Canvas API and SVG Complete Guide

Table of Contents

Canvas API Overview

What is Canvas?

The **HTML5 Canvas API** provides a means for drawing graphics via JavaScript. It's a bitmap-based drawing surface that allows you to create dynamic, scriptable rendering of 2D shapes, images, and animations.

Key Characteristics

Canvas is Raster-Based (Bitmap):

- Draws pixel by pixel
- Once drawn, shapes become pixels (no memory of individual objects)
- Resolution-dependent (pixelates when scaled)
- Better performance for complex scenes with many objects

When to Use Canvas:

- Games and animations
- Real-time data visualization
- Image manipulation
- Particle effects

- Complex visual effects
- Performance-critical graphics (thousands of objects)

Canvas Limitations:

- No DOM elements (can't attach event listeners to individual shapes)
- Not accessible to screen readers
- Requires manual redrawing for updates
- Pixelates when scaled up

Canvas Setup and Context

Creating a Canvas Element

HTML:

```
<canvas id="myCanvas" width="800" height="600"></canvas>
```

Important Notes:

- Canvas dimensions should be set via `width` and `height` attributes (not CSS)
- Default size is 300×150 pixels if not specified
- Setting size via CSS will stretch/distort the canvas

Why Width/Height Attributes Matter:

```
<!-- CORRECT - Using attributes -->
<canvas width="800" height="600"></canvas>

<!-- WRONG - Using CSS -->
<canvas style="width: 800px; height: 600px;"></canvas>
<!-- This stretches 300×150 to 800×600, causing distortion -->
```

Getting the Context

The context is the object that provides the drawing methods.

```
const canvas = document.querySelector("canvas");
const context = canvas.getContext("2d");
```

Available Context Types:

- `"2d"` : 2D graphics (most common)
- `"webgl"` : 3D graphics using WebGL
- `"webgl2"` : WebGL 2.0
- `"bitmaprenderer"` : Bitmap rendering

In Your Code:

```
// Select canvas element
const canvas = document.querySelector("canvas");

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

The context object (`context`) contains all the methods and properties for drawing on the canvas.

Setting Canvas Size Dynamically

```
const canvas = document.querySelector("canvas");
const context = canvas.getContext("2d");

// Set canvas to 80% of window width, 60% of window height
canvas.width = innerWidth * 0.8;
canvas.height = innerHeight * 0.6;

// Responsive - update on window resize
window.addEventListener('resize', function() {
    canvas.width = innerWidth * 0.8;
    canvas.height = innerHeight * 0.6;
    // Note: Resizing clears the canvas, need to redraw
});
```

Critical Note:
Changing `canvas.width` or `canvas.height` clears the entire canvas and resets all context properties (colors, line width, etc.).

Drawing Rectangles

Filled Rectangles

Method:

```
context.fillRect(x, y, width, height);
```

Parameters:

- `x` : X-coordinate of top-left corner
- `y` : Y-coordinate of top-left corner
- `width` : Rectangle width
- `height` : Rectangle height

Example from Your Code:

```
// Draw black rectangle (default color)
context.fillRect(50, 100, 100, 60);

// Change fill color to blue
context.fillStyle = "blue";

// Draw blue rectangle
context.fillRect(160, 50, 50, 50);

// Change fill color to green
context.fillStyle = "green";

// Draw green rectangle
context.fillRect(300, 100, 80, 70);
```

Understanding fillStyle:

```
// Color order matters!
context.fillStyle = "blue";  // Set color first
context.fillRect(50, 50, 100, 100);  // Then draw (uses blue)

context.fillRect(200, 50, 100, 100);  // Still blue
context.fillStyle = "red";  // Change color
context.fillRect(350, 50, 100, 100);  // Now red
```

fillStyle Property:
The `fillStyle` property sets the color/pattern for filled shapes.

Accepted Values:

```
// Named colors
context.fillStyle = "blue";
context.fillStyle = "red";

// Hex colors
context.fillStyle = "#FF5733";
context.fillStyle = "#3498db";

// RGB
context.fillStyle = "rgb(255, 0, 0)";

// RGBA (with transparency)
context.fillStyle = "rgba(255, 0, 0, 0.5)";  // 50% transparent red

// HSL
context.fillStyle = "hsl(120, 100%, 50%)";

// Gradients (covered later)
const gradient = context.createLinearGradient(0, 0, 200, 0);
context.fillStyle = gradient;

// Patterns (covered later)
const pattern = context.createPattern(image, 'repeat');
context.fillStyle = pattern;
```

Stroked Rectangles (Outlined)

Method:

```
context.strokeRect(x, y, width, height);
```

Example from Your Code:

```
// Set stroke color to red
context.strokeStyle = "red";

// Set line width to 3 pixels
context.lineWidth = 3;

// Draw outlined rectangle
context.strokeRect(430, 50, 100, 60);
```

strokeStyle Property:
Similar to `fillStyle`, but for outlines/strokes.

```
context.strokeStyle = "red";
context.lineWidth = 5;
context.strokeRect(50, 50, 100, 100);
```

lineWidth Property:
Controls the thickness of lines and strokes.

```
context.lineWidth = 1;   // Thin line
context.lineWidth = 5;   // Medium line
context.lineWidth = 10;  // Thick line
```

Combining Fill and Stroke

Example from Your Code:

```
// Draw filled rectangle
context.fillRect(100, 230, 150, 80);

// Draw outline on same rectangle
context.strokeStyle = "yellow";
context.strokeRect(100, 230, 150, 80);
```

This creates a rectangle with both fill (default black) and yellow outline.

Clearing Rectangles

Method:

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

Purpose: Erases a rectangular area, making it transparent.

Example from Your Code:

```
// Draw filled rectangle
context.fillRect(100, 230, 150, 80);

// Draw outline
context.strokeStyle = "yellow";
context.strokeRect(100, 230, 150, 80);
```

```
// Clear a small square in the middle (creates a "hole")
context.clearRect(130, 260, 30, 30);
```

Common Uses:

```
// Clear entire canvas
context.clearRect(0, 0, canvas.width, canvas.height);

// Create eraser effect
canvas.addEventListener('mousemove', function(e) {
    const x = e.clientX - canvas.offsetLeft;
    const y = e.clientY - canvas.offsetTop;
    context.clearRect(x - 10, y - 10, 20, 20);  // Erase 20×20 area
});
```

Drawing Lines and Paths

Path Concept

Canvas uses **paths** to draw complex shapes. A path is a series of connected points.

Path Methods:

- `beginPath()` : Start a new path
- `moveTo(x, y)` : Move to a point without drawing
- `lineTo(x, y)` : Draw line to a point
- `closePath()` : Close the path by connecting to start point
- `stroke()` : Draw the path outline
- `fill()` : Fill the path

Drawing Lines

Basic Line from Your Code:

```
// Start new path
context.beginPath();

// Move to starting point (250, 60)
context.moveTo(250, 60);
```

```
// Draw line to ending point (400, 60)
context.lineTo(400, 60);

// Render the line
context.stroke();
```

Understanding the Process:

Step 1: beginPath()

```
context.beginPath();
```

Starts a new path. Without this, new lines would connect to previous paths.

Step 2: moveTo(x, y)

```
context.moveTo(250, 60);
```

Moves the "pen" to position (250, 60) without drawing. This sets the starting point.

Step 3: lineTo(x, y)

```
context.lineTo(400, 60);
```

Draws a line from current position to (400, 60).

Step 4: stroke()

```
context.stroke();
```

Actually renders the path. Without this, nothing appears on canvas.

Complete Line Example:

```
// Horizontal line (y values are same)
context.beginPath();
context.moveTo(250, 60);
context.lineTo(400, 60);  // y1 = y2 = 60 (horizontal)
context.stroke();

// Vertical line (x values are same)
context.beginPath();
context.moveTo(500, 160);
```

```
context.lineTo(500, 320);  // x1 = x2 = 500 (vertical)
context.stroke();

// Diagonal line
context.beginPath();
context.moveTo(500, 160);
context.lineTo(300, 320);  // Different x and y (diagonal)
context.stroke();
```

Line Types:

```
// Horizontal line: y1 = y2
// Example: (100, 50) to (300, 50)

// Vertical line: x1 = x2
// Example: (150, 50) to (150, 200)

// Diagonal line: x and y both change
// Example: (100, 100) to (300, 250)
```

Why beginPath() is Important

Without beginPath() (Wrong):

```
context.strokeStyle = "red";
context.moveTo(50, 50);
context.lineTo(150, 50);
context.stroke();

context.strokeStyle = "blue";
context.moveTo(50, 100);
context.lineTo(150, 100);
context.stroke();  // This redraws BOTH lines in blue!
```

With beginPath() (Correct):

```
context.strokeStyle = "red";
context.beginPath();  // Start fresh
context.moveTo(50, 50);
context.lineTo(150, 50);
context.stroke();

context.strokeStyle = "blue";
context.beginPath();  // Start fresh again
```

```
context.moveTo(50, 100);
context.lineTo(150, 100);
context.stroke();  // Only draws the blue line
```

Multiple Connected Lines

```
context.beginPath();
context.moveTo(100, 100);  // Start
context.lineTo(200, 100);  // First line
context.lineTo(200, 200);  // Second line (connected)
context.lineTo(100, 200);  // Third line (connected)
context.lineTo(100, 100);  // Fourth line (back to start)
context.stroke();
```

Drawing Shapes

Drawing Triangles

Method 1: Using lineTo() and closePath()

From Your Code:

```
context.beginPath();

// Set stroke color
context.strokeStyle = "green";

// Define three points of triangle
context.moveTo(620, 50);   // Top point
context.lineTo(520, 100);  // Bottom-left point
context.lineTo(720, 100);  // Bottom-right point

// Automatically connect back to start
context.closePath();

// Draw outline
context.stroke();

// Or fill
// context.fill();
```

Method 2: Manual Close (Less Efficient)

```
context.beginPath();
context.moveTo(620, 50);
context.lineTo(520, 100);
context.lineTo(720, 100);
context.lineTo(620, 50);  // Manually draw line back to start
context.stroke();
```

closePath() vs Manual Close:

With closePath():

- Automatically connects last point to first point
- Creates perfect corner joints
- More efficient

Without closePath():

- Need manual lineTo() back to start
- May have slight gaps at corners
- Less efficient

Filled vs Stroked:

```
// Filled triangle
context.beginPath();
context.moveTo(100, 50);
context.lineTo(50, 150);
context.lineTo(150, 150);
context.closePath();
context.fillStyle = "blue";
context.fill();

// Outlined triangle
context.beginPath();
context.moveTo(200, 50);
context.lineTo(150, 150);
context.lineTo(250, 150);
context.closePath();
context.strokeStyle = "red";
context.lineWidth = 3;
context.stroke();

// Both filled and outlined
context.beginPath();
```

```
context.moveTo(300, 50);
context.lineTo(250, 150);
context.lineTo(350, 150);
context.closePath();
context.fillStyle = "yellow";
context.fill();
context.strokeStyle = "black";
context.lineWidth = 2;
context.stroke();
```

Drawing Circles and Arcs

Method:

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

Parameters:

- `x` : Center X-coordinate
- `y` : Center Y-coordinate
- `radius` : Circle radius
- `startAngle` : Starting angle in radians
- `endAngle` : Ending angle in radians
- `counterclockwise` : Optional boolean (default: false/clockwise)

Understanding Radians:

```
0 radians = 0° (right/east)
Math.PI / 2 = 90° (bottom/south)
Math.PI = 180° (left/west)
Math.PI * 1.5 = 270° (top/north)
Math.PI * 2 = 360° (full circle)
```

Full Circle:

```
context.beginPath();
context.arc(100, 100, 50, 0, Math.PI * 2);
context.stroke();
```

Semi-Circle (From Your Code):

```
context.beginPath();
context.arc(550, 220, 50, 0, Math.PI, true);
context.stroke();
```

Parameters Explained:

- Center: (550, 220)
- Radius: 50
- Start: 0 (right side)
- End: Math.PI (left side)
- Direction: true (counterclockwise = draws top half)

Different Arcs:

```
// Full circle
context.beginPath();
context.arc(100, 100, 50, 0, Math.PI * 2);
context.stroke();

// Semi-circle (top half, counterclockwise)
context.beginPath();
context.arc(250, 100, 50, 0, Math.PI, true);
context.stroke();

// Semi-circle (bottom half, clockwise)
context.beginPath();
context.arc(400, 100, 50, 0, Math.PI, false);
context.stroke();

// Quarter circle (90°)
context.beginPath();
context.arc(550, 100, 50, 0, Math.PI / 2);
context.stroke();

// Three-quarter circle (270°)
context.beginPath();
context.arc(700, 100, 50, 0, Math.PI * 1.5);
context.stroke();

// Custom arc (45° to 135°)
context.beginPath();
context.arc(100, 250, 50, Math.PI / 4, Math.PI * 0.75);
context.stroke();
```

Clockwise vs Counterclockwise:

```
// Clockwise (default, false)
context.beginPath();
context.arc(100, 100, 50, 0, Math.PI, false);
context.stroke();
// Draws from 0° to 180° going clockwise (bottom arc)

// Counterclockwise (true)
context.beginPath();
context.arc(250, 100, 50, 0, Math.PI, true);
context.stroke();
// Draws from 0° to 180° going counterclockwise (top arc)
```

Filled Circle:

```
context.beginPath();
context.arc(100, 100, 50, 0, Math.PI * 2);
context.fillStyle = "blue";
context.fill();
```

Drawing Ellipses

Method:

```
context.ellipse(x, y, radiusX, radiusY, rotation, startAngle, endAngle,
counterclockwise);
```

Parameters:

- `x` : Center X-coordinate
- `y` : Center Y-coordinate
- `radiusX` : Horizontal radius
- `radiusY` : Vertical radius
- `rotation` : Rotation angle in radians
- `startAngle` : Starting angle in radians
- `endAngle` : Ending angle in radians
- `counterclockwise` : Optional boolean

From Your Code:

```
context.beginPath();
context.ellipse(550, 280, 60, 40, 0, 0, 2 * Math.PI);
context.stroke();
```

Parameters Explained:

- Center: (550, 280)
- Horizontal radius: 60
- Vertical radius: 40
- Rotation: 0 (no rotation)
- Full ellipse: 0 to 2π

Different Ellipses:

```
// Horizontal ellipse (wide)
context.beginPath();
context.ellipse(100, 100, 80, 40, 0, 0, Math.PI * 2);
context.stroke();

// Vertical ellipse (tall)
context.beginPath();
context.ellipse(250, 100, 40, 80, 0, 0, Math.PI * 2);
context.stroke();

// Rotated ellipse (45°)
context.beginPath();
context.ellipse(400, 100, 80, 40, Math.PI / 4, 0, Math.PI * 2);
context.stroke();

// Half ellipse
context.beginPath();
context.ellipse(550, 100, 80, 40, 0, 0, Math.PI);
context.stroke();

// Filled ellipse
context.beginPath();
context.ellipse(100, 250, 60, 40, 0, 0, Math.PI * 2);
context.fillStyle = "purple";
context.fill();
```

Rotation Parameter:

```
// No rotation (0 radians)
context.ellipse(100, 100, 80, 40, 0, 0, Math.PI * 2);

// 45° rotation (π/4 radians)
context.ellipse(250, 100, 80, 40, Math.PI / 4, 0, Math.PI * 2);

// 90° rotation (π/2 radians)
context.ellipse(400, 100, 80, 40, Math.PI / 2, 0, Math.PI * 2);
```

Drawing Text

Filled Text

Method:

```
context.fillText(text, x, y, maxWidth);
```

Parameters:

- `text` : String to draw
- `x` : X-coordinate (text baseline start)
- `y` : Y-coordinate (text baseline)
- `maxWidth` : Optional maximum width

From Your Code:

```
// Set text color
context.fillStyle = "red";

// Set font
context.font = "40px Arial";

// Draw filled text
context.fillText("Hello, Canvas!", 50, 210);
```

font Property:

```
// Format: "[style] [variant] [weight] size family"

context.font = "40px Arial";
```

```
context.font = "bold 30px Verdana";
context.font = "italic 25px Georgia";
context.font = "bold italic 35px 'Times New Roman'";
context.font = "20px sans-serif";
```

Font Components:

```
// Style: normal | italic | oblique
context.font = "italic 30px Arial";

// Variant: normal | small-caps
context.font = "small-caps 30px Arial";

// Weight: normal | bold | bolder | lighter | 100-900
context.font = "bold 30px Arial";
context.font = "300 30px Arial";  // Light
context.font = "700 30px Arial";  // Bold

// Size: Required (px, em, rem, pt, etc.)
context.font = "30px Arial";
context.font = "2em Arial";

// Family: Font name (required)
context.font = "30px Arial";
context.font = "30px 'Comic Sans MS'";  // Use quotes for multi-word fonts
```

Stroked Text (Outlined)

Method:

```
context.strokeText(text, x, y, maxWidth);
```

From Your Code:

```
// Set line width
context.lineWidth = 1;

// Set font
context.font = "60px Arial";

// Draw outlined text
context.strokeText("Hello, Canvas!", 270, 350);
```

Filled vs Stroked Text:

```
// Filled text (solid)
context.fillStyle = "blue";
context.font = "40px Arial";
context.fillText("Filled Text", 50, 50);

// Stroked text (outlined)
context.strokeStyle = "red";
context.lineWidth = 2;
context.font = "40px Arial";
context.strokeText("Stroked Text", 50, 100);

// Both filled and stroked
context.fillStyle = "yellow";
context.fillText("Both", 50, 150);
context.strokeStyle = "black";
context.lineWidth = 2;
context.strokeText("Both", 50, 150);
```

Text Alignment

textAlign Property:

```
context.textAlign = "start";   // Default (left for LTR)
context.textAlign = "end";     // Right for LTR
context.textAlign = "left";    // Always left
context.textAlign = "right";   // Always right
context.textAlign = "center";  // Center
```

Example:

```
const centerX = canvas.width / 2;

context.textAlign = "left";
context.fillText("Left aligned", centerX, 50);

context.textAlign = "center";
context.fillText("Center aligned", centerX, 100);

context.textAlign = "right";
context.fillText("Right aligned", centerX, 150);
```

textBaseline Property:

```
context.textBaseline = "alphabetic";  // Default
context.textBaseline = "top";
context.textBaseline = "hanging";
context.textBaseline = "middle";
context.textBaseline = "ideographic";
context.textBaseline = "bottom";
```

Visual Comparison:

```
const y = 200;

context.textBaseline = "top";
context.fillText("Top", 50, y);

context.textBaseline = "middle";
context.fillText("Middle", 150, y);

context.textBaseline = "bottom";
context.fillText("Bottom", 300, y);
```

Measuring Text

measureText() Method:

```
const text = "Hello, Canvas!";
context.font = "40px Arial";

const metrics = context.measureText(text);
console.log("Width:", metrics.width);  // Text width in pixels

// Use for centering
const x = (canvas.width - metrics.width) / 2;
context.fillText(text, x, 100);
```

Canvas Animation

Animation Basics

Animation on canvas involves:

1. Clear the canvas

2. Draw frame
3. Update positions
4. Repeat

From Your Code - Bouncing Ball:

```
const canvas = document.querySelector("canvas");
const context = canvas.getContext("2d");
const p = document.querySelector("p");

// Set canvas size
canvas.width = innerWidth * 0.8;
canvas.height = innerHeight * 0.6;

// Ball properties
let x = canvas.width / 2;   // Start at center X
let y = canvas.height / 2;  // Start at center Y
let r = 20;                  // Radius

// Velocity
let dx = 4;  // X velocity (pixels per frame)
let dy = 3;  // Y velocity (pixels per frame)

let counter = 0;  // Edge hit counter

function animate() {
    // 1. Clear previous frame
    context.clearRect(0, 0, canvas.width, canvas.height);

    // 2. Draw circle at current position
    context.fillStyle = "blue";
    context.beginPath();
    context.arc(x, y, r, 0, Math.PI * 2);
    context.fill();

    // 3. Update position
    x += dx;
    y += dy;

    // 4. Collision detection - horizontal walls
    if (x + r > canvas.width || x - r < 0) {
        dx = -dx;  // Reverse horizontal direction
        p.textContent = `Edge Hits: ${++counter}`;
    }
```

```
    // 5. Collision detection - vertical walls
    if (y + r > canvas.height || y - r < 0) {
        dy = -dy;  // Reverse vertical direction
        p.textContent = `Edge Hits: ${++counter}`;
    }
}

// Run animation at ~60fps (16ms per frame)
setInterval(animate, 16);
```

Animation Step-by-Step Explanation

Step 1: Clear Canvas

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

Erases previous frame to prevent trails.

Step 2: Draw Current Frame

```
context.fillStyle = "blue";
context.beginPath();
context.arc(x, y, r, 0, Math.PI * 2);
context.fill();
```

Draws circle at current (x, y) position.

Step 3: Update Position

```
x += dx;  // Move right by dx pixels
y += dy;  // Move down by dy pixels
```

Changes position for next frame.

Step 4: Collision Detection (Horizontal)

```
if (x + r > canvas.width || x - r < 0) {
    dx = -dx;
}
```

Explanation:

- `x + r > canvas.width` : Ball hit right wall

- `x - r < 0`: Ball hit left wall
- `dx = -dx`: Reverse horizontal direction (bounce)

Collision Detection (Vertical)

```
if (y + r > canvas.height || y - r < 0) {
    dy = -dy;
}
```

Explanation:

- `y + r > canvas.height`: Ball hit bottom wall
- `y - r < 0`: Ball hit top wall
- `dy = -dy`: Reverse vertical direction (bounce)

Why x + r and x - r?

```
Ball center at (x, y)
Ball radius = r

Left edge of ball: x - r
Right edge of ball: x + r
Top edge of ball: y - r
Bottom edge of ball: y + r

Canvas boundaries:
Left: 0
Right: canvas.width
Top: 0
Bottom: canvas.height
```

Animation Timing Methods

Method 1: setInterval (Your Code)

```
setInterval(animate, 16);  // Run every 16ms (~60fps)
```

Pros:

- Simple to understand
- Fixed frame rate

Cons:

- Doesn't sync with display refresh
- Continues when tab is inactive
- Can cause stuttering

Method 2: requestAnimationFrame (Better)

```
function animate() {
    // Clear and draw
    context.clearRect(0, 0, canvas.width, canvas.height);
    context.beginPath();
    context.arc(x, y, r, 0, Math.PI * 2);
    context.fill();

    // Update position
    x += dx;
    y += dy;

    // Collision detection
    if (x + r > canvas.width || x - r < 0) dx = -dx;
    if (y + r > canvas.height || y - r < 0) dy = -dy;

    // Request next frame
    requestAnimationFrame(animate);
}

// Start animation
requestAnimationFrame(animate);
```

Pros:

- Syncs with display refresh rate
- Pauses when tab inactive (saves CPU/battery)
- Smoother animation
- Better performance

Stopping Animation:

```
let animationId;

function animate() {
    // Animation code...

    animationId = requestAnimationFrame(animate);
}
```

```
// Start
animationId = requestAnimationFrame(animate);

// Stop
cancelAnimationFrame(animationId);
```

Advanced Animation - Multiple Balls

```
class Ball {
    constructor(x, y, dx, dy, r, color) {
        this.x = x;
        this.y = y;
        this.dx = dx;
        this.dy = dy;
        this.r = r;
        this.color = color;
    }

    draw() {
        context.beginPath();
        context.arc(this.x, this.y, this.r, 0, Math.PI * 2);
        context.fillStyle = this.color;
        context.fill();
    }

    update() {
        // Move
        this.x += this.dx;
        this.y += this.dy;

        // Bounce off walls
        if (this.x + this.r > canvas.width || this.x - this.r < 0) {
            this.dx = -this.dx;
        }
        if (this.y + this.r > canvas.height || this.y - this.r < 0) {
            this.dy = -this.dy;
        }

        this.draw();
    }
}

// Create multiple balls
const balls = [];
```

```
for (let i = 0; i < 10; i++) {
    const x = Math.random() * canvas.width;
    const y = Math.random() * canvas.height;
    const dx = (Math.random() - 0.5) * 4;
    const dy = (Math.random() - 0.5) * 4;
    const r = Math.random() * 20 + 10;
    const color = `hsl(${Math.random() * 360}, 70%, 50%)`;

    balls.push(new Ball(x, y, dx, dy, r, color));
}

function animate() {
    context.clearRect(0, 0, canvas.width, canvas.height);

    balls.forEach(ball => ball.update());

    requestAnimationFrame(animate);
}

animate();
```

Frame Rate Independence

For smooth animation across different frame rates:

```
let lastTime = 0;

function animate(timestamp) {
    // Calculate delta time
    const deltaTime = timestamp - lastTime;
    lastTime = timestamp;

    // Clear canvas
    context.clearRect(0, 0, canvas.width, canvas.height);

    // Update position based on time elapsed
    const speed = 200;  // pixels per second
    x += dx * speed * (deltaTime / 1000);
    y += dy * speed * (deltaTime / 1000);

    // Draw
    context.beginPath();
    context.arc(x, y, r, 0, Math.PI * 2);
    context.fill();
```

```
    requestAnimationFrame(animate);
}

requestAnimationFrame(animate);
```

Canvas Events and Interaction

Mouse Events

From Your Code:

```
canvas.addEventListener("mousemove", function(e) {
    console.log(e.clientX, e.clientY);
});
```

Understanding Mouse Coordinates:

clientX / clientY:

- Relative to viewport (browser window)
- Does NOT account for canvas position or scroll

Getting Correct Canvas Coordinates:

```
canvas.addEventListener("mousemove", function(e) {
    // Get canvas bounding rectangle
    const rect = canvas.getBoundingClientRect();

    // Calculate position relative to canvas
    const x = e.clientX - rect.left;
    const y = e.clientY - rect.top;

    console.log("Canvas coordinates:", x, y);
});
```

Alternative Method:

```
canvas.addEventListener("mousemove", function(e) {
    const x = e.offsetX;  // X relative to canvas
    const y = e.offsetY;  // Y relative to canvas
```

```
    console.log(x, y);
});
```

Drawing with Mouse

Paint Program Example:

```
const canvas = document.querySelector("canvas");
const context = canvas.getContext("2d");

let isDrawing = false;

// Mouse down - start drawing
canvas.addEventListener("mousedown", function(e) {
    isDrawing = true;
    context.beginPath();
    context.moveTo(e.offsetX, e.offsetY);
});

// Mouse move - draw line
canvas.addEventListener("mousemove", function(e) {
    if (isDrawing) {
        context.lineTo(e.offsetX, e.offsetY);
        context.stroke();
    }
});

// Mouse up - stop drawing
canvas.addEventListener("mouseup", function() {
    isDrawing = false;
});

// Mouse leaves canvas - stop drawing
canvas.addEventListener("mouseleave", function() {
    isDrawing = false;
});
```

Click to Place Objects

```
canvas.addEventListener("click", function(e) {
    const x = e.offsetX;
    const y = e.offsetY;

    // Draw circle at click position
    context.beginPath();
```

```
    context.arc(x, y, 20, 0, Math.PI * 2);
    context.fillStyle = "blue";
    context.fill();
});
```

Interactive Animation - Follow Mouse

```
let mouseX = 0;
let mouseY = 0;
let x = canvas.width / 2;
let y = canvas.height / 2;

canvas.addEventListener("mousemove", function(e) {
    mouseX = e.offsetX;
    mouseY = e.offsetY;
});

function animate() {
    context.clearRect(0, 0, canvas.width, canvas.height);

    // Move toward mouse position
    x += (mouseX - x) * 0.1;
    y += (mouseY - y) * 0.1;

    // Draw circle
    context.beginPath();
    context.arc(x, y, 20, 0, Math.PI * 2);
    context.fillStyle = "blue";
    context.fill();

    requestAnimationFrame(animate);
}

animate();
```

Keyboard Controls

```
let x = canvas.width / 2;
let y = canvas.height / 2;
const speed = 5;

document.addEventListener("keydown", function(e) {
    switch(e.key) {
        case "ArrowUp":
```

```
            y -= speed;
            break;
        case "ArrowDown":
            y += speed;
            break;
        case "ArrowLeft":
            x -= speed;
            break;
        case "ArrowRight":
            x += speed;
            break;
    }

    // Redraw
    context.clearRect(0, 0, canvas.width, canvas.height);
    context.beginPath();
    context.arc(x, y, 20, 0, Math.PI * 2);
    context.fill();
});
```

SVG (Scalable Vector Graphics)

What is SVG?

SVG (Scalable Vector Graphics) is an XML-based markup language for describing two-dimensional vector graphics.

Key Characteristics

SVG is Vector-Based:

- Defined by mathematical equations, not pixels
- Resolution-independent (always crisp, no pixelation)
- Can be scaled infinitely without quality loss
- Each shape is a DOM element

When to Use SVG:

- Icons and logos
- Charts and graphs
- UI elements
- Illustrations

- Interactive graphics (need event listeners on individual shapes)
- Graphics that need to scale (responsive design)

SVG Advantages:

- Infinitely scalable
- Small file sizes for simple graphics
- Fully accessible (screen readers can read text)
- CSS styleable
- Can attach events to individual elements
- Search engines can index SVG text

SVG Limitations:

- Performance degrades with many elements (1000+)
- Not suitable for photo-realistic images
- More complex than raster for complex scenes

Basic SVG Structure

```
<svg width="700" height="400">
    <!-- Shapes go here -->
</svg>
```

Attributes:

- `width` : SVG width
- `height` : SVG height
- `viewBox` : Defines coordinate system (optional)

SVG Rectangles

From Your Code:

```
<svg width="700" height="400">
    <!-- Filled rectangle -->
    <rect x="50" y="50" width="100" height="80" fill="blue" />

    <!-- Outlined rectangle -->
    <rect x="200" y="100" width="60" height="40"
          stroke="red" stroke-width="4" fill="none" />
</svg>
```

Attributes:

- `x` : X-coordinate of top-left corner
- `y` : Y-coordinate of top-left corner
- `width` : Rectangle width
- `height` : Rectangle height
- `fill` : Fill color (or "none")
- `stroke` : Stroke/outline color
- `stroke-width` : Stroke thickness

Comparison to Canvas:

```
// Canvas
context.fillStyle = "blue";
context.fillRect(50, 50, 100, 80);

// SVG (equivalent)
<rect x="50" y="50" width="100" height="80" fill="blue" />
```

SVG Lines

From Your Code:

```
<line x1="50" y1="300" x2="250" y2="300"
      stroke="green" stroke-width="3" />
```

Attributes:

- `x1` , `y1` : Starting point coordinates
- `x2` , `y2` : Ending point coordinates
- `stroke` : Line color (required, lines have no fill)
- `stroke-width` : Line thickness

Multiple Lines:

```
<svg width="500" height="300">
    <!-- Horizontal line -->
    <line x1="50" y1="50" x2="250" y2="50" stroke="red" stroke-width="2" />

    <!-- Vertical line -->
    <line x1="150" y1="50" x2="150" y2="250" stroke="blue" stroke-width="2" />
```

```
    <!-- Diagonal line -->
    <line x1="50" y1="50" x2="250" y2="250" stroke="green" stroke-width="2" />
</svg>
```

SVG Circles and Ellipses

Circle (From Your Code):

```
<circle cx="400" cy="150" r="50" />
```

Attributes:

- `cx` : Center X-coordinate
- `cy` : Center Y-coordinate
- `r` : Radius

Ellipse (From Your Code):

```
<ellipse cx="400" cy="250" rx="60" ry="30"
         fill="none" stroke="green" stroke-width="4" />
```

Attributes:

- `cx` : Center X-coordinate
- `cy` : Center Y-coordinate
- `rx` : Horizontal radius
- `ry` : Vertical radius

Examples:

```
<svg width="700" height="300">
    <!-- Filled circle -->
    <circle cx="100" cy="150" r="50" fill="blue" />

    <!-- Outlined circle -->
    <circle cx="250" cy="150" r="50" fill="none" stroke="red" stroke-width="3"
/>

    <!-- Filled ellipse -->
    <ellipse cx="400" cy="150" rx="80" ry="50" fill="purple" />

    <!-- Outlined ellipse -->
    <ellipse cx="600" cy="150" rx="80" ry="50"
```

```
            fill="none" stroke="orange" stroke-width="3" />
</svg>
```

SVG Polygons

Polygon (From Your Code):

```
<polygon points="600,50 500,150 700,150"
         fill="none" stroke="orange" stroke-width="3" />
```

Attributes:

- `points` : Space or comma-separated list of x,y coordinates
- Format: "x1,y1 x2,y2 x3,y3 ..."

Polygon automatically closes the shape (connects last point to first).

Examples:

```
<svg width="700" height="400">
    <!-- Triangle -->
    <polygon points="100,50 50,150 150,150" fill="blue" />

    <!-- Pentagon -->
    <polygon points="300,50 250,100 270,170 330,170 350,100"
             fill="red" />

    <!-- Hexagon -->
    <polygon points="500,80 450,120 450,180 500,220 550,180 550,120"
             fill="green" />

    <!-- Star -->
    <polygon points="400,280 420,340 480,350 430,390 450,450 400,420
                     350,450 370,390 320,350 380,340"
             fill="gold" stroke="orange" stroke-width="2" />
</svg>
```

SVG Polylines

Polyline (From Your Code):

```
<polyline points="600,50 500,150 700,150 600,50"
          fill="none" stroke="orange" stroke-width="3" />
```

Difference from Polygon:

- **Polyline**: Does NOT automatically close (open path)
- **Polygon**: Automatically closes (closed path)

Comparison:

```
<svg width="500" height="200">
    <!-- Polyline - NOT closed -->
    <polyline points="50,50 100,100 150,50 200,100"
              fill="none" stroke="blue" stroke-width="3" />

    <!-- Polygon - Automatically closed -->
    <polygon points="250,50 300,100 350,50 400,100"
             fill="none" stroke="red" stroke-width="3" />
</svg>
```

SVG Text

From Your Code:

```
<text x="50" y="200">Hello, SVG</text>
```

Attributes:

- `x` : X-coordinate (text start)
- `y` : Y-coordinate (text baseline)

Styled Text:

```
<svg width="700" height="300">
    <!-- Basic text -->
    <text x="50" y="50">Hello, SVG</text>

    <!-- Styled text -->
    <text x="50" y="100"
          font-family="Arial"
          font-size="30"
          fill="blue"
          font-weight="bold">
        Styled Text
    </text>

    <!-- Outlined text -->
```

```
    <text x="50" y="150"
          font-size="40"
          fill="none"
          stroke="red"
          stroke-width="2">
        Outlined Text
    </text>

    <!-- Rotated text -->
    <text x="50" y="200"
          font-size="25"
          transform="rotate(45 50 200)">
        Rotated Text
    </text>
</svg>
```

SVG Paths

Most powerful SVG element - can create any shape.

Basic Path:

```
<path d="M 50 50 L 150 50 L 100 150 Z"
      fill="blue" stroke="black" stroke-width="2" />
```

Path Commands:

- `M x y` : Move to (x, y)
- `L x y` : Line to (x, y)
- `H x` : Horizontal line to x
- `V y` : Vertical line to y
- `Z` : Close path
- `C x1 y1, x2 y2, x y` : Cubic Bezier curve
- `Q x1 y1, x y` : Quadratic Bezier curve
- `A rx ry rotation large-arc sweep x y` : Arc

Examples:

```
<!-- Triangle using path -->
<path d="M 100 50 L 50 150 L 150 150 Z" fill="blue" />

<!-- Curve -->
<path d="M 50 200 Q 100 100 150 200" fill="none" stroke="red" stroke-width="3"
```

```
/>

<!-- Complex shape -->
<path d="M 200 50 L 250 100 L 300 50 L 350 100 Z"
      fill="yellow" stroke="black" stroke-width="2" />
```

SVG Groups

Group elements together:

```
<svg width="500" height="300">
    <g id="house" transform="translate(100, 100)">
        <!-- House base -->
        <rect x="0" y="50" width="100" height="80" fill="brown" />

        <!-- Roof -->
        <polygon points="0,50 50,0 100,50" fill="red" />

        <!-- Door -->
        <rect x="40" y="90" width="20" height="40" fill="white" />

        <!-- Window -->
        <rect x="20" y="70" width="15" height="15" fill="lightblue" />
        <rect x="65" y="70" width="15" height="15" fill="lightblue" />
    </g>
</svg>
```

SVG Styling with CSS

SVG elements can be styled with CSS:

```
<style>
    .my-circle {
        fill: blue;
        stroke: red;
        stroke-width: 3;
        transition: fill 0.3s;
    }

    .my-circle:hover {
        fill: red;
        cursor: pointer;
    }
</style>
```

```svg
<svg width="200" height="200">
    <circle class="my-circle" cx="100" cy="100" r="50" />
</svg>
```

SVG with JavaScript

```javascript
// Select SVG element
const svg = document.querySelector('svg');

// Create new circle
const circle = document.createElementNS('http://www.w3.org/2000/svg',
'circle');
circle.setAttribute('cx', '100');
circle.setAttribute('cy', '100');
circle.setAttribute('r', '50');
circle.setAttribute('fill', 'blue');

// Add to SVG
svg.appendChild(circle);

// Add event listener
circle.addEventListener('click', function() {
    circle.setAttribute('fill', 'red');
});
```

Canvas vs SVG

Detailed Comparison

Feature	Canvas	SVG
Type	Raster (bitmap)	Vector
Scalability	Pixelates when scaled	Infinitely scalable
DOM	No DOM elements	Each shape is DOM element
Events	Canvas-level only	Per-element events
Performance	Better for many objects (1000+)	Slower with many objects
Accessibility	Poor	Good (screen reader compatible)
File Size	Large for complex images	Small for simple graphics
Animation	Manual frame-by-frame	CSS/SMIL animations possible

Feature	Canvas	SVG
Text	Rasterized	Selectable, searchable
Memory	Less memory	More memory (DOM nodes)

When to Use Canvas

Use Canvas For:

1. **Games**: Fast-paced games with many moving objects
2. **Real-time Data Visualization**: Live graphs, charts updating frequently
3. **Image Manipulation**: Filters, effects, pixel-level operations
4. **Particle Effects**: Thousands of particles
5. **Complex Animations**: Many animated elements (100+)
6. **Pixel Art**: Drawing/painting applications

Canvas Strengths:

- High performance with many objects
- Pixel-level control
- Good for photo manipulation
- Lower memory usage

Example Use Cases:

```
// Canvas is better for:
// - Bouncing balls animation (your code example)
// - Drawing applications
// - Image filters
// - Real-time graphing (stock charts)
// - Game development
```

When to Use SVG

Use SVG For:

1. **Icons and Logos**: Need to scale without loss
2. **UI Elements**: Buttons, decorative elements
3. **Charts/Graphs**: Static or simple interactive charts
4. **Infographics**: Need to be crisp at any size
5. **Interactive Elements**: Need click events on individual shapes
6. **Responsive Graphics**: Must look good on any screen

SVG Strengths:

- Resolution independence
- Accessibility
- Individual element control
- CSS styling
- Smaller file size for simple graphics
- Text is selectable

Example Use Cases:

```
<!-- SVG is better for: -->
<!-- - Company logos -->
<!-- - Icon sets -->
<!-- - Data visualizations (simple charts) -->
<!-- - Interactive diagrams -->
<!-- - Responsive illustrations -->
```

Performance Comparison

Canvas Performance:

```
// Canvas handles 10,000 circles well
for (let i = 0; i < 10000; i++) {
    context.beginPath();
    context.arc(Math.random() * width, Math.random() * height, 5, 0, Math.PI *
2);
    context.fill();
}
// Still performant
```

SVG Performance:

```
<!-- SVG struggles with 1,000+ elements -->
<svg width="800" height="600">
    <!-- 10,000 circles would cause significant slowdown -->
</svg>
```

Rule of Thumb:

- **< 100 objects**: SVG is fine
- **100-1,000 objects**: Either works, depends on use case

- **> 1,000 objects**: Use Canvas

Hybrid Approach

You can use both together:

```
<!-- SVG for UI elements -->
<svg style="position: absolute; top: 0; left: 0;">
    <rect x="10" y="10" width="100" height="30" fill="white" />
    <text x="20" y="30">Score: 100</text>
</svg>

<!-- Canvas for game graphics -->
<canvas id="game"></canvas>

<!-- SVG for controls -->
<svg style="position: absolute; bottom: 0;">
    <circle id="button1" cx="50" cy="50" r="40" fill="blue" />
    <circle id="button2" cx="150" cy="50" r="40" fill="red" />
</svg>
```

Decision Flowchart

```
Need to draw graphics?
├─ Static image/icon that needs to scale?
│  └─ Use SVG
│
├─ Need events on individual elements?
│  └─ Use SVG
│
├─ More than 1,000 objects?
│  └─ Use Canvas
│
├─ Need pixel-level manipulation?
│  └─ Use Canvas
│
├─ Animation with many objects?
│  └─ Use Canvas
│
└─ Interactive chart/diagram (< 100 elements)?
   └─ Use SVG
```

Summary

Canvas Key Points

1. **Raster-based** graphics drawn with JavaScript
2. **Immediate mode**: Draw and forget (no object memory)
3. **High performance** for many objects
4. **Manual animation** with frame-by-frame clearing
5. **Event handling** only at canvas level
6. **Best for**: Games, animations, image manipulation

SVG Key Points

1. **Vector-based** graphics defined in XML/HTML
2. **Retained mode**: Each shape is a DOM element
3. **Infinitely scalable** without quality loss
4. **Individual events** on each element
5. **CSS styleable** and accessible
6. **Best for**: Icons, logos, UI elements, simple charts

Code Patterns Summary

Canvas Pattern:

```
// 1. Get context
const context = canvas.getContext('2d');

// 2. Set styles
context.fillStyle = "blue";
context.strokeStyle = "red";

// 3. Draw shapes
context.fillRect(x, y, width, height);
context.beginPath();
context.arc(x, y, r, 0, Math.PI * 2);
context.fill();

// 4. Animation
function animate() {
    context.clearRect(0, 0, canvas.width, canvas.height);
    // Draw frame
    requestAnimationFrame(animate);
}
```

SVG Pattern:

```
<svg width="700" height="400">
    <rect x="50" y="50" width="100" height="80" fill="blue" />
    <circle cx="200" cy="100" r="50" fill="red" />
    <text x="50" y="200">Hello, SVG</text>
</svg>
```

Both Canvas and SVG are powerful tools for web graphics. Choose based on your specific needs: Canvas for performance and complex animations, SVG for scalability and interactivity.

Abdullah Ali

Contact : +201012613453