on the canvas.

```
<canvas id="alignCanvas" width="400" height="200" style="border:1px solid #ccc"></canvas>
<script>
  const canvas = document.getElementById("alignCanvas");
  const ctx = canvas.getContext("2d");

  ctx.font = "20px Arial";
  ctx.strokeStyle = "gray";
  ctx.lineWidth = 1;

  // Draw reference lines for center
  const x = canvas.width / 2;
  const y = canvas.height / 2;
  ctx.beginPath();
  ctx.moveTo(x, 0);
  ctx.lineTo(x, canvas.height);
  ctx.moveTo(0, y);
  ctx.lineTo(canvas.width, y);
  ctx.stroke();

  const alignments = ["left", "center", "right"];
  const baselines = ["top", "middle", "alphabetic", "bottom"];

  alignments.forEach((align, i) => {
    baselines.forEach((baseline, j) => {
      const posX = 100 + i * 120;
```

```
      const posY = 30 + j * 40;

      ctx.textAlign = align;
      ctx.textBaseline = baseline;
      ctx.fillStyle = "black";
      ctx.fillText(`A`, posX, posY);

      // Label with alignment info
      ctx.fillStyle = "gray";
      ctx.font = "10px Arial";
      ctx.fillText(`${align} / ${baseline}`, posX, posY + 15);
      ctx.font = "20px Arial";
    });
  });
</script>
```

The gray lines show the canvas center. The letter “A” is drawn multiple times with different alignments and baselines. Notice how the position shifts depending on these settings.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

  ctx.font = "20px Arial";
  ctx.strokeStyle = "gray";
  ctx.lineWidth = 1;

  // Draw reference lines for center
  const x = canvas.width / 2;
  const y = canvas.height / 2;
  ctx.beginPath();
  ctx.moveTo(x, 0);
  ctx.lineTo(x, canvas.height);
  ctx.moveTo(0, y);
  ctx.lineTo(canvas.width, y);
  ctx.stroke();

  const alignments = ["left", "center", "right"];
  const baselines = ["top", "middle", "alphabetic", "bottom"];

  alignments.forEach((align, i) => {
    baselines.forEach((baseline, j) => {
      const posX = 100 + i * 120;
      const posY = 30 + j * 40;

      ctx.textAlign = align;
      ctx.textBaseline = baseline;
```

```
      ctx.fillStyle = "black";
      ctx.fillText(`A`, posX, posY);

      // Label with alignment info
      ctx.fillStyle = "gray";
      ctx.font = "10px Arial";
      ctx.fillText(`${align} / ${baseline}`, posX, posY + 15);
      ctx.font = "20px Arial";
    });
  });
</script>

</body>
</html>
```

3.2.8 Try Your Own: Draw Labels on Canvas Corners

Try drawing text labels aligned to each corner and center of the canvas:

```
ctx.font = "18px Verdana";
ctx.fillStyle = "darkblue";

// Top-left corner
ctx.textAlign = "left";
ctx.textBaseline = "top";
ctx.fillText("Top Left", 0, 0);

// Top-center
ctx.textAlign = "center";
ctx.textBaseline = "top";
ctx.fillText("Top Center", canvas.width / 2, 0);

// Top-right
ctx.textAlign = "right";
ctx.textBaseline = "top";
ctx.fillText("Top Right", canvas.width, 0);

// Middle-left
ctx.textAlign = "left";
ctx.textBaseline = "middle";
ctx.fillText("Middle Left", 0, canvas.height / 2);

// Center
ctx.textAlign = "center";
ctx.textBaseline = "middle";
ctx.fillText("Center", canvas.width / 2, canvas.height / 2);

// Middle-right
ctx.textAlign = "right";
ctx.textBaseline = "middle";
ctx.fillText("Middle Right", canvas.width, canvas.height / 2);

// Bottom-left
```

```
ctx.textAlign = "left";
ctx.textBaseline = "bottom";
ctx.fillText("Bottom Left", 0, canvas.height);

// Bottom-center
ctx.textAlign = "center";
ctx.textBaseline = "bottom";
ctx.fillText("Bottom Center", canvas.width / 2, canvas.height);

// Bottom-right
ctx.textAlign = "right";
ctx.textBaseline = "bottom";
ctx.fillText("Bottom Right", canvas.width, canvas.height);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.font = "18px Verdana";
ctx.fillStyle = "darkblue";

// Top-left corner
ctx.textAlign = "left";
ctx.textBaseline = "top";
ctx.fillText("Top Left", 0, 0);

// Top-center
ctx.textAlign = "center";
ctx.textBaseline = "top";
ctx.fillText("Top Center", canvas.width / 2, 0);

// Top-right
ctx.textAlign = "right";
ctx.textBaseline = "top";
ctx.fillText("Top Right", canvas.width, 0);

// Middle-left
ctx.textAlign = "left";
ctx.textBaseline = "middle";
ctx.fillText("Middle Left", 0, canvas.height / 2);

// Center
ctx.textAlign = "center";
ctx.textBaseline = "middle";
ctx.fillText("Center", canvas.width / 2, canvas.height / 2);

// Middle-right
ctx.textAlign = "right";
```

```
ctx.textBaseline = "middle";
ctx.fillText("Middle Right", canvas.width, canvas.height / 2);

// Bottom-left
ctx.textAlign = "left";
ctx.textBaseline = "bottom";
ctx.fillText("Bottom Left", 0, canvas.height);

// Bottom-center
ctx.textAlign = "center";
ctx.textBaseline = "bottom";
ctx.fillText("Bottom Center", canvas.width / 2, canvas.height);

// Bottom-right
ctx.textAlign = "right";
ctx.textBaseline = "bottom";
ctx.fillText("Bottom Right", canvas.width, canvas.height);
</script>

</body>
</html>
```

3.2.9 Summary

- Use the `font` property to set the **font family**, **size**, and **weight** (like CSS shorthand).
- Control **horizontal** text alignment with `textAlign`: `"left"`, `"right"`, `"center"`, `"start"`, `"end"`.
- Control **vertical** positioning with `textBaseline`: `"top"`, `"middle"`, `"alphabetic"`, `"bottom"`, and others.
- Experiment with these to position text precisely on canvas for labels, headings, or UI elements.

In the next section, we'll explore how to measure text size and metrics, enabling you to build perfectly aligned and dynamically sized text layouts.

3.3 Measuring Text Metrics

When working with text on the canvas, knowing the **exact dimensions** of your text is crucial for precise layout, alignment, truncation, or wrapping. The Canvas API provides a handy method called `measureText()` that helps you get these measurements.

3.3.1 What is `measureText()`?

The method `ctx.measureText(text)` returns a **TextMetrics** object containing measurements about how the specified text will be rendered with the current font settings.

3.3.2 Why is it important?

- **Centering text:** Knowing the width lets you position text exactly in the middle of an area.
- **Fitting text:** You can truncate or resize text to fit inside a box.
- **Wrapping text:** Helps determine when to break lines in long paragraphs.
- **Highlighting:** You can draw bounding boxes or backgrounds around the text.

3.3.3 Basic Usage: Measuring Text Width

```
ctx.font = "20px Arial";
const text = "Hello, Canvas!";
const metrics = ctx.measureText(text);
console.log("Text width:", metrics.width);
```

The `.width` property gives the width in pixels that the text will occupy.

3.3.4 Example: Centering Text Using `measureText()`

```
const canvasWidth = canvas.width;
const text = "Centered Text";
ctx.font = "24px Verdana";
ctx.fillStyle = "black";

const textWidth = ctx.measureText(text).width;
const x = (canvasWidth - textWidth) / 2;  // center horizontally

ctx.fillText(text, x, 50);
```

3.3.5 Visualizing Text Metrics with Bounding Boxes

Besides `.width`, the **TextMetrics** object has other useful properties (supported in many browsers):

- `actualBoundingBoxLeft`
- `actualBoundingBoxRight`
- `actualBoundingBoxAscent`
- `actualBoundingBoxDescent`

These let you draw an accurate bounding rectangle around the text.

3.3.6 Demo: Draw bounding box around text

```
const text = "Bounding Box Demo";
ctx.font = "30px Arial";
ctx.fillStyle = "blue";

const x = 50;
const y = 100;
ctx.fillText(text, x, y);

const metrics = ctx.measureText(text);

ctx.strokeStyle = "red";
ctx.lineWidth = 1;
ctx.strokeRect(
  x + metrics.actualBoundingBoxLeft,
  y - metrics.actualBoundingBoxAscent,
  metrics.actualBoundingBoxLeft + metrics.actualBoundingBoxRight,
  metrics.actualBoundingBoxAscent + metrics.actualBoundingBoxDescent
);
```

This code draws a red rectangle exactly around the rendered text.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

const text = "Bounding Box Demo";
ctx.font = "30px Arial";
ctx.fillStyle = "blue";
```

```
const x = 50;
const y = 100;
ctx.fillText(text, x, y);

const metrics = ctx.measureText(text);

ctx.strokeStyle = "red";
ctx.lineWidth = 1;
ctx.strokeRect(
  x + metrics.actualBoundingBoxLeft,
  y - metrics.actualBoundingBoxAscent,
  metrics.actualBoundingBoxLeft + metrics.actualBoundingBoxRight,
  metrics.actualBoundingBoxAscent + metrics.actualBoundingBoxDescent
);
</script>

</body>
</html>
```

3.3.7 Wrapping Long Text Using `measureText()`

Canvas doesn't support automatic text wrapping, but you can implement it yourself by measuring text and breaking it into lines that fit a given width.

3.3.8 Example: Simple text wrap function

```
function wrapText(ctx, text, x, y, maxWidth, lineHeight) {
  const words = text.split(" ");
  let line = "";
  let lineCount = 0;

  for (let n = 0; n < words.length; n++) {
    const testLine = line + words[n] + " ";
    const testWidth = ctx.measureText(testLine).width;

    if (testWidth > maxWidth && n > 0) {
      ctx.fillText(line, x, y + lineCount * lineHeight);
      line = words[n] + " ";
      lineCount++;
    } else {
      line = testLine;
    }
  }
  ctx.fillText(line, x, y + lineCount * lineHeight);
}
```

3.3.9 Usage:

```
ctx.font = "16px Georgia";
ctx.fillStyle = "#333";

const text = "This is a long sentence that will automatically wrap inside a specified width using measu
wrapText(ctx, text, 20, 50, 300, 22);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

function wrapText(ctx, text, x, y, maxWidth, lineHeight) {
  const words = text.split(" ");
  let line = "";
  let lineCount = 0;

  for (let n = 0; n < words.length; n++) {
    const testLine = line + words[n] + " ";
    const testWidth = ctx.measureText(testLine).width;

    if (testWidth > maxWidth && n > 0) {
      ctx.fillText(line, x, y + lineCount * lineHeight);
      line = words[n] + " ";
      lineCount++;
    } else {
      line = testLine;
    }
  }
  ctx.fillText(line, x, y + lineCount * lineHeight);
}

ctx.font = "16px Georgia";
ctx.fillStyle = "#333";

const text = "This is a long sentence that will automatically wrap inside a specified width using measu
wrapText(ctx, text, 20, 50, 300, 22);
</script>

</body>
</html>
```

3.3.10 Adaptive Labels: Truncating Text with Ellipsis

You can also **truncate** text to fit inside a given width by repeatedly measuring and shortening the string:

```
function truncateText(ctx, text, maxWidth) {
  let truncated = text;
  while (ctx.measureText(truncated + "...").width > maxWidth && truncated.length > 0) {
    truncated = truncated.slice(0, -1);
  }
  return truncated + (truncated.length < text.length ? "..." : "");
}

ctx.font = "18px Arial";
ctx.fillStyle = "black";

const label = "Very long label text that might not fit";
const maxWidth = 150;
const displayText = truncateText(ctx, label, maxWidth);

ctx.fillText(displayText, 20, 150);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

function truncateText(ctx, text, maxWidth) {
  let truncated = text;
  while (ctx.measureText(truncated + "...").width > maxWidth && truncated.length > 0) {
    truncated = truncated.slice(0, -1);
  }
  return truncated + (truncated.length < text.length ? "..." : "");
}

ctx.font = "18px Arial";
ctx.fillStyle = "black";

const label = "Very long label text that might not fit";
const maxWidth = 150;
const displayText = truncateText(ctx, label, maxWidth);

ctx.fillText(displayText, 20, 150);
</script>

</body>
</html>
```

3.3.11 Summary

- Use `ctx.measureText(text)` to get the width and detailed metrics of your text.
- The `.width` property helps you center or constrain text dynamically.
- Bounding box metrics allow you to draw precise backgrounds or hit areas around text.
- Implement **text wrapping** by measuring words and breaking lines manually.
- Truncate overly long text gracefully by measuring and adding ellipses.

With these tools, you can create dynamic, well-aligned, and visually appealing text layouts on canvas.

Next, we'll use these measurement techniques to build **dynamic text rendering and animations** in a practical example.

3.4 Practical Example: Dynamic Text Rendering and Animations

In this section, we'll create an animated text example that combines everything we've learned so far:

- Drawing text with **font styles**
- Using **text measurements** for positioning
- Animating with a **frame update loop**
- Adding **interactivity** to change messages and styles dynamically

3.4.1 What We'll Build

A canvas displaying a **color-changing, bouncing text** that moves horizontally across the screen. Users can type their own message and select font size and color from input controls.

3.4.2 Complete HTML JavaScript Code

```
<div class="controls">
  <input id="textInput" type="text" value="Hello Canvas!" placeholder="Enter your text" />
  <input id="colorInput" type="color" value="#0077cc" title="Choose text color" />
  <input id="fontSizeInput" type="number" min="12" max="72" value="36" title="Font size (px)" />
</div>

<canvas id="animCanvas" width="600" height="150"></canvas>
```

Javascript:

```
<script>
  const canvas = document.getElementById('animCanvas');
  const ctx = canvas.getContext('2d');

  // Controls
  const textInput = document.getElementById('textInput');
  const colorInput = document.getElementById('colorInput');
  const fontSizeInput = document.getElementById('fontSizeInput');

  // State
  let message = textInput.value;
  let color = colorInput.value;
  let fontSize = parseInt(fontSizeInput.value, 10);

  let x = 0;
  let speed = 2; // pixels per frame
  let direction = 1; // 1 = right, -1 = left

  // Animate function called 60 times per second
  function animate() {
    // Clear the entire canvas for new frame
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    // Set font with dynamic size
    ctx.font = `${fontSize}px Verdana`;
    ctx.fillStyle = color;
    ctx.textBaseline = 'middle';

    // Measure text width for bouncing effect
    const textWidth = ctx.measureText(message).width;
    const y = canvas.height / 2;

    // Draw text at current x position
    ctx.fillText(message, x, y);

    // Update x position for next frame
    x += speed * direction;

    // Bounce back when reaching canvas edges
    if (x + textWidth > canvas.width) {
      direction = -1;  // Move left
    }
    if (x < 0) {
      direction = 1;   // Move right
    }

    // Request next animation frame
    requestAnimationFrame(animate);
  }

  // Start animation loop
  animate();

  // Update state on input changes
  textInput.addEventListener('input', () => {
```

```
      message = textInput.value || " ";
      x = 0;  // Reset position for new text
    });

    colorInput.addEventListener('input', () => {
      color = colorInput.value;
    });

    fontSizeInput.addEventListener('input', () => {
      const size = parseInt(fontSizeInput.value, 10);
      if (!isNaN(size) && size >= 12 && size <= 72) {
        fontSize = size;
        x = 0;  // Reset position to avoid clipping
      }
    });
  </script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Canvas Text Animation</title>
  <style>
    body {
      text-align: center;
      font-family: Arial, sans-serif;
      margin: 20px;
    }
    canvas {
      border: 1px solid #ccc;
      display: block;
      margin: 0 auto 20px auto;
      background: #f0f0f0;
    }
    .controls > * {
      margin: 5px;
      font-size: 16px;
    }
  </style>
</head>
<body>

  <div class="controls">
    <input id="textInput" type="text" value="Hello Canvas!" placeholder="Enter your text" />
    <input id="colorInput" type="color" value="#0077cc" title="Choose text color" />
    <input id="fontSizeInput" type="number" min="12" max="72" value="36" title="Font size (px)" />
  </div>

  <canvas id="animCanvas" width="600" height="150"></canvas>

  <script>
    const canvas = document.getElementById('animCanvas');
    const ctx = canvas.getContext('2d');

    // Controls
    const textInput = document.getElementById('textInput');
    const colorInput = document.getElementById('colorInput');
```

```
const fontSizeInput = document.getElementById('fontSizeInput');

// State
let message = textInput.value;
let color = colorInput.value;
let fontSize = parseInt(fontSizeInput.value, 10);

let x = 0;
let speed = 2; // pixels per frame
let direction = 1; // 1 = right, -1 = left

// Animate function called 60 times per second
function animate() {
  // Clear the entire canvas for new frame
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Set font with dynamic size
  ctx.font = `${fontSize}px Verdana`;
  ctx.fillStyle = color;
  ctx.textBaseline = 'middle';

  // Measure text width for bouncing effect
  const textWidth = ctx.measureText(message).width;
  const y = canvas.height / 2;

  // Draw text at current x position
  ctx.fillText(message, x, y);

  // Update x position for next frame
  x += speed * direction;

  // Bounce back when reaching canvas edges
  if (x + textWidth > canvas.width) {
    direction = -1;  // Move left
  }
  if (x < 0) {
    direction = 1;   // Move right
  }

  // Request next animation frame
  requestAnimationFrame(animate);
}

// Start animation loop
animate();

// Update state on input changes
textInput.addEventListener('input', () => {
  message = textInput.value || " ";
  x = 0;  // Reset position for new text
});

colorInput.addEventListener('input', () => {
  color = colorInput.value;
});

fontSizeInput.addEventListener('input', () => {
  const size = parseInt(fontSizeInput.value, 10);
```

```
      if (!isNaN(size) && size >= 12 && size <= 72) {
        fontSize = size;
        x = 0;  // Reset position to avoid clipping
      }
    });
  </script>
</body>
</html>
```

3.4.3 How It Works

Frame Updates and Clearing

- Every frame, we clear the canvas using `ctx.clearRect()` to erase the previous frame.
- Then we draw the updated text at the new `x` coordinate.

Animation Loop

- The `animate()` function uses `requestAnimationFrame()` for smooth 60fps animation.
- `x` position is updated by `speed * direction` to move the text horizontally.
- When the text reaches the canvas edges, the `direction` reverses to create a bounce effect.

Measuring Text

- We use `ctx.measureText(message).width` to know the text's width.
- This ensures the bounce reverses precisely at edges so text never disappears outside the canvas.

Interactivity

- Inputs for message, color, and font size allow dynamic changes.
- Event listeners update the state immediately and reset the `x` position so new text starts fresh.

3.4.4 Try Your Own

- Change the **speed** variable to control movement pace.
- Add vertical bouncing by modifying the `y` position similarly.
- Animate colors by gradually changing RGB or using `hsl()` color rotation.
- Add shadow or outline effects with `ctx.shadowColor` and `ctx.strokeText()`.

3.4.5 Summary

This example combines:

- **Text drawing** with `fillText()`
- **Font styling** via the `font` property
- **Text measurement** for layout and bouncing boundaries
- **Animation loop** with `requestAnimationFrame()`
- **User interactivity** to change content and styles dynamically

These techniques are the foundation for engaging, animated, and interactive canvas text effects. Next, we'll explore advanced text transformations and integrating text with other canvas graphics.

Chapter 4.

Paths and Complex Shapes

1. Creating Paths with `beginPath()`, `moveTo()`, and `lineTo()`
2. Drawing Curves and Bezier Paths
3. Using `closePath()` and Filling Complex Shapes
4. Practical Example: Drawing a Star or Custom Shape

4 Paths and Complex Shapes

4.1 Creating Paths with `beginPath()`, `moveTo()`, and `lineTo()`

When drawing on the canvas, simple shapes like rectangles and circles are just the beginning. For **custom shapes and intricate designs**, the Canvas API offers a powerful way to define **paths**—a sequence of points connected by lines or curves.

4.1.1 What is a Path?

A **path** is a series of connected points that the canvas can stroke (outline) or fill. Paths let you create any shape, from simple polygons to complex figures, by specifying exactly how lines connect and flow.

4.1.2 Resetting Paths with `beginPath()`

Before you start drawing a new shape, you call:

```
ctx.beginPath();
```

This clears the current path and tells the canvas to start fresh. Without it, new drawing commands might connect to the previous shape unexpectedly.

4.1.3 Moving the Pen with `moveTo()`

`moveTo(x, y)` moves the drawing cursor to a new starting position **without** drawing anything. Think of it as lifting the pen and placing it somewhere else on the paper.

Example:

```
ctx.moveTo(50, 50);
```

This moves the "pen" to coordinates (50, 50).

4.1.4 Drawing Lines with `lineTo()`

After moving the pen, `lineTo(x, y)` draws a straight line from the current position to the new point (x, y).

Example:

```
ctx.lineTo(150, 50);
```

Draws a line from the current point to (150, 50).

4.1.5 Putting It All Together: Drawing Paths

By combining `moveTo()` and multiple `lineTo()` calls, you can build **open** or **closed** shapes:

- **Open path:** Lines connect sequential points but do not automatically close.
- **Closed path:** You manually connect the last point back to the first (or use `closePath()` to close the shape).

4.1.6 Example 1: Drawing a Triangle (Closed Polygon)

```
ctx.beginPath();          // Start new path
ctx.moveTo(100, 50);      // Move to first vertex
ctx.lineTo(150, 150);     // Draw line to second vertex
ctx.lineTo(50, 150);      // Draw line to third vertex
ctx.closePath();          // Close path to first vertex
ctx.stroke();             // Outline the triangle
ctx.fillStyle = 'lightblue';
ctx.fill();               // Fill the triangle with color
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.beginPath();          // Start new path
ctx.moveTo(100, 50);      // Move to first vertex
ctx.lineTo(150, 150);     // Draw line to second vertex
```

```
ctx.lineTo(50, 150);      // Draw line to third vertex
ctx.closePath();          // Close path to first vertex
ctx.stroke();             // Outline the triangle
ctx.fillStyle = 'lightblue';
ctx.fill();               // Fill the triangle with color
</script>

</body>
</html>
```

4.1.7 Example 2: Drawing an Open Zigzag Line

```
ctx.beginPath();
ctx.moveTo(50, 100);
ctx.lineTo(100, 50);
ctx.lineTo(150, 100);
ctx.lineTo(200, 50);
ctx.lineTo(250, 100);
ctx.stroke();
```

This draws a zigzag line by connecting alternating points.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.beginPath();
ctx.moveTo(50, 100);
ctx.lineTo(100, 50);
ctx.lineTo(150, 100);
ctx.lineTo(200, 50);
ctx.lineTo(250, 100);
ctx.stroke();
</script>

</body>
</html>
```

4.1.8 Example 3: Drawing an Irregular Shape

```
ctx.beginPath();
ctx.moveTo(80, 80);
ctx.lineTo(120, 60);
ctx.lineTo(160, 90);
ctx.lineTo(140, 130);
ctx.lineTo(100, 120);
ctx.lineTo(70, 110);
ctx.closePath();
ctx.stroke();
ctx.fillStyle = 'coral';
ctx.fill();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.beginPath();
ctx.moveTo(80, 80);
ctx.lineTo(120, 60);
ctx.lineTo(160, 90);
ctx.lineTo(140, 130);
ctx.lineTo(100, 120);
ctx.lineTo(70, 110);
ctx.closePath();
ctx.stroke();
ctx.fillStyle = 'coral';
ctx.fill();
</script>

</body>
</html>
```

4.1.9 Summary

- Use `beginPath()` to start fresh and avoid connecting unwanted lines.
- Use `moveTo(x, y)` to position the drawing cursor without creating lines.
- Use `lineTo(x, y)` to draw straight lines from the current point to a new point.
- Paths allow you to create complex shapes by chaining moves and lines.
- Use `closePath()` or manually connect the last point to the first to close polygons.

- After defining paths, use `stroke()` to outline and `fill()` to color inside.

Mastering paths opens up a world of custom drawings beyond basic shapes—next, you'll learn how to draw smooth curves and bezier paths!

4.2 Drawing Curves and Bezier Paths

Paths aren't limited to straight lines. To create **smooth, flowing curves**, the Canvas API provides two powerful methods: `quadraticCurveTo()` and `bezierCurveTo()`. These let you draw elegant shapes like waves, ribbons, and speech bubbles.

4.2.1 Understanding Curves on Canvas

Both methods use **control points** that "pull" the curve toward them, shaping the smooth path between your start and end points.

4.2.2 `quadraticCurveTo(cpX, cpY, x, y)`

- Draws a **quadratic Bézier curve** from the current point to `(x, y)`.
- The curve is influenced by **one control point** `(cpX, cpY)`.
- The control point acts like a magnet pulling the curve, determining its bend.

Visual Explanation:

```
Start Point ---- Curve bends toward Control Point ---- End Point
```

4.2.3 `bezierCurveTo(cp1X, cp1Y, cp2X, cp2Y, x, y)`

- Draws a **cubic Bézier curve** from the current point to `(x, y)`.
- Uses **two control points** `(cp1X, cp1Y)` and `(cp2X, cp2Y)`.
- Gives finer control over the curve's shape and smoothness.

Visual Explanation:

```
Start Point -- Curve shaped by Control Point 1 & Control Point 2 -- End Point
```

4.2.4 Annotated Diagram of Control Points (Conceptual)

Example 1: Drawing a Smooth Wave with `quadraticCurveTo()`

```
ctx.beginPath();
ctx.moveTo(50, 100);

ctx.quadraticCurveTo(150, 50, 250, 100); // Curve up and down
ctx.quadraticCurveTo(350, 150, 450, 100); // Curve down and up

ctx.strokeStyle = "blue";
ctx.lineWidth = 3;
ctx.stroke();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.beginPath();
ctx.moveTo(50, 100);

ctx.quadraticCurveTo(150, 50, 250, 100); // Curve up and down
ctx.quadraticCurveTo(350, 150, 450, 100); // Curve down and up

ctx.strokeStyle = "blue";
ctx.lineWidth = 3;
ctx.stroke();
</script>

</body>
</html>
```

Example 2: Drawing a Speech Bubble with `bezierCurveTo()`

```
ctx.beginPath();
ctx.moveTo(100, 100);

// Top curve
ctx.bezierCurveTo(150, 50, 250, 50, 300, 100);

// Right curve
ctx.bezierCurveTo(350, 150, 350, 250, 300, 300);

// Bottom curve
ctx.bezierCurveTo(250, 350, 150, 350, 100, 300);
```

```
// Left curve
ctx.bezierCurveTo(50, 250, 50, 150, 100, 100);

// Tail of bubble (triangle)
ctx.lineTo(80, 350);
ctx.lineTo(130, 300);

ctx.closePath();
ctx.fillStyle = "#fff8dc";
ctx.fill();
ctx.strokeStyle = "#333";
ctx.lineWidth = 2;
ctx.stroke();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.beginPath();
ctx.moveTo(100, 100);

// Top curve
ctx.bezierCurveTo(150, 50, 250, 50, 300, 100);

// Right curve
ctx.bezierCurveTo(350, 150, 350, 250, 300, 300);

// Bottom curve
ctx.bezierCurveTo(250, 350, 150, 350, 100, 300);

// Left curve
ctx.bezierCurveTo(50, 250, 50, 150, 100, 100);

// Tail of bubble (triangle)
ctx.lineTo(80, 350);
ctx.lineTo(130, 300);

ctx.closePath();
ctx.fillStyle = "#fff8dc";
ctx.fill();
ctx.strokeStyle = "#333";
ctx.lineWidth = 2;
ctx.stroke();
</script>

</body>
</html>
```

4.2.5 Experimenting with Control Points

- Move the **control points** closer to or further from the line between start and end points.
- Adjusting these points changes the **curve's height, steepness, and direction**.
- Try drawing the same curve with different control points to visualize the effect.

4.2.6 Summary

- Use `quadraticCurveTo(cpX, cpY, x, y)` for curves with **one control point**.
- Use `bezierCurveTo(cp1X, cp1Y, cp2X, cp2Y, x, y)` for more complex curves with **two control points**.
- Control points influence the curve's shape, acting like magnetic pulls.
- Curves let you create natural shapes such as waves, bubbles, or ribbons.
- Experimentation with control points is key to mastering canvas curves.

Next, we'll see how to close these paths and fill complex shapes for even richer graphics!

4.3 Using `closePath()` and Filling Complex Shapes

When working with paths, connecting points is essential—but sometimes you want to **automatically close** a shape by linking the last point back to the first. That's exactly what `closePath()` does.

4.3.1 What Does `closePath()` Do?

- It **draws a straight line** from the current point to the **starting point** of the path.
- It completes the shape, making it a **closed path**.
- This is especially useful for polygons or any shape that needs a sealed outline.

Without `closePath()`, your shape remains **open**, and the last point won't connect back to the start unless you do it manually.

4.3.2 When to Use `fill()` vs `stroke()`

- **`fill()`** paints the **interior area** of a closed path with the current fill style.
- **`stroke()`** draws the **outline** of the path using the current stroke style.

4.3.3 Important:

- `fill()` only fills **closed** paths (either explicitly closed with `closePath()` or implicitly closed by connecting start/end points).
- You can both **fill and stroke** the same path to color the inside and outline it.

4.3.4 Example: Open vs Closed Path

```
// Open Path (no closePath)
ctx.beginPath();
ctx.moveTo(50, 50);
ctx.lineTo(150, 50);
ctx.lineTo(100, 150);
ctx.stroke();  // Only outlines the two sides, no closure
```

vs.

```
// Closed Path using closePath()
ctx.beginPath();
ctx.moveTo(50, 50);
ctx.lineTo(150, 50);
ctx.lineTo(100, 150);
ctx.closePath(); // Connects last point to first
ctx.fillStyle = 'lightgreen';
ctx.fill();      // Fills the triangle area
ctx.stroke();    // Outlines the entire triangle
```

4.3.5 Building a Reusable Polygon Function

Let's create a helper function that draws **any polygon** by connecting multiple points and optionally fills and strokes it:

```
/**
 * Draws a polygon given an array of points.
 * @param {CanvasRenderingContext2D} ctx - The canvas context.
 * @param {Array} points - Array of {x, y} objects.
 * @param {string} fillColor - Color to fill the polygon.
 * @param {string} strokeColor - Color for the outline.
 */
function drawPolygon(ctx, points, fillColor, strokeColor) {
  if (points.length < 2) return; // Need at least 2 points

  ctx.beginPath();
  ctx.moveTo(points[0].x, points[0].y);

  for (let i = 1; i < points.length; i++) {
```

```
    ctx.lineTo(points[i].x, points[i].y);
  }

  ctx.closePath(); // Automatically close the shape

  if (fillColor) {
    ctx.fillStyle = fillColor;
    ctx.fill();
  }

  if (strokeColor) {
    ctx.strokeStyle = strokeColor;
    ctx.stroke();
  }
}
```

4.3.6 Using the Polygon Function

```
const triangle = [
  { x: 100, y: 50 },
  { x: 200, y: 150 },
  { x: 50,  y: 150 }
];

drawPolygon(ctx, triangle, 'skyblue', 'navy');
```

This draws a filled and stroked triangle using just an array of points.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
function drawPolygon(ctx, points, fillColor, strokeColor) {
  if (points.length < 2) return; // Need at least 2 points

  ctx.beginPath();
  ctx.moveTo(points[0].x, points[0].y);

  for (let i = 1; i < points.length; i++) {
    ctx.lineTo(points[i].x, points[i].y);
  }

  ctx.closePath(); // Automatically close the shape
```

```
  if (fillColor) {
    ctx.fillStyle = fillColor;
    ctx.fill();
  }

  if (strokeColor) {
    ctx.strokeStyle = strokeColor;
    ctx.stroke();
  }
}
const triangle = [
  { x: 100, y: 50 },
  { x: 200, y: 150 },
  { x: 50,  y: 150 }
];

drawPolygon(ctx, triangle, 'skyblue', 'navy');
</script>

</body>
</html>
```

4.3.7 Summary

- `closePath()` closes a path by connecting the last point back to the start.
- Closed paths allow filling their interiors with `fill()`.
- Open paths can only be stroked (outlined).
- Use both `fill()` and `stroke()` together for rich shapes.
- Reusable polygon functions simplify drawing complex shapes from points.

Mastering path closing and filling unlocks endless possibilities for custom shapes and intricate graphics on the canvas.

Next, we'll combine these techniques in a practical example by drawing a star!

4.4 Practical Example: Drawing a Star or Custom Shape

In this section, we'll create a **dynamic, reusable function** to draw a star shape on the canvas. We'll use path commands, loops, and math to draw multi-point stars, and add interactivity to customize the number of points and size.

4.4.1 Why a Star?

Stars are classic examples of complex shapes that require careful control of points and lines. They demonstrate how to combine loops, path drawing, and trigonometry in canvas graphics.

4.4.2 How a Star Works

A star consists of **outer and inner points** arranged around a center. To draw it:

1. Calculate coordinates of the outer points (at radius `outerRadius`).
2. Calculate coordinates of the inner points (at radius `innerRadius`).
3. Alternate connecting outer and inner points to form the star shape.

4.4.3 Step 1: Star Drawing Function

```
/**
 * Draws a star shape on the canvas.
 * @param {CanvasRenderingContext2D} ctx - The canvas context.
 * @param {number} centerX - X-coordinate of star center.
 * @param {number} centerY - Y-coordinate of star center.
 * @param {number} points - Number of star points (e.g. 5).
 * @param {number} outerRadius - Radius of outer points.
 * @param {number} innerRadius - Radius of inner points.
 * @param {string} fillColor - Color to fill the star.
 * @param {string} strokeColor - Color to stroke the star.
 */
function drawStar(ctx, centerX, centerY, points, outerRadius, innerRadius, fillColor, strokeColor) {
  if (points < 2) return; // Need at least 2 points to draw a star

  const step = Math.PI / points; // Half angle between star points

  ctx.beginPath();

  for (let i = 0; i < 2 * points; i++) {
    // Use outer radius for even points, inner radius for odd points
    const radius = (i % 2 === 0) ? outerRadius : innerRadius;
    const angle = i * step - Math.PI / 2; // Start at top (12 o'clock)

    const x = centerX + radius * Math.cos(angle);
    const y = centerY + radius * Math.sin(angle);

    if (i === 0) {
      ctx.moveTo(x, y);
    } else {
      ctx.lineTo(x, y);
    }
  }
```

```
  ctx.closePath();

  if (fillColor) {
    ctx.fillStyle = fillColor;
    ctx.fill();
  }

  if (strokeColor) {
    ctx.strokeStyle = strokeColor;
    ctx.lineWidth = 2;
    ctx.stroke();
  }
}
```

4.4.4 Step 2: Basic Usage

Draw a simple 5-pointed star in the center of the canvas:

```
const canvas = document.getElementById('starCanvas');
const ctx = canvas.getContext('2d');

drawStar(ctx, canvas.width / 2, canvas.height / 2, 5, 80, 40, 'gold', 'orange');
```

4.4.5 Step 3: Adding Interactivity

Create HTML controls to let users specify the number of points and the size dynamically:

```
<label>
  Number of Points:
  <input type="number" id="pointsInput" value="5" min="3" max="20" />
</label>
<label>
  Outer Radius:
  <input type="number" id="outerRadiusInput" value="80" min="10" max="200" />
</label>
<label>
  Inner Radius:
  <input type="number" id="innerRadiusInput" value="40" min="5" max="100" />
</label>

<canvas id="starCanvas" width="400" height="400" style="border:1px solid #ccc;"></canvas>
```

4.4.6 Step 4: Handling Input and Redrawing

```
const pointsInput = document.getElementById('pointsInput');
const outerRadiusInput = document.getElementById('outerRadiusInput');
const innerRadiusInput = document.getElementById('innerRadiusInput');
const canvas = document.getElementById('starCanvas');
const ctx = canvas.getContext('2d');

function draw() {
  // Clear canvas
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Parse user inputs
  const points = Math.max(3, parseInt(pointsInput.value, 10) || 5);
  const outerRadius = Math.max(10, parseInt(outerRadiusInput.value, 10) || 80);
  const innerRadius = Math.max(5, parseInt(innerRadiusInput.value, 10) || 40);

  // Draw star with current parameters
  drawStar(ctx, canvas.width / 2, canvas.height / 2, points, outerRadius, innerRadius, 'gold', 'orange'
}

// Attach event listeners to inputs
pointsInput.addEventListener('input', draw);
outerRadiusInput.addEventListener('input', draw);
innerRadiusInput.addEventListener('input', draw);

// Initial draw
draw();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Interactive Star Drawer</title>
  <style>
    body {
      font-family: sans-serif;
      padding: 1rem;
    }
    label {
      display: block;
      margin-bottom: 0.5rem;
    }
    input {
      width: 60px;
      margin-left: 0.5rem;
    }
  </style>
</head>
<body>

<h2>Draw a Star</h2>

<label>
  Number of Points:
  <input type="number" id="pointsInput" value="5" min="3" max="20" />
```

```
</label>
<label>
  Outer Radius:
  <input type="number" id="outerRadiusInput" value="80" min="10" max="200" />
</label>
<label>
  Inner Radius:
  <input type="number" id="innerRadiusInput" value="40" min="5" max="100" />
</label>

<canvas id="starCanvas" width="400" height="400" style="border:1px solid #ccc;"></canvas>

<script>
  function drawStar(ctx, centerX, centerY, points, outerRadius, innerRadius, fillColor, strokeColor) {
    if (points < 2) return;
    const step = Math.PI / points;
    ctx.beginPath();
    for (let i = 0; i < 2 * points; i++) {
      const radius = (i % 2 === 0) ? outerRadius : innerRadius;
      const angle = i * step - Math.PI / 2;
      const x = centerX + radius * Math.cos(angle);
      const y = centerY + radius * Math.sin(angle);
      if (i === 0) ctx.moveTo(x, y);
      else ctx.lineTo(x, y);
    }
    ctx.closePath();
    if (fillColor) {
      ctx.fillStyle = fillColor;
      ctx.fill();
    }
    if (strokeColor) {
      ctx.strokeStyle = strokeColor;
      ctx.lineWidth = 2;
      ctx.stroke();
    }
  }

  const pointsInput = document.getElementById('pointsInput');
  const outerRadiusInput = document.getElementById('outerRadiusInput');
  const innerRadiusInput = document.getElementById('innerRadiusInput');
  const canvas = document.getElementById('starCanvas');
  const ctx = canvas.getContext('2d');

  function draw() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    const points = Math.max(3, parseInt(pointsInput.value, 10) || 5);
    const outerRadius = Math.max(10, parseInt(outerRadiusInput.value, 10) || 80);
    const innerRadius = Math.max(5, parseInt(innerRadiusInput.value, 10) || 40);
    drawStar(ctx, canvas.width / 2, canvas.height / 2, points, outerRadius, innerRadius, 'gold', 'orang
  }

  pointsInput.addEventListener('input', draw);
  outerRadiusInput.addEventListener('input', draw);
  innerRadiusInput.addEventListener('input', draw);

  draw(); // initial draw
</script>
```

```
</body>
</html>
```

4.4.7 Experiment and Extend

- Try changing colors dynamically.
- Modify the function to draw stars with curved edges using `bezierCurveTo()`.
- Use the star function to create icons, badges, or even more complex shapes by combining stars.
- Add animation by gradually changing the number of points or radius.

4.4.8 Summary

- We created a **parameterized star function** using loops and path commands.
- The star is drawn by alternating **outer and inner points** around a center.
- The function is reusable for any number of points and sizes.
- Interactivity allows dynamic changes and immediate redraws.
- This pattern can be extended to other complex shapes and icons.

Mastering this approach empowers you to create intricate shapes programmatically and build engaging, customizable canvas graphics!

Chapter 5.

Images and Patterns

1. Loading and Drawing Images on Canvas
2. Using Images as Patterns
3. Manipulating Image Pixels with `getImageData()` and `putImageData()`
4. Practical Example: Simple Image Editor

5 Images and Patterns

5.1 Loading and Drawing Images on Canvas

Images play a crucial role in canvas graphics, allowing you to combine pictures with your drawings, create textures, or build complex scenes. This section explains how to **load external images** and draw them onto the canvas using JavaScript.

5.1.1 Loading Images

You can load images in two common ways:

5.1.2 Using the `Image()` Constructor in JavaScript

```
const img = new Image();
img.src = 'path/to/image.jpg';
```

- The image starts loading asynchronously when you set the `src` property.
- You must wait for the image to finish loading before drawing it on the canvas.
- Use the `onload` event to know when the image is ready.

5.1.3 Using an HTML `img` Element

You can also place an `<img>` tag in your HTML and access it in JavaScript:

```
<img id="myImage" src="path/to/image.jpg" style="display:none" />
```

```
const img = document.getElementById('myImage');
img.onload = function() {
  // draw image here
};
```

The advantage is easier HTML management, but the principle is the same: wait for loading before drawing.

5.1.4 Drawing Images with `drawImage()`

Once loaded, you use the canvas **2D context's** `drawImage()` method to render the image.

5.1.5 Basic Syntax:

```
ctx.drawImage(image, dx, dy);
```

- Draws the whole image at canvas coordinates `(dx, dy)`.
- The image is drawn at its natural size.

5.1.6 Drawing the Whole Image Example:

```
const img = new Image();
img.src = 'flower.jpg';
img.onload = () => {
  ctx.drawImage(img, 0, 0);
};
```

This draws the image at the top-left corner `(0, 0)` using its original size.

5.1.7 Drawing and Scaling the Image:

```
ctx.drawImage(img, dx, dy, dWidth, dHeight);
```

- Draws and **scales** the image to the width and height specified.
- Useful for resizing or fitting the image into a particular canvas area.

Example:

```
ctx.drawImage(img, 10, 10, 200, 150);
```

Draws the image at `(10, 10)` scaled to `200px` wide and `150px` tall.

5.1.8 Cropping (Source and Destination Rectangles)

`drawImage()` can also draw **cropped** portions of the image:

```
ctx.drawImage(img, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight);
```

- `(sx, sy)` — top-left coordinates of the source rectangle on the image.
- `(sWidth, sHeight)` — size of the cropped rectangle.
- `(dx, dy)` — destination coordinates on the canvas.
- `(dWidth, dHeight)` — size to draw the cropped area on the canvas.

Example — drawing just a 100x100 portion of the image, scaled:

```
ctx.drawImage(img, 50, 50, 100, 100, 10, 10, 200, 200);
```

5.1.9 Image Smoothing

By default, canvas **smooths images when scaling**, creating a nicer effect.

You can control this with:

```
ctx.imageSmoothingEnabled = true;  // Default is true
ctx.imageSmoothingQuality = 'high'; // Options: 'low', 'medium', 'high'
```

Set to `false` for pixelated, sharp edges (useful for pixel art).

5.1.10 Performance Tips for Large Images

- Loading large images can slow down rendering and increase memory usage.
- Consider scaling images down before drawing if large resolution isn't needed.
- Cache images in variables to avoid repeated loading.
- Use compressed image formats (JPEG, PNG) optimized for web.
- Avoid drawing images before they are fully loaded to prevent errors.

5.1.11 Complete Example

```
<canvas id="canvas" width="400" height="300"></canvas>
<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');
```

```
  const img = new Image();
  img.src = 'https://example.com/flower.jpg';

  img.onload = () => {
    // Draw entire image at (0,0)
    ctx.drawImage(img, 0, 0);

    // Draw scaled version at (210, 10)
    ctx.drawImage(img, 210, 10, 150, 100);

    // Draw cropped portion (100x100 area starting at 50,50) scaled to 100x100 at (10, 160)
    ctx.drawImage(img, 50, 50, 100, 100, 10, 160, 100, 100);
  };
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

  const img = new Image();
  img.src = 'https://example.com/flower.jpg';

  img.onload = () => {
    // Draw entire image at (0,0)
    ctx.drawImage(img, 0, 0);

    // Draw scaled version at (210, 10)
    ctx.drawImage(img, 210, 10, 150, 100);

    // Draw cropped portion (100x100 area starting at 50,50) scaled to 100x100 at (10, 160)
    ctx.drawImage(img, 50, 50, 100, 100, 10, 160, 100, 100);
  };
</script>

</body>
</html>
```

5.1.12 Summary

- Load images asynchronously using the `Image()` constructor or `<img>` element.
- Always wait for the image to finish loading (`onload`) before drawing.
- Use `drawImage()` to draw images in multiple ways:

  - Draw full image at original size.
  - Scale image to fit any size.
  - Crop portions of the image with source/destination rectangles.
- Control image smoothing for crisp or smooth scaling.
- Optimize large images for performance.

Mastering image loading and drawing unlocks powerful canvas capabilities—next, we'll explore how to **use images as repeating patterns**!

5.2 Using Images as Patterns

In addition to drawing whole images on the canvas, you can use images to create **repeatable patterns** that fill shapes, backgrounds, or even text. This is done with the powerful `createPattern()` method.

5.2.1 What Is `createPattern()`?

The `createPattern()` method takes an image (or canvas) and generates a **pattern object** that can be used as a fill style. Instead of painting a single image, the pattern repeats the image across the filled area according to a specified repeat mode.

5.2.2 Syntax:

```
const pattern = ctx.createPattern(image, repetition);
```

- `image`: An `HTMLImageElement`, `<canvas>`, or `<video>` element.
- `repetition`: A string defining how the image repeats:
  - `'repeat'` — repeats both horizontally and vertically (default).
  - `'repeat-x'` — repeats horizontally only.
  - `'repeat-y'` — repeats vertically only.
  - `'no-repeat'` — no repetition; draws image once.

5.2.3 Using Patterns as Fill Styles

Once created, assign the pattern to `ctx.fillStyle` and use `fill()` or other fill methods to apply it.

5.2.4 Example 1: Filling the Canvas Background with a Repeating Pattern

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const img = new Image();
img.src = 'https://example.com/pattern.png';

img.onload = () => {
  const pattern = ctx.createPattern(img, 'repeat');
  ctx.fillStyle = pattern;
  ctx.fillRect(0, 0, canvas.width, canvas.height);
};
```

This will tile the pattern image across the entire canvas area.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

const img = new Image();
img.src = 'https://example.com/pattern.png';

img.onload = () => {
  const pattern = ctx.createPattern(img, 'repeat');
  ctx.fillStyle = pattern;
  ctx.fillRect(0, 0, canvas.width, canvas.height);
};

</script>

</body>
</html>
```

5.2.5 Example 2: Filling a Shape with a Pattern (Circle)

```
ctx.beginPath();
ctx.arc(150, 150, 100, 0, 2 * Math.PI);
ctx.closePath();

ctx.fillStyle = pattern;
ctx.fill();
```

Here, the circle is filled with the repeating image pattern.

5.2.6 Example 3: Different Repeat Modes

Try changing the repeat mode to see how the pattern behaves:

```
// Horizontal repeat only
const patternX = ctx.createPattern(img, 'repeat-x');

// Vertical repeat only
const patternY = ctx.createPattern(img, 'repeat-y');

// No repeat (single image)
const patternNone = ctx.createPattern(img, 'no-repeat');
```

Assign these to `fillStyle` to observe the effect on fills.

5.2.7 Transparency

Patterns respect the transparency of the image file (like PNGs with alpha). Use transparent images for layered effects or subtle textures.

5.2.8 Rotation

Canvas patterns **do not have built-in rotation**, but you can rotate the canvas context before filling:

```
ctx.save(); // Save current state
ctx.translate(centerX, centerY); // Move origin to center
ctx.rotate(Math.PI / 4);         // Rotate 45 degrees
ctx.translate(-centerX, -centerY); // Move origin back

ctx.fillStyle = pattern;
```

```
ctx.fillRect(0, 0, canvas.width, canvas.height);

ctx.restore(); // Restore original state
```

This rotates the entire coordinate system, causing the pattern to appear rotated.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
const img = new Image();
img.src = 'https://example.com/pattern.png';

img.onload = () => {
  const pattern = ctx.createPattern(img, 'repeat');
  ctx.fillStyle = pattern;
  ctx.fillRect(0, 0, canvas.width, canvas.height);
};
ctx.save(); // Save current state
ctx.translate(centerX, centerY); // Move origin to center
ctx.rotate(Math.PI / 4);         // Rotate 45 degrees
ctx.translate(-centerX, -centerY); // Move origin back

ctx.fillStyle = pattern;
ctx.fillRect(0, 0, canvas.width, canvas.height);

ctx.restore(); // Restore original state
</script>

</body>
</html>
```

5.2.9 Summary

- Use `createPattern(image, repetition)` to turn images into repeatable fill styles.
- Supported repetition modes: `'repeat'`, `'repeat-x'`, `'repeat-y'`, and `'no-repeat'`.
- Patterns can fill backgrounds, shapes, or even text (when combined with clipping).
- Transparent images add subtlety and layering.
- Rotate patterns by rotating the canvas context before filling.
- Experiment with patterns to create rich textures and backgrounds!

Next, we'll dive into pixel-level control by manipulating image data with `getImageData()` and `putImageData()`.

5.3 Manipulating Image Pixels with `getImageData()` and `putImageData()`

5.3.1 Manipulating Image Pixels with `getImageData()` and `putImageData()`

One of the most powerful features of the HTML5 canvas is the ability to **access and manipulate raw pixel data**. This allows you to implement custom image processing effects like grayscale conversion, color inversion, brightness adjustments, and much more.

5.3.2 What is `getImageData()`?

The `getImageData()` method retrieves an object containing the pixel data from a specified rectangle of the canvas.

5.3.3 Syntax:

```
const imageData = ctx.getImageData(x, y, width, height);
```

- `x`, `y`: The coordinates of the top-left corner of the rectangle.
- `width`, `height`: Dimensions of the rectangle.

The returned `imageData` object contains:

- `.width` and `.height`: The size of the rectangle.
- `.data`: A `Uint8ClampedArray` holding RGBA pixel values for every pixel in the rectangle.

5.3.4 How Pixel Data is Structured

- The `.data` array contains **4 bytes per pixel**, in this order:
  - `R` (red)
  - `G` (green)
  - `B` (blue)
  - `A` (alpha/opacity)
- The length of `.data` is `width * height * 4`.

For example, the pixel at `(px, py)` is at index:

```
const index = (py * width + px) * 4;
```

- Red is at `index`
- Green at `index + 1`
- Blue at `index + 2`
- Alpha at `index + 3`

5.3.5 Modifying Pixels: Reading and Writing

You can loop through the pixel array to read or modify colors:

```
const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
const data = imageData.data;

for (let i = 0; i < data.length; i += 4) {
  // Access pixels: data[i] = R, data[i+1] = G, data[i+2] = B, data[i+3] = A
  // Modify pixel here
}

ctx.putImageData(imageData, 0, 0);
```

After modification, use `putImageData()` to write the pixel data back to the canvas.

5.3.6 Example 1: Grayscale Filter

Convert each pixel to grayscale by averaging its red, green, and blue values:

```
for (let i = 0; i < data.length; i += 4) {
  const r = data[i];
  const g = data[i + 1];
  const b = data[i + 2];

  const avg = (r + g + b) / 3;

  data[i] = avg;     // Red
  data[i + 1] = avg; // Green
  data[i + 2] = avg; // Blue
  // Alpha remains unchanged
}
ctx.putImageData(imageData, 0, 0);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
```

```
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

for (let i = 0; i < data.length; i += 4) {
  const r = data[i];
  const g = data[i + 1];
  const b = data[i + 2];

  const avg = (r + g + b) / 3;

  data[i] = avg;     // Red
  data[i + 1] = avg; // Green
  data[i + 2] = avg; // Blue
  // Alpha remains unchanged
}
ctx.putImageData(imageData, 0, 0);
</script>

</body>
</html>
```

5.3.7 Example 2: Color Inversion

Invert colors by subtracting each RGB value from 255:

```
for (let i = 0; i < data.length; i += 4) {
  data[i] = 255 - data[i];       // Red
  data[i + 1] = 255 - data[i + 1]; // Green
  data[i + 2] = 255 - data[i + 2]; // Blue
}
ctx.putImageData(imageData, 0, 0);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

for (let i = 0; i < data.length; i += 4) {
  data[i] = 255 - data[i];       // Red
```

```
  data[i + 1] = 255 - data[i + 1]; // Green
  data[i + 2] = 255 - data[i + 2]; // Blue
}
ctx.putImageData(imageData, 0, 0);
</script>

</body>
</html>
```

5.3.8 Example 3: Brightness Adjustment

Increase or decrease brightness by adding a value to RGB channels (clamping between 0 and 255):

```
const brightness = 40; // Positive to brighten, negative to darken

for (let i = 0; i < data.length; i += 4) {
  data[i] = Math.min(255, Math.max(0, data[i] + brightness));     // Red
  data[i + 1] = Math.min(255, Math.max(0, data[i + 1] + brightness)); // Green
  data[i + 2] = Math.min(255, Math.max(0, data[i + 2] + brightness)); // Blue
}
ctx.putImageData(imageData, 0, 0);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

const brightness = 40; // Positive to brighten, negative to darken

for (let i = 0; i < data.length; i += 4) {
  data[i] = Math.min(255, Math.max(0, data[i] + brightness));     // Red
  data[i + 1] = Math.min(255, Math.max(0, data[i + 1] + brightness)); // Green
  data[i + 2] = Math.min(255, Math.max(0, data[i + 2] + brightness)); // Blue
}
ctx.putImageData(imageData, 0, 0);
</script>

</body>
</html>
```

5.3.9 Performance Considerations

- Pixel manipulation can be **computationally expensive**, especially for large canvases or frequent updates.
- Avoid calling `getImageData()` and `putImageData()` repeatedly inside animation loops unless necessary.
- Try to minimize the rectangle size you access to improve performance.
- For heavy processing, consider **offscreen canvases** or **Web Workers** to keep the UI responsive.

5.3.10 Summary

- `getImageData()` lets you read pixel RGBA values from the canvas.
- Pixels are stored sequentially as 4 bytes per pixel.
- Modify pixels using loops and write back changes with `putImageData()`.
- Common filters like grayscale, invert, and brightness are easy to implement.
- Be mindful of performance; optimize by working on smaller areas and limiting frequency.

In the next section, we'll apply these techniques in a **simple image editor** that combines drawing, filters, and interactivity!

5.4 Practical Example: Simple Image Editor

In this section, we will build a **simple yet powerful image editor** using the canvas. This editor will let users:

- Upload an image,
- Apply basic filters (grayscale, sepia, invert),
- Download the edited image.

We'll combine the techniques you've learned: loading images, pixel manipulation with `getImageData()`/`putImageData()`, and drawing on canvas.

5.4.1 Step 1: HTML Setup

```
<input type="file" id="upload" accept="image/*" />
<br />

<button id="grayscale">Grayscale</button>
```

```
<button id="sepia">Sepia</button>
<button id="invert">Invert</button>
<button id="reset">Reset</button>
<button id="download">Download</button>

<br /><br />

<canvas id="canvas" width="500" height="400" style="border:1px solid #ccc;"></canvas>
```

5.4.2 Step 2: JavaScript Core Logic

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const uploadInput = document.getElementById('upload');
const grayscaleBtn = document.getElementById('grayscale');
const sepiaBtn = document.getElementById('sepia');
const invertBtn = document.getElementById('invert');
const resetBtn = document.getElementById('reset');
const downloadBtn = document.getElementById('download');

let originalImageData = null;

// Load and draw uploaded image
uploadInput.addEventListener('change', (e) => {
  const file = e.target.files[0];
  if (!file) return;

  const img = new Image();
  img.onload = () => {
    // Resize canvas to fit image
    canvas.width = img.width;
    canvas.height = img.height;

    ctx.drawImage(img, 0, 0);
    // Save original image data for reset
    originalImageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
  };

  img.src = URL.createObjectURL(file);
});

// Apply filter helper
function applyFilter(filterFn) {
  if (!originalImageData) return;

  // Copy original to avoid stacking effects
  const imageData = ctx.createImageData(originalImageData);
  imageData.data.set(originalImageData.data);

  const data = imageData.data;
```

```
  for (let i = 0; i < data.length; i += 4) {
    filterFn(data, i);
  }

  ctx.putImageData(imageData, 0, 0);
}

// Filters

function grayscale(data, i) {
  const avg = (data[i] + data[i + 1] + data[i + 2]) / 3;
  data[i] = data[i + 1] = data[i + 2] = avg;
}

function sepia(data, i) {
  const r = data[i];
  const g = data[i + 1];
  const b = data[i + 2];

  data[i]     = Math.min(255, 0.393 * r + 0.769 * g + 0.189 * b);
  data[i + 1] = Math.min(255, 0.349 * r + 0.686 * g + 0.168 * b);
  data[i + 2] = Math.min(255, 0.272 * r + 0.534 * g + 0.131 * b);
}

function invert(data, i) {
  data[i]     = 255 - data[i];
  data[i + 1] = 255 - data[i + 1];
  data[i + 2] = 255 - data[i + 2];
}

// Event listeners for buttons

grayscaleBtn.addEventListener('click', () => applyFilter(grayscale));
sepiaBtn.addEventListener('click', () => applyFilter(sepia));
invertBtn.addEventListener('click', () => applyFilter(invert));

resetBtn.addEventListener('click', () => {
  if (!originalImageData) return;
  ctx.putImageData(originalImageData, 0, 0);
});

downloadBtn.addEventListener('click', () => {
  const link = document.createElement('a');
  link.download = 'edited-image.png';
  link.href = canvas.toDataURL();
  link.click();
});
```

5.4.3 How It Works

1. **Upload Image:** The user selects an image file, which is loaded into an `Image` object. Once loaded, the canvas resizes to the image dimensions and draws it.

2. **Store Original:** The pixel data of the original image is stored in `originalImageData` so filters can be applied repeatedly without stacking effects.
3. **Apply Filters:** Each filter is a function that manipulates pixel data inside a loop by modifying the RGBA values.
4. **Filter Pipeline:** The `applyFilter()` function copies the original pixel data, applies the filter, and updates the canvas with `putImageData()`.
5. **Reset:** Restores the original image from saved pixel data.
6. **Download:** Converts the canvas content to a PNG data URL and triggers a download.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Image Filter Editor</title>
  <style>
    body {
      font-family: sans-serif;
      padding: 1rem;
    }
    canvas {
      display: block;
      margin-top: 1rem;
    }
    button {
      margin-right: 0.5rem;
    }
  </style>
</head>
<body>

<h2>Image Filter Editor</h2>

<input type="file" id="upload" accept="image/*" />
<br><br>

<button id="grayscale">Grayscale</button>
<button id="sepia">Sepia</button>
<button id="invert">Invert</button>
<button id="reset">Reset</button>
<button id="download">Download</button>

<canvas id="canvas" width="500" height="400" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');

  const uploadInput = document.getElementById('upload');
  const grayscaleBtn = document.getElementById('grayscale');
  const sepiaBtn = document.getElementById('sepia');
  const invertBtn = document.getElementById('invert');
  const resetBtn = document.getElementById('reset');
  const downloadBtn = document.getElementById('download');
```

```
let originalImageData = null;

// Load and draw uploaded image
uploadInput.addEventListener('change', (e) => {
  const file = e.target.files[0];
  if (!file) return;

  const img = new Image();
  img.onload = () => {
    canvas.width = img.width;
    canvas.height = img.height;
    ctx.drawImage(img, 0, 0);
    originalImageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
  };
  img.src = URL.createObjectURL(file);
});

function applyFilter(filterFn) {
  if (!originalImageData) return;
  const imageData = ctx.createImageData(originalImageData);
  imageData.data.set(originalImageData.data);
  const data = imageData.data;

  for (let i = 0; i < data.length; i += 4) {
    filterFn(data, i);
  }

  ctx.putImageData(imageData, 0, 0);
}

function grayscale(data, i) {
  const avg = (data[i] + data[i + 1] + data[i + 2]) / 3;
  data[i] = data[i + 1] = data[i + 2] = avg;
}

function sepia(data, i) {
  const r = data[i], g = data[i + 1], b = data[i + 2];
  data[i]     = Math.min(255, 0.393 * r + 0.769 * g + 0.189 * b);
  data[i + 1] = Math.min(255, 0.349 * r + 0.686 * g + 0.168 * b);
  data[i + 2] = Math.min(255, 0.272 * r + 0.534 * g + 0.131 * b);
}

function invert(data, i) {
  data[i]     = 255 - data[i];
  data[i + 1] = 255 - data[i + 1];
  data[i + 2] = 255 - data[i + 2];
}

grayscaleBtn.addEventListener('click', () => applyFilter(grayscale));
sepiaBtn.addEventListener('click', () => applyFilter(sepia));
invertBtn.addEventListener('click', () => applyFilter(invert));

resetBtn.addEventListener('click', () => {
  if (originalImageData) {
    ctx.putImageData(originalImageData, 0, 0);
  }
});
```

```
  downloadBtn.addEventListener('click', () => {
    const link = document.createElement('a');
    link.download = 'edited-image.png';
    link.href = canvas.toDataURL();
    link.click();
  });
</script>

</body>
</html>
```

5.4.4 Summary

- This editor combines **image loading**, **pixel manipulation**, and **canvas drawing**.
- Filters like grayscale, sepia, and invert modify pixels by adjusting RGBA values.
- Storing original pixel data prevents unwanted cumulative filter effects.
- The UI provides simple buttons to apply filters, reset, and download results.
- The modular code allows easy extension with new filters or features.

You now have a solid foundation for building more advanced image editors, combining powerful canvas APIs with user interactivity!

Chapter 6.

Transformations

1. Translation, Rotation, and Scaling
2. Using `save()` and `restore()` to Manage State
3. Transforming Coordinates and Shapes
4. Practical Example: Rotating and Scaling an Object

6 Transformations

6.1 Translation, Rotation, and Scaling

Canvas transformations allow you to **move, rotate, and resize** your drawings by modifying the drawing context's coordinate system. Understanding how these transformations work is essential for creating dynamic, animated, or complex graphics.

6.1.1 The Canvas Transformation Matrix

The canvas uses a **transformation matrix** to track how coordinates are transformed before drawing. Every time you call a transformation method, the matrix updates and changes the coordinate system for all future drawing commands.

6.1.2 Transformation Methods

`translate(x, y)`

Moves (translates) the origin of the coordinate system by `(x, y)`. After translation, all subsequent drawings are offset by this amount.

- Example: `ctx.translate(100, 50)` shifts the origin right by 100 pixels and down by 50 pixels.

`rotate(angle)`

Rotates the coordinate system by an `angle` in **radians** around the origin `(0, 0)`.

- Positive values rotate clockwise.
- Use `Math.PI` for 180°, `Math.PI / 2` for 90°, etc.
- Rotation happens **around the current origin**.

`scale(sx, sy)`

Scales the coordinate system by factors `sx` (horizontal) and `sy` (vertical).

- Values greater than 1 stretch the drawing.
- Values between 0 and 1 shrink it.
- Negative values flip (mirror) the drawing across the axis.

6.1.3 Important: Transformations Affect the Context, Not the Shapes Directly

- Transformations apply to the **entire drawing context**, changing how coordinates map to pixels.
- You don't transform shapes themselves but rather the coordinate system they are drawn in.
- Once applied, all drawing commands use the transformed coordinates.

6.1.4 Visual Example: Drawing a Rectangle Multiple Times with Different Transformations

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

ctx.fillStyle = 'steelblue';

// Draw original rectangle at origin
ctx.fillRect(0, 0, 100, 50);

// Translate and draw second rectangle
ctx.translate(150, 0);
ctx.fillRect(0, 0, 100, 50);

// Rotate and draw third rectangle
ctx.translate(150, 75); // Move origin further right and down
ctx.rotate(Math.PI / 4); // Rotate 45 degrees
ctx.fillRect(0, 0, 100, 50);

// Scale and draw fourth rectangle
ctx.setTransform(1, 0, 0, 1, 0, 0); // Reset transform to identity
ctx.translate(100, 200);
ctx.scale(1.5, 0.5);
ctx.fillRect(0, 0, 100, 50);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.fillStyle = 'steelblue';

// Draw original rectangle at origin
```

```
ctx.fillRect(0, 0, 100, 50);

// Translate and draw second rectangle
ctx.translate(150, 0);
ctx.fillRect(0, 0, 100, 50);

// Rotate and draw third rectangle
ctx.translate(150, 75); // Move origin further right and down
ctx.rotate(Math.PI / 4); // Rotate 45 degrees
ctx.fillRect(0, 0, 100, 50);

// Scale and draw fourth rectangle
ctx.setTransform(1, 0, 0, 1, 0, 0); // Reset transform to identity
ctx.translate(100, 200);
ctx.scale(1.5, 0.5);
ctx.fillRect(0, 0, 100, 50);

</script>

</body>
</html>
```

6.1.5 Explanation:

- The first rectangle draws at the default origin `(0, 0)`.
- The second rectangle is shifted right by 150 pixels.
- The third rectangle is moved and rotated 45°, so it appears tilted.
- The fourth rectangle is scaled wider and shorter, demonstrating non-uniform scaling.

6.1.6 Rotation and Scaling Around the Origin

Because rotation and scaling happen **around the origin (0, 0)**, it's often necessary to translate the context **before** or **after** these transformations to control the pivot point.

Example: To rotate a rectangle around its center:

```
const x = 100;
const y = 100;
const width = 80;
const height = 40;

ctx.translate(x + width / 2, y + height / 2); // Move origin to rectangle center
ctx.rotate(Math.PI / 6);                      // Rotate 30 degrees
ctx.fillRect(-width / 2, -height / 2, width, height); // Draw rectangle centered
ctx.resetTransform();                         // Reset transformation matrix
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

const x = 100;
const y = 100;
const width = 80;
const height = 40;

ctx.translate(x + width / 2, y + height / 2); // Move origin to rectangle center
ctx.rotate(Math.PI / 6);                      // Rotate 30 degrees
ctx.fillRect(-width / 2, -height / 2, width, height); // Draw rectangle centered
ctx.resetTransform();                         // Reset transformation matrix
</script>

</body>
</html>
```

6.1.7 Summary

- `translate()`, `rotate()`, and `scale()` modify the canvas coordinate system via its transformation matrix.
- Transformations affect **how** shapes are drawn, not the shapes themselves.
- Rotation and scaling happen around the origin `(0, 0)`, so adjusting the origin with `translate()` is key for controlled effects.
- Use combinations of transformations to draw multiple instances of shapes with different positions, orientations, and sizes.

Mastering these transformations unlocks fluid and complex graphics, animations, and interactive effects on canvas!

6.2 Using `save()` and `restore()` to Manage State

When working with the canvas, you'll quickly realize that the **drawing context maintains a state** — a snapshot of all current settings such as styles, transformations, line widths, fonts, and more. Managing this state properly is essential to avoid unintended side effects during complex drawings.

6.2.1 What Is the Canvas Drawing State?

The canvas context holds various properties that affect drawing, including:

- **Transformations:** `translate()`, `rotate()`, `scale()`
- **Styles:** `fillStyle`, `strokeStyle`, `lineWidth`, `font`, etc.
- **Other settings:** `lineCap`, `lineJoin`, `globalAlpha`, `shadow*` properties

Every drawing operation uses the **current state** of the context.

6.2.2 Why Use `save()` and `restore()`?

- The context state **accumulates changes**; for example, if you translate the canvas origin once, all subsequent drawings are offset by that amount.
- Without control, multiple transformations or style changes **stack up**, leading to bugs like wrong positions, colors, or shapes.
- The methods `ctx.save()` and `ctx.restore()` help you **save** the current state and later **restore** it, making your drawing code modular and predictable.

6.2.3 How They Work

- `ctx.save()` pushes the current state onto a **stack**.
- `ctx.restore()` pops the last saved state off the stack and applies it.
- You can save and restore multiple times (nested), managing layers of changes.

6.2.4 Example: Drawing Two Rotated Rectangles Without save/restore (Buggy)

```
// Rotate 45 degrees and draw first rectangle
ctx.rotate(Math.PI / 4);
ctx.fillRect(50, 20, 100, 50);

// Rotate again (total 90 degrees) and draw second rectangle
ctx.rotate(Math.PI / 4);
ctx.fillRect(150, 20, 100, 50);
```

- The second rectangle is rotated **twice** because rotation accumulates.
- This often causes unexpected placements or orientations.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

// Rotate 45 degrees and draw first rectangle
ctx.rotate(Math.PI / 4);
ctx.fillRect(50, 20, 100, 50);

// Rotate again (total 90 degrees) and draw second rectangle
ctx.rotate(Math.PI / 4);
ctx.fillRect(150, 20, 100, 50);
</script>

</body>
</html>
```

6.2.5 Fixing with `save()` and `restore()`

```
// Draw first rectangle rotated 45 degrees
ctx.save();
ctx.rotate(Math.PI / 4);
ctx.fillRect(50, 20, 100, 50);
ctx.restore(); // Reset state after drawing

// Draw second rectangle rotated 45 degrees independently
ctx.save();
ctx.rotate(Math.PI / 4);
ctx.fillRect(150, 20, 100, 50);
ctx.restore();
```

- Each rectangle's rotation is **localized** inside a save/restore pair.
- After `restore()`, the context returns to its original unrotated state.
- This prevents transformations from accumulating unintentionally.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>
```

```
<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

// Draw first rectangle rotated 45 degrees
ctx.save();
ctx.rotate(Math.PI / 4);
ctx.fillRect(50, 20, 100, 50);
ctx.restore(); // Reset state after drawing

// Draw second rectangle rotated 45 degrees independently
ctx.save();
ctx.rotate(Math.PI / 4);
ctx.fillRect(150, 20, 100, 50);
ctx.restore();
</script>

</body>
</html>
```

6.2.6 Example: Combining Transformations and Styles

```
ctx.save();
ctx.translate(100, 100);
ctx.rotate(Math.PI / 6);
ctx.fillStyle = 'tomato';
ctx.fillRect(-50, -25, 100, 50);
ctx.restore();

ctx.save();
ctx.translate(250, 100);
ctx.scale(1.5, 0.5);
ctx.strokeStyle = 'navy';
ctx.lineWidth = 5;
ctx.strokeRect(0, 0, 100, 50);
ctx.restore();
```

- Each drawing block changes transforms and styles locally.
- After each block, the original context state is restored.
- This structure avoids accidental interference between shapes.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
```

```
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.save();
ctx.translate(100, 100);
ctx.rotate(Math.PI / 6);
ctx.fillStyle = 'tomato';
ctx.fillRect(-50, -25, 100, 50);
ctx.restore();

ctx.save();
ctx.translate(250, 100);
ctx.scale(1.5, 0.5);
ctx.strokeStyle = 'navy';
ctx.lineWidth = 5;
ctx.strokeRect(0, 0, 100, 50);
ctx.restore();
</script>

</body>
</html>
```

6.2.7 Summary

- Canvas context maintains a **drawing state** that includes transformations, styles, and other settings.
- State changes **accumulate**, which can cause bugs if not managed carefully.
- Use `save()` before applying transformations or style changes, and `restore()` after finishing those drawings.
- This allows **isolated, predictable modifications** of the canvas state.
- Nested save/restore calls let you build complex scenes with multiple layers and effects cleanly.

Mastering state management is key to clean, bug-free canvas programming!

6.3 Transforming Coordinates and Shapes

Transformations on the canvas allow you to **programmatically move, rotate, and scale shapes**, giving you powerful control over how graphics appear. This section teaches how to apply these transformations thoughtfully, enabling smooth animations and dynamic drawings.

6.3.1 Applying Transformations to Move and Orient Shapes

When you draw shapes, transformations change the coordinate system used for drawing. This lets you:

- Move shapes to different positions.
- Rotate shapes around points (e.g., their center).
- Scale shapes dynamically.

Transformations affect **all drawing commands** after they are applied until the context state is changed or reset.

6.3.2 Example 1: Rotating Around a Shapes Center

By default, `rotate()` spins the canvas around the origin `(0, 0)`. To rotate a shape around its own center, you must:

1. **Translate** the origin to the shape's center.
2. **Rotate** the canvas.
3. Draw the shape **centered at the new origin**.
4. Optionally reset the transformation.

```
function drawRotatedRect(ctx, x, y, width, height, angle) {
  ctx.save();

  // Move origin to rectangle center
  ctx.translate(x + width / 2, y + height / 2);

  // Rotate the canvas
  ctx.rotate(angle);

  // Draw rectangle centered at origin
  ctx.fillStyle = 'teal';
  ctx.fillRect(-width / 2, -height / 2, width, height);

  ctx.restore();
}

// Usage:
drawRotatedRect(ctx, 100, 100, 120, 60, Math.PI / 4); // 45 degrees
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>
```

```
<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

function drawRotatedRect(ctx, x, y, width, height, angle) {
  ctx.save();

  // Move origin to rectangle center
  ctx.translate(x + width / 2, y + height / 2);

  // Rotate the canvas
  ctx.rotate(angle);

  // Draw rectangle centered at origin
  ctx.fillStyle = 'teal';
  ctx.fillRect(-width / 2, -height / 2, width, height);

  ctx.restore();
}

// Usage:
drawRotatedRect(ctx, 100, 100, 120, 60, Math.PI / 4); // 45 degrees
</script>

</body>
</html>
```

6.3.3 Example 2: Scaling a Shape Dynamically

You can scale shapes on the fly by applying `scale()` before drawing:

```
function drawScaledCircle(ctx, x, y, radius, scaleX, scaleY) {
  ctx.save();

  // Move origin to circle center
  ctx.translate(x, y);

  // Scale the coordinate system
  ctx.scale(scaleX, scaleY);

  // Draw circle centered at origin
  ctx.beginPath();
  ctx.arc(0, 0, radius, 0, Math.PI * 2);
  ctx.fillStyle = 'orange';
  ctx.fill();

  ctx.restore();
}

// Usage:
drawScaledCircle(ctx, 200, 150, 50, 1.5, 0.75); // Wider horizontally, shorter vertically
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
function drawScaledCircle(ctx, x, y, radius, scaleX, scaleY) {
  ctx.save();

  // Move origin to circle center
  ctx.translate(x, y);

  // Scale the coordinate system
  ctx.scale(scaleX, scaleY);

  // Draw circle centered at origin
  ctx.beginPath();
  ctx.arc(0, 0, radius, 0, Math.PI * 2);
  ctx.fillStyle = 'orange';
  ctx.fill();

  ctx.restore();
}

// Usage:
drawScaledCircle(ctx, 200, 150, 50, 1.5, 0.75); // Wider horizontally, shorter vertically

</script>

</body>
</html>
```

6.3.4 Using `setTransform()` for Direct Matrix Control

While `translate()`, `rotate()`, and `scale()` update the transformation matrix cumulatively, **`setTransform()` resets the matrix** to a specific state instantly.

6.3.5 Syntax:

```
ctx.setTransform(a, b, c, d, e, f);
```

Where `a`–`f` represent the matrix components:

```
| a  c  e |
| b  d  f |
| 0  0  1 |
```

- Usually, `setTransform(1, 0, 0, 1, 0, 0)` resets to the identity matrix (no transform).
- This method is useful to **reset transformations quickly** before drawing new shapes.

6.3.6 Example:

```
ctx.setTransform(1, 0, 0, 1, 0, 0); // Reset transforms
ctx.fillRect(10, 10, 100, 50);      // Draw normally
```

6.3.7 Utility Functions for Reusable Transformations

Organize your code by encapsulating transformation patterns in functions.

```
function withTransform(ctx, x, y, angle, scaleX, scaleY, drawFn) {
  ctx.save();

  ctx.translate(x, y);
  ctx.rotate(angle);
  ctx.scale(scaleX, scaleY);

  drawFn(ctx); // Custom drawing inside transformed space

  ctx.restore();
}

// Usage example: draw a rotated, scaled star
withTransform(ctx, 250, 200, Math.PI / 3, 1.2, 1.2, (ctx) => {
  drawStar(ctx, 0, 0, 5, 50, 25);
});
```

This approach keeps transformation logic modular and improves readability.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Transformed Star Example</title>
  <style>
    body { background: #f0f0f0; margin: 20px; }
  </style>
</head>
<body>
```

```
<canvas id="myCanvas" width="400" height="300" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

  // Utility: Run drawing inside transformed context
  function withTransform(ctx, x, y, angle, scaleX, scaleY, drawFn) {
    ctx.save();
    ctx.translate(x, y);
    ctx.rotate(angle);
    ctx.scale(scaleX, scaleY);
    drawFn(ctx);
    ctx.restore();
  }

  // Star drawing function
  function drawStar(ctx, x, y, points, outerRadius, innerRadius) {
    const step = Math.PI / points;
    ctx.beginPath();
    for (let i = 0; i < 2 * points; i++) {
      const radius = i % 2 === 0 ? outerRadius : innerRadius;
      const angle = i * step;
      const sx = x + radius * Math.cos(angle);
      const sy = y + radius * Math.sin(angle);
      if (i === 0) ctx.moveTo(sx, sy);
      else ctx.lineTo(sx, sy);
    }
    ctx.closePath();
    ctx.fillStyle = "gold";
    ctx.strokeStyle = "black";
    ctx.lineWidth = 2;
    ctx.fill();
    ctx.stroke();
  }

  // Draw rotated, scaled star
  withTransform(ctx, 250, 200, Math.PI / 3, 1.2, 1.2, (ctx) => {
    drawStar(ctx, 0, 0, 5, 50, 25);
  });
</script>

</body>
</html>
```

6.3.8 Summary

- Transformations move and orient shapes by modifying the canvas coordinate system.
- To rotate or scale shapes around their center, **translate to the shape's center first**.
- `setTransform()` lets you directly set or reset the transformation matrix.
- Use **utility wrapper functions** to apply complex transformations cleanly.
- Combining these techniques enables flexible, reusable drawing code for dynamic graphics

and animations.

Next, we'll see how to combine transformations with state management to build sophisticated effects!

6.4 Practical Example: Rotating and Scaling an Object

In this practical example, we will create an interactive canvas demo where users can **rotate** and **scale** a rectangle using sliders. This hands-on example demonstrates how to combine transformations (`translate()`, `rotate()`, `scale()`) with state management (`save()`, `restore()`), and how user input can drive real-time graphics updates.

6.4.1 HTML Setup

```
<div>
  <label for="rotation">Rotation (degrees): </label>
  <input type="range" id="rotation" min="0" max="360" value="0" />
</div>

<div>
  <label for="scale">Scale (X and Y): </label>
  <input type="range" id="scale" min="-2" max="2" step="0.1" value="1" />
</div>

<canvas id="canvas" width="400" height="300" style="border:1px solid #ccc;"></canvas>
```

6.4.2 JavaScript Code

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const rotationSlider = document.getElementById('rotation');
const scaleSlider = document.getElementById('scale');

// Rectangle properties
const rectWidth = 120;
const rectHeight = 80;
const rectX = canvas.width / 2;
const rectY = canvas.height / 2;

function draw() {
  const angle = (rotationSlider.value * Math.PI) / 180; // Convert to radians
```

```
  const scale = parseFloat(scaleSlider.value);

  ctx.clearRect(0, 0, canvas.width, canvas.height);

  ctx.save();

  // Move origin to rectangle center
  ctx.translate(rectX, rectY);

  // Apply rotation and scaling
  ctx.rotate(angle);
  ctx.scale(scale, scale);

  // Draw rectangle centered at origin
  ctx.fillStyle = '#3498db';
  ctx.fillRect(-rectWidth / 2, -rectHeight / 2, rectWidth, rectHeight);

  ctx.restore();

  // Draw outline to visualize canvas edges
  ctx.strokeStyle = '#555';
  ctx.strokeRect(0, 0, canvas.width, canvas.height);
}

// Initial draw
draw();

// Update on slider input
rotationSlider.addEventListener('input', draw);
scaleSlider.addEventListener('input', draw);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Canvas Rotation & Scaling</title>
  <style>
    body {
      font-family: sans-serif;
      padding: 1rem;
    }
    canvas {
      border: 1px solid #ccc;
      display: block;
      margin-top: 1rem;
    }
    label {
      display: block;
      margin-top: 1rem;
    }
  </style>
</head>
<body>

<h2>Rotate & Scale Rectangle</h2>

<label>
```

```
  Rotation (degrees):
  <input type="range" id="rotation" min="0" max="360" value="0" />
</label>

<label>
  Scale:
  <input type="range" id="scale" min="0.1" max="3" step="0.1" value="1" />
</label>

<canvas id="canvas" width="400" height="300"></canvas>

<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');

  const rotationSlider = document.getElementById('rotation');
  const scaleSlider = document.getElementById('scale');

  const rectWidth = 120;
  const rectHeight = 80;
  const rectX = canvas.width / 2;
  const rectY = canvas.height / 2;

  function draw() {
    const angle = (rotationSlider.value * Math.PI) / 180;
    const scale = parseFloat(scaleSlider.value);

    ctx.clearRect(0, 0, canvas.width, canvas.height);

    ctx.save();

    ctx.translate(rectX, rectY);
    ctx.rotate(angle);
    ctx.scale(scale, scale);

    ctx.fillStyle = '#3498db';
    ctx.fillRect(-rectWidth / 2, -rectHeight / 2, rectWidth, rectHeight);

    ctx.restore();

    ctx.strokeStyle = '#555';
    ctx.strokeRect(0, 0, canvas.width, canvas.height);
  }

  draw();

  rotationSlider.addEventListener('input', draw);
  scaleSlider.addEventListener('input', draw);
</script>

</body>
</html>
```

6.4.3 Explanation

- **Canvas Setup:** We use a 400x300 canvas and center the rectangle in the middle.
- **User Controls:** Two sliders let users adjust rotation (0–360 degrees) and scale (-2 to 2, allowing flipping with negative values).
- **Transformations:**
  - `translate(rectX, rectY)` moves the origin to the rectangle's center.
  - `rotate(angle)` applies rotation in radians.
  - `scale(scale, scale)` scales uniformly on both axes; negative values flip the rectangle.
- **State Management:** Wrapping transformations and drawing between `ctx.save()` and `ctx.restore()` keeps these changes localized.
- **Real-Time Updates:** Event listeners redraw the rectangle instantly as slider values change.

6.4.4 Try These Variations

- **Negative Scale:** Slide the scale below zero to see the rectangle flip horizontally and vertically.
- **Full Rotation:** Move the rotation slider beyond 360° by adjusting max and min to watch continuous spinning.
- **Non-Uniform Scale:** Add separate sliders for `scaleX` and `scaleY` to distort the shape independently.
- **Use Images:** Replace the rectangle with an image for richer visual effects.

6.4.5 Summary

This example demonstrates:

- Applying **translation, rotation, and scaling** transformations in sequence.
- Using **`save()` and `restore()`** to isolate transformation effects.
- Connecting **user input to canvas drawing** for interactive graphics.
- The visual impact of **negative scaling** and **angle rotation** on shapes.

Experiment with the sliders to build an intuitive feel for canvas transformations—these are foundational tools for creating dynamic and animated graphics!

Chapter 7.

Animation Basics

1. Introduction to Animation Loops with `requestAnimationFrame`
2. Clearing and Redrawing the Canvas Efficiently
3. Basic Moving Objects Animation
4. Practical Example: Bouncing Ball Animation

7 Animation Basics

7.1 Introduction to Animation Loops with `requestAnimationFrame`

Animating on the web requires updating the canvas repeatedly to create the illusion of motion. To achieve smooth and efficient animations, JavaScript provides a powerful method called **`requestAnimationFrame`**.

7.1.1 What Is `requestAnimationFrame`?

`requestAnimationFrame` is a browser API designed specifically for animation loops. It tells the browser, *"Please call this function before the next repaint."* This allows your code to:

- Run **at the optimal frame rate**, typically matching the display's refresh rate (usually 60 frames per second).
- **Synchronize animations with the browser's rendering**, resulting in smoother motion.
- **Save CPU and battery** by pausing animations in inactive tabs or windows.

7.1.2 Why Not Use `setInterval` or `setTimeout`?

While `setInterval` and `setTimeout` can create loops, they have limitations for animation:

- They run on a **fixed timer**, which may not sync with the display's refresh rate, causing stutters or tearing.
- They continue running even if the tab is inactive, wasting resources.
- `requestAnimationFrame` pauses automatically when the tab is not visible, improving performance and battery life.

7.1.3 Basic Animation Loop with `requestAnimationFrame`

Here's a simple example demonstrating how to create an animation loop using `requestAnimationFrame`:

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

let x = 0;

function animate() {
```

```
  // Clear the canvas
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Draw a moving square
  ctx.fillStyle = 'tomato';
  ctx.fillRect(x, 50, 50, 50);

  // Update position
  x += 2;
  if (x > canvas.width) x = -50; // Reset position when off-screen

  // Request next frame
  requestAnimationFrame(animate);
}

// Start the animation loop
requestAnimationFrame(animate);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

let x = 0;

function animate() {
  // Clear the canvas
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Draw a moving square
  ctx.fillStyle = 'tomato';
  ctx.fillRect(x, 50, 50, 50);

  // Update position
  x += 2;
  if (x > canvas.width) x = -50; // Reset position when off-screen

  // Request next frame
  requestAnimationFrame(animate);
}

// Start the animation loop
requestAnimationFrame(animate);
</script>

</body>
</html>
```

7.1.4 How It Works:

- The `animate` function performs **one frame's work**: clearing the canvas, drawing, and updating the position.
- At the end of the function, `requestAnimationFrame(animate)` tells the browser to call `animate` again just before the next repaint.
- This creates a **recursive loop** that runs continuously and efficiently.

7.1.5 Understanding the Signature and Recursive Nature

- `requestAnimationFrame()` accepts a **callback function** that takes a **timestamp** parameter indicating when the frame is scheduled.
- You don't have to use the timestamp, but it can help with precise timing and animations based on elapsed time.
- The callback **must call `requestAnimationFrame` again** to keep the animation going, forming a loop.

Example with timestamp:

```
function animate(timestamp) {
  // Use timestamp for smooth, time-based animation
  // ...

  requestAnimationFrame(animate);
}
requestAnimationFrame(animate);
```

7.1.6 Summary

- Use `requestAnimationFrame` for smooth, browser-optimized animation loops.
- It synchronizes rendering to the display refresh rate, outperforming `setInterval`/`setTimeout`.
- Animation loops work by recursively calling the function inside `requestAnimationFrame`.
- This is the foundation for building games, interactive graphics, and fluid UI animations on canvas.

Next, we'll explore how to efficiently clear and redraw canvas content during animations!

7.2 Clearing and Redrawing the Canvas Efficiently

When creating animations on canvas, one of the most important steps is **clearing the canvas** before drawing each new frame. This prevents the previous frame's drawings from piling up and causing unwanted visual artifacts known as **ghosting** or **trails**.

7.2.1 Why Clear the Canvas?

Canvas works as a **bitmap** where each drawing operation paints pixels directly. Unlike DOM elements, the canvas does **not** automatically erase or update parts of the scene between frames.

If you don't clear the canvas before redrawing:

- Previous frames remain visible.
- Moving objects leave **trails** or **ghost images**, making the animation look blurry or messy.

7.2.2 How to Clear the Canvas: Using `clearRect()`

The most common and straightforward method is `clearRect()`:

```
ctx.clearRect(0, 0, canvas.width, canvas.height);
```

This clears the entire canvas by making all pixels fully transparent, resetting it for the next frame.

7.2.3 Example: Clearing the canvas each frame

```
function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Draw your animated objects here

  requestAnimationFrame(animate);
}
```

7.2.4 Alternative: Using Semi-Transparent Overlays for Motion Trails

Instead of fully clearing the canvas, you can draw a **semi-transparent rectangle** over the entire canvas. This creates a **motion trail** or **ghosting effect** that can look visually appealing in some animations.

```
ctx.fillStyle = 'rgba(255, 255, 255, 0.1)'; // White with low opacity
ctx.fillRect(0, 0, canvas.width, canvas.height);
```

- The low opacity overlay slowly fades previous frames.
- It creates a **blurred path** behind moving objects.
- This technique is often used for **particle effects** or **motion blur** simulations.

7.2.5 Best Practices for Optimizing Redraws

- **Clear only what's necessary:** If only part of the scene changes, you can clear and redraw only those regions.
- **Use dirty rectangles:** Track which areas have changed and update only those.
- This can improve performance, especially on large canvases or complex animations.

However, for simple animations, clearing the entire canvas with `clearRect()` is usually sufficient and straightforward.

7.2.6 Visual Comparison Example

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

let x = 0;

// Toggle to switch between clearRect and overlay mode
let useClearRect = true;

function animate() {
  if (useClearRect) {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
  } else {
    ctx.fillStyle = 'rgba(255, 255, 255, 0.1)';
    ctx.fillRect(0, 0, canvas.width, canvas.height);
  }

  // Draw a moving square
  ctx.fillStyle = 'tomato';
  ctx.fillRect(x, 50, 50, 50);
```

```
  x += 2;
  if (x > canvas.width) x = -50;

  requestAnimationFrame(animate);
}

// Switch mode every 5 seconds to observe difference
setInterval(() => {
  useClearRect = !useClearRect;
  ctx.clearRect(0, 0, canvas.width, canvas.height);
}, 5000);

animate();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

let x = 0;

// Toggle to switch between clearRect and overlay mode
let useClearRect = true;

function animate() {
  if (useClearRect) {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
  } else {
    ctx.fillStyle = 'rgba(255, 255, 255, 0.1)';
    ctx.fillRect(0, 0, canvas.width, canvas.height);
  }

  // Draw a moving square
  ctx.fillStyle = 'tomato';
  ctx.fillRect(x, 50, 50, 50);

  x += 2;
  if (x > canvas.width) x = -50;

  requestAnimationFrame(animate);
}

// Switch mode every 5 seconds to observe difference
setInterval(() => {
  useClearRect = !useClearRect;
  ctx.clearRect(0, 0, canvas.width, canvas.height);
}, 5000);

animate();
```

```
</script>

</body>
</html>
```

- For the first 5 seconds, the square moves cleanly with no trails (`clearRect()` mode).
- After 5 seconds, the semi-transparent overlay creates a **motion trail** behind the moving square.
- This toggling visually demonstrates the effect of clearing vs not clearing fully.

7.2.7 Summary

- Always **clear the canvas** before drawing the next animation frame to avoid ghosting.
- Use `clearRect()` for a clean slate or semi-transparent overlays for creative motion trails.
- Optimize redraws by limiting cleared areas when necessary.
- Understanding and controlling how you clear and redraw is key for smooth and visually appealing animations.

Next, we'll learn how to create basic moving objects that utilize these clearing techniques!

7.3 Basic Moving Objects Animation

Animating objects on the canvas involves updating their **position** over time based on their **velocity** (speed and direction). This section guides you through creating a simple moving object that travels smoothly across the screen by updating its position each animation frame.

7.3.1 Understanding Motion Basics

To animate an object, you need to track:

- **Position**: Where the object currently is on the canvas (`x` and `y` coordinates).
- **Velocity**: How fast and in which direction the object moves (`vx` for horizontal speed, `vy` for vertical speed).

At each frame, you update the position by adding the velocity:

```
x = x + vx
y = y + vy
```

7.3.2 Example 1: Horizontal Motion

Here's a basic example of a square moving horizontally across the canvas:

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

let x = 0;       // Initial horizontal position
const y = 50;    // Fixed vertical position
const speed = 3; // Horizontal speed (pixels per frame)

function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Draw the square
  ctx.fillStyle = 'blue';
  ctx.fillRect(x, y, 50, 50);

  // Update position
  x += speed;

  // Reset position when moving off-screen
  if (x > canvas.width) {
    x = -50; // Start again from the left
  }

  requestAnimationFrame(animate);
}

animate();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

let x = 0;       // Initial horizontal position
const y = 50;    // Fixed vertical position
const speed = 3; // Horizontal speed (pixels per frame)

function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Draw the square
  ctx.fillStyle = 'blue';
  ctx.fillRect(x, y, 50, 50);

  // Update position
  x += speed;
```

```
  // Reset position when moving off-screen
  if (x > canvas.width) {
    x = -50; // Start again from the left
  }

  requestAnimationFrame(animate);
}

animate();
</script>

</body>
</html>
```

7.3.3 Example 2: Vertical and Diagonal Motion

By introducing vertical velocity (`vy`), you can move objects up/down or diagonally:

```
let x = 0;
let y = 0;
const vx = 2;   // Horizontal speed
const vy = 1.5; // Vertical speed

function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  ctx.fillStyle = 'green';
  ctx.fillRect(x, y, 40, 40);

  x += vx;
  y += vy;

  // Wrap around edges horizontally and vertically
  if (x > canvas.width) x = -40;
  if (y > canvas.height) y = -40;

  requestAnimationFrame(animate);
}

animate();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
```

```
  const ctx = canvas.getContext("2d");

let x = 0;
let y = 0;
const vx = 2;   // Horizontal speed
const vy = 1.5; // Vertical speed

function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  ctx.fillStyle = 'green';
  ctx.fillRect(x, y, 40, 40);

  x += vx;
  y += vy;

  // Wrap around edges horizontally and vertically
  if (x > canvas.width) x = -40;
  if (y > canvas.height) y = -40;

  requestAnimationFrame(animate);
}

animate();
</script>

</body>
</html>
```

7.3.4 Adding Boundaries and Direction Changes

You can add logic to **reverse direction** when the object hits the edges, creating a bouncing effect:

```
let x = 100;
let y = 100;
let vx = 3;
let vy = 2;

function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  ctx.fillStyle = 'red';
  ctx.fillRect(x, y, 50, 50);

  x += vx;
  y += vy;

  // Bounce off left/right edges
  if (x + 50 > canvas.width || x < 0) {
    vx = -vx; // Reverse horizontal velocity
  }
```

```
  // Bounce off top/bottom edges
  if (y + 50 > canvas.height || y < 0) {
    vy = -vy; // Reverse vertical velocity
  }

  requestAnimationFrame(animate);
}

animate();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

let x = 100;
let y = 100;
let vx = 3;
let vy = 2;

function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  ctx.fillStyle = 'red';
  ctx.fillRect(x, y, 50, 50);

  x += vx;
  y += vy;

  // Bounce off left/right edges
  if (x + 50 > canvas.width || x < 0) {
    vx = -vx; // Reverse horizontal velocity
  }

  // Bounce off top/bottom edges
  if (y + 50 > canvas.height || y < 0) {
    vy = -vy; // Reverse vertical velocity
  }

  requestAnimationFrame(animate);
}

animate();
</script>

</body>
</html>
```

7.3.5 Summary and Practice Tips

- Motion is controlled by updating **position** based on **velocity**.
- You can move objects horizontally, vertically, or diagonally by adjusting `vx` and `vy`.
- Use **conditionals** to detect boundaries and change direction or stop motion.
- Experiment by changing speed, direction, and boundary behaviors to build more dynamic animations.

Next, we'll combine these concepts into a complete bouncing ball animation!

7.4 Practical Example: Bouncing Ball Animation

In this section, we'll build a **bouncing ball animation** on the canvas that reacts realistically when hitting the edges. We'll simulate simple physics like reversing velocity on collisions and add a basic gravity effect for natural motion.

7.4.1 Full Example Code

```
<canvas id="canvas" width="500" height="300" style="border:1px solid #ccc;"></canvas>
```

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

// Ball properties
const ball = {
  x: 100,           // Horizontal position
  y: 50,            // Vertical position
  radius: 20,       // Ball radius
  vx: 4,            // Horizontal velocity
  vy: 2,            // Vertical velocity
  gravity: 0.3,     // Gravity acceleration
  friction: 0.8     // Energy loss on bounce
};

function drawBall() {
  ctx.beginPath();
  ctx.arc(ball.x, ball.y, ball.radius, 0, Math.PI * 2);
  ctx.fillStyle = 'orange';
  ctx.fill();
  ctx.strokeStyle = 'darkorange';
  ctx.stroke();
  ctx.closePath();
}

function animate() {
```

```
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  drawBall();

  // Apply gravity to vertical velocity
  ball.vy += ball.gravity;

  // Update ball position
  ball.x += ball.vx;
  ball.y += ball.vy;

  // Check collision with floor
  if (ball.y + ball.radius > canvas.height) {
    ball.y = canvas.height - ball.radius;    // Reset position at floor
    ball.vy = -ball.vy * ball.friction;      // Reverse and reduce velocity (bounce)
  }

  // Check collision with ceiling
  if (ball.y - ball.radius < 0) {
    ball.y = ball.radius;
    ball.vy = -ball.vy * ball.friction;
  }

  // Check collision with right wall
  if (ball.x + ball.radius > canvas.width) {
    ball.x = canvas.width - ball.radius;
    ball.vx = -ball.vx * ball.friction;
  }

  // Check collision with left wall
  if (ball.x - ball.radius < 0) {
    ball.x = ball.radius;
    ball.vx = -ball.vx * ball.friction;
  }

  requestAnimationFrame(animate);
}

// Start animation
animate();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

// Ball properties
const ball = {
  x: 100,           // Horizontal position
```

```
  y: 50,            // Vertical position
  radius: 20,       // Ball radius
  vx: 4,            // Horizontal velocity
  vy: 2,            // Vertical velocity
  gravity: 0.3,     // Gravity acceleration
  friction: 0.8     // Energy loss on bounce
};

function drawBall() {
  ctx.beginPath();
  ctx.arc(ball.x, ball.y, ball.radius, 0, Math.PI * 2);
  ctx.fillStyle = 'orange';
  ctx.fill();
  ctx.strokeStyle = 'darkorange';
  ctx.stroke();
  ctx.closePath();
}

function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  drawBall();

  // Apply gravity to vertical velocity
  ball.vy += ball.gravity;

  // Update ball position
  ball.x += ball.vx;
  ball.y += ball.vy;

  // Check collision with floor
  if (ball.y + ball.radius > canvas.height) {
    ball.y = canvas.height - ball.radius;    // Reset position at floor
    ball.vy = -ball.vy * ball.friction;      // Reverse and reduce velocity (bounce)
  }

  // Check collision with ceiling
  if (ball.y - ball.radius < 0) {
    ball.y = ball.radius;
    ball.vy = -ball.vy * ball.friction;
  }

  // Check collision with right wall
  if (ball.x + ball.radius > canvas.width) {
    ball.x = canvas.width - ball.radius;
    ball.vx = -ball.vx * ball.friction;
  }

  // Check collision with left wall
  if (ball.x - ball.radius < 0) {
    ball.x = ball.radius;
    ball.vx = -ball.vx * ball.friction;
  }

  requestAnimationFrame(animate);
}

// Start animation
```

```
animate();
</script>

</body>
</html>
```

7.4.2 Explanation

- The ball has a position `(x, y)`, radius, and velocity components `(vx, vy)`.
- **Gravity** is added each frame by increasing `vy`, pulling the ball downward.
- When the ball hits the **floor** or **ceiling**, the vertical velocity reverses and reduces by multiplying with friction, simulating energy loss.
- Similarly, horizontal velocity reverses and reduces when hitting **side walls**.
- The ball's position is constrained to stay within canvas bounds by resetting position to the edge when a collision is detected.
- `requestAnimationFrame` drives the smooth, continuous animation.

7.4.3 Experiment and Extend

- **Change the ball size:** Adjust `ball.radius` to see how size affects bounce behavior.
- **Modify gravity or friction:** Try smaller or larger values for more or less realistic motion.
- **Add multiple balls:** Store several ball objects in an array and animate them independently.
- **Introduce user controls:** Add sliders for gravity, speed, or ball count to make it interactive.

7.4.4 Summary

This bouncing ball example combines:

- Basic **motion equations** using position and velocity updates.
- Collision detection with canvas edges.
- Realistic bounce with velocity reversal and friction.
- Gravity for natural downward acceleration.

By mastering this example, you'll be ready to build more complex physics simulations and interactive animations on the canvas!

Chapter 8.

Advanced Animation Techniques

1. Frame Rate Control and Time-Based Animation
2. Easing Functions and Tweening
3. Multiple Object Animations and Interactions
4. Practical Example: Simple Particle System

8 Advanced Animation Techniques

8.1 Frame Rate Control and Time-Based Animation

When creating animations, one common challenge is ensuring smooth and consistent motion regardless of how fast or slow the animation loop runs. This section explains how to control animation timing effectively by using **time-based animation** and managing **frame rate variance**.

8.1.1 Fixed-Step vs. Time-Based Animation

Fixed-Step Animation

- Updates object positions by a fixed amount every frame.
- Assumes a constant frame rate (e.g., 60 frames per second).
- If the actual frame rate varies, animation speed will appear inconsistent — objects move faster on faster devices or slower on laggy ones.

Example fixed-step update:

```
x += 5; // move 5 pixels per frame, no timing adjustment
```

Time-Based Animation

- Updates movement based on the actual **elapsed time** between frames (`deltaTime`).
- Ensures consistent motion speed even if frame rates fluctuate.
- Object movement is proportional to time passed, not frames rendered.

Example time-based update:

```
x += speed * deltaTime; // speed is pixels per millisecond, deltaTime in ms
```

8.1.2 Using `requestAnimationFrame` Timestamps

The `requestAnimationFrame` callback receives a high-resolution **timestamp** parameter representing the current time in milliseconds.

You can calculate the time elapsed since the previous frame like this:

```
let lastTime = 0;

function animate(currentTime) {
  if (!lastTime) lastTime = currentTime; // initialize lastTime
```

```
  const deltaTime = currentTime - lastTime; // time elapsed since last frame (ms)
  lastTime = currentTime;

  // Use deltaTime for smooth animation updates here

  requestAnimationFrame(animate);
}

requestAnimationFrame(animate);
```

8.1.3 Why Use `deltaTime`?

When frame rates vary (e.g., due to system load), `deltaTime` changes accordingly. Using it to scale movements keeps objects moving at the same **real-world speed**, regardless of frame rate.

For example, if an object moves at 100 pixels per second:

```
const speed = 0.1; // pixels per millisecond

x += speed * deltaTime;
```

- If `deltaTime` is 16 ms (~60 FPS), object moves ~1.6 pixels this frame.
- If `deltaTime` spikes to 33 ms (~30 FPS), object moves ~3.3 pixels this frame.
- This keeps motion uniform in real time.

8.1.4 Example: Logging Frame Intervals and Smooth Movement

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

let x = 0;
const speed = 0.2; // pixels per millisecond

let lastTime = 0;

function animate(timestamp) {
  if (!lastTime) lastTime = timestamp;

  const deltaTime = timestamp - lastTime;
  lastTime = timestamp;

  // Clear canvas
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Move object based on deltaTime
```

```
  x += speed * deltaTime;

  // Wrap around
  if (x > canvas.width) x = -50;

  // Draw rectangle
  ctx.fillStyle = 'purple';
  ctx.fillRect(x, 50, 50, 50);

  // Log deltaTime (frame interval)
  console.log(`Frame interval: ${deltaTime.toFixed(2)} ms`);

  requestAnimationFrame(animate);
}

requestAnimationFrame(animate);
```

- This example logs frame intervals so you can see the variation.
- The rectangle moves smoothly at the same speed regardless of frame rate changes.
- Try throttling your browser or switching tabs to observe effects on `deltaTime`.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

let x = 0;
const speed = 0.2; // pixels per millisecond

let lastTime = 0;

function animate(timestamp) {
  if (!lastTime) lastTime = timestamp;

  const deltaTime = timestamp - lastTime;
  lastTime = timestamp;

  // Clear canvas
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Move object based on deltaTime
  x += speed * deltaTime;

  // Wrap around
  if (x > canvas.width) x = -50;

  // Draw rectangle
  ctx.fillStyle = 'purple';
  ctx.fillRect(x, 50, 50, 50);
```

```
  // Log deltaTime (frame interval)
  console.log(`Frame interval: ${deltaTime.toFixed(2)} ms`);

  requestAnimationFrame(animate);
}

requestAnimationFrame(animate);
</script>

</body>
</html>
```

8.1.5 Summary

- **Fixed-step animations** can produce inconsistent speeds if frame rate varies.
- Use the **timestamp parameter** from `requestAnimationFrame` to calculate `deltaTime` — the elapsed time since the last frame.
- Apply `deltaTime` to your motion updates to ensure **time-based, consistent animation speed**.
- This technique is fundamental for professional, smooth, and responsive animations.

Next, we'll explore how to create smooth easing effects and tweening animations using mathematical functions!

8.2 Easing Functions and Tweening

In real-world motion, objects rarely move at a constant speed. Instead, they **accelerate**, **decelerate**, or follow smooth, natural curves when starting or stopping. This is where **easing functions** and **tweening** come into play, making your animations feel polished and lifelike.

8.2.1 What is Easing?

Easing controls the rate of change of a parameter over time. Instead of linear (constant speed) motion, easing creates more interesting effects:

- **Linear:** Constant speed, no acceleration or deceleration.
- **Ease-In:** Starts slow and accelerates.
- **Ease-Out:** Starts fast and decelerates.
- **Ease-In-Out:** Combines ease-in and ease-out — slow start, fast middle, slow end.

8.2.2 Tweening Explained

Tweening (short for "in-betweening") means calculating intermediate values between a start and an end state over time, using easing to shape the transition.

For example, moving a shape's position from `x = 0` to `x = 100` over 1 second, applying easing to control speed.

8.2.3 Basic Tweening Functions

Here are some common easing functions, where `t` is the normalized time between 0 and 1:

```
// Linear: no easing, constant speed
function linear(t) {
  return t;
}

// Ease-in (quadratic): slow start, accelerating
function easeInQuad(t) {
  return t * t;
}

// Ease-out (quadratic): fast start, slowing down
function easeOutQuad(t) {
  return t * (2 - t);
}

// Ease-in-out (quadratic): slow start and end
function easeInOutQuad(t) {
  return t < 0.5 ? 2 * t * t : -1 + (4 - 2 * t) * t;
}
```

8.2.4 Applying Tweening in Animation

Here's a simple example animating a circle's horizontal position with different easing functions:

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

let startTime = null;
const duration = 2000; // 2 seconds

const startX = 50;
const endX = 350;
const y = 100;
const radius = 30;

// Choose easing function here:
```

```
const easing = easeInOutQuad;

function animate(timestamp) {
  if (!startTime) startTime = timestamp;

  let elapsed = timestamp - startTime;
  let t = Math.min(elapsed / duration, 1); // normalize time 0 to 1

  // Calculate eased progress
  let progress = easing(t);

  // Calculate current x position
  let currentX = startX + (endX - startX) * progress;

  // Clear canvas
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Draw circle
  ctx.beginPath();
  ctx.arc(currentX, y, radius, 0, Math.PI * 2);
  ctx.fillStyle = 'steelblue';
  ctx.fill();

  if (t < 1) {
    requestAnimationFrame(animate);
  }
}

requestAnimationFrame(animate);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Easing Animation Demo</title>
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

  let startTime = null;
  const duration = 2000; // 2 seconds

  const startX = 50;
  const endX = 350;
  const y = 100;
  const radius = 30;

  // Easing function: easeInOutQuad
  function easeInOutQuad(t) {
    return t < 0.5
      ? 2 * t * t
      : -1 + (4 - 2 * t) * t;
```

```
  }

  function animate(timestamp) {
    if (!startTime) startTime = timestamp;

    let elapsed = timestamp - startTime;
    let t = Math.min(elapsed / duration, 1); // normalize time 0 to 1

    // Calculate eased progress
    let progress = easeInOutQuad(t);

    // Calculate current x position
    let currentX = startX + (endX - startX) * progress;

    // Clear canvas
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    // Draw circle
    ctx.beginPath();
    ctx.arc(currentX, y, radius, 0, Math.PI * 2);
    ctx.fillStyle = 'steelblue';
    ctx.fill();

    if (t < 1) {
      requestAnimationFrame(animate);
    }
  }

  requestAnimationFrame(animate);
</script>

</body>
</html>
```

8.2.5 Visualizing Easing Curves

Try swapping out the `easing` function with `linear`, `easeInQuad`, `easeOutQuad`, or others to see how the motion changes.

8.2.6 Tweaking Easing Curves

- Experiment by adjusting the math inside easing functions.
- Use easing libraries like Robert Penner's easing functions or built-in options from animation frameworks.
- Combine multiple easing functions for complex effects.

8.2.7 Summary

- **Easing** makes animations feel smooth and natural by controlling speed over time.
- **Tweening** calculates intermediate animation states using easing.
- Implementing easing functions enhances realism and user experience.
- Try animating different properties like size, color, or position with easing to master this technique.

In the next section, we'll explore animating multiple objects and handling their interactions!

8.3 Multiple Object Animations and Interactions

Animating multiple objects simultaneously opens up a world of possibilities—from bubbles floating up the screen, to interactive games with many sprites. In this section, we'll explore how to manage many independent objects efficiently, update their states, render them in the correct order, and handle basic interactions.

8.3.1 Managing Multiple Objects

The most common way to handle multiple animated objects is to store them in an **array** or use an **object-oriented** approach where each object has its own properties and methods.

8.3.2 Example: Array of Bubble Objects

Each bubble can have properties like position, velocity, size, and color. In each animation frame, you update all bubbles and then draw them.

8.3.3 Structure of the Animation Loop

1. **Update Phase:** Update the position, velocity, or other state properties for each object.
2. **Render Phase:** Draw all objects on the canvas in the desired order.
3. **Interaction Handling:** Check for collisions or user interactions (e.g., mouse clicks).

8.3.4 Demo: Animated Bubbles with Interaction

```
<canvas id="canvas" width="500" height="400" style="border:1px solid #ccc;"></canvas>
```

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

class Bubble {
  constructor(x, y, radius, vx, vy, color) {
    this.x = x;
    this.y = y;
    this.radius = radius;
    this.vx = vx;
    this.vy = vy;
    this.color = color;
  }

  update() {
    this.x += this.vx;
    this.y += this.vy;

    // Bounce off walls
    if (this.x + this.radius > canvas.width || this.x - this.radius < 0) {
      this.vx = -this.vx;
    }
    if (this.y + this.radius > canvas.height || this.y - this.radius < 0) {
      this.vy = -this.vy;
    }
  }

  draw() {
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, Math.PI * 2);
    ctx.fillStyle = this.color;
    ctx.fill();
    ctx.strokeStyle = 'black';
    ctx.stroke();
  }

  isPointInside(px, py) {
    const dx = px - this.x;
    const dy = py - this.y;
    return dx * dx + dy * dy <= this.radius * this.radius;
  }
}

// Create bubbles
const bubbles = [];
for (let i = 0; i < 20; i++) {
  bubbles.push(
    new Bubble(
      Math.random() * canvas.width,
      Math.random() * canvas.height,
      15 + Math.random() * 10,
      (Math.random() - 0.5) * 4,
      (Math.random() - 0.5) * 4,
```

```
      `hsl(${Math.random() * 360}, 70%, 60%)`
    )
  );
}

// Handle clicks to pop bubbles
canvas.addEventListener('click', (e) => {
  const rect = canvas.getBoundingClientRect();
  const mouseX = e.clientX - rect.left;
  const mouseY = e.clientY - rect.top;

  for (let i = bubbles.length - 1; i >= 0; i--) {
    if (bubbles[i].isPointInside(mouseX, mouseY)) {
      bubbles.splice(i, 1); // Remove popped bubble
      break;
    }
  }
});

function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Update and draw all bubbles
  bubbles.forEach(bubble => {
    bubble.update();
    bubble.draw();
  });

  requestAnimationFrame(animate);
}

animate();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

class Bubble {
  constructor(x, y, radius, vx, vy, color) {
    this.x = x;
    this.y = y;
    this.radius = radius;
    this.vx = vx;
    this.vy = vy;
    this.color = color;
  }

  update() {
```

```
    this.x += this.vx;
    this.y += this.vy;

    // Bounce off walls
    if (this.x + this.radius > canvas.width || this.x - this.radius < 0) {
      this.vx = -this.vx;
    }
    if (this.y + this.radius > canvas.height || this.y - this.radius < 0) {
      this.vy = -this.vy;
    }
  }

  draw() {
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, Math.PI * 2);
    ctx.fillStyle = this.color;
    ctx.fill();
    ctx.strokeStyle = 'black';
    ctx.stroke();
  }

  isPointInside(px, py) {
    const dx = px - this.x;
    const dy = py - this.y;
    return dx * dx + dy * dy <= this.radius * this.radius;
  }
}

// Create bubbles
const bubbles = [];
for (let i = 0; i < 20; i++) {
  bubbles.push(
    new Bubble(
      Math.random() * canvas.width,
      Math.random() * canvas.height,
      15 + Math.random() * 10,
      (Math.random() - 0.5) * 4,
      (Math.random() - 0.5) * 4,
      `hsl(${Math.random() * 360}, 70%, 60%)`
    )
  );
}

// Handle clicks to pop bubbles
canvas.addEventListener('click', (e) => {
  const rect = canvas.getBoundingClientRect();
  const mouseX = e.clientX - rect.left;
  const mouseY = e.clientY - rect.top;

  for (let i = bubbles.length - 1; i >= 0; i--) {
    if (bubbles[i].isPointInside(mouseX, mouseY)) {
      bubbles.splice(i, 1); // Remove popped bubble
      break;
    }
  }
});

function animate() {
```

```
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Update and draw all bubbles
  bubbles.forEach(bubble => {
    bubble.update();
    bubble.draw();
  });

  requestAnimationFrame(animate);
}

animate();
</script>

</body>
</html>
```

8.3.5 Key Points in This Example

- **Object-oriented design:** Each bubble encapsulates its properties and behavior.
- **Array management:** All bubbles are stored in an array for easy iteration.
- **Collision with canvas edges:** Velocity reverses on boundary hits.
- **Interaction:** Clicking on a bubble removes it from the array.
- **Efficient looping:** Updating and rendering are done separately but in sequence for clarity and performance.

8.3.6 Organizing for Performance and Scalability

- **Batch updates and renders:** Avoid redundant calculations inside loops.
- **Spatial partitioning:** For many objects, optimize collision checks using grids or quadtrees.
- **Use `requestAnimationFrame`:** Synchronizes drawing with browser refresh for smooth animations.
- **Offscreen canvases:** For complex static backgrounds or repeated patterns.

8.3.7 Summary

Animating multiple objects requires:

- Managing collections (arrays or objects) of independent entities.
- Updating each object's state and rendering them every frame.
- Handling interactions like collisions or user input.

- Writing organized, modular code to keep updates and rendering clear and maintainable.

Next, we'll put these concepts together in a fun **Simple Particle System** example that demonstrates many moving particles with interaction and effects!

8.4 Practical Example: Simple Particle System

Particle systems are a classic way to create visually appealing effects like fireworks, smoke, rain, or explosions. In this example, we'll build a simple particle system where particles are emitted from a point, move in random directions, and fade out over time.

8.4.1 Whats Happening Here?

- Particles are created at an **emitter point**.
- Each particle has random velocity, color, and lifespan.
- Particles move each frame, gradually becoming more transparent.
- When a particle's lifespan ends, it is removed from the system.
- We use an array to store particles and manage their life cycle.

8.4.2 Step-by-Step Code

```
<canvas id="canvas" width="500" height="400" style="border:1px solid #ccc;"></canvas>
```

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

class Particle {
  constructor(x, y) {
    this.x = x;
    this.y = y;
    this.radius = 5 + Math.random() * 5;
    this.vx = (Math.random() - 0.5) * 4; // random velocity X
    this.vy = (Math.random() - 0.5) * 4; // random velocity Y
    this.alpha = 1; // opacity
    this.life = 100 + Math.random() * 50; // lifespan in frames
    this.color = `hsl(${Math.random() * 360}, 70%, 60%)`;
  }

  update() {
    this.x += this.vx;
    this.y += this.vy;
```

```
    // Optional: Gravity effect
    this.vy += 0.05;

    // Fade out
    this.alpha -= 1 / this.life;

    // Reduce radius slowly (optional)
    this.radius *= 0.98;
  }

  draw() {
    ctx.save();
    ctx.globalAlpha = Math.max(this.alpha, 0); // prevent negative opacity
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, Math.PI * 2);
    ctx.fillStyle = this.color;
    ctx.fill();
    ctx.restore();
  }

  isAlive() {
    return this.alpha > 0 && this.radius > 0.5;
  }
}

// Particle system array
const particles = [];
const emitterX = canvas.width / 2;
const emitterY = canvas.height / 2;

function emitParticles(count) {
  for (let i = 0; i < count; i++) {
    particles.push(new Particle(emitterX, emitterY));
  }
}

function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Emit new particles continuously
  emitParticles(3);

  // Update and draw particles
  for (let i = particles.length - 1; i >= 0; i--) {
    const p = particles[i];
    p.update();
    p.draw();

    // Remove dead particles
    if (!p.isAlive()) {
      particles.splice(i, 1);
    }
  }

  requestAnimationFrame(animate);
}

animate();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="500" height="400" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

class Particle {
  constructor(x, y) {
    this.x = x;
    this.y = y;
    this.radius = 5 + Math.random() * 5;
    this.vx = (Math.random() - 0.5) * 4; // random velocity X
    this.vy = (Math.random() - 0.5) * 4; // random velocity Y
    this.alpha = 1; // opacity
    this.life = 100 + Math.random() * 50; // lifespan in frames
    this.color = `hsl(${Math.random() * 360}, 70%, 60%)`;
  }

  update() {
    this.x += this.vx;
    this.y += this.vy;

    // Optional: Gravity effect
    this.vy += 0.05;

    // Fade out
    this.alpha -= 1 / this.life;

    // Reduce radius slowly (optional)
    this.radius *= 0.98;
  }

  draw() {
    ctx.save();
    ctx.globalAlpha = Math.max(this.alpha, 0); // prevent negative opacity
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, Math.PI * 2);
    ctx.fillStyle = this.color;
    ctx.fill();
    ctx.restore();
  }

  isAlive() {
    return this.alpha > 0 && this.radius > 0.5;
  }
}

// Particle system array
const particles = [];
```

```
const emitterX = canvas.width / 2;
const emitterY = canvas.height / 2;

function emitParticles(count) {
  for (let i = 0; i < count; i++) {
    particles.push(new Particle(emitterX, emitterY));
  }
}

function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Emit new particles continuously
  emitParticles(3);

  // Update and draw particles
  for (let i = particles.length - 1; i >= 0; i--) {
    const p = particles[i];
    p.update();
    p.draw();

    // Remove dead particles
    if (!p.isAlive()) {
      particles.splice(i, 1);
    }
  }

  requestAnimationFrame(animate);
}

animate();
</script>

</body>
</html>
```

8.4.3 How It Works

- **Particle creation:** The `emitParticles` function adds new particles every frame at the emitter position.
- **Randomness:** Each particle has random velocity, size, color, and lifespan for a natural look.
- **Movement and gravity:** Velocity changes position; gravity pulls particles downward.
- **Fading out:** The alpha transparency gradually decreases until the particle disappears.
- **Cleanup:** Dead particles are removed from the array to keep the system efficient.

8.4.4 Experiment and Extend

- **Gravity:** Modify gravity strength (`this.vy += 0.05`) or invert it for floating effects.
- **Explosion:** Emit many particles once in a burst instead of continuously.
- **Mouse emitter:** Update `emitterX` and `emitterY` based on mouse position.
- **Color & size variations:** Change how colors and sizes evolve over time.
- **Add collision or bouncing:** Make particles bounce off canvas edges.

8.4.5 Summary

This simple particle system illustrates core animation concepts:

- Managing multiple independent objects in an array.
- Using randomness for natural variation.
- Updating position, velocity, and opacity each frame.
- Removing expired particles to optimize performance.

Particle systems are powerful building blocks for visual effects — tweak parameters and build on this foundation to create impressive animations!

Chapter 9.

User Interaction and Event Handling

1. Capturing Mouse and Keyboard Events
2. Interactive Drawing Applications
3. Drag-and-Drop on Canvas
4. Practical Example: Drawing and Editing with Mouse

9 User Interaction and Event Handling

9.1 Capturing Mouse and Keyboard Events

User interaction is key to building engaging and dynamic canvas applications. This section covers how to capture mouse and keyboard events, translate coordinates correctly, and use these inputs to control objects on the canvas.

9.1.1 Registering Event Listeners

In JavaScript, the primary way to respond to user input is by **registering event listeners** with the `addEventListener()` method. You can attach listeners to the `<canvas>` element itself or to the `window` or `document` objects for keyboard events.

9.1.2 Common Events

Event Type	Description	Typical Target
`mousedown`	Mouse button pressed	Canvas or Window
`mouseup`	Mouse button released	Canvas or Window
`mousemove`	Mouse moved	Canvas
`keydown`	Key pressed down	Window or Document
`keyup`	Key released	Window or Document

9.1.3 Example: Registering Mouse Events on Canvas

```
const canvas = document.getElementById('canvas');

canvas.addEventListener('mousedown', (event) => {
  console.log('Mouse down at', event.clientX, event.clientY);
});

canvas.addEventListener('mousemove', (event) => {
  console.log('Mouse moved at', event.clientX, event.clientY);
});
```

9.1.4 Translating Mouse Coordinates Relative to Canvas

Mouse event coordinates like `event.clientX` and `event.clientY` are relative to the **viewport** (the browser window), not the canvas itself. To get the mouse position **inside the canvas**, you must adjust for the canvas's position on the page.

9.1.5 How to Translate Coordinates

```
function getMousePos(canvas, event) {
  const rect = canvas.getBoundingClientRect(); // Get canvas position and size
  return {
    x: event.clientX - rect.left,
    y: event.clientY - rect.top
  };
}

canvas.addEventListener('mousedown', (event) => {
  const pos = getMousePos(canvas, event);
  console.log('Mouse position relative to canvas:', pos.x, pos.y);
});
```

This calculation accounts for the canvas's position, borders, and scrolling.

9.1.6 Capturing Keyboard Input

Keyboard events are usually registered on the `window` or `document` object because the canvas does not automatically receive keyboard focus.

9.1.7 Example: Using Keyboard Events to Move an Object

```
let x = 50;
let y = 50;

window.addEventListener('keydown', (event) => {
  switch(event.key) {
    case 'ArrowUp':
      y -= 10;
      break;
    case 'ArrowDown':
      y += 10;
      break;
    case 'ArrowLeft':
```

```
      x -= 10;
      break;
    case 'ArrowRight':
      x += 10;
      break;
  }
  console.log(`Object position: (${x}, ${y})`);
  // You can redraw your object at the new position here
});
```

9.1.8 Ensuring Canvas Can Receive Keyboard Input

If you need the canvas element to capture keyboard events directly, it needs to be **focusable**. You can make the canvas focusable by adding a `tabindex` attribute:

```
<canvas id="canvas" width="400" height="300" tabindex="0"></canvas>
```

Then, you can add keyboard listeners to the canvas:

```
canvas.addEventListener('keydown', (event) => {
  console.log('Key pressed on canvas:', event.key);
});
canvas.focus(); // To set focus programmatically when needed
```

9.1.9 Summary

- Use `addEventListener()` to listen for mouse and keyboard events.
- Translate mouse coordinates from viewport to canvas-relative coordinates for accurate interaction.
- Keyboard events are usually captured on `window` or `document` but can be directed to canvas with `tabindex`.
- Combine these event inputs to create interactive canvas applications controlled by mouse and keyboard.

In the next section, we will see how to use these interactions to build an interactive drawing application!

9.2 Interactive Drawing Applications

9.2.1 Interactive Drawing Applications

One of the most engaging uses of the HTML5 canvas is creating interactive drawing applications. In this section, we'll build a basic app that allows users to draw freehand lines by clicking and dragging the mouse. This example introduces core concepts like tracking mouse state, storing previous coordinates, and drawing paths dynamically.

9.2.2 Core Concepts: Tracking Mouse State and Coordinates

To draw continuously as the user drags the mouse, you need to:

- Track **whether the user is currently drawing** (a boolean flag like `isDrawing`).
- Store the **previous mouse coordinates** so you can draw a line segment from the last point to the current point.
- Update the canvas with new line segments as the mouse moves.

9.2.3 Basic Drawing Application Example

```
<canvas id="canvas" width="600" height="400" style="border:1px solid #ccc;"></canvas>
```

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

let isDrawing = false;
let lastX = 0;
let lastY = 0;

// Mouse down - start drawing
canvas.addEventListener('mousedown', (e) => {
  const rect = canvas.getBoundingClientRect();
  lastX = e.clientX - rect.left;
  lastY = e.clientY - rect.top;
  isDrawing = true;
});

// Mouse move - draw if mouse is down
canvas.addEventListener('mousemove', (e) => {
  if (!isDrawing) return;

  const rect = canvas.getBoundingClientRect();
  const currentX = e.clientX - rect.left;
  const currentY = e.clientY - rect.top;
```

```
  ctx.beginPath();
  ctx.moveTo(lastX, lastY);      // Start from previous point
  ctx.lineTo(currentX, currentY); // Draw line to current point
  ctx.stroke();

  lastX = currentX; // Update last positions
  lastY = currentY;
});

// Mouse up or leave - stop drawing
canvas.addEventListener('mouseup', () => {
  isDrawing = false;
});
canvas.addEventListener('mouseout', () => {
  isDrawing = false;
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

let isDrawing = false;
let lastX = 0;
let lastY = 0;

// Mouse down - start drawing
canvas.addEventListener('mousedown', (e) => {
  const rect = canvas.getBoundingClientRect();
  lastX = e.clientX - rect.left;
  lastY = e.clientY - rect.top;
  isDrawing = true;
});

// Mouse move - draw if mouse is down
canvas.addEventListener('mousemove', (e) => {
  if (!isDrawing) return;

  const rect = canvas.getBoundingClientRect();
  const currentX = e.clientX - rect.left;
  const currentY = e.clientY - rect.top;

  ctx.beginPath();
  ctx.moveTo(lastX, lastY);      // Start from previous point
  ctx.lineTo(currentX, currentY); // Draw line to current point
  ctx.stroke();

  lastX = currentX; // Update last positions
  lastY = currentY;
```

```
});

// Mouse up or leave - stop drawing
canvas.addEventListener('mouseup', () => {
  isDrawing = false;
});
canvas.addEventListener('mouseout', () => {
  isDrawing = false;
});
</script>

</body>
</html>
```

9.2.4 How This Works

- **`mousedown` event:** Starts the drawing process, captures the initial coordinates, and sets `isDrawing = true`.
- **`mousemove` event:** If `isDrawing` is true, draws a line segment from the last known point to the current mouse position.
- **`mouseup` and `mouseout` events:** End the drawing by setting `isDrawing` to false.

9.2.5 Enhancements to Explore

This basic app can be expanded with many useful features:

- **Color Pickers:** Allow users to select stroke colors.
- **Stroke Width:** Let users adjust line thickness dynamically with sliders.
- **Undo/Redo:** Maintain a stack of canvas states or paths to enable undoing mistakes.
- **Shape Tools:** Add buttons to draw rectangles, circles, or other shapes.
- **Save and Export:** Save drawings as images or JSON data.

9.2.6 Why Canvas for Drawing?

Unlike SVG or HTML elements which represent vector shapes and DOM nodes, canvas works with pixel-based rendering. This makes canvas ideal for:

- High-performance freehand drawing.
- Complex pixel manipulation and image editing.
- Games and applications requiring fast, dynamic graphics updates.

However, since canvas pixels are just bitmap data, shapes are not independently selectable

after drawing, which is why state management or redrawing logic is important for interactive apps.

9.2.7 Summary

This simple drawing app demonstrates:

- Tracking mouse state to enable smooth drawing.
- Using canvas path methods to draw continuous lines.
- Translating mouse coordinates for accurate rendering on canvas.

Try modifying the code to add color selection or brush sizes to make the app your own! In the next sections, we will look at drag-and-drop and more complex editing features.

9.3 Drag-and-Drop on Canvas

Interactivity on the canvas goes beyond just drawing — you can also implement **drag-and-drop functionality** to move shapes or objects around dynamically. This section explains how to detect clicks inside shapes, track mouse movement, and update the positions of shapes as they are dragged.

9.3.1 Core Concepts for Drag-and-Drop

- **Hit Detection:** Determine if a mouse click is inside a shape. This often uses bounding boxes or geometric checks.
- **Tracking Drag State:** Use flags like `isDragging` and track which shape is currently being moved.
- **Coordinate Tracking:** Save the offset between the mouse position and the shape's origin to keep dragging smooth.
- **Updating Position:** As the mouse moves, update the shape's coordinates and redraw the canvas.

9.3.2 Example: Moving Rectangles on Canvas

Let's build a simple example where the user can drag rectangles around the canvas.

```
<canvas id="canvas" width="600" height="400" style="border:1px solid #ccc;"></canvas>
```

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const rectangles = [
  { x: 50, y: 60, width: 100, height: 80, color: 'tomato' },
  { x: 200, y: 150, width: 120, height: 90, color: 'steelblue' }
];

let isDragging = false;
let dragIndex = null;
let offsetX, offsetY;

// Draw all rectangles
function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  rectangles.forEach(rect => {
    ctx.fillStyle = rect.color;
    ctx.fillRect(rect.x, rect.y, rect.width, rect.height);
  });
}

draw();

// Utility: Check if point is inside a rectangle
function isInsideRect(x, y, rect) {
  return x >= rect.x &&
         x <= rect.x + rect.width &&
         y >= rect.y &&
         y <= rect.y + rect.height;
}

// Mouse down: Check if clicking inside any rectangle to start dragging
canvas.addEventListener('mousedown', (e) => {
  const rect = canvas.getBoundingClientRect();
  const mouseX = e.clientX - rect.left;
  const mouseY = e.clientY - rect.top;

  for (let i = 0; i < rectangles.length; i++) {
    if (isInsideRect(mouseX, mouseY, rectangles[i])) {
      isDragging = true;
      dragIndex = i;
      offsetX = mouseX - rectangles[i].x;
      offsetY = mouseY - rectangles[i].y;
      break;
    }
  }
});

// Mouse move: If dragging, update rectangle position and redraw
canvas.addEventListener('mousemove', (e) => {
  if (!isDragging) return;

  const rect = canvas.getBoundingClientRect();
  const mouseX = e.clientX - rect.left;
  const mouseY = e.clientY - rect.top;
```

```
  rectangles[dragIndex].x = mouseX - offsetX;
  rectangles[dragIndex].y = mouseY - offsetY;

  draw();
});

// Mouse up: Stop dragging
canvas.addEventListener('mouseup', () => {
  isDragging = false;
  dragIndex = null;
});

// Mouse leave: Cancel dragging if mouse leaves canvas
canvas.addEventListener('mouseout', () => {
  isDragging = false;
  dragIndex = null;
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="600" height="400" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

const rectangles = [
  { x: 50, y: 60, width: 100, height: 80, color: 'tomato' },
  { x: 200, y: 150, width: 120, height: 90, color: 'steelblue' }
];

let isDragging = false;
let dragIndex = null;
let offsetX, offsetY;

// Draw all rectangles
function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  rectangles.forEach(rect => {
    ctx.fillStyle = rect.color;
    ctx.fillRect(rect.x, rect.y, rect.width, rect.height);
  });
}

draw();

// Utility: Check if point is inside a rectangle
function isInsideRect(x, y, rect) {
  return x >= rect.x &&
         x <= rect.x + rect.width &&
         y >= rect.y &&
         y <= rect.y + rect.height;
```

```
}

// Mouse down: Check if clicking inside any rectangle to start dragging
canvas.addEventListener('mousedown', (e) => {
  const rect = canvas.getBoundingClientRect();
  const mouseX = e.clientX - rect.left;
  const mouseY = e.clientY - rect.top;

  for (let i = 0; i < rectangles.length; i++) {
    if (isInsideRect(mouseX, mouseY, rectangles[i])) {
      isDragging = true;
      dragIndex = i;
      offsetX = mouseX - rectangles[i].x;
      offsetY = mouseY - rectangles[i].y;
      break;
    }
  }
});

// Mouse move: If dragging, update rectangle position and redraw
canvas.addEventListener('mousemove', (e) => {
  if (!isDragging) return;

  const rect = canvas.getBoundingClientRect();
  const mouseX = e.clientX - rect.left;
  const mouseY = e.clientY - rect.top;

  rectangles[dragIndex].x = mouseX - offsetX;
  rectangles[dragIndex].y = mouseY - offsetY;

  draw();
});

// Mouse up: Stop dragging
canvas.addEventListener('mouseup', () => {
  isDragging = false;
  dragIndex = null;
});

// Mouse leave: Cancel dragging if mouse leaves canvas
canvas.addEventListener('mouseout', () => {
  isDragging = false;
  dragIndex = null;
});
</script>

</body>
</html>
```

9.3.3 How This Works

- When the user presses the mouse down (`mousedown`), we check if the mouse is inside any rectangle using `isInsideRect()`.

- If a rectangle is selected, we store the index and the offset of the mouse from the rectangle's top-left corner.
- On `mousemove`, if dragging, the rectangle's position is updated based on the current mouse position minus the offset to keep the drag smooth.
- The canvas is cleared and redrawn after every move to show the updated position.
- Dragging stops on `mouseup` or when the mouse leaves the canvas.

9.3.4 Enhancements to Consider

- **Snapping to Grid:** Snap dragged objects to fixed grid points by rounding positions.
- **Bounding Constraints:** Prevent shapes from being dragged outside the canvas edges.
- **Multiple Shapes & Selection:** Add selection outlines, multi-select, or layering.
- **Dragging Other Shapes:** Extend hit detection for circles, polygons, or complex shapes using geometric math or libraries.

9.3.5 Summary

Drag-and-drop on canvas involves:

- Detecting if clicks occur inside shapes (hit detection).
- Tracking mouse movement to update positions.
- Redrawing the canvas to reflect changes in real time.

This functionality lays the foundation for interactive editors, games, and custom UI components. Next, we'll explore combining these techniques into a practical drawing and editing tool.

9.4 Practical Example: Drawing and Editing with Mouse

In this section, we combine the concepts from previous lessons to create a **simple interactive canvas tool** that allows users to:

- Draw rectangles by clicking and dragging,
- Select and move existing rectangles,
- Delete selected rectangles with a keyboard key.

This example emphasizes modular code design—separating event handling, shape management, and rendering—making it easier to extend or customize later.

9.4.1 Features Overview

- **Drawing mode:** Click and drag to create new rectangles.
- **Select mode:** Click a shape to select it; drag to move it.
- **Delete:** Press the `Delete` or `Backspace` key to remove the selected shape.
- **Visual feedback:** Highlight the selected shape.
- **Simple toolbar:** Toggle between draw and select modes.

9.4.2 HTML Structure

```
<div>
  <button id="drawBtn">Draw</button>
  <button id="selectBtn">Select</button>
</div>
<canvas id="canvas" width="700" height="400" style="border:1px solid #ccc;"></canvas>
```

9.4.3 JavaScript Code

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const drawBtn = document.getElementById('drawBtn');
const selectBtn = document.getElementById('selectBtn');

let mode = 'draw'; // Current mode: 'draw' or 'select'
let shapes = [];   // Array to store rectangles
let selectedShapeIndex = null;

let isDrawing = false;
let startX = 0;
let startY = 0;

let isDragging = false;
let dragOffsetX = 0;
let dragOffsetY = 0;

// Utility: Check if a point is inside a rectangle
function isInsideRect(x, y, rect) {
  return x >= rect.x &&
         x <= rect.x + rect.width &&
         y >= rect.y &&
         y <= rect.y + rect.height;
}

// Draw all shapes, highlighting selected one
function draw() {
```

```
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  shapes.forEach((rect, i) => {
    ctx.fillStyle = rect.color;
    ctx.fillRect(rect.x, rect.y, rect.width, rect.height);

    if (i === selectedShapeIndex) {
      ctx.strokeStyle = 'orange';
      ctx.lineWidth = 3;
      ctx.strokeRect(rect.x, rect.y, rect.width, rect.height);
    }
  });
}

// Set current mode and update button styles
function setMode(newMode) {
  mode = newMode;
  drawBtn.disabled = (mode === 'draw');
  selectBtn.disabled = (mode === 'select');
  selectedShapeIndex = null;
  draw();
}

drawBtn.addEventListener('click', () => setMode('draw'));
selectBtn.addEventListener('click', () => setMode('select'));

// Mouse down event handler
canvas.addEventListener('mousedown', (e) => {
  const rect = canvas.getBoundingClientRect();
  const mouseX = e.clientX - rect.left;
  const mouseY = e.clientY - rect.top;

  if (mode === 'draw') {
    // Start drawing new rectangle
    isDrawing = true;
    startX = mouseX;
    startY = mouseY;
  } else if (mode === 'select') {
    // Check if clicking on any existing shape (top-down)
    selectedShapeIndex = null;
    for (let i = shapes.length - 1; i >= 0; i--) {
      if (isInsideRect(mouseX, mouseY, shapes[i])) {
        selectedShapeIndex = i;
        dragOffsetX = mouseX - shapes[i].x;
        dragOffsetY = mouseY - shapes[i].y;
        isDragging = true;
        break;
      }
    }
    draw();
  }
});

// Mouse move event handler
canvas.addEventListener('mousemove', (e) => {
  if (mode === 'draw' && isDrawing) {
    const rect = canvas.getBoundingClientRect();
    const mouseX = e.clientX - rect.left;
```

```
    const mouseY = e.clientY - rect.top;

    // Draw a preview rectangle (clear and redraw all)
    draw();
    ctx.strokeStyle = 'black';
    ctx.lineWidth = 1;
    ctx.strokeRect(startX, startY, mouseX - startX, mouseY - startY);
  } else if (mode === 'select' && isDragging && selectedShapeIndex !== null) {
    const rect = canvas.getBoundingClientRect();
    const mouseX = e.clientX - rect.left;
    const mouseY = e.clientY - rect.top;

    // Move the selected shape
    shapes[selectedShapeIndex].x = mouseX - dragOffsetX;
    shapes[selectedShapeIndex].y = mouseY - dragOffsetY;
    draw();
  }
});

// Mouse up event handler
canvas.addEventListener('mouseup', (e) => {
  if (mode === 'draw' && isDrawing) {
    const rect = canvas.getBoundingClientRect();
    const mouseX = e.clientX - rect.left;
    const mouseY = e.clientY - rect.top;

    // Create new rectangle and add to shapes array
    const newRect = {
      x: Math.min(startX, mouseX),
      y: Math.min(startY, mouseY),
      width: Math.abs(mouseX - startX),
      height: Math.abs(mouseY - startY),
      color: 'skyblue'
    };

    // Only add if rectangle has size
    if (newRect.width > 5 && newRect.height > 5) {
      shapes.push(newRect);
    }
    isDrawing = false;
    draw();
  } else if (mode === 'select' && isDragging) {
    isDragging = false;
  }
});

// Keyboard event for deleting selected shape
window.addEventListener('keydown', (e) => {
  if ((e.key === 'Delete' || e.key === 'Backspace') && selectedShapeIndex !== null) {
    shapes.splice(selectedShapeIndex, 1);
    selectedShapeIndex = null;
    draw();
  }
});

// Initialize buttons and drawing
setMode('draw');
draw();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Canvas Draw and Select</title>
  <style>
    body {
      font-family: sans-serif;
      padding: 1rem;
    }
    canvas {
      margin-top: 1rem;
      border: 1px solid #ccc;
      display: block;
    }
    button {
      margin-right: 10px;
    }
  </style>
</head>
<body>

<h3>Canvas: Draw & Select Rectangles</h3>

<div>
  <button id="drawBtn">Draw</button>
  <button id="selectBtn">Select</button>
</div>

<canvas id="canvas" width="700" height="400"></canvas>

<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');

  const drawBtn = document.getElementById('drawBtn');
  const selectBtn = document.getElementById('selectBtn');

  let mode = 'draw';
  let shapes = [];
  let selectedShapeIndex = null;

  let isDrawing = false;
  let startX = 0;
  let startY = 0;

  let isDragging = false;
  let dragOffsetX = 0;
  let dragOffsetY = 0;

  function isInsideRect(x, y, rect) {
    return x >= rect.x &&
           x <= rect.x + rect.width &&
           y >= rect.y &&
           y <= rect.y + rect.height;
  }

  function draw() {
```

```
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  shapes.forEach((rect, i) => {
    ctx.fillStyle = rect.color;
    ctx.fillRect(rect.x, rect.y, rect.width, rect.height);

    if (i === selectedShapeIndex) {
      ctx.strokeStyle = 'orange';
      ctx.lineWidth = 3;
      ctx.strokeRect(rect.x, rect.y, rect.width, rect.height);
    }
  });
}

function setMode(newMode) {
  mode = newMode;
  drawBtn.disabled = (mode === 'draw');
  selectBtn.disabled = (mode === 'select');
  selectedShapeIndex = null;
  draw();
}

drawBtn.addEventListener('click', () => setMode('draw'));
selectBtn.addEventListener('click', () => setMode('select'));

canvas.addEventListener('mousedown', (e) => {
  const rect = canvas.getBoundingClientRect();
  const mouseX = e.clientX - rect.left;
  const mouseY = e.clientY - rect.top;

  if (mode === 'draw') {
    isDrawing = true;
    startX = mouseX;
    startY = mouseY;
  } else if (mode === 'select') {
    selectedShapeIndex = null;
    for (let i = shapes.length - 1; i >= 0; i--) {
      if (isInsideRect(mouseX, mouseY, shapes[i])) {
        selectedShapeIndex = i;
        dragOffsetX = mouseX - shapes[i].x;
        dragOffsetY = mouseY - shapes[i].y;
        isDragging = true;
        break;
      }
    }
    draw();
  }
});

canvas.addEventListener('mousemove', (e) => {
  const rect = canvas.getBoundingClientRect();
  const mouseX = e.clientX - rect.left;
  const mouseY = e.clientY - rect.top;

  if (mode === 'draw' && isDrawing) {
    draw();
    ctx.strokeStyle = 'black';
    ctx.lineWidth = 1;
```

```
      ctx.strokeRect(startX, startY, mouseX - startX, mouseY - startY);
    } else if (mode === 'select' && isDragging && selectedShapeIndex !== null) {
      shapes[selectedShapeIndex].x = mouseX - dragOffsetX;
      shapes[selectedShapeIndex].y = mouseY - dragOffsetY;
      draw();
    }
  });

  canvas.addEventListener('mouseup', (e) => {
    const rect = canvas.getBoundingClientRect();
    const mouseX = e.clientX - rect.left;
    const mouseY = e.clientY - rect.top;

    if (mode === 'draw' && isDrawing) {
      const newRect = {
        x: Math.min(startX, mouseX),
        y: Math.min(startY, mouseY),
        width: Math.abs(mouseX - startX),
        height: Math.abs(mouseY - startY),
        color: 'skyblue'
      };
      if (newRect.width > 5 && newRect.height > 5) {
        shapes.push(newRect);
      }
      isDrawing = false;
      draw();
    } else if (mode === 'select' && isDragging) {
      isDragging = false;
    }
  });

  window.addEventListener('keydown', (e) => {
    if ((e.key === 'Delete' || e.key === 'Backspace') && selectedShapeIndex !== null) {
      shapes.splice(selectedShapeIndex, 1);
      selectedShapeIndex = null;
      draw();
    }
  });

  setMode('draw');
  draw();
</script>

</body>
</html>
```

9.4.4 Explanation

- **Mode management:** Two modes (`draw` and `select`) toggle what mouse actions do.
- **Drawing:** In draw mode, users click and drag to create new rectangles, with a preview shown during drag.
- **Selecting and dragging:** In select mode, clicking a rectangle highlights it; dragging

moves it smoothly.
- **Deleting:** The keyboard's Delete or Backspace key removes the selected shape.
- **Rendering:** The `draw()` function clears and redraws all shapes each frame, highlighting the selected one with an orange outline.
- **Modularity:** Event listeners are cleanly organized for different interaction states.

9.4.5 Ideas for Further Improvement

- Add color pickers to change the fill color of shapes.
- Support resizing of shapes with drag handles.
- Implement undo/redo stacks for editing history.
- Add more shape types (circles, polygons).
- Save/load drawings using JSON or images.

9.4.6 Summary

This practical example demonstrates how to build a versatile, interactive canvas app that supports:

- Drawing new shapes,
- Selecting and moving existing shapes,
- Deleting shapes using keyboard input,
- Clean state management for intuitive user experience.

By combining drawing, event handling, and stateful shape management, you're well on your way to creating powerful graphical applications!

Chapter 10.

Working with Canvas State and Compositing

1. Global Alpha and Transparency
2. Compositing Operations (`globalCompositeOperation`)
3. Clipping Regions and Masks
4. Practical Example: Layered Image Effects

10 Working with Canvas State and Compositing

10.1 Global Alpha and Transparency

The `globalAlpha` property is one of the simplest yet most powerful ways to control transparency in canvas drawings. It sets a global opacity level that applies to *all* drawing operations performed on the canvas context, allowing you to create layered effects and smooth blending.

10.1.1 What Is `globalAlpha`?

- **Definition:** `globalAlpha` is a property of the canvas 2D rendering context that takes a value between `0.0` (completely transparent) and `1.0` (fully opaque).
- **Effect:** It multiplies the alpha (opacity) of all subsequent drawing operations by this value.
- **Scope:** This applies to all shapes, lines, text, and images drawn *after* setting `globalAlpha`, until it is changed again.

```
ctx.globalAlpha = 0.5;  // All future drawing will be 50% transparent
```

10.1.2 How `globalAlpha` Works with Colors and Images

`globalAlpha` works in combination with the colors and images you draw:

- If your `fillStyle` or `strokeStyle` already has transparency (e.g., `rgba(255, 0, 0, 0.7)`), then the resulting opacity is the product of both the style's alpha and `globalAlpha`.
- When drawing images, `globalAlpha` adjusts the overall opacity of the image rendering, allowing you to create soft overlays or ghosted effects.

10.1.3 Example: Drawing Semi-Transparent Overlapping Rectangles

```
<canvas id="canvas" width="300" height="150" style="border:1px solid #ccc;"></canvas>
<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');

  // Draw first solid blue rectangle
  ctx.fillStyle = 'blue';
  ctx.globalAlpha = 1.0; // fully opaque
```

```
  ctx.fillRect(20, 20, 100, 100);

  // Draw second semi-transparent red rectangle overlapping the first
  ctx.fillStyle = 'red';
  ctx.globalAlpha = 0.5; // 50% opacity
  ctx.fillRect(60, 60, 100, 100);

  // Reset globalAlpha to fully opaque for any future drawing
  ctx.globalAlpha = 1.0;
</script>
```

In this example:

- The blue rectangle is fully opaque.
- The red rectangle is drawn on top at 50% opacity.
- The overlapping area visually blends red and blue because of the transparency.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

  // Draw first solid blue rectangle
  ctx.fillStyle = 'blue';
  ctx.globalAlpha = 1.0; // fully opaque
  ctx.fillRect(20, 20, 100, 100);

  // Draw second semi-transparent red rectangle overlapping the first
  ctx.fillStyle = 'red';
  ctx.globalAlpha = 0.5; // 50% opacity
  ctx.fillRect(60, 60, 100, 100);

  // Reset globalAlpha to fully opaque for any future drawing
  ctx.globalAlpha = 1.0;
</script>

</body>
</html>
```

10.1.4 Transparency and Layering

Because `globalAlpha` affects the alpha channel of every pixel drawn, it influences how layers visually combine:

- When you draw multiple semi-transparent shapes on top of each other, the colors blend based on their alpha values.
- Lower `globalAlpha` means the new drawing will allow more of the underlying canvas content to show through.
- This makes `globalAlpha` useful for effects like shadows, glows, or fade-ins/fade-outs.

10.1.5 Combining `globalAlpha` with Images

You can use `globalAlpha` when drawing images to create transparent overlays or fade effects:

```
const img = new Image();
img.src = 'example.png';

img.onload = () => {
  ctx.globalAlpha = 0.6; // 60% opacity
  ctx.drawImage(img, 10, 10);

  ctx.globalAlpha = 1.0; // Reset alpha for other drawings
};
```

10.1.6 Summary

- `globalAlpha` controls the transparency for **all** subsequent canvas drawing operations.
- It multiplies with any existing color or image transparency.
- Useful for layering semi-transparent shapes, images, or effects.
- Always remember to reset it to `1.0` after your transparent drawing to avoid unintended transparency later.

Experiment by combining different values of `globalAlpha` and overlapping drawings to see how blending changes the visual output—this is a fundamental technique in canvas graphics!

10.2 Compositing Operations (`globalCompositeOperation`)

When drawing on a canvas, new shapes and images don't simply cover the existing content—they blend with it according to specific rules. The **`globalCompositeOperation`** property controls *how* new pixels combine with the pixels already on the canvas, enabling a wide range of visual effects from simple layering to complex masking.

10.2.1 What Is `globalCompositeOperation`?

- It is a property of the canvas 2D context.
- Defines the **compositing/blending mode** used when drawing new shapes or images over existing content.
- Controls how source pixels (new drawing) and destination pixels (existing canvas content) are combined.

```
ctx.globalCompositeOperation = 'source-over'; // Default mode
```

10.2.2 Common Composite Modes and Their Effects

Here are some key compositing modes you'll use frequently:

Mode	Description	Visual Effect	Use Cases
source-over	Default. New content drawn *over* existing content.	Normal drawing, top layer covers underlying.	Basic drawing
destination-over	New content is drawn *behind* existing content.	New shapes appear beneath existing ones.	Background fills, layering
xor	Draws pixels where either source or destination exists, but not both.	Creates "cut-out" or exclusion effect.	Erasing, masking
lighter	Adds the pixel values of source and destination, producing a brightened effect.	Colors become brighter where they overlap.	Glow, light effects
multiply	Multiplies source and destination pixel values, darkening the overlapping areas.	Produces shadow or shading effects.	Shadows, shading, dark overlays

10.2.3 Visualizing Composite Modes with Examples

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

// Draw blue rectangle
ctx.fillStyle = 'blue';
ctx.fillRect(20, 20, 100, 100);

// Change composite mode
```

```
ctx.globalCompositeOperation = 'lighter';  // Try changing to 'destination-over', 'xor', etc.

// Draw semi-transparent red rectangle overlapping the blue
ctx.fillStyle = 'rgba(255, 0, 0, 0.6)';
ctx.fillRect(60, 60, 100, 100);

// Reset to default for future drawing
ctx.globalCompositeOperation = 'source-over';
```

- **Try changing** the `globalCompositeOperation` value to see how the red rectangle blends differently with the blue.
- Notice how some modes draw the new shape *behind* the existing one (`destination-over`), or create special effects like glowing (`lighter`) or cutting out shapes (`xor`).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

// Draw blue rectangle
ctx.fillStyle = 'blue';
ctx.fillRect(20, 20, 100, 100);

// Change composite mode
ctx.globalCompositeOperation = 'lighter';  // Try changing to 'destination-over', 'xor', etc.

// Draw semi-transparent red rectangle overlapping the blue
ctx.fillStyle = 'rgba(255, 0, 0, 0.6)';
ctx.fillRect(60, 60, 100, 100);

// Reset to default for future drawing
ctx.globalCompositeOperation = 'source-over';
</script>

</body>
</html>
```

10.2.4 Use Cases for Compositing Modes

Erasing and Masking

By combining composite modes like `destination-out` or `xor`, you can erase parts of drawings or create masks that reveal/hide certain canvas areas without needing separate layers.

Glow and Light Effects

Modes like `lighter` brighten overlapping areas, simulating light or glow effects around shapes, useful in games and dynamic visualizations.

Complex Layering

Modes like `multiply` let you create shadows and shading by darkening pixels where layers overlap, enriching your graphics with depth.

10.2.5 Experiment and Discover

Try layering multiple shapes, images, or text with different `globalCompositeOperation` settings:

- Draw a pattern with `source-over` and add shadows with `multiply`.
- Use `destination-over` to place backgrounds without covering your foreground shapes.
- Combine `xor` with transparent shapes for interesting cut-out effects.

10.2.6 Summary

- **`globalCompositeOperation`** controls how new drawings blend with existing canvas pixels.
- Different modes offer powerful blending effects beyond simple layering.
- Use it to create erasing, masking, lighting, and shading effects.
- Experiment with modes to find creative ways to enrich your canvas graphics.

Mastering compositing is essential for advanced canvas effects and interactive graphics!

10.3 Clipping Regions and Masks

Clipping is a powerful canvas feature that restricts drawing operations to a specified region or shape, much like putting a stencil over your canvas. This allows you to control *where* rendering happens, enabling creative effects like custom-shaped image displays, text cutouts, or complex masking.

10.3.1 What Is a Clipping Region?

- A **clipping region** limits all subsequent drawing operations to the area inside the defined path.
- Anything drawn *outside* the clipping path is **not rendered** on the canvas.
- This is especially useful for masking effects or focusing drawing on irregular shapes.

10.3.2 How to Define a Clipping Path

1. Begin a path as usual with `beginPath()`.
2. Use drawing commands to create the clipping shape (e.g., circle, star, text).
3. Call `ctx.clip()` to set the clipping region to the current path.
4. Any drawing after `clip()` is restricted to inside this region.

10.3.3 Important: Use `save()` and `restore()`

Clipping affects *all* subsequent drawing operations on the context. To avoid clipping everything else unintentionally:

- Use `ctx.save()` before defining the clipping path.
- Use `ctx.restore()` after you finish the clipped drawing section.

This way, the clipping region is **isolated** and undone after `restore()`, allowing normal drawing outside the clipped area.

10.3.4 Example: Clipping a Circle and Drawing an Image Inside

```
<canvas id="canvas" width="300" height="200" style="border:1px solid #ccc;"></canvas>
<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');
  const img = new Image();
  img.src = 'https://picsum.photos/300/200';

  img.onload = () => {
    ctx.save();  // Save current state

    // Create circular clipping path
    ctx.beginPath();
    ctx.arc(150, 100, 80, 0, Math.PI * 2);
    ctx.clip();
```

```
    // Draw the image - only the circular part will show
    ctx.drawImage(img, 0, 0, 300, 200);

    ctx.restore();  // Restore state, removing clipping
  };
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

  const img = new Image();
  img.src = 'https://picsum.photos/300/200';

  img.onload = () => {
    ctx.save();  // Save current state

    // Create circular clipping path
    ctx.beginPath();
    ctx.arc(150, 100, 80, 0, Math.PI * 2);
    ctx.clip();

    // Draw the image - only the circular part will show
    ctx.drawImage(img, 0, 0, 300, 200);

    ctx.restore();  // Restore state, removing clipping
  };
</script>

</body>
</html>
```

10.3.5 Clipping with Text and Complex Shapes

You can also clip to text or custom shapes:

```
ctx.save();

ctx.font = '48px serif';
ctx.textAlign = 'center';
ctx.textBaseline = 'middle';

// Create text path and clip to it
```

```
ctx.beginPath();
ctx.fillText('Hello', 150, 100);
ctx.clip();

// Draw gradient only inside text shape
const gradient = ctx.createLinearGradient(0, 0, 300, 0);
gradient.addColorStop(0, 'blue');
gradient.addColorStop(1, 'red');
ctx.fillStyle = gradient;
ctx.fillRect(0, 0, 300, 200);

ctx.restore();
```

Here, the colorful gradient fills only the area shaped like the text “Hello”.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

ctx.save();

ctx.font = '48px serif';
ctx.textAlign = 'center';
ctx.textBaseline = 'middle';

// Create text path and clip to it
ctx.beginPath();
ctx.fillText('Hello', 150, 100);
ctx.clip();

// Draw gradient only inside text shape
const gradient = ctx.createLinearGradient(0, 0, 300, 0);
gradient.addColorStop(0, 'blue');
gradient.addColorStop(1, 'red');
ctx.fillStyle = gradient;
ctx.fillRect(0, 0, 300, 200);

ctx.restore();
</script>

</body>
</html>
```

10.3.6 Masks: Reusing Clipping for Selective Reveal

While `clip()` defines a clipping path directly, masks can be built by:

- Drawing the mask shape to an offscreen canvas.
- Using compositing modes (`globalCompositeOperation`) combined with clipping to selectively reveal or hide parts of your drawing.
- This enables reusable and complex masking effects (e.g., soft edges, multiple overlapping masks).

10.3.7 Summary

- The `clip()` method restricts all drawing to a defined path or region.
- Use `beginPath()` to create the clipping shape (circle, star, text, polygon).
- Always use `save()` and `restore()` to isolate clipping and avoid unintended restrictions.
- Clipping enables creative masking and focus effects on canvas drawings.
- Combine clipping with compositing for even more advanced masking capabilities.

Experiment by clipping to different shapes and layering drawings to discover how you can selectively reveal your canvas artwork in exciting ways!

10.4 Practical Example: Layered Image Effects

In this section, we'll build an interactive canvas demo that stacks multiple images and shapes using different blending modes (`globalCompositeOperation`) and transparency (`globalAlpha`). We'll also add a toggle for clipping, turning our canvas into a simple but powerful **layered image effects tool** — perfect for experimenting with creative visual compositions.

10.4.1 What You'll Build

- Several overlapping shapes and images rendered on different layers.
- Controls to dynamically adjust:
  - **Opacity** of the entire canvas content (`globalAlpha`).
  - **Blend mode** (`globalCompositeOperation`) for how new drawing layers mix with existing pixels.
  - Enable or disable a **clipping region** that restricts rendering to a shape.
- Real-time updates as you tweak the settings.

This example highlights how compositing and transparency combine to create artistic filters, overlays, and masking effects.

10.4.2 Complete Demo Code

```
<h2>Layered Image Effects with Canvas</h2>

<canvas id="canvas" width="500" height="400"></canvas>

<label for="alphaRange">Opacity (globalAlpha):</label>
<input type="range" id="alphaRange" min="0" max="1" step="0.01" value="1" />

<label for="blendModeSelect">Blend Mode (globalCompositeOperation):</label>
<select id="blendModeSelect">
  <option value="source-over" selected>source-over (default)</option>
  <option value="multiply">multiply</option>
  <option value="screen">screen</option>
  <option value="overlay">overlay</option>
  <option value="lighter">lighter</option>
  <option value="xor">xor</option>
  <option value="destination-over">destination-over</option>
</select>

<label><input type="checkbox" id="clipToggle" /> Enable Clipping Circle</label>
```

Javascript:

```
<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');
  const alphaRange = document.getElementById('alphaRange');
  const blendModeSelect = document.getElementById('blendModeSelect');
  const clipToggle = document.getElementById('clipToggle');

  // Load images
  const img1 = new Image();
  const img2 = new Image();
  img1.src = 'https://picsum.photos/id/1015/500/400';  // Nature image
  img2.src = 'https://picsum.photos/id/1018/500/400';  // Another nature image

  // Wait for images to load before drawing
  let imagesLoaded = 0;
  [img1, img2].forEach(img => {
    img.onload = () => {
      imagesLoaded++;
      if (imagesLoaded === 2) drawScene();
    };
  });

  function drawScene() {
    // Clear canvas first
    ctx.clearRect(0, 0, canvas.width, canvas.height);
```

```
      // Apply global opacity
      ctx.globalAlpha = parseFloat(alphaRange.value);

      // Optionally apply clipping circle
      if (clipToggle.checked) {
        ctx.save();
        ctx.beginPath();
        ctx.arc(canvas.width / 2, canvas.height / 2, 150, 0, Math.PI * 2);
        ctx.clip();
      }

      // Draw first image fully opaque
      ctx.globalCompositeOperation = 'source-over';
      ctx.drawImage(img1, 0, 0, canvas.width, canvas.height);

      // Draw a semi-transparent colored rectangle on top
      ctx.globalAlpha = 0.6 * parseFloat(alphaRange.value);
      ctx.fillStyle = 'rgba(255, 140, 0, 0.7)'; // Orange overlay
      ctx.fillRect(50, 50, 400, 300);

      // Draw second image with selected blend mode
      ctx.globalAlpha = 0.8 * parseFloat(alphaRange.value);
      ctx.globalCompositeOperation = blendModeSelect.value;
      ctx.drawImage(img2, 0, 0, canvas.width, canvas.height);

      // Draw a decorative circle with a gradient
      ctx.globalCompositeOperation = 'source-over';
      ctx.globalAlpha = 0.9 * parseFloat(alphaRange.value);
      const grad = ctx.createRadialGradient(250, 200, 30, 250, 200, 100);
      grad.addColorStop(0, 'rgba(0, 255, 255, 0.8)');
      grad.addColorStop(1, 'transparent');
      ctx.fillStyle = grad;
      ctx.beginPath();
      ctx.arc(250, 200, 100, 0, Math.PI * 2);
      ctx.fill();

      if (clipToggle.checked) {
        ctx.restore();  // Restore after clipping
      }
    }

    // Update scene on control changes
    alphaRange.addEventListener('input', drawScene);
    blendModeSelect.addEventListener('change', drawScene);
    clipToggle.addEventListener('change', drawScene);
  </script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Layered Image Effects Demo</title>
  <style>
    body { font-family: Arial, sans-serif; margin: 20px; }
    canvas { border: 1px solid #ccc; display: block; margin-bottom: 10px; }
    label, select, input { margin-right: 10px; }
  </style>
```

```
</head>
<body>

  <h2>Layered Image Effects with Canvas</h2>

  <canvas id="canvas" width="500" height="400"></canvas>

  <label for="alphaRange">Opacity (globalAlpha):</label>
  <input type="range" id="alphaRange" min="0" max="1" step="0.01" value="1" />

  <label for="blendModeSelect">Blend Mode (globalCompositeOperation):</label>
  <select id="blendModeSelect">
    <option value="source-over" selected>source-over (default)</option>
    <option value="multiply">multiply</option>
    <option value="screen">screen</option>
    <option value="overlay">overlay</option>
    <option value="lighter">lighter</option>
    <option value="xor">xor</option>
    <option value="destination-over">destination-over</option>
  </select>

  <label><input type="checkbox" id="clipToggle" /> Enable Clipping Circle</label>

  <script>
    const canvas = document.getElementById('canvas');
    const ctx = canvas.getContext('2d');
    const alphaRange = document.getElementById('alphaRange');
    const blendModeSelect = document.getElementById('blendModeSelect');
    const clipToggle = document.getElementById('clipToggle');

    // Load images
    const img1 = new Image();
    const img2 = new Image();
    img1.src = 'https://picsum.photos/id/1015/500/400';  // Nature image
    img2.src = 'https://picsum.photos/id/1018/500/400';  // Another nature image

    // Wait for images to load before drawing
    let imagesLoaded = 0;
    [img1, img2].forEach(img => {
      img.onload = () => {
        imagesLoaded++;
        if (imagesLoaded === 2) drawScene();
      };
    });

    function drawScene() {
      // Clear canvas first
      ctx.clearRect(0, 0, canvas.width, canvas.height);

      // Apply global opacity
      ctx.globalAlpha = parseFloat(alphaRange.value);

      // Optionally apply clipping circle
      if (clipToggle.checked) {
        ctx.save();
        ctx.beginPath();
        ctx.arc(canvas.width / 2, canvas.height / 2, 150, 0, Math.PI * 2);
        ctx.clip();
```

```
      }

      // Draw first image fully opaque
      ctx.globalCompositeOperation = 'source-over';
      ctx.drawImage(img1, 0, 0, canvas.width, canvas.height);

      // Draw a semi-transparent colored rectangle on top
      ctx.globalAlpha = 0.6 * parseFloat(alphaRange.value);
      ctx.fillStyle = 'rgba(255, 140, 0, 0.7)'; // Orange overlay
      ctx.fillRect(50, 50, 400, 300);

      // Draw second image with selected blend mode
      ctx.globalAlpha = 0.8 * parseFloat(alphaRange.value);
      ctx.globalCompositeOperation = blendModeSelect.value;
      ctx.drawImage(img2, 0, 0, canvas.width, canvas.height);

      // Draw a decorative circle with a gradient
      ctx.globalCompositeOperation = 'source-over';
      ctx.globalAlpha = 0.9 * parseFloat(alphaRange.value);
      const grad = ctx.createRadialGradient(250, 200, 30, 250, 200, 100);
      grad.addColorStop(0, 'rgba(0, 255, 255, 0.8)');
      grad.addColorStop(1, 'transparent');
      ctx.fillStyle = grad;
      ctx.beginPath();
      ctx.arc(250, 200, 100, 0, Math.PI * 2);
      ctx.fill();

      if (clipToggle.checked) {
        ctx.restore();  // Restore after clipping
      }
    }

    // Update scene on control changes
    alphaRange.addEventListener('input', drawScene);
    blendModeSelect.addEventListener('change', drawScene);
    clipToggle.addEventListener('change', drawScene);
  </script>

</body>
</html>
```

10.4.3 How It Works

- **Layering**: The two images and shapes are drawn in sequence. The second image uses the blend mode selected by the user to combine visually with what's already on the canvas.
- **Opacity Control**: The slider adjusts the global alpha affecting all drawing operations, allowing for transparent layering effects.
- **Clipping**: The checkbox toggles a circular clipping region, restricting rendering to a round area centered in the canvas.
- **Blend Modes**: Options like `multiply`, `screen`, and `overlay` create diverse visual

results by changing how pixels from new drawings blend with the existing pixels on the canvas.

10.4.4 Experiment Ideas

- Try other blend modes like `difference`, `color-dodge`, or `darken`.
- Change the clipping path to a star or text shape for different masking effects.
- Add sliders to control individual layer opacities for more nuanced control.
- Replace images with patterns or gradients to explore more creative compositions.

10.4.5 Summary

This practical example combines transparency, compositing, and clipping to create a flexible and creative layered image effect tool. By interacting with the controls, you can immediately see how canvas drawing states influence the final rendered output — a fundamental skill in advanced graphics programming with HTML5 Canvas.

Chapter 11.

Working with Pixel Data and Filters

1. Manipulating Pixels Directly
2. Creating Custom Filters and Effects
3. Using Offscreen Canvases for Performance
4. Practical Example: Grayscale and Invert Color Filters

11 Working with Pixel Data and Filters

11.1 Manipulating Pixels Directly

One of the most powerful features of the HTML5 Canvas API is the ability to access and manipulate the raw pixel data of an image or canvas area. This gives you full control to create custom visual effects, analyze image content, or build complex filters — all by working directly with the RGBA color values of each pixel.

11.1.1 Accessing Raw Pixel Data: `getImageData()`

The method `getImageData(x, y, width, height)` extracts pixel data from a specified rectangular area on the canvas. It returns an `ImageData` object containing a `data` property — a one-dimensional `Uint8ClampedArray` holding the pixel colors.

11.1.2 Pixel Data Structure

Each pixel consists of **four consecutive bytes** in this array representing:

- **R** (Red) — value from 0 to 255
- **G** (Green) — value from 0 to 255
- **B** (Blue) — value from 0 to 255
- **A** (Alpha/opacity) — value from 0 (transparent) to 255 (opaque)

The array length is therefore `width * height * 4`, and the pixel at `(x, y)` can be accessed using the index:

```
const index = (y * width + x) * 4;
```

11.1.3 Modifying Pixels: `putImageData()`

After processing or modifying the pixel array, you can write the changes back to the canvas using `putImageData(imageData, dx, dy)`, which paints the pixel data onto the canvas at position `(dx, dy)`.

11.1.4 Example: Highlighting a Region by Increasing Red Channel

Below is an example that loads an image on the canvas, grabs its pixel data, then increases the red component for a rectangular area to create a red highlight effect:

```
<canvas id="canvas" width="300" height="200"></canvas>
<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');

  const img = new Image();
  img.src = 'https://picsum.photos/300/200';
  img.onload = () => {
    ctx.drawImage(img, 0, 0);

    // Get pixel data of entire canvas
    const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
    const data = imageData.data;

    // Highlight a rectangle area (50,50 to 150,150) by boosting red channel
    for (let y = 50; y < 150; y++) {
      for (let x = 50; x < 150; x++) {
        const idx = (y * canvas.width + x) * 4;
        data[idx] = Math.min(255, data[idx] + 100); // Increase red channel safely
      }
    }

    // Put the modified pixels back on the canvas
    ctx.putImageData(imageData, 0, 0);
  };
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<body>

<canvas id="myCanvas" width="400" height="200" style="border:1px solid #ccc;"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

  const img = new Image();
  img.src = 'https://picsum.photos/300/200';
  img.onload = () => {
    ctx.drawImage(img, 0, 0);

    // Get pixel data of entire canvas
    const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
    const data = imageData.data;

    // Highlight a rectangle area (50,50 to 150,150) by boosting red channel
    for (let y = 50; y < 150; y++) {
```

```
      for (let x = 50; x < 150; x++) {
        const idx = (y * canvas.width + x) * 4;
        data[idx] = Math.min(255, data[idx] + 100); // Increase red channel safely
      }
    }

    // Put the modified pixels back on the canvas
    ctx.putImageData(imageData, 0, 0);
  };
</script>

</body>
</html>
```

11.1.5 More Advanced Uses: Edge Detection and Visual Effects

By reading pixel values and comparing neighbors, you can detect edges or transitions, e.g., applying simple Sobel or Laplacian filters. Similarly, you can invert colors, adjust brightness, or apply grayscale effects by altering R, G, B values accordingly.

11.1.6 Performance Considerations

- **Pixel Manipulation Is Costly**: Accessing and writing pixel data causes CPU overhead, especially for large canvases or repeated animations.
- **Minimize Calls**: Try to batch your pixel processing and avoid unnecessary repeated `getImageData()` or `putImageData()` calls.
- **Use Offscreen Canvases**: For heavy processing, consider using offscreen canvases or Web Workers to prevent UI blocking.
- **Clamp Values**: The `Uint8ClampedArray` clamps out-of-range values (less than 0 or greater than 255), which simplifies color calculations.

11.1.7 Summary

Working directly with pixel data unlocks infinite possibilities for custom image effects and analysis on the canvas. By understanding the RGBA data structure and efficiently looping through pixel arrays, you can build sophisticated filters and visuals, with awareness of the performance trade-offs involved.

11.2 Creating Custom Filters and Effects

After learning how to access and manipulate raw pixel data, the next step is to create custom filters and effects. These filters transform the image in creative ways, such as adjusting brightness, contrast, or applying blur and sharpening. Many effects rely on **pixel math** and **convolution** techniques to process image data.

11.2.1 Brightness and Contrast Adjustments

Brightness and contrast filters directly modify the color values of each pixel.

- **Brightness**: Increase or decrease each color channel by a constant.
- **Contrast**: Scale pixel color values around the midpoint (128) to increase or decrease contrast.

Here's a simple example function that applies brightness and contrast:

```
function applyBrightnessContrast(imageData, brightness = 0, contrast = 0) {
  const data = imageData.data;
  const factor = (259 * (contrast + 255)) / (255 * (259 - contrast));

  for (let i = 0; i < data.length; i += 4) {
    // Apply brightness (add) and contrast (scale)
    data[i] = truncate(factor * (data[i] - 128) + 128 + brightness);     // Red
    data[i + 1] = truncate(factor * (data[i + 1] - 128) + 128 + brightness); // Green
    data[i + 2] = truncate(factor * (data[i + 2] - 128) + 128 + brightness); // Blue
  }

  return imageData;
}

function truncate(value) {
  return Math.min(255, Math.max(0, value));
}
```

11.2.2 Convolution Filters: Blur, Sharpen, Edge Detection

Many image effects are built using **convolution**, a process where each pixel is recalculated based on a weighted sum of its neighbors. The weights are defined by a **kernel matrix**.

11.2.3 How Convolution Works

- For each pixel, multiply neighboring pixel values by the corresponding kernel values.

- Sum these products to get the new pixel value.
- Apply this separately for R, G, and B channels.

11.2.4 Common Kernels

Effect	Kernel Example (3x3)
Blur	$1/9 \times$ [[1,1,1], [1,1,1], [1,1,1]]
Sharpen	[[0,-1,0], [-1,5,-1], [0,-1,0]]
Edge Detect	[[-1,-1,-1], [-1,8,-1], [-1,-1,-1]]

11.2.5 Reusable Convolution Function

Here's a generic function to apply a convolution kernel to an image:

```
function applyConvolution(imageData, kernel, divisor = 1, offset = 0) {
  const { width, height, data } = imageData;
  const output = new Uint8ClampedArray(data.length);
  const side = Math.sqrt(kernel.length);
  const half = Math.floor(side / 2);

  for (let y = 0; y < height; y++) {
    for (let x = 0; x < width; x++) {
      let r = 0, g = 0, b = 0;

      for (let ky = 0; ky < side; ky++) {
        for (let kx = 0; kx < side; kx++) {
          const px = x + kx - half;
          const py = y + ky - half;

          if (px >= 0 && px < width && py >= 0 && py < height) {
            const idx = (py * width + px) * 4;
            const weight = kernel[ky * side + kx];
            r += data[idx] * weight;
            g += data[idx + 1] * weight;
            b += data[idx + 2] * weight;
          }
        }
      }

      const i = (y * width + x) * 4;
      output[i] = truncate(r / divisor + offset);
      output[i + 1] = truncate(g / divisor + offset);
      output[i + 2] = truncate(b / divisor + offset);
      output[i + 3] = data[i + 3]; // copy alpha
    }
  }
```

```
  // Copy output back to imageData.data
  for (let i = 0; i < output.length; i++) {
    data[i] = output[i];
  }

  return imageData;
}
```

11.2.6 Example: Applying a Blur Filter

```
const blurKernel = [
  1, 1, 1,
  1, 1, 1,
  1, 1, 1
];
const divisor = 9; // Sum of kernel weights

// Use like this:
const blurredImageData = applyConvolution(imageData, blurKernel, divisor);
ctx.putImageData(blurredImageData, 0, 0);
```

11.2.7 Building a Modular Filter Pipeline

You can chain multiple filters by applying one after another:

```
function applyFilters(imageData, filters) {
  return filters.reduce((imgData, filter) => filter(imgData), imageData);
}

// Example usage
const filters = [
  imgData => applyBrightnessContrast(imgData, 20, 40),
  imgData => applyConvolution(imgData, blurKernel, 9)
];
const result = applyFilters(imageData, filters);
ctx.putImageData(result, 0, 0);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Brightness, Contrast & Convolution Filters</title>
<style>
  body { font-family: sans-serif; padding: 20px; background: #fafafa; }
  canvas { border: 1px solid #ccc; margin-top: 10px; max-width: 100%; }
  button, input[type="range"] { margin: 5px; }
```

```
  label { display: block; margin-top: 10px; }
</style>
</head>
<body>

<h2>Image Filters: Brightness, Contrast, Blur, Sharpen, Edge Detect</h2>

<input type="file" id="upload" accept="image/*" />
<br/>

<label>
  Brightness: <input type="range" id="brightness" min="-100" max="100" value="0" />
</label>
<label>
  Contrast: <input type="range" id="contrast" min="-100" max="100" value="0" />
</label>

<button id="blurBtn">Blur</button>
<button id="sharpenBtn">Sharpen</button>
<button id="edgeBtn">Edge Detect</button>
<button id="resetBtn">Reset</button>

<br/>

<canvas id="canvas" width="600" height="400"></canvas>

<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
const upload = document.getElementById('upload');
const brightnessSlider = document.getElementById('brightness');
const contrastSlider = document.getElementById('contrast');
const blurBtn = document.getElementById('blurBtn');
const sharpenBtn = document.getElementById('sharpenBtn');
const edgeBtn = document.getElementById('edgeBtn');
const resetBtn = document.getElementById('resetBtn');

let originalImageData = null;
let currentImageData = null;

// Helpers
function truncate(value) {
  return Math.min(255, Math.max(0, value));
}

function applyBrightnessContrast(imageData, brightness = 0, contrast = 0) {
  const data = imageData.data;
  const factor = (259 * (contrast + 255)) / (255 * (259 - contrast));

  for (let i = 0; i < data.length; i += 4) {
    data[i]     = truncate(factor * (data[i] - 128) + 128 + brightness);     // R
    data[i + 1] = truncate(factor * (data[i + 1] - 128) + 128 + brightness); // G
    data[i + 2] = truncate(factor * (data[i + 2] - 128) + 128 + brightness); // B
  }
  return imageData;
}

function applyConvolution(imageData, kernel, divisor = 1, offset = 0) {
```

```
  const { width, height, data } = imageData;
  const output = new Uint8ClampedArray(data.length);
  const side = Math.sqrt(kernel.length);
  const half = Math.floor(side / 2);

  for (let y = 0; y < height; y++) {
    for (let x = 0; x < width; x++) {
      let r = 0, g = 0, b = 0;

      for (let ky = 0; ky < side; ky++) {
        for (let kx = 0; kx < side; kx++) {
          const px = x + kx - half;
          const py = y + ky - half;

          if (px >= 0 && px < width && py >= 0 && py < height) {
            const idx = (py * width + px) * 4;
            const weight = kernel[ky * side + kx];
            r += data[idx] * weight;
            g += data[idx + 1] * weight;
            b += data[idx + 2] * weight;
          }
        }
      }

      const i = (y * width + x) * 4;
      output[i] = truncate(r / divisor + offset);
      output[i + 1] = truncate(g / divisor + offset);
      output[i + 2] = truncate(b / divisor + offset);
      output[i + 3] = data[i + 3]; // alpha
    }
  }

  for (let i = 0; i < output.length; i++) {
    data[i] = output[i];
  }

  return imageData;
}

// Kernels
const blurKernel = [
  1, 1, 1,
  1, 1, 1,
  1, 1, 1
];

const sharpenKernel = [
  0, -1, 0,
  -1, 5, -1,
  0, -1, 0
];

const edgeKernel = [
  -1, -1, -1,
  -1, 8, -1,
  -1, -1, -1
];
```

```
// Render image data on canvas
function render(imageData) {
  ctx.putImageData(imageData, 0, 0);
  currentImageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
}

// Upload image handler
upload.addEventListener('change', e => {
  const file = e.target.files[0];
  if (!file) return;

  const img = new Image();
  img.onload = () => {
    // Resize canvas to image size
    canvas.width = img.width;
    canvas.height = img.height;

    ctx.drawImage(img, 0, 0);
    originalImageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
    currentImageData = ctx.getImageData(0, 0, canvas.width, canvas.height);

    // Reset sliders
    brightnessSlider.value = 0;
    contrastSlider.value = 0;
  };
  img.src = URL.createObjectURL(file);
});

// Update brightness and contrast live
function updateBrightnessContrast() {
  if (!originalImageData) return;
  const brightness = parseInt(brightnessSlider.value, 10);
  const contrast = parseInt(contrastSlider.value, 10);

  // Clone originalImageData to avoid stacking effects
  let imageCopy = new ImageData(
    new Uint8ClampedArray(originalImageData.data),
    originalImageData.width,
    originalImageData.height
  );

  applyBrightnessContrast(imageCopy, brightness, contrast);
  render(imageCopy);
}

brightnessSlider.addEventListener('input', updateBrightnessContrast);
contrastSlider.addEventListener('input', updateBrightnessContrast);

// Convolution button handlers
blurBtn.addEventListener('click', () => {
  if (!currentImageData) return;
  applyConvolution(currentImageData, blurKernel, 9);
  render(currentImageData);
});
sharpenBtn.addEventListener('click', () => {
  if (!currentImageData) return;
  applyConvolution(currentImageData, sharpenKernel, 1);
  render(currentImageData);
```

```
});
edgeBtn.addEventListener('click', () => {
  if (!currentImageData) return;
  applyConvolution(currentImageData, edgeKernel, 1);
  render(currentImageData);
});

// Reset button
resetBtn.addEventListener('click', () => {
  if (!originalImageData) return;
  brightnessSlider.value = 0;
  contrastSlider.value = 0;
  render(originalImageData);
});
</script>

</body>
</html>
```

11.2.8 Summary and Experimentation

Creating custom filters with pixel math and convolution empowers you to build endless effects — from subtle adjustments like brightness to artistic styles like sharpening or edge detection.

- Experiment with different kernel matrices to see how they affect the image.
- Adjust filter parameters for dynamic results.
- Use modular pipelines to combine multiple effects smoothly.

In the next sections, we'll explore how to optimize these filters and build interactive tools around them.

11.3 Using Offscreen Canvases for Performance

When working with pixel data and complex image filters, performance can become a concern—especially when manipulating large images or running animations. The **`OffscreenCanvas`** API offers a way to improve rendering efficiency by allowing canvas drawing and processing to happen off the main UI thread, which keeps the interface smooth and responsive.

11.3.1 What Is `OffscreenCanvas`?

`OffscreenCanvas` is a canvas element that exists **entirely off the visible document**, meaning it is not part of the DOM. It can be used to:

- Perform drawing operations without blocking the main thread.
- Work with Web Workers for multi-threaded rendering.
- Cache expensive drawing or filtering results.
- Prepare images or graphics in the background before displaying them on the visible canvas.

This contrasts with the traditional method of creating a hidden canvas via:

```
const hiddenCanvas = document.createElement('canvas');
```

which still runs on the main thread but simply isn't added to the page.

11.3.2 Creating and Using an Offscreen Canvas

```
// Create an offscreen canvas of width 300, height 150
const offscreen = new OffscreenCanvas(300, 150);
const ctx = offscreen.getContext('2d');

// Draw or manipulate image data offscreen
ctx.fillStyle = 'blue';
ctx.fillRect(0, 0, 300, 150);

// Later, transfer or draw this offscreen canvas onto a visible canvas
const visibleCanvas = document.getElementById('myCanvas');
const visibleCtx = visibleCanvas.getContext('2d');

// Draw offscreen canvas onto visible canvas
visibleCtx.drawImage(offscreen, 0, 0);
```

11.3.3 Using OffscreenCanvas with Image Processing

You can perform filtering or pixel manipulation off the main thread:

```
// Example: applying filter offscreen and then rendering

function applyFilterOffscreen(imageBitmap) {
  const offscreen = new OffscreenCanvas(imageBitmap.width, imageBitmap.height);
  const ctx = offscreen.getContext('2d');
  ctx.drawImage(imageBitmap, 0, 0);
```

```
  let imageData = ctx.getImageData(0, 0, offscreen.width, offscreen.height);

  // Modify pixel data here (e.g., grayscale or invert)
  // ... pixel manipulation code ...

  ctx.putImageData(imageData, 0, 0);

  return offscreen.transferToImageBitmap(); // Efficiently transfers for rendering
}
```

This approach enables you to keep your UI responsive, especially during heavy image processing tasks.

11.3.4 OffscreenCanvas vs Hidden DOM Canvas

Feature	OffscreenCanvas	Hidden DOM Canvas
Runs on Main Thread	No, can run on worker threads	Yes
Visible in DOM	No	No (if not appended)
Performance Benefit	Higher, multi-threading	Limited to main thread
API Availability	Modern browsers only	Universal

11.3.5 Browser Support and Fallbacks

- **Modern browsers** like Chrome, Firefox, Edge support `OffscreenCanvas`, but **Safari support is limited or missing**.
- When `OffscreenCanvas` is not available, fallback to creating a hidden canvas in the DOM:

```
function createCanvas(width, height) {
  if (typeof OffscreenCanvas !== 'undefined') {
    return new OffscreenCanvas(width, height);
  } else {
    const canvas = document.createElement('canvas');
    canvas.width = width;
    canvas.height = height;
    return canvas;
  }
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>OffscreenCanvas Demo</title>
```

```
  <style>
    #myCanvas {
      border: 1px solid #ccc;
      margin-top: 10px;
      width: 300px;
      height: 150px;
    }
  </style>
</head>
<body>

<h3>OffscreenCanvas Drawing Demo</h3>
<canvas id="myCanvas" width="300" height="150"></canvas>

<script>
  // Utility to create OffscreenCanvas or fallback hidden DOM canvas
  function createCanvas(width, height) {
    if (typeof OffscreenCanvas !== 'undefined') {
      return new OffscreenCanvas(width, height);
    } else {
      const canvas = document.createElement('canvas');
      canvas.width = width;
      canvas.height = height;
      canvas.style.display = 'none';
      document.body.appendChild(canvas);
      return canvas;
    }
  }

  // Create offscreen canvas of size 300x150
  const offscreen = createCanvas(300, 150);
  const ctx = offscreen.getContext('2d');

  // Draw something on offscreen canvas
  ctx.fillStyle = 'blue';
  ctx.fillRect(0, 0, 300, 150);
  ctx.fillStyle = 'white';
  ctx.font = '24px sans-serif';
  ctx.fillText('Offscreen Canvas!', 50, 80);

  // Draw offscreen canvas content onto visible canvas
  const visibleCanvas = document.getElementById('myCanvas');
  const visibleCtx = visibleCanvas.getContext('2d');

  // If OffscreenCanvas supports transferToImageBitmap (modern), use it
  if (offscreen instanceof OffscreenCanvas && typeof offscreen.transferToImageBitmap === 'function') {
    const bitmap = offscreen.transferToImageBitmap();
    visibleCtx.drawImage(bitmap, 0, 0);
  } else {
    // Otherwise, just draw the fallback hidden canvas directly
    visibleCtx.drawImage(offscreen, 0, 0);
  }

  // Example function to demonstrate offscreen filtering and returning ImageBitmap
  async function applyFilterOffscreen(imageBitmap) {
    const offscreenFilterCanvas = createCanvas(imageBitmap.width, imageBitmap.height);
    const offscreenCtx = offscreenFilterCanvas.getContext('2d');
    offscreenCtx.drawImage(imageBitmap, 0, 0);
```

```
    let imageData = offscreenCtx.getImageData(0, 0, offscreenFilterCanvas.width, offscreenFilterCanvas.
    // Simple invert filter
    for (let i = 0; i < imageData.data.length; i += 4) {
      imageData.data[i] = 255 - imageData.data[i];       // R
      imageData.data[i+1] = 255 - imageData.data[i+1];   // G
      imageData.data[i+2] = 255 - imageData.data[i+2];   // B
    }

    offscreenCtx.putImageData(imageData, 0, 0);

    if (offscreenFilterCanvas instanceof OffscreenCanvas && typeof offscreenFilterCanvas.transferToImag
      return offscreenFilterCanvas.transferToImageBitmap();
    } else {
      // For fallback, create ImageBitmap from canvas
      return createImageBitmap(offscreenFilterCanvas);
    }
  }

  // (Optional) Usage example: invert colors after 2 seconds
  setTimeout(async () => {
    if (offscreen instanceof OffscreenCanvas && typeof offscreen.transferToImageBitmap === 'function')
      const bitmap = offscreen.transferToImageBitmap();
      const invertedBitmap = await applyFilterOffscreen(bitmap);
      visibleCtx.clearRect(0, 0, visibleCanvas.width, visibleCanvas.height);
      visibleCtx.drawImage(invertedBitmap, 0, 0);
    }
  }, 2000);

</script>

</body>
</html>
```

11.3.6 Summary

Using `OffscreenCanvas` is a powerful technique for:

- Improving performance by offloading expensive drawing or filtering tasks.
- Keeping the UI responsive during complex canvas operations.
- Preparing and caching visual content before rendering it on screen.

As browser support grows, this API will become an essential tool for advanced canvas programming and smooth user experiences.

Next, we will combine these techniques in a practical example that applies grayscale and invert filters efficiently.

11.4 Practical Example: Grayscale and Invert Color Filters

In this section, we'll build a simple but interactive image processor that allows you to switch between the original image, a grayscale version, and an inverted color version. This hands-on example will help you practice using `getImageData()` and `putImageData()` for pixel manipulation and understand how to efficiently apply filters based on user input.

11.4.1 Step 1: Setting Up the Canvas and Image

First, create an HTML structure with a canvas, buttons to switch filters, and an image loaded onto the canvas:

```
<canvas id="canvas" width="400" height="300"></canvas>
<br />
<button id="originalBtn">Original</button>
<button id="grayscaleBtn">Grayscale</button>
<button id="invertBtn">Invert Colors</button>
```

In your JavaScript, load an image and draw it onto the canvas:

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
const img = new Image();

img.src = 'your-image.jpg'; // Use any image URL or local path
img.onload = () => {
  ctx.drawImage(img, 0, 0, canvas.width, canvas.height);
};
```

11.4.2 Step 2: Accessing and Storing Image Data

We'll keep a copy of the original image data so we can revert or apply filters from the base image:

```
let originalImageData = null;

img.onload = () => {
  ctx.drawImage(img, 0, 0, canvas.width, canvas.height);
  originalImageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
};
```

11.4.3 Step 3: Creating the Filter Functions

Grayscale Filter

To convert to grayscale, average the red, green, and blue components for each pixel and assign that value back to all three channels:

```
function applyGrayscale(imageData) {
  const data = imageData.data;
  for (let i = 0; i < data.length; i += 4) {
    const avg = (data[i] + data[i + 1] + data[i + 2]) / 3;
    data[i] = data[i + 1] = data[i + 2] = avg;
  }
  return imageData;
}
```

Invert Colors Filter

Invert colors by subtracting each color component from 255:

```
function applyInvert(imageData) {
  const data = imageData.data;
  for (let i = 0; i < data.length; i += 4) {
    data[i] = 255 - data[i];     // Red
    data[i + 1] = 255 - data[i + 1]; // Green
    data[i + 2] = 255 - data[i + 2]; // Blue
  }
  return imageData;
}
```

11.4.4 Step 4: Handling Button Clicks to Switch Filters

Add event listeners to the buttons to apply the selected filter and update the canvas:

```
document.getElementById('originalBtn').addEventListener('click', () => {
  if (originalImageData) {
    ctx.putImageData(originalImageData, 0, 0);
  }
});

document.getElementById('grayscaleBtn').addEventListener('click', () => {
  if (originalImageData) {
    // Create a copy to avoid modifying original data
    const imageDataCopy = new ImageData(
      new Uint8ClampedArray(originalImageData.data),
      originalImageData.width,
      originalImageData.height
    );
    ctx.putImageData(applyGrayscale(imageDataCopy), 0, 0);
  }
});
```

```
document.getElementById('invertBtn').addEventListener('click', () => {
  if (originalImageData) {
    const imageDataCopy = new ImageData(
      new Uint8ClampedArray(originalImageData.data),
      originalImageData.width,
      originalImageData.height
    );
    ctx.putImageData(applyInvert(imageDataCopy), 0, 0);
  }
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Canvas Image Filters</title>
  <style>
    canvas {
      border: 1px solid #ccc;
      display: block;
      margin-bottom: 10px;
    }
    button {
      margin-right: 5px;
      padding: 5px 10px;
    }
  </style>
</head>
<body>

<canvas id="canvas" width="400" height="300"></canvas>
<br />
<button id="originalBtn">Original</button>
<button id="grayscaleBtn">Grayscale</button>
<button id="invertBtn">Invert Colors</button>

<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');
  const img = new Image();

  // Use a sample image URL (CORS-enabled)
  img.crossOrigin = "anonymous";
  img.src = 'https://upload.wikimedia.org/wikipedia/commons/thumb/a/a9/Example.jpg/400px-Example.jpg';

  let originalImageData = null;

  img.onload = () => {
    ctx.drawImage(img, 0, 0, canvas.width, canvas.height);
    originalImageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
  };

  function applyGrayscale(imageData) {
    const data = imageData.data;
    for (let i = 0; i < data.length; i += 4) {
      const avg = (data[i] + data[i + 1] + data[i + 2]) / 3;
      data[i] = data[i + 1] = data[i + 2] = avg;
```

```
    }
    return imageData;
  }

  function applyInvert(imageData) {
    const data = imageData.data;
    for (let i = 0; i < data.length; i += 4) {
      data[i] = 255 - data[i];       // Red
      data[i + 1] = 255 - data[i + 1]; // Green
      data[i + 2] = 255 - data[i + 2]; // Blue
    }
    return imageData;
  }

  document.getElementById('originalBtn').addEventListener('click', () => {
    if (originalImageData) {
      ctx.putImageData(originalImageData, 0, 0);
    }
  });

  document.getElementById('grayscaleBtn').addEventListener('click', () => {
    if (originalImageData) {
      const imageDataCopy = new ImageData(
        new Uint8ClampedArray(originalImageData.data),
        originalImageData.width,
        originalImageData.height
      );
      ctx.putImageData(applyGrayscale(imageDataCopy), 0, 0);
    }
  });

  document.getElementById('invertBtn').addEventListener('click', () => {
    if (originalImageData) {
      const imageDataCopy = new ImageData(
        new Uint8ClampedArray(originalImageData.data),
        originalImageData.width,
        originalImageData.height
      );
      ctx.putImageData(applyInvert(imageDataCopy), 0, 0);
    }
  });
</script>

</body>
</html>
```

11.4.5 Step 5: Optimizations and Enhancements

- **Efficient Looping:** The loops increment by 4 because each pixel consists of 4 values (Red, Green, Blue, Alpha). This is the most efficient way to iterate pixels.
- **Selective Filtering:** You can extend the functions to apply filters only within a selected rectangular region or on mouse hover by limiting the loop to the corresponding

pixel indices.

- **Modular Design:** The filter functions take and return an `ImageData` object, making it easy to combine filters or build pipelines.

11.4.6 Summary

This example combines key canvas pixel manipulation techniques:

- Using `getImageData()` to access raw pixel RGBA data.
- Processing pixels efficiently with loops.
- Applying filters like grayscale and invert by modifying color values.
- Using `putImageData()` to update the canvas display.
- Keeping the original image data intact for toggling filters seamlessly.

Try extending this project by adding sliders for brightness or contrast, or implementing additional filters like sepia or blur. Experiment with applying filters dynamically on mouse movement or selecting parts of the image to edit!

Chapter 12.

3D Effects and WebGL Basics

1. Introduction to WebGL and Differences from 2D Canvas
2. Setting Up a Basic WebGL Context
3. Drawing Simple 3D Shapes
4. Practical Example: Rotating Cube with WebGL

12 3D Effects and WebGL Basics

12.1 Introduction to WebGL and Differences from 2D Canvas

As you become more comfortable with 2D canvas programming, you may start wondering how to create immersive 3D graphics—rotating cubes, animated terrains, or data visualizations with depth and perspective. This is where **WebGL** comes into play.

12.1.1 What Is WebGL?

WebGL (Web Graphics Library) is a JavaScript API that allows you to render 2D and **3D graphics directly in the browser** using the GPU. Unlike the familiar 2D canvas API, WebGL is a **low-level** interface that provides access to advanced graphics pipelines, making it powerful but also more complex.

WebGL is based on **OpenGL ES**, a graphics standard for embedded systems, and is designed to bring high-performance, real-time rendering to the web.

12.1.2 WebGL vs. 2D Canvas: Key Differences

Feature	2D Canvas	WebGL
Rendering Mode	Immediate-mode, CPU-based	Retained-mode, GPU-accelerated
Drawing API	Simple shapes and styles	Requires shaders and buffers
3D Support	None (2D only)	Full 3D (perspective, depth, lighting)
Abstraction Level	High (easy to use)	Low (requires setup and shaders)
Use Cases	Charts, games, art, UI	3D games, scientific visualization, VR
Performance	Fine for basic graphics	Better for complex or dynamic scenes

12.1.3 Why Use WebGL?

WebGL enables things that 2D canvas simply can't do—or can't do efficiently. Here are some common use cases:

- **3D Games**: Many modern browser games (e.g., using Three.js) rely on WebGL.
- **Data Visualization**: Render thousands of data points or surfaces with depth.
- **Augmented & Virtual Reality**: Integrate 3D elements into immersive applications.
- **3D Modeling and Simulation**: Physics simulations, architectural previews, etc.

12.1.4 GPU Acceleration and Shaders

The biggest shift from 2D canvas to WebGL is **how rendering happens**:

- Instead of issuing draw calls like `fillRect()` or `arc()`, in WebGL you create **geometry** (vertex data), send it to the GPU, and write **shaders**—small GPU programs—to control how the shapes appear.
- There are two types of shaders:
  - **Vertex shaders**: Calculate vertex positions.
  - **Fragment shaders**: Calculate pixel color.

This design offers immense power and flexibility—but it also means **more setup**, **mathematics**, and **boilerplate** compared to 2D drawing.

12.1.5 The Learning Curve: Worth It?

If you're coming from the 2D canvas world, WebGL can initially feel overwhelming. You'll need to learn new concepts like:

- Matrix transformations
- Vertex buffers
- Shading languages (GLSL)
- Coordinate systems in 3D

However, **gaining a foundational understanding of WebGL is incredibly valuable**—even for 2D developers. You'll better understand how GPUs work, and you'll be equipped to explore high-performance graphics, animation, and visualization in the browser.

Many libraries such as **Three.js** or **Babylon.js** build on WebGL and provide higher-level APIs. But learning raw WebGL gives you **total control and insight**, and allows you to write your own optimized rendering systems.

12.1.6 Summary

WebGL opens the door to the 3D web. Compared to the 2D canvas API, it's a **lower-level**, more **powerful**, and **GPU-accelerated** system that requires more effort—but rewards you with the ability to render complex, interactive scenes in real time.

In the next section, you'll set up your first **WebGL context** and draw a blank screen. From there, we'll build up to rendering 3D objects like a rotating cube using shaders and transformations.

Let's step into the world of 3D.

12.2 Setting Up a Basic WebGL Context

Before we can render anything in 3D, we need to properly initialize a WebGL context and understand the setup process. Unlike the 2D canvas API, WebGL involves more boilerplate—but once you're past the initial steps, it becomes a powerful rendering engine.

12.2.1 Creating a WebGL Context

To start using WebGL, you first need to get a WebGL context from a `<canvas>` element. This context is your interface to the GPU.

```
<canvas id="glcanvas" width="500" height="500"></canvas>
```

```
const canvas = document.getElementById("glcanvas");
const gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");

if (!gl) {
  alert("WebGL not supported on this browser.");
}
```

12.2.2 WebGL Context Options

When requesting a context, you can pass options like this:

```
const gl = canvas.getContext("webgl", {
  alpha: false,    // disable alpha channel (useful for opaque scenes)
  depth: true,     // enable depth buffer (required for 3D)
  antialias: true, // smoother edges
});
```

Always check if the context was returned—some devices or browsers might not support WebGL.

12.2.3 Writing a Minimal WebGL Program

WebGL rendering is done by GPU programs called *shaders*. You'll need at least two shaders: a **vertex shader** and a **fragment shader**.

12.2.4 Step 1: Define Shaders

```
const vertexShaderSource = `
  attribute vec2 a_position;
  void main() {
    gl_Position = vec4(a_position, 0.0, 1.0);
  }
`;

const fragmentShaderSource = `
  void main() {
    gl_FragColor = vec4(1.0, 0.2, 0.6, 1.0); // pinkish triangle
  }
`;
```

12.2.5 Step 2: Compile Shaders and Link Program

```
function createShader(gl, type, source) {
  const shader = gl.createShader(type);
  gl.shaderSource(shader, source);
  gl.compileShader(shader);
  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    throw new Error(gl.getShaderInfoLog(shader));
  }
  return shader;
}

function createProgram(gl, vertexSource, fragmentSource) {
  const vShader = createShader(gl, gl.VERTEX_SHADER, vertexSource);
  const fShader = createShader(gl, gl.FRAGMENT_SHADER, fragmentSource);
  const program = gl.createProgram();
  gl.attachShader(program, vShader);
  gl.attachShader(program, fShader);
  gl.linkProgram(program);
  if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
    throw new Error(gl.getProgramInfoLog(program));
  }
  return program;
}
```

12.2.6 Step 3: Create a Buffer and Draw a Triangle

```
const program = createProgram(gl, vertexShaderSource, fragmentShaderSource);
gl.useProgram(program);

// Define triangle points in clip space (-1 to 1)
```

```
const positionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
const vertices = new Float32Array([
   0.0,  0.5,
  -0.5, -0.5,
   0.5, -0.5,
]);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

// Link buffer to attribute in shader
const aPosition = gl.getAttribLocation(program, "a_position");
gl.enableVertexAttribArray(aPosition);
gl.vertexAttribPointer(aPosition, 2, gl.FLOAT, false, 0, 0);

// Set viewport and clear canvas
gl.viewport(0, 0, canvas.width, canvas.height);
gl.clearColor(0.1, 0.1, 0.1, 1.0); // dark background
gl.clear(gl.COLOR_BUFFER_BIT);

// Draw triangle
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

12.2.7 Minimal Working Example

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>WebGL Triangle</title>
  <style>canvas { border: 1px solid #ccc; }</style>
</head>
<body>
  <canvas id="glcanvas" width="500" height="500"></canvas>
  <script>
    // Setup code shown above goes here...
    // (copy createShader, createProgram, and rendering logic)
  </script>
</body>
</html>
```

When you open this page, you should see a colored triangle centered on a dark background. This is your first working WebGL scene!

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Minimal WebGL Triangle</title>
  <style>
    canvas { border: 1px solid #ccc; }
  </style>
```

```
</head>
<body>
  <canvas id="glcanvas" width="500" height="500"></canvas>
  <script>
    const canvas = document.getElementById("glcanvas");
    const gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");

    if (!gl) {
      alert("WebGL not supported on this browser.");
      throw new Error("WebGL not supported.");
    }

    const vertexShaderSource = `
      attribute vec2 a_position;
      void main() {
        gl_Position = vec4(a_position, 0.0, 1.0);
      }
    `;

    const fragmentShaderSource = `
      void main() {
        gl_FragColor = vec4(1.0, 0.2, 0.6, 1.0); // pinkish triangle
      }
    `;

    function createShader(gl, type, source) {
      const shader = gl.createShader(type);
      gl.shaderSource(shader, source);
      gl.compileShader(shader);
      if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        console.error(gl.getShaderInfoLog(shader));
        gl.deleteShader(shader);
        return null;
      }
      return shader;
    }

    function createProgram(gl, vertexSource, fragmentSource) {
      const vShader = createShader(gl, gl.VERTEX_SHADER, vertexSource);
      const fShader = createShader(gl, gl.FRAGMENT_SHADER, fragmentSource);
      const program = gl.createProgram();
      gl.attachShader(program, vShader);
      gl.attachShader(program, fShader);
      gl.linkProgram(program);
      if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
        console.error(gl.getProgramInfoLog(program));
        gl.deleteProgram(program);
        return null;
      }
      return program;
    }

    const program = createProgram(gl, vertexShaderSource, fragmentShaderSource);
    gl.useProgram(program);

    const vertices = new Float32Array([
       0.0,  0.5,
      -0.5, -0.5,
```

```
      0.5, -0.5,
    ]);

    const positionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

    const aPosition = gl.getAttribLocation(program, "a_position");
    gl.enableVertexAttribArray(aPosition);
    gl.vertexAttribPointer(aPosition, 2, gl.FLOAT, false, 0, 0);

    gl.viewport(0, 0, canvas.width, canvas.height);
    gl.clearColor(0.1, 0.1, 0.1, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);

    gl.drawArrays(gl.TRIANGLES, 0, 3);
  </script>
</body>
</html>
```

12.2.8 Summary

In this section, you've:

- Created a WebGL context
- Defined and compiled shaders
- Created a buffer with vertex data
- Rendered your first triangle to the screen

This foundation is essential for building more advanced 3D scenes. Next, you'll learn how to draw real 3D shapes using transformations and perspective projection.

Let's get rendering!

12.3 Drawing Simple 3D Shapes

In this section, you'll learn how to draw simple 3D shapes using WebGL. While 2D rendering on a canvas can feel immediate, 3D rendering involves more components: vertices, buffers, transformation matrices, and sometimes lighting. We'll walk through drawing a colored rotating cube and touch on the math and techniques used behind the scenes.

12.3.1 3D Rendering Concepts

Before jumping into code, here are the foundational elements of 3D rendering in WebGL:

- **Vertices**: Points in 3D space that define the shape's geometry.
- **Indices**: Optional—used to reuse vertices when forming triangles.
- **Buffers**: Store vertex data and send it to the GPU.
- **Shaders**: Programs running on the GPU to transform vertices and color pixels.
- **Matrices**: Used to rotate, scale, move, and project objects in 3D space.
- **Camera/View**: The perspective through which we see the scene.

12.3.2 Step 1: Define Cube Geometry

Let's start with a cube. A cube has 8 unique vertices, but to give each face its own color and normal direction, we often define **24 vertices** (4 per face × 6 faces).

```
const cubeVertices = new Float32Array([
  // X, Y, Z       R, G, B
  // Front face
  -1, -1,  1,      1, 0, 0,
   1, -1,  1,      1, 0, 0,
   1,  1,  1,      1, 0, 0,
  -1,  1,  1,      1, 0, 0,
  // ... define other 5 faces similarly
]);

const cubeIndices = new Uint16Array([
  0, 1, 2,   0, 2, 3, // front
  // ... other faces
]);
```

Each vertex has 6 values: 3 for position and 3 for color.

12.3.3 Step 2: Use glMatrix for Transformations

WebGL doesn't include matrix math tools, but libraries like glMatrix make it easier:

```
<script src="https://cdn.jsdelivr.net/npm/gl-matrix@3.4.3/gl-matrix-min.js"></script>
```

You'll need matrices for model, view, and projection transformations:

```
const modelMatrix = mat4.create();       // object transform
const viewMatrix = mat4.create();        // camera
const projectionMatrix = mat4.create();  // perspective

mat4.lookAt(viewMatrix, [0, 0, 6], [0, 0, 0], [0, 1, 0]);
```

```
mat4.perspective(projectionMatrix, Math.PI / 4, canvas.width / canvas.height, 0.1, 100);
```

You combine these matrices in the vertex shader to place objects in the 3D scene.

12.3.4 Step 3: Minimal Shader Code

Vertex Shader

```
attribute vec3 a_position;
attribute vec3 a_color;

uniform mat4 u_model;
uniform mat4 u_view;
uniform mat4 u_projection;

varying vec3 v_color;

void main() {
  gl_Position = u_projection * u_view * u_model * vec4(a_position, 1.0);
  v_color = a_color;
}
```

Fragment Shader

```
precision mediump float;
varying vec3 v_color;

void main() {
  gl_FragColor = vec4(v_color, 1.0);
}
```

12.3.5 Step 4: Rendering Loop with Rotation

Use `requestAnimationFrame` to animate the cube:

```
function render(time) {
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  // Apply rotation
  mat4.rotateY(modelMatrix, modelMatrix, 0.01);
  mat4.rotateX(modelMatrix, modelMatrix, 0.005);

  gl.uniformMatrix4fv(modelLoc, false, modelMatrix);
  gl.uniformMatrix4fv(viewLoc, false, viewMatrix);
  gl.uniformMatrix4fv(projectionLoc, false, projectionMatrix);
```

```
  gl.drawElements(gl.TRIANGLES, cubeIndices.length, gl.UNSIGNED_SHORT, 0);
  requestAnimationFrame(render);
}
```

12.3.6 Step 5: Add Depth and Enable 3D Settings

Don't forget to enable depth testing so closer faces hide those behind them:

```
gl.enable(gl.DEPTH_TEST);
gl.clearColor(0.1, 0.1, 0.1, 1);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Rotating 3D Cube - WebGL</title>
  <style>
    canvas { border: 1px solid #ccc; display: block; margin: 20px auto; }
  </style>
</head>
<body>
<canvas id="glcanvas" width="500" height="500"></canvas>
<script src="https://cdn.jsdelivr.net/npm/gl-matrix@3.4.3/gl-matrix-min.js"></script>
<script>
const canvas = document.getElementById("glcanvas");
const gl = canvas.getContext("webgl");
if (!gl) {
  alert("WebGL not supported");
  throw new Error("WebGL not supported");
}

// Set viewport to canvas size
gl.viewport(0, 0, canvas.width, canvas.height);

// Enable depth testing
gl.enable(gl.DEPTH_TEST);
gl.clearColor(0.1, 0.1, 0.1, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

// --- Vertex Data (cube) ---
const cubeVertices = new Float32Array([
  // X, Y, Z       R, G, B
  // Front
  -1, -1,  1,     1, 0, 0,
   1, -1,  1,     1, 0, 0,
   1,  1,  1,     1, 0, 0,
  -1,  1,  1,     1, 0, 0,
  // Back
  -1, -1, -1,     0, 1, 0,
   1, -1, -1,     0, 1, 0,
   1,  1, -1,     0, 1, 0,
  -1,  1, -1,     0, 1, 0,
```

```
  // Top
  -1,  1, -1,     0, 0, 1,
   1,  1, -1,     0, 0, 1,
   1,  1,  1,     0, 0, 1,
  -1,  1,  1,     0, 0, 1,
  // Bottom
  -1, -1, -1,     1, 1, 0,
   1, -1, -1,     1, 1, 0,
   1, -1,  1,     1, 1, 0,
  -1, -1,  1,     1, 1, 0,
  // Right
   1, -1, -1,     0, 1, 1,
   1,  1, -1,     0, 1, 1,
   1,  1,  1,     0, 1, 1,
   1, -1,  1,     0, 1, 1,
  // Left
  -1, -1, -1,     1, 0, 1,
  -1,  1, -1,     1, 0, 1,
  -1,  1,  1,     1, 0, 1,
  -1, -1,  1,     1, 0, 1,
]);

const cubeIndices = new Uint16Array([
  0, 1, 2, 0, 2, 3,       // front
  4, 5, 6, 4, 6, 7,       // back
  8, 9,10, 8,10,11,       // top
 12,13,14,12,14,15,       // bottom
 16,17,18,16,18,19,       // right
 20,21,22,20,22,23        // left
]);

// --- Shader Setup ---
const vsSource = `
  attribute vec3 a_position;
  attribute vec3 a_color;
  uniform mat4 u_model;
  uniform mat4 u_view;
  uniform mat4 u_projection;
  varying vec3 v_color;
  void main() {
    gl_Position = u_projection * u_view * u_model * vec4(a_position, 1.0);
    v_color = a_color;
  }
`;

const fsSource = `
  precision mediump float;
  varying vec3 v_color;
  void main() {
    gl_FragColor = vec4(v_color, 1.0);
  }
`;

function createShader(type, source) {
  const shader = gl.createShader(type);
  gl.shaderSource(shader, source);
  gl.compileShader(shader);
  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
```

```
    console.error(gl.getShaderInfoLog(shader));
    return null;
  }
  return shader;
}

function createProgram(vs, fs) {
  const program = gl.createProgram();
  gl.attachShader(program, vs);
  gl.attachShader(program, fs);
  gl.linkProgram(program);
  if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
    console.error(gl.getProgramInfoLog(program));
    return null;
  }
  return program;
}

const vertexShader = createShader(gl.VERTEX_SHADER, vsSource);
const fragmentShader = createShader(gl.FRAGMENT_SHADER, fsSource);
const program = createProgram(vertexShader, fragmentShader);
gl.useProgram(program);

// --- Buffers ---
const vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, cubeVertices, gl.STATIC_DRAW);

const indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, cubeIndices, gl.STATIC_DRAW);

// Attributes
const aPosition = gl.getAttribLocation(program, 'a_position');
gl.vertexAttribPointer(aPosition, 3, gl.FLOAT, false, 6 * 4, 0);
gl.enableVertexAttribArray(aPosition);

const aColor = gl.getAttribLocation(program, 'a_color');
gl.vertexAttribPointer(aColor, 3, gl.FLOAT, false, 6 * 4, 3 * 4);
gl.enableVertexAttribArray(aColor);

// Uniforms
const modelLoc = gl.getUniformLocation(program, 'u_model');
const viewLoc = gl.getUniformLocation(program, 'u_view');
const projectionLoc = gl.getUniformLocation(program, 'u_projection');
const mat4 = glMatrix.mat4;
// Matrices
const modelMatrix = mat4.create();
const viewMatrix = mat4.create();
const projectionMatrix = mat4.create();

mat4.lookAt(viewMatrix, [0, 0, 6], [0, 0, 0], [0, 1, 0]);
mat4.perspective(projectionMatrix, Math.PI / 4, canvas.width / canvas.height, 0.1, 100);

// Animation Loop
function render() {
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

```
  mat4.rotateY(modelMatrix, modelMatrix, 0.01);
  mat4.rotateX(modelMatrix, modelMatrix, 0.005);

  gl.uniformMatrix4fv(modelLoc, false, modelMatrix);
  gl.uniformMatrix4fv(viewLoc, false, viewMatrix);
  gl.uniformMatrix4fv(projectionLoc, false, projectionMatrix);

  gl.drawElements(gl.TRIANGLES, cubeIndices.length, gl.UNSIGNED_SHORT, 0);
  requestAnimationFrame(render);
}

render();
</script>
</body>
</html>
```

12.3.7 Visual Output

The cube should spin in place, each face with a different color. Try modifying:

- Rotation speed
- Face colors
- Shape geometry (try a pyramid!)
- Camera position

12.3.8 Summary

You've now:

YES Defined a 3D shape using vertices and indices YES Set up view and projection matrices YES Written vertex and fragment shaders YES Rendered a rotating 3D object on the screen

12.3.9 Challenge: Try Drawing a Pyramid or Sphere

- For a **pyramid**, use 5 faces: a square base and 4 triangle sides.
- For a **sphere**, you'll need to generate lat/lon-based vertices—more complex, but visually satisfying.

In the next section, we'll build a full rotating cube demo with UI controls to toggle rotation, color, and projection type.

Let's continue building your 3D toolkit!

12.4 Practical Example: Rotating Cube with WebGL

In this final section of the chapter, we'll walk through building a complete WebGL project: a colorful 3D cube that rotates on the canvas. We'll animate it over time, use shaders to color it, and add basic interactivity with mouse or keyboard controls.

By the end, you'll understand how to build a simple WebGL application and how all the components—geometry, buffers, shaders, and animation—work together.

12.4.1 Objective

Create a rotating 3D cube using WebGL that:

- Spins continuously around one or more axes.
- Uses shaders for solid coloring or basic lighting.
- Accepts user input (e.g., arrow keys) to control rotation.
- Provides a solid foundation for expanding into textures or lighting effects.

12.4.2 Step 1: Setup the Canvas and WebGL Context

```
<canvas id="glcanvas" width="500" height="500"></canvas>
<script src="https://cdn.jsdelivr.net/npm/gl-matrix@3.4.3/gl-matrix-min.js"></script>
```

```
const canvas = document.getElementById('glcanvas');
const gl = canvas.getContext('webgl');

if (!gl) {
  alert('WebGL not supported in this browser.');
}
```

12.4.3 Step 2: Define Shaders

Vertex Shader:

```
attribute vec3 a_position;
attribute vec3 a_color;

uniform mat4 u_model;
uniform mat4 u_view;
uniform mat4 u_projection;
```

```
varying vec3 v_color;

void main() {
  gl_Position = u_projection * u_view * u_model * vec4(a_position, 1.0);
  v_color = a_color;
}
```

Fragment Shader:

```
precision mediump float;
varying vec3 v_color;

void main() {
  gl_FragColor = vec4(v_color, 1.0);
}
```

12.4.4 Step 3: Create Cube Geometry and Colors

Define cube vertices and their associated colors:

```
const positions = [
  // Front face
  -1, -1,  1,   1, 0, 0,
   1, -1,  1,   1, 0, 0,
   1,  1,  1,   1, 0, 0,
  -1,  1,  1,   1, 0, 0,
  // Back face
  -1, -1, -1,   0, 1, 0,
   1, -1, -1,   0, 1, 0,
   1,  1, -1,   0, 1, 0,
  -1,  1, -1,   0, 1, 0,
];

const indices = [
  0, 1, 2,  0, 2, 3,    // front
  4, 5, 6,  4, 6, 7,    // back
  0, 3, 7,  0, 7, 4,    // left
  1, 5, 6,  1, 6, 2,    // right
  3, 2, 6,  3, 6, 7,    // top
  0, 1, 5,  0, 5, 4     // bottom
];
```

12.4.5 Step 4: Animate the Cube

Use `requestAnimationFrame` to update the cube's rotation based on time and user input:

```
let rotationX = 0;
let rotationY = 0;
let rotateSpeedX = 0.01;
let rotateSpeedY = 0.01;

function drawScene(time) {
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  const modelMatrix = mat4.create();
  mat4.rotateX(modelMatrix, modelMatrix, rotationX);
  mat4.rotateY(modelMatrix, modelMatrix, rotationY);

  gl.uniformMatrix4fv(u_modelLoc, false, modelMatrix);
  gl.uniformMatrix4fv(u_viewLoc, false, viewMatrix);
  gl.uniformMatrix4fv(u_projectionLoc, false, projectionMatrix);

  gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_SHORT, 0);

  rotationX += rotateSpeedX;
  rotationY += rotateSpeedY;

  requestAnimationFrame(drawScene);
}
```

12.4.6 Step 5: Add Keyboard Controls

Let users control the speed of rotation with arrow keys:

```
window.addEventListener('keydown', (e) => {
  if (e.key === 'ArrowLeft') rotateSpeedY -= 0.01;
  if (e.key === 'ArrowRight') rotateSpeedY += 0.01;
  if (e.key === 'ArrowUp') rotateSpeedX -= 0.01;
  if (e.key === 'ArrowDown') rotateSpeedX += 0.01;
});
```

12.4.7 Step 6: Run It!

Make sure:

- You call `gl.enable(gl.DEPTH_TEST);`
- Set clear color: `gl.clearColor(0, 0, 0, 1);`
- Bind and enable position and color buffers.

Start the animation with:

```
requestAnimationFrame(drawScene);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Rotating WebGL Cube with Controls</title>
  <style>
    canvas { border: 1px solid #ccc; display: block; margin: 20px auto; }
  </style>
</head>
<body>
  <canvas id="glcanvas" width="500" height="500"></canvas>
  <script src="https://cdn.jsdelivr.net/npm/gl-matrix@3.4.3/gl-matrix-min.js"></script>
  <script>
    const canvas = document.getElementById('glcanvas');
    const gl = canvas.getContext('webgl');
    if (!gl) {
      alert('WebGL not supported in this browser.');
      throw new Error('WebGL not supported.');
    }

    // Enable depth
    gl.enable(gl.DEPTH_TEST);
    gl.clearColor(0, 0, 0, 1);

    // Vertex shader
    const vsSource = `
      attribute vec3 a_position;
      attribute vec3 a_color;
      uniform mat4 u_model;
      uniform mat4 u_view;
      uniform mat4 u_projection;
      varying vec3 v_color;
      void main() {
        gl_Position = u_projection * u_view * u_model * vec4(a_position, 1.0);
        v_color = a_color;
      }
    `;

    // Fragment shader
    const fsSource = `
      precision mediump float;
      varying vec3 v_color;
      void main() {
        gl_FragColor = vec4(v_color, 1.0);
      }
    `;

    function createShader(gl, type, source) {
      const shader = gl.createShader(type);
      gl.shaderSource(shader, source);
      gl.compileShader(shader);
      if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        console.error(gl.getShaderInfoLog(shader));
        return null;
      }
      return shader;
    }
```

```
function createProgram(gl, vs, fs) {
  const program = gl.createProgram();
  gl.attachShader(program, vs);
  gl.attachShader(program, fs);
  gl.linkProgram(program);
  if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
    console.error(gl.getProgramInfoLog(program));
    return null;
  }
  return program;
}

const vertexShader = createShader(gl, gl.VERTEX_SHADER, vsSource);
const fragmentShader = createShader(gl, gl.FRAGMENT_SHADER, fsSource);
const program = createProgram(gl, vertexShader, fragmentShader);
gl.useProgram(program);

// Geometry and colors
const positions = new Float32Array([
  // Front
  -1, -1,  1,   1, 0, 0,
   1, -1,  1,   1, 0, 0,
   1,  1,  1,   1, 0, 0,
  -1,  1,  1,   1, 0, 0,
  // Back
  -1, -1, -1,   0, 1, 0,
   1, -1, -1,   0, 1, 0,
   1,  1, -1,   0, 1, 0,
  -1,  1, -1,   0, 1, 0,
]);

const indices = new Uint16Array([
  0, 1, 2, 0, 2, 3,     // front
  4, 5, 6, 4, 6, 7,     // back
  0, 3, 7, 0, 7, 4,     // left
  1, 5, 6, 1, 6, 2,     // right
  3, 2, 6, 3, 6, 7,     // top
  0, 1, 5, 0, 5, 4      // bottom
]);

// Create and bind buffers
const vbo = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
gl.bufferData(gl.ARRAY_BUFFER, positions, gl.STATIC_DRAW);

const ibo = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, ibo);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);

// Attributes
const aPos = gl.getAttribLocation(program, 'a_position');
const aCol = gl.getAttribLocation(program, 'a_color');
gl.enableVertexAttribArray(aPos);
gl.enableVertexAttribArray(aCol);
gl.vertexAttribPointer(aPos, 3, gl.FLOAT, false, 6 * 4, 0);
gl.vertexAttribPointer(aCol, 3, gl.FLOAT, false, 6 * 4, 3 * 4);

// Uniforms
```

```
    const u_modelLoc = gl.getUniformLocation(program, 'u_model');
    const u_viewLoc = gl.getUniformLocation(program, 'u_view');
    const u_projectionLoc = gl.getUniformLocation(program, 'u_projection');

    const mat4 = glMatrix.mat4;
    const modelMatrix = mat4.create();
    const viewMatrix = mat4.create();
    const projectionMatrix = mat4.create();

    mat4.lookAt(viewMatrix, [0, 0, 6], [0, 0, 0], [0, 1, 0]);
    mat4.perspective(projectionMatrix, Math.PI / 4, canvas.width / canvas.height, 0.1, 100);

    let rotationX = 0;
    let rotationY = 0;
    let rotateSpeedX = 0.01;
    let rotateSpeedY = 0.01;

    function drawScene() {
      gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

      mat4.identity(modelMatrix);
      mat4.rotateX(modelMatrix, modelMatrix, rotationX);
      mat4.rotateY(modelMatrix, modelMatrix, rotationY);

      gl.uniformMatrix4fv(u_modelLoc, false, modelMatrix);
      gl.uniformMatrix4fv(u_viewLoc, false, viewMatrix);
      gl.uniformMatrix4fv(u_projectionLoc, false, projectionMatrix);

      gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_SHORT, 0);

      rotationX += rotateSpeedX;
      rotationY += rotateSpeedY;

      requestAnimationFrame(drawScene);
    }

    window.addEventListener('keydown', (e) => {
      if (e.key === 'ArrowLeft') rotateSpeedY -= 0.01;
      if (e.key === 'ArrowRight') rotateSpeedY += 0.01;
      if (e.key === 'ArrowUp') rotateSpeedX -= 0.01;
      if (e.key === 'ArrowDown') rotateSpeedX += 0.01;
    });

    requestAnimationFrame(drawScene);
  </script>
</body>
</html>
```

12.4.8 Extensions and Ideas

Once your cube is working, try extending it:

- **Textures** – Use `gl.TEXTURE_2D` to wrap an image around the cube.

- **Lighting** – Add basic directional lighting with normals and a light vector.
- **Mouse interaction** – Let users click and drag to rotate the cube.
- **UI controls** – Use sliders or checkboxes to change rotation speed, toggle axes, or swap colors.

12.4.9 Summary

You've now built a complete 3D rotating cube in WebGL:

- Defined shaders and a rendering pipeline
- Created buffers and geometry
- Added animation and interactivity
- Prepared the foundation for more advanced 3D scenes

In future chapters, you'll learn how to add textures, advanced lighting, and interactable 3D scenes. Keep experimenting!

Chapter 13.

Performance Optimization Techniques

1. Reducing Overdraw and Optimizing Draw Calls
2. Using Offscreen Canvases and Buffering
3. Efficient Animation and Resource Management
4. Practical Example: High-Performance Canvas Animation

13 Performance Optimization Techniques

13.1 Reducing Overdraw and Optimizing Draw Calls

As canvas-based applications grow in complexity—especially games, animations, and interactive tools—**performance becomes critical**. One major performance drain is **overdraw**, where the same pixels are drawn multiple times unnecessarily.

This section introduces what overdraw is, how to detect and avoid it, and practical strategies to **optimize draw calls** for smoother performance.

13.1.1 What Is Overdraw?

Overdraw happens when the same pixel on the canvas is redrawn multiple times per frame, often without visual benefit. For example:

- Drawing a background image, then a full-screen semi-transparent overlay, then more shapes on top.
- Redrawing the *entire* canvas even though only a small object moved.

In many apps, overdraw can quickly lead to wasted GPU/CPU cycles—especially on lower-powered devices.

13.1.2 Key Concepts

Term	Description
Overdraw	Rendering operations that paint the same pixel repeatedly in a single frame.
Draw Calls	Individual calls to `fillRect()`, `drawImage()`, `stroke()`, etc.
Dirty Rectangles	Technique that redraws only the parts of the screen that changed.
Layered Canvases	Using multiple `<canvas>` layers stacked via CSS to separate static and dynamic content.

13.1.3 Baseline: Full Redraw Example

```
function drawSceneFull(context) {
  context.clearRect(0, 0, canvas.width, canvas.height);
  drawBackground(context); // Always draws full background
  drawAllSprites(context); // Redraws every object, even unchanged
}
```

While this works fine for small scenes, it's **wasteful** if most content stays static.

13.1.4 Optimized Strategy: Dirty Rectangle Tracking

Only update regions that actually changed.

```
function drawDirtyRegion(context, rect) {
  context.clearRect(rect.x, rect.y, rect.width, rect.height);
  drawMovingObject(context, rect.x, rect.y); // Only redraw what changed
}
```

You'll need to **track the old and new positions** of moving objects and compute the union of the affected rectangles.

```
function getDirtyRect(oldPos, newPos, size) {
  const x = Math.min(oldPos.x, newPos.x);
  const y = Math.min(oldPos.y, newPos.y);
  const w = size + Math.abs(oldPos.x - newPos.x);
  const h = size + Math.abs(oldPos.y - newPos.y);
  return { x, y, width: w, height: h };
}
```

13.1.5 Optimization Technique: Multiple Canvas Layers

Separate static and dynamic drawings:

```
<div style="position: relative;">
  <canvas id="bg" width="500" height="500" style="position: absolute;"></canvas>
  <canvas id="fg" width="500" height="500" style="position: absolute;"></canvas>
</div>
```

Draw background once to `#bg`, then only update dynamic elements on `#fg`.

```
const bgCtx = document.getElementById('bg').getContext('2d');
const fgCtx = document.getElementById('fg').getContext('2d');

drawBackground(bgCtx); // One-time draw
```

```
function drawForeground() {
  fgCtx.clearRect(0, 0, canvas.width, canvas.height);
  drawMovingSprites(fgCtx);
}
```

This reduces redraws dramatically.

13.1.6 Profiling and Diagnosing Overdraw

You can detect overdraw visually and with tools:

Chrome DevTools Performance Layers

- Use screenshots or rendering stats to detect unnecessary painting.

WebGL Overdraw Visualization

- Tools like Spector.js (for WebGL apps) can show how many times pixels are drawn.

Manual Overdraw Visualizer (2D Canvas)

You can manually debug with a solid-color overlay to see how often a region is being painted:

```
context.fillStyle = 'rgba(255, 0, 0, 0.1)';
context.fillRect(dirtyRect.x, dirtyRect.y, dirtyRect.width, dirtyRect.height);
```

Repeated red overlays show overdraw clearly.

13.1.7 Draw Call Minimization Tips

Tip	Description
Batch draw calls	Group drawing of similar items (e.g. same fill/stroke style) to avoid redundant state changes.
Avoid transparent fill redraws	They trigger blending and are more expensive. Only use transparency when needed.
Cache static scenes	Use `offscreenCanvas` or a separate canvas to cache complex but unchanging drawings.
Clip drawing regions	Use `context.save()` and `context.clip()` to limit rendering when appropriate.

13.1.8 Comparison: Full vs. Optimized Redraw

Method	Frame Time	CPU Usage	GPU Load
Full Redraw	High	High	Medium
Dirty Rectangles	Low	Low	Low
Layered Canvases	Very Low	Minimal	Low

Try profiling both approaches with a large number of moving objects and see the performance difference.

13.1.9 Summary

Reducing overdraw and optimizing draw calls are essential for maintaining **high-performance canvas apps**.

Best practices:

- Redraw only what changes (dirty rectangles)
- Use layered canvases for static vs dynamic content
- Profile frequently to detect inefficiencies
- Keep draw calls minimal and batched when possible

In the next section, you'll learn how **offscreen canvases** can offload rendering and further enhance performance.

13.2 Using Offscreen Canvases and Buffering

As canvas applications grow more dynamic and visually complex, performance becomes a top concern—especially when redrawing large backgrounds, tile maps, or many repeated visuals. One powerful technique for reducing rendering cost is to use **offscreen canvases** for buffering.

This section introduces two approaches: using the `OffscreenCanvas` API and standard in-memory `<canvas>` elements for pre-rendering. Both allow you to **render once**, then **re-use the result**, reducing the workload in the main draw loop.

13.2.1 What is an Offscreen Canvas?

An *offscreen canvas* is a canvas that exists in memory but isn't displayed directly in the DOM. You can draw complex scenes to it once, then copy its contents to the visible canvas repeatedly using `drawImage()`.

Two Ways to Create Offscreen Canvases:

1. **Using `document.createElement('canvas')`:** Works in all modern browsers and is ideal for simple caching.

```
const bufferCanvas = document.createElement('canvas');
const bufferCtx = bufferCanvas.getContext('2d');
```

2. **Using `OffscreenCanvas`:** Introduced for multithreading and Web Workers. Great for advanced use cases but not supported everywhere yet.

```
const offscreen = new OffscreenCanvas(500, 500);
const offCtx = offscreen.getContext('2d');
```

13.2.2 Why Use Offscreen Canvases?

Benefit	Description
Pre-rendering static content	Draw once to a hidden canvas, re-use every frame.
Tile-based games	Render terrain or levels once, then cache them.
Reduce redundant calculations	Save time by avoiding expensive redraws.
Parallel rendering (with `OffscreenCanvas`)	Offload drawing to Web Workers for smoother main-thread performance.

13.2.3 Basic Example: Background Buffering

Let's say we have a complex background grid or terrain that doesn't change every frame.

Step 1: Draw the background once

```
const bgBuffer = document.createElement('canvas');
bgBuffer.width = 800;
bgBuffer.height = 600;
const bgCtx = bgBuffer.getContext('2d');

// Pre-render background
function renderBackgroundToBuffer(ctx) {
  for (let x = 0; x < 800; x += 50) {
    for (let y = 0; y < 600; y += 50) {
```

```
      ctx.fillStyle = (x + y) % 100 === 0 ? 'lightblue' : 'lightgray';
      ctx.fillRect(x, y, 50, 50);
    }
  }
}

renderBackgroundToBuffer(bgCtx);
```

Step 2: Use `drawImage()` to reuse it in each frame

```
function renderFrame(mainCtx) {
  mainCtx.clearRect(0, 0, canvas.width, canvas.height);
  mainCtx.drawImage(bgBuffer, 0, 0); // Fast copy
  drawMovingObjects(mainCtx);
}
```

This is significantly faster than re-computing every tile each frame.

13.2.4 Pattern: Drawing Once, Reusing Many Times

```
function createCachedCircle(radius, fillStyle) {
  const cache = document.createElement('canvas');
  cache.width = cache.height = radius * 2;
  const ctx = cache.getContext('2d');

  ctx.fillStyle = fillStyle;
  ctx.beginPath();
  ctx.arc(radius, radius, radius, 0, Math.PI * 2);
  ctx.fill();

  return cache;
}

// Then inside render loop:
mainCtx.drawImage(cachedCircle, x - r, y - r);
```

13.2.5 Use Cases

Scenario	Benefit of Buffering
Tile Maps	Render a full level to a buffer and scroll it around.
Repeated Patterns	Cache one tile, use `drawImage()` to repeat.
Particles	Render base particle once, then animate using transformations.
Text Effects	Expensive styled text can be drawn to a buffer and reused.

Scenario	Benefit of Buffering
Static Layers	Sky, terrain, scenery, or UI backdrops drawn once per scene.

13.2.6 OffscreenCanvas in Workers (Advanced)

If you're using Web Workers, `OffscreenCanvas` lets you move rendering off the main thread:

```
// In main.js
const worker = new Worker('renderWorker.js');
const offscreen = canvas.transferControlToOffscreen();
worker.postMessage({ canvas: offscreen }, [offscreen]);
```

```
// In renderWorker.js
onmessage = (e) => {
  const ctx = e.data.canvas.getContext('2d');
  renderToOffscreen(ctx);
};
```

> WARNING **Note**: `OffscreenCanvas` isn't supported in all environments—check Can I Use for compatibility.

13.2.7 Performance Considerations

Tip	Explanation
Minimize overdraw	Buffers help avoid redrawing static regions repeatedly.
Avoid memory bloat	Don't keep too many unused canvases in memory.
Pre-scale buffers	Create buffers at the final rendering scale to avoid expensive resizing.
Profile usage	Use browser dev tools to measure canvas performance and rendering times.

13.2.8 Summary

Offscreen canvases and in-memory buffering are essential tools for **high-performance canvas apps**:

- **Render once, reuse many times** for static visuals.

- Use standard `<canvas>` for broad compatibility.
- Use `OffscreenCanvas` with workers for advanced offloading.
- Cache complex drawings and apply efficient copying with `drawImage()`.

These techniques unlock smooth rendering even in graphically rich applications.

Next up: In Section 3, we'll explore how to manage **animations and resources** efficiently over time, including pooling and garbage collection strategies. Ready to continue?

13.3 Efficient Animation and Resource Management

As canvas animations become more complex—with many moving parts, repeated computations, and continuous redraws—**efficient animation and resource management** becomes essential. Without care, performance will degrade due to memory leaks, frame rate drops, and CPU overuse.

This section explores key optimization strategies including **delta time control**, **frame rate stability**, **object pooling**, and proper **resource cleanup**.

13.3.1 Managing Animation State

Every animation involves a set of objects (particles, characters, UI elements) whose state changes over time. These include:

- **Position, velocity, acceleration**
- **Lifetime and visibility**
- **Rendering properties (color, shape, rotation)**

Keeping this state well-organized and bounded is crucial to performance.

13.3.2 Frame Rate Stabilization with Delta Time

Different systems run at different frame rates. If you move an object by 5 pixels per frame, a 60 FPS monitor will behave differently than a 30 FPS one.

To fix this, use **delta time** (`dt`) to make updates time-based instead of frame-based:

```
let lastTime = performance.now();

function animate(now) {
  const deltaTime = (now - lastTime) / 1000; // seconds
  lastTime = now;
```

```
  update(deltaTime);
  render();

  requestAnimationFrame(animate);
}
requestAnimationFrame(animate);
```

Now, `update()` can scale motion:

```
function update(dt) {
  particle.x += particle.vx * dt;
  particle.y += particle.vy * dt;
}
```

YES This results in **consistent animation speed**, regardless of frame rate.

13.3.3 Object Pooling for Reusable Entities

Creating and discarding JavaScript objects each frame (e.g., thousands of new particles) causes **frequent garbage collection** and memory churn.

Instead, reuse objects through **object pooling**:

```
function createParticlePool(size) {
  const pool = [];
  for (let i = 0; i < size; i++) {
    pool.push({ alive: false, x: 0, y: 0, vx: 0, vy: 0, life: 0 });
  }
  return pool;
}

function spawnParticle(pool, x, y) {
  const p = pool.find(p => !p.alive);
  if (p) {
    p.x = x;
    p.y = y;
    p.vx = (Math.random() - 0.5) * 100;
    p.vy = (Math.random() - 0.5) * 100;
    p.life = 1;
    p.alive = true;
  }
}
```

13.3.4 Updating and Cleaning Up

```
function updateParticles(pool, dt) {
  for (const p of pool) {
    if (!p.alive) continue;
    p.life -= dt;
    if (p.life <= 0) {
      p.alive = false;
      continue;
    }
    p.x += p.vx * dt;
    p.y += p.vy * dt;
  }
}
```

YES No new allocations — only state updates and recycling.

13.3.5 Cleaning Up Animation Resources

Failing to release unused resources leads to **memory leaks**. Here's what to watch for:

Resource	Clean-up Practice
`setInterval` / `setTimeout`	Always call `clearInterval()` / `clearTimeout()` when no longer needed.
`requestAnimationFrame()`	Use a flag or cancel logic to stop when appropriate (e.g. on pause or tab close).
Canvas elements	Remove hidden/inactive canvases from DOM or memory.
Event listeners	Call `removeEventListener()` when a component is destroyed.

Example:

```
let running = true;
function animate(now) {
  if (!running) return;
  // ...
  requestAnimationFrame(animate);
}
function stopAnimation() {
  running = false;
}
```

13.3.6 Example: Efficient Particle System with Pooling

```
<canvas id="canvas" width="800" height="400"></canvas>
```

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const pool = createParticlePool(200);
let lastTime = performance.now();

function animate(now) {
  const dt = (now - lastTime) / 1000;
  lastTime = now;

  ctx.clearRect(0, 0, canvas.width, canvas.height);

  spawnParticle(pool, canvas.width / 2, canvas.height / 2);
  updateParticles(pool, dt);
  drawParticles(pool);

  requestAnimationFrame(animate);
}
requestAnimationFrame(animate);

function drawParticles(pool) {
  for (const p of pool) {
    if (!p.alive) continue;
    ctx.fillStyle = `rgba(255, 165, 0, ${p.life})`;
    ctx.beginPath();
    ctx.arc(p.x, p.y, 4, 0, Math.PI * 2);
    ctx.fill();
  }
}
```

YES Efficient: Reuses particle objects and scales motion by delta time.

13.3.7 Best Practices Summary

Strategy	Why It Matters
Delta time updates	Keeps motion consistent across devices.
Object pooling	Reduces GC pressure and improves FPS.
Cleanup of animations/events	Prevents memory leaks and browser slowdowns.
Avoid global state leaks	Use closures and object boundaries to isolate references.

13.3.8 Wrap-Up

Efficient animations aren't just about drawing pretty frames—they're about **keeping memory usage tight**, **ensuring smooth framerates**, and **reusing resources intelligently**.

By implementing delta time logic, object reuse, and cleanup strategies, you're now ready to scale your canvas apps to handle **hundreds or thousands of animated objects** without performance bottlenecks.

Next: We'll bring it all together in a **high-performance canvas animation demo** using real-time updates, buffers, and pooling for maximum speed.

13.4 Practical Example: High-Performance Canvas Animation

Let's now put theory into action by building a performance-optimized canvas animation. In this section, we'll develop a **scrolling starfield** simulation with hundreds of animated elements—using **partial redraws**, **object pooling**, and **offscreen rendering**.

We'll also add an **FPS counter** so you can measure performance gains in real time.

13.4.1 Project Goal: Scrolling Starfield

The starfield will simulate the illusion of movement through space by animating stars that move downward and recycle when they exit the screen.

Key performance techniques used:

- YES **Object pooling**: Reusing star objects instead of recreating them.
- YES **Offscreen rendering**: Pre-rendering static backgrounds.
- YES **Efficient update/draw loop**: Using `requestAnimationFrame()` with `deltaTime`.
- YES **FPS meter**: Tracking and displaying current frame rate.

13.4.2 Step 1: Setup HTML Canvas

```
<canvas id="canvas" width="800" height="400"></canvas>
<div id="fps" style="color: white; font-family: monospace;"></div>
<style>
  body { margin: 0; background: black; overflow: hidden; }
  canvas { display: block; background: black; }
</style>
```

13.4.3 Step 2: Star Object Pool

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
const stars = [];

function createStarPool(count) {
  for (let i = 0; i < count; i++) {
    stars.push({
      x: Math.random() * canvas.width,
      y: Math.random() * canvas.height,
      radius: Math.random() * 2 + 1,
      speed: Math.random() * 50 + 30
    });
  }
}
createStarPool(300); // Try increasing this number!
```

13.4.4 Step 3: Animation Loop with Delta Time

```
let lastTime = performance.now();
let fpsDisplay = document.getElementById('fps');

function animate(now) {
  const dt = (now - lastTime) / 1000;
  lastTime = now;

  update(dt);
  render();

  trackFPS(now);
  requestAnimationFrame(animate);
}
requestAnimationFrame(animate);
```

13.4.5 Step 4: Update and Recycle Stars

```
function update(dt) {
  for (const star of stars) {
    star.y += star.speed * dt;
    if (star.y > canvas.height) {
      star.y = 0;
      star.x = Math.random() * canvas.width;
    }
  }
```

```
}
```

13.4.6 Step 5: Offscreen Background Rendering

Let's cache a subtle star glow using an **offscreen canvas**:

```
const starCanvas = document.createElement('canvas');
const starCtx = starCanvas.getContext('2d');
starCanvas.width = 4;
starCanvas.height = 4;

starCtx.fillStyle = 'white';
starCtx.beginPath();
starCtx.arc(2, 2, 2, 0, Math.PI * 2);
starCtx.fill();
```

13.4.7 Step 6: Drawing Stars with Offscreen Canvas

```
function render() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  for (const star of stars) {
    ctx.globalAlpha = 0.8;
    ctx.drawImage(starCanvas, star.x, star.y, star.radius * 2, star.radius * 2);
  }
  ctx.globalAlpha = 1.0;
}
```

13.4.8 Step 7: FPS Counter

```
let fps = 0, lastFpsTime = 0, frames = 0;

function trackFPS(now) {
  frames++;
  if (now - lastFpsTime >= 1000) {
    fps = frames;
    frames = 0;
    lastFpsTime = now;
    fpsDisplay.textContent = `FPS: ${fps}`;
  }
}
```

13.4.9 Step 8: Stress Test & Optimization Tips

Try increasing the number of stars:

```
createStarPool(1000); // Push performance limits
```

If performance drops:

- **Lower draw frequency**: Skip every other frame.
- **Reduce overdraw**: Don't render offscreen stars.
- **Batch draws**: Group similar draw operations.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Starfield with FPS</title>
  <style>
    body { margin: 0; background: black; overflow: hidden; }
    canvas { display: block; background: black; }
    #fps {
      position: absolute;
      top: 10px;
      left: 10px;
      color: white;
      font-family: monospace;
      font-size: 16px;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="800" height="400"></canvas>
  <div id="fps">FPS: 0</div>

  <script>
    // Setup canvas and context
    const canvas = document.getElementById('canvas');
    const ctx = canvas.getContext('2d');
    const stars = [];

    // Star object pool
    function createStarPool(count) {
      for (let i = 0; i < count; i++) {
        stars.push({
          x: Math.random() * canvas.width,
          y: Math.random() * canvas.height,
          radius: Math.random() * 2 + 1,
          speed: Math.random() * 50 + 30
        });
      }
    }
    createStarPool(300); // Try 1000 for stress test

    // Offscreen canvas for star glow
    const starCanvas = document.createElement('canvas');
    starCanvas.width = 4;
```

```
  starCanvas.height = 4;
  const starCtx = starCanvas.getContext('2d');
  starCtx.fillStyle = 'white';
  starCtx.beginPath();
  starCtx.arc(2, 2, 2, 0, Math.PI * 2);
  starCtx.fill();

  // Update logic with delta time
  function update(dt) {
    for (const star of stars) {
      star.y += star.speed * dt;
      if (star.y > canvas.height) {
        star.y = 0;
        star.x = Math.random() * canvas.width;
      }
    }
  }

  // Render stars
  function render() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    for (const star of stars) {
      ctx.globalAlpha = 0.8;
      ctx.drawImage(starCanvas, star.x, star.y, star.radius * 2, star.radius * 2);
    }
    ctx.globalAlpha = 1.0;
  }

  // FPS tracking
  let fps = 0, frames = 0;
  let lastTime = performance.now();
  let lastFpsTime = lastTime;
  const fpsDisplay = document.getElementById('fps');

  function trackFPS(now) {
    frames++;
    if (now - lastFpsTime >= 1000) {
      fps = frames;
      frames = 0;
      lastFpsTime = now;
      fpsDisplay.textContent = `FPS: ${fps}`;
    }
  }

  // Animation loop
  function animate(now) {
    const dt = (now - lastTime) / 1000;
    lastTime = now;

    update(dt);
    render();
    trackFPS(now);

    requestAnimationFrame(animate);
  }

  requestAnimationFrame(animate);
</script>
```

```
</body>
</html>
```

13.4.10 Summary: What You Learned

Technique	Benefit
`requestAnimationFrame()` + `deltaTime`	Smooth, consistent animations
Object pooling	Avoids GC and repeated allocations
Offscreen canvas	Faster rendering for repeated elements
FPS meter	Performance feedback during runtime

13.4.11 Challenge for the Reader

Try extending this animation by:

- Adding **star layers** at different speeds for parallax effect.
- Introducing **user control**: accelerate scroll with key press.
- Switching to **tile-based backgrounds** for side-scrollers.

13.4.12 Conclusion

High-performance animations depend not just on what you draw, but **how efficiently you manage rendering and memory**. With pooling, offscreen buffers, and smart timing, you can scale your canvas projects to thousands of animated elements without breaking a sweat.

Next up: Advanced rendering techniques and visual polish using shaders, filters, and more!

Chapter 14.

Advanced Projects and Integrations

1. Building a Paint Application with Undo/Redo
2. Integrating Canvas with Other Web APIs (Audio, Video)
3. Exporting Canvas to Images and PDFs
4. Practical Example: Interactive Game UI with Canvas

14 Advanced Projects and Integrations

14.1 Building a Paint Application with Undo/Redo

In this section, we'll walk through building a **paint-style drawing application** using the `<canvas>` element. We'll support **freehand drawing with the mouse**, add **undo/redo** functionality, and emphasize a **modular design** that separates concerns such as input, rendering, and state management.

14.1.1 Goal: A Minimal Drawing Tool with Undo/Redo

Features to include:

- Mouse-based freehand drawing
- Line width and color control
- Undo and redo with `getImageData()` snapshots
- Clear canvas option

14.1.2 Step 1: Basic HTML Canvas Setup

```
<canvas id="canvas" width="800" height="600" style="border:1px solid #ccc;"></canvas>
<div>
  <button onclick="undo()">Undo</button>
  <button onclick="redo()">Redo</button>
  <button onclick="clearCanvas()">Clear</button>
  <input type="color" id="colorPicker" value="#000000">
  <input type="range" id="lineWidth" min="1" max="20" value="2">
</div>
```

14.1.3 Step 2: Drawing Logic

Track mouse position and draw lines between points while the user is dragging.

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

let isDrawing = false;
let lastX = 0, lastY = 0;

canvas.addEventListener('mousedown', (e) => {
  isDrawing = true;
```

```
  [lastX, lastY] = getMousePos(e);
  saveState(); // Save before drawing
});

canvas.addEventListener('mousemove', (e) => {
  if (!isDrawing) return;
  const [x, y] = getMousePos(e);
  ctx.strokeStyle = document.getElementById('colorPicker').value;
  ctx.lineWidth = document.getElementById('lineWidth').value;
  ctx.lineCap = 'round';
  ctx.beginPath();
  ctx.moveTo(lastX, lastY);
  ctx.lineTo(x, y);
  ctx.stroke();
  [lastX, lastY] = [x, y];
});

canvas.addEventListener('mouseup', () => isDrawing = false);
canvas.addEventListener('mouseleave', () => isDrawing = false);

function getMousePos(e) {
  const rect = canvas.getBoundingClientRect();
  return [e.clientX - rect.left, e.clientY - rect.top];
}
```

14.1.4 Step 3: Undo/Redo with Image Snapshots

Use stacks to store canvas states as `ImageData` or `toDataURL()`.

```
const undoStack = [];
const redoStack = [];
const MAX_HISTORY = 20;

function saveState() {
  if (undoStack.length >= MAX_HISTORY) undoStack.shift();
  undoStack.push(ctx.getImageData(0, 0, canvas.width, canvas.height));
  redoStack.length = 0; // clear redo stack
}

function undo() {
  if (undoStack.length === 0) return;
  redoStack.push(ctx.getImageData(0, 0, canvas.width, canvas.height));
  const previous = undoStack.pop();
  ctx.putImageData(previous, 0, 0);
}

function redo() {
  if (redoStack.length === 0) return;
  undoStack.push(ctx.getImageData(0, 0, canvas.width, canvas.height));
  const next = redoStack.pop();
  ctx.putImageData(next, 0, 0);
}
```

14.1.5 Step 4: Clear the Canvas

```
function clearCanvas() {
  saveState();
  ctx.clearRect(0, 0, canvas.width, canvas.height);
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Canvas Drawing with Undo/Redo</title>
  <style>
    body { font-family: sans-serif; padding: 20px; }
    canvas { display: block; border: 1px solid #ccc; margin-bottom: 10px; }
    button, input[type="color"], input[type="range"] { margin-right: 8px; }
  </style>
</head>
<body>

  <!-- Canvas Element -->
  <canvas id="canvas" width="800" height="600"></canvas>

  <!-- Controls -->
  <div>
    <button onclick="undo()">Undo</button>
    <button onclick="redo()">Redo</button>
    <button onclick="clearCanvas()">Clear</button>
    <input type="color" id="colorPicker" value="#000000">
    <input type="range" id="lineWidth" min="1" max="20" value="2">
  </div>

  <!-- JavaScript Logic -->
  <script>
    const canvas = document.getElementById('canvas');
    const ctx = canvas.getContext('2d');

    let isDrawing = false;
    let lastX = 0, lastY = 0;

    const undoStack = [];
    const redoStack = [];
    const MAX_HISTORY = 20;

    // Helpers
    function getMousePos(e) {
      const rect = canvas.getBoundingClientRect();
      return [e.clientX - rect.left, e.clientY - rect.top];
    }

    function saveState() {
      if (undoStack.length >= MAX_HISTORY) undoStack.shift();
      undoStack.push(ctx.getImageData(0, 0, canvas.width, canvas.height));
      redoStack.length = 0;
    }
```

```
    function undo() {
      if (undoStack.length === 0) return;
      redoStack.push(ctx.getImageData(0, 0, canvas.width, canvas.height));
      const prev = undoStack.pop();
      ctx.putImageData(prev, 0, 0);
    }

    function redo() {
      if (redoStack.length === 0) return;
      undoStack.push(ctx.getImageData(0, 0, canvas.width, canvas.height));
      const next = redoStack.pop();
      ctx.putImageData(next, 0, 0);
    }

    function clearCanvas() {
      saveState();
      ctx.clearRect(0, 0, canvas.width, canvas.height);
    }

    // Drawing Events
    canvas.addEventListener('mousedown', (e) => {
      isDrawing = true;
      [lastX, lastY] = getMousePos(e);
      saveState();
    });

    canvas.addEventListener('mousemove', (e) => {
      if (!isDrawing) return;
      const [x, y] = getMousePos(e);
      ctx.strokeStyle = document.getElementById('colorPicker').value;
      ctx.lineWidth = document.getElementById('lineWidth').value;
      ctx.lineCap = 'round';

      ctx.beginPath();
      ctx.moveTo(lastX, lastY);
      ctx.lineTo(x, y);
      ctx.stroke();
      [lastX, lastY] = [x, y];
    });

    canvas.addEventListener('mouseup', () => isDrawing = false);
    canvas.addEventListener('mouseleave', () => isDrawing = false);
  </script>
</body>
</html>
```

14.1.6 Performance and Input Considerations

- **ImageData vs toDataURL**: `getImageData()` is faster and lighter than storing strings from `toDataURL()`. However, it doesn't persist across page reloads.
- **History depth**: Limit history depth to save memory. You can store fewer states or compress them periodically.

- **Touch support**: Add `touchstart`, `touchmove`, and `touchend` events to support drawing on mobile or tablets.

14.1.7 Suggested Modular Design

Structure your code in a scalable way:

```
- draw.js      → Handles input and rendering
- history.js   → Manages undo/redo stacks
- ui.js        → UI buttons and input controls
- main.js      → Initializes everything
```

This separation makes it easier to add new tools (like shapes, fill, or text) without rewriting core logic.

14.1.8 Enhancement Ideas

Encourage readers to explore:

- Save/load drawings using `toDataURL()`
- Add keyboard shortcuts for undo/redo
- Introduce shape tools: lines, rectangles, ellipses
- Add a mini preview/history timeline
- Implement real-time collaboration with WebSockets

14.1.9 Summary

You now have a functional, lightweight paint application built entirely with canvas and JavaScript. You've practiced:

- Managing mouse input
- Drawing smooth lines
- Using the canvas as a state machine with undo/redo
- Thinking in terms of modular, reusable components

This project also lays the groundwork for more complex editors, such as vector tools or animation timelines.

14.2 Integrating Canvas with Other Web APIs (Audio, Video)

Canvas is a powerful visual rendering tool—but its full potential shines when integrated with other web APIs, particularly **audio** and **video**. This section demonstrates how to combine `<canvas>` with APIs like `AudioContext`, `getUserMedia`, and `<video>` to build interactive, multimedia experiences such as **audio visualizations** and **live video effects**.

14.2.1 Drawing Audio Visualizations with `AudioContext`

Using the **Web Audio API**, you can capture live audio (microphone or media), analyze its frequency or waveform data, and visualize it in real time on canvas.

Basic Audio Analyzer Example

```
const canvas = document.getElementById("audioCanvas");
const ctx = canvas.getContext("2d");

const audioCtx = new AudioContext();
const analyser = audioCtx.createAnalyser();
analyser.fftSize = 256;

navigator.mediaDevices.getUserMedia({ audio: true }).then(stream => {
  const source = audioCtx.createMediaStreamSource(stream);
  source.connect(analyser);
  draw();
});

function draw() {
  requestAnimationFrame(draw);
  const dataArray = new Uint8Array(analyser.frequencyBinCount);
  analyser.getByteFrequencyData(dataArray);

  ctx.clearRect(0, 0, canvas.width, canvas.height);

  const barWidth = canvas.width / dataArray.length;
  dataArray.forEach((value, i) => {
    ctx.fillStyle = `rgb(${value + 50}, 50, 200)`;
    ctx.fillRect(i * barWidth, canvas.height - value, barWidth - 2, value);
  });
}
```

YES *This visualizes microphone input as animated bars.* You can switch to `getByteTimeDomainData()` for waveform style.

14.2.2 Using `video` with Canvas and `getUserMedia()`

Canvas can also render **live video streams**, either from local files or the user's webcam. This is especially useful for creating **real-time video filters**, **frame analysis**, or **screenshot tools**.

Drawing Webcam Feed to Canvas

```
<video id="video" autoplay muted playsinline></video>
<canvas id="videoCanvas"></canvas>
```

```
const video = document.getElementById('video');
const canvas = document.getElementById('videoCanvas');
const ctx = canvas.getContext('2d');

navigator.mediaDevices.getUserMedia({ video: true }).then(stream => {
  video.srcObject = stream;
});

function renderFrame() {
  ctx.drawImage(video, 0, 0, canvas.width, canvas.height);
  requestAnimationFrame(renderFrame);
}
video.addEventListener('play', renderFrame);
```

This streams live video to a `<video>` element and mirrors each frame on the canvas.

14.2.3 Real-Time Video Filters (Inversion)

You can apply pixel-level filters on live video using `getImageData()` and `putImageData()`:

```
function applyInvert() {
  const frame = ctx.getImageData(0, 0, canvas.width, canvas.height);
  const data = frame.data;
  for (let i = 0; i < data.length; i += 4) {
    data[i] = 255 - data[i];     // R
    data[i+1] = 255 - data[i+1]; // G
    data[i+2] = 255 - data[i+2]; // B
  }
  ctx.putImageData(frame, 0, 0);
}

function renderFilteredFrame() {
  ctx.drawImage(video, 0, 0, canvas.width, canvas.height);
  applyInvert();  // or any custom filter
  requestAnimationFrame(renderFilteredFrame);
}
video.addEventListener('play', renderFilteredFrame);
```

You can swap `applyInvert()` with your own grayscale, sepia, or edge-detection filters.

14.2.4 Media Integration Tips

1. **Asynchronous Loading**: Media streams may take time to initialize. Always use `.then()` for `getUserMedia` and `canplay` or `play` events for `<video>` readiness.
2. **Cross-Origin Video**: When using external video sources (like `<video src="https://...">`), the canvas may be "tainted" unless CORS headers are enabled. This prevents pixel access (`getImageData()`), so prefer local media or use the `crossOrigin` attribute correctly.
3. **Performance**: Applying filters per frame can be CPU-intensive. Consider limiting frame rates or offloading to Web Workers/OffscreenCanvas where supported.

14.2.5 Experiment Ideas

- Create a **music visualizer** that reacts to sound from an audio player.
- Apply **green screen masking** by comparing pixel colors in live video.
- Combine audio and video by syncing visual effects with beat detection.
- Allow recording modified canvas output to video with `MediaRecorder`.

14.2.6 Summary

By integrating `<canvas>` with audio and video APIs, you can create dynamic, media-rich applications like:

- Real-time music visualizers
- Webcam-based AR effects
- Custom video players with drawing overlays
- Interactive art and installations

Mastering these integrations opens doors to **creative coding**, **media tools**, and even **educational visualizations**.

14.3 Exporting Canvas to Images and PDFs

Canvas is not only great for dynamic drawing but also for creating graphics that users can save or share. This section covers how to **export your canvas content** to image files (PNG, JPEG) and PDF documents, enabling users to download or print your artwork, charts, or other visualizations.

14.3.1 Exporting Canvas as Images

The two primary methods to export canvas content are:

- `canvas.toDataURL(type, quality)` — returns a Base64 encoded URL representing the image.
- `canvas.toBlob(callback, type, quality)` — asynchronously creates a Blob object, better for large files or memory efficiency.

Exporting as PNG (default)

```
const canvas = document.getElementById('myCanvas');
const dataURL = canvas.toDataURL(); // defaults to 'image/png'
console.log(dataURL); // Base64 PNG image string

// Trigger download
const link = document.createElement('a');
link.href = dataURL;
link.download = 'drawing.png';
link.click();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Canvas to PNG Download</title>
  <style>
    body { font-family: sans-serif; padding: 20px; }
    canvas { border: 1px solid #ccc; display: block; margin-bottom: 10px; }
  </style>
</head>
<body>

  <canvas id="myCanvas" width="500" height="300"></canvas>
  <button id="downloadBtn">Download as PNG</button>

  <script>
    const canvas = document.getElementById('myCanvas');
    const ctx = canvas.getContext('2d');
    const downloadBtn = document.getElementById('downloadBtn');

    // Simple drawing setup
    let isDrawing = false;
    let lastX = 0, lastY = 0;

    canvas.addEventListener('mousedown', (e) => {
      isDrawing = true;
      [lastX, lastY] = [e.offsetX, e.offsetY];
    });

    canvas.addEventListener('mousemove', (e) => {
      if (!isDrawing) return;
      ctx.strokeStyle = 'black';
      ctx.lineWidth = 2;
      ctx.lineCap = 'round';
```

```
      ctx.beginPath();
      ctx.moveTo(lastX, lastY);
      ctx.lineTo(e.offsetX, e.offsetY);
      ctx.stroke();
      [lastX, lastY] = [e.offsetX, e.offsetY];
    });

    canvas.addEventListener('mouseup', () => isDrawing = false);
    canvas.addEventListener('mouseleave', () => isDrawing = false);

    // Download logic
    downloadBtn.addEventListener('click', () => {
      const dataURL = canvas.toDataURL(); // image/png
      const link = document.createElement('a');
      link.href = dataURL;
      link.download = 'drawing.png';
      link.click();
    });
  </script>
</body>
</html>
```

Exporting as JPEG with quality setting

```
const jpegDataURL = canvas.toDataURL('image/jpeg', 0.8); // quality between 0 and 1

const downloadJPEG = document.createElement('a');
downloadJPEG.href = jpegDataURL;
downloadJPEG.download = 'photo.jpg';
downloadJPEG.click();
```

JPEG is useful for photographs or complex images where smaller file sizes are preferred over transparency.

Using `toBlob()` for Efficient Export

`toBlob()` is preferred when working with large canvases or when you want to avoid blocking the main thread.

```
canvas.toBlob(blob => {
  const url = URL.createObjectURL(blob);
  const link = document.createElement('a');
  link.href = url;
  link.download = 'export.png';
  link.click();

  // Release object URL after download
  URL.revokeObjectURL(url);
}, 'image/png');
```

14.3.2 Exporting Canvas Content to PDF

Exporting canvas drawings as PDFs is a great way to prepare printable reports, certificates, or complex documents. The popular `jsPDF` library simplifies this process.

Basic Example: Canvas to PDF with `jsPDF`

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jspdf/2.5.1/jspdf.umd.min.js"></script>
```

```
const { jsPDF } = window.jspdf;
const pdf = new jsPDF();

const canvas = document.getElementById('myCanvas');
const imgData = canvas.toDataURL('image/png');

pdf.addImage(imgData, 'PNG', 10, 10, 180, 160);
pdf.save('canvas-export.pdf');
```

- This example grabs the canvas as a PNG image and embeds it into a PDF page.
- You can control image positioning and size within the PDF.
- `jsPDF` also supports adding text, shapes, and multiple pages for complex documents.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Canvas to PDF Example</title>
  <style>
    body { font-family: sans-serif; padding: 20px; }
    canvas { border: 1px solid #ccc; display: block; margin-bottom: 10px; }
  </style>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jspdf/2.5.1/jspdf.umd.min.js"></script>
</head>
<body>

  <h2>Draw on the Canvas and Export to PDF</h2>
  <canvas id="myCanvas" width="400" height="300"></canvas>
  <button id="downloadPdfBtn">Download PDF</button>

  <script>
    // Drawing setup
    const canvas = document.getElementById('myCanvas');
    const ctx = canvas.getContext('2d');

    let isDrawing = false;
    let lastX = 0, lastY = 0;

    canvas.addEventListener('mousedown', e => {
      isDrawing = true;
      [lastX, lastY] = [e.offsetX, e.offsetY];
    });

    canvas.addEventListener('mousemove', e => {
      if (!isDrawing) return;
```

```
      ctx.strokeStyle = 'black';
      ctx.lineWidth = 2;
      ctx.lineCap = 'round';
      ctx.beginPath();
      ctx.moveTo(lastX, lastY);
      ctx.lineTo(e.offsetX, e.offsetY);
      ctx.stroke();
      [lastX, lastY] = [e.offsetX, e.offsetY];
    });

    canvas.addEventListener('mouseup', () => isDrawing = false);
    canvas.addEventListener('mouseleave', () => isDrawing = false);

    // PDF Export
    document.getElementById('downloadPdfBtn').addEventListener('click', () => {
      const { jsPDF } = window.jspdf;
      const pdf = new jsPDF();

      const imgData = canvas.toDataURL('image/png');
      pdf.addImage(imgData, 'PNG', 10, 10, 180, 135); // adjust size as needed
      pdf.save('canvas-export.pdf');
    });
  </script>

</body>
</html>
```

14.3.3 Example: Download Drawing with Button

Here's a quick example that lets users download their canvas drawing as PNG by clicking a button:

```
<canvas id="drawCanvas" width="400" height="300"></canvas>
<button id="downloadBtn">Download Image</button>

<script>
  const canvas = document.getElementById('drawCanvas');
  const ctx = canvas.getContext('2d');
  const btn = document.getElementById('downloadBtn');

  // Example drawing
  ctx.fillStyle = 'lightblue';
  ctx.fillRect(50, 50, 300, 200);
  ctx.fillStyle = 'darkblue';
  ctx.font = '30px Arial';
  ctx.fillText('Hello Canvas!', 100, 150);

  btn.addEventListener('click', () => {
    const dataURL = canvas.toDataURL('image/png');
    const link = document.createElement('a');
    link.href = dataURL;
    link.download = 'myCanvasImage.png';
    link.click();
```

```
  });
</script>
```

14.3.4 Summary

- `toDataURL()` is simple but memory-heavy for large canvases; ideal for quick image exports.
- `toBlob()` is more efficient for large or frequent exports.
- Exporting to **PDF** with libraries like `jsPDF` enables professional documents embedding canvas graphics.
- Adding download buttons enhances user experience for saving artwork or visualizations.

Experiment with different formats and qualities to optimize file size and visual fidelity according to your needs.

14.4 Practical Example: Interactive Game UI with Canvas

In this section, we'll build a mini game UI on canvas, illustrating how to draw interactive buttons, health bars, and score counters — and how to connect these visuals with game logic. This example integrates user input, simple animations, and sound effects to demonstrate a rich, responsive UI system on canvas.

14.4.1 Designing the Game UI

A typical game UI includes:

- **Buttons** — clickable areas to trigger actions.
- **Health Bar** — visual indicator of player health.
- **Score Counter** — numeric display updated dynamically.
- **Interactive Feedback** — animations and sound to enrich UX.

We'll break down how to draw these components on canvas and detect clicks with hit detection.

14.4.2 Step 1: Setup Canvas and Game State

```
<canvas id="gameUI" width="400" height="200" style="border:1px solid #ccc;"></canvas>
<script>
  const canvas = document.getElementById('gameUI');
  const ctx = canvas.getContext('2d');

  // Game state
  const state = {
    health: 75,       // out of 100
    score: 0,
    buttonClicked: false
  };
</script>
```

14.4.3 Step 2: Drawing UI Components

Health Bar

```
function drawHealthBar(ctx, x, y, width, height, health) {
  ctx.fillStyle = 'gray';
  ctx.fillRect(x, y, width, height);

  ctx.fillStyle = health > 50 ? 'green' : (health > 20 ? 'orange' : 'red');
  ctx.fillRect(x, y, (health / 100) * width, height);

  ctx.strokeStyle = 'black';
  ctx.strokeRect(x, y, width, height);
}
```

Score Counter

```
function drawScore(ctx, x, y, score) {
  ctx.fillStyle = 'black';
  ctx.font = '20px Arial';
  ctx.fillText(`Score: ${score}`, x, y);
}
```

Button with Hit Detection

```
const button = {
  x: 300,
  y: 140,
  width: 80,
  height: 40,
  label: 'Hit Me'
};
```

```
function drawButton(ctx, btn) {
  ctx.fillStyle = state.buttonClicked ? '#4CAF50' : '#2196F3';
  ctx.fillRect(btn.x, btn.y, btn.width, btn.height);

  ctx.strokeStyle = 'black';
  ctx.strokeRect(btn.x, btn.y, btn.width, btn.height);

  ctx.fillStyle = 'white';
  ctx.font = '18px Arial';
  ctx.textAlign = 'center';
  ctx.textBaseline = 'middle';
  ctx.fillText(btn.label, btn.x + btn.width / 2, btn.y + btn.height / 2);
}
```

14.4.4 Step 3: Hit Detection and Interactivity

Detect mouse clicks within the button's bounding box:

```
canvas.addEventListener('click', e => {
  const rect = canvas.getBoundingClientRect();
  const mouseX = e.clientX - rect.left;
  const mouseY = e.clientY - rect.top;

  if (
    mouseX >= button.x &&
    mouseX <= button.x + button.width &&
    mouseY >= button.y &&
    mouseY <= button.y + button.height
  ) {
    onButtonClick();
  }
});

function onButtonClick() {
  state.buttonClicked = true;
  state.score += 10;
  state.health = Math.max(0, state.health - 5);

  // Play a simple sound (if supported)
  playClickSound();

  // Reset button color after 200ms
  setTimeout(() => {
    state.buttonClicked = false;
    render();
  }, 200);

  render();
}
```

14.4.5 Step 4: Adding Sound Effects

Use the Web Audio API for a simple click sound:

```
function playClickSound() {
  const ctxAudio = new (window.AudioContext || window.webkitAudioContext)();
  const oscillator = ctxAudio.createOscillator();
  const gainNode = ctxAudio.createGain();

  oscillator.type = 'square';
  oscillator.frequency.setValueAtTime(440, ctxAudio.currentTime);
  gainNode.gain.setValueAtTime(0.1, ctxAudio.currentTime);

  oscillator.connect(gainNode);
  gainNode.connect(ctxAudio.destination);

  oscillator.start();
  oscillator.stop(ctxAudio.currentTime + 0.1);
}
```

14.4.6 Step 5: Rendering Loop

```
function render() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  drawHealthBar(ctx, 20, 150, 200, 20, state.health);
  drawScore(ctx, 20, 30, state.score);
  drawButton(ctx, button);
}

// Initial render
render();
```

14.4.7 Step 6: Adding Animation (Optional)

Add a subtle pulse animation to the button when clicked:

```
let pulse = 0;
function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Pulse effect on button background brightness
  const brightness = state.buttonClicked ? 1 + Math.sin(pulse) * 0.3 : 1;
  ctx.filter = `brightness(${brightness})`;

  drawHealthBar(ctx, 20, 150, 200, 20, state.health);
  drawScore(ctx, 20, 30, state.score);
```

```
  drawButton(ctx, button);

  ctx.filter = 'none';

  pulse += 0.1;
  requestAnimationFrame(animate);
}

// Start animation
animate();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Game UI Example</title>
  <style>
    body {
      background: #eee;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      margin: 0;
      font-family: sans-serif;
    }
  </style>
</head>
<body>
  <canvas id="gameUI" width="400" height="200" style="border:1px solid #ccc;"></canvas>

  <script>
    const canvas = document.getElementById('gameUI');
    const ctx = canvas.getContext('2d');

    // Game state
    const state = {
      health: 75,       // out of 100
      score: 0,
      buttonClicked: false
    };

    // Button definition
    const button = {
      x: 300,
      y: 140,
      width: 80,
      height: 40,
      label: 'Hit Me'
    };

    // Draw health bar
    function drawHealthBar(ctx, x, y, width, height, health) {
      ctx.fillStyle = 'gray';
      ctx.fillRect(x, y, width, height);

      ctx.fillStyle = health > 50 ? 'green' : (health > 20 ? 'orange' : 'red');
```

```
    ctx.fillRect(x, y, (health / 100) * width, height);

    ctx.strokeStyle = 'black';
    ctx.strokeRect(x, y, width, height);
  }

  // Draw score
  function drawScore(ctx, x, y, score) {
    ctx.fillStyle = 'black';
    ctx.font = '20px Arial';
    ctx.textAlign = 'left';
    ctx.fillText(`Score: ${score}`, x, y);
  }

  // Draw button
  function drawButton(ctx, btn) {
    ctx.fillStyle = state.buttonClicked ? '#4CAF50' : '#2196F3';
    ctx.fillRect(btn.x, btn.y, btn.width, btn.height);

    ctx.strokeStyle = 'black';
    ctx.strokeRect(btn.x, btn.y, btn.width, btn.height);

    ctx.fillStyle = 'white';
    ctx.font = '18px Arial';
    ctx.textAlign = 'center';
    ctx.textBaseline = 'middle';
    ctx.fillText(btn.label, btn.x + btn.width / 2, btn.y + btn.height / 2);
  }

  // Handle button click
  canvas.addEventListener('click', e => {
    const rect = canvas.getBoundingClientRect();
    const mouseX = e.clientX - rect.left;
    const mouseY = e.clientY - rect.top;

    if (
      mouseX >= button.x &&
      mouseX <= button.x + button.width &&
      mouseY >= button.y &&
      mouseY <= button.y + button.height
    ) {
      onButtonClick();
    }
  });

  function onButtonClick() {
    state.buttonClicked = true;
    state.score += 10;
    state.health = Math.max(0, state.health - 5);
    playClickSound();

    setTimeout(() => {
      state.buttonClicked = false;
    }, 200);
  }

  // Simple Web Audio click
  function playClickSound() {
```

```
      const ctxAudio = new (window.AudioContext || window.webkitAudioContext)();
      const oscillator = ctxAudio.createOscillator();
      const gainNode = ctxAudio.createGain();

      oscillator.type = 'square';
      oscillator.frequency.setValueAtTime(440, ctxAudio.currentTime);
      gainNode.gain.setValueAtTime(0.1, ctxAudio.currentTime);

      oscillator.connect(gainNode);
      gainNode.connect(ctxAudio.destination);

      oscillator.start();
      oscillator.stop(ctxAudio.currentTime + 0.1);
    }

    // Animation loop
    let pulse = 0;
    function animate() {
      ctx.clearRect(0, 0, canvas.width, canvas.height);

      const brightness = state.buttonClicked ? 1 + Math.sin(pulse) * 0.3 : 1;
      ctx.filter = `brightness(${brightness})`;

      drawHealthBar(ctx, 20, 150, 200, 20, state.health);
      drawScore(ctx, 20, 30, state.score);
      drawButton(ctx, button);

      ctx.filter = 'none';

      pulse += 0.1;
      requestAnimationFrame(animate);
    }

    animate();
  </script>
</body>
</html>
```

14.4.8 Summary and Encouragement

This example shows how to:

- Draw and manage multiple UI elements on canvas.
- Implement hit detection for interactivity.
- Blend game state logic with visuals dynamically.
- Add audio feedback and simple animations for engagement.

You can expand this by creating reusable UI component classes or modules, adding more controls like sliders or toggles, and integrating more advanced animations or sound effects.

Experiment with customizing shapes, colors, and interactions to design your own compelling game interfaces entirely on canvas!

Chapter 15.

Debugging and Testing Canvas Applications

1. Using Browser DevTools for Canvas Debugging
2. Unit Testing Canvas Drawing Logic
3. Performance Profiling and Memory Leak Detection
4. Practical Example: Debugging a Complex Animation

15 Debugging and Testing Canvas Applications

15.1 Using Browser DevTools for Canvas Debugging

Debugging canvas applications can be challenging because the canvas itself is a bitmap rendering surface — it doesn't retain a DOM structure to inspect. Fortunately, modern browsers provide powerful developer tools specifically designed to help debug canvas and related JavaScript code. In this section, we'll explore how to use Chrome and Firefox Developer Tools effectively to inspect, profile, and debug your canvas projects.

15.1.1 Inspecting the Canvas Element and 2D Context

1. **Inspect the Canvas Element** Open Developer Tools (`F12` or `Ctrl+Shift+I` / `Cmd+Option+I`), then select the **Elements** panel to find your `<canvas>` element in the DOM. You can verify size, CSS styles, and other properties here.
2. **Access the Canvas Context in Console** You can grab the 2D rendering context directly from the console for live testing and debugging. For example:

   ```
   const canvas = document.querySelector('canvas');
   const ctx = canvas.getContext('2d');
   ```

   This lets you run drawing commands interactively and experiment with code snippets without refreshing your page.

15.1.2 Setting Breakpoints and Debugging Drawing Logic

- Use the **Sources** panel to set breakpoints in your JavaScript code related to canvas drawing.
- Step through drawing functions to see how state and variables change frame by frame.
- Inspect variable values like coordinates, colors, or transformation matrices during the drawing cycle.

This approach helps catch logic errors or unexpected parameter values causing rendering issues.

15.1.3 Visual Debugging: Logging and Overlays

Sometimes you want to visualize what's happening inside the canvas during user interaction or animation. Here are some practical techniques:

- **Draw Debug Bounding Boxes**

  Wrap your drawing functions with code that overlays outlines or hit areas:

```
ctx.strokeStyle = 'red';
ctx.lineWidth = 2;
ctx.strokeRect(x, y, width, height);  // Highlight the interactive region
```

- **Trace Mouse Positions**

  Listen to mouse events and draw markers or logs directly:

```
canvas.addEventListener('mousemove', e => {
  const rect = canvas.getBoundingClientRect();
  const mouseX = e.clientX - rect.left;
  const mouseY = e.clientY - rect.top;

  ctx.clearRect(0, 0, canvas.width, canvas.height);
  drawGameElements();  // Your existing drawing logic

  ctx.fillStyle = 'blue';
  ctx.beginPath();
  ctx.arc(mouseX, mouseY, 5, 0, Math.PI * 2);
  ctx.fill();

  console.log(`Mouse at (${mouseX}, ${mouseY})`);
});
```

- **Intercept Canvas API Calls**

  For deep debugging, you can override canvas methods temporarily to log calls and parameters:

```
const originalFillRect = ctx.fillRect;
ctx.fillRect = function(x, y, w, h) {
  console.log('fillRect called with:', x, y, w, h);
  originalFillRect.call(this, x, y, w, h);
};
```

15.1.4 Performance Profiling Tools

- Use the **Performance** panel (Chrome) or **Performance** tab (Firefox) to record and analyze canvas rendering times and CPU usage.
- Look for heavy draw calls, excessive repaints, or large memory consumption.
- Frame-by-frame inspection helps optimize redraws and improve animation smoothness.

15.1.5 Useful Extensions and Browser Flags

- **Canvas Inspector Extensions** Chrome and Firefox have extensions like *Canvas Inspector* that provide enhanced capabilities for visualizing draw calls and frame buffers.

- **Browser Flags** Some browsers support experimental flags to expose additional canvas debugging info (check your browser's `about:flags` or `chrome://flags`).

15.1.6 Summary

Mastering canvas debugging involves a mix of:

- Using Developer Tools to inspect canvas elements and JS code.
- Visual overlays and console logging for interactive insights.
- Profiling to identify bottlenecks.
- Experimenting with API interception for deep dives.

These techniques will empower you to diagnose and fix tricky canvas bugs efficiently — a critical skill for any canvas programmer.

15.2 Unit Testing Canvas Drawing Logic

Testing canvas-based applications can feel tricky because the canvas is inherently visual—outputting pixels rather than data structures. However, by organizing your code smartly and applying testing strategies, you can ensure your drawing logic is robust and bug-free.

15.2.1 Separating Drawing Logic from Rendering

The key to effective testing is **separation of concerns**:

- **Pure Logic Functions:** Isolate calculations, geometry, animation state updates, and event handling into pure functions or classes that don't directly call canvas APIs.
- **Rendering Functions:** Keep the actual `canvas` context drawing calls in separate modules or methods.

For example, instead of mixing your shape coordinates and rendering code, you might:

```
// Pure function returning points for a star shape
function createStarPoints(cx, cy, spikes, outerRadius, innerRadius) {
  // Calculate and return an array of points without drawing
  // ...
  return pointsArray;
}

// Rendering function that takes points and draws on canvas
function drawStar(ctx, points) {
  ctx.beginPath();
```

```
  ctx.moveTo(points[0].x, points[0].y);
  // ...
  ctx.stroke();
}
```

You can then **unit test** `createStarPoints()` easily without needing a canvas.

15.2.2 Testing Shape Creation and Animation Logic with Jest

Using a test framework like Jest, you can:

- Write tests for shape geometry, coordinate calculations, and animation state changes.
- Mock or stub inputs and verify outputs.

Example test for star points:

```
test('createStarPoints returns correct number of points', () => {
  const points = createStarPoints(0, 0, 5, 50, 25);
  expect(points.length).toBe(10);  // 5 spikes + 5 inner points
});
```

For animations, test that position updates based on velocity and delta time behave correctly:

```
test('updatePosition moves object correctly', () => {
  const obj = { x: 0, y: 0, vx: 10, vy: 5 };
  const dt = 0.1;  // 100ms frame time
  updatePosition(obj, dt);
  expect(obj.x).toBeCloseTo(1);
  expect(obj.y).toBeCloseTo(0.5);
});
```

15.2.3 Input Handling Tests

You can simulate mouse or keyboard events programmatically to test input handlers that affect drawing or animation state. Jest along with tools like jsdom can simulate DOM environments for such tests.

Example:

```
test('mouseDown sets drawing flag', () => {
  const state = { isDrawing: false };
  const event = new MouseEvent('mousedown');
  handleMouseDown(event, state);
  expect(state.isDrawing).toBe(true);
});
```

15.2.4 Snapshot Testing and Visual Regression

Since canvas output is pixel-based, **visual regression testing** can catch unintended changes. Two common approaches:

1. **Snapshot Testing with `toDataURL()`** Render the canvas in a headless environment or test browser, call `canvas.toDataURL()` to get a base64 PNG string, and compare it to stored snapshots.
2. **Pixel-by-Pixel Comparison** Use libraries like `pixelmatch` to compare rendered images with baseline images, highlighting differences.

These methods help detect rendering bugs or subtle visual changes, especially in animations or complex shapes.

15.2.5 Best Practices

- Keep logic testable and pure where possible.
- Mock canvas context in unit tests if testing rendering calls.
- Automate visual regression in your CI/CD pipeline for stability.
- Use descriptive tests covering geometry, animation, and input handling.

By applying these strategies, you'll build confidence that your canvas drawing logic works correctly — even before visually inspecting the rendered output.

15.3 Performance Profiling and Memory Leak Detection

When building canvas applications—especially those with animations and complex rendering—keeping performance smooth and memory usage stable is crucial. Browser developer tools offer powerful ways to profile your canvas code, find bottlenecks, and detect memory leaks.

15.3.1 Profiling Canvas Performance with DevTools

Most modern browsers like Chrome and Firefox include detailed profiling tools. Here's how to leverage them for canvas projects:

- **Open the Performance Tab:** Record a session while your animation or interaction runs. The timeline shows frame rates, scripting time, rendering times, and paint events.
- **Inspect Frame Rendering Time:** Look for frames that take longer than 16ms (for 60fps). These cause visible stutters.

- **Analyze Call Stacks:** Expand expensive functions to identify slow drawing calls, inefficient loops, or excessive computations.
- **Memory Snapshot:** Use the Memory tab to take heap snapshots, check for detached DOM nodes or growing JavaScript objects, and track down leaks.

15.3.2 Common Sources of Performance Issues and Memory Leaks

- **Unstopped Animation Loops:** Forgetting to call `cancelAnimationFrame()` leads to animations running in the background, eating CPU and memory. Always cancel loops when they're no longer needed (e.g., on page hide or stop).
- **Growing Arrays or Objects:** Particle systems or animations that append new objects without cleanup cause memory growth. Implement pooling or regularly prune unused items.
- **Excessive Image Loading:** Loading images repeatedly instead of caching can spike memory and cause jank. Load once and reuse image objects.
- **Detached DOM Elements:** Creating canvases or overlays but never removing them from the DOM causes leaks. Clean up elements when no longer in use.

15.3.3 Monitoring Canvas Memory in Chrome

- **Performance Tab + Memory:** Capture a performance profile and switch to the Memory tab to watch canvas-related memory usage over time.
- **Lighthouse Audits:** Run Lighthouse in Chrome DevTools for automated performance and memory recommendations. It can highlight issues like large paint areas or inefficient rendering.
- **Canvas-Specific Debugging:** Use Chrome's "Rendering" tab (under More Tools) to enable paint flashing, helping you see when the canvas is redrawn unnecessarily.

15.3.4 Practical Tips to Avoid Performance Pitfalls

```
// Always cancel animation frames when done
let animationId;

function animate() {
  animationId = requestAnimationFrame(animate);
  // Drawing code...
}

// Start animation
```

```
animate();

// Stop animation (e.g., on page unload or stop)
cancelAnimationFrame(animationId);
```

- **Reuse objects instead of creating new ones every frame.**
- **Limit redraw areas using dirty rectangles to minimize painting cost.**
- **Cache complex drawings to offscreen canvases.**
- **Throttle input events that trigger redraws.**

15.3.5 Summary

Using browser dev tools to profile and monitor your canvas application helps you maintain smooth animations and avoid slowdowns or crashes. By identifying memory leaks early and optimizing draw calls, your canvas projects will remain performant and stable.

15.4 Practical Example: Debugging a Complex Animation

Building complex animations like bouncing ball simulations with trails and collision logic often reveals subtle bugs—such as unexpected frame rate drops, incorrect physics responses, or memory leaks—that can be tricky to diagnose. In this section, we will explore a sample animation with intentional flaws and demonstrate how to methodically debug and optimize it.

15.4.1 The Scenario: Bouncing Ball with Trail and Collision

Imagine a canvas animation where a ball bounces within boundaries and leaves a fading trail behind. The simulation includes:

- Gravity and friction effects
- Collision detection with edges
- A trail effect using semi-transparent drawing
- Velocity vector visualization for debugging

Despite working "on the surface," the animation suffers from:

- Frame rate drops after running for a while
- Bounces that don't reflect proper physics angles
- Increasing memory consumption over time

15.4.2 Step 1: Visual Inspection and Logging

Start by visually inspecting the animation:

- **Look for stutters or lag:** Does the animation slow down after several seconds?
- **Observe ball behavior:** Are bounce angles consistent or does the ball appear to "stick" or move strangely near edges?
- **Watch the trail:** Does the trail grow indefinitely or fade properly?

Add simple logging to output position and velocity each frame:

```
console.log(`Position: (${ball.x.toFixed(2)}, ${ball.y.toFixed(2)}), Velocity: (${ball.vx.toFixed(2)}, $
```

This helps detect sudden spikes or values that violate expected ranges.

15.4.3 Step 2: Drawing Debug Visuals

Add visual cues directly on the canvas to understand physics and rendering:

- **Velocity Vector:** Draw an arrow representing velocity from the ball's center. This shows direction and magnitude, helping verify bounce logic.

```
ctx.beginPath();
ctx.moveTo(ball.x, ball.y);
ctx.lineTo(ball.x + ball.vx * 10, ball.y + ball.vy * 10);
ctx.strokeStyle = 'red';
ctx.stroke();
```

- **Bounding Box:** Draw rectangle bounds or collision zones.
- **Trail Management:** Visualize alpha fading for the trail.

These overlays help identify discrepancies between expected and actual behavior.

15.4.4 Step 3: Profiling Performance and Memory

Use Chrome DevTools or Firefox Developer Tools:

- **Performance Tab:** Record a session and check if frame rendering times increase over time. Spikes indicate expensive operations or memory pressure.
- **Memory Tab:** Take heap snapshots at intervals and watch for growing objects related to the animation (e.g., particle arrays or retained canvas bitmaps).
- **Paint Flashing:** Enable paint flashing to check if unnecessary full-canvas redraws occur every frame.

15.4.5 Step 4: Identifying Common Issues

From the above observations, you might find:

- **Frame rate drops due to trail accumulation:** If the trail is implemented by drawing semi-transparent rectangles repeatedly without clearing or capping opacity, it leads to an increasingly "dirty" canvas.
- **Incorrect bounce angles from logic errors:** The velocity inversion might only negate one component without considering direction, or collisions might trigger multiple times per frame.
- **Memory leak from forgotten references:** New objects created each frame (e.g., vectors, snapshots) without cleanup cause gradual memory growth.

15.4.6 Step 5: Fixing Bugs and Optimizing

- **Trail Optimization:** Instead of drawing a persistent semi-transparent overlay every frame, limit trail length by storing previous positions in a fixed-size array and clearing old points. Alternatively, clear the canvas each frame and redraw trails with decreasing opacity.
- **Collision Correction:** Adjust the collision response to invert velocity components properly and ensure velocity is only updated once per collision. Use flags or timestamp checks if needed.
- **Memory Management:** Reuse objects via pooling instead of instantiating new ones every frame. Cancel any unused animation loops and avoid holding references longer than needed.

15.4.7 Step 6: Enhancing Debugging and Testing

- **Add Conditional Breakpoints:** Break when velocity exceeds a threshold or when collisions occur.
- **Unit Test Core Logic:** Extract bounce calculation and physics into pure functions and test with Jest or other frameworks.
- **Performance Benchmarks:** Use automated tests to measure frame rates and memory before and after fixes.

15.4.8 Summary

Debugging complex canvas animations requires a combination of visual cues, logging, and profiling tools to identify subtle issues. By systematically isolating frame rate problems, collision logic bugs, and memory leaks, you can significantly improve the robustness and responsiveness of your animations. Incorporating test coverage and performance monitoring ensures your canvas projects remain maintainable and performant as they grow in complexity.