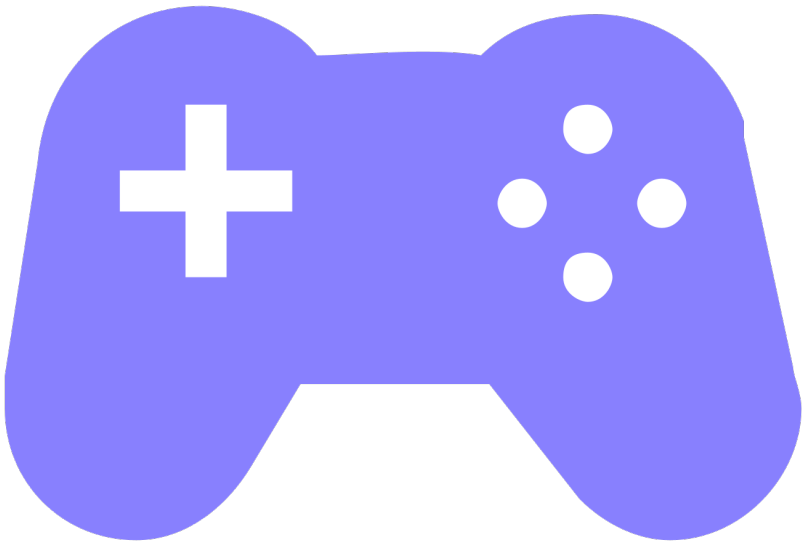
JavaScript HTML5 Animation



readbytes

JavaScript HTML5 Animation

Creating Dynamic and Interactive Web Experiences

readbytes.github.io

2025-07-28

This page is intentionally left blank.

Contents

Chapter 1.

JavaScript Animation Foundations

1. Basic Animation with JavaScript
2. Introduction to HTML5 and Canvas
3. Canvas Support and Performance
4. Creating an HTML5 Document

1 JavaScript Animation Foundations

1.1 Basic Animation with JavaScript

JavaScript plays a central role in web animation, acting as the engine that drives dynamic motion and interaction. While CSS can handle simple transitions and animations, JavaScript gives you fine-grained, programmatic control—allowing you to animate virtually anything: position, color, opacity, size, rotation, and more. JavaScript is what brings logic and decision-making into animation, enabling interactivity, user-driven motion, and procedural behaviors.

Before modern techniques like `requestAnimationFrame()`, early JavaScript animations were often built using timing functions such as `setTimeout()` and `setInterval()`. These functions allow developers to run code at specified intervals, which can be used to simulate motion by repeatedly updating and redrawing elements on the screen.

Let's begin with a simple example using `setInterval()`:

```
  <div id="box" style="width:50px;height:50px;background:red;position:absolute;"></div>

  <script>
    let box = document.getElementById("box");
    let x = 0;

    setInterval(function() {
      x += 2;
      box.style.left = x + "px";
    }, 16); // roughly 60 frames per second
  </script>
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<head>
  <title>Simple Animation</title>
</head>
<body>
  <div id="box" style="width:50px;height:50px;background:red;position:absolute;"></div>

  <script>
    let box = document.getElementById("box");
    let x = 0;

    setInterval(function() {
      x += 2;
      box.style.left = x + "px";
    }, 16); // roughly 60 frames per second
  </script>
</body>
</html>
```

In this example, a red box is moved to the right by increasing its `left` style property every 16 milliseconds (approx. 60fps). The result is a very basic animation—smooth enough for simple effects.

Alternatively, `setTimeout()` can be used recursively to achieve similar results:

```
function moveBox() {
  x += 2;
  box.style.left = x + "px";
  setTimeout(moveBox, 16);
}
moveBox();
```

While these timing functions make animation relatively straightforward, they have important limitations. First, they are not synchronized with the browser's internal rendering cycle. This can lead to jittery motion, dropped frames, or performance issues—especially if multiple animations or heavy computations are involved.

Second, `setInterval()` doesn't account for how long the function's code actually takes to run. If the logic inside takes longer than the interval, the animation may lag or behave unpredictably. Also, if a browser tab is inactive or minimized, many browsers throttle or pause these timers to save resources, disrupting the animation loop.

These shortcomings led to the development of a better solution: `requestAnimationFrame()`, which is designed specifically for rendering smooth, high-performance animations. This function ensures that animation updates are synchronized with the display's refresh rate and provides more efficient and consistent timing.

Despite their limitations, `setTimeout()` and `setInterval()` are still valuable tools for learning the basics of animation logic. They help illustrate the core principle: repeatedly update an object's state (like its position) and redraw it on the screen. Understanding these timing functions lays the groundwork for mastering more advanced and efficient animation techniques that follow in later sections.

1.2 Introduction to HTML5 and Canvas

The introduction of the `<canvas>` element in HTML5 revolutionized web-based graphics and animation. Before `<canvas>`, dynamic visual content on the web relied heavily on plug-ins like Flash or complex DOM manipulations. With `<canvas>`, developers gained the power to draw directly onto a pixel-based surface using JavaScript—enabling fast, flexible, and highly interactive 2D (and even 3D) graphics in modern browsers.

The `<canvas>` element is essentially a blank slate: a drawable region defined in HTML with a specified width and height. It doesn't render anything on its own until JavaScript is used to draw on it. Unlike DOM elements (like `<div>` or `<img>`), the content of a canvas is not made up of HTML elements. Instead, it is drawn procedurally, pixel by pixel, using a JavaScript-based drawing context.

Here's a basic example of a canvas element and a script that draws a filled rectangle:

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

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

  ctx.fillStyle = "blue";
  ctx.fillRect(50, 50, 100, 100);
</script>
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="myCanvas" width="400" height="300"></canvas>

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

  ctx.fillStyle = "blue";
  ctx.fillRect(50, 50, 100, 100);
</script>
</body>
</html>
```

In this example, we access the canvas using `getElementById()` and then retrieve the 2D rendering context using `getContext("2d")`. The context (`ctx`) provides an API of drawing functions—such as `fillRect()`, `stroke()`, `beginPath()`, and `arc()`—that let us render shapes, text, images, and more directly onto the canvas.

The true power of `<canvas>` comes when combined with JavaScript animation techniques. By updating the canvas on a per-frame basis, developers can simulate motion, physics, and interaction in real time. Animations typically involve clearing the canvas, updating object positions or states, and redrawing the scene in a loop—effectively repainting every frame.

Because canvas works at the pixel level, it allows for fine-tuned control over visual output. This makes it ideal for scenarios where precision and performance matter: games, data visualizations, particle systems, interactive interfaces, and custom visual effects. Unlike DOM-based animations, which are bound by the structure and layout of HTML elements, canvas animations are only limited by the logic in your JavaScript and the capabilities of the user's device.

Moreover, canvas integrates seamlessly with modern JavaScript APIs such as `requestAnimationFrame()` for optimized rendering. It also pairs well with event handling, image loading, and audio to create rich multimedia experiences.

In short, the HTML5 `<canvas>` element provides a lightweight, performant, and flexible platform for building advanced visual experiences on the web. When driven by JavaScript, it becomes a powerful tool for creating dynamic, interactive animations that respond in real time to user input and application logic.

1.3 Canvas Support and Performance

The HTML5 `<canvas>` element is widely supported across all modern browsers, including Chrome, Firefox, Safari, Edge, and even mobile browsers like Android WebView and Safari on iOS. This broad compatibility makes `<canvas>` a dependable foundation for building rich, animated web experiences. However, achieving smooth, high-performance animation goes beyond just using the canvas—it requires careful attention to performance, rendering strategies, and system resources.

One major factor in canvas performance is *hardware acceleration*. Modern browsers leverage the GPU (Graphics Processing Unit) whenever possible to offload intensive rendering operations from the CPU. This results in faster, smoother animations and better responsiveness. In most cases, canvas rendering in a browser is GPU-accelerated by default, but performance can still vary based on the complexity of the drawing operations and the device's hardware.

To ensure smooth animations, aim for a consistent frame rate of 60 frames per second (fps), which matches the refresh rate of most monitors. If rendering or computations take too long per frame, the animation may drop frames or appear choppy. Several strategies can help you maintain good performance:

Minimize Redrawing

Redrawing the entire canvas on every frame is the most common approach in canvas animation, but it can be expensive. If only a portion of the canvas changes, consider *partial redraws*—only clearing and redrawing the affected regions. This is particularly useful in applications like charting, particle effects, or games with static backgrounds.

Use `requestAnimationFrame()`

Avoid older timing methods like `setInterval()` for animation. Instead, use `requestAnimationFrame()`, which is optimized for animation and ensures that updates are synchronized with the display's refresh rate. This method reduces CPU usage and provides smoother rendering.

```
function animate() {
  update();       // Update animation state
  draw();         // Redraw canvas
  requestAnimationFrame(animate);
}
animate();
```

Avoid Expensive Operations in Loops

Heavy computation inside your animation loop—such as image processing, DOM access, or large object creation—can cause frame delays. Precompute values where possible and reuse objects to avoid garbage collection pauses.

Canvas Size and Resolution

A larger canvas size means more pixels to manage and draw. Use only the dimensions you need, and consider scaling your drawings appropriately for high-DPI (Retina) displays to

balance quality and performance.

Image and Asset Optimization

Loading large images, frequent pattern fills, or uncompressed media can slow rendering. Compress and optimize assets, and cache images in memory instead of reloading them in each frame.

By understanding how canvas interacts with hardware and the browser rendering engine, you can make design choices that keep animations fluid and responsive. Optimizing your drawing strategies and animation logic will ensure your application performs well across a wide range of devices.

1.4 Creating an HTML5 Document

Before you can build canvas-based animations, you need a well-structured HTML5 document to serve as your foundation. A clean and complete document ensures compatibility, scalability, and ease of development across modern browsers.

Here's a basic example of an HTML5 document ready for Canvas animation:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Canvas Animation Example</title>
  <style>
    body {
      margin: 0;
      overflow: hidden; /* Prevent scrollbars when animating */
    }
    canvas {
      display: block;
      background: #f0f0f0;
    }
  </style>
</head>
<body>
  <canvas id="animationCanvas" width="800" height="600"></canvas>

  <script src="animation.js"></script>
</body>
</html>
```

Key Components:

- **`<!DOCTYPE html>`**: Declares the document as HTML5, enabling modern browser features.
- **`<meta charset="UTF-8">`**: Ensures proper text encoding.
- **`<meta name="viewport">`**: Makes the page responsive on mobile and scales properly.

- **`<title>`**: Sets the title shown in the browser tab.
- **`<style>`**: Includes basic CSS for layout and aesthetics. Disabling scrollbars and setting canvas display to `block` helps center the focus on the animation.
- **`<canvas>` element**: The primary drawing surface. `id`, `width`, and `height` attributes are essential for targeting and sizing the canvas.
- **`<script src="animation.js">`**: Loads your JavaScript animation logic from an external file. Keeping logic in separate files improves maintainability and reusability.

This minimal setup gives you a canvas drawing surface ready to animate. In upcoming sections, you'll learn how to use JavaScript to access the canvas, draw objects, and animate them in real time.

Chapter 2.

JavaScript Animation Foundations 2

1. CSS and Additional Scripts
2. Debugging Basics
3. Animation Loops

2 JavaScript Animation Foundations 2

2.1 CSS and Additional Scripts

Although the `<canvas>` element itself is a programmable drawing surface, CSS still plays an important role in how it appears and behaves on the page. Proper styling ensures that your canvas is positioned correctly, scales as needed, and integrates seamlessly into your layout or fullscreen animation environment.

Styling the Canvas with CSS

By default, a canvas is an inline element with no styling beyond its dimensions. To control its appearance and placement, you'll often use CSS for:

- **Sizing and Positioning**: Use CSS to center the canvas, fill the screen, or fit within a container.

```
canvas {
  display: block;
  margin: 0 auto;
  background-color: #000;
}
```

- **Fullscreen Canvas**: For immersive animations or games, it's common to make the canvas cover the entire viewport:

```
html, body {
  margin: 0;
  padding: 0;
  overflow: hidden;
  height: 100%;
}
canvas {
  width: 100%;
  height: 100%;
  display: block;
}
```

Note: Setting the canvas size in CSS only affects its *display size*, not its *drawing resolution*. To ensure crisp rendering, always match the canvas's width and height *attributes* in HTML or JavaScript to its CSS dimensions.

Additional JavaScript Libraries and Utilities

While raw JavaScript and the Canvas API provide low-level control, animation often benefits from reusable libraries or utility functions. Here are a few popular tools that can accelerate development:

- **GSAP (GreenSock Animation Platform)**: A powerful animation library with timeline control, easing, and robust performance.
- **anime.js**: Lightweight and great for animating properties, transforms, and SVG, often used alongside canvas animations.

- **dat.GUI**: Useful for adding developer-facing controls (like sliders and toggles) to test animation parameters in real time.
- **stats.js**: A small performance monitor that helps track frame rates and CPU usage while tuning your animation loop.

Integrating these libraries can reduce boilerplate code and allow you to focus on creative aspects rather than technical complexity. That said, understanding the fundamentals of canvas and animation logic will always be essential, even when using libraries.

By combining thoughtful CSS styling with optional animation utilities, you create a strong foundation for responsive, maintainable, and visually polished canvas-based animations.

2.2 Debugging Basics

Debugging animations in JavaScript and HTML5 Canvas requires a combination of general JavaScript debugging skills and animation-specific strategies. Because animations run in real time, issues like dropped frames, jerky motion, or incorrect rendering can be subtle and timing-sensitive. Fortunately, modern browsers offer powerful developer tools to help isolate and fix these problems.

Use the Browser Console

The first line of defense is the browser's JavaScript console (usually accessible with F12 or right-click → "Inspect"). Use `console.log()` statements to track variable values, function calls, and object states during the animation loop. This helps you understand the sequence of updates and identify unexpected behaviors.

Example:

```
console.log("x position:", x);
```

Just be careful not to log too much inside tight animation loops—it can slow down rendering or overwhelm the console. For performance-critical sections, log only on condition or at set intervals.

Set Breakpoints and Step Through Code

Using the **Sources** tab in Chrome or **Debugger** tab in Firefox, you can set breakpoints to pause execution and inspect the call stack, variable states, and execution flow. This is especially helpful when debugging complex logic like collision detection, object movement, or state changes.

Monitor Frame Rate and Performance

For animation, frame rate is crucial. Use tools like:

- **Chrome DevTools Performance tab**: Record an animation session and analyze

frame rendering time, JavaScript execution, and paint operations.
- **FPS meter in DevTools** (Rendering tab → "FPS meter"): Displays a live graph of your animation's frame rate.
- **Third-party tools like stats.js**: Add a real-time FPS counter directly in your animation viewport to monitor performance while developing.

Common Pitfalls

- **Forgetting to clear the canvas**: Results in "ghosting" or trails.
- **Incorrect `requestAnimationFrame()` usage**: Not calling it recursively means your animation will run only once.
- **Improper canvas sizing**: Mismatch between CSS and attribute dimensions can distort visuals.
- **Memory leaks**: Unmanaged object creation inside loops can cause slowdowns or crashes.

By using developer tools effectively and watching for common issues, you can debug animation code with confidence and build smoother, more reliable experiences.

2.3 Animation Loops

At the heart of every dynamic animation lies the **animation loop**—a continuous cycle that updates the state of your scene and redraws it on the screen, frame by frame. Without this loop, your animation would be static and motionless. The animation loop is what drives the illusion of movement, turning a series of still images into fluid, engaging visuals.

Why Animation Loops Are Essential

Animations are not one-time events; they require constant updates over time. An animation loop ensures that your program repeatedly:

1. **Updates** the properties of objects (like position, size, color, or velocity) based on elapsed time or user input.
2. **Clears** the canvas or drawing area to remove previous frames.
3. **Draws** the updated objects to render the current frame.

This repetition happens many times per second—typically around 60 frames per second—to create smooth motion.

Without an animation loop, you'd have to manually trigger every frame update, which is impractical and inefficient. The loop automates this process and synchronizes it with the browser's rendering cycle for optimal performance.

Implementing a Basic Animation Loop

A simple animation loop can be created using `setInterval()`, `setTimeout()`, or, more effectively, `requestAnimationFrame()`. The key idea is to create a function that calls itself repeatedly to update and render your animation.

Here's a basic example using `requestAnimationFrame()`:

```
function animate() {
  update();      // Update animation state (positions, physics, etc.)
  draw();        // Clear canvas and redraw scene

  requestAnimationFrame(animate); // Schedule the next frame
}

animate(); // Start the loop
```

In this setup:

- The `animate()` function is responsible for the entire animation cycle.
- It first calls `update()`, where you change object properties to reflect movement or other changes.
- Then, it calls `draw()`, which clears the canvas and redraws everything in its new state.
- Finally, it calls `requestAnimationFrame(animate)`, which tells the browser to call `animate()` again before the next repaint.

Why Use `requestAnimationFrame()`?

`requestAnimationFrame()` is designed specifically for animations. It synchronizes your updates with the browser's refresh rate (usually 60 Hz), resulting in smoother motion and more efficient CPU and GPU usage compared to timers like `setInterval()`. It also pauses animations automatically when the tab is inactive, saving resources.

Handling Time in Animation Loops

To ensure consistent movement regardless of frame rate variations, it's common to track the time elapsed between frames and update animation states accordingly. This technique, called *delta timing*, helps animations run at the same speed on different devices.

In summary, animation loops are the engine that powers continuous motion in web animations. Mastering them is crucial for creating smooth, performant, and interactive animated experiences with JavaScript and HTML5 Canvas.

Chapter 3.

JavaScript Animation Foundations 3

1. Using `requestAnimationFrame()`
2. JavaScript Objects and Prototypes
3. Functional Programming Style
4. User Interaction: Events and Listeners

3 JavaScript Animation Foundations 3

3.1 Using `requestAnimationFrame()`

When it comes to creating smooth and efficient animations on the web, the `requestAnimationFrame()` API is the modern gold standard. It's specifically designed to handle animation rendering by syncing your animation updates with the browser's native refresh cycle, providing superior performance and visual quality compared to traditional timers like `setTimeout()` and `setInterval()`.

What is `requestAnimationFrame()`?

`requestAnimationFrame()` is a method provided by the browser that tells it you want to perform an animation and requests that the browser calls a specified callback function *before* the next repaint. This timing ensures that animations update at the optimal moment—right before the screen is redrawn—helping avoid unnecessary renders and visual glitches such as flickering or tearing.

Advantages Over Traditional Timers

1. **Synchronization with Display Refresh Rate** Most modern screens refresh at 60 frames per second (fps). Traditional timers run independently of this refresh rate and may trigger more or fewer updates than necessary. `requestAnimationFrame()` naturally aligns animation updates to this refresh cycle, which leads to smoother motion.
2. **Better Performance and Power Efficiency** Since `requestAnimationFrame()` only triggers when the browser is ready to repaint, it reduces CPU and GPU workload. When the browser tab is hidden or minimized, the API automatically pauses, saving processing power and battery life—a feature traditional timers lack.
3. **Improved Timing Accuracy** Timers like `setTimeout()` are subject to delays caused by other scripts and browser workload, potentially causing uneven frame intervals. `requestAnimationFrame()` provides more consistent frame timing, which is essential for fluid animation.

How to Use `requestAnimationFrame()`

Here's a simple example illustrating its use:

```
const canvas = document.getElementById('myCanvas');
const ctx = canvas.getContext('2d');
let x = 0;

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

  ctx.fillStyle = 'green';
  ctx.fillRect(x, 50, 50, 50);  // Draw a square moving across the canvas

  x += 2;  // Update position
```

```
  if (x < canvas.width) {
    requestAnimationFrame(animate);  // Schedule next frame
  }
}

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

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="myCanvas" width="400" height="300"></canvas>

<script>
const canvas = document.getElementById('myCanvas');
const ctx = canvas.getContext('2d');
let x = 0;

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

  ctx.fillStyle = 'green';
  ctx.fillRect(x, 50, 50, 50);  // Draw a square moving across the canvas

  x += 2;  // Update position

  if (x < canvas.width) {
    requestAnimationFrame(animate);  // Schedule next frame
  }
}

animate();  // Start animation loop
</script>
</body>
</html>
```

In this example, the `animate()` function draws a green square and then schedules itself to run again before the next repaint using `requestAnimationFrame()`. The square moves smoothly across the canvas because the browser controls the timing of each frame.

Using the Timestamp Parameter

`requestAnimationFrame()` passes a high-resolution timestamp to the callback function representing the exact time the frame is scheduled to be painted. This timestamp enables you to implement *delta timing*—updating animations based on the time elapsed rather than assuming a fixed frame rate.

Example with delta timing:

```
let lastTime = 0;
let x = 0;

function animate(time) {
  if (!lastTime) lastTime = time;
  const deltaTime = time - lastTime;
```

```
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.fillRect(x, 50, 50, 50);

  x += 0.1 * deltaTime;  // Move based on time elapsed

  lastTime = time;

  if (x < canvas.width) {
    requestAnimationFrame(animate);
  }
}

requestAnimationFrame(animate);
```

By factoring in `deltaTime`, your animations stay consistent regardless of fluctuations in frame rate.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="myCanvas" width="400" height="300"></canvas>

<script>
const canvas = document.getElementById('myCanvas');
const ctx = canvas.getContext('2d');
let lastTime = 0;
let x = 0;

function animate(time) {
  if (!lastTime) lastTime = time;
  const deltaTime = time - lastTime;

  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.fillRect(x, 50, 50, 50);

  x += 0.1 * deltaTime;  // Move based on time elapsed

  lastTime = time;

  if (x < canvas.width) {
    requestAnimationFrame(animate);
  }
}

requestAnimationFrame(animate);
</script>
</body>
</html>
```

Summary

`requestAnimationFrame()` is a critical API for performant, smooth animations in modern web development. Its synchronization with the browser's refresh rate, automatic pausing in

inactive tabs, and high-precision timing make it far superior to traditional timer functions. Using it effectively lets you build animations that are visually pleasing and efficient, enhancing the user experience across devices and platforms.

3.2 JavaScript Objects and Prototypes

When building animations, especially complex or interactive ones, organizing your code effectively becomes essential. JavaScript's **objects** and **prototypes** provide powerful tools for structuring animation logic, creating reusable components, and managing state cleanly.

What Are JavaScript Objects?

An object in JavaScript is a collection of properties and methods that represent a thing or concept. For animations, objects can represent entities like shapes, sprites, or UI elements. Each object can hold data such as position, velocity, color, and also methods to update or draw itself.

Here's a simple example of an animated object representing a moving square:

```
function Square(x, y, size, color, speed) {
  this.x = x;
  this.y = y;
  this.size = size;
  this.color = color;
  this.speed = speed;
}

Square.prototype.update = function() {
  this.x += this.speed;
};

Square.prototype.draw = function(ctx) {
  ctx.fillStyle = this.color;
  ctx.fillRect(this.x, this.y, this.size, this.size);
};
```

In this example:

- `Square` is a **constructor function** that creates new square objects with specific properties.
- The methods `update()` and `draw()` are defined on the `Square.prototype`. This means all instances of `Square` share these methods, saving memory and promoting reuse.
- The `update()` method changes the square's position, while `draw()` renders it on the canvas.

You can create multiple squares easily:

```
const squares = [
  new Square(10, 50, 30, 'red', 2),
  new Square(20, 100, 40, 'blue', 3),
  new Square(30, 150, 50, 'green', 1.5),
```

```
];
```

Then, within your animation loop, you can update and draw each one:

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

  squares.forEach(square => {
    square.update();
    square.draw(ctx);
  });

  requestAnimationFrame(animate);
}

animate();
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Animated Squares</title>
  <style>
    body {
      margin: 0;
      background: #111;
    }
    canvas {
      display: block;
      background: #222;
    }
  </style>
</head>
<body>
  <canvas id="canvas"></canvas>

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

    function resizeCanvas() {
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
    }
    window.addEventListener('resize', resizeCanvas);
    resizeCanvas();

    // Square constructor
    function Square(x, y, size, color, speed) {
      this.x = x;
      this.y = y;
      this.size = size;
      this.color = color;
      this.speed = speed;
```

```
    }

    Square.prototype.update = function() {
      this.x += this.speed;
      if (this.x > canvas.width) this.x = -this.size; // wrap around
    };

    Square.prototype.draw = function(ctx) {
      ctx.fillStyle = this.color;
      ctx.fillRect(this.x, this.y, this.size, this.size);
    };

    // Create multiple squares
    const squares = [
      new Square(10, 50, 30, 'red', 2),
      new Square(20, 100, 40, 'blue', 3),
      new Square(30, 150, 50, 'lime', 1.5),
      new Square(80, 200, 25, 'orange', 2.5),
    ];

    // Animation loop
    function animate() {
      ctx.clearRect(0, 0, canvas.width, canvas.height);
      squares.forEach(square => {
        square.update();
        square.draw(ctx);
      });
      requestAnimationFrame(animate);
    }

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

Why Use Prototypes?

JavaScript uses **prototypes** to share methods among all instances created by a constructor. This approach is more memory-efficient than attaching methods directly to each object. It also helps keep your code organized by grouping related functionality.

Modern Alternative: ES6 Classes

In modern JavaScript, you can use **classes**, which provide cleaner syntax for defining objects and prototypes:

```
class Square {
  constructor(x, y, size, color, speed) {
    this.x = x;
    this.y = y;
    this.size = size;
    this.color = color;
    this.speed = speed;
  }

  update() {
    this.x += this.speed;
```

```
  }

  draw(ctx) {
    ctx.fillStyle = this.color;
    ctx.fillRect(this.x, this.y, this.size, this.size);
  }
}
```

Using objects and prototypes (or classes) makes your animation code modular, easier to maintain, and reusable. By encapsulating behavior and state in objects, you build a strong foundation for creating dynamic, interactive, and scalable animations.

3.3 Functional Programming Style

Functional programming (FP) principles can greatly improve the clarity, maintainability, and predictability of animation code. At its core, FP encourages writing **pure functions**, avoiding side effects, and embracing **immutability**—all of which help create code that is easier to reason about and debug.

What Are Pure Functions?

A pure function is a function that, given the same input, always returns the same output and does not modify any external state. It does not rely on or change variables outside its scope. This makes pure functions predictable and easy to test.

For example, consider a function that calculates the next position of an object moving at a constant speed:

```
function calculateNextPosition(position, speed) {
  return position + speed;
}
```

This function is pure: it doesn't modify the original position or any other variable; it just returns a new value.

Embracing Immutability

Immutability means never changing existing data structures directly. Instead, create new copies with the necessary updates. This avoids unexpected bugs caused by shared state mutations and makes your animation state easier to track.

Instead of updating an object's position property directly, you can create a new object with the updated position:

Full runnable code:

```
function moveObject(object, speed) {
  return {
    ...object,
```

```
    x: object.x + speed,
  };
}

const original = { x: 10, y: 20 };
const moved = moveObject(original, 5);

console.log(original.x); // 10 (unchanged)
console.log(moved.x);    // 15 (new object)
```

Benefits for Animation Code

Applying FP principles to animation code results in:

- **Predictable updates:** Pure functions make it easier to understand how state changes frame-to-frame.
- **Simplified debugging:** Since functions don't cause side effects, you can test them in isolation.
- **Cleaner state management:** Immutability helps track changes over time, making features like undo/rewind easier to implement.
- **Better concurrency support:** Avoiding shared mutable state reduces race conditions and timing bugs.

Example: Functional Animation Update

Here's a simplified example where an animation's state is updated functionally:

```
function updateState(state, deltaTime) {
  return {
    ...state,
    x: state.x + state.vx * deltaTime,
    y: state.y + state.vy * deltaTime,
  };
}

// Initial state
let state = { x: 0, y: 0, vx: 100, vy: 50 };

function animate(time = 0) {
  const deltaTime = 0.016; // Assume 60fps for simplicity
  state = updateState(state, deltaTime);

  draw(state);

  requestAnimationFrame(animate);
}
```

In this pattern, each animation frame produces a *new* state based on the previous one without mutating the original, preserving clarity and making the flow of data explicit.

By adopting functional programming styles, you create animation code that is robust, maintainable, and easier to extend—an especially valuable approach for complex or interactive projects.

3.4 User Interaction: Events and Listeners

Interactivity is a crucial element of engaging animations, and JavaScript's event system allows you to respond to user actions such as clicks, mouse movements, and keyboard input. By handling these events, you can make your animations dynamic, personalized, and immersive.

Understanding Events and Event Listeners

An **event** represents an action or occurrence that happens in the browser, like a mouse click or a key press. To respond to these events, you use **event listeners**—functions that run when a specific event occurs on a particular element.

You add an event listener with the `addEventListener()` method:

```
element.addEventListener('eventType', callbackFunction);
```

Here, `eventType` might be `'click'`, `'mousemove'`, `'keydown'`, and `callbackFunction` is the function that executes when the event happens.

Handling Mouse Events

Mouse events are essential for interactive canvas animations. For example, you might want to move an object when the user moves their mouse or trigger an animation when they click.

```
const canvas = document.getElementById('myCanvas');
canvas.addEventListener('mousemove', function(event) {
  const rect = canvas.getBoundingClientRect();
  const mouseX = event.clientX - rect.left;
  const mouseY = event.clientY - rect.top;

  // Update animation state based on mouse position
  console.log(`Mouse position: (${mouseX}, ${mouseY})`);
});
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="myCanvas" width="400" height="300"></canvas>

<script>
const canvas = document.getElementById('myCanvas');
canvas.addEventListener('mousemove', function(event) {
  const rect = canvas.getBoundingClientRect();
  const mouseX = event.clientX - rect.left;
  const mouseY = event.clientY - rect.top;

  // Update animation state based on mouse position
  console.log(`Mouse position: (${mouseX}, ${mouseY})`);
});
</script>
</body>
</html>
```

This listener calculates the mouse position relative to the canvas and can be used to move objects or trigger effects dynamically.

Handling Keyboard Events

Keyboard input can control animations or game mechanics. Keyboard events are typically attached to the `window` or `document` object:

```
window.addEventListener('keydown', function(event) {
  if (event.key === 'ArrowRight') {
    // Move object right
    console.log('Right arrow pressed');
  }
});
```

Using keyboard events, you can create controls for your animations—like moving a character or triggering actions.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="myCanvas" width="400" height="300"></canvas>

<script>
window.addEventListener('keydown', function(event) {
  if (event.key === 'ArrowRight') {
    // Move object right
    console.log('Right arrow pressed');
  }
});
</script>
</body>
</html>
```

Updating Animation State

When an event occurs, you usually want to update the animation's internal state so that the next frame reflects the change. For example:

```
let circle = { x: 50, y: 50 };

canvas.addEventListener('click', function(event) {
  const rect = canvas.getBoundingClientRect();
  circle.x = event.clientX - rect.left;
  circle.y = event.clientY - rect.top;
});
```

Then in your animation loop, draw the circle at its updated position:

```
function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.beginPath();
  ctx.arc(circle.x, circle.y, 20, 0, Math.PI * 2);
  ctx.fill();
```

```
  requestAnimationFrame(animate);
}

animate();
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Click to Move Circle</title>
  <style>
    body {
      margin: 0;
      background: #111;
    }
    canvas {
      display: block;
      background: #222;
      cursor: crosshair;
    }
  </style>
</head>
<body>
  <canvas id="canvas"></canvas>

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

    function resizeCanvas() {
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
    }
    window.addEventListener('resize', resizeCanvas);
    resizeCanvas();

    // Initial circle state
    let circle = { x: canvas.width / 2, y: canvas.height / 2 };

    // Update circle position on click
    canvas.addEventListener('click', function(event) {
      const rect = canvas.getBoundingClientRect();
      circle.x = event.clientX - rect.left;
      circle.y = event.clientY - rect.top;
    });

    // Animation loop
    function animate() {
      ctx.clearRect(0, 0, canvas.width, canvas.height);

      ctx.beginPath();
      ctx.arc(circle.x, circle.y, 20, 0, Math.PI * 2);
      ctx.fillStyle = 'deepskyblue';
      ctx.fill();
```

```
      requestAnimationFrame(animate);
    }

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

Summary

By combining event listeners with your animation logic, you transform static motion into rich, interactive experiences. Whether responding to clicks, tracking mouse movements, or processing keyboard input, user events enable your animations to react intuitively to user behavior—making your web experiences engaging and fun.

Chapter 4.

Trigonometry for Animation 1

1. Understanding Angles, Radians, and Degrees
2. Coordinate Systems and Triangle Sides
3. Trigonometric Functions
4. Rotation and Circular Motion

4 Trigonometry for Animation 1

4.1 Understanding Angles, Radians, and Degrees

In animation, angles play a crucial role in controlling direction, rotation, and circular motion. Whether you're rotating a shape, simulating wave patterns, or moving objects in an arc, understanding how angles are measured—and how they relate to the coordinate system—is fundamental.

Degrees vs. Radians

Angles can be measured in **degrees** or **radians**. Degrees are more familiar in everyday contexts—360 degrees in a circle, 90 degrees for a right angle, and so on. Radians, on the other hand, are more natural to mathematical functions and, therefore, more common in programming environments like JavaScript.

A **radian** is defined as the angle created when the arc length of a circle is equal to the radius. There are exactly **2Pi radians** in a full circle, which corresponds to **360 degrees**.

Conversion Between Degrees and Radians

Since radians are often required in animation formulas—especially those using trigonometric functions like `Math.sin()` and `Math.cos()`—you'll frequently need to convert between degrees and radians.

The conversion formulas are:

- **Degrees to Radians**:
  ```
  radians = degrees * (Math.PI / 180);
  ```
- **Radians to Degrees**:
  ```
  degrees = radians * (180 / Math.PI);
  ```

For example, to convert 90 degrees to radians:

```
const angle = 90;
const radians = angle * (Math.PI / 180);  //  1.5708
```

Why Use Radians in Programming?

JavaScript's built-in trigonometric functions (`Math.sin`, `Math.cos`, `Math.tan`) all use radians—not degrees. This convention comes from mathematics, where radians provide a more direct relationship between angles and arc lengths, making them better suited for calculations involving motion and curves.

Using radians enables smooth integration with circular and oscillatory motion:

```
let angle = 0;

function animate() {
  const x = Math.cos(angle) * 100;
```

```
  const y = Math.sin(angle) * 100;

  angle += 0.05; // radians

  requestAnimationFrame(animate);
}
```

In this example, the object moves in a circular path because `x` and `y` are derived from cosine and sine, which require the angle in radians.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Circle Animation</title>
  <style>
    body {
      margin: 0;
      background: #111;
    }
    canvas {
      display: block;
      background: #222;
    }
  </style>
</head>
<body>
  <canvas id="canvas"></canvas>
  <script>
    const canvas = document.getElementById('canvas');
    const ctx = canvas.getContext('2d');

    function resizeCanvas() {
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
    }
    window.addEventListener('resize', resizeCanvas);
    resizeCanvas();

    let angle = 0;

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

      const centerX = canvas.width / 2;
      const centerY = canvas.height / 2;
      const radius = 100;

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

      ctx.beginPath();
      ctx.arc(x, y, 10, 0, Math.PI * 2);
      ctx.fillStyle = 'lime';
      ctx.fill();
```

```
      angle += 0.05;

      requestAnimationFrame(animate);
    }

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

Summary

To animate rotations or wave-like patterns in JavaScript, you need to understand angles—and work in radians. While degrees are intuitive, radians are essential for the math behind motion. Mastering the conversion and internalizing what a radian represents will allow you to control your animations with greater precision and fluidity.

4.2 Coordinate Systems and Triangle Sides

To animate objects effectively in a 2D space, it's essential to understand how positions are defined and how distances and angles are calculated. This begins with the **Cartesian coordinate system**, the foundational framework for almost all 2D animation on the web.

Cartesian Coordinate System in 2D

In a two-dimensional (2D) Cartesian coordinate system, every point in space is represented by an **(x, y)** pair:

- The **x-axis** runs horizontally, increasing to the right.
- The **y-axis** runs vertically, increasing *downward* in most screen-based coordinate systems like the HTML5 `<canvas>` (this is the opposite of traditional mathematical graphs, where y increases upward).
- The **origin (0, 0)** is the top-left corner of the canvas unless explicitly transformed.

For example, a point `(100, 200)` refers to a position 100 pixels right and 200 pixels down from the top-left corner.

This system is key to placing, moving, and animating objects.

Triangles in Animation

Triangles are a critical geometric shape in animation because they allow you to break down complex shapes and movements into simple, calculable parts. One of the most useful types is the **right triangle**, where one angle is 90 degrees.

A right triangle gives us a powerful tool: the **Pythagorean theorem**:

```
a² + b² = c²
```

Where:

- `a` and `b` are the legs (horizontal and vertical sides),
- `c` is the hypotenuse (the side opposite the right angle).

This formula is essential for calculating **distance** between two points:

```
function distance(x1, y1, x2, y2) {
  const dx = x2 - x1;
  const dy = y2 - y1;
  return Math.sqrt(dx * dx + dy * dy);
}
```

Positioning with Triangles

Suppose you want to animate an object moving at an angle. You can treat its motion as the hypotenuse of a triangle and break it down into horizontal (`dx`) and vertical (`dy`) components using trigonometry (covered in the next section).

Understanding the relationship between a point's coordinates and its triangle-based properties (like angle and distance) allows you to:

- Move an object toward a target,
- Rotate objects around a point,
- Simulate physics like velocity and direction.

Summary

Mastering the 2D Cartesian system and the geometry of triangles equips you with the tools to calculate position, direction, and distance. These fundamentals form the backbone of animation logic and prepare you for more complex trigonometric transformations and motion techniques.

4.3 Trigonometric Functions

Trigonometric functions play a critical role in animation by enabling smooth, natural motion and shape manipulation. Among these functions, the most fundamental are **sine (`sin`)**, **cosine (`cos`)**, and **tangent (`tan`)**. These functions relate the angles of a triangle to the ratios of its sides and are invaluable for tasks like rotating objects, simulating oscillations, or placing points along a circular path.

The Basics

Trigonometric functions are typically defined in terms of a **right triangle**:

- **Sine (sin)** of an angle = *opposite side / hypotenuse*
- **Cosine (cos)** of an angle = *adjacent side / hypotenuse*

- **Tangent (tan)** of an angle = *opposite side / adjacent side*

When applied in JavaScript animations, these functions allow you to compute **x** and **y** positions using angles, which is especially useful for circular motion and angular positioning.

Unit Circle and Radians

In animation, trigonometric functions are usually applied in the context of the **unit circle**, a circle with a radius of 1 centered at the origin `(0, 0)`. For any angle (in **radians**, not degrees), you can find a point on the circle like this:

```
x = Math.cos(angle);
y = Math.sin(angle);
```

This technique is used frequently to compute directions or animate movement in circular patterns.

Example: Rotating an Object Around a Point

Suppose you want to make a dot rotate around the center of a canvas. Using sine and cosine, you can calculate the position of the dot on each frame based on an angle:

```
const centerX = 300;
const centerY = 200;
const radius = 100;
let angle = 0;

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

  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.beginPath();
  ctx.arc(x, y, 10, 0, Math.PI * 2);
  ctx.fill();

  angle += 0.05;
  requestAnimationFrame(draw);
}
draw();
```

Here, the circle's position is recalculated each frame using cosine and sine. As the angle increases, the dot moves smoothly around the center, forming a circular path.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Rotating Dot Example</title>
  <style>
    body {
      margin: 0;
      background: #1e1e1e;
    }
```

```
    canvas {
      display: block;
      background-color: #111;
    }
  </style>
</head>
<body>
  <canvas id="canvas"></canvas>

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

    // Resize canvas to fill the window
    function resize() {
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
    }
    window.addEventListener('resize', resize);
    resize();

    const centerX = canvas.width / 2;
    const centerY = canvas.height / 2;
    const radius = 100;
    let angle = 0;

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

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

      // Draw central point (optional)
      ctx.beginPath();
      ctx.arc(centerX, centerY, 4, 0, Math.PI * 2);
      ctx.fillStyle = "#ccc";
      ctx.fill();

      // Draw orbiting dot
      ctx.beginPath();
      ctx.arc(x, y, 10, 0, Math.PI * 2);
      ctx.fillStyle = "cyan";
      ctx.fill();

      angle += 0.05;

      requestAnimationFrame(draw);
    }

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

Tangent in Animation

While sine and cosine are more common in positioning, **tangent** is useful for calculating **slopes** or angles between points. For example, to find the angle between two points:

```
const dx = x2 - x1;
const dy = y2 - y1;
const angle = Math.atan2(dy, dx);
```

This is particularly useful for rotating an object to face another point, such as a sprite turning toward the mouse.

Summary

Understanding sine, cosine, and tangent functions unlocks powerful capabilities in animation: calculating precise positions, rotating objects, generating waves, and creating natural, fluid motion. With just these tools, you can simulate or animate countless dynamic behaviors in your HTML5 canvas projects.

4.4 Rotation and Circular Motion

Trigonometric functions are essential tools for creating rotation and circular motion in animation. With the help of `Math.sin()` and `Math.cos()`, you can calculate positions around a circle and rotate shapes smoothly. These functions allow objects to orbit around a center, spin in place, or face a particular direction based on angles.

Rotating Around a Point

To rotate an object (such as a dot, sprite, or shape) around a pivot point, you can use sine and cosine to compute its new position based on an angle. This is especially useful for simulating planetary orbits, rotating gears, or even animating clock hands.

The formula is simple:

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

Here, `centerX` and `centerY` are the coordinates of the pivot, `radius` is the distance from the pivot to the object, and `angle` is in radians.

Example: Circular Motion

Below is a basic example using the `<canvas>` element to animate a circle moving around a central point.

```
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");
```

```
const centerX = canvas.width / 2;
const centerY = canvas.height / 2;
const radius = 100;
let angle = 0;

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

  // Compute the orbiting position
  const x = centerX + Math.cos(angle) * radius;
  const y = centerY + Math.sin(angle) * radius;

  // Draw orbiting circle
  ctx.beginPath();
  ctx.arc(x, y, 10, 0, Math.PI * 2);
  ctx.fillStyle = "#ff6600";
  ctx.fill();

  // Draw center point
  ctx.beginPath();
  ctx.arc(centerX, centerY, 4, 0, Math.PI * 2);
  ctx.fillStyle = "#333";
  ctx.fill();

  angle += 0.02;
  requestAnimationFrame(animate);
}
animate();
</script>
```

This script creates an orange circle orbiting the center of the canvas using trigonometric calculations. The `angle` variable increments each frame, and the `Math.cos()` and `Math.sin()` values translate that angle into X and Y coordinates.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

const centerX = canvas.width / 2;
const centerY = canvas.height / 2;
const radius = 100;
let angle = 0;

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

  // Compute the orbiting position
  const x = centerX + Math.cos(angle) * radius;
  const y = centerY + Math.sin(angle) * radius;
```

```
  // Draw orbiting circle
  ctx.beginPath();
  ctx.arc(x, y, 10, 0, Math.PI * 2);
  ctx.fillStyle = "#ff6600";
  ctx.fill();

  // Draw center point
  ctx.beginPath();
  ctx.arc(centerX, centerY, 4, 0, Math.PI * 2);
  ctx.fillStyle = "#333";
  ctx.fill();

  angle += 0.02;
  requestAnimationFrame(animate);
}
animate();
</script>
</body>
</html>
```

Rotating a Shape in Place

To rotate a shape like a rectangle in place (without changing its center), you use the canvas context's transform methods:

```
ctx.translate(x, y);         // Move the origin to the center
ctx.rotate(angle);           // Rotate the canvas
ctx.fillRect(-w/2, -h/2, w, h); // Draw the shape centered on the origin
ctx.setTransform(1, 0, 0, 1, 0, 0); // Reset transformation
```

This approach is great for rotating static shapes like pointers, turbines, or rotating characters.

Summary

Using trigonometric functions, you can animate rotation and circular motion with mathematical precision. Whether orbiting around a point or spinning in place, `Math.sin()` and `Math.cos()` give you full control over motion paths, enabling lifelike, engaging animations that enhance interactivity and visual appeal.

Chapter 5.

Trigonometry for Animation 2

1. Creating Waves and Oscillations
2. Circles, Ellipses, and Distance Calculation

5 Trigonometry for Animation 2

5.1 Creating Waves and Oscillations

Trigonometric functions such as `Math.sin()` and `Math.cos()` are powerful tools for creating smooth, periodic motion in JavaScript animations. These functions naturally produce oscillations—repeating up-and-down or side-to-side motions—making them perfect for simulating effects like bouncing balls, waving flags, pulsing lights, or bobbing characters.

Understanding Oscillation with Sine and Cosine

The sine and cosine functions generate values between -1 and 1 as their input (angle, in radians) increases. This predictable wave pattern lets you map time to movement. By scaling and shifting the output, you can control amplitude (height), frequency (speed), and vertical offset (baseline) of the motion.

The general formula:

```
value = amplitude * Math.sin(frequency * time + phase) + offset;
```

- `amplitude`: how far the motion swings (e.g., pixels)
- `frequency`: how fast it oscillates
- `time`: usually tied to `performance.now()` or an `angle` variable
- `phase`: shift of the wave start (optional)
- `offset`: base position around which the value oscillates

Example: Vertical Bobbing Animation

Here's a simple animation where a circle moves up and down smoothly:

```
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

let time = 0;

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

  const centerX = canvas.width / 2;
  const baseY = canvas.height / 2;
  const amplitude = 40;
  const frequency = 0.05;

  const y = baseY + Math.sin(time) * amplitude;

  ctx.beginPath();
  ctx.arc(centerX, y, 20, 0, Math.PI * 2);
  ctx.fillStyle = "#66ccff";
  ctx.fill();

  time += frequency;
  requestAnimationFrame(animate);
```

```
}
animate();
</script>
```

This creates a circle that smoothly oscillates up and down around the center line, mimicking a floating or bouncing motion.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

let time = 0;

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

  const centerX = canvas.width / 2;
  const baseY = canvas.height / 2;
  const amplitude = 40;
  const frequency = 0.05;

  const y = baseY + Math.sin(time) * amplitude;

  ctx.beginPath();
  ctx.arc(centerX, y, 20, 0, Math.PI * 2);
  ctx.fillStyle = "#66ccff";
  ctx.fill();

  time += frequency;
  requestAnimationFrame(animate);
}
animate();
</script>
</body>
</html>
```

Simulating a Waving Flag

To simulate a waving flag, you can apply a sine wave across multiple vertical points:

```
for (let x = 0; x < canvas.width; x += 10) {
  const y = baseY + Math.sin(x * 0.1 + time) * 20;
  ctx.lineTo(x, y);
}
```

This creates a continuous sine wave, and by animating the `time` variable, the wave appears to travel horizontally. Attach this wave pattern to the top edge of a flag graphic or shape, and the rest of the fabric can be interpolated downward to simulate wind.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Waving Flag Simulation</title>
  <style>
    body {
      margin: 0;
      overflow: hidden;
      background: #003366;
    }
    canvas {
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="flagCanvas"></canvas>

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

    function resizeCanvas() {
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
    }

    window.addEventListener("resize", resizeCanvas);
    resizeCanvas();

    let time = 0;

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

      const baseY = canvas.height / 2;

      ctx.beginPath();
      ctx.moveTo(0, baseY);

      for (let x = 0; x <= canvas.width; x += 10) {
        const y = baseY + Math.sin(x * 0.02 + time) * 30;
        ctx.lineTo(x, y);
      }

      ctx.lineTo(canvas.width, canvas.height);
      ctx.lineTo(0, canvas.height);
      ctx.closePath();

      ctx.fillStyle = "#cc0000"; // Red flag color
      ctx.fill();

      time += 0.05;
      requestAnimationFrame(drawFlag);
    }

    drawFlag();
```

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

Practical Use Cases

- **Bouncing Balls**: Use a sine wave for subtle bounces at rest or combine with gravity for realistic rebound.
- **Floating Objects**: Apply sine-based vertical offset to icons or sprites.
- **Heartbeat Effects**: Use cosine-based pulsing for scaling or opacity animations.

Trigonometric oscillations add life and realism to animations. Their natural rhythm and mathematical predictability make them a reliable choice for smooth, repeatable, and eye-catching motion across countless use cases.

5.2 Circles, Ellipses, and Distance Calculation

Circles and ellipses are fundamental shapes in animation. Whether you're drawing orbiting planets, simulating bubbles, or creating collision detection between objects, understanding the trigonometry behind these curves—and how to measure distance between points—is essential.

Drawing Circles with Trigonometry

A circle can be defined using sine and cosine functions:

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

As the `angle` increases from 0 to 2Pi radians (or 0° to 360°), the point (x, y) traces the circle's perimeter. Here's an example that animates a dot around a circular path:

```
const radius = 100;
const centerX = 300;
const centerY = 200;
let angle = 0;

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

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

  ctx.beginPath();
  ctx.arc(x, y, 10, 0, Math.PI * 2);
  ctx.fillStyle = "orange";
  ctx.fill();

  angle += 0.02;
  requestAnimationFrame(animate);
}
```

```
animate();
```

This code produces a dot revolving around the center point in a smooth circle.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Animated Circle with Trigonometry</title>
  <style>
    body {
      margin: 0;
      overflow: hidden;
      background: #111;
    }
    canvas {
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="circleCanvas"></canvas>

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

    function resizeCanvas() {
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
    }

    window.addEventListener("resize", resizeCanvas);
    resizeCanvas();

    const radius = 100;
    const centerX = canvas.width / 2;
    const centerY = canvas.height / 2;
    let angle = 0;

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

      // Draw static circle
      ctx.beginPath();
      ctx.arc(centerX, centerY, radius, 0, Math.PI * 2);
      ctx.strokeStyle = "white";
      ctx.lineWidth = 2;
      ctx.stroke();

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

      // Draw animated dot
```

```
      ctx.beginPath();
      ctx.arc(x, y, 10, 0, Math.PI * 2);
      ctx.fillStyle = "orange";
      ctx.fill();

      angle += 0.02;
      requestAnimationFrame(animate);
    }

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

Drawing Ellipses

An ellipse stretches the circle along one or both axes. The only change is using separate radii for x and y directions:

```
x = centerX + radiusX * Math.cos(angle);
y = centerY + radiusY * Math.sin(angle);
```

This approach animates objects along elliptical orbits—useful for simulating moons, satellites, or stylized motion.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Animated Circle with Trigonometry</title>
  <style>
    body {
      margin: 0;
      overflow: hidden;
      background: #111;
    }
    canvas {
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="circleCanvas"></canvas>

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

    function resizeCanvas() {
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
    }

    window.addEventListener("resize", resizeCanvas);
```

```
    resizeCanvas();

    const radius = 100;
    const centerX = canvas.width / 2;
    const centerY = canvas.height / 2;
    let angle = 0;

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

      // Draw static circle
      ctx.beginPath();
      ctx.arc(centerX, centerY, radius, 0, Math.PI * 2);
      ctx.strokeStyle = "white";
      ctx.lineWidth = 2;
      ctx.stroke();

      // Calculate dot position
      x = centerX + radiusX * Math.cos(angle);
      y = centerY + radiusY * Math.sin(angle);

      // Draw animated dot
      ctx.beginPath();
      ctx.arc(x, y, 10, 0, Math.PI * 2);
      ctx.fillStyle = "orange";
      ctx.fill();

      angle += 0.02;
      requestAnimationFrame(animate);
    }

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

Distance Between Points

Calculating the distance between two points is key to handling interactions like collisions or proximity triggers. The standard Euclidean formula is:

```
function distance(x1, y1, x2, y2) {
  const dx = x2 - x1;
  const dy = y2 - y1;
  return Math.sqrt(dx * dx + dy * dy);
}
```

This can be used for:

- **Collision detection** (e.g., between two circles):

  ```
  if (distance(a.x, a.y, b.x, b.y) < a.radius + b.radius) {
    // Collision occurred
  }
  ```

- **Constraining motion**: Keeping an object within a certain radius from a point.
- **Trigger zones**: Checking if the mouse or player is near a target.

Combining Concepts

With circular paths, elliptical motion, and distance checking, you can create sophisticated interactions—like an object orbiting until it collides with another, or an ellipse-based path that a sprite follows while responding to user input.

These trigonometric tools and geometric calculations form the bedrock of dynamic, responsive animations in canvas-based games and visualizations.

Chapter 6.

Drawing on the Canvas 1

1. Color Management and Transparency
2. The Drawing API and Canvas Context
3. Drawing Lines, Curves, and Paths
4. Filling Shapes and Creating Gradients

6 Drawing on the Canvas 1

6.1 Color Management and Transparency

In canvas-based animation and graphics, color is more than decoration—it defines mood, hierarchy, interactivity, and visual feedback. The HTML5 `<canvas>` element provides rich options for setting colors, handling transparency, and layering visuals through compositing.

Setting Colors

You can set colors for fills and strokes using three main formats: RGB, HEX, and HSL.

- **RGB (Red, Green, Blue)**: Each component ranges from 0 to 255.

```
ctx.fillStyle = "rgb(255, 0, 0)"; // Bright red
ctx.strokeStyle = "rgb(0, 0, 255)"; // Blue outline
```

- **HEX (Hexadecimal)**: A compact format, often used in web design.

```
ctx.fillStyle = "#00ff00"; // Bright green
```

- **HSL (Hue, Saturation, Lightness)**: Useful for creating dynamic color changes by adjusting hue.

```
ctx.fillStyle = "hsl(200, 100%, 50%)"; // Vivid cyan-blue
```

You can assign these formats to `fillStyle` (for filling shapes) or `strokeStyle` (for outlines).

Using Alpha Transparency

To make shapes partially see-through, you can use RGBA or HSLA formats, or adjust the `globalAlpha` property.

- **RGBA**: Adds a fourth parameter for alpha (0.0 to 1.0):

```
ctx.fillStyle = "rgba(255, 0, 0, 0.5)"; // Semi-transparent red
```

- **HSLA**:

```
ctx.fillStyle = "hsla(120, 100%, 50%, 0.3)";
```

- **globalAlpha**: Sets transparency for *all* drawing operations:

```
ctx.globalAlpha = 0.6;
ctx.fillRect(50, 50, 100, 100);
ctx.globalAlpha = 1.0; // Reset to opaque
```

Compositing and Blending

Compositing allows you to define how new drawings are layered over existing content using `globalCompositeOperation`.

- **Default mode**: `"source-over"` draws on top.
- **Other modes** include:

- `"destination-over"`: draw behind existing content.
- `"lighter"`: adds color values for glow effects.
- `"multiply"`: multiplies colors for shading.
- `"xor"`: creates interesting cutout effects.

Example:

```
ctx.globalCompositeOperation = "lighter";
ctx.fillStyle = "rgba(255, 0, 0, 0.6)";
ctx.beginPath();
ctx.arc(100, 100, 50, 0, Math.PI * 2);
ctx.fill();
```

Mastering color formats and transparency lets you craft vibrant, layered visual effects. Whether you're blending shadows, animating glowing particles, or adjusting highlights, understanding these tools is key to expressive canvas graphics.

6.2 The Drawing API and Canvas Context

The heart of HTML5 animation and graphics is the **Canvas 2D API**. This API allows you to render shapes, images, text, and more directly onto a drawing surface. To use it, you must first access the **drawing context**, which provides methods for path creation, transformations, and rendering.

Accessing the 2D Context

Start by selecting the `<canvas>` element and retrieving its context:

```
<canvas id="myCanvas" width="600" height="400"></canvas>
```

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

The `ctx` object is the **2D rendering context**, giving you access to all drawing capabilities.

Drawing Workflow: Begin Path Render

The Canvas API uses an immediate-mode graphics model. Here's how the typical rendering pipeline works:

1. **Begin a path**: `ctx.beginPath()`
2. **Define shapes**: `ctx.moveTo()`, `ctx.lineTo()`, `ctx.arc()`, etc.
3. **Render**:
   - `ctx.stroke()` for outlines
   - `ctx.fill()` for solid fills
4. **Repeat as needed**

Example:

```
ctx.beginPath();
ctx.moveTo(100, 100);
ctx.lineTo(200, 100);
ctx.lineTo(200, 200);
ctx.closePath(); // Optionally closes to the starting point
ctx.stroke(); // Draws the triangle outline
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

ctx.fillStyle = "blue";
ctx.strokeStyle = "red";
ctx.beginPath();
ctx.moveTo(100, 100);
ctx.lineTo(200, 100);
ctx.lineTo(200, 200);
ctx.closePath(); // Optionally closes to the starting point
ctx.stroke(); // Draws the triangle outline
</script>
</body>
</html>
```

Clearing the Canvas

Animations often require redrawing each frame from scratch. To clear the canvas, use:

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

This is essential to prevent overlapping frames during animation loops.

Path Construction: Reusable and Efficient

You can define multiple shapes within a single path, or isolate them into separate calls for better control. Use `ctx.save()` and `ctx.restore()` to manage drawing state (e.g., styles, transforms).

```
ctx.save();
ctx.fillStyle = "blue";
ctx.beginPath();
ctx.arc(150, 150, 40, 0, Math.PI * 2);
ctx.fill();
ctx.restore();
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
```

```
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

ctx.fillStyle = "blue";
ctx.strokeStyle = "red";
ctx.save();
ctx.beginPath();
ctx.arc(150, 150, 40, 0, Math.PI * 2);
ctx.fill();
ctx.restore();
</script>
</body>
</html>
```

Immediate Rendering and State

Canvas doesn't retain objects—once something is drawn, it's just pixels. If you need to update or move a shape, you must:

1. Clear the canvas
2. Redraw everything from scratch
3. Animate with a loop (e.g., using `requestAnimationFrame()`)

This model differs from retained-mode systems like SVG, which track each element separately.

Mastering the Canvas API starts with understanding the 2D context and drawing pipeline. With a few foundational functions—`beginPath()`, `moveTo()`, `lineTo()`, `fill()`, `stroke()`, and `clearRect()`—you can create complex and dynamic visual effects, frame by frame.

6.3 Drawing Lines, Curves, and Paths

In canvas-based animation and graphics, the ability to draw precise **lines, curves, and paths** is fundamental. The Canvas 2D API provides low-level methods for constructing these visual elements, allowing you to create everything from simple shapes to intricate vector paths. Let's explore the most common drawing primitives and how to use them effectively.

Lines: The Building Blocks

To draw a line, you first begin a path, move to a starting point, and then draw to a new point using `lineTo()`:

```
ctx.beginPath();
ctx.moveTo(50, 50);     // Start at (50, 50)
ctx.lineTo(200, 50);    // Draw line to (200, 50)
ctx.stroke();           // Render the line
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

ctx.fillStyle = "blue";
ctx.strokeStyle = "red";

ctx.beginPath();
ctx.moveTo(50, 50);     // Start at (50, 50)
ctx.lineTo(200, 50);    // Draw line to (200, 50)
ctx.stroke();           // Render the line
</script>
</body>
</html>
```

You can chain multiple `lineTo()` calls to create polygonal shapes:

```
ctx.beginPath();
ctx.moveTo(100, 100);
ctx.lineTo(200, 100);
ctx.lineTo(200, 200);
ctx.lineTo(100, 200);
ctx.closePath();        // Closes path back to (100, 100)
ctx.stroke();           // Outline the square
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

ctx.fillStyle = "blue";
ctx.strokeStyle = "red";

ctx.beginPath();
ctx.moveTo(100, 100);
ctx.lineTo(200, 100);
ctx.lineTo(200, 200);
ctx.lineTo(100, 200);
ctx.closePath();        // Closes path back to (100, 100)
ctx.stroke();           // Outline the square
</script>
</body>
</html>
```

Arcs and Circles

The `arc()` method allows you to draw circular or partial arc segments:

```
ctx.beginPath();
ctx.arc(150, 150, 75, 0, Math.PI * 2); // Full circle
ctx.stroke();
```

- Parameters:
  - `(x, y)`: Center point
  - `radius`: Radius of the arc
  - `startAngle`, `endAngle`: In **radians**
  - Optional `anticlockwise`: Boolean (default is `false`)

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

ctx.beginPath();
ctx.arc(150, 150, 75, 0, Math.PI * 2); // Full circle
ctx.stroke();
</script>
</body>
</html>
```

To draw a semicircle:

```
ctx.beginPath();
ctx.arc(300, 150, 75, 0, Math.PI); // Half circle
ctx.stroke();
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");
ctx.fillStyle = "blue";
ctx.strokeStyle = "red";

ctx.beginPath();
ctx.arc(300, 150, 75, 0, Math.PI); // Half circle
ctx.stroke();
</script>
</body>
</html>
```

Quadratic Bezier Curves

Quadratic curves use a single control point to bend the line between start and end points. Use `quadraticCurveTo()`:

```
ctx.beginPath();
ctx.moveTo(50, 250);                    // Start point
ctx.quadraticCurveTo(150, 100, 250, 250); // Control point, End point
ctx.stroke();
```

This creates a gentle curve, useful for waves or rounded edges.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");
ctx.fillStyle = "blue";
ctx.strokeStyle = "red";

ctx.beginPath();
ctx.moveTo(50, 250);                    // Start point
ctx.quadraticCurveTo(150, 100, 250, 250); // Control point, End point
ctx.stroke();
</script>
</body>
</html>
```

Cubic Bezier Curves

Cubic Bezier curves offer more control with two control points, using `bezierCurveTo()`:

```
ctx.beginPath();
ctx.moveTo(50, 300);
ctx.bezierCurveTo(150, 100, 250, 500, 350, 300);
ctx.stroke();
```

These are ideal for complex curves like swooshes, calligraphy, or animation easing paths.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");


ctx.beginPath();
ctx.fillStyle = "blue";
ctx.strokeStyle = "red";
```

```
ctx.moveTo(50, 300);
ctx.bezierCurveTo(150, 100, 250, 500, 350, 300);
ctx.stroke();
</script>
</body>
</html>
```

Complex Paths and Shapes

You can mix all drawing primitives in one path to build intricate shapes. Use `closePath()` when you want to connect the last point back to the start:

```
ctx.beginPath();
ctx.moveTo(100, 100);
ctx.lineTo(150, 50);
ctx.arcTo(200, 50, 200, 100, 50); // Smooth corner arc
ctx.lineTo(200, 150);
ctx.closePath();
ctx.stroke();
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");
ctx.fillStyle = "blue";
ctx.strokeStyle = "red";

ctx.beginPath();
ctx.moveTo(100, 100);
ctx.lineTo(150, 50);
ctx.arcTo(200, 50, 200, 100, 50); // Smooth corner arc
ctx.lineTo(200, 150);
ctx.closePath();
ctx.stroke();
</script>
</body>
</html>
```

Styling the Paths

- `ctx.lineWidth = 5` — sets thickness.
- `ctx.strokeStyle = 'red'` — sets line color.
- `ctx.lineJoin` and `ctx.lineCap` — control corner and end style.

Example:

```
ctx.lineWidth = 4;
ctx.strokeStyle = "#00f";
ctx.lineCap = "round";
ctx.lineJoin = "round";
```

By combining lines, arcs, and Bezier curves, you can create anything from basic wireframes to expressive illustrations. These path-building tools are essential for constructing dynamic visuals and animated transformations in canvas-based applications.

6.4 Filling Shapes and Creating Gradients

Once you've created shapes using paths on the canvas, adding color fills transforms simple outlines into vibrant, visually appealing graphics. The Canvas 2D API supports several filling techniques: solid colors, patterns, and gradients—each unlocking different creative possibilities.

Filling Shapes with Solid Colors

The simplest way to fill a shape is by setting the `fillStyle` property to a color string:

```
ctx.fillStyle = "#3498db"; // Solid blue color
ctx.beginPath();
ctx.rect(50, 50, 150, 100);
ctx.fill();
```

The `fill()` method fills the current path with the specified color, creating solid shapes like buttons, backgrounds, or character parts.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

ctx.fillStyle = "blue";
ctx.strokeStyle = "red";

ctx.beginPath();
ctx.rect(50, 50, 150, 100);
ctx.fill();
</script>
</body>
</html>
```

Using Patterns for Texture

Patterns enable filling shapes with repeating images or textures, useful for more complex or natural-looking fills like bricks, wood grain, or fabric:

```
const img = new Image();
img.src = "https://readbytes.github.io/images/60x60/1.png";
```

```
img.onload = () => {
  const pattern = ctx.createPattern(img, "repeat"); // Options: repeat, repeat-x, repeat-y, no-repeat
  ctx.fillStyle = pattern;
  ctx.beginPath();
  ctx.arc(200, 200, 75, 0, Math.PI * 2);
  ctx.fill();
};
```

Patterns repeat the source image across the fill area, adding rich texture without complex drawing.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

const img = new Image();
img.src = "https://readbytes.github.io/images/60x60/1.png";

img.onload = () => {
  const pattern = ctx.createPattern(img, "repeat"); // Options: repeat, repeat-x, repeat-y, no-repeat
  ctx.fillStyle = pattern;
  ctx.beginPath();
  ctx.arc(200, 200, 75, 0, Math.PI * 2);
  ctx.fill();
};
</script>
</body>
</html>
```

Linear Gradients

Linear gradients create a smooth color transition along a straight line, useful for shading, lighting effects, or adding depth:

```
const gradient = ctx.createLinearGradient(50, 50, 200, 50);
gradient.addColorStop(0, "red");       // Start color
gradient.addColorStop(0.5, "yellow");  // Middle color
gradient.addColorStop(1, "green");     // End color

ctx.fillStyle = gradient;
ctx.fillRect(50, 50, 150, 100);
```

You define the gradient by specifying start and end coordinates, then add multiple color stops between 0 and 1 to control the blending.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
```

```
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

const gradient = ctx.createLinearGradient(50, 50, 200, 50);
gradient.addColorStop(0, "red");       // Start color
gradient.addColorStop(0.5, "yellow");  // Middle color
gradient.addColorStop(1, "green");     // End color

ctx.fillStyle = gradient;
ctx.fillRect(50, 50, 150, 100);
</script>
</body>
</html>
```

Radial Gradients

Radial gradients radiate colors outward from a central point, ideal for glow effects, spotlighting, or simulating spherical lighting:

```
const radialGradient = ctx.createRadialGradient(300, 150, 20, 300, 150, 75);
radialGradient.addColorStop(0, "white");
radialGradient.addColorStop(1, "blue");

ctx.fillStyle = radialGradient;
ctx.beginPath();
ctx.arc(300, 150, 75, 0, Math.PI * 2);
ctx.fill();
```

Parameters define the inner circle (center and radius) and the outer circle, creating a smooth radial color transition.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

const radialGradient = ctx.createRadialGradient(300, 150, 20, 300, 150, 75);
radialGradient.addColorStop(0, "white");
radialGradient.addColorStop(1, "blue");

ctx.fillStyle = radialGradient;
ctx.beginPath();
ctx.arc(300, 150, 75, 0, Math.PI * 2);
ctx.fill();
</script>
</body>
</html>
```

Practical Use Cases

- **Buttons and UI Elements**: Gradients can simulate lighting and depth.
- **Natural Textures**: Patterns recreate surfaces like wood or fabric.
- **Lighting and Shadows**: Radial gradients produce soft glows and highlight effects.
- **Backgrounds**: Solid or gradient fills help set mood and visual hierarchy.

By combining these fill techniques, you can create rich, engaging visuals that enhance your animations and interactive experiences. Experimenting with colors, stops, and patterns unlocks endless creative possibilities on the canvas.

Chapter 7.

Drawing on the Canvas 2

1. Image and Video Rendering
2. Pixel Manipulation Techniques

7 Drawing on the Canvas 2

7.1 Image and Video Rendering

The HTML5 Canvas API not only supports drawing vector shapes but also allows you to render **images** and **videos** directly onto the canvas. This capability enables rich multimedia animations, combining bitmaps with dynamic drawings to create complex scenes, game sprites, and interactive visuals.

Drawing Images

The simplest way to draw an image is by using the `drawImage()` method:

```
const img = new Image();
img.src = "sprite.png";

img.onload = () => {
  ctx.drawImage(img, 50, 50);
};
```

This draws the image at coordinates (50, 50) at its original size.

Scaling and Positioning Images

You can specify destination width and height to scale the image:

```
ctx.drawImage(img, 50, 50, 100, 100); // Draw image resized to 100x100
```

This is useful for sprite animation or fitting images into a specific layout.

Cropping Images (Sprite Sheets)

To draw part of an image (e.g., a single frame from a sprite sheet), `drawImage` accepts source rectangle parameters:

```
ctx.drawImage(
  img,
  sx, sy, sw, sh,   // Source rectangle in the image
  dx, dy, dw, dh    // Destination rectangle on canvas
);
```

For example, to draw a 64x64 pixel sprite from coordinates (128, 0) on the source image to position (200, 100) scaled to 64x64:

```
ctx.drawImage(img, 128, 0, 64, 64, 200, 100, 64, 64);
```

Drawing Video Frames

You can also render live frames from a `<video>` element onto the canvas, enabling custom video effects or overlay animations:

```
<video id="video" src="video.mp4" autoplay muted loop></video>
```

```
const video = document.getElementById("video");
```

```
function drawVideoFrame() {
  ctx.drawImage(video, 0, 0, canvas.width, canvas.height);
  requestAnimationFrame(drawVideoFrame);
}

video.addEventListener("play", drawVideoFrame);
```

This continuously draws the current video frame onto the canvas, synchronized with the browser's rendering.

Combining with Other Drawings

Images and videos can be layered with other shapes, lines, or text on the canvas. For example, you might draw a video background and then overlay animated graphics:

```
ctx.drawImage(video, 0, 0, canvas.width, canvas.height);

// Draw overlay circle
ctx.beginPath();
ctx.arc(150, 150, 50, 0, Math.PI * 2);
ctx.fillStyle = "rgba(255, 0, 0, 0.5)";
ctx.fill();
```

Practical Tips

- Wait for images/videos to fully load before drawing to avoid errors.
- Use `requestAnimationFrame()` to synchronize rendering with the browser's refresh rate.
- Consider performance impacts of scaling large images or videos.
- Combine transformations (`translate`, `rotate`, `scale`) with images to create dynamic motion and effects.

By mastering image and video rendering on canvas, you can blend bitmap media seamlessly with vector graphics, opening up limitless possibilities for interactive, multimedia-rich web experiences.

7.2 Pixel Manipulation Techniques

One of the powerful features of the HTML5 Canvas API is the ability to access and manipulate image pixels directly. This pixel-level control allows you to create custom effects, apply filters, and transform colors dynamically—giving you fine-grained control over your animations and graphics.

Accessing Pixel Data: `getImageData`

The `getImageData()` method retrieves the color data for a rectangular area on the canvas. It returns an `ImageData` object containing a `data` property—an array of color values for every pixel.

```
const imageData = ctx.getImageData(x, y, width, height);
const pixels = imageData.data;
```

- `pixels` is a `Uint8ClampedArray` where every group of four values represents one pixel's RGBA components:
  - `pixels[0]` = red
  - `pixels[1]` = green
  - `pixels[2]` = blue
  - `pixels[3]` = alpha (opacity)

For example, the first pixel uses indices 0 to 3, the second pixel 4 to 7, and so on.

Modifying Pixels

You can loop through the pixel array and alter colors to create effects like grayscale, color shifts, or transparency changes.

Example: converting an image region to grayscale:

```
for (let i = 0; i < pixels.length; i += 4) {
  let r = pixels[i];
  let g = pixels[i + 1];
  let b = pixels[i + 2];
  let avg = (r + g + b) / 3;
  pixels[i] = pixels[i + 1] = pixels[i + 2] = avg; // Set all to average
}
```

Applying Changes: `putImageData`

After modifying the pixel array, you update the canvas by putting the changed data back using `putImageData()`:

```
ctx.putImageData(imageData, x, y);
```

This redraws the specified rectangle with your pixel changes applied.

Common Pixel Effects

- **Filters**: Apply blur, sharpen, or edge detection by manipulating pixels based on neighbors.
- **Color transformations**: Adjust brightness, contrast, or apply color tints by changing RGB values.
- **Transparency and alpha masking**: Modify alpha channel for fade or masking effects.
- **Custom distortions**: Create ripple or wave effects by repositioning pixels before drawing.

Performance Considerations

Pixel manipulation can be computationally intensive, especially on large areas or high frame rates. To optimize:

- Limit the size of the area you process.

- Cache results when possible.
- Perform expensive calculations outside animation loops.
- Use Web Workers for complex processing without freezing the UI.

Example: Inverting Colors

Here's a quick example that inverts the colors in a selected canvas region:

```
const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
const pixels = imageData.data;

for (let i = 0; i < pixels.length; i += 4) {
  pixels[i] = 255 - pixels[i];       // Red
  pixels[i + 1] = 255 - pixels[i + 1]; // Green
  pixels[i + 2] = 255 - pixels[i + 2]; // Blue
}

ctx.putImageData(imageData, 0, 0);
```

By mastering pixel manipulation, you unlock the potential to craft unique, dynamic visual effects that go beyond basic drawing commands. This fine-grained control is especially valuable for creative animations, interactive filters, and game graphics.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Invert Canvas Colors</title>
  <style>
    body {
      margin: 0;
      background: #222;
      color: white;
      display: flex;
      flex-direction: column;
      align-items: center;
      justify-content: center;
      height: 100vh;
      font-family: sans-serif;
    }
    canvas {
      border: 2px solid #fff;
    }
    button {
      margin-top: 20px;
      padding: 10px 20px;
      font-size: 16px;
      background: #444;
      color: white;
      border: none;
      cursor: pointer;
    }
    button:hover {
      background: #666;
    }
```

```
  </style>
</head>
<body>
  <canvas id="canvas" width="400" height="300"></canvas>
  <button id="invertBtn">Invert Colors</button>

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

    // Draw something on the canvas
    function drawInitialScene() {
      // Background
      ctx.fillStyle = "skyblue";
      ctx.fillRect(0, 0, canvas.width, canvas.height);

      // A sun
      ctx.beginPath();
      ctx.arc(300, 100, 40, 0, Math.PI * 2);
      ctx.fillStyle = "yellow";
      ctx.fill();

      // A rectangle (house)
      ctx.fillStyle = "sienna";
      ctx.fillRect(100, 150, 100, 100);

      // Roof
      ctx.beginPath();
      ctx.moveTo(100, 150);
      ctx.lineTo(150, 100);
      ctx.lineTo(200, 150);
      ctx.closePath();
      ctx.fillStyle = "maroon";
      ctx.fill();
    }

    drawInitialScene();

    document.getElementById("invertBtn").addEventListener("click", () => {
      const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
      const pixels = imageData.data;

      for (let i = 0; i < pixels.length; i += 4) {
        pixels[i] = 255 - pixels[i];         // Red
        pixels[i + 1] = 255 - pixels[i + 1]; // Green
        pixels[i + 2] = 255 - pixels[i + 2]; // Blue
        // Alpha (pixels[i + 3]) is left unchanged
      }

      ctx.putImageData(imageData, 0, 0);
    });
  </script>
</body>
</html>
```

Chapter 8.

Velocity and Acceleration 1

1. Introduction to Velocity and Vectors
2. Velocity on One and Two Axes
3. Angular Velocity and Vector Addition
4. Mouse Following Behavior

8 Velocity and Acceleration 1

8.1 Introduction to Velocity and Vectors

In animation and physics, **velocity** is more than just speed; it's a vector quantity that combines both how fast an object is moving and the direction in which it travels. Understanding velocity as a vector is essential for creating realistic, dynamic motion in 2D animations.

What Is Velocity?

Velocity describes the rate of change of an object's position over time, **including direction**. For example, a ball rolling to the right at 5 pixels per second has a velocity of 5 pixels/second toward the right, while if it's rolling left at the same speed, its velocity is the same magnitude but opposite direction.

Velocity can be positive or negative along an axis, depending on direction, and must always be described with both magnitude (speed) and direction.

Introducing Vectors

A **vector** is a quantity that has both **magnitude** (length or size) and **direction**. In 2D animation, vectors are usually represented as arrows pointing from one point to another.

Vectors are often written in component form as:

$$\mathbf{v} = (v_x, v_y)$$

where:

- v_x is the velocity component along the x-axis (horizontal),
- v_y is the velocity component along the y-axis (vertical).

Together, these components tell you both how fast and in which direction the object moves.

Visualizing Velocity Vectors

Imagine an object at position (x, y) moving with velocity vector $\mathbf{v} = (3, 4)$. This means the object moves 3 units right and 4 units down (or up, depending on your coordinate system) every time step.

You can visualize this velocity as an arrow starting at the object's position, extending 3 units horizontally and 4 units vertically. The **length** of this arrow represents the speed:

$$\text{speed} = |\mathbf{v}| = \sqrt{v_x^2 + v_y^2} = \sqrt{3^2 + 4^2} = 5$$

So the speed is 5 units per time interval, moving diagonally.

Why Are Vectors Useful in Animation?

Using velocity vectors lets you:

- Move objects smoothly in any direction by updating positions:

$$x_{\text{new}} = x + v_x, \quad y_{\text{new}} = y + v_y$$

- Combine multiple motions by adding velocity vectors (vector addition).
- Easily handle reflections, rotations, and direction changes.
- Model realistic physics like bouncing, gravity, and friction.

Simple Example in Code

```
let position = { x: 100, y: 100 };
let velocity = { x: 2, y: -3 }; // moves right and up

function update() {
  position.x += velocity.x;
  position.y += velocity.y;
  // draw object at new position
}
```

This example moves an object diagonally by adding the velocity vector components to its position every frame.

Understanding velocity as a vector opens the door to creating natural, controllable motion in your animations. It forms the foundation for more advanced concepts like acceleration, forces, and angular velocity covered later in this chapter.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Simple Motion Example</title>
  <style>
    body {
      margin: 0;
      background: #111;
    }
    canvas {
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas"></canvas>

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

```
    function resizeCanvas() {
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
    }
    window.addEventListener("resize", resizeCanvas);
    resizeCanvas();

    let position = { x: 100, y: 100 };
    let velocity = { x: 2, y: -3 }; // moves right and up

    function update() {
      // Update position
      position.x += velocity.x;
      position.y += velocity.y;

      // Bounce off edges
      if (position.x <= 0 || position.x >= canvas.width) velocity.x *= -1;
      if (position.y <= 0 || position.y >= canvas.height) velocity.y *= -1;

      // Clear and draw
      ctx.clearRect(0, 0, canvas.width, canvas.height);
      ctx.beginPath();
      ctx.arc(position.x, position.y, 20, 0, Math.PI * 2);
      ctx.fillStyle = "cyan";
      ctx.fill();

      requestAnimationFrame(update);
    }

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

8.2 Velocity on One and Two Axes

Velocity describes how an object's position changes over time. In animation, understanding velocity along one or two axes is essential for controlling motion precisely.

Velocity on One Axis (1D)

In one-dimensional (1D) motion, velocity affects position along a single axis — usually the **x-axis** or **y-axis**. For example, imagine a ball rolling along a straight horizontal line.

If the velocity along the x-axis is v_x, then the new position x_{new} after a time step Δt is:

$$x_{\text{new}} = x_{\text{old}} + v_x \times \Delta t$$

In most simple animations, Δt is treated as 1 frame unit, so we often simplify this to:

```
x += vx;
```

```
let x = 50;       // Initial position
let vx = 3;       // Velocity in pixels per frame

function update() {
  x += vx;        // Move right by 3 pixels each frame
  drawObjectAt(x, 100);
}
```

Example: Moving Along the X-Axis This moves an object horizontally to the right at a constant speed of 3 pixels per frame.

Velocity on Two Axes (2D)

In two-dimensional (2D) motion, velocity has two components: one along the x-axis (v_x) and one along the y-axis (v_y). The position updates on both axes independently:

$$x_{\text{new}} = x_{\text{old}} + v_x \times \Delta t$$

$$y_{\text{new}} = y_{\text{old}} + v_y \times \Delta t$$

Again, with $\Delta t = 1$, the code becomes:

```
x += vx;
y += vy;
```

```
let position = { x: 50, y: 50 };
let velocity = { x: 2, y: -1 }; // Move right and up

function update() {
  position.x += velocity.x;
  position.y += velocity.y;
  drawObjectAt(position.x, position.y);
}
```

Example: Moving Along X and Y Axes This moves an object diagonally, 2 pixels to the right and 1 pixel upward per frame.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Move Along X and Y</title>
  <style>
    body {
      margin: 0;
      background: #111;
```

```
    }
    canvas {
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas"></canvas>

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

    function resizeCanvas() {
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
    }
    window.addEventListener("resize", resizeCanvas);
    resizeCanvas();

    let position = { x: 50, y: 50 };
    let velocity = { x: 2, y: -1 }; // Move right and up

    function drawObjectAt(x, y) {
      ctx.beginPath();
      ctx.arc(x, y, 15, 0, Math.PI * 2);
      ctx.fillStyle = "orange";
      ctx.fill();
    }

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

      position.x += velocity.x;
      position.y += velocity.y;

      // Bounce off walls
      if (position.x < 0 || position.x > canvas.width) velocity.x *= -1;
      if (position.y < 0 || position.y > canvas.height) velocity.y *= -1;

      drawObjectAt(position.x, position.y);

      requestAnimationFrame(update);
    }

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

Calculating Speed and Direction

To find the **speed** (magnitude) of the velocity vector:

$$\text{speed} = \sqrt{v_x^2 + v_y^2}$$

The **direction** or angle θ of motion relative to the x-axis is:

$$\theta = \arctan\left(\frac{v_y}{v_x}\right)$$

Knowing speed and direction is useful for physics simulations and for setting velocity vectors based on desired motion angles.

Why Separate X and Y Components?

Separating velocity into x and y components lets you:

- Animate movement in any direction.
- Independently control horizontal and vertical speeds.
- Easily add forces like gravity (vertical acceleration).
- Handle collisions and bouncing on each axis separately.

Putting It All Together: Position Update Loop

A typical animation loop updating 2D position with velocity looks like this:

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

  // Update position
  position.x += velocity.x;
  position.y += velocity.y;

  // Draw the object
  drawObjectAt(position.x, position.y);

  requestAnimationFrame(animate);
}
```

By mastering velocity on one and two axes, you gain the ability to move objects smoothly and realistically across the screen, forming the foundation for more complex motion behavior covered later.

8.3 Angular Velocity and Vector Addition

When working with animations, understanding both **angular velocity**—the rate of rotation—and **vector addition**—the combination of multiple velocities—is crucial for creating realistic, dynamic motion.

What Is Angular Velocity?

Angular velocity describes how fast an object rotates around a pivot point. It is typically measured in **radians per second** (or degrees per second), representing how much the object turns in a unit of time.

For example, if a wheel spins at $\omega = \pi$ radians per second, it completes half a rotation every second because one full rotation is 2π radians.

Applying Angular Velocity in Animation

To animate rotation, you update the object's **angle** over time by adding angular velocity each frame:

$$\theta_{\text{new}} = \theta_{\text{old}} + \omega \times \Delta t$$

If Δt is one frame (commonly 1/60th of a second), then:

```
angle += angularVelocity; // angle in radians
```

This updated angle is then used to draw or transform the object rotated by θ.

```
let angle = 0;
let angularVelocity = 0.05; // radians per frame

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

  ctx.save();
  ctx.translate(centerX, centerY);
  ctx.rotate(angle);
  drawShape(); // Draw your object centered at origin
  ctx.restore();

  angle += angularVelocity; // Update rotation

  requestAnimationFrame(animate);
}
```

Simple Rotation Example Here, the object smoothly rotates around its center.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Simple Rotation Example</title>
  <style>
    body {
      margin: 0;
      background: #111;
    }
    canvas {
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas"></canvas>
```

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

    function resizeCanvas() {
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
    }
    window.addEventListener("resize", resizeCanvas);
    resizeCanvas();

    const centerX = canvas.width / 2;
    const centerY = canvas.height / 2;
    let angle = 0;
    let angularVelocity = 0.05; // radians per frame

    function drawShape() {
      ctx.fillStyle = "lime";
      ctx.fillRect(-50, -25, 100, 50); // Draw centered rectangle
    }

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

      ctx.save();
      ctx.translate(centerX, centerY);
      ctx.rotate(angle);
      drawShape();
      ctx.restore();

      angle += angularVelocity;

      requestAnimationFrame(animate);
    }

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

Vector Addition: Combining Velocities

In 2D motion, objects often have multiple velocity components acting simultaneously—like moving diagonally or combining wind and thrust forces. These velocities combine through **vector addition**.

If two velocity vectors are:

$$\mathbf{v_1} = (v_{1x}, v_{1y}), \quad \mathbf{v_2} = (v_{2x}, v_{2y})$$

Their resultant velocity $\mathbf{v}$ is:

$$\mathbf{v} = \mathbf{v_1} + \mathbf{v_2} = (v_{1x} + v_{2x}, \quad v_{1y} + v_{2y})$$

Practical Use: Combining Movement Directions

Imagine a spaceship affected by:

- A forward thrust vector pushing it upward $\mathbf{v_1} = (0, -5)$
- A wind pushing rightward $\mathbf{v_2} = (3, 0)$

The total velocity is:

$$\mathbf{v} = (0 + 3, -5 + 0) = (3, -5)$$

So the spaceship moves diagonally up-right.

```
let velocity1 = { x: 0, y: -5 };
let velocity2 = { x: 3, y: 0 };

let combinedVelocity = {
  x: velocity1.x + velocity2.x,
  y: velocity1.y + velocity2.y,
};

position.x += combinedVelocity.x;
position.y += combinedVelocity.y;
```

Vector Addition Example in Code Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Vector Addition Example</title>
  <style>
    body {
      margin: 0;
      background: #222;
    }
    canvas {
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas"></canvas>

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

    function resizeCanvas() {
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
    }
    window.addEventListener("resize", resizeCanvas);
    resizeCanvas();
```

```
let position = { x: canvas.width / 2, y: canvas.height / 2 };
let velocity1 = { x: 0, y: -5 };
let velocity2 = { x: 3, y: 0 };

function drawObject(x, y) {
  ctx.beginPath();
  ctx.arc(x, y, 20, 0, Math.PI * 2);
  ctx.fillStyle = "cyan";
  ctx.fill();

  // Draw velocity vectors for visualization
  ctx.strokeStyle = "yellow";
  ctx.lineWidth = 2;

  // velocity1 vector (red)
  ctx.beginPath();
  ctx.moveTo(x, y);
  ctx.lineTo(x + velocity1.x * 10, y + velocity1.y * 10);
  ctx.strokeStyle = "red";
  ctx.stroke();

  // velocity2 vector (green)
  ctx.beginPath();
  ctx.moveTo(x, y);
  ctx.lineTo(x + velocity2.x * 10, y + velocity2.y * 10);
  ctx.strokeStyle = "green";
  ctx.stroke();

  // combinedVelocity vector (yellow)
  ctx.beginPath();
  ctx.moveTo(x, y);
  ctx.lineTo(x + combinedVelocity.x * 10, y + combinedVelocity.y * 10);
  ctx.strokeStyle = "yellow";
  ctx.stroke();
}

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

  // Calculate combined velocity
  combinedVelocity = {
    x: velocity1.x + velocity2.x,
    y: velocity1.y + velocity2.y,
  };

  // Update position
  position.x += combinedVelocity.x;
  position.y += combinedVelocity.y;

  // Bounce off edges
  if (position.x < 20 || position.x > canvas.width - 20) {
    velocity1.x *= -1;
    velocity2.x *= -1;
  }
  if (position.y < 20 || position.y > canvas.height - 20) {
    velocity1.y *= -1;
    velocity2.y *= -1;
  }
```

```
      drawObject(position.x, position.y);

      requestAnimationFrame(update);
    }

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

Visualizing Vector Addition

Graphically, vector addition corresponds to placing the tail of the second vector at the head of the first and drawing the resultant vector from the tail of the first to the head of the second.

This technique helps you think intuitively about combining forces, velocities, or motions in your animations.

Run the following code in browser to see the demo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Visualizing Vector Addition</title>
  <style>
    body {
      margin: 0;
      background: #222;
      color: #eee;
      font-family: Arial, sans-serif;
      display: flex;
      flex-direction: column;
      align-items: center;
      padding: 20px;
      user-select: none;
    }
    h2 {
      margin-bottom: 10px;
    }
    svg {
      border: 1px solid #555;
      background: #444;
      margin-top: 10px;
      width: 600px;
      height: 400px;
    }
    text {
      font-size: 14px;
      fill: #eee;
      font-weight: bold;
    }
    .vector-label {
      font-weight: normal;
      fill: #ccc;
```

```
      font-size: 12px;
    }
    .arrow {
      stroke-width: 3;
      fill: none;
      stroke:rgb(47, 121, 233);
      marker-end: url(#arrowhead);
    }
    .v1 {
      stroke: #e74c3c; /* red */
    }
    .v2 {
      stroke: #2ecc71; /* green */
      stroke-dasharray: 10 5;
    }
    .result {
      stroke: #f1c40f; /* yellow */
    }
  </style>
</head>
<body>
  <h2>Visualizing Vector Addition</h2>
  <p>Vector addition: place tail of second vector at head of first, then draw resultant from tail of fi

  <svg viewBox="0 0 600 400" xmlns="http://www.w3.org/2000/svg" aria-label="Vector addition diagram">
    <!-- Arrowhead marker definition -->
    <defs>
      <marker id="arrowhead" markerWidth="10" markerHeight="7"
              refX="10" refY="3.5" orient="auto" fill="inherit">
        <polygon points="0 0, 10 3.5, 0 7" />
      </marker>
    </defs>

    <!-- Axes -->
    <line x1="50" y1="350" x2="550" y2="350" stroke="#555" stroke-width="1" />
    <line x1="50" y1="350" x2="50" y2="50" stroke="#555" stroke-width="1" />
    <text x="555" y="355" fill="#888" font-size="12">X</text>
    <text x="40" y="45" fill="#888" font-size="12">Y</text>

    <!-- Origin point -->
    <circle cx="50" cy="350" r="4" fill="#fff" />
    <text x="30" y="370" fill="#eee">O (0,0)</text>

    <!-- Vector 1 -->
    <line x1="50" y1="350" x2="250" y2="200" class="arrow v1" />
    <text x="150" y="270" class="vector-label">v </text>
    <circle cx="250" cy="200" r="5" fill="#e74c3c" />

    <!-- Vector 2 (placed tail at head of v1) -->
    <line x1="250" y1="200" x2="450" y2="300" class="arrow v2" />
    <text x="360" y="230" class="vector-label">v </text>
    <circle cx="450" cy="300" r="5" fill="#2ecc71" />

    <!-- Resultant vector -->
    <line x1="50" y1="350" x2="450" y2="300" class="arrow result" />
    <text x="240" y="340" class="vector-label">v  + v </text>
    <circle cx="450" cy="300" r="5" fill="#f1c40f" />
```

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

8.3.1 Summary

- **Angular velocity** controls how fast objects rotate, updated by incrementing an angle each frame.
- **Vector addition** combines multiple velocity vectors into a single resultant motion.
- Together, these concepts allow you to animate rotations and complex directional movements such as swirling particles, spinning wheels, or objects affected by multiple forces.

Mastering angular velocity and vector addition opens the door to smooth, natural, and physically inspired animations.

8.4 Mouse Following Behavior

Creating animations where objects smoothly follow the mouse cursor adds a dynamic and interactive feel to your projects. This behavior involves updating an object's velocity vector to "chase" the cursor's position, combined with smoothing techniques to avoid abrupt jumps and create natural, easing motion.

Basic Concept

To make an object follow the mouse, you calculate a **velocity vector** pointing from the object's current position toward the mouse position. Instead of instantly snapping to the cursor, the object moves gradually by adjusting its velocity, creating a smooth following effect.

Step 1: Track Mouse Position

First, listen for mouse movement events and store the current mouse coordinates.

```
let mouse = { x: 0, y: 0 };

canvas.addEventListener('mousemove', (event) => {
  mouse.x = event.clientX;
  mouse.y = event.clientY;
});
```

Step 2: Calculate Direction and Velocity

In each animation frame, calculate the difference vector between the mouse and object positions:

$$\Delta x = x_{\text{mouse}} - x_{\text{object}}$$

$$\Delta y = y_{\text{mouse}} - y_{\text{object}}$$

This vector points toward the cursor.

Step 3: Apply Smoothing with Easing

Instead of moving the object directly to the mouse position, multiply the difference vector by a small **easing factor** (e.g., 0.1) to gradually move closer each frame.

```
let position = { x: 100, y: 100 };
const easing = 0.1;

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

  // Calculate difference vector
  let dx = mouse.x - position.x;
  let dy = mouse.y - position.y;

  // Apply easing to velocity
  position.x += dx * easing;
  position.y += dy * easing;

  // Draw the object at updated position
  drawObjectAt(position.x, position.y);

  requestAnimationFrame(animate);
}
```

Why Easing?

Easing smooths movement by reducing velocity as the object nears the target. Without easing, the object would jump or overshoot the cursor, resulting in unnatural motion.

Step 4: Enhancing with Velocity Vectors

For more control, you can explicitly calculate velocity vectors, incorporating acceleration and friction for realistic effects:

```
let velocity = { x: 0, y: 0 };
const accelerationFactor = 0.2;
const friction = 0.8;

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

  let dx = mouse.x - position.x;
  let dy = mouse.y - position.y;

  // Accelerate toward mouse
  velocity.x += dx * accelerationFactor;
  velocity.y += dy * accelerationFactor;
```

```
  // Apply friction to slow down velocity
  velocity.x *= friction;
  velocity.y *= friction;

  // Update position with velocity
  position.x += velocity.x;
  position.y += velocity.y;

  drawObjectAt(position.x, position.y);

  requestAnimationFrame(animate);
}
```

This approach creates a more fluid, natural "chasing" motion as the object speeds up and slows down smoothly.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Smooth Mouse Follow with Easing & Velocity</title>
  <style>
    body {
      margin: 0;
      background: #111;
      overflow: hidden;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      color: #eee;
      font-family: Arial, sans-serif;
      user-select: none;
      flex-direction: column;
    }
    canvas {
      border: 2px solid #444;
      background: #222;
      display: block;
    }
    #modeBtn {
      margin-top: 15px;
      padding: 8px 16px;
      background: #444;
      border: none;
      color: #eee;
      font-size: 16px;
      cursor: pointer;
      border-radius: 4px;
    }
    #modeBtn:hover {
      background: #666;
    }
  </style>
</head>
```

```
<body>

  <canvas id="canvas" width="600" height="400"></canvas>
  <button id="modeBtn">Switch to Velocity Mode</button>

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

    // Track mouse position
    let mouse = { x: canvas.width / 2, y: canvas.height / 2 };
    canvas.addEventListener('mousemove', (e) => {
      mouse.x = e.clientX;
      mouse.y = e.clientY;
    });

    // Object position & parameters
    let position = { x: canvas.width / 2, y: canvas.height / 2 };
    let velocity = { x: 0, y: 0 };
    const easing = 0.1;
    const accelerationFactor = 0.2;
    const friction = 0.8;

    // Mode toggle: true = velocity mode, false = easing mode
    let velocityMode = false;
    const modeBtn = document.getElementById("modeBtn");

    modeBtn.addEventListener("click", () => {
      velocityMode = !velocityMode;
      modeBtn.textContent = velocityMode ? "Switch to Easing Mode" : "Switch to Velocity Mode";

      // Reset position and velocity on mode switch
      position = { x: canvas.width / 2, y: canvas.height / 2 };
      velocity = { x: 0, y: 0 };
    });

    function drawObjectAt(x, y) {
      ctx.beginPath();
      ctx.arc(x, y, 15, 0, Math.PI * 2);
      ctx.fillStyle = velocityMode ? "#f39c12" : "#3498db";
      ctx.shadowColor = ctx.fillStyle;
      ctx.shadowBlur = 15;
      ctx.fill();
      ctx.shadowBlur = 0;
    }

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

      let dx = mouse.x - position.x;
      let dy = mouse.y - position.y;

      if (!velocityMode) {
        // Easing mode: smooth interpolation toward mouse
        position.x += dx * easing;
        position.y += dy * easing;
      } else {
        // Velocity mode: accelerate toward mouse and apply friction
```

```
        velocity.x += dx * accelerationFactor;
        velocity.y += dy * accelerationFactor;

        velocity.x *= friction;
        velocity.y *= friction;

        position.x += velocity.x;
        position.y += velocity.y;
      }

      drawObjectAt(position.x, position.y);
      requestAnimationFrame(animate);
    }

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

8.4.1 Summary

- Use the difference between mouse and object positions to calculate direction.
- Apply easing or velocity vectors to smooth motion and prevent abrupt jumps.
- Adjust easing and friction factors to customize the responsiveness and feel of the following behavior.

By combining velocity vectors with easing, you create animations that feel alive and responsive—perfect for interactive web experiences where users control elements simply by moving their mouse.

Chapter 9.

Velocity and Acceleration 2

1. Introduction to Acceleration
2. Gravity and Angular Acceleration
3. Spaceship Motion and Controls

9 Velocity and Acceleration 2

9.1 Introduction to Acceleration

In animation and physics, **acceleration** is a fundamental concept that controls how objects speed up, slow down, or change direction over time. Simply put, acceleration is the **rate of change of velocity**—how quickly an object's velocity increases or decreases.

What Is Acceleration?

While velocity tells us how fast and in what direction an object moves, acceleration tells us how that velocity changes. If an object's velocity changes from 2 pixels/frame to 5 pixels/frame over time, it is accelerating. Conversely, if velocity decreases, the object is **decelerating** or experiencing negative acceleration.

Mathematically, acceleration **a** is defined as:

$$\mathbf{a} = \frac{\Delta \mathbf{v}}{\Delta t}$$

where:

- $\Delta \mathbf{v}$ is the change in velocity vector,
- Δt is the change in time.

Why Is Acceleration Important in Animation?

Acceleration enables more realistic and dynamic movement compared to constant velocity. It simulates natural phenomena such as:

- Objects speeding up (like a car pressing the gas pedal),
- Slowing down (braking or friction),
- Gravity pulling objects down,
- Forces pushing or pulling objects.

By adjusting velocity incrementally through acceleration, animations become fluid and believable.

How Acceleration Affects Motion

Imagine an object starting at rest with zero velocity. When acceleration is applied, its velocity gradually increases, causing the object to speed up. Similarly, applying negative acceleration reduces velocity and eventually stops the object.

```
let position = 0;
let velocity = 0;
const acceleration = 0.2; // pixels per frame squared

function update() {
```

```
  velocity += acceleration; // Increase velocity
  position += velocity;     // Update position

  drawObjectAt(position, 100);
}
```

Basic Example: Accelerating Motion in 1D In this example, the object's velocity grows by 0.2 pixels per frame every frame, causing it to move faster and faster.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Accelerating Motion in 1D</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      border: 2px solid #444;
      background: #222;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="200"></canvas>

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

    let position = 0;
    let velocity = 0;
    const acceleration = 0.2; // pixels per frame squared

    function drawObjectAt(x, y) {
      ctx.beginPath();
      ctx.arc(x, y, 15, 0, Math.PI * 2);
      ctx.fillStyle = "lime";
      ctx.fill();
      ctx.font = "16px monospace";
      ctx.fillStyle = "#eee";
      ctx.fillText(`Position: ${x.toFixed(1)}`, 10, 30);
      ctx.fillText(`Velocity: ${velocity.toFixed(2)}`, 10, 50);
      ctx.fillText(`Acceleration: ${acceleration}`, 10, 70);
    }

    function update() {
```

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

      velocity += acceleration;  // Increase velocity
      position += velocity;      // Update position

      // Reset if the object moves out of canvas bounds
      if (position > canvas.width + 20) {
        position = -20;
        velocity = 0;
      }

      drawObjectAt(position, canvas.height / 2);
      requestAnimationFrame(update);
    }

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

Acceleration as a Vector

Acceleration, like velocity, is a vector with components in 2D space:

$$\mathbf{a} = (a_x, a_y)$$

Each frame, you update the velocity vector by adding acceleration components, then update position by adding velocity:

```
velocity.x += acceleration.x;
velocity.y += acceleration.y;

position.x += velocity.x;
position.y += velocity.y;
```

This allows for realistic motion in any direction, such as simulating gravity pulling down while an object moves forward.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>2D Vector Acceleration Example</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
```

```
    canvas {
      border: 2px solid #444;
      background: #222;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="400"></canvas>

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

    // Initial position and velocity vectors
    let position = { x: 50, y: 50 };
    let velocity = { x: 2, y: 0 };
    let acceleration = { x: 0, y: 0.15 }; // gravity pulling down

    function drawObjectAt(x, y) {
      ctx.beginPath();
      ctx.arc(x, y, 20, 0, Math.PI * 2);
      ctx.fillStyle = "orange";
      ctx.fill();

      ctx.fillStyle = "#eee";
      ctx.font = "16px monospace";
      ctx.fillText(`Pos: (${x.toFixed(1)}, ${y.toFixed(1)})`, 10, 20);
      ctx.fillText(`Vel: (${velocity.x.toFixed(2)}, ${velocity.y.toFixed(2)})`, 10, 40);
      ctx.fillText(`Acc: (${acceleration.x.toFixed(2)}, ${acceleration.y.toFixed(2)})`, 10, 60);
    }

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

      // Update velocity with acceleration
      velocity.x += acceleration.x;
      velocity.y += acceleration.y;

      // Update position with velocity
      position.x += velocity.x;
      position.y += velocity.y;

      // Bounce off bottom edge
      if (position.y > canvas.height - 20) {
        position.y = canvas.height - 20;
        velocity.y *= -0.7; // dampened bounce
      }
      // Bounce off sides
      if (position.x > canvas.width - 20) {
        position.x = canvas.width - 20;
        velocity.x *= -0.7;
      } else if (position.x < 20) {
        position.x = 20;
        velocity.x *= -0.7;
      }

      drawObjectAt(position.x, position.y);
```

```
      requestAnimationFrame(update);
    }

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

Practical Uses of Acceleration in Animation

- **Gravity:** Constant downward acceleration simulates falling objects.
- **Friction:** Negative acceleration slows objects over time.
- **User Input:** Accelerate objects smoothly in response to controls.
- **Easing:** Gradual start or stop of animations using acceleration curves.

9.1.1 Summary

Acceleration controls how velocity changes over time, enabling smooth speeding up and slowing down. By incrementally adjusting velocity through acceleration vectors, animations achieve natural and dynamic motion, making objects feel more lifelike and responsive. Mastering acceleration is key to creating compelling animations that go beyond simple linear movement.

9.2 Gravity and Angular Acceleration

Gravity and angular acceleration are two important concepts that add realism and complexity to animation by influencing both linear and rotational motion. Understanding how these accelerations work lets you simulate falling objects, spinning wheels, and other dynamic effects naturally.

Gravity: Constant Vertical Acceleration

Gravity is a force that constantly pulls objects downward, causing them to accelerate toward the ground. In animation, gravity is typically modeled as a **constant acceleration vector pointing down the y-axis**.

For example, if gravity has an acceleration of $g = 9.8\,\text{pixels/s}^2$, the vertical velocity v_y of a falling object changes each frame:

$$v_{y,\text{new}} = v_{y,\text{old}} + g \times \Delta t$$

where Δt is the time elapsed per frame.

The object's vertical position updates based on this velocity:

$$y_{\text{new}} = y_{\text{old}} + v_y \times \Delta t$$

```
let position = { x: 100, y: 0 };
let velocity = { x: 0, y: 0 };
const gravity = 0.5; // pixels per frame squared

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

  velocity.y += gravity;       // Apply gravity to vertical velocity
  position.y += velocity.y;     // Update vertical position

  drawBall(position.x, position.y);

  requestAnimationFrame(animate);
}
```

Example: Simulating a Falling Ball In this example, the ball starts from rest and accelerates downward, speeding up as it falls—just like real gravity.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Simulating a Falling Ball</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      border: 2px solid #444;
      background: #222;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="400" height="600"></canvas>

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

    let position = { x: canvas.width / 2, y: 0 };
    let velocity = { x: 0, y: 0 };
    const gravity = 0.5; // pixels per frame squared
    const radius = 25;
```

```
    function drawBall(x, y) {
      ctx.beginPath();
      ctx.arc(x, y, radius, 0, Math.PI * 2);
      ctx.fillStyle = "skyblue";
      ctx.shadowColor = "cyan";
      ctx.shadowBlur = 20;
      ctx.fill();
      ctx.shadowBlur = 0;

      // Draw info text
      ctx.fillStyle = "#eee";
      ctx.font = "16px monospace";
      ctx.fillText(`Position Y: ${y.toFixed(1)}`, 10, 20);
      ctx.fillText(`Velocity Y: ${velocity.y.toFixed(2)}`, 10, 40);
    }

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

      velocity.y += gravity;      // Apply gravity to vertical velocity
      position.y += velocity.y;   // Update vertical position

      // Stop at bottom of canvas
      if (position.y > canvas.height - radius) {
        position.y = canvas.height - radius;
        velocity.y = 0; // stop falling
      }

      drawBall(position.x, position.y);

      requestAnimationFrame(animate);
    }

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

Angular Acceleration: Changing Rotational Speed

While angular velocity controls how fast an object rotates, **angular acceleration** determines how that rotational speed changes over time. It is the rotational equivalent of linear acceleration.

Angular acceleration, denoted as α, updates angular velocity ω each frame:

$$\omega_{\text{new}} = \omega_{\text{old}} + \alpha \times \Delta t$$

The rotation angle θ then updates using the new angular velocity:

$$\theta_{\text{new}} = \theta_{\text{old}} + \omega \times \Delta t$$

```
let angle = 0;
let angularVelocity = 0;
const angularAcceleration = 0.001; // radians per frame squared

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

  angularVelocity += angularAcceleration; // Increase rotation speed
  angle += angularVelocity;                // Update rotation angle

  ctx.save();
  ctx.translate(centerX, centerY);
  ctx.rotate(angle);
  drawShape();
  ctx.restore();

  requestAnimationFrame(animate);
}
```

Example: Accelerating Rotation Here, the object spins faster and faster as angular velocity increases over time.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Accelerating Rotation Example</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="400" height="400"></canvas>

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

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

```
    let angle = 0;
    let angularVelocity = 0;
    const angularAcceleration = 0.001; // radians per frame squared

    function drawShape() {
      ctx.fillStyle = "#f39c12";
      ctx.fillRect(-50, -25, 100, 50); // Centered rectangle
      ctx.strokeStyle = "#e67e22";
      ctx.lineWidth = 3;
      ctx.strokeRect(-50, -25, 100, 50);
    }

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

      angularVelocity += angularAcceleration; // Increase rotation speed
      angle += angularVelocity;                // Update rotation angle

      ctx.save();
      ctx.translate(centerX, centerY);
      ctx.rotate(angle);
      drawShape();
      ctx.restore();

      // Display rotation info
      ctx.fillStyle = "#eee";
      ctx.font = "16px monospace";
      ctx.fillText(`Angle: ${(angle % (2 * Math.PI)).toFixed(2)} rad`, 10, 20);
      ctx.fillText(`Angular Velocity: ${angularVelocity.toFixed(4)} rad/frame²`, 10, 40);
      ctx.fillText(`Angular Acceleration: ${angularAcceleration}`, 10, 60);

      requestAnimationFrame(animate);
    }

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

Combining Gravity and Angular Acceleration

You can combine linear acceleration due to gravity with angular acceleration to create complex, lifelike animations. For example, a falling leaf might accelerate downward due to gravity while spinning faster because of angular acceleration caused by wind or turbulence.

Summary

- **Gravity** acts as a constant vertical acceleration, increasing downward velocity and causing objects to fall faster over time.
- **Angular acceleration** changes an object's rotational speed, making rotations speed up or slow down smoothly.
- Both linear and angular acceleration updates velocities incrementally each frame, then use those velocities to update positions or angles.
- Applying these accelerations in your animations enhances realism by simulating natural

physical behaviors like falling, spinning, and swirling.

Mastering gravity and angular acceleration opens doors to creating dynamic scenes with convincing motion physics, enriching your interactive web experiences.

9.3 Spaceship Motion and Controls

Simulating a spaceship's motion in 2D animation is a classic example that combines velocity, acceleration, and angular velocity to create smooth, realistic movement and rotation. In this section, we'll explore how to control a spaceship using keyboard inputs for thrust and rotation, applying physics principles to update its position and orientation.

Core Concepts

- **Velocity** controls the spaceship's speed and direction.
- **Acceleration** is applied when the ship's thrusters fire, changing velocity over time.
- **Angular velocity** controls how fast the ship rotates.
- **Input handling** translates user keyboard presses into motion commands.

Input Handling: Thrust and Rotation

Typically, we control the spaceship with arrow keys or WASD:

- **Left/Right arrows**: rotate the ship by adjusting angular velocity.
- **Up arrow**: apply thrust in the direction the ship is currently facing, accelerating the ship forward.
- **No input**: velocity slowly reduces due to friction or drag to simulate space resistance.

Updating Ship State: Physics in Motion

1. **Rotation**: The ship's current angle θ updates based on angular velocity ω:

$$\theta_{\text{new}} = \theta_{\text{old}} + \omega \times \Delta t$$

2. **Acceleration (Thrust)**: Thrust accelerates the ship in the direction it's facing. Using trigonometry:

$$a_x = \text{thrust} \times \cos(\theta)$$

$$a_y = \text{thrust} \times \sin(\theta)$$

3. **Velocity Update**: Add acceleration to velocity components:

$$v_x = v_x + a_x \times \Delta t$$

$$v_y = v_y + a_y \times \Delta t$$

4. **Position Update**:

$$x = x + v_x \times \Delta t$$

$$y = y + v_y \times \Delta t$$

Complete Interactive Example

Here is a simplified example combining all these principles:

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

let ship = {
  x: canvas.width / 2,
  y: canvas.height / 2,
  angle: 0,              // Current rotation angle in radians
  velocity: { x: 0, y: 0 },
  angularVelocity: 0,
  acceleration: 0,
};

const thrustPower = 0.1;
const rotationSpeed = 0.05;
const friction = 0.99;   // Simulates space drag

const keys = { left: false, right: false, up: false };

// Listen for key presses
window.addEventListener('keydown', (e) => {
  if (e.code === 'ArrowLeft') keys.left = true;
  if (e.code === 'ArrowRight') keys.right = true;
  if (e.code === 'ArrowUp') keys.up = true;
});

window.addEventListener('keyup', (e) => {
  if (e.code === 'ArrowLeft') keys.left = false;
  if (e.code === 'ArrowRight') keys.right = false;
  if (e.code === 'ArrowUp') keys.up = false;
});

// Draw the ship as a triangle
function drawShip(x, y, angle) {
  ctx.save();
  ctx.translate(x, y);
  ctx.rotate(angle);

  ctx.beginPath();
  ctx.moveTo(15, 0);
  ctx.lineTo(-10, 10);
  ctx.lineTo(-10, -10);
  ctx.closePath();
```

```
  ctx.fillStyle = 'white';
  ctx.fill();
  ctx.restore();
}

function update() {
  // Rotation controls
  if (keys.left) ship.angularVelocity = -rotationSpeed;
  else if (keys.right) ship.angularVelocity = rotationSpeed;
  else ship.angularVelocity = 0;

  ship.angle += ship.angularVelocity;

  // Thrust controls
  if (keys.up) {
    // Calculate acceleration components based on current angle
    ship.velocity.x += Math.cos(ship.angle) * thrustPower;
    ship.velocity.y += Math.sin(ship.angle) * thrustPower;
  }

  // Apply friction to velocity
  ship.velocity.x *= friction;
  ship.velocity.y *= friction;

  // Update position based on velocity
  ship.x += ship.velocity.x;
  ship.y += ship.velocity.y;

  // Wrap around canvas edges
  if (ship.x > canvas.width) ship.x = 0;
  else if (ship.x < 0) ship.x = canvas.width;
  if (ship.y > canvas.height) ship.y = 0;
  else if (ship.y < 0) ship.y = canvas.height;
}

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

  update();
  drawShip(ship.x, ship.y, ship.angle);

  requestAnimationFrame(animate);
}

animate();
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

let ship = {
```

```
  x: canvas.width / 2,
  y: canvas.height / 2,
  angle: 0,              // Current rotation angle in radians
  velocity: { x: 0, y: 0 },
  angularVelocity: 0,
  acceleration: 0,
};

const thrustPower = 0.1;
const rotationSpeed = 0.05;
const friction = 0.99;   // Simulates space drag

const keys = { left: false, right: false, up: false };

// Listen for key presses
window.addEventListener('keydown', (e) => {
  if (e.code === 'ArrowLeft') keys.left = true;
  if (e.code === 'ArrowRight') keys.right = true;
  if (e.code === 'ArrowUp') keys.up = true;
});

window.addEventListener('keyup', (e) => {
  if (e.code === 'ArrowLeft') keys.left = false;
  if (e.code === 'ArrowRight') keys.right = false;
  if (e.code === 'ArrowUp') keys.up = false;
});

// Draw the ship as a triangle
function drawShip(x, y, angle) {
  ctx.save();
  ctx.translate(x, y);
  ctx.rotate(angle);

  ctx.beginPath();
  ctx.moveTo(15, 0);
  ctx.lineTo(-10, 10);
  ctx.lineTo(-10, -10);
  ctx.closePath();

  ctx.fillStyle = 'white';
  ctx.fill();
  ctx.restore();
}

function update() {
  // Rotation controls
  if (keys.left) ship.angularVelocity = -rotationSpeed;
  else if (keys.right) ship.angularVelocity = rotationSpeed;
  else ship.angularVelocity = 0;

  ship.angle += ship.angularVelocity;

  // Thrust controls
  if (keys.up) {
    // Calculate acceleration components based on current angle
    ship.velocity.x += Math.cos(ship.angle) * thrustPower;
    ship.velocity.y += Math.sin(ship.angle) * thrustPower;
  }
```

```
  // Apply friction to velocity
  ship.velocity.x *= friction;
  ship.velocity.y *= friction;

  // Update position based on velocity
  ship.x += ship.velocity.x;
  ship.y += ship.velocity.y;

  // Wrap around canvas edges
  if (ship.x > canvas.width) ship.x = 0;
  else if (ship.x < 0) ship.x = canvas.width;
  if (ship.y > canvas.height) ship.y = 0;
  else if (ship.y < 0) ship.y = canvas.height;
}

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

  update();
  drawShip(ship.x, ship.y, ship.angle);

  requestAnimationFrame(animate);
}

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

Explanation

- The ship rotates left or right by changing `angularVelocity`.
- Pressing the up arrow applies forward acceleration in the current facing direction.
- Velocity is adjusted every frame by acceleration and slowed by friction.
- The ship's position updates using velocity.
- Wrapping around canvas edges keeps the ship on screen.
- Drawing uses canvas transformations to rotate the triangular ship shape.

9.3.1 Summary

By combining velocity, acceleration, and angular velocity, you can create smooth, natural spaceship controls in your animations. User input modifies angular velocity for rotation and acceleration for thrust, updating the ship's velocity and position accordingly. This approach mirrors real physics principles and forms a foundation for many interactive and game-like web animations.

Chapter 10.

Boundaries and Friction

1. Setting and Handling Environmental Boundaries
2. Object Removal and Regeneration
3. Screen Wrapping and Bouncing
4. Applying Friction (Simple and Advanced)
5. Friction & Boundary Formulas

10 Boundaries and Friction

10.1 Setting and Handling Environmental Boundaries

In animation, **environmental boundaries** define the limits within which objects can move. These boundaries often correspond to the edges of the canvas or viewport and are crucial for controlling the behavior of objects—whether they stop, bounce, wrap around, or trigger events upon reaching these limits.

Why Boundaries Matter

Without boundaries, objects could drift off-screen, disappear, or behave unpredictably, reducing user engagement and breaking immersion. Boundaries help maintain a coherent animation space, enforce game rules, or create interesting effects like screen wrapping or collision responses.

Detecting Boundaries: Collision with Edges

To handle boundaries, you first need to detect when an object approaches or crosses an edge. For a rectangular canvas, boundaries are usually defined by the minimum and maximum x and y coordinates:

- Left edge: $x = 0$
- Right edge: $x =$ canvas.width
- Top edge: $y = 0$
- Bottom edge: $y =$ canvas.height

Each frame, compare the object's position with these limits to detect collisions.

Example: Basic Boundary Detection

Consider an object with position `(x, y)` and size `(width, height)`. To detect if it touches or crosses boundaries:

```
if (x < 0) {
  x = 0; // Prevent moving left beyond the canvas
}
if (x + width > canvas.width) {
  x = canvas.width - width; // Prevent moving right beyond the canvas
}
if (y < 0) {
  y = 0; // Prevent moving above the canvas
}
if (y + height > canvas.height) {
  y = canvas.height - height; // Prevent moving below the canvas
}
```

This simple approach **clamps** the object inside the canvas, stopping it at edges.

Responding to Boundaries: Collision Reactions

After detecting a boundary collision, you can decide how the object should react. Common responses include:

- **Stopping:** The object halts movement when it reaches the edge.
- **Bouncing:** The object reverses velocity to simulate a bounce.
- **Wrapping:** The object appears on the opposite side (like in classic arcade games).

Example: Bouncing Off Edges To make an object bounce horizontally:

```
if (x < 0 || x + width > canvas.width) {
  velocity.x = -velocity.x; // Reverse horizontal velocity
}
if (y < 0 || y + height > canvas.height) {
  velocity.y = -velocity.y; // Reverse vertical velocity
}
```

This simple velocity inversion causes the object to "bounce" off edges.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Bouncing Off Edges Example</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="400"></canvas>

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

    // Object properties
    let position = { x: 100, y: 100 };
    let velocity = { x: 3, y: 2 };
    const width = 50;
    const height = 50;
```

```
    function drawObject(x, y) {
      ctx.fillStyle = "#3498db";
      ctx.fillRect(x, y, width, height);
    }

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

      // Update position
      position.x += velocity.x;
      position.y += velocity.y;

      // Bounce horizontally
      if (position.x < 0 || position.x + width > canvas.width) {
        velocity.x = -velocity.x;
      }

      // Bounce vertically
      if (position.y < 0 || position.y + height > canvas.height) {
        velocity.y = -velocity.y;
      }

      drawObject(position.x, position.y);
      requestAnimationFrame(update);
    }

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

Handling Complex Shapes and Multiple Objects

For more complex shapes or multiple objects, collision detection can involve:

- **Bounding boxes:** Using rectangles that enclose shapes.
- **Circle collisions:** Checking distance between centers vs. radius.
- **Pixel-perfect detection:** Comparing pixel data for precise collisions.

These advanced methods help detect boundary contacts more accurately and handle interactions between multiple objects.

Summary

- Environmental boundaries define the movement limits within your canvas or viewport.
- Detecting collisions with boundaries involves comparing object positions to edge coordinates.
- Once detected, you can clamp positions, reverse velocities to bounce, or wrap objects around edges.
- Proper boundary handling is essential for controlled, believable animations and gameplay mechanics.

By mastering boundary detection and response, you ensure objects behave predictably and your animation environment feels consistent and immersive.

10.2 Object Removal and Regeneration

In many animations and games, objects move dynamically within or beyond the visible canvas area. To keep your animation smooth and resource-efficient, you often need to **remove objects** once they leave the visible area and **regenerate** or respawn new ones to maintain continuous action or visual interest.

Why Remove Objects?

Objects that travel off-screen continue to consume memory and processing power if not removed. Over time, this can slow down your animation or even crash the browser. Removing off-screen objects frees up resources and keeps the animation running smoothly.

Detecting Off-Screen Objects

To remove objects that exit the canvas, check their position against the canvas boundaries. For example, if an object's position plus its size moves outside the canvas edges, it's considered off-screen:

```
function isOffScreen(obj, canvas) {
  return (
    obj.x + obj.width < 0 ||          // Left of canvas
    obj.x > canvas.width ||            // Right of canvas
    obj.y + obj.height < 0 ||          // Above canvas
    obj.y > canvas.height              // Below canvas
  );
}
```

Removing Objects from Arrays

If you store objects in an array (e.g., particles, enemies, or bullets), you can filter out off-screen objects each animation frame:

```
objects = objects.filter(obj => !isOffScreen(obj, canvas));
```

This keeps only visible or partially visible objects, removing those fully off-screen.

Regenerating or Respawning Objects

To maintain animation flow, you can regenerate new objects either at random intervals or immediately after removal. For example, you might spawn new enemies entering from the left edge or create continuous particle effects.

```
function spawnObject() {
  return {
    x: Math.random() * canvas.width,
    y: -20, // Start above the canvas
    width: 20,
    height: 20,
    velocityY: Math.random() * 2 + 1,
  };
}
```

```
let fallingObjects = [];

function update() {
  fallingObjects.forEach(obj => {
    obj.y += obj.velocityY;
  });

  // Remove off-screen objects
  fallingObjects = fallingObjects.filter(obj => !isOffScreen(obj, canvas));

  // Spawn new objects if fewer than desired
  while (fallingObjects.length < 10) {
    fallingObjects.push(spawnObject());
  }
}
```

Example: Respawning Falling Objects Here, falling objects are removed once they pass below the canvas. New ones are spawned at the top to keep a steady number of falling items.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Respawning Falling Objects</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="400"></canvas>

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

    function spawnObject() {
      return {
        x: Math.random() * (canvas.width - 20),
        y: -20, // Start above the canvas
        width: 20,
        height: 20,
```

```
        velocityY: Math.random() * 2 + 1,
        color: `hsl(${Math.random() * 360}, 70%, 60%)`,
      };
    }

    function isOffScreen(obj, canvas) {
      return obj.y > canvas.height;
    }

    let fallingObjects = [];

    function drawObject(obj) {
      ctx.fillStyle = obj.color;
      ctx.fillRect(obj.x, obj.y, obj.width, obj.height);
      ctx.strokeStyle = "#000";
      ctx.strokeRect(obj.x, obj.y, obj.width, obj.height);
    }

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

      fallingObjects.forEach(obj => {
        obj.y += obj.velocityY;
        drawObject(obj);
      });

      // Remove off-screen objects
      fallingObjects = fallingObjects.filter(obj => !isOffScreen(obj, canvas));

      // Spawn new objects if fewer than desired
      while (fallingObjects.length < 10) {
        fallingObjects.push(spawnObject());
      }

      requestAnimationFrame(update);
    }

    // Start the animation
    update();
  </script>
</body>
</html>
```

Tips for Smooth Regeneration

- **Randomize spawn positions and speeds** to create natural variation.
- **Use pools** of reusable objects to avoid costly object creation/deletion.
- **Delay spawning** or trigger based on timers/events to control flow.
- **Match regeneration to gameplay** or animation needs, adjusting quantity or frequency accordingly.

Summary

Removing off-screen objects is essential to optimize animation performance and memory use. Combining this with intelligent regeneration ensures your animation remains lively and

consistent without wasting resources. By tracking object positions and managing arrays carefully, you can maintain smooth, ongoing animation flows that engage viewers seamlessly.

10.3 Screen Wrapping and Bouncing

Handling how objects interact with the edges of your animation canvas adds depth and realism to your scenes. Two popular techniques for managing objects reaching boundaries are **screen wrapping** and **bouncing**. Both create dynamic behaviors but achieve very different effects.

Screen Wrapping: Seamless Edge Transition

Screen wrapping allows an object that moves off one edge of the screen to reappear on the opposite edge, creating a continuous loop of motion. This technique is common in classic arcade games like *Asteroids*, where spaceships or asteroids exiting one side of the screen immediately re-enter from the other.

How it works: When an object's position crosses a boundary, you reset its position to the opposite edge. For example:

- If the object moves past the right edge (`x > canvas.width`), reset `x` to `0`.
- If it moves past the left edge (`x < 0`), reset `x` to `canvas.width`.
- Similarly for vertical edges.

```
function wrapPosition(obj, canvas) {
  if (obj.x > canvas.width) {
    obj.x = 0;
  } else if (obj.x < 0) {
    obj.x = canvas.width;
  }

  if (obj.y > canvas.height) {
    obj.y = 0;
  } else if (obj.y < 0) {
    obj.y = canvas.height;
  }
}
```

Example: This logic ensures objects continuously cycle through the screen, giving the illusion of an infinite space.

Bouncing: Reflecting Off Boundaries

Bouncing simulates physical collisions where an object reverses direction upon hitting an edge. This creates a more natural, physics-based response often seen in ball or particle animations.

How it works: When the object reaches a boundary, invert the corresponding component of its velocity vector to "bounce" back:

- Reverse horizontal velocity when hitting left or right edges.
- Reverse vertical velocity when hitting top or bottom edges.

```
function bounceObject(obj, canvas) {
  if (obj.x < 0 || obj.x + obj.width > canvas.width) {
    obj.velocity.x = -obj.velocity.x;
  }
  if (obj.y < 0 || obj.y + obj.height > canvas.height) {
    obj.velocity.y = -obj.velocity.y;
  }
}
```

Example: This changes the direction of motion instantly, causing a bouncing effect.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Bouncing Off Boundaries</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="400"></canvas>

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

    // Object factory
    function createObject() {
      return {
        x: Math.random() * (canvas.width - 30),
        y: Math.random() * (canvas.height - 30),
        width: 30,
        height: 30,
        velocity: {
```

```
      x: (Math.random() * 4) + 1,   // speed 1 to 5 px/frame
      y: (Math.random() * 4) + 1
    },
    color: `hsl(${Math.random() * 360}, 70%, 60%)`
  };
}

// Bounce logic: reflect velocity if hitting edges
function bounceObject(obj, canvas) {
  if (obj.x < 0) {
    obj.x = 0;
    obj.velocity.x = -obj.velocity.x;
  }
  else if (obj.x + obj.width > canvas.width) {
    obj.x = canvas.width - obj.width;
    obj.velocity.x = -obj.velocity.x;
  }

  if (obj.y < 0) {
    obj.y = 0;
    obj.velocity.y = -obj.velocity.y;
  }
  else if (obj.y + obj.height > canvas.height) {
    obj.y = canvas.height - obj.height;
    obj.velocity.y = -obj.velocity.y;
  }
}

function drawObject(obj) {
  ctx.fillStyle = obj.color;
  ctx.fillRect(obj.x, obj.y, obj.width, obj.height);
  ctx.strokeStyle = "#000";
  ctx.strokeRect(obj.x, obj.y, obj.width, obj.height);
}

// Create multiple bouncing objects
const objects = [];
const numObjects = 8;
for (let i = 0; i < numObjects; i++) {
  let obj = createObject();
  // Randomize velocity direction (+/-)
  obj.velocity.x *= Math.random() < 0.5 ? 1 : -1;
  obj.velocity.y *= Math.random() < 0.5 ? 1 : -1;
  objects.push(obj);
}

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

  objects.forEach(obj => {
    obj.x += obj.velocity.x;
    obj.y += obj.velocity.y;

    bounceObject(obj, canvas);
    drawObject(obj);
  });

  requestAnimationFrame(update);
```

```
    }

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

Visual Demonstration: Combining Both Effects

Imagine animating a ball:

- **With wrapping:** The ball disappears off the right side and instantly reappears on the left, creating an endless looping motion.
- **With bouncing:** The ball hits the right edge and reverses direction, bouncing back and forth within the screen.

Choosing Between Wrapping and Bouncing

- Use **screen wrapping** for open-ended spaces or infinite loops, great for arcade-style gameplay or background animations.
- Use **bouncing** to simulate realistic physics interactions and keep objects confined.

10.3.1 Summary

- **Screen wrapping** teleports objects from one edge to the opposite, enabling continuous motion across boundaries.
- **Bouncing** reverses velocity on collision with edges, mimicking physical rebounds.
- Both techniques enrich your animations by defining how objects behave at boundaries, offering different gameplay and visual dynamics.

By implementing these methods, you can add polish and professionalism to your animations, creating smooth and engaging user experiences.

10.4 Applying Friction (Simple and Advanced)

In animation and physics simulations, **friction** is the force that opposes motion, gradually slowing down moving objects until they come to rest or reach a steady speed. Accurately modeling friction enhances realism by mimicking how objects interact with surfaces and air resistance.

Simple Friction: Slowing Velocity Over Time

The simplest way to model friction is by reducing an object's velocity by a constant factor each frame, simulating a gradual loss of speed. This is often called **linear friction** or **damping**.

How it works: Each animation frame, multiply the velocity by a friction coefficient between 0 and 1 (e.g., 0.95):

```
const friction = 0.95;
velocity.x *= friction;
velocity.y *= friction;
```

If the velocity becomes very small, you can set it to zero to stop the object completely.

Example: Imagine a ball rolling on a rough surface; each frame, friction slows it down:

```
function update() {
  // Apply friction
  velocity.x *= 0.95;
  velocity.y *= 0.95;

  // Update position
  position.x += velocity.x;
  position.y += velocity.y;

  // Stop ball if very slow
  if (Math.abs(velocity.x) < 0.01) velocity.x = 0;
  if (Math.abs(velocity.y) < 0.01) velocity.y = 0;
}
```

This simple model gives a natural slowing effect without complex calculations.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Rolling Ball with Friction</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="300"></canvas>

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

```
    let position = { x: 100, y: canvas.height / 2 };
    let velocity = { x: 10, y: 0 }; // Initial speed to the right

    function drawBall(x, y) {
      ctx.beginPath();
      ctx.arc(x, y, 20, 0, Math.PI * 2);
      ctx.fillStyle = "tomato";
      ctx.shadowColor = "orangered";
      ctx.shadowBlur = 20;
      ctx.fill();
      ctx.shadowBlur = 0;

      // Display velocity info
      ctx.fillStyle = "#eee";
      ctx.font = "16px monospace";
      ctx.fillText(`Velocity X: ${velocity.x.toFixed(3)}`, 10, 20);
    }

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

      // Apply friction
      velocity.x *= 0.95;
      velocity.y *= 0.95;

      // Stop ball if very slow
      if (Math.abs(velocity.x) < 0.01) velocity.x = 0;
      if (Math.abs(velocity.y) < 0.01) velocity.y = 0;

      // Update position
      position.x += velocity.x;
      position.y += velocity.y;

      // Prevent ball from leaving canvas
      if (position.x > canvas.width - 20) {
        position.x = canvas.width - 20;
        velocity.x = 0;
      }

      drawBall(position.x, position.y);

      // Continue animation only if ball still moves
      if (velocity.x !== 0 || velocity.y !== 0) {
        requestAnimationFrame(update);
      } else {
        ctx.fillStyle = "#eee";
        ctx.font = "20px monospace";
        ctx.fillText("Ball stopped due to friction.", 150, canvas.height / 2 - 40);
      }
    }

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

Advanced Friction: Surface and Drag Effects

More advanced friction models account for different surface types and **drag forces** that depend on velocity magnitude.

1. **Static vs Kinetic Friction:** Static friction prevents motion from starting until enough force is applied, while kinetic friction slows objects already moving. For animations, this distinction can be simplified but is important for realistic behavior.
2. **Drag Force (Air or Fluid Resistance):** Drag depends on velocity squared and opposes motion, becoming stronger as speed increases:

$$F_{\text{drag}} = -c_d \times v^2$$

   where c_d is the drag coefficient and v is velocity.

Implementing Drag: Calculate drag force proportional to velocity squared, then update velocity accordingly:

```
const dragCoefficient = 0.01;

function applyDrag(velocity) {
  const speed = Math.sqrt(velocity.x * velocity.x + velocity.y * velocity.y);
  if (speed > 0) {
    const dragForce = dragCoefficient * speed * speed;
    const dragX = dragForce * (velocity.x / speed);
    const dragY = dragForce * (velocity.y / speed);

    velocity.x -= dragX;
    velocity.y -= dragY;
  }
}
```

This model slows fast-moving objects more strongly, simulating air resistance or viscous fluid effects.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Drag Force Example</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
```

```
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="300"></canvas>

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

    let position = { x: 100, y: canvas.height / 2 };
    let velocity = { x: 15, y: -5 }; // initial velocity
    const dragCoefficient = 0.01;

    function applyDrag(velocity) {
      const speed = Math.sqrt(velocity.x * velocity.x + velocity.y * velocity.y);
      if (speed > 0) {
        const dragForce = dragCoefficient * speed * speed;
        const dragX = dragForce * (velocity.x / speed);
        const dragY = dragForce * (velocity.y / speed);

        velocity.x -= dragX;
        velocity.y -= dragY;
      }
    }

    function drawBall(x, y) {
      ctx.beginPath();
      ctx.arc(x, y, 20, 0, Math.PI * 2);
      ctx.fillStyle = "#1abc9c";
      ctx.shadowColor = "#16a085";
      ctx.shadowBlur = 20;
      ctx.fill();
      ctx.shadowBlur = 0;

      ctx.fillStyle = "#eee";
      ctx.font = "16px monospace";
      ctx.fillText(`Velocity X: ${velocity.x.toFixed(3)}`, 10, 20);
      ctx.fillText(`Velocity Y: ${velocity.y.toFixed(3)}`, 10, 40);
      ctx.fillText(`Speed: ${Math.sqrt(velocity.x*velocity.x + velocity.y*velocity.y).toFixed(3)}`, 10,
    }

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

      applyDrag(velocity);

      // Update position
      position.x += velocity.x;
      position.y += velocity.y;

      // Keep ball inside canvas bounds horizontally
      if (position.x > canvas.width - 20) {
        position.x = canvas.width - 20;
        velocity.x = -velocity.x * 0.7; // bounce with some energy loss
```

```
      } else if (position.x < 20) {
        position.x = 20;
        velocity.x = -velocity.x * 0.7;
      }

      // Keep ball inside canvas bounds vertically
      if (position.y > canvas.height - 20) {
        position.y = canvas.height - 20;
        velocity.y = -velocity.y * 0.7;
      } else if (position.y < 20) {
        position.y = 20;
        velocity.y = -velocity.y * 0.7;
      }

      // Stop ball if very slow
      const speed = Math.sqrt(velocity.x * velocity.x + velocity.y * velocity.y);
      if (speed < 0.05) {
        velocity.x = 0;
        velocity.y = 0;
      }

      drawBall(position.x, position.y);

      if (velocity.x !== 0 || velocity.y !== 0) {
        requestAnimationFrame(update);
      } else {
        ctx.fillStyle = "#eee";
        ctx.font = "20px monospace";
        ctx.fillText("Ball stopped due to drag.", 150, canvas.height / 2 - 40);
      }
    }

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

Combining Friction and Drag in Animations

To achieve nuanced motion, combine linear friction and drag:

- Use linear friction for surface roughness.
- Use drag to simulate resistance from air or water.

Visual Effects of Friction

- **Without friction:** Objects keep moving indefinitely, creating unrealistic "infinite glide."
- **With friction:** Objects gradually slow, eventually stop, or reach a terminal velocity.
- **With drag:** Fast objects slow more rapidly, giving dynamic, natural motion.

10.4.1 Summary

- Friction opposes motion and slows objects.
- **Simple friction** models reduce velocity by a fixed factor, easy to implement and effective for many cases.
- **Advanced friction** considers forces like drag that depend on velocity, offering realism for faster or fluid environments.
- Combining these models lets you create smooth, believable animations where objects interact naturally with their environment.

Understanding and applying friction properly ensures your animations respond realistically, enhancing user immersion and visual appeal.

10.5 Friction & Boundary Formulas

In animation, understanding the core formulas for friction and boundary responses helps you create smooth, realistic motion and collision behavior. Below are essential mathematical expressions and practical tips for implementing them.

Friction Formulas

1. **Linear Friction (Damping):** A simple way to simulate friction is by scaling velocity each frame:

$$v_{\text{new}} = v_{\text{old}} \times (1 - f)$$

where

- v is velocity,
- f is the friction coefficient (a small value like 0.05).

In code, this becomes:

```
velocity *= (1 - friction);
```

This reduces velocity gradually until the object stops.

2. **Drag Force:** Drag depends on velocity squared and acts opposite to movement:

$$F_{\text{drag}} = -c_d \times v^2$$

where c_d is the drag coefficient. To apply drag force as acceleration:

$$a_{\text{drag}} = \frac{F_{\text{drag}}}{m} = -\frac{c_d}{m} v^2$$

You update velocity using acceleration:

$$v_{\text{new}} = v_{\text{old}} + a \times \Delta t$$

In JavaScript:

```
const speed = Math.sqrt(velocity.x ** 2 + velocity.y ** 2);
const dragForce = dragCoefficient * speed * speed;
const dragAccelX = -(dragForce / mass) * (velocity.x / speed);
const dragAccelY = -(dragForce / mass) * (velocity.y / speed);

velocity.x += dragAccelX * deltaTime;
velocity.y += dragAccelY * deltaTime;
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Drag Force with Mass and DeltaTime</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
<canvas id="canvas" width="600" height="300"></canvas>

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

  // Physics parameters
  const dragCoefficient = 0.05;
  const mass = 1; // mass of the object
  let position = { x: 100, y: canvas.height / 2 };
  let velocity = { x: 100, y: -40 }; // pixels per second

  // For deltaTime calculation
  let lastTime = performance.now();

  function drawBall(x, y) {
    ctx.beginPath();
```

```
  ctx.arc(x, y, 20, 0, Math.PI * 2);
  ctx.fillStyle = "#3498db";
  ctx.shadowColor = "#2980b9";
  ctx.shadowBlur = 15;
  ctx.fill();
  ctx.shadowBlur = 0;

  ctx.fillStyle = "#eee";
  ctx.font = "16px monospace";
  ctx.fillText(`Velocity X: ${velocity.x.toFixed(2)} px/s`, 10, 20);
  ctx.fillText(`Velocity Y: ${velocity.y.toFixed(2)} px/s`, 10, 40);
}

function update(currentTime) {
  const deltaTime = (currentTime - lastTime) / 1000; // convert ms to seconds
  lastTime = currentTime;

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

  // Calculate speed
  const speed = Math.sqrt(velocity.x ** 2 + velocity.y ** 2);

  if (speed > 0) {
    // Calculate drag force magnitude: F_drag = c * v^2
    const dragForce = dragCoefficient * speed * speed;

    // Calculate acceleration from drag: a = F / m
    const dragAccelX = -(dragForce / mass) * (velocity.x / speed);
    const dragAccelY = -(dragForce / mass) * (velocity.y / speed);

    // Update velocity with acceleration * deltaTime
    velocity.x += dragAccelX * deltaTime;
    velocity.y += dragAccelY * deltaTime;
  }

  // Update position
  position.x += velocity.x * deltaTime;
  position.y += velocity.y * deltaTime;

  // Keep ball inside canvas horizontally (bounce)
  if (position.x > canvas.width - 20) {
    position.x = canvas.width - 20;
    velocity.x = -velocity.x * 0.7;
  } else if (position.x < 20) {
    position.x = 20;
    velocity.x = -velocity.x * 0.7;
  }

  // Keep ball inside canvas vertically (bounce)
  if (position.y > canvas.height - 20) {
    position.y = canvas.height - 20;
    velocity.y = -velocity.y * 0.7;
  } else if (position.y < 20) {
    position.y = 20;
    velocity.y = -velocity.y * 0.7;
  }

  // Stop ball if velocity is very low
```

```
    if (Math.abs(velocity.x) < 0.01) velocity.x = 0;
    if (Math.abs(velocity.y) < 0.01) velocity.y = 0;

    drawBall(position.x, position.y);

    // Continue animation if ball is moving
    if (velocity.x !== 0 || velocity.y !== 0) {
      requestAnimationFrame(update);
    } else {
      ctx.fillStyle = "#eee";
      ctx.font = "20px monospace";
      ctx.fillText("Ball stopped due to drag.", 180, canvas.height / 2 - 40);
    }
  }

  requestAnimationFrame(update);
</script>
</body>
</html>
```

Boundary Response Formulas

1. **Bouncing (Reflection):** When an object hits a boundary, reverse the velocity component perpendicular to that boundary:

$$v_x = -v_x \quad \text{or} \quad v_y = -v_y$$

To simulate energy loss, apply a restitution coefficient r (between 0 and 1):

$$v_{\text{new}} = -r \times v_{\text{old}}$$

Example:

```
if (obj.x < 0 || obj.x + obj.width > canvas.width) {
  velocity.x = -velocity.x * restitution;
}
```

2. **Screen Wrapping:** To wrap positions across boundaries:

$$x_{\text{new}} = \begin{cases} 0 & \text{if } x > \text{width} \\ \text{width} & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

Similarly for the y-axis.

10.5.1 Practical Tips

- Choose friction and restitution coefficients experimentally to balance realism and responsiveness.
- Remember to factor in **delta time** (Δt) for frame rate independence.
- For complex boundaries, consider separating position updates and collision checks for stable behavior.

By applying these formulas, you can control how objects slow down, bounce, or wrap around edges, creating smooth, lifelike animations that respond intuitively to their environment.

Chapter 11.

User Interaction Techniques

1. Pressing, Releasing, and Touch Events
2. Dragging and Combining with Motion
3. Simulating Throwing

11 User Interaction Techniques

11.1 Pressing, Releasing, and Touch Events

User interaction is key to creating engaging animations and games. Handling input events such as mouse clicks, keyboard presses, and touch gestures lets your animations respond dynamically to user actions. This section introduces how to manage these events effectively using JavaScript event listeners, along with best practices to ensure responsive and smooth interactions.

Event Listeners: The Basics

In JavaScript, **event listeners** are functions that run when a specific user action occurs. For example, when the user presses a key or clicks the mouse, the browser triggers an event. You attach listeners to DOM elements, such as the entire document, a canvas, or buttons, to detect and respond to these events.

Common event types include:

- **`mousedown` / `mouseup`** – Detect mouse button press and release
- **`keydown` / `keyup`** – Detect keyboard key press and release
- **`touchstart` / `touchend`** – Detect finger touches on mobile devices

Example of attaching a mouse press listener:

```
canvas.addEventListener('mousedown', function(event) {
  console.log('Mouse pressed at', event.clientX, event.clientY);
});
```

Event Propagation and Bubbling

Events in the DOM have a propagation path:

- **Capturing phase:** Event travels from the root down to the target element.
- **Target phase:** Event reaches the target element.
- **Bubbling phase:** Event bubbles back up to the root.

By default, event listeners listen during the bubbling phase, but you can specify the capturing phase by passing a third argument (`true`) to `addEventListener`.

Understanding propagation helps when you have nested elements and want to control which element reacts to an event. You can stop propagation using:

```
event.stopPropagation();
```

Handling Keyboard Input

Keyboard events provide a `key` or `code` property that identifies which key was pressed or released:

```
document.addEventListener('keydown', function(event) {
  if (event.key === 'ArrowUp') {
    // Move object up
```

```
  }
});
```

Handling both `keydown` and `keyup` lets you track whether a key is held down or released, useful for smooth continuous movement.

Touch Events on Mobile Devices

Touch events mirror mouse events but support multiple fingers and gestures:

- `touchstart` — finger touches the screen
- `touchmove` — finger moves on screen
- `touchend` — finger lifts off

Example:

```
canvas.addEventListener('touchstart', function(event) {
  const touch = event.touches[0];
  console.log('Touch at', touch.clientX, touch.clientY);
});
```

For multi-touch gestures, you can track multiple `touches` in the event object.

Best Practices for Responsiveness

- **Debounce and throttle events:** Avoid flooding your animation loop with rapid events by limiting how often handlers run.
- **Use passive listeners** for scroll and touch to improve performance:
  ```
  element.addEventListener('touchmove', handler, { passive: true });
  ```
- **Normalize input:** Provide consistent behavior across mouse, touch, and keyboard inputs.
- **Prevent default behaviors** only when necessary (e.g., prevent scrolling on swipe during drag interactions) using `event.preventDefault()`.

Practical Demo: Click to Change Color

```
const canvas = document.getElementById('myCanvas');
const ctx = canvas.getContext('2d');
let color = 'blue';

canvas.addEventListener('mousedown', () => {
  color = color === 'blue' ? 'red' : 'blue';
  draw();
});

function draw() {
  ctx.fillStyle = color;
  ctx.fillRect(0, 0, canvas.width, canvas.height);
}

draw();
```

This simple example toggles the canvas color when the mouse is pressed, illustrating basic event handling.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Click to Change Color</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      border: 2px solid #444;
      cursor: pointer;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="myCanvas" width="400" height="300"></canvas>

  <script>
    const canvas = document.getElementById('myCanvas');
    const ctx = canvas.getContext('2d');
    let color = 'blue';

    canvas.addEventListener('mousedown', () => {
      color = color === 'blue' ? 'red' : 'blue';
      draw();
    });

    function draw() {
      ctx.fillStyle = color;
      ctx.fillRect(0, 0, canvas.width, canvas.height);
    }

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

11.1.1 Summary

Mastering pressing, releasing, and touch events is essential for interactive animations. Using event listeners properly, understanding event propagation, and optimizing for responsiveness will enable you to build engaging, fluid user experiences across devices.

11.2 Dragging and Combining with Motion

Dragging objects on the canvas is a fundamental interaction technique that, when combined with existing motion mechanics, allows you to create rich, intuitive animations. In this section, we'll explore how to implement dragging with smooth following behavior, apply constraints to keep objects within bounds, and add inertia effects for more realistic motion.

Basic Dragging Setup

To start, you detect when the user clicks and holds an object on the canvas. You listen for mouse or touch events, track the pointer's position, and update the object's location accordingly.

Here's the typical event flow:

1. **`mousedown` or `touchstart`** – Detect start of dragging; check if the pointer is over the object.
2. **`mousemove` or `touchmove`** – While dragging, update the object's position to follow the pointer.
3. **`mouseup` or `touchend`** – Release the object, ending the drag.

Smooth Following

To avoid jittery movement, use smoothing techniques such as linear interpolation or easing when updating the object's position toward the pointer location:

```
function lerp(start, end, t) {
  return start + (end - start) * t;
}
```

Within the drag loop, instead of snapping directly to the pointer, update the position incrementally:

```
object.x = lerp(object.x, pointer.x, 0.2);
object.y = lerp(object.y, pointer.y, 0.2);
```

This creates a natural "follow" effect.

Full runnable code:

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

```
  <meta charset="UTF-8" />
  <title>Smooth Following with lerp</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
      cursor: crosshair;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="400"></canvas>

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

    const object = { x: 300, y: 200, radius: 20 };
    const pointer = { x: 300, y: 200 };

    // Linear interpolation function
    function lerp(start, end, t) {
      return start + (end - start) * t;
    }

    canvas.addEventListener("mousemove", (e) => {
      const rect = canvas.getBoundingClientRect();
      pointer.x = e.clientX - rect.left;
      pointer.y = e.clientY - rect.top;
    });

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

      // Smoothly update object position towards pointer
      object.x = lerp(object.x, pointer.x, 0.2);
      object.y = lerp(object.y, pointer.y, 0.2);

      // Draw the object
      ctx.beginPath();
      ctx.arc(object.x, object.y, object.radius, 0, Math.PI * 2);
      ctx.fillStyle = "#f39c12";
      ctx.fill();

      requestAnimationFrame(animate);
    }

    animate();
```

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

Applying Constraints

Often, you want to prevent the object from being dragged outside the visible area or specific boundaries. You can clamp the position values within limits:

```
object.x = Math.min(canvas.width - object.width, Math.max(0, object.x));
object.y = Math.min(canvas.height - object.height, Math.max(0, object.y));
```

This ensures the object stays within the canvas edges.

Combining Dragging with Existing Motion

In many animations, objects have their own velocity and acceleration. When dragging, you override their position with the pointer, but on release, you can apply inertia by transferring the drag's velocity back into the object's motion.

Track the velocity during dragging by measuring pointer movement over time:

```
let lastPointerX, lastPointerY, velocityX = 0, velocityY = 0;

function onPointerMove(event) {
  const currentX = event.clientX;
  const currentY = event.clientY;

  velocityX = currentX - lastPointerX;
  velocityY = currentY - lastPointerY;

  lastPointerX = currentX;
  lastPointerY = currentY;

  // Update object position
  object.x = currentX;
  object.y = currentY;
}
```

Upon release, assign these velocities to the object:

```
function onPointerUp() {
  object.vx = velocityX;
  object.vy = velocityY;
  dragging = false;
}
```

The object then continues moving with inertia, gradually slowing down due to friction or drag.

Putting It All Together: Sample Dragging Code

```
let dragging = false;
let dragOffsetX = 0;
let dragOffsetY = 0;
```

```
canvas.addEventListener('mousedown', (e) => {
  if (isPointerOnObject(e.clientX, e.clientY)) {
    dragging = true;
    dragOffsetX = e.clientX - object.x;
    dragOffsetY = e.clientY - object.y;
    lastPointerX = e.clientX;
    lastPointerY = e.clientY;
  }
});

canvas.addEventListener('mousemove', (e) => {
  if (dragging) {
    const currentX = e.clientX;
    const currentY = e.clientY;

    velocityX = currentX - lastPointerX;
    velocityY = currentY - lastPointerY;

    lastPointerX = currentX;
    lastPointerY = currentY;

    object.x = currentX - dragOffsetX;
    object.y = currentY - dragOffsetY;

    // Apply constraints
    object.x = Math.min(canvas.width - object.width, Math.max(0, object.x));
    object.y = Math.min(canvas.height - object.height, Math.max(0, object.y));
  }
});

canvas.addEventListener('mouseup', () => {
  if (dragging) {
    object.vx = velocityX;
    object.vy = velocityY;
    dragging = false;
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Drag and Throw Demo</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      background: #222;
```

```
      border: 2px solid #444;
      display: block;
      cursor: grab;
    }
    canvas:active {
      cursor: grabbing;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="400"></canvas>

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

    const object = {
      x: 100,
      y: 100,
      width: 80,
      height: 60,
      vx: 0,
      vy: 0,
      color: "#f39c12",
    };

    let dragging = false;
    let dragOffsetX = 0;
    let dragOffsetY = 0;
    let lastPointerX = 0;
    let lastPointerY = 0;
    let velocityX = 0;
    let velocityY = 0;

    // Check if pointer is inside object
    function isPointerOnObject(px, py) {
      const rect = canvas.getBoundingClientRect();
      const x = px - rect.left;
      const y = py - rect.top;
      return (
        x >= object.x &&
        x <= object.x + object.width &&
        y >= object.y &&
        y <= object.y + object.height
      );
    }

    canvas.addEventListener("mousedown", (e) => {
      if (isPointerOnObject(e.clientX, e.clientY)) {
        dragging = true;
        const rect = canvas.getBoundingClientRect();
        dragOffsetX = e.clientX - rect.left - object.x;
        dragOffsetY = e.clientY - rect.top - object.y;
        lastPointerX = e.clientX;
        lastPointerY = e.clientY;
      }
    });
```

```
canvas.addEventListener("mousemove", (e) => {
  if (dragging) {
    const currentX = e.clientX;
    const currentY = e.clientY;

    velocityX = currentX - lastPointerX;
    velocityY = currentY - lastPointerY;

    lastPointerX = currentX;
    lastPointerY = currentY;

    const rect = canvas.getBoundingClientRect();
    object.x = currentX - rect.left - dragOffsetX;
    object.y = currentY - rect.top - dragOffsetY;

    // Clamp inside canvas
    object.x = Math.min(canvas.width - object.width, Math.max(0, object.x));
    object.y = Math.min(canvas.height - object.height, Math.max(0, object.y));
  }
});

canvas.addEventListener("mouseup", () => {
  if (dragging) {
    object.vx = velocityX;
    object.vy = velocityY;
    dragging = false;
  }
});

// Optional: friction to slow down thrown object
const friction = 0.95;

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

  if (!dragging) {
    // Apply velocity with friction
    object.vx *= friction;
    object.vy *= friction;

    object.x += object.vx;
    object.y += object.vy;

    // Clamp inside canvas and reverse velocity on collision (bounce effect)
    if (object.x < 0) {
      object.x = 0;
      object.vx = -object.vx * 0.7;
    } else if (object.x + object.width > canvas.width) {
      object.x = canvas.width - object.width;
      object.vx = -object.vx * 0.7;
    }

    if (object.y < 0) {
      object.y = 0;
      object.vy = -object.vy * 0.7;
    } else if (object.y + object.height > canvas.height) {
      object.y = canvas.height - object.height;
      object.vy = -object.vy * 0.7;
```

```
        }

        // Stop very slow movement
        if (Math.abs(object.vx) < 0.1) object.vx = 0;
        if (Math.abs(object.vy) < 0.1) object.vy = 0;
      }

      // Draw the object
      ctx.fillStyle = object.color;
      ctx.fillRect(object.x, object.y, object.width, object.height);

      requestAnimationFrame(animate);
    }

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

11.2.1 Summary

Combining dragging with motion mechanics involves tracking pointer positions and velocities, smoothing movement to avoid jitter, constraining positions within boundaries, and applying inertia after release. These techniques create natural, fluid animations that respond intuitively to user input, enhancing interactivity and immersion in your canvas applications.

11.3 Simulating Throwing

Simulating a "throw" or "flick" gesture on an object is a powerful interaction technique that mimics real-world physics by applying momentum after a user releases an object. This technique involves tracking the user's motion (usually drag), capturing the final velocity upon release, and using that velocity to continue the object's movement with inertia and friction.

This section demonstrates how to build a simple throwing mechanic by tracking velocity during user interaction and applying it to an object when released.

Step 1: Tracking Motion During Drag

To simulate throwing, we first need to measure how fast the user moves the pointer (mouse or touch) while dragging. This involves capturing the pointer's position each frame and calculating the difference over time.

```
let isDragging = false;
let lastX = 0, lastY = 0;
let velocityX = 0, velocityY = 0;
let object = { x: 100, y: 100, vx: 0, vy: 0 };
```

On `mousedown`, initiate dragging and record the current pointer location.

```
canvas.addEventListener('mousedown', (e) => {
  if (isPointerOnObject(e.clientX, e.clientY)) {
    isDragging = true;
    lastX = e.clientX;
    lastY = e.clientY;
  }
});
```

On `mousemove`, update the object's position and calculate the velocity:

```
canvas.addEventListener('mousemove', (e) => {
  if (isDragging) {
    const dx = e.clientX - lastX;
    const dy = e.clientY - lastY;

    velocityX = dx;
    velocityY = dy;

    object.x += dx;
    object.y += dy;

    lastX = e.clientX;
    lastY = e.clientY;
  }
});
```

Step 2: Releasing and Applying Momentum

When the user releases the mouse (or lifts a finger), we stop dragging and apply the captured velocity to the object's motion:

```
canvas.addEventListener('mouseup', () => {
  if (isDragging) {
    object.vx = velocityX;
    object.vy = velocityY;
    isDragging = false;
  }
});
```

Now the object continues to move even after the drag ends, just like a flicked ball.

Step 3: Animating Continued Motion with Friction

To make the motion feel realistic, apply gradual friction to slow the object over time:

```
function update() {
  if (!isDragging) {
    object.x += object.vx;
    object.y += object.vy;

    // Apply friction
    object.vx *= 0.95;
    object.vy *= 0.95;

    // Optional: stop the object when speed is low
    if (Math.abs(object.vx) < 0.1) object.vx = 0;
    if (Math.abs(object.vy) < 0.1) object.vy = 0;
```

```
  }

  draw();
  requestAnimationFrame(update);
}
```

The object will slow down smoothly, simulating momentum loss due to friction or air resistance.

Step 4: Drawing the Object

Here's a simple rendering function for a circular object:

```
function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.beginPath();
  ctx.arc(object.x, object.y, 20, 0, Math.PI * 2);
  ctx.fillStyle = '#3498db';
  ctx.fill();
}
```

Call `update()` once to kick off the animation loop:

```
update();
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Drag and Flick with Momentum</title>
<style>
  body {
    margin: 0;
    background: #111;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    user-select: none;
  }
  canvas {
    background: #222;
    border: 2px solid #444;
    display: block;
    cursor: grab;
  }
  canvas:active {
    cursor: grabbing;
  }
</style>
</head>
<body>
<canvas id="canvas" width="600" height="400"></canvas>

<script>
  const canvas = document.getElementById("canvas");
```

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

let object = {
  x: 100,
  y: 100,
  vx: 0,
  vy: 0,
  radius: 20,
};

let isDragging = false;
let lastX = 0, lastY = 0;
let velocityX = 0, velocityY = 0;

// Check if pointer is inside the circle
function isPointerOnObject(px, py) {
  const rect = canvas.getBoundingClientRect();
  const x = px - rect.left;
  const y = py - rect.top;
  const dx = x - object.x;
  const dy = y - object.y;
  return dx * dx + dy * dy <= object.radius * object.radius;
}

canvas.addEventListener('mousedown', (e) => {
  if (isPointerOnObject(e.clientX, e.clientY)) {
    isDragging = true;
    lastX = e.clientX;
    lastY = e.clientY;
  }
});

canvas.addEventListener('mousemove', (e) => {
  if (isDragging) {
    const dx = e.clientX - lastX;
    const dy = e.clientY - lastY;

    velocityX = dx;
    velocityY = dy;

    object.x += dx;
    object.y += dy;

    lastX = e.clientX;
    lastY = e.clientY;

    clampPosition();
  }
});

canvas.addEventListener('mouseup', () => {
  if (isDragging) {
    object.vx = velocityX;
    object.vy = velocityY;
    isDragging = false;
  }
});
```

```
  // Also stop dragging if mouse leaves canvas
  canvas.addEventListener('mouseleave', () => {
    if (isDragging) {
      object.vx = velocityX;
      object.vy = velocityY;
      isDragging = false;
    }
  });

  function clampPosition() {
    // Keep object inside canvas bounds
    if (object.x < object.radius) object.x = object.radius;
    if (object.x > canvas.width - object.radius) object.x = canvas.width - object.radius;
    if (object.y < object.radius) object.y = object.radius;
    if (object.y > canvas.height - object.radius) object.y = canvas.height - object.radius;
  }

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

    ctx.beginPath();
    ctx.arc(object.x, object.y, object.radius, 0, Math.PI * 2);
    ctx.fillStyle = '#3498db';
    ctx.fill();

    // Optional: display velocity for debugging
    ctx.fillStyle = '#eee';
    ctx.font = '16px monospace';
    ctx.fillText(`Velocity X: ${object.vx.toFixed(1)}`, 10, 20);
    ctx.fillText(`Velocity Y: ${object.vy.toFixed(1)}`, 10, 40);
  }

  function update() {
    if (!isDragging) {
      object.x += object.vx;
      object.y += object.vy;

      // Apply friction
      object.vx *= 0.95;
      object.vy *= 0.95;

      clampPosition();

      // Stop if very slow
      if (Math.abs(object.vx) < 0.1) object.vx = 0;
      if (Math.abs(object.vy) < 0.1) object.vy = 0;
    }

    draw();
    requestAnimationFrame(update);
  }

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

11.3.1 Summary

Simulating a throw in JavaScript canvas animations involves tracking the speed of user movement, applying the calculated velocity when the object is released, and animating with inertia. This technique allows you to create natural, physics-like interactions ideal for games, physics simulations, or intuitive UI gestures. You can further enhance realism by adding collision detection, boundary bouncing, or advanced friction models.

Chapter 12.

Easing and Springing 1

1. Understanding Proportional Motion
2. Simple and Advanced Easing
3. Springing in One and Two Dimensions

12 Easing and Springing 1

12.1 Understanding Proportional Motion

Proportional motion is one of the most fundamental concepts in easing and animation. At its core, it describes a behavior where an object moves toward a target at a speed proportional to the distance remaining. The farther away the object is, the faster it moves; as it nears the target, it slows down naturally—creating smooth, intuitive motion often used in UI transitions, animations, and even simulations like springing.

The Core Concept

Imagine you want an object to move toward a destination, say a circle moving toward your mouse cursor. If the object always moved at a fixed speed, it would either overshoot and jitter (requiring stopping logic), or it would instantly stop when close enough—often feeling abrupt or robotic.

Proportional motion solves this by reducing the speed as the object gets closer, making the motion feel more organic.

The basic formula for proportional motion is:

```
velocity = (target - current) * easing
```

Where:

- `target` is the destination value (e.g., mouse position),
- `current` is the object's current position,
- `easing` is a constant between 0 and 1 that determines the "responsiveness" of the motion.

You then update the object's position by adding the velocity:

```
current += velocity;
```

Simple Example

Let's say we want a circle to follow the mouse cursor using proportional motion:

```
let x = 100;
let y = 100;
let easing = 0.05;

canvas.addEventListener('mousemove', (e) => {
  mouseX = e.clientX;
  mouseY = e.clientY;
});

function update() {
  let dx = mouseX - x;
  let dy = mouseY - y;

  x += dx * easing;
  y += dy * easing;
```

```
  draw();
  requestAnimationFrame(update);
}
```

In this example, `dx * easing` and `dy * easing` are the proportional movements. The object speeds up when far from the cursor and slows as it nears it.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Circle Following Mouse with Easing</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
      cursor: crosshair;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="400"></canvas>

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

    let x = 100;
    let y = 100;
    let easing = 0.05;

    // Start mouse position in center initially
    let mouseX = canvas.width / 2;
    let mouseY = canvas.height / 2;

    canvas.addEventListener('mousemove', (e) => {
      const rect = canvas.getBoundingClientRect();
      mouseX = e.clientX - rect.left;
      mouseY = e.clientY - rect.top;
    });

    function draw() {
      ctx.clearRect(0, 0, canvas.width, canvas.height);
      ctx.beginPath();
      ctx.arc(x, y, 20, 0, Math.PI * 2);
```

```
      ctx.fillStyle = '#f39c12';
      ctx.fill();
    }

    function update() {
      const dx = mouseX - x;
      const dy = mouseY - y;

      x += dx * easing;
      y += dy * easing;

      draw();
      requestAnimationFrame(update);
    }

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

Behavior Tuning

- **Lower easing values** (e.g., `0.02`) produce slower, smoother animations.
- **Higher values** (e.g., `0.2`) make the object respond quickly but can result in overshooting if other physics (like inertia) are introduced later.

You can also stop the motion entirely when the distance to the target is very small, which is useful in snapping UI elements into place.

Practical Use

Proportional motion is the basis for many animation behaviors:

- Tooltip or UI element following a cursor
- Object chasing a target in a game
- Animations that ease into a final state
- Smooth camera panning or scrolling

Because it's simple and requires no velocity tracking or acceleration, it's often the first easing technique used in real-time animation systems.

In upcoming sections, we'll explore how proportional motion extends into more expressive models like advanced easing and spring dynamics.

12.2 Simple and Advanced Easing

Easing defines how an animation progresses over time. Rather than moving at a constant speed, easing functions modify velocity to create more natural and engaging motion. Whether you're sliding a panel into view, fading elements, or animating characters, easing helps make transitions feel intentional and expressive.

Simple Easing Functions

At the most basic level, easing functions are mathematical formulas that take an input time value (usually between 0 and 1) and return a modified output—also between 0 and 1. This output determines the progress of the animation at a given time.

Here are the most common simple easing types:

- **Linear**: Constant speed throughout.

```
function linear(t) {
  return t;
}
```

  Use linear motion when you want a mechanical, uniform pace. But for natural effects, it's often too stiff.

- **Ease-In**: Starts slow, speeds up.

```
function easeIn(t) {
  return t * t;
}
```

  Great for things accelerating into motion, like a button scaling up on hover.

- **Ease-Out**: Starts fast, slows down.

```
function easeOut(t) {
  return 1 - (1 - t) * (1 - t);
}
```

  Works well when an object comes to a gentle stop.

- **Ease-In-Out**: Starts slow, speeds up, then slows down again.

```
function easeInOut(t) {
  return t < 0.5
    ? 2 * t * t
    : 1 - Math.pow(-2 * t + 2, 2) / 2;
}
```

  Often used in smooth UI transitions and page scrolls.

Implementing Easing in Animation

Here's how to apply an easing function in practice. Suppose we want to animate an object from point A to B in one second:

```
let startTime = null;
let duration = 1000;
let startX = 100;
let endX = 400;

function animate(timestamp) {
  if (!startTime) startTime = timestamp;
  let elapsed = timestamp - startTime;
  let t = Math.min(elapsed / duration, 1); // normalized time
  let easedT = easeOut(t);
  let x = startX + (endX - startX) * easedT;
```

```
  draw(x);
  if (t < 1) requestAnimationFrame(animate);
}
```

This example uses `easeOut` to move an object from 100 to 400 pixels on the X-axis, gradually slowing as it reaches the end.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Easing Animation Example</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="150"></canvas>

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

    let startTime = null;
    const duration = 1000; // 1 second
    const startX = 100;
    const endX = 400;
    const y = canvas.height / 2;
    const radius = 20;

    // Ease out cubic function (fast start, slow end)
    function easeOut(t) {
      return 1 - Math.pow(1 - t, 3);
    }

    function draw(x) {
      ctx.clearRect(0, 0, canvas.width, canvas.height);
      ctx.beginPath();
      ctx.arc(x, y, radius, 0, Math.PI * 2);
      ctx.fillStyle = '#e67e22';
      ctx.fill();
    }
```

```
    function animate(timestamp) {
      if (!startTime) startTime = timestamp;
      let elapsed = timestamp - startTime;
      let t = Math.min(elapsed / duration, 1); // normalize to [0,1]

      let easedT = easeOut(t);
      let x = startX + (endX - startX) * easedT;

      draw(x);

      if (t < 1) {
        requestAnimationFrame(animate);
      }
    }

    requestAnimationFrame(animate);
  </script>
</body>
</html>
```

Advanced Easing with Cubic Bezier

CSS and JavaScript both support cubic Bézier curves, allowing fine-tuned control over easing. A cubic Bézier easing is defined using four control points:

```
transition-timing-function: cubic-bezier(0.42, 0, 0.58, 1); /* ease-in-out */
```

In JavaScript, libraries like d3-ease or GSAP support Bézier and many pre-defined easing functions, such as `bounce`, `elastic`, and `back`.

Here's an example using GSAP:

```
gsap.to(object, {
  duration: 1,
  x: 400,
  ease: "bounce.out"
});
```

This would cause the object to bounce to its destination, mimicking physical elasticity.

Choosing the Right Easing

Each easing style creates a different emotional or physical feel:

- **Linear**: Machine-like
- **Ease-In**: Gaining energy
- **Ease-Out**: Slowing to rest
- **Bounce/Elastic**: Playful, physical
- **Custom Bézier**: Tailored motion, branding

Understanding and applying easing functions allows you to elevate your animations from merely functional to fluid, engaging, and characterful. Up next, we'll explore how spring dynamics extend these ideas into physically reactive systems.

12.3 Springing in One and Two Dimensions

Spring-based motion simulates the natural oscillations we see in the real world—like a weight on a spring or a bobblehead toy. In animation, spring physics creates dynamic, lifelike responses as objects stretch, bounce, and settle into place. This section explores how to implement springing behavior in both one and two dimensions.

Understanding Spring Motion

At its core, spring motion is driven by **Hooke's Law**:

`F = -k * x`

Where:

- `F` is the restoring force
- `k` is the spring constant (stiffness)
- `x` is the displacement from the rest position

This force pulls the object back toward its resting position and can be combined with **damping** (resistance) to slow the motion over time:

`acceleration = -spring * (position - target) - damping * velocity`

This formula applies both in one dimension (horizontal/vertical) and in two dimensions (X and Y).

Springing in One Dimension

Let's animate an object moving along the X-axis toward a target using spring physics:

```
let x = 0;
let vx = 0;
let target = 300;
let spring = 0.1;
let damping = 0.8;

function update() {
  let force = -spring * (x - target);
  vx += force;
  vx *= damping;
  x += vx;

  draw(x); // draw the object at position x
  requestAnimationFrame(update);
}
```

In this code:

- `force` pulls the object toward the target.
- `vx` is the velocity, updated by the force and reduced by damping.
- `x` is the position, updated by velocity.

This produces a natural springy movement where the object overshoots the target, bounces back, and gradually settles.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>1D Spring Animation</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="150"></canvas>

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

    let x = 0;
    let vx = 0;
    const target = 300;
    const spring = 0.1;
    const damping = 0.8;
    const y = canvas.height / 2;
    const radius = 20;

    function draw(x) {
      ctx.clearRect(0, 0, canvas.width, canvas.height);
      ctx.beginPath();
      ctx.arc(x, y, radius, 0, Math.PI * 2);
      ctx.fillStyle = '#1abc9c';
      ctx.fill();

      // Draw target position
      ctx.beginPath();
      ctx.moveTo(target, 0);
      ctx.lineTo(target, canvas.height);
      ctx.strokeStyle = '#e74c3c';
      ctx.lineWidth = 2;
      ctx.setLineDash([5, 5]);
      ctx.stroke();
      ctx.setLineDash([]);
    }

    function update() {
```

```
      const force = -spring * (x - target);
      vx += force;
      vx *= damping;
      x += vx;

      draw(x);
      requestAnimationFrame(update);
    }

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

Springing in Two Dimensions

To animate a spring motion in 2D (X and Y), you simply apply the same logic to each axis independently:

```
let pos = { x: 100, y: 100 };
let vel = { x: 0, y: 0 };
let target = { x: 400, y: 300 };
let spring = 0.1;
let damping = 0.85;

function update() {
  // X-axis
  let fx = -spring * (pos.x - target.x);
  vel.x += fx;
  vel.x *= damping;
  pos.x += vel.x;

  // Y-axis
  let fy = -spring * (pos.y - target.y);
  vel.y += fy;
  vel.y *= damping;
  pos.y += vel.y;

  draw(pos.x, pos.y);
  requestAnimationFrame(update);
}
```

Now the object moves naturally toward the target point in 2D space with bouncing motion in both axes. The result feels more alive and physically believable compared to basic interpolation.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>2D Spring Animation</title>
  <style>
    body {
      margin: 0;
```

```
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="400"></canvas>

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

    let pos = { x: 100, y: 100 };
    let vel = { x: 0, y: 0 };
    const target = { x: 400, y: 300 };
    const spring = 0.1;
    const damping = 0.85;
    const radius = 20;

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

      // Draw target position
      ctx.beginPath();
      ctx.arc(target.x, target.y, 10, 0, Math.PI * 2);
      ctx.fillStyle = '#e74c3c';
      ctx.fill();

      // Draw moving object
      ctx.beginPath();
      ctx.arc(x, y, radius, 0, Math.PI * 2);
      ctx.fillStyle = '#1abc9c';
      ctx.fill();
    }

    function update() {
      // X-axis spring force and update
      let fx = -spring * (pos.x - target.x);
      vel.x += fx;
      vel.x *= damping;
      pos.x += vel.x;

      // Y-axis spring force and update
      let fy = -spring * (pos.y - target.y);
      vel.y += fy;
      vel.y *= damping;
      pos.y += vel.y;

      draw(pos.x, pos.y);
```

```
      requestAnimationFrame(update);
    }

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

Tuning Spring Behavior

You can fine-tune spring effects by adjusting the constants:

- **Spring constant (`k`)** controls how strong the pull is. Higher values make it snap faster.
- **Damping** controls how quickly the motion slows. Low damping = more bounce; high damping = quicker stop.

For example:

- `spring = 0.05`, `damping = 0.95` → soft, floaty motion.
- `spring = 0.2`, `damping = 0.7` → quick, snappy motion.

Use Cases

Springing is ideal for:

- UI elements like modals or buttons that "bounce" into place.
- Game objects with rubbery or elastic behavior.
- Physics-based simulations like dragging objects with tension.

In the next section, we'll take springing even further with **target following** and **chaining** multiple springs for connected effects.

Chapter 13.

Easing and Springing 2

1. Target Following and Chaining Springs
2. Springing Multiple Objects

13 Easing and Springing 2

13.1 Target Following and Chaining Springs

Chaining springs opens the door to rich, lifelike animations where multiple objects follow each other with fluid, spring-like behavior. This technique simulates systems such as snake-like movement, multi-segment tails, ropes, or even a train of bubbles. Each object in the chain springs toward the position of the one ahead of it, creating a cascading, elastic motion.

Basic Spring Target Following

Before we build a chain, let's recap how a single object can follow a target with a spring:

```
let pos = { x: 100, y: 100 };
let vel = { x: 0, y: 0 };
let target = { x: 0, y: 0 };
let spring = 0.1;
let damping = 0.8;

canvas.addEventListener('mousemove', (e) => {
  target.x = e.clientX;
  target.y = e.clientY;
});

function update() {
  let fx = -spring * (pos.x - target.x);
  vel.x += fx;
  vel.x *= damping;
  pos.x += vel.x;

  let fy = -spring * (pos.y - target.y);
  vel.y += fy;
  vel.y *= damping;
  pos.y += vel.y;

  drawCircle(pos.x, pos.y);
  requestAnimationFrame(update);
}
```

This code moves a dot smoothly toward the mouse with spring dynamics.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Spring Target Following</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
```

```
      height: 100vh;
      user-select: none;
      cursor: crosshair;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="400"></canvas>

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

    let pos = { x: 100, y: 100 };
    let vel = { x: 0, y: 0 };
    let target = { x: 100, y: 100 }; // start at same spot
    const spring = 0.1;
    const damping = 0.8;
    const radius = 20;

    canvas.addEventListener('mousemove', (e) => {
      const rect = canvas.getBoundingClientRect();
      target.x = e.clientX - rect.left;
      target.y = e.clientY - rect.top;
    });

    function drawCircle(x, y) {
      ctx.clearRect(0, 0, canvas.width, canvas.height);
      ctx.beginPath();
      ctx.arc(x, y, radius, 0, Math.PI * 2);
      ctx.fillStyle = '#3498db';
      ctx.fill();
    }

    function update() {
      let fx = -spring * (pos.x - target.x);
      vel.x += fx;
      vel.x *= damping;
      pos.x += vel.x;

      let fy = -spring * (pos.y - target.y);
      vel.y += fy;
      vel.y *= damping;
      pos.y += vel.y;

      drawCircle(pos.x, pos.y);
      requestAnimationFrame(update);
    }

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

Chaining Springs for Linked Motion

Now, let's scale this up by making several segments, where each one follows the position of the one before it:

```
const numSegments = 10;
const segments = [];
for (let i = 0; i < numSegments; i++) {
  segments.push({
    pos: { x: 0, y: 0 },
    vel: { x: 0, y: 0 }
  });
}

let spring = 0.15;
let damping = 0.75;
let target = { x: 0, y: 0 };

canvas.addEventListener('mousemove', (e) => {
  target.x = e.clientX;
  target.y = e.clientY;
});

function update() {
  let lead = target;

  for (let i = 0; i < segments.length; i++) {
    let seg = segments[i];

    let fx = -spring * (seg.pos.x - lead.x);
    seg.vel.x += fx;
    seg.vel.x *= damping;
    seg.pos.x += seg.vel.x;

    let fy = -spring * (seg.pos.y - lead.y);
    seg.vel.y += fy;
    seg.vel.y *= damping;
    seg.pos.y += seg.vel.y;

    lead = seg.pos;
  }

  drawChain(segments);
  requestAnimationFrame(update);
}
```

Each segment springs toward the previous one's position (`lead`). This chaining behavior produces a fluid and natural animation that feels like a creature, rope, or even a drag trail.

Drawing the Chain

For visual feedback, draw a line or circle at each segment:

```
function drawChain(segments) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.beginPath();
  ctx.moveTo(segments[0].pos.x, segments[0].pos.y);

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

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

  ctx.strokeStyle = "#00aaff";
  ctx.lineWidth = 4;
  ctx.stroke();

  segments.forEach(seg => {
    ctx.beginPath();
    ctx.arc(seg.pos.x, seg.pos.y, 5, 0, Math.PI * 2);
    ctx.fillStyle = "#ffaa00";
    ctx.fill();
  });
}
```

This results in a smooth rope or tail-like effect trailing your cursor.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Chaining Springs Example</title>
  <style>
    body {
      margin: 0;
      background: #111;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      user-select: none;
      cursor: crosshair;
    }
    canvas {
      background: #222;
      border: 2px solid #444;
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="400"></canvas>

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

    const numSegments = 10;
    const segments = [];
    for (let i = 0; i < numSegments; i++) {
      segments.push({
        pos: { x: canvas.width / 2, y: canvas.height / 2 },
        vel: { x: 0, y: 0 }
      });
    }
```

```
const spring = 0.15;
const damping = 0.75;
const target = { x: canvas.width / 2, y: canvas.height / 2 };

canvas.addEventListener('mousemove', (e) => {
  const rect = canvas.getBoundingClientRect();
  target.x = e.clientX - rect.left;
  target.y = e.clientY - rect.top;
});

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

  // Draw connecting line
  ctx.beginPath();
  ctx.moveTo(segments[0].pos.x, segments[0].pos.y);
  for (let i = 1; i < segments.length; i++) {
    ctx.lineTo(segments[i].pos.x, segments[i].pos.y);
  }
  ctx.strokeStyle = "#00aaff";
  ctx.lineWidth = 4;
  ctx.stroke();

  // Draw segment circles
  segments.forEach(seg => {
    ctx.beginPath();
    ctx.arc(seg.pos.x, seg.pos.y, 5, 0, Math.PI * 2);
    ctx.fillStyle = "#ffaa00";
    ctx.fill();
  });
}

function update() {
  let lead = target;

  for (let i = 0; i < segments.length; i++) {
    const seg = segments[i];

    // Spring physics X
    let fx = -spring * (seg.pos.x - lead.x);
    seg.vel.x += fx;
    seg.vel.x *= damping;
    seg.pos.x += seg.vel.x;

    // Spring physics Y
    let fy = -spring * (seg.pos.y - lead.y);
    seg.vel.y += fy;
    seg.vel.y *= damping;
    seg.pos.y += seg.vel.y;

    lead = seg.pos;
  }

  drawChain(segments);
  requestAnimationFrame(update);
}

update();
```

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

Advanced Behaviors

You can take this further by:

- Varying spring and damping values per segment for a ripple effect.
- Delaying updates per segment to create a lagging motion.
- Attaching objects to physics-based simulations or characters for dynamic accessories or hair.

Real-World Applications

Chained spring systems are popular in:

- Character animation (tails, arms, ropes).
- UI design (drag trails, loading indicators).
- Games (enemy chains, particles, effects).

This approach not only adds realism but also emotional depth—animated chains feel alive and reactive. In the next section, we'll use this concept to animate **multiple spring-based objects simultaneously** for even richer effects.

13.2 Springing Multiple Objects

Spring motion becomes even more powerful and dynamic when applied to multiple objects—whether they're independent (each with their own spring behavior) or interconnected (responding to shared forces). In interactive animations and games, simulating many spring-based motions simultaneously requires careful design for performance and visual coherence.

Independent Spring Objects

Let's begin with independent objects, such as a set of UI elements or particles that spring toward different targets.

Each object maintains its own position, velocity, and target, and is updated using the spring physics formula:

```
class SpringObject {
  constructor(x, y) {
    this.pos = { x, y };
    this.vel = { x: 0, y: 0 };
    this.target = { x, y };
  }

  update(spring = 0.1, damping = 0.8) {
    let fx = -spring * (this.pos.x - this.target.x);
```

```
    this.vel.x += fx;
    this.vel.x *= damping;
    this.pos.x += this.vel.x;

    let fy = -spring * (this.pos.y - this.target.y);
    this.vel.y += fy;
    this.vel.y *= damping;
    this.pos.y += this.vel.y;
  }

  draw(ctx) {
    ctx.beginPath();
    ctx.arc(this.pos.x, this.pos.y, 5, 0, Math.PI * 2);
    ctx.fillStyle = "#33ccff";
    ctx.fill();
  }
}

// Create multiple springs
const springs = [];
for (let i = 0; i < 20; i++) {
  springs.push(new SpringObject(Math.random() * canvas.width, Math.random() * canvas.height));
}
```

Each object can be assigned a unique or shared target—for example, they can follow the mouse or disperse to grid positions.

```
canvas.addEventListener('mousemove', (e) => {
  springs.forEach(s => {
    s.target.x = e.clientX + Math.random() * 100 - 50;
    s.target.y = e.clientY + Math.random() * 100 - 50;
  });
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Independent Spring Objects</title>
  <style>
    body {
      margin: 0;
      background: #111;
      overflow: hidden;
      cursor: crosshair;
    }
    canvas {
      display: block;
      background: #222;
    }
  </style>
</head>
<body>
  <canvas id="canvas"></canvas>
```

```
<script>
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');
  canvas.width = window.innerWidth;
  canvas.height = window.innerHeight;

  class SpringObject {
    constructor(x, y) {
      this.pos = { x, y };
      this.vel = { x: 0, y: 0 };
      this.target = { x, y };
    }

    update(spring = 0.1, damping = 0.8) {
      let fx = -spring * (this.pos.x - this.target.x);
      this.vel.x += fx;
      this.vel.x *= damping;
      this.pos.x += this.vel.x;

      let fy = -spring * (this.pos.y - this.target.y);
      this.vel.y += fy;
      this.vel.y *= damping;
      this.pos.y += this.vel.y;
    }

    draw(ctx) {
      ctx.beginPath();
      ctx.arc(this.pos.x, this.pos.y, 6, 0, Math.PI * 2);
      ctx.fillStyle = "#33ccff";
      ctx.fill();
    }
  }

  const springs = [];
  for (let i = 0; i < 20; i++) {
    const x = Math.random() * canvas.width;
    const y = Math.random() * canvas.height;
    springs.push(new SpringObject(x, y));
  }

  canvas.addEventListener('mousemove', (e) => {
    const rect = canvas.getBoundingClientRect();
    const mouseX = e.clientX - rect.left;
    const mouseY = e.clientY - rect.top;

    springs.forEach(s => {
      s.target.x = mouseX + (Math.random() * 100 - 50);
      s.target.y = mouseY + (Math.random() * 100 - 50);
    });
  });

  function animate() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    for (const s of springs) {
      s.update();
      s.draw(ctx);
    }
    requestAnimationFrame(animate);
```

```
    }

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

Connected Spring Systems

To simulate more organic systems like soft bodies, flags, or tentacles, objects can be connected in spring chains, sharing forces or positions with one another. Each object becomes a node in a system of constraints.

For example, a basic two-dimensional soft grid can use springs connecting neighbors (left, right, top, bottom) with rest lengths and spring constants. This is conceptually similar to cloth or net simulation.

Performance Considerations

When animating many springing objects, consider the following:

- **Update rate:** Reduce the number of updates per frame, or use `requestAnimationFrame` throttling.
- **Batch drawing:** Group draw calls and minimize context switches in `canvas`.
- **Garbage collection:** Reuse object instances rather than constantly creating new ones.
- **Spatial partitioning (advanced):** For connected systems, only update segments that are visible or active.

Coordination Tips

To synchronize spring behaviors:

- Use global `spring` and `damping` variables to control all objects uniformly.
- Introduce a stagger or delay based on index for waves or ripple effects.
- Use distance-based modifiers—objects closer to the target respond faster or stronger.

```
springs.forEach((s, i) => {
  let dist = Math.hypot(s.target.x - s.pos.x, s.target.y - s.pos.y);
  let spring = 0.05 + 0.05 * Math.min(dist / 200, 1);
  s.update(spring);
});
```

With proper structure, you can animate dozens—or even hundreds—of springing objects interactively. This sets the foundation for dynamic environments, expressive interfaces, and natural motion. In the next section, we'll formalize the mathematics behind easing and springing for maximum control.

Chapter 14.

Collision Detection 1

1. Basic Collision Detection Methods
2. Geometric Hit Testing
3. Distance-Based Detection

14 Collision Detection 1

14.1 Basic Collision Detection Methods

Collision detection is a fundamental component of interactive animation and game development. It allows you to determine when objects in your scene intersect, touch, or overlap—events that often trigger visual changes, physics responses, or gameplay mechanics. Whether you're making a bouncing ball, platformer, or space shooter, reliable collision detection is critical.

In this section, we introduce the simplest and most commonly used methods: **bounding box detection** and **overlap testing**.

Axis-Aligned Bounding Box (AABB) Collision

The most straightforward collision method uses rectangles to enclose objects and checks whether those rectangles overlap. This is called **Axis-Aligned Bounding Box (AABB)** collision, and it works great for rectangular or roughly box-shaped sprites.

Here's a basic AABB collision function:

```
function isColliding(a, b) {
  return (
    a.x < b.x + b.width &&
    a.x + a.width > b.x &&
    a.y < b.y + b.height &&
    a.y + a.height > b.y
  );
}
```

Each object must have `x`, `y`, `width`, and `height` properties. This function returns `true` if their boxes intersect. It's efficient and widely used in 2D games.

Circle Overlap Detection

For round objects like balls or bubbles, circle-based collision detection is more accurate and still fast. It involves comparing the distance between centers to the sum of radii.

```
function isCircleColliding(a, b) {
  let dx = a.x - b.x;
  let dy = a.y - b.y;
  let distance = Math.hypot(dx, dy);
  return distance < a.radius + b.radius;
}
```

This technique is ideal for simulations involving projectiles, particles, or physics-based games with round entities.

Pixel-Level and Complex Shape Detection

While bounding boxes and circles are efficient, sometimes you need more precision—especially with irregular or detailed shapes. More advanced methods include:

- **Per-pixel detection**: Compare pixel data between overlapping regions.
- **Polygon collision**: Use the Separating Axis Theorem (SAT) for convex shapes.

However, these are more computationally expensive and should only be used when simpler methods aren't sufficient.

Why Collision Detection Matters

Collision detection drives interactive feedback in animation:

- **Bouncing**: Reverse velocity when a shape hits a wall.
- **Damage**: Apply health reduction when enemies hit the player.
- **Events**: Trigger sounds, particle effects, or level transitions on contact.

Without it, animated elements exist in isolation with no interaction or consequence.

Practical Example

Here's a quick application of AABB:

```
let player = { x: 50, y: 50, width: 20, height: 20 };
let block = { x: 100, y: 100, width: 30, height: 30 };

if (isColliding(player, block)) {
  console.log("Collision!");
}
```

This approach can scale to multiple objects by looping through arrays and checking for overlaps.

These basic collision techniques lay the foundation for more dynamic interactions in your animations. In the next sections, we'll explore geometric, distance-based, and spring-driven reactions in more depth.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>AABB Collision Detection</title>
  <style>
    body {
      margin: 0;
      background: #111;
    }
    canvas {
      display: block;
      background: #222;
      cursor: pointer;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="400"></canvas>
  <script>
```

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

const player = { x: 50, y: 50, width: 40, height: 40 };
const block = { x: 250, y: 150, width: 60, height: 60 };

let dragging = false;
let offsetX = 0;
let offsetY = 0;

function isColliding(a, b) {
  return (
    a.x < b.x + b.width &&
    a.x + a.width > b.x &&
    a.y < b.y + b.height &&
    a.y + a.height > b.y
  );
}

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

  // Draw static block
  ctx.fillStyle = "#444";
  ctx.fillRect(block.x, block.y, block.width, block.height);

  // Change color on collision
  if (isColliding(player, block)) {
    ctx.fillStyle = "#ff3333";
  } else {
    ctx.fillStyle = "#33ccff";
  }

  // Draw player
  ctx.fillRect(player.x, player.y, player.width, player.height);
}

canvas.addEventListener('mousedown', (e) => {
  const rect = canvas.getBoundingClientRect();
  const mx = e.clientX - rect.left;
  const my = e.clientY - rect.top;

  if (
    mx >= player.x &&
    mx <= player.x + player.width &&
    my >= player.y &&
    my <= player.y + player.height
  ) {
    dragging = true;
    offsetX = mx - player.x;
    offsetY = my - player.y;
  }
});

canvas.addEventListener('mousemove', (e) => {
  if (dragging) {
    const rect = canvas.getBoundingClientRect();
    player.x = e.clientX - rect.left - offsetX;
```

```
        player.y = e.clientY - rect.top - offsetY;
        draw();
      }
    });

    canvas.addEventListener('mouseup', () => {
      dragging = false;
    });

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

14.2 Geometric Hit Testing

Geometric hit testing is the process of determining whether two or more shapes intersect, touch, or overlap using their geometric properties. This is a core part of collision detection in games, simulations, and interactive animations. In this section, we'll dive into several geometric techniques for hit testing involving **rectangles**, **circles**, and **polygons**, with accompanying explanations and code samples.

Rectangle-Rectangle Collision (AABB)

Axis-Aligned Bounding Boxes (AABBs) are rectangles that do not rotate. They are easy to test for overlap using comparison of edges:

```
function rectsIntersect(r1, r2) {
  return (
    r1.x < r2.x + r2.width &&
    r1.x + r1.width > r2.x &&
    r1.y < r2.y + r2.height &&
    r1.y + r1.height > r2.y
  );
}
```

Use case: Platformers, where players and platforms are usually box-shaped. **Demo idea:** Two draggable rectangles that change color when intersecting.

Circle-Circle Collision

Circle hit testing is based on the distance between centers compared to the sum of radii:

```
function circlesIntersect(c1, c2) {
  const dx = c1.x - c2.x;
  const dy = c1.y - c2.y;
  const distance = Math.hypot(dx, dy);
  return distance < c1.radius + c2.radius;
}
```

Use case: Ball bouncing, particle systems, bubble-popping games. **Demo idea:** Move one

circle with the mouse and highlight collision when it touches another.

Rectangle-Circle Collision

To test a circle against a rectangle, find the closest point on the rectangle to the circle center, then check if that point is within the circle's radius:

```
function rectCircleCollide(circle, rect) {
  const closestX = Math.max(rect.x, Math.min(circle.x, rect.x + rect.width));
  const closestY = Math.max(rect.y, Math.min(circle.y, rect.y + rect.height));
  const dx = circle.x - closestX;
  const dy = circle.y - closestY;
  return dx * dx + dy * dy < circle.radius * circle.radius;
}
```

Use case: A circular projectile hitting a rectangular target. **Demo idea:** Circle thrown at a wall, detecting collision and changing bounce direction.

Polygon Collision (SAT)

Polygon-polygon collision detection typically uses the **Separating Axis Theorem (SAT)**. The idea is: if you can find an axis along which the projections of two polygons do **not** overlap, then the polygons are not colliding.

Implementing SAT is more complex, but here's a high-level outline:

1. For each edge in both polygons:
    - Compute the **normal vector** (perpendicular axis).
    - Project both polygons onto that axis.
    - If projections do not overlap → no collision.
2. If all projections overlap → collision detected.

SAT works best for **convex polygons**.

For simpler convex shapes like triangles or pentagons, you can use a JavaScript library like SAT.js, or implement a basic SAT engine for educational purposes.

Use case: Complex characters or objects that rotate and deform, such as spacecraft or vehicles. **Demo idea:** Rotatable polygons with arrows — turn one shape and detect collision when edges intersect.

Point-in-Polygon Test

Another common task is detecting whether a point (like a mouse click) is inside a polygon. The **ray-casting algorithm** is the classic method:

```
function pointInPolygon(point, vertices) {
  let inside = false;
  for (let i = 0, j = vertices.length - 1; i < vertices.length; j = i++) {
    const xi = vertices[i][0], yi = vertices[i][1];
    const xj = vertices[j][0], yj = vertices[j][1];

    const intersect = ((yi > point.y) !== (yj > point.y)) &&
```

```
                      (point.x < ((xj - xi) * (point.y - yi)) / (yj - yi) + xi);
    if (intersect) inside = !inside;
  }
  return inside;
}
```

Use case: Clicking or tapping on irregular buttons or icons. **Demo idea:** Polygonal buttons that highlight when hovered or clicked.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Point in Polygon</title>
  <style>
    body {
      margin: 0;
      background: #111;
    }
    canvas {
      display: block;
      background: #222;
    }
  </style>
</head>
<body>
  <canvas id="canvas" width="600" height="400"></canvas>
  <script>
    const canvas = document.getElementById("canvas");
    const ctx = canvas.getContext("2d");

    // A simple polygon (pentagon)
    const polygon = [
      [200, 100],
      [300, 80],
      [370, 180],
      [270, 300],
      [130, 220]
    ];

    let isInside = false;

    function pointInPolygon(point, vertices) {
      let inside = false;
      for (let i = 0, j = vertices.length - 1; i < vertices.length; j = i++) {
        const xi = vertices[i][0], yi = vertices[i][1];
        const xj = vertices[j][0], yj = vertices[j][1];

        const intersect = ((yi > point.y) !== (yj > point.y)) &&
                          (point.x < ((xj - xi) * (point.y - yi)) / (yj - yi) + xi);
        if (intersect) inside = !inside;
      }
      return inside;
    }
```

```
    function drawPolygon(poly, color) {
      ctx.beginPath();
      ctx.moveTo(poly[0][0], poly[0][1]);
      for (let i = 1; i < poly.length; i++) {
        ctx.lineTo(poly[i][0], poly[i][1]);
      }
      ctx.closePath();
      ctx.fillStyle = color;
      ctx.fill();
      ctx.strokeStyle = "#fff";
      ctx.stroke();
    }

    function draw() {
      ctx.clearRect(0, 0, canvas.width, canvas.height);
      drawPolygon(polygon, isInside ? "#00cc44" : "#cc3333");
    }

    canvas.addEventListener("click", (e) => {
      const rect = canvas.getBoundingClientRect();
      const mouse = {
        x: e.clientX - rect.left,
        y: e.clientY - rect.top
      };

      isInside = pointInPolygon(mouse, polygon);
      draw();
    });

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

14.2.1 Summary

Shape Combo	Method	Efficiency	Use Case
Rect vs. Rect	AABB Check	Very fast	Platform collisions
Circle vs. Circle	Radius + Distance	Fast	Particle or ball physics
Rect vs. Circle	Closest point logic	Moderate	Projectile hits
Polygon vs. Polygon	SAT	Slower	Complex shapes/rotations
Point vs. Polygon	Ray Casting	Moderate	UI hit detection

In animation and game logic, geometric hit testing enables interactive realism, precision response, and complex behaviors. The method you choose depends on the shape and performance needs of your application.

14.3 Distance-Based Detection

Distance-based detection is one of the most intuitive and efficient methods for identifying collisions between objects, especially when dealing with **circles**, **points**, and **radial zones**. Unlike bounding box methods, this approach directly compares the spatial separation between objects, making it ideal for circular or fluid-like interactions where precise edge matching isn't required.

Understanding the Principle

The core idea is simple: if the distance between two objects is less than a certain threshold, a collision has occurred.

For two circles:

- Collision occurs when the distance between their centers is **less than the sum of their radii**.

For a point and a circle (like a cursor entering a bubble):

- Collision occurs when the distance between the point and the circle's center is **less than the circle's radius**.

Calculating Distance

Use the **Pythagorean theorem** to calculate distance between two points:

```
function getDistance(x1, y1, x2, y2) {
  const dx = x2 - x1;
  const dy = y2 - y1;
  return Math.sqrt(dx * dx + dy * dy);
}
```

For performance-critical applications, consider comparing **squared distance** to avoid using `Math.sqrt()`:

```
function isWithinDistance(obj1, obj2, maxDistance) {
  const dx = obj2.x - obj1.x;
  const dy = obj2.y - obj1.y;
  return (dx * dx + dy * dy) < maxDistance * maxDistance;
}
```

Circle-Circle Collision Example

```
function circlesCollide(c1, c2) {
  const dx = c2.x - c1.x;
  const dy = c2.y - c1.y;
  const distance = Math.sqrt(dx * dx + dy * dy);
  return distance < (c1.radius + c2.radius);
}
```

You can visualize this by animating two circles and detecting when they touch or overlap.

Point-Circle Detection Example

This is useful for detecting if a mouse click is inside a circular object:

```
function pointInCircle(px, py, circle) {
  const dx = px - circle.x;
  const dy = py - circle.y;
  return (dx * dx + dy * dy) < circle.radius * circle.radius;
}
```

You could use this to build clickable round buttons or highlight objects under the cursor.

Distance Thresholds for Triggers

Distance-based detection also enables **trigger zones**, where objects don't have to collide but only get close to activate an effect:

```
if (getDistance(player.x, player.y, enemy.x, enemy.y) < 150) {
  enemy.alert = true; // enemy notices player
}
```

This can be expanded for AI detection, proximity effects, or soft collisions.

Interactive Example: Proximity Activation

Imagine a floating orb that glows brighter as the player approaches. Use the inverse of the distance to determine glow intensity:

```
const distance = getDistance(player.x, player.y, orb.x, orb.y);
orb.intensity = Math.max(0, 1 - distance / maxDistance);
```

This creates a subtle but powerful illusion of interaction without physical touch.

14.3.1 Summary

Distance-based collision detection is best for:

- Circular objects (balls, particles)
- Point interactions (clicks, taps)
- Proximity-based behaviors (radar, activation zones)

It's computationally light and conceptually simple, but not suitable for non-circular or axis-aligned shapes. When used appropriately, it adds fluidity and realism to HTML5 animations with minimal overhead.

Chapter 15.

Collision Detection 2

1. Spring Reactions on Collision
2. Handling Multiple Object Collisions

15 Collision Detection 2

15.1 Spring Reactions on Collision

When objects collide in animations or games, a simple collision detection often isn't enough to make the interaction feel natural and dynamic. Instead of instantly stopping or just reversing direction, objects can respond with a **springy bounce or recoil effect** — mimicking how real-world objects behave with elasticity and energy transfer. This is where **spring physics** come into play.

What Are Spring Reactions?

Spring reactions simulate the force generated when two objects collide and then push away from each other, like compressing and releasing a spring. This force depends on how much the objects overlap (penetration depth) and how fast they are moving apart or together. Instead of a harsh stop, the spring force makes the motion smooth, bouncy, and natural.

The Physics Behind Springs

A simple spring force can be modeled using Hooke's Law:

$$F = -k \times x$$

Where:

- F is the restoring force applied by the spring
- k is the spring constant (stiffness)
- x is the displacement from the spring's rest position (penetration depth)

In collision terms, x is how much the objects overlap, and F is the force pushing them apart.

Applying Spring Forces on Collision

When two objects collide, you can:

1. **Calculate the overlap distance** — how deeply the objects intersect.
2. **Apply a force proportional to this overlap** pushing the objects apart.
3. **Adjust velocities** based on this force to simulate bounce and recoil.

This approach makes collisions feel elastic and prevents objects from sticking together.

Code Example: Simple Springy Collision Reaction

```
class Circle {
  constructor(x, y, radius, vx = 0, vy = 0) {
    this.x = x;
    this.y = y;
    this.radius = radius;
    this.vx = vx; // velocity x
    this.vy = vy; // velocity y
```

```
  }

  update() {
    this.x += this.vx;
    this.y += this.vy;
  }
}

function springCollision(c1, c2, k = 0.1) {
  const dx = c2.x - c1.x;
  const dy = c2.y - c1.y;
  const distance = Math.sqrt(dx * dx + dy * dy);
  const minDist = c1.radius + c2.radius;

  if (distance < minDist) {
    const overlap = minDist - distance;
    const nx = dx / distance; // normalized collision vector x
    const ny = dy / distance; // normalized collision vector y

    // Apply spring force proportional to overlap
    const force = k * overlap;

    // Adjust velocities to simulate bounce
    c1.vx -= force * nx;
    c1.vy -= force * ny;
    c2.vx += force * nx;
    c2.vy += force * ny;
  }
}

// Example usage
const ball1 = new Circle(100, 100, 20, 2, 1);
const ball2 = new Circle(130, 120, 20, -1, -1);

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

  springCollision(ball1, ball2);

  ball1.update();
  ball2.update();

  // Draw balls
  drawCircle(ball1);
  drawCircle(ball2);

  requestAnimationFrame(animate);
}
```

This example demonstrates two circles bouncing off each other smoothly by applying a spring force when they collide. The `k` value controls stiffness: higher values make the bounce sharper, while lower values create a softer spring effect.

Full runnable code:

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

```
<head>
  <meta charset="UTF-8">
  <title>Springy Circle Collision</title>
  <style>
    body {
      margin: 0;
      background: #111;
    }
    canvas {
      display: block;
      background: #222;
    }
  </style>
</head>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

class Circle {
  constructor(x, y, radius, vx = 0, vy = 0) {
    this.x = x;
    this.y = y;
    this.radius = radius;
    this.vx = vx;
    this.vy = vy;
  }

  update() {
    this.x += this.vx;
    this.y += this.vy;

    // Bounce off walls
    if (this.x - this.radius < 0 || this.x + this.radius > canvas.width) {
      this.vx *= -1;
    }
    if (this.y - this.radius < 0 || this.y + this.radius > canvas.height) {
      this.vy *= -1;
    }
  }

  draw(ctx) {
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, Math.PI * 2);
    ctx.fillStyle = "#33ccff";
    ctx.fill();
    ctx.strokeStyle = "#fff";
    ctx.stroke();
  }
}

function springCollision(c1, c2, k = 0.1) {
  const dx = c2.x - c1.x;
  const dy = c2.y - c1.y;
  const distance = Math.sqrt(dx * dx + dy * dy);
  const minDist = c1.radius + c2.radius;
```

```
  if (distance < minDist && distance > 0) {
    const overlap = minDist - distance;
    const nx = dx / distance;
    const ny = dy / distance;

    const force = k * overlap;

    // Apply spring force
    c1.vx -= force * nx;
    c1.vy -= force * ny;
    c2.vx += force * nx;
    c2.vy += force * ny;
  }
}

const ball1 = new Circle(150, 150, 30, 2, 1);
const ball2 = new Circle(350, 200, 30, -1, -1);

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

  springCollision(ball1, ball2);

  ball1.update();
  ball2.update();

  ball1.draw(ctx);
  ball2.draw(ctx);

  requestAnimationFrame(animate);
}

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

Enhancing Spring Reactions

To make spring reactions more realistic:

- **Damping** can be added to reduce velocity over time, simulating friction or energy loss.
- **Mass and inertia** can modify how much each object responds to the force.
- Springs can also be used for **attached objects** or joints, creating chain-like or flexible animations.

15.1.1 Summary

Using spring physics for collision reactions elevates your animations from stiff and unrealistic to lively and dynamic. By applying forces proportional to penetration depth and adjusting velocities accordingly, you simulate natural bounce and recoil effects. This approach enhances player immersion in games and creates satisfying interactive animations in any canvas-based

project.

15.2 Handling Multiple Object Collisions

When animating or building games with many moving objects, managing collisions efficiently becomes crucial. Detecting and responding to collisions between all pairs of objects can quickly become computationally expensive, especially as the number of objects grows. In this section, we'll explore techniques to detect and manage collisions among multiple objects while balancing performance and accuracy.

Naive Collision Detection: The O(n²) Approach

The simplest method to detect collisions among multiple objects is to check every possible pair. For **n** objects, that means checking $\frac{n(n-1)}{2}$ pairs each frame:

```
for (let i = 0; i < objects.length; i++) {
  for (let j = i + 1; j < objects.length; j++) {
    if (checkCollision(objects[i], objects[j])) {
      handleCollision(objects[i], objects[j]);
    }
  }
}
```

While straightforward, this approach is inefficient for large **n** because the number of checks grows quadratically. For example, with 1000 objects, you perform nearly 500,000 collision checks every frame — far too slow for real-time animation.

Optimized Strategies for Performance

To reduce unnecessary collision checks, several optimization techniques are commonly used:

Spatial Partitioning Spatial partitioning divides the animation space into regions (cells, grids, or trees) and only checks collisions between objects within the same or neighboring regions.

- **Uniform Grid**: The canvas is split into a grid of fixed-size cells. Each object is assigned to one or more cells based on its position. Only objects sharing cells are checked for collisions.
- **Quadtrees**: A hierarchical tree structure where the space subdivides recursively into four quadrants. Objects are placed in leaf nodes corresponding to their position, and collisions are checked only among objects in the same or adjacent nodes.

```
const cellSize = 100;
const grid = new Map();

function getCellKey(x, y) {
```

```
  return `${Math.floor(x / cellSize)}:${Math.floor(y / cellSize)}`;
}

function assignObjectsToGrid(objects) {
  grid.clear();
  for (const obj of objects) {
    const key = getCellKey(obj.x, obj.y);
    if (!grid.has(key)) grid.set(key, []);
    grid.get(key).push(obj);
  }
}

function detectCollisions(objects) {
  assignObjectsToGrid(objects);
  for (const cellObjects of grid.values()) {
    for (let i = 0; i < cellObjects.length; i++) {
      for (let j = i + 1; j < cellObjects.length; j++) {
        if (checkCollision(cellObjects[i], cellObjects[j])) {
          handleCollision(cellObjects[i], cellObjects[j]);
        }
      }
    }
  }
}
```

Example: Simple Uniform Grid This technique dramatically reduces the number of checks, especially when objects are evenly spread out.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Uniform Grid Collision</title>
  <style>
    canvas { background: #111; display: block; margin: 0 auto; }
  </style>
</head>
<body>
<canvas id="canvas" width="800" height="600"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

const cellSize = 100;
const grid = new Map();
const balls = [];
const ballCount = 100;

// Create random balls
for (let i = 0; i < ballCount; i++) {
  balls.push({
    x: Math.random() * canvas.width,
    y: Math.random() * canvas.height,
    vx: Math.random() * 4 - 2,
    vy: Math.random() * 4 - 2,
```

```
    radius: 10,
    color: "#33ccff"
  });
}

function getCellKey(x, y) {
  return `${Math.floor(x / cellSize)}:${Math.floor(y / cellSize)}`;
}

function assignObjectsToGrid(objects) {
  grid.clear();
  for (const obj of objects) {
    const key = getCellKey(obj.x, obj.y);
    if (!grid.has(key)) grid.set(key, []);
    grid.get(key).push(obj);
  }
}

function checkCollision(a, b) {
  const dx = a.x - b.x;
  const dy = a.y - b.y;
  const distSq = dx * dx + dy * dy;
  const minDist = a.radius + b.radius;
  return distSq < minDist * minDist;
}

function handleCollision(a, b) {
  a.color = "#ff4444";
  b.color = "#ff4444";
}

function detectCollisions(objects) {
  assignObjectsToGrid(objects);
  for (const cellObjects of grid.values()) {
    for (let i = 0; i < cellObjects.length; i++) {
      for (let j = i + 1; j < cellObjects.length; j++) {
        if (checkCollision(cellObjects[i], cellObjects[j])) {
          handleCollision(cellObjects[i], cellObjects[j]);
        }
      }
    }
  }
}

function update() {
  for (const b of balls) {
    b.x += b.vx;
    b.y += b.vy;

    if (b.x < b.radius || b.x > canvas.width - b.radius) b.vx *= -1;
    if (b.y < b.radius || b.y > canvas.height - b.radius) b.vy *= -1;

    b.color = "#33ccff"; // Reset color every frame
  }

  detectCollisions(balls);
}
```

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

  for (const b of balls) {
    ctx.beginPath();
    ctx.arc(b.x, b.y, b.radius, 0, Math.PI * 2);
    ctx.fillStyle = b.color;
    ctx.fill();
  }
}

function loop() {
  update();
  draw();
  requestAnimationFrame(loop);
}

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

Performance Considerations

- **Update Cost**: Maintaining spatial data structures (like quadtrees) adds overhead. For a small number of objects, naive checking may be faster.
- **Object Movement**: Fast-moving objects might cross multiple cells per frame, requiring careful updating to avoid missed collisions.
- **Collision Frequency**: Dense clusters might still require many checks, but spatial partitioning localizes those computations.

Practical Tips

- Start with naive detection for small or simple projects.
- Implement spatial partitioning as object count grows beyond a few dozen.
- Profile your code to identify bottlenecks.
- Use bounding volume checks (like bounding boxes or circles) before detailed collision tests to save computation.

15.2.1 Summary

Handling multiple object collisions efficiently requires balancing accuracy and performance. Naive pairwise checking is easy but scales poorly with object count. Spatial partitioning strategies such as uniform grids and quadtrees help reduce unnecessary collision checks by grouping nearby objects, leading to faster and more scalable collision detection. Combining these methods with bounding volume tests and efficient data structures ensures smooth animations and responsive interactions in complex scenes.

Chapter 16.

Rotation and Angular Bouncing

1. Simple and Advanced Coordinate Rotation
2. Bouncing Off Angled Surfaces
3. Dynamic and Optimized Code Techniques
4. Multi-Angle Bounce Handling

16 Rotation and Angular Bouncing

16.1 Simple and Advanced Coordinate Rotation

Rotation is a fundamental operation in animations, allowing you to spin objects or points around a specific origin smoothly. Understanding the math behind rotation helps you create visually compelling effects, from simple spinning shapes to complex angular motion.

The Basics: Rotating a Point Around the Origin

Consider a point (x, y) in 2D space that you want to rotate by an angle θ (in radians) around the origin $(0, 0)$. The new coordinates (x', y') are calculated using the rotation matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

Which expands to:

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta$$

This formula rotates the point counterclockwise by θ radians about the origin.

Rotating Around an Arbitrary Origin

Most times, you'll want to rotate a point or object around a pivot other than $(0, 0)$, such as the center of a shape or any custom point (ox, oy). The process involves:

1. **Translate** the point so the origin is at the pivot:

$$x_{trans} = x - ox$$

$$y_{trans} = y - oy$$

2. **Rotate** the translated point using the rotation matrix:

$$x'_{trans} = x_{trans}\cos\theta - y_{trans}\sin\theta$$

$$y'_{trans} = x_{trans}\sin\theta + y_{trans}\cos\theta$$

3. **Translate back** to the original coordinate system:

$$x' = x'_{trans} + ox$$

$$y' = y'_{trans} + oy$$

Step-by-Step Rotation in JavaScript

Here's a practical function to rotate a point (x, y) around an origin (ox, oy) by angle θ:

```
function rotatePoint(x, y, ox, oy, angle) {
  // Translate point to origin
  let dx = x - ox;
  let dy = y - oy;

  // Apply rotation matrix
  let cosA = Math.cos(angle);
  let sinA = Math.sin(angle);

  let rx = dx * cosA - dy * sinA;
  let ry = dx * sinA + dy * cosA;

  // Translate back
  return {
    x: rx + ox,
    y: ry + oy
  };
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Rotate Point Demo</title>
  <style>
    canvas {
      background: #111;
      display: block;
      margin: 40px auto;
      border: 2px solid #444;
    }
  </style>
</head>
<body>
<canvas id="canvas" width="500" height="500"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

// Rotation center
const origin = { x: canvas.width / 2, y: canvas.height / 2 };
// Point to rotate (initially to the right)
let point = { x: origin.x + 100, y: origin.y };
let angle = 0;
```

```
function rotatePoint(x, y, ox, oy, angle) {
  let dx = x - ox;
  let dy = y - oy;

  let cosA = Math.cos(angle);
  let sinA = Math.sin(angle);

  let rx = dx * cosA - dy * sinA;
  let ry = dx * sinA + dx * sinA + dy * cosA;

  return {
    x: rx + ox,
    y: ry + oy
  };
}

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

  // Draw origin
  ctx.beginPath();
  ctx.arc(origin.x, origin.y, 5, 0, Math.PI * 2);
  ctx.fillStyle = "#00ff99";
  ctx.fill();

  // Draw path circle
  ctx.beginPath();
  ctx.arc(origin.x, origin.y, 100, 0, Math.PI * 2);
  ctx.strokeStyle = "#444";
  ctx.stroke();

  // Draw rotating point
  ctx.beginPath();
  ctx.arc(point.x, point.y, 8, 0, Math.PI * 2);
  ctx.fillStyle = "#ffaa00";
  ctx.fill();
}

function animate() {
  point = rotatePoint(point.x, point.y, origin.x, origin.y, 0.05);
  draw();
  requestAnimationFrame(animate);
}

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

Advanced Techniques for Smooth Rotation

- **Using the Canvas `save()`, `translate()`, and `rotate()` Methods:** Instead of manually calculating rotated points, you can manipulate the canvas context transformation matrix to rotate whole objects:

```
ctx.save();
ctx.translate(ox, oy);       // Move origin to pivot point
```

```
ctx.rotate(angle);            // Rotate canvas
ctx.translate(-ox, -oy);      // Move origin back
// Draw your shape here
ctx.restore();
```

This method simplifies rotation for complex shapes and improves performance.

- **Incremental Rotation:** To animate rotation, increment the angle each frame smoothly:

```
let angle = 0;
function animate() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.save();
  ctx.translate(ox, oy);
  ctx.rotate(angle);
  ctx.translate(-ox, -oy);
  drawShape();
  ctx.restore();
  angle += 0.02;  // Adjust speed here
  requestAnimationFrame(animate);
}
animate();
```

- **Combining Rotation with Scaling and Translation:** You can chain transformations to create complex motion, always remembering the order matters: scale, then rotate, then translate.

Summary

Rotating points and objects involves applying rotation matrices to transform coordinates. While simple formulas work well for single points, leveraging canvas transformations provides a powerful, efficient way to rotate complex shapes smoothly. Understanding these fundamentals and advanced techniques lets you build rich animations with precise rotational control.

16.2 Bouncing Off Angled Surfaces

When an object collides with an angled or sloped surface, its bounce direction depends not just on its incoming velocity but also on the orientation of that surface. Understanding how to calculate these bounce directions is crucial for realistic animations, games, and physics simulations. This section explores the vector math behind bouncing and provides examples such as billiard ball collisions.

The Concept of Vector Reflection

The bounce direction after hitting an angled surface can be understood using **vector reflection**. When a moving object (like a ball) hits a surface, its velocity vector reflects off the surface according to the surface's **normal vector** — a perpendicular vector pointing outward from the surface.

Given:

- Incoming velocity vector $\mathbf{v}$
- Surface normal vector $\mathbf{n}$ (unit vector perpendicular to the surface)

The reflected velocity vector $\mathbf{r}$ is calculated by the formula:

$$\mathbf{r} = \mathbf{v} - 2(\mathbf{v} \cdot \mathbf{n})\mathbf{n}$$

Where $\mathbf{v} \cdot \mathbf{n}$ is the **dot product** of the velocity and normal vectors.

Breaking Down the Formula

- The dot product $\mathbf{v} \cdot \mathbf{n}$ measures how much of $\mathbf{v}$ is pointing in the direction of $\mathbf{n}$.
- Multiplying the dot product by $\mathbf{n}$ projects the velocity onto the normal.
- Doubling and subtracting this projection from the original velocity effectively "mirrors" the velocity about the surface normal, producing the bounce direction.

Real-World Example: Billiard Ball Bounce

Imagine a billiard ball rolling toward the edge of the table:

- The ball's velocity vector $\mathbf{v}$ points toward the rail.
- The rail has a normal vector $\mathbf{n}$ pointing directly away from the table edge.
- Upon impact, the ball bounces by reflecting its velocity vector about $\mathbf{n}$ using the formula above.
- This causes the ball to leave the rail at the same angle it approached, but in the opposite direction — a classic billiard bounce.

Code Example: Calculating Bounce Direction

Here's how you might implement this in JavaScript using vectors:

```
function dotProduct(v1, v2) {
  return v1.x * v2.x + v1.y * v2.y;
}

function reflectVector(v, n) {
  // Ensure normal is normalized (unit vector)
  const dot = dotProduct(v, n);
  return {
    x: v.x - 2 * dot * n.x,
    y: v.y - 2 * dot * n.y
  };
}

// Example usage:
let velocity = { x: 3, y: -4 };           // Incoming velocity
let surfaceNormal = { x: 0, y: 1 };       // Upward pointing normal (horizontal surface)

let bouncedVelocity = reflectVector(velocity, surfaceNormal);
console.log(bouncedVelocity);  // { x: 3, y: 4 }
```

In this example, the ball moving downward with velocity `(3, -4)` hits a flat horizontal surface with normal `(0, 1)` and bounces upward with velocity `(3, 4)`.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Bouncing Off Angled Surfaces</title>
  <style>
    body {
      background: #111;
      margin: 0;
      overflow: hidden;
    }
    canvas {
      display: block;
      margin: 0 auto;
      background: #222;
    }
  </style>
</head>
<body>
<canvas id="canvas" width="800" height="600"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

// Utility vector math
function dot(a, b) {
  return a.x * b.x + a.y * b.y;
}
function reflect(velocity, normal) {
  const d = dot(velocity, normal);
  return {
    x: velocity.x - 2 * d * normal.x,
    y: velocity.y - 2 * d * normal.y
  };
}
function normalize(v) {
  const len = Math.hypot(v.x, v.y);
  return { x: v.x / len, y: v.y / len };
}

// Ball
const ball = {
  x: 100,
  y: 100,
  radius: 10,
  vx: 4,
  vy: 2,
};

// Angled surfaces (as segments)
const surfaces = [
  { x1: 100, y1: 500, x2: 700, y2: 400 },  // sloped up
  { x1: 700, y1: 400, x2: 700, y2: 100 },  // vertical
```

```
  { x1: 700, y1: 100, x2: 100, y2: 100 },  // top horizontal
  { x1: 100, y1: 100, x2: 100, y2: 500 },  // left vertical
];

function updateBall() {
  ball.x += ball.vx;
  ball.y += ball.vy;

  // Check collisions with surfaces
  for (const s of surfaces) {
    // Line segment as vector
    const dx = s.x2 - s.x1;
    const dy = s.y2 - s.y1;

    // Surface normal (perpendicular)
    const normal = normalize({ x: -dy, y: dx });

    // From segment start to ball
    const px = ball.x - s.x1;
    const py = ball.y - s.y1;

    // Distance from point to line (dot with normal)
    const dist = px * normal.x + py * normal.y;

    // If close enough to surface (simple bounding)
    if (Math.abs(dist) < ball.radius) {
      // Projected point on line
      const t = ((px * dx + py * dy) / (dx * dx + dy * dy));
      if (t >= 0 && t <= 1) {
        // Reflect velocity
        const newVel = reflect({ x: ball.vx, y: ball.vy }, normal);
        ball.vx = newVel.x;
        ball.vy = newVel.y;

        // Push out of surface
        ball.x += normal.x * (ball.radius - dist);
        ball.y += normal.y * (ball.radius - dist);
      }
    }
  }
}

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

  // Draw surfaces
  ctx.strokeStyle = "#66ccff";
  ctx.lineWidth = 4;
  for (const s of surfaces) {
    ctx.beginPath();
    ctx.moveTo(s.x1, s.y1);
    ctx.lineTo(s.x2, s.y2);
    ctx.stroke();
  }

  // Draw ball
  ctx.beginPath();
  ctx.arc(ball.x, ball.y, ball.radius, 0, Math.PI * 2);
```

```
  ctx.fillStyle = "#ffaa00";
  ctx.fill();
}

function loop() {
  updateBall();
  draw();
  requestAnimationFrame(loop);
}

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

Handling Angled Surfaces

For surfaces angled arbitrarily, you must calculate the surface normal. If the surface line has a direction vector $\mathbf{s}$, the normal is perpendicular:

$$\mathbf{n} = (-s_y, s_x)$$

Normalize this vector before applying the reflection formula.

Summary

Bouncing off angled surfaces hinges on reflecting the velocity vector about the surface normal. This vector reflection formula ensures realistic bounce directions, applicable in billiard balls, pinball machines, or any collision scenario involving slopes. Understanding and applying this concept allows for natural and visually convincing animations of bouncing objects in dynamic environments.

16.3 Dynamic and Optimized Code Techniques

When implementing rotation and bounce mechanics in animations or games, performance becomes critical, especially if many objects are interacting simultaneously. This section covers coding practices to optimize these calculations, ensuring smooth, efficient animations without sacrificing accuracy.

Avoid Redundant Calculations

One common inefficiency arises from repeating the same mathematical operations multiple times per frame or per object. For example, when rotating points or calculating bounce directions, trigonometric functions like `Math.sin()` and `Math.cos()` can be expensive.

Optimization Tip: Calculate sine and cosine values once per angle and reuse them, rather than recalculating inside loops or repeatedly for the same angle.

```
const angle = rotationAngle; // radians
const cosA = Math.cos(angle);
const sinA = Math.sin(angle);

function rotatePoint(x, y) {
  return {
    x: x * cosA - y * sinA,
    y: x * sinA + y * cosA
  };
}
```

This memoization reduces CPU cycles significantly, especially for animations with many points or frames.

Use Efficient Data Structures

Storing points, velocities, and normals in arrays or objects with predictable, flat structures improves cache locality and speeds up access.

For example, instead of nested objects:

```
const points = [
  { x: 10, y: 20 },
  { x: 15, y: 25 }
];
```

Consider using **typed arrays** or flat arrays for large datasets:

```
const points = new Float32Array([10, 20, 15, 25]); // [x1, y1, x2, y2]
```

This format is especially beneficial when interfacing with WebGL or GPU-accelerated contexts and can improve loop unrolling and vectorization by JavaScript engines.

Precompute Surface Normals and Use Immutable Vectors

If your environment includes many static or slowly changing surfaces, precompute their normal vectors and store them for reuse instead of recalculating every frame.

Additionally, design your vector operations to avoid unnecessary object creation inside loops. Reusing vector instances or applying in-place updates reduces garbage collection overhead.

Simplify Bounce Logic Where Possible

Sometimes, you can approximate bounce behaviors to save computation, especially when the precise physical accuracy is less critical. For instance:

- Use axis-aligned bounding boxes (AABB) for collision detection before performing expensive bounce calculations.
- Limit bounce calculations only to objects currently colliding or near collision boundaries.
- Skip bounce calculations for objects with very low velocity (below a threshold) to prevent jitter.

Batch Processing and Loop Unrolling

Process multiple objects' rotation and bounce calculations inside a single loop rather than dispersing logic across many function calls. This approach reduces function call overhead.

You can also manually unroll small loops to improve performance in critical sections.

Example: Optimized Bounce Reflection

Here's a snippet that caches normals and uses memoized sine/cosine for rotation and bounce calculations:

```
// Precompute normal for a surface at 45 degrees
const angle = Math.PI / 4;
const normal = { x: Math.cos(angle), y: Math.sin(angle) };

// Reflect velocity vector v
function reflect(v, n) {
  const dot = v.x * n.x + v.y * n.y;
  return {
    x: v.x - 2 * dot * n.x,
    y: v.y - 2 * dot * n.y
  };
}

// Rotate vector v by precomputed sin/cos
const cosA = Math.cos(angle);
const sinA = Math.sin(angle);

function rotate(v) {
  return {
    x: v.x * cosA - v.y * sinA,
    y: v.x * sinA + v.y * cosA
  };
}
```

16.3.1 Summary

Optimizing rotation and bounce computations involves memoizing repeated calculations, choosing efficient data structures, and simplifying logic without compromising the animation's visual quality. These techniques ensure your animations run smoothly, even under heavy computational loads or on lower-end devices. Efficient, clean code also improves maintainability and scalability for complex interactive projects.

16.4 Multi-Angle Bounce Handling

In real-world animations and games, objects often collide with surfaces of varying angles — not just flat horizontal or vertical walls. Handling bounces off such angled or moving surfaces

requires dynamic calculations of collision responses to ensure realistic motion.

Generalizing Bounce Directions

At the core of multi-angle bounce handling is the **reflection of the velocity vector** about the surface's **normal vector**. The normal is a perpendicular vector to the surface at the point of contact, and it changes based on the surface's orientation.

The reflection formula for a velocity vector **v** hitting a surface with normal **n** is:

$$v_{reflected} = v - 2(v \cdot n)n$$

where $v \cdot n$ is the dot product.

This formula applies regardless of the surface angle, making it ideal for complex or dynamic collisions.

Handling Complex Shapes

For polygons or curved surfaces, determine the surface normal at the collision point:

- **Polygons**: The normal of the edge the object collides with.
- **Curved surfaces**: Compute the gradient or use the radius vector (for circles) to find the normal.

For example, in a polygon:

```
function getEdgeNormal(p1, p2) {
  const dx = p2.x - p1.x;
  const dy = p2.y - p1.y;
  // Normal vector is perpendicular to the edge vector
  return { x: -dy, y: dx };
}
```

Normalize this vector before using it for reflection.

Example: Dynamic Bounce on Rotating Surface

Imagine a paddle rotating around a pivot, like in a pong game. To handle bounce:

1. Calculate the surface normal by rotating a base normal vector with the paddle's angle.
2. Reflect the ball's velocity using the rotated normal.

```
// Rotate a vector by angle
function rotateVector(v, angle) {
  const cosA = Math.cos(angle);
  const sinA = Math.sin(angle);
  return {
    x: v.x * cosA - v.y * sinA,
    y: v.x * sinA + v.y * cosA
  };
}

// Base normal pointing straight up
const baseNormal = { x: 0, y: -1 };
```

```
// Paddle rotation angle (radians)
const paddleAngle = Math.PI / 6; // 30 degrees

// Rotated normal
const normal = rotateVector(baseNormal, paddleAngle);

// Ball velocity vector
let velocity = { x: 5, y: -3 };

// Dot product
const dot = velocity.x * normal.x + velocity.y * normal.y;

// Reflection calculation
const reflected = {
  x: velocity.x - 2 * dot * normal.x,
  y: velocity.y - 2 * dot * normal.y
};

velocity = reflected;
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Dynamic Bounce on Rotating Paddle</title>
  <style>
    body {
      margin: 0;
      background: #111;
    }
    canvas {
      display: block;
      margin: 0 auto;
      background: #222;
    }
  </style>
</head>
<body>
<canvas id="canvas" width="800" height="600"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

const ball = {
  x: 400,
  y: 100,
  radius: 10,
  vx: 3,
  vy: 3,
};

const paddle = {
  x: 400,
  y: 500,
  length: 200,
```

```
  angle: 0,
  speed: 0.02 // radians per frame
};

// Utility: rotate vector
function rotateVector(v, angle) {
  const cos = Math.cos(angle);
  const sin = Math.sin(angle);
  return {
    x: v.x * cos - v.y * sin,
    y: v.x * sin + v.y * cos
  };
}

// Utility: reflect vector over normal
function reflect(velocity, normal) {
  const dot = velocity.x * normal.x + velocity.y * normal.y;
  return {
    x: velocity.x - 2 * dot * normal.x,
    y: velocity.y - 2 * dot * normal.y
  };
}

function update() {
  // Move paddle
  paddle.angle += paddle.speed;
  if (paddle.angle > Math.PI / 4 || paddle.angle < -Math.PI / 4) {
    paddle.speed *= -1;
  }

  // Move ball
  ball.x += ball.vx;
  ball.y += ball.vy;

  // Collision with paddle
  const dx = ball.x - paddle.x;
  const dy = ball.y - paddle.y;
  const dist = Math.abs(dx * Math.sin(paddle.angle) - dy * Math.cos(paddle.angle));

  if (dist < ball.radius + 5 && dy > -20 && dy < 20) {
    // Reflect using rotated normal
    const baseNormal = { x: 0, y: -1 };
    const normal = rotateVector(baseNormal, paddle.angle);
    const reflected = reflect({ x: ball.vx, y: ball.vy }, normal);
    ball.vx = reflected.x;
    ball.vy = reflected.y;

    // Move ball slightly away to prevent sticking
    ball.x += normal.x * 2;
    ball.y += normal.y * 2;
  }

  // Bounce off walls
  if (ball.x < ball.radius || ball.x > canvas.width - ball.radius) ball.vx *= -1;
  if (ball.y < ball.radius || ball.y > canvas.height - ball.radius) ball.vy *= -1;
}

function draw() {
```

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

  // Draw paddle
  ctx.save();
  ctx.translate(paddle.x, paddle.y);
  ctx.rotate(paddle.angle);
  ctx.fillStyle = "#66ccff";
  ctx.fillRect(-paddle.length / 2, -5, paddle.length, 10);
  ctx.restore();

  // Draw ball
  ctx.beginPath();
  ctx.arc(ball.x, ball.y, ball.radius, 0, Math.PI * 2);
  ctx.fillStyle = "#ffaa00";
  ctx.fill();
}

function loop() {
  update();
  draw();
  requestAnimationFrame(loop);
}

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

Moving Surfaces

If the surface itself moves or rotates, add the surface velocity to the collision response to simulate realistic interaction:

$$v_{new} = v_{reflected} + v_{surface}$$

This accounts for momentum transfer between objects.

16.4.1 Summary

Multi-angle bounce handling hinges on dynamically computing the surface normal at collision points, then reflecting velocity vectors accordingly. This approach generalizes well for complex shapes, moving platforms, and rotating paddles. Combining these calculations with real-time updates for moving surfaces enables realistic, engaging animations and gameplay physics.

By mastering these techniques, you can simulate rich interactive scenes where objects respond naturally to their environments, regardless of surface orientation or motion.

Chapter 17.

Simulating Billiard Ball Physics

1. Mass and Momentum Concepts
2. Conservation of Momentum (1D and 2D)
3. JavaScript Implementation Techniques

17 Simulating Billiard Ball Physics

17.1 Mass and Momentum Concepts

In physics, **mass** and **momentum** are fundamental properties that govern how objects move and interact, especially during collisions. Understanding these concepts is crucial for simulating realistic billiard ball physics in animations and games.

Mass: The Measure of Matter

Mass represents how much matter an object contains. It is a scalar quantity (just a number, no direction) usually measured in kilograms (kg). In billiards, all balls typically have the same mass, but in animations, you might simulate objects with different masses to see varied behaviors.

Mass affects how an object responds to forces. Heavier objects are harder to accelerate or stop, while lighter ones react more easily.

Momentum: Mass in Motion

Momentum is the product of an object's mass and velocity:

$$\vec{p} = m\vec{v}$$

Here, $\vec{p}$ (momentum) and $\vec{v}$ (velocity) are **vectors**, meaning they have both magnitude and direction. Momentum captures how much motion an object has and where it's heading.

- In **one dimension (1D)**, velocity and momentum are along a single line (e.g., left-right).
- In **two dimensions (2D)**, velocity and momentum have both x and y components.

Why Are Mass and Momentum Important?

During collisions, especially elastic ones like billiard balls, momentum determines how objects exchange motion:

- **Mass** tells us how much "inertia" the object has — its resistance to changes in motion.
- **Momentum** tells us the quantity of motion and its direction.

When two billiard balls collide, their masses and momenta govern how their velocities change post-collision, ensuring realistic bounces.

Key Takeaway

Mass and momentum are the backbone of simulating motion and collisions:

- **Mass** determines how much force is needed to change motion.
- **Momentum** conveys how motion is transferred during collisions.

By modeling these quantities in your animation, you create believable, dynamic interactions

that mimic real-world billiard physics. The next section will build on these concepts by exploring conservation of momentum, ensuring energy and motion transfer follow physical laws accurately.

17.2 Conservation of Momentum (1D and 2D)

The **conservation of momentum** is a fundamental principle in physics stating that within a closed system with no external forces, the total momentum before and after a collision remains constant. This law is crucial for simulating realistic billiard ball collisions, where balls bounce off each other without losing energy to friction or deformation — known as **elastic collisions**.

Conservation of Momentum in 1D Elastic Collisions

Consider two billiard balls moving along a single straight line (one dimension). Let:

- m_1, m_2 be the masses of the two balls,
- v_1, v_2 their velocities before collision,
- v_1', v_2' their velocities after collision.

The conservation of momentum equation is:

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2'$$

Because the collision is elastic, **kinetic energy** is also conserved:

$$\frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2 = \frac{1}{2} m_1 {v_1'}^2 + \frac{1}{2} m_2 {v_2'}^2$$

Solving these simultaneously gives the new velocities:

$$v_1' = \frac{(m_1 - m_2) v_1 + 2 m_2 v_2}{m_1 + m_2}$$

$$v_2' = \frac{(m_2 - m_1) v_2 + 2 m_1 v_1}{m_1 + m_2}$$

Example: Two equal mass balls where Ball 1 moves at 5 units/s and Ball 2 is stationary:

$$v_1' = 0, \quad v_2' = 5$$

Ball 1 stops, transferring its velocity to Ball 2 — a classic billiard break shot effect.

Extending to 2D Collisions

In 2D, collisions are more complex because velocity has both **x** and **y** components. Instead of dealing with velocities directly, it's easier to work with **velocity vectors** and decompose them along the collision normal and tangent directions.

Key steps:

1. Compute the **normal vector n** between ball centers at collision.
2. Compute the **tangent vector t**, perpendicular to **n**.
3. Project velocities onto **n** and **t**:

$$v_n = \mathbf{v} \cdot \mathbf{n}, \quad v_t = \mathbf{v} \cdot \mathbf{t}$$

4. After collision, the **tangential components** remain unchanged (no friction along the tangent), but **normal components** exchange according to 1D elastic collision formulas:

$$v'_{1n} = \frac{(m_1 - m_2)v_{1n} + 2m_2 v_{2n}}{m_1 + m_2}$$

$$v'_{2n} = \frac{(m_2 - m_1)v_{2n} + 2m_1 v_{1n}}{m_1 + m_2}$$

5. Recombine vectors:

$$\mathbf{v}'_1 = v'_{1n}\mathbf{n} + v_{1t}\mathbf{t}$$

$$\mathbf{v}'_2 = v'_{2n}\mathbf{n} + v_{2t}\mathbf{t}$$

Practical Example

Suppose two balls collide with velocities:

$$\mathbf{v}_1 = (3, 2), \quad \mathbf{v}_2 = (-1, 1)$$

and their centers form a normal vector **n**. By projecting, swapping normal velocities using 1D formulas, and reconstructing, you simulate realistic bouncing where directions and speeds adjust naturally.

Summary

- Conservation of momentum ensures the total motion remains constant through collisions.
- In **1D**, velocities swap and scale based on mass.
- In **2D**, velocities are decomposed and only normal components exchange.

- This method preserves energy and realistic ball interactions, fundamental for billiard animations.

In the next section, we'll explore how to implement these calculations efficiently in JavaScript for smooth and responsive billiard ball simulations.

17.3 JavaScript Implementation Techniques

Translating billiard ball physics into JavaScript involves implementing velocity, mass, and collision response calculations in code and integrating these with your animation loop. Here, we'll break down the key steps and provide a detailed example simulating 2D elastic collisions between balls.

Representing Balls as Objects

First, define a `Ball` class to encapsulate position, velocity, radius, and mass:

```
class Ball {
  constructor(x, y, vx, vy, radius, mass) {
    this.x = x;       // x position
    this.y = y;       // y position
    this.vx = vx;     // velocity x-component
    this.vy = vy;     // velocity y-component
    this.radius = radius;
    this.mass = mass;
  }

  // Update position based on velocity and time delta
  update(dt) {
    this.x += this.vx * dt;
    this.y += this.vy * dt;
  }
}
```

Detecting Collisions

To detect collisions between two balls, check if the distance between their centers is less than or equal to the sum of their radii:

```
function areColliding(ball1, ball2) {
  const dx = ball2.x - ball1.x;
  const dy = ball2.y - ball1.y;
  const distance = Math.sqrt(dx * dx + dy * dy);
  return distance <= (ball1.radius + ball2.radius);
}
```

Handling 2D Elastic Collision Response

When two balls collide, apply the momentum conservation and vector reflection logic. The algorithm involves:

- Calculating the **normal vector** between the balls,

- Projecting velocities onto the normal and tangent,
- Swapping normal components using 1D elastic collision formulas,
- Reconstructing the new velocity vectors.

Here's a function to resolve the collision:

```
function resolveCollision(ball1, ball2) {
  // Calculate the normal vector
  const dx = ball2.x - ball1.x;
  const dy = ball2.y - ball1.y;
  const distance = Math.sqrt(dx * dx + dy * dy);

  // Normalized normal vector
  const nx = dx / distance;
  const ny = dy / distance;

  // Tangent vector (perpendicular to normal)
  const tx = -ny;
  const ty = nx;

  // Project velocities onto the normal and tangent vectors
  const v1n = ball1.vx * nx + ball1.vy * ny;
  const v1t = ball1.vx * tx + ball1.vy * ty;
  const v2n = ball2.vx * nx + ball2.vy * ny;
  const v2t = ball2.vx * tx + ball2.vy * ty;

  // Compute new normal velocities using 1D elastic collision formula
  const v1nAfter = (v1n * (ball1.mass - ball2.mass) + 2 * ball2.mass * v2n) / (ball1.mass + ball2.mass)
  const v2nAfter = (v2n * (ball2.mass - ball1.mass) + 2 * ball1.mass * v1n) / (ball1.mass + ball2.mass)

  // Convert scalar normal and tangent velocities back into vectors
  ball1.vx = v1nAfter * nx + v1t * tx;
  ball1.vy = v1nAfter * ny + v1t * ty;
  ball2.vx = v2nAfter * nx + v2t * tx;
  ball2.vy = v2nAfter * ny + v2t * ty;

  // Prevent balls from overlapping by repositioning them
  const overlap = 0.5 * (ball1.radius + ball2.radius - distance + 0.01);
  ball1.x -= overlap * nx;
  ball1.y -= overlap * ny;
  ball2.x += overlap * nx;
  ball2.y += overlap * ny;
}
```

Integrating with the Animation Loop

Now, incorporate the balls' motion and collision detection into your animation loop:

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
const balls = [
  new Ball(100, 100, 150, 120, 20, 1),
  new Ball(300, 200, -100, -80, 20, 1)
];

let lastTime = performance.now();

function animate(time) {
```

```
  const dt = (time - lastTime) / 1000;  // Convert ms to seconds
  lastTime = time;

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

  // Update positions
  balls.forEach(ball => ball.update(dt));

  // Detect and resolve collisions
  for (let i = 0; i < balls.length; i++) {
    for (let j = i + 1; j < balls.length; j++) {
      if (areColliding(balls[i], balls[j])) {
        resolveCollision(balls[i], balls[j]);
      }
    }
  }

  // Draw balls
  balls.forEach(ball => {
    ctx.beginPath();
    ctx.arc(ball.x, ball.y, ball.radius, 0, Math.PI * 2);
    ctx.fillStyle = 'dodgerblue';
    ctx.fill();
    ctx.stroke();
  });

  requestAnimationFrame(animate);
}

requestAnimationFrame(animate);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Billiard Ball Physics Simulation</title>
  <style>
    body {
      margin: 0;
      background: #111;
    }
    canvas {
      display: block;
      margin: auto;
      background: #222;
      border: 1px solid #555;
    }
  </style>
</head>
<body>
<canvas id="canvas" width="800" height="600"></canvas>
<script>
// ----- Ball Class -----
class Ball {
```

```
  constructor(x, y, vx, vy, radius, mass) {
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.radius = radius;
    this.mass = mass;
  }

  update(dt) {
    this.x += this.vx * dt;
    this.y += this.vy * dt;

    // Bounce off walls
    if (this.x - this.radius < 0) {
      this.x = this.radius;
      this.vx *= -1;
    } else if (this.x + this.radius > canvas.width) {
      this.x = canvas.width - this.radius;
      this.vx *= -1;
    }

    if (this.y - this.radius < 0) {
      this.y = this.radius;
      this.vy *= -1;
    } else if (this.y + this.radius > canvas.height) {
      this.y = canvas.height - this.radius;
      this.vy *= -1;
    }
  }
}

// ----- Collision Detection -----
function areColliding(ball1, ball2) {
  const dx = ball2.x - ball1.x;
  const dy = ball2.y - ball1.y;
  const distance = Math.sqrt(dx * dx + dy * dy);
  return distance <= (ball1.radius + ball2.radius);
}

// ----- Elastic Collision Resolution -----
function resolveCollision(ball1, ball2) {
  const dx = ball2.x - ball1.x;
  const dy = ball2.y - ball1.y;
  const distance = Math.sqrt(dx * dx + dy * dy);
  if (distance === 0) return; // avoid NaN division

  const nx = dx / distance;
  const ny = dy / distance;

  const tx = -ny;
  const ty = nx;

  const v1n = ball1.vx * nx + ball1.vy * ny;
  const v1t = ball1.vx * tx + ball1.vy * ty;
  const v2n = ball2.vx * nx + ball2.vy * ny;
  const v2t = ball2.vx * tx + ball2.vy * ty;
```

```
  const v1nAfter = (v1n * (ball1.mass - ball2.mass) + 2 * ball2.mass * v2n) / (ball1.mass + ball2.mass)
  const v2nAfter = (v2n * (ball2.mass - ball1.mass) + 2 * ball1.mass * v1n) / (ball1.mass + ball2.mass)

  ball1.vx = v1nAfter * nx + v1t * tx;
  ball1.vy = v1nAfter * ny + v1t * ty;
  ball2.vx = v2nAfter * nx + v2t * tx;
  ball2.vy = v2nAfter * ny + v2t * ty;

  // Resolve overlap
  const overlap = 0.5 * (ball1.radius + ball2.radius - distance + 0.01);
  ball1.x -= overlap * nx;
  ball1.y -= overlap * ny;
  ball2.x += overlap * nx;
  ball2.y += overlap * ny;
}

// ----- Setup -----
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const balls = [
  new Ball(200, 300, 150, 100, 20, 1),
  new Ball(600, 300, -100, -120, 20, 1)
];

let lastTime = performance.now();

// ----- Animation Loop -----
function animate(time) {
  const dt = (time - lastTime) / 1000;
  lastTime = time;

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

  // Update positions
  balls.forEach(ball => ball.update(dt));

  // Detect & resolve collisions
  for (let i = 0; i < balls.length; i++) {
    for (let j = i + 1; j < balls.length; j++) {
      if (areColliding(balls[i], balls[j])) {
        resolveCollision(balls[i], balls[j]);
      }
    }
  }

  // Draw balls
  balls.forEach(ball => {
    ctx.beginPath();
    ctx.arc(ball.x, ball.y, ball.radius, 0, Math.PI * 2);
    ctx.fillStyle = 'deepskyblue';
    ctx.fill();
    ctx.strokeStyle = 'white';
    ctx.stroke();
  });

  requestAnimationFrame(animate);
}
```

```
requestAnimationFrame(animate);
</script>
</body>
</html>
```

Notes and Enhancements

- **Time Delta:** The `dt` variable ensures movement remains consistent regardless of frame rate fluctuations.
- **Overlap Correction:** Adjusting positions after collision avoids balls sticking together.
- **Extensibility:** This setup can be extended for multiple balls, variable masses, or even collision with boundaries.
- **Performance:** For many balls, consider spatial partitioning to optimize collision checks.

By following these steps, you can implement realistic billiard ball motion and collisions in JavaScript, providing a solid foundation for physics-based animation projects. This example combines physics principles with practical coding techniques for smooth, interactive simulations.

Chapter 18.

Gravity and Particle Attraction

1. Particle Systems and Gravitational Forces
2. Collision Reactions and Orbit Simulation
3. Springy Node Gardens and Connected Nodes

18 Gravity and Particle Attraction

18.1 Particle Systems and Gravitational Forces

Particle systems are a fundamental tool in animation and simulation, consisting of numerous small objects—called particles—that move and interact according to defined rules. These systems are used to model phenomena like smoke, fire, flocking birds, and celestial bodies. A key aspect of many particle systems is how particles respond to forces, especially gravitational attraction.

What Is a Particle System?

A particle system is essentially a collection of independent or interacting points, each with properties such as position, velocity, acceleration, and mass. The overall behavior of the system emerges from how these particles move and influence one another over time.

For example, imagine a cloud of tiny particles floating in space. Each particle moves based on its velocity, but also feels forces from other particles, such as gravity pulling them together or pushing them apart.

Gravitational Attraction Between Particles

Gravity is a natural force causing objects with mass to attract each other. Newton's law of universal gravitation defines the force F between two masses m_1 and m_2 separated by distance r as:

$$F = G\frac{m_1 m_2}{r^2}$$

where G is the gravitational constant.

In animation, we often simplify G to a smaller constant to control the effect's strength and scale. The force acts along the line connecting the two particles and can be converted into acceleration applied to each particle (using Newton's second law, $F = ma$).

Applying Gravitational Forces in Animations

To simulate gravitational attraction, calculate the force vector between each pair of particles, convert it to acceleration, and update their velocities accordingly.

Here's the basic approach for a pair of particles:

1. Calculate the difference vector:

```
let dx = p2.x - p1.x;
let dy = p2.y - p1.y;
let distSq = dx*dx + dy*dy;
let dist = Math.sqrt(distSq);
```

2. Calculate force magnitude (using a chosen gravitational constant `G`):

```
let G = 0.1; // tuned for animation scale
let force = G * (p1.mass * p2.mass) / distSq;
```

3. Calculate acceleration components for each particle:

```
let ax = force * dx / dist / p1.mass;
let ay = force * dy / dist / p1.mass;
```

4. Update velocities of `p1` and similarly for `p2` (in opposite direction):

```
p1.vx += ax;
p1.vy += ay;
// For p2, use negative ax, ay
```

Practical Examples: Clustering and Orbiting

- **Particle Clustering:** When multiple particles attract each other, they tend to cluster, forming groups or "clouds." This is useful in simulating star clusters or fluid-like behaviors.
- **Orbit Simulation:** By carefully tuning initial velocities, particles can orbit around a central mass, mimicking planetary systems. The balance of gravitational pull and velocity causes curved trajectories.

Sample Code Snippet

```
class Particle {
  constructor(x, y, mass) {
    this.x = x; this.y = y;
    this.vx = 0; this.vy = 0;
    this.mass = mass;
  }

  applyForce(fx, fy) {
    this.vx += fx / this.mass;
    this.vy += fy / this.mass;
  }

  update() {
    this.x += this.vx;
    this.y += this.vy;
  }
}

const G = 0.1;
const particles = [/* array of Particle objects */];

function applyGravity() {
  for (let i = 0; i < particles.length; i++) {
    for (let j = i + 1; j < particles.length; j++) {
      let p1 = particles[i];
      let p2 = particles[j];
      let dx = p2.x - p1.x;
      let dy = p2.y - p1.y;
      let distSq = dx*dx + dy*dy;
      let dist = Math.sqrt(distSq);
```

```
      if (dist > 1) { // avoid division by zero
        let force = G * (p1.mass * p2.mass) / distSq;
        let fx = force * dx / dist;
        let fy = force * dy / dist;

        p1.applyForce(fx, fy);
        p2.applyForce(-fx, -fy);
      }
    }
  }
}

// In animation loop:
function animate() {
  applyGravity();
  particles.forEach(p => p.update());
  // draw particles here
  requestAnimationFrame(animate);
}
```

By combining particle systems with gravitational forces, you can create rich, dynamic animations of particles clustering, orbiting, and interacting—opening the door to naturalistic and engaging visual effects in your JavaScript animations.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Particle Clustering and Orbiting</title>
  <style>
    body {
      margin: 0;
      background: #000;
      overflow: hidden;
    }
    canvas {
      display: block;
      margin: auto;
      background: #111;
      border: 1px solid #222;
    }
  </style>
</head>
<body>
<canvas id="canvas" width="800" height="600"></canvas>
<script>
class Particle {
  constructor(x, y, mass) {
    this.x = x;
    this.y = y;
    this.vx = 0;
    this.vy = 0;
    this.mass = mass;
    this.radius = Math.cbrt(mass) * 4; // radius for visualization, scaled by mass
```

```
  }

  applyForce(fx, fy) {
    this.vx += fx / this.mass;
    this.vy += fy / this.mass;
  }

  update() {
    this.x += this.vx;
    this.y += this.vy;
  }
}

const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
const G = 0.5; // gravitational constant

// Create particles: one heavy center mass + orbiters + smaller cluster
const particles = [];

// Central massive particle (like a star)
const centerMass = new Particle(canvas.width/2, canvas.height/2, 1000);
particles.push(centerMass);

// Orbiters around center
for(let i = 0; i < 5; i++) {
  let angle = (i / 5) * Math.PI * 2;
  let distance = 120 + i * 30;
  let px = centerMass.x + distance * Math.cos(angle);
  let py = centerMass.y + distance * Math.sin(angle);
  let p = new Particle(px, py, 10);

  // Calculate orbital velocity for circular orbit: v = sqrt(G * M / r)
  let speed = Math.sqrt(G * centerMass.mass / distance);
  p.vx = speed * Math.cos(angle + Math.PI / 2);
  p.vy = speed * Math.sin(angle + Math.PI / 2);

  particles.push(p);
}

// Small cluster of particles nearby to show clustering behavior
for(let i = 0; i < 20; i++) {
  let px = canvas.width/2 + (Math.random() - 0.5) * 300;
  let py = canvas.height/2 + (Math.random() - 0.5) * 300;
  let p = new Particle(px, py, 5);
  particles.push(p);
}

// Gravity force between all pairs
function applyGravity() {
  for(let i = 0; i < particles.length; i++) {
    for(let j = i + 1; j < particles.length; j++) {
      let p1 = particles[i];
      let p2 = particles[j];
      let dx = p2.x - p1.x;
      let dy = p2.y - p1.y;
      let distSq = dx*dx + dy*dy;
      let dist = Math.sqrt(distSq);
```

```
      if(dist > 1) { // avoid division by zero & extreme forces
        let force = G * (p1.mass * p2.mass) / distSq;
        let fx = force * dx / dist;
        let fy = force * dy / dist;

        p1.applyForce(fx, fy);
        p2.applyForce(-fx, -fy);
      }
    }
  }
}

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

  for(let p of particles) {
    ctx.beginPath();
    ctx.arc(p.x, p.y, p.radius, 0, Math.PI * 2);
    if(p === centerMass) {
      ctx.fillStyle = 'yellow';
      ctx.shadowColor = 'yellow';
      ctx.shadowBlur = 20;
    } else {
      ctx.fillStyle = 'deepskyblue';
      ctx.shadowColor = 'deepskyblue';
      ctx.shadowBlur = 10;
    }
    ctx.fill();
    ctx.shadowBlur = 0;
  }
}

function animate() {
  applyGravity();
  particles.forEach(p => p.update());
  draw();
  requestAnimationFrame(animate);
}

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

18.2 Collision Reactions and Orbit Simulation

In particle systems influenced by gravity, collisions play a crucial role in shaping realistic behaviors like orbiting, clustering, or bouncing. When particles attract each other gravitationally, they can also collide and react physically, adding rich dynamics to your animations.

Combining Gravity with Collision Handling

Gravity continuously pulls particles toward each other, while collisions prevent them from overlapping unrealistically. Proper collision detection and response create believable interactions: particles bounce off or merge, or settle into stable orbits.

To handle collisions:

- **Detect overlaps:** Check if the distance between particles is less than the sum of their radii.
- **Resolve penetration:** Separate overlapping particles to avoid "sticking."
- **Calculate new velocities:** Use conservation of momentum and energy principles to simulate elastic or inelastic collisions.

By combining this with gravitational attraction, you can simulate:

- **Orbiting bodies:** A smaller particle orbiting a larger one, with gravity pulling inward and collision forces preventing penetration.
- **Clustering effects:** Particles drawn together but bouncing softly off one another, mimicking gas clouds or swarms.

Example Simulation Overview

Here's a conceptual overview of how to combine gravity and collision reactions in code:

1. **Update positions and velocities due to gravity:** Apply gravitational forces as accelerations between each particle pair.
2. **Check collisions:** For each particle pair, measure distance. If overlapping, calculate collision response.
3. **Apply collision impulses:** Adjust velocities to reflect realistic bouncing or sticking behaviors.
4. **Update particle positions:** Move particles based on their updated velocities.

Sample Collision Response Code Snippet

```
function resolveCollision(p1, p2) {
  const dx = p2.x - p1.x;
  const dy = p2.y - p1.y;
  const dist = Math.sqrt(dx*dx + dy*dy);
  const overlap = p1.radius + p2.radius - dist;

  if (overlap > 0) {
    // Normalize collision vector
    const nx = dx / dist;
    const ny = dy / dist;

    // Separate particles to prevent overlap
    const totalMass = p1.mass + p2.mass;
    p1.x -= nx * overlap * (p2.mass / totalMass);
    p1.y -= ny * overlap * (p2.mass / totalMass);
    p2.x += nx * overlap * (p1.mass / totalMass);
```

```
    p2.y += ny * overlap * (p1.mass / totalMass);

    // Calculate relative velocity along collision normal
    const vxRel = p2.vx - p1.vx;
    const vyRel = p2.vy - p1.vy;
    const velAlongNormal = vxRel * nx + vyRel * ny;

    if (velAlongNormal > 0) return; // Already moving apart

    // Calculate restitution (elasticity)
    const restitution = 0.9; // 1 for perfect elastic

    // Impulse scalar
    const impulse = -(1 + restitution) * velAlongNormal / (1/p1.mass + 1/p2.mass);

    // Apply impulse to velocities
    const impulseX = impulse * nx;
    const impulseY = impulse * ny;

    p1.vx -= impulseX / p1.mass;
    p1.vy -= impulseY / p1.mass;
    p2.vx += impulseX / p2.mass;
    p2.vy += impulseY / p2.mass;
  }
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Collision Response Demo</title>
<style>
  body { margin: 0; background: #222; display: flex; justify-content: center; align-items: center; heigh
  canvas { background: #111; border: 1px solid #444; }
</style>
</head>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

class Particle {
  constructor(x, y, vx, vy, radius, mass, color) {
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.radius = radius;
    this.mass = mass;
    this.color = color;
  }

  update(dt) {
    this.x += this.vx * dt;
```

```
    this.y += this.vy * dt;

    // Bounce off walls
    if (this.x - this.radius < 0) {
      this.x = this.radius;
      this.vx = -this.vx;
    }
    if (this.x + this.radius > canvas.width) {
      this.x = canvas.width - this.radius;
      this.vx = -this.vx;
    }
    if (this.y - this.radius < 0) {
      this.y = this.radius;
      this.vy = -this.vy;
    }
    if (this.y + this.radius > canvas.height) {
      this.y = canvas.height - this.radius;
      this.vy = -this.vy;
    }
  }

  draw(ctx) {
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, Math.PI*2);
    ctx.fillStyle = this.color;
    ctx.fill();
    ctx.strokeStyle = '#ccc';
    ctx.stroke();
  }
}

function resolveCollision(p1, p2) {
  const dx = p2.x - p1.x;
  const dy = p2.y - p1.y;
  const dist = Math.sqrt(dx*dx + dy*dy);
  const overlap = p1.radius + p2.radius - dist;

  if (overlap > 0) {
    // Normalize collision vector
    const nx = dx / dist;
    const ny = dy / dist;

    // Separate particles to prevent overlap
    const totalMass = p1.mass + p2.mass;
    p1.x -= nx * overlap * (p2.mass / totalMass);
    p1.y -= ny * overlap * (p2.mass / totalMass);
    p2.x += nx * overlap * (p1.mass / totalMass);
    p2.y += ny * overlap * (p1.mass / totalMass);

    // Calculate relative velocity along collision normal
    const vxRel = p2.vx - p1.vx;
    const vyRel = p2.vy - p1.vy;
    const velAlongNormal = vxRel * nx + vyRel * ny;

    if (velAlongNormal > 0) return; // Already moving apart

    // Calculate restitution (elasticity)
    const restitution = 0.9; // 1 for perfect elastic
```

```
    // Impulse scalar
    const impulse = -(1 + restitution) * velAlongNormal / (1/p1.mass + 1/p2.mass);

    // Apply impulse to velocities
    const impulseX = impulse * nx;
    const impulseY = impulse * ny;

    p1.vx -= impulseX / p1.mass;
    p1.vy -= impulseY / p1.mass;
    p2.vx += impulseX / p2.mass;
    p2.vy += impulseY / p2.mass;
  }
}

// Create two particles
const p1 = new Particle(150, 200, 100, 50, 30, 3, 'tomato');
const p2 = new Particle(450, 200, -120, -80, 40, 5, 'deepskyblue');

let lastTime = performance.now();

function animate(time) {
  const dt = (time - lastTime) / 1000; // seconds elapsed
  lastTime = time;

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

  p1.update(dt);
  p2.update(dt);

  resolveCollision(p1, p2);

  p1.draw(ctx);
  p2.draw(ctx);

  requestAnimationFrame(animate);
}

requestAnimationFrame(animate);
</script>
</body>
</html>
```

Challenges: Stability and Performance

Stability: Simulating multiple particles with gravity and collisions can lead to instability, such as jittering or particles tunneling through each other. To improve stability:

- Use small time steps for physics updates.
- Limit maximum velocities or apply damping.
- Employ more sophisticated collision resolution techniques like position correction.

Performance: Collision detection is expensive—checking every pair of particles scales quadratically. To optimize:

- Use spatial partitioning structures (e.g., grids, quadtrees) to reduce collision checks.
- Limit collision detection to nearby particles only.

- Balance simulation accuracy with performance needs.

Summary

By combining gravitational forces with collision handling, your particle animations can mimic natural phenomena such as orbiting planets or swirling clusters. While the implementation requires careful tuning for stability and performance, the results are highly rewarding, producing dynamic, engaging visuals that respond realistically to user interactions or simulation parameters.

18.3 Springy Node Gardens and Connected Nodes

Springs offer a powerful way to connect particles and create dynamic, flexible structures often called **node gardens**. These systems simulate networks of nodes connected by spring-like forces, allowing for organic movement, stretching, and oscillations. This approach is widely used in animations such as network visualizations, soft body simulations, and interactive data displays.

Springs Connecting Particles

In a springy node garden, each node (particle) is linked to one or more neighbors by springs that exert forces based on their relative positions. Springs strive to maintain a rest length, pulling nodes closer if stretched, or pushing them apart if compressed. This produces smooth, natural motions that respond to external forces or user interactions.

The spring force $\mathbf{F}$ between two connected nodes is generally modeled by Hooke's Law:

$$\mathbf{F} = -k(d - L)\hat{\mathbf{d}}$$

Where:

- k = spring constant (stiffness)
- d = current distance between nodes
- L = rest length of the spring
- $\hat{\mathbf{d}}$ = normalized vector from one node to the other

Practical Use Cases

- **Network visualizations:** Springs connect nodes representing data points, allowing the layout to dynamically adjust, avoiding overlaps while clustering related nodes.
- **Soft body physics:** Nodes connected by springs form flexible shapes that deform naturally when forces are applied, such as bouncing balls or jiggling blobs.
- **Interactive art:** Springy connections create visually engaging, organic motions reacting to user input.

Example: Simple Spring Connection

Here's a simplified JavaScript example illustrating a spring between two particles:

```
class Particle {
  constructor(x, y, mass = 1) {
    this.x = x;
    this.y = y;
    this.vx = 0;
    this.vy = 0;
    this.mass = mass;
  }

  applyForce(fx, fy) {
    this.vx += fx / this.mass;
    this.vy += fy / this.mass;
  }

  update() {
    this.x += this.vx;
    this.y += this.vy;
    // Apply some damping for stability
    this.vx *= 0.9;
    this.vy *= 0.9;
  }
}

function applySpringForce(p1, p2, restLength, k) {
  let dx = p2.x - p1.x;
  let dy = p2.y - p1.y;
  let dist = Math.sqrt(dx*dx + dy*dy);
  if (dist === 0) return;

  let forceMag = -k * (dist - restLength);
  let fx = (dx / dist) * forceMag;
  let fy = (dy / dist) * forceMag;

  p1.applyForce(fx, fy);
  p2.applyForce(-fx, -fy);
}

// Usage example
const p1 = new Particle(100, 100);
const p2 = new Particle(200, 100);
const restLength = 100;
const stiffness = 0.1;

function animate() {
  applySpringForce(p1, p2, restLength, stiffness);

  p1.update();
  p2.update();

  // Drawing and animation code would go here...

  requestAnimationFrame(animate);
}

animate();
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Simple Spring Connection</title>
<style>
  body {
    margin: 0;
    background: #222;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    user-select: none;
  }
  canvas {
    background: #111;
    border: 1px solid #444;
  }
</style>
</head>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
class Particle {
  constructor(x, y, mass = 1) {
    this.x = x;
    this.y = y;
    this.vx = 0;
    this.vy = 0;
    this.mass = mass;
  }

  applyForce(fx, fy) {
    this.vx += fx / this.mass;
    this.vy += fy / this.mass;
  }

  update() {
    this.x += this.vx;
    this.y += this.vy;
    // Apply some damping for stability
    this.vx *= 0.9;
    this.vy *= 0.9;
  }
}

function applySpringForce(p1, p2, restLength, k) {
  let dx = p2.x - p1.x;
  let dy = p2.y - p1.y;
  let dist = Math.sqrt(dx*dx + dy*dy);
  if (dist === 0) return;

  let forceMag = -k * (dist - restLength);
  let fx = (dx / dist) * forceMag;
  let fy = (dy / dist) * forceMag;
```

```
  p1.applyForce(fx, fy);
  p2.applyForce(-fx, -fy);
}

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

const p1 = new Particle(100, 100);
const p2 = new Particle(200, 100);
const restLength = 100;
const stiffness = 0.1;

// For interaction: drag particles with mouse
let draggingParticle = null;
let offsetX = 0;
let offsetY = 0;

canvas.addEventListener('mousedown', (e) => {
  const rect = canvas.getBoundingClientRect();
  const mx = e.clientX - rect.left;
  const my = e.clientY - rect.top;
  [p1, p2].forEach(p => {
    const dist = Math.hypot(p.x - mx, p.y - my);
    if (dist < 15) {
      draggingParticle = p;
      offsetX = p.x - mx;
      offsetY = p.y - my;
    }
  });
});

canvas.addEventListener('mousemove', (e) => {
  if (draggingParticle) {
    const rect = canvas.getBoundingClientRect();
    draggingParticle.x = e.clientX - rect.left + offsetX;
    draggingParticle.y = e.clientY - rect.top + offsetY;
    // Reset velocity while dragging for stability
    draggingParticle.vx = 0;
    draggingParticle.vy = 0;
  }
});

canvas.addEventListener('mouseup', () => {
  draggingParticle = null;
});

canvas.addEventListener('mouseleave', () => {
  draggingParticle = null;
});

function drawParticle(p) {
  ctx.beginPath();
  ctx.arc(p.x, p.y, 10, 0, Math.PI * 2);
  ctx.fillStyle = '#33ccff';
  ctx.fill();
  ctx.strokeStyle = '#0af';
  ctx.lineWidth = 2;
```

```
  ctx.stroke();
}

function drawSpring(p1, p2) {
  ctx.beginPath();
  ctx.moveTo(p1.x, p1.y);
  ctx.lineTo(p2.x, p2.y);
  ctx.strokeStyle = '#ffaa00';
  ctx.lineWidth = 3;
  ctx.stroke();
}

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

  if (!draggingParticle) {
    applySpringForce(p1, p2, restLength, stiffness);
    p1.update();
    p2.update();
  }

  drawSpring(p1, p2);
  drawParticle(p1);
  drawParticle(p2);

  requestAnimationFrame(animate);
}

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

Expanding to Multiple Nodes

By connecting many particles with springs, you can create complex, springy networks that gently oscillate and respond to forces or interactions. Each spring continuously applies forces to its connected nodes, and the entire system stabilizes over time or reacts dynamically when disturbed.

Summary

Springy node gardens provide a versatile and visually appealing way to simulate connected structures that behave organically. By tuning parameters like stiffness and rest length, you can create a variety of effects — from elastic data layouts to soft, bouncing bodies. This approach opens up creative possibilities in animation and interactive graphics, bridging physics and art seamlessly.

Chapter 19.

Forward Kinematics

1. Basics of Kinematics in Animation
2. Moving Single and Multiple Segments
3. Automating Walk Cycles
4. Handling Gravity and Collision in Walks

19 Forward Kinematics

19.1 Basics of Kinematics in Animation

Forward kinematics is a foundational technique in animation that deals with calculating the positions and orientations of jointed segments in a hierarchy. This approach allows animators and programmers to simulate natural movement for complex structures such as limbs, robotic arms, or any chain of connected parts.

Key Terminology

- **Joint:** A point of rotation or connection between two segments. Joints act like hinges or pivots, allowing segments to rotate relative to each other.
- **Segment:** A rigid part or link connected by joints. For example, an arm's upper arm, forearm, and hand are segments connected by elbow and wrist joints.
- **Degrees of Freedom (DoF):** The number of independent ways a joint can move. For simple forward kinematics, rotation around one axis (e.g., bending a knee) represents one degree of freedom.

How Forward Kinematics Works

In forward kinematics, the position and orientation of each segment are computed by starting from a base or root joint and moving outward through the chain of segments. The key concept is that each segment's transform depends on the position and rotation of its parent segment.

For example, imagine a simple two-segment arm:

1. The **base joint** defines the shoulder's position and rotation.
2. The first segment (upper arm) extends from this shoulder joint.
3. The **elbow joint** at the end of the upper arm rotates, affecting the position and angle of the forearm.
4. The second segment (forearm) extends from the elbow.

By applying rotations to each joint, the final position of the hand (end effector) is determined by combining transformations from the shoulder and elbow.

Mathematical Overview

Each segment's position is calculated by applying rotation and translation transforms relative to its parent. Using 2D as an example, the position (x_n, y_n) of the endpoint of segment n is given by:

$$x_n = x_{n-1} + L_n \cos(\theta_1 + \theta_2 + \cdots + \theta_n)$$

$$y_n = y_{n-1} + L_n \sin(\theta_1 + \theta_2 + \cdots + \theta_n)$$

Where:

- x_{n-1}, y_{n-1} are the coordinates of the previous joint (parent),
- L_n is the length of the current segment,
- $\theta_1, \theta_2, \ldots, \theta_n$ are the rotation angles of each joint up to the current one.

The cumulative sum of joint angles reflects how each joint's rotation adds up along the chain.

Visual Example

Imagine a robotic arm made of three segments, each 100 pixels long:

- The shoulder joint is at $(100, 100)$ with a rotation of 30°.
- The elbow joint rotates 45° relative to the upper arm.
- The wrist joint rotates 20° relative to the forearm.

Calculating the hand's final position involves adding vectors sequentially, accounting for each rotation and segment length.

Why Forward Kinematics Matters

Forward kinematics is intuitive and straightforward to implement because you control the joints directly. It works well for many animation scenarios where joint rotations are known or controlled by user input, such as robotic arms or simple character limbs.

However, because forward kinematics calculates positions based on rotations, achieving precise end-effector placement (e.g., making a hand touch a specific point) can be challenging. This is where inverse kinematics (covered in later chapters) becomes useful.

Summary

- Forward kinematics calculates segment positions from base joints outward.
- Joints provide rotation points; segments connect these joints.
- Degrees of freedom describe how joints can move.
- Each segment's position depends on the combined rotations of all parent joints.
- This method is foundational for animating articulated structures realistically.

This theoretical framework sets the stage for practical applications in animating limbs, mechanical systems, and more, which we will explore in the following sections.

19.2 Moving Single and Multiple Segments

In forward kinematics, animating jointed structures involves controlling the rotation angles of each segment and calculating their positions relative to their parent joints. This section demonstrates how to programmatically move single segments and chains of connected segments, helping you build flexible, articulated animations.

Moving a Single Segment

Let's start with the simplest case: a single segment rotating around a fixed point (the origin or a base joint). Suppose you have a segment of length `L` attached at point `(x0, y0)`, rotating by an angle (in radians).

The endpoint `(x1, y1)` of the segment can be calculated using basic trigonometry:

```
// Base position
const x0 = 200;
const y0 = 200;

// Segment length and rotation angle
const L = 100;
let theta = Math.PI / 4; // 45 degrees

// Calculate endpoint
const x1 = x0 + L * Math.cos(theta);
const y1 = y0 + L * Math.sin(theta);

console.log(`Endpoint: (${x1.toFixed(2)}, ${y1.toFixed(2)})`);
```

This snippet calculates the position of the segment's end based on the rotation angle. If you change `theta` dynamically (e.g., with user input or animation), the segment will rotate around the base point.

Moving Multiple Connected Segments

For a chain of segments, each connected to the previous segment's endpoint, the position of each subsequent joint depends on the cumulative rotation angles of all preceding joints.

Consider two segments connected in a chain:

- Segment 1 length: `L1`
- Segment 2 length: `L2`
- Rotation angles: `1` for segment 1, `2` for segment 2 (relative to segment 1)

To calculate the position `(x2, y2)` of the end of the second segment:

```
// Base position
const x0 = 200;
const y0 = 200;

// Segment lengths
const L1 = 100;
const L2 = 80;

// Rotation angles in radians
let theta1 = Math.PI / 4;  // 45 degrees
let theta2 = Math.PI / 6;  // 30 degrees

// Endpoint of segment 1
const x1 = x0 + L1 * Math.cos(theta1);
const y1 = y0 + L1 * Math.sin(theta1);

// Endpoint of segment 2
const x2 = x1 + L2 * Math.cos(theta1 + theta2);
```

```
const y2 = y1 + L2 * Math.sin(theta1 + theta2);

console.log(`Segment 1 endpoint: (${x1.toFixed(2)}, ${y1.toFixed(2)})`);
console.log(`Segment 2 endpoint: (${x2.toFixed(2)}, ${y2.toFixed(2)})`);
```

Notice how the second segment's rotation is relative to the first, so we add angles when calculating its endpoint.

Runnable Example: Interactive Segment Control

Here's a basic example using HTML5 Canvas and JavaScript where you can control two segments with sliders to see their combined effect:

```
<canvas id="canvas" width="400" height="400"></canvas>
<label>
  Segment 1 Angle:
  <input id="angle1" type="range" min="0" max="360" value="45">
</label>
<label>
  Segment 2 Angle:
  <input id="angle2" type="range" min="0" max="360" value="30">
</label>

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

const x0 = 200;
const y0 = 200;
const L1 = 100;
const L2 = 80;

const angle1Slider = document.getElementById('angle1');
const angle2Slider = document.getElementById('angle2');

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

  const theta1 = (angle1Slider.value * Math.PI) / 180;
  const theta2 = (angle2Slider.value * Math.PI) / 180;

  // Segment 1 endpoint
  const x1 = x0 + L1 * Math.cos(theta1);
  const y1 = y0 + L1 * Math.sin(theta1);

  // Segment 2 endpoint
  const x2 = x1 + L2 * Math.cos(theta1 + theta2);
  const y2 = y1 + L2 * Math.sin(theta1 + theta2);

  // Draw base
  ctx.fillStyle = 'black';
  ctx.beginPath();
  ctx.arc(x0, y0, 5, 0, 2 * Math.PI);
  ctx.fill();

  // Draw segment 1
  ctx.strokeStyle = 'blue';
```

```
  ctx.lineWidth = 5;
  ctx.beginPath();
  ctx.moveTo(x0, y0);
  ctx.lineTo(x1, y1);
  ctx.stroke();

  // Draw joint 1
  ctx.fillStyle = 'red';
  ctx.beginPath();
  ctx.arc(x1, y1, 5, 0, 2 * Math.PI);
  ctx.fill();

  // Draw segment 2
  ctx.strokeStyle = 'green';
  ctx.beginPath();
  ctx.moveTo(x1, y1);
  ctx.lineTo(x2, y2);
  ctx.stroke();

  // Draw end effector
  ctx.fillStyle = 'orange';
  ctx.beginPath();
  ctx.arc(x2, y2, 5, 0, 2 * Math.PI);
  ctx.fill();
}

// Update drawing when sliders change
angle1Slider.addEventListener('input', draw);
angle2Slider.addEventListener('input', draw);

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

This code lets you interactively adjust the rotation angles of two connected segments, showing how joint rotations propagate down the chain.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
<label>
  Segment 1 Angle:
  <input id="angle1" type="range" min="0" max="360" value="45">
</label>
<label>
  Segment 2 Angle:
  <input id="angle2" type="range" min="0" max="360" value="30">
</label>

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

const x0 = 200;
```

```
const y0 = 200;
const L1 = 100;
const L2 = 80;

const angle1Slider = document.getElementById('angle1');
const angle2Slider = document.getElementById('angle2');

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

  const theta1 = (angle1Slider.value * Math.PI) / 180;
  const theta2 = (angle2Slider.value * Math.PI) / 180;

  // Segment 1 endpoint
  const x1 = x0 + L1 * Math.cos(theta1);
  const y1 = y0 + L1 * Math.sin(theta1);

  // Segment 2 endpoint
  const x2 = x1 + L2 * Math.cos(theta1 + theta2);
  const y2 = y1 + L2 * Math.sin(theta1 + theta2);

  // Draw base
  ctx.fillStyle = 'black';
  ctx.beginPath();
  ctx.arc(x0, y0, 5, 0, 2 * Math.PI);
  ctx.fill();

  // Draw segment 1
  ctx.strokeStyle = 'blue';
  ctx.lineWidth = 5;
  ctx.beginPath();
  ctx.moveTo(x0, y0);
  ctx.lineTo(x1, y1);
  ctx.stroke();

  // Draw joint 1
  ctx.fillStyle = 'red';
  ctx.beginPath();
  ctx.arc(x1, y1, 5, 0, 2 * Math.PI);
  ctx.fill();

  // Draw segment 2
  ctx.strokeStyle = 'green';
  ctx.beginPath();
  ctx.moveTo(x1, y1);
  ctx.lineTo(x2, y2);
  ctx.stroke();

  // Draw end effector
  ctx.fillStyle = 'orange';
  ctx.beginPath();
  ctx.arc(x2, y2, 5, 0, 2 * Math.PI);
  ctx.fill();
}

// Update drawing when sliders change
angle1Slider.addEventListener('input', draw);
angle2Slider.addEventListener('input', draw);
```

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

</body>
</html>
```

Extending to Multiple Segments

To animate or control longer chains, you can loop through segments, summing rotations and calculating endpoints iteratively:

```
let segments = [
  { length: 100, angle: Math.PI / 4 },
  { length: 80, angle: Math.PI / 6 },
  { length: 60, angle: Math.PI / 8 },
];

let baseX = 200, baseY = 200;
let currentX = baseX;
let currentY = baseY;
let totalAngle = 0;

for (let i = 0; i < segments.length; i++) {
  totalAngle += segments[i].angle;
  const nextX = currentX + segments[i].length * Math.cos(totalAngle);
  const nextY = currentY + segments[i].length * Math.sin(totalAngle);

  // Draw or process segment from (currentX, currentY) to (nextX, nextY)

  currentX = nextX;
  currentY = nextY;
}
```

Summary

- Single segments are moved by applying rotation and calculating the endpoint using trigonometry.
- Multiple connected segments require summing rotation angles to find positions downstream.
- Interactive examples help visualize how changes in joint angles propagate.
- Iterative approaches easily scale for chains with many segments.

This technique is crucial for animating limbs, tentacles, robotic arms, and more, offering a flexible way to create complex articulated motion programmatically.

19.3 Automating Walk Cycles

Creating natural-looking walk cycles is a classic challenge in animation, and forward kinematics (FK) provides a powerful framework to simulate leg movement by controlling joint rotations

over time. In this section, we'll explore how to automate walk cycles by sequencing segment rotations, generating smooth, believable leg motion.

Understanding Walk Cycles with Forward Kinematics

A walk cycle typically involves animating limbs—like legs and arms—through a repetitive sequence of joint angles that simulate stepping motions. Each leg consists of multiple segments connected by joints (e.g., thigh, shin, foot), and FK lets us calculate each joint's position based on the rotation angles of preceding joints.

The key to automating walk cycles is to:

- Define target rotation angles for each joint over time.
- Interpolate angles smoothly to simulate motion.
- Synchronize limbs so steps alternate naturally.

Sequencing Joint Rotations Over Time

To animate joints, we often use **periodic functions**, such as sine or cosine waves, to represent smooth, continuous angle changes. For example, a simple oscillation for the thigh joint angle could be:

```
let time = performance.now() / 1000; // seconds
const thighAngle = Math.sin(time * walkSpeed) * maxThighRotation;
```

- `walkSpeed` controls how fast the leg swings.
- `maxThighRotation` is the maximum angle of swing.
- `time` ensures continuous cycling.

For the shin and foot, the rotation angles usually lag or lead the thigh by a phase difference to mimic natural bending and foot positioning.

Coordinating Multiple Limbs

A full walk cycle coordinates two legs in opposing phases:

- When the left leg swings forward, the right leg swings backward.
- Arms typically swing opposite to the legs for balance.

You can implement this by offsetting the phase of sine waves controlling each limb. For example:

```
const leftThighAngle = Math.sin(time * walkSpeed) * maxThighRotation;
const rightThighAngle = Math.sin(time * walkSpeed + Math.PI) * maxThighRotation; // Opposite phase
```

This results in a natural alternating step motion.

Example: Animating a Two-Legged Walk Cycle

Below is a simplified example controlling a leg with thigh and shin segments using FK and sine wave-driven angles:

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

```
const baseX = 200;
const baseY = 300;

const thighLength = 80;
const shinLength = 70;

const maxThighRotation = Math.PI / 6;  // 30 degrees
const maxShinRotation = Math.PI / 4;   // 45 degrees

const walkSpeed = 2; // cycles per second

function drawLeg(time, baseX, baseY, phase = 0) {
  // Calculate angles using sine waves with phase offset
  const thighAngle = Math.sin(time * walkSpeed * 2 * Math.PI + phase) * maxThighRotation;
  // Shin angle peaks when thigh is mid-swing (offset by PI/2)
  const shinAngle = Math.sin(time * walkSpeed * 2 * Math.PI + phase + Math.PI / 2) * maxShinRotation;

  // Calculate thigh endpoint
  const thighX = baseX + thighLength * Math.cos(thighAngle - Math.PI / 2);
  const thighY = baseY + thighLength * Math.sin(thighAngle - Math.PI / 2);

  // Calculate shin endpoint
  const shinX = thighX + shinLength * Math.cos(thighAngle + shinAngle - Math.PI / 2);
  const shinY = thighY + shinLength * Math.sin(thighAngle + shinAngle - Math.PI / 2);

  // Draw thigh
  ctx.lineWidth = 8;
  ctx.strokeStyle = 'blue';
  ctx.beginPath();
  ctx.moveTo(baseX, baseY);
  ctx.lineTo(thighX, thighY);
  ctx.stroke();

  // Draw shin
  ctx.strokeStyle = 'green';
  ctx.beginPath();
  ctx.moveTo(thighX, thighY);
  ctx.lineTo(shinX, shinY);
  ctx.stroke();

  // Draw joints
  ctx.fillStyle = 'red';
  ctx.beginPath();
  ctx.arc(baseX, baseY, 6, 0, 2 * Math.PI);
  ctx.fill();

  ctx.beginPath();
  ctx.arc(thighX, thighY, 6, 0, 2 * Math.PI);
  ctx.fill();

  ctx.beginPath();
  ctx.arc(shinX, shinY, 6, 0, 2 * Math.PI);
  ctx.fill();
}

function animate(time) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
```

```
  const t = time / 1000; // convert to seconds

  // Draw left leg (phase 0)
  drawLeg(t, 150, 300, 0);

  // Draw right leg (phase PI for opposite movement)
  drawLeg(t, 250, 300, Math.PI);

  requestAnimationFrame(animate);
}

requestAnimationFrame(animate);
```

This example uses sine waves to continuously update the thigh and shin angles, producing a smooth walking motion with legs alternating naturally.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const baseX = 200;
const baseY = 300;

const thighLength = 80;
const shinLength = 70;

const maxThighRotation = Math.PI / 6;  // 30 degrees
const maxShinRotation = Math.PI / 4;   // 45 degrees

const walkSpeed = 2; // cycles per second

function drawLeg(time, baseX, baseY, phase = 0) {
  // Calculate angles using sine waves with phase offset
  const thighAngle = Math.sin(time * walkSpeed * 2 * Math.PI + phase) * maxThighRotation;
  // Shin angle peaks when thigh is mid-swing (offset by PI/2)
  const shinAngle = Math.sin(time * walkSpeed * 2 * Math.PI + phase + Math.PI / 2) * maxShinRotation;

  // Calculate thigh endpoint
  const thighX = baseX + thighLength * Math.cos(thighAngle - Math.PI / 2);
  const thighY = baseY + thighLength * Math.sin(thighAngle - Math.PI / 2);

  // Calculate shin endpoint
  const shinX = thighX + shinLength * Math.cos(thighAngle + shinAngle - Math.PI / 2);
  const shinY = thighY + shinLength * Math.sin(thighAngle + shinAngle - Math.PI / 2);

  // Draw thigh
  ctx.lineWidth = 8;
  ctx.strokeStyle = 'blue';
  ctx.beginPath();
  ctx.moveTo(baseX, baseY);
  ctx.lineTo(thighX, thighY);
```

```
  ctx.stroke();

  // Draw shin
  ctx.strokeStyle = 'green';
  ctx.beginPath();
  ctx.moveTo(thighX, thighY);
  ctx.lineTo(shinX, shinY);
  ctx.stroke();

  // Draw joints
  ctx.fillStyle = 'red';
  ctx.beginPath();
  ctx.arc(baseX, baseY, 6, 0, 2 * Math.PI);
  ctx.fill();

  ctx.beginPath();
  ctx.arc(thighX, thighY, 6, 0, 2 * Math.PI);
  ctx.fill();

  ctx.beginPath();
  ctx.arc(shinX, shinY, 6, 0, 2 * Math.PI);
  ctx.fill();
}

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

  const t = time / 1000; // convert to seconds

  // Draw left leg (phase 0)
  drawLeg(t, 150, 300, 0);

  // Draw right leg (phase PI for opposite movement)
  drawLeg(t, 250, 300, Math.PI);

  requestAnimationFrame(animate);
}

requestAnimationFrame(animate);
</script>
</body>
</html>
```

Enhancing Realism

- **Adding feet and arms:** Extend the chain of segments using the same principles, adding rotation phases to simulate foot placement and arm swings.
- **Adjusting speed and amplitude:** Control how fast the character walks and how exaggerated the movement is.
- **Incorporating easing:** Smooth transitions between movements using easing functions can improve natural feel.
- **Sync with ground contact:** Adding logic to detect when feet touch the ground helps avoid sliding.

Summary

- Walk cycles can be automated by sequencing joint rotations using periodic functions like sine waves.
- Coordinating phase offsets for limbs creates natural alternating steps.
- Forward kinematics calculates segment positions based on these angles for smooth motion.
- Practical implementations can control multiple segments and limbs, resulting in believable animations.

By mastering these techniques, you can bring characters and creatures to life with convincing, fluid walking animations using straightforward math and programming principles.

19.4 Handling Gravity and Collision in Walks

To create truly believable walk cycles, it's essential to incorporate environmental factors like gravity and collision detection. Forward kinematics (FK) alone moves segments based on joint angles, but without physics integration, characters can appear to float or clip through the ground. This section explores how to blend gravity and collision handling with FK to produce natural, grounded animations.

Why Incorporate Gravity and Collision?

In real life, gravity constantly pulls objects downward, forcing feet to land on solid surfaces and preventing unnatural floating or sinking. Collision detection identifies when body parts interact with the environment, enabling the animation to respond appropriately — for example, by stopping downward movement when a foot hits the ground.

When combined with FK, these forces ensure limbs and feet react to the terrain realistically, enhancing immersion and believability.

Gravity in Walk Cycles

Gravity can be modeled as a constant acceleration pulling downward (along the y-axis in most 2D systems). For a foot segment or the whole character, gravity influences vertical velocity, which updates the position frame-by-frame.

Basic gravity application:

```
const gravity = 9.8; // pixels per second squared (adjust scale for your animation)
let velocityY = 0;
let posY = initialY;

function updatePosition(deltaTime) {
  velocityY += gravity * deltaTime;  // accelerate downwards
  posY += velocityY * deltaTime;     // update position based on velocity
}
```

Here, `deltaTime` is the elapsed time between animation frames, which keeps movement

smooth and framerate independent.

Collision Detection with the Ground

To prevent the foot from falling below the ground, we check for collisions by comparing the foot's vertical position with the ground level.

```
const groundLevel = 300; // y-coordinate of ground

function checkCollision() {
  if (posY > groundLevel) {
    posY = groundLevel;   // reset to ground height
    velocityY = 0;        // stop downward velocity (foot "lands")
  }
}
```

This simple check halts downward motion when the foot reaches the ground, simulating a step landing.

Adjusting Segment Motion on Ground Contact

In FK, joint angles determine segment positions, so to keep the foot on the ground, you must adjust the leg's joint rotations accordingly. When a foot collides with the ground:

- The ankle or foot joint angle should lock or adjust to maintain contact.
- The thigh and shin rotations can shift slightly to prevent unnatural penetration.
- Vertical movement stops, but horizontal motion (forward stepping) continues.

One approach is to detect foot position, then tweak thigh and shin angles using inverse kinematics or constrained FK to ensure the foot stays at ground level.

Integrating Gravity and FK: Example

Here's a simplified example combining gravity, ground collision, and FK for a leg:

```
const gravity = 9.8;
let velocityY = 0;
let footY = 100;          // initial vertical position of foot
const groundY = 300;
const deltaTime = 0.016;  // approx 60fps

// Joint angles (in radians)
let thighAngle = 0;
let shinAngle = 0;

function update() {
  // Apply gravity to foot vertical velocity
  velocityY += gravity * deltaTime;
  footY += velocityY * deltaTime;

  // Ground collision
  if (footY > groundY) {
    footY = groundY;
    velocityY = 0;

    // Adjust angles to maintain foot on ground
    // Example: slightly bend shin and thigh
```

```
    thighAngle = Math.min(thighAngle + 0.01, Math.PI / 6);
    shinAngle = Math.min(shinAngle + 0.02, Math.PI / 4);
  } else {
    // Foot is airborne, reset angles for swing
    thighAngle = Math.sin(performance.now() / 500) * Math.PI / 8;
    shinAngle = Math.sin(performance.now() / 300) * Math.PI / 6;
  }

  drawLeg(thighAngle, shinAngle, footY);
}

function drawLeg(thighAngle, shinAngle, footY) {
  // Calculate positions using FK based on angles and footY
  // ... draw thigh, shin, foot ...
}

// Animation loop
function animate() {
  update();
  requestAnimationFrame(animate);
}

animate();
```

This code demonstrates:

- Gravity pulling the foot down.
- Collision stopping the foot at `groundY`.
- Adjusting joint angles on ground contact to simulate stance.
- Swinging angles when foot is in the air.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Gravity + FK Leg Example</title>
<style>
  body {
    margin: 0;
    background: #222;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    user-select: none;
  }
  canvas {
    background: #111;
    border: 1px solid #444;
  }
</style>
</head>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
```

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

const gravity = 980; // pixels per second squared (scaled up for visibility)
let velocityY = 0;
let footY = 100;          // initial vertical position of foot
const groundY = 300;
const deltaTime = 0.016;  // ~60fps

// Leg segment lengths
const thighLength = 100;
const shinLength = 80;

// Joint angles (radians)
let thighAngle = 0;
let shinAngle = 0;

function update() {
  // Apply gravity to foot vertical velocity
  velocityY += gravity * deltaTime;
  footY += velocityY * deltaTime;

  // Ground collision
  if (footY > groundY) {
    footY = groundY;
    velocityY = 0;

    // Slightly bend thigh and shin when on ground
    thighAngle = Math.min(thighAngle + 0.01, Math.PI / 6);
    shinAngle = Math.min(shinAngle + 0.02, Math.PI / 4);
  } else {
    // Foot is airborne, swing angles with sine functions
    const now = performance.now();
    thighAngle = Math.sin(now / 500) * Math.PI / 8;
    shinAngle = Math.sin(now / 300) * Math.PI / 6;
  }
}

function drawLeg(thighAngle, shinAngle, footY) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Base hip position (fixed)
  const hipX = canvas.width / 2;
  const hipY = groundY - thighLength - shinLength;

  ctx.strokeStyle = '#ffaa00';
  ctx.lineWidth = 8;
  ctx.lineCap = 'round';

  // Calculate knee position (thigh end)
  const kneeX = hipX + thighLength * Math.sin(thighAngle);
  const kneeY = hipY + thighLength * Math.cos(thighAngle);

  // Calculate foot position (shin end)
  // Shin angle relative to thigh angle (forward kinematics)
  const footX = kneeX + shinLength * Math.sin(thighAngle + shinAngle);
  // footY is controlled by gravity simulation, override shin's y if on ground
```

```
  const effectiveFootY = footY < groundY ?
    kneeY + shinLength * Math.cos(thighAngle + shinAngle) :
    footY;

  // Draw thigh
  ctx.beginPath();
  ctx.moveTo(hipX, hipY);
  ctx.lineTo(kneeX, kneeY);
  ctx.stroke();

  // Draw shin
  ctx.beginPath();
  ctx.moveTo(kneeX, kneeY);
  ctx.lineTo(footX, effectiveFootY);
  ctx.stroke();

  // Draw joints
  ctx.fillStyle = '#33ccff';
  ctx.beginPath();
  ctx.arc(hipX, hipY, 10, 0, Math.PI * 2);
  ctx.fill();

  ctx.beginPath();
  ctx.arc(kneeX, kneeY, 10, 0, Math.PI * 2);
  ctx.fill();

  ctx.beginPath();
  ctx.arc(footX, effectiveFootY, 12, 0, Math.PI * 2);
  ctx.fill();

  // Draw ground line
  ctx.strokeStyle = '#00ff00';
  ctx.lineWidth = 2;
  ctx.beginPath();
  ctx.moveTo(0, groundY);
  ctx.lineTo(canvas.width, groundY);
  ctx.stroke();
}

function animate() {
  update();
  drawLeg(thighAngle, shinAngle, footY);
  requestAnimationFrame(animate);
}

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

Challenges and Tips

- **Stable foot placement:** Small errors can cause foot sliding or jittering. Using inverse kinematics or constraints helps keep the foot firmly on the ground.
- **Smooth transitions:** Gradually interpolate angles when switching between swing and stance phases to avoid sudden jerks.

- **Multiple limbs:** Apply the same gravity and collision logic to all legs or arms to maintain balance.
- **Performance:** Optimize by limiting collision checks to relevant segments and using efficient math.

Summary

Incorporating gravity and collision detection into walk cycles makes animations grounded and realistic. By applying downward acceleration, detecting ground contact, and adjusting joint angles accordingly, you create natural foot placement and leg movement. Combining simple physics with FK transforms basic joint rotations into life-like locomotion, enhancing the overall animation quality and viewer engagement.

Chapter 20.

Inverse Kinematics

1. Reaching and Dragging with Segments
2. Multi-Segment Dragging and Reaching
3. Implementing the Law of Cosines

20 Inverse Kinematics

20.1 Reaching and Dragging with Segments

Inverse Kinematics (IK) is a powerful technique used to control articulated structures—like robotic arms, character limbs, or tentacles—by specifying the desired position of an end segment rather than manually rotating each joint. This section introduces the fundamental concepts of IK, especially in the context of reaching and dragging segment endpoints.

What Is Inverse Kinematics?

Unlike **Forward Kinematics (FK)**, where you start with joint angles to calculate the position of each segment, **Inverse Kinematics** works backward: you specify the desired position of the endpoint (often called the "effector"), and the system calculates the required joint angles to reach that position.

For example, imagine a simple two-segment arm: in FK, you set the angles of the shoulder and elbow joints and then determine where the hand ends up. In IK, you tell the hand where you want it to go, and the system figures out how to rotate the shoulder and elbow to get there.

Why Use IK?

IK is crucial for animations and robotics where you want intuitive control over the end effectors (hands, feet, tools) without manually adjusting every joint. It's widely used for:

- **Character animation:** Making a character's hand reach a door handle or a foot plant on uneven ground.
- **Robotics:** Calculating the joint movements needed for a robotic arm to grasp an object.
- **Interactive applications:** Letting users drag and position limbs or tentacles naturally.

Basic IK Concept: Reaching a Target

Consider a single segment with a fixed base point and a flexible end that you want to move to a target point. If the target is within the segment's reach (length), IK will calculate the joint angle needed so the segment's endpoint matches the target exactly.

For two connected segments (like an upper arm and forearm), IK involves solving for two angles so the combined segments reach the target. This is often done using geometric methods, like the **Law of Cosines**, to determine joint angles from segment lengths and target positions.

Dragging Endpoints with IK

In interactive applications, users often drag the end of a limb or chain. The IK solver continuously recalculates joint angles as the endpoint moves, updating the segment rotations in real time. This creates smooth, natural movement where the rest of the limb follows the dragged point logically, respecting physical constraints like segment lengths.

Visual Example

Imagine a two-segment arm:

- Segment 1 (upper arm) rotates at the shoulder.
- Segment 2 (forearm) rotates at the elbow.
- The goal is to place the hand (end of segment 2) at the cursor location.

As you drag the cursor around, the IK algorithm adjusts shoulder and elbow angles so the hand "follows" perfectly, bending naturally within the arm's length limits.

Summary

Inverse Kinematics flips the FK problem on its head by calculating joint rotations from desired end positions instead of vice versa. This method is fundamental for animating complex, articulated systems with intuitive controls and realistic motion. Mastering IK opens doors to sophisticated animation, robotics, and interactive design, making objects respond naturally to user input or environment constraints.

In the next sections, we will explore multi-segment IK and the math behind calculating joint angles, setting the foundation for building your own IK solvers in JavaScript.

20.2 Multi-Segment Dragging and Reaching

20.2.1 Multi-Segment Dragging and Reaching

Building on the basics of inverse kinematics (IK) with single or two-segment chains, this section explores how IK principles extend to **multiple connected segments**—often called **kinematic chains**. Multi-segment IK enables realistic animation of limbs, tails, tentacles, or robotic arms with many joints, where each joint's rotation contributes to the final position of the end effector.

The Challenge of Multi-Segment IK

When dealing with multiple segments, calculating the exact joint angles to reach a target point becomes mathematically complex. Unlike simple two-segment arms, analytical solutions are not always feasible or practical for longer chains. Instead, **iterative numerical methods** are commonly used.

Key challenges include:

- **Joint constraints:** Physical limits on joint rotation angles to prevent unnatural bending.
- **Reachability:** The target may be outside the total reach of the segments.
- **Smoothness:** Avoiding jitter or snapping by smoothing angle adjustments over time.

Iterative IK Methods

Two popular iterative approaches for multi-segment IK are:

1. **Cyclic Coordinate Descent (CCD):** CCD adjusts each joint starting from the one closest to the end effector, rotating it to reduce the distance between the chain's endpoint and the target. This process cycles through all joints multiple times until the end effector is close enough to the target or a max iteration count is reached.
2. **Jacobian Inverse or Pseudoinverse Methods:** These use differential calculus and matrix operations to approximate the best small changes to all joints simultaneously to minimize the distance to the target.

For simplicity and efficiency, CCD is widely used in animation and game development.

How CCD Works Step by Step

1. Start from the end effector (last joint).
2. For each joint going backward:
    - Calculate the vector from the joint to the end effector.
    - Calculate the vector from the joint to the target.
    - Compute the angle between these vectors.
    - Rotate the joint by this angle toward the target.
3. Repeat until the end effector reaches the target within an acceptable threshold.

Example: JavaScript CCD IK Solver for Multi-Segment Chain

Below is a runnable example of a simple CCD IK solver for a chain of segments:

```
class Segment {
  constructor(length, angle = 0) {
    this.length = length;
    this.angle = angle; // current rotation in radians
    this.x = 0; // end position X (calculated)
    this.y = 0; // end position Y (calculated)
  }
}

function updateChainPositions(segments, baseX, baseY) {
  let x = baseX;
  let y = baseY;
  let totalAngle = 0;

  for (let seg of segments) {
    totalAngle += seg.angle;
    seg.x = x + seg.length * Math.cos(totalAngle);
    seg.y = y + seg.length * Math.sin(totalAngle);
    x = seg.x;
    y = seg.y;
  }
}

function distance(x1, y1, x2, y2) {
```

```
  return Math.hypot(x2 - x1, y2 - y1);
}

function ccdIK(segments, baseX, baseY, targetX, targetY, threshold = 1, maxIter = 10) {
  for (let iter = 0; iter < maxIter; iter++) {
    // Update current positions
    updateChainPositions(segments, baseX, baseY);

    let end = segments[segments.length - 1];
    let distToTarget = distance(end.x, end.y, targetX, targetY);
    if (distToTarget < threshold) break; // close enough

    // Iterate backward over joints
    for (let i = segments.length - 1; i >= 0; i--) {
      let jointX = i === 0 ? baseX : segments[i - 1].x;
      let jointY = i === 0 ? baseY : segments[i - 1].y;

      let toEnd = [end.x - jointX, end.y - jointY];
      let toTarget = [targetX - jointX, targetY - jointY];

      // Calculate angles
      let angleToEnd = Math.atan2(toEnd[1], toEnd[0]);
      let angleToTarget = Math.atan2(toTarget[1], toTarget[0]);
      let deltaAngle = angleToTarget - angleToEnd;

      // Adjust joint angle
      segments[i].angle += deltaAngle;

      // Update chain positions after adjustment
      updateChainPositions(segments, baseX, baseY);

      // Update end position after rotation
      end = segments[segments.length - 1];

      distToTarget = distance(end.x, end.y, targetX, targetY);
      if (distToTarget < threshold) break;
    }
  }
}

// Usage example:
const segments = [
  new Segment(100),
  new Segment(80),
  new Segment(60),
];

const baseX = 300;
const baseY = 300;

let targetX = 400;
let targetY = 200;

// Animate or update:
function animate() {
  ccdIK(segments, baseX, baseY, targetX, targetY);

  // Draw the chain for visualization (using canvas or other)
```

```
  // ...
}

// To interactively drag, update targetX, targetY and call animate()
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>CCD IK Solver Example</title>
<style>
  body {
    margin: 0;
    background: #222;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    user-select: none;
  }
  canvas {
    background: #111;
    border: 1px solid #444;
    cursor: crosshair;
  }
</style>
</head>
<body>
<canvas id="canvas" width="600" height="600"></canvas>
<script>
class Segment {
  constructor(length, angle = 0) {
    this.length = length;
    this.angle = angle; // radians
    this.x = 0; // end point x
    this.y = 0; // end point y
  }
}

function updateChainPositions(segments, baseX, baseY) {
  let x = baseX;
  let y = baseY;
  let totalAngle = 0;
  for (const seg of segments) {
    totalAngle += seg.angle;
    seg.x = x + seg.length * Math.cos(totalAngle);
    seg.y = y + seg.length * Math.sin(totalAngle);
    x = seg.x;
    y = seg.y;
  }
}

function distance(x1, y1, x2, y2) {
  return Math.hypot(x2 - x1, y2 - y1);
}
```

```
function ccdIK(segments, baseX, baseY, targetX, targetY, threshold = 1, maxIter = 15) {
  for (let iter = 0; iter < maxIter; iter++) {
    updateChainPositions(segments, baseX, baseY);
    let end = segments[segments.length - 1];
    let distToTarget = distance(end.x, end.y, targetX, targetY);
    if (distToTarget < threshold) break;

    for (let i = segments.length - 1; i >= 0; i--) {
      let jointX = i === 0 ? baseX : segments[i - 1].x;
      let jointY = i === 0 ? baseY : segments[i - 1].y;

      let toEnd = [end.x - jointX, end.y - jointY];
      let toTarget = [targetX - jointX, targetY - jointY];

      let angleToEnd = Math.atan2(toEnd[1], toEnd[0]);
      let angleToTarget = Math.atan2(toTarget[1], toTarget[0]);
      let deltaAngle = angleToTarget - angleToEnd;

      segments[i].angle += deltaAngle;

      updateChainPositions(segments, baseX, baseY);
      end = segments[segments.length - 1];

      distToTarget = distance(end.x, end.y, targetX, targetY);
      if (distToTarget < threshold) break;
    }
  }
}

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

const segments = [
  new Segment(100),
  new Segment(80),
  new Segment(60),
];

const baseX = 300;
const baseY = 300;

let targetX = baseX + 100;
let targetY = baseY - 100;

canvas.addEventListener('mousemove', (e) => {
  const rect = canvas.getBoundingClientRect();
  targetX = e.clientX - rect.left;
  targetY = e.clientY - rect.top;
});

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

  // Draw base joint
  ctx.fillStyle = '#ffaa00';
  ctx.beginPath();
  ctx.arc(baseX, baseY, 10, 0, Math.PI * 2);
```

```
  ctx.fill();

  // Draw segments and joints
  ctx.strokeStyle = '#00aaff';
  ctx.lineWidth = 8;
  ctx.lineCap = 'round';

  let prevX = baseX;
  let prevY = baseY;
  for (const seg of segments) {
    ctx.beginPath();
    ctx.moveTo(prevX, prevY);
    ctx.lineTo(seg.x, seg.y);
    ctx.stroke();

    ctx.fillStyle = '#33ccff';
    ctx.beginPath();
    ctx.arc(seg.x, seg.y, 8, 0, Math.PI * 2);
    ctx.fill();

    prevX = seg.x;
    prevY = seg.y;
  }

  // Draw target
  ctx.fillStyle = '#ff4444';
  ctx.beginPath();
  ctx.arc(targetX, targetY, 12, 0, Math.PI * 2);
  ctx.fill();
}

function animate() {
  ccdIK(segments, baseX, baseY, targetX, targetY);
  draw();
  requestAnimationFrame(animate);
}

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

Handling Joint Constraints

To make the motion natural, you can clamp each joint's angle to allowed ranges:

```
const minAngle = -Math.PI / 2;
const maxAngle = Math.PI / 2;

segments[i].angle = Math.min(maxAngle, Math.max(minAngle, segments[i].angle));
```

This prevents unnatural bending or twisting.

Summary

Multi-segment IK solvers like CCD allow complex chains of segments to reach targets naturally by iteratively adjusting joint angles. While computationally more involved than simple FK,

these methods enable intuitive, realistic animations for characters and robotic arms. Handling constraints and optimizing iteration count are key to smooth performance and believable motion.

Next, we'll dive into the mathematical foundation with the Law of Cosines, essential for precise joint angle calculations in IK systems.

20.3 Implementing the Law of Cosines

In inverse kinematics (IK), one of the fundamental tasks is determining the angles of joints in a segment chain so that the end effector reaches a desired target position. For two connected segments forming a triangle with the target point, the **Law of Cosines** is an essential mathematical tool that helps solve this triangle and find those joint angles.

Understanding the Triangle in IK

Consider a simple two-segment arm with lengths L_1 and L_2, connected at a joint, with the base at the origin and the end effector at the tip of the second segment. The target point T lies somewhere in the plane. This forms a triangle:

- Side $a = L_1$ (length of the first segment),
- Side $b = L_2$ (length of the second segment),
- Side c is the distance from the base to the target T.

Our goal is to find the angles θ_1 (at the base joint) and θ_2 (between segments) that position the arm's tip exactly at T.

The Law of Cosines

The Law of Cosines states for any triangle with sides a, b, and c, and angle γ opposite side c:

$$c^2 = a^2 + b^2 - 2ab\cos(\gamma)$$

We can rearrange to solve for $\cos(\gamma)$:

$$\cos(\gamma) = \frac{a^2 + b^2 - c^2}{2ab}$$

Applying Law of Cosines in IK

1. **Calculate c, the distance to the target:**

$$c = \sqrt{(T_x - x_0)^2 + (T_y - y_0)^2}$$

where (x_0, y_0) is the base position, often $(0, 0)$.

2. **Check reachability:**

If $c > L_1 + L_2$, the target is unreachable; the arm must stretch fully toward the target.

3. **Calculate the elbow angle θ_2:**

Using sides $a = L_1$, $b = L_2$, and c as above:

$$\cos(\theta_2) = \frac{L_1^2 + L_2^2 - c^2}{2L_1L_2}$$

Then,

$$\theta_2 = \arccos(\cos(\theta_2))$$

This is the angle **between the two segments**.

4. **Calculate the base joint angle θ_1:**

We find the angle α between the line from the base to the target and the horizontal axis:

$$\alpha = \arctan 2(T_y - y_0, T_x - x_0)$$

Then, calculate angle β opposite side $b = L_2$ using:

$$\cos(\beta) = \frac{L_1^2 + c^2 - L_2^2}{2L_1c}$$

So,

$$\beta = \arccos(\cos(\beta))$$

Finally, the base joint angle is:

$$\theta_1 = \alpha - \beta$$

Putting It All Together in Code

```
function lawOfCosinesIK(baseX, baseY, targetX, targetY, L1, L2) {
  // Calculate distance to target
  const dx = targetX - baseX;
  const dy = targetY - baseY;
  const c = Math.hypot(dx, dy);

  // Handle unreachable targets by clamping c
  const reach = L1 + L2;
  const dist = Math.min(c, reach);
```

```
  // Calculate angle theta2 (elbow angle)
  const cosTheta2 = (L1*L1 + L2*L2 - dist*dist) / (2 * L1 * L2);
  const theta2 = Math.acos(Math.min(Math.max(cosTheta2, -1), 1)); // clamp for numerical errors

  // Calculate angle alpha (angle from base to target)
  const alpha = Math.atan2(dy, dx);

  // Calculate angle beta using Law of Cosines
  const cosBeta = (L1*L1 + dist*dist - L2*L2) / (2 * L1 * dist);
  const beta = Math.acos(Math.min(Math.max(cosBeta, -1), 1)); // clamp

  // Calculate base joint angle theta1
  const theta1 = alpha - beta;

  return { theta1, theta2 };
}
```

Visualizing the Result

With θ_1 and θ_2, you can compute the positions of each joint:

```
const jointX = baseX + L1 * Math.cos(theta1);
const jointY = baseY + L1 * Math.sin(theta1);

const endX = jointX + L2 * Math.cos(theta1 + theta2);
const endY = jointY + L2 * Math.sin(theta1 + theta2);
```

Plotting these points will show the arm reaching toward the target.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Law of Cosines IK Solver</title>
<style>
  body {
    margin: 0;
    background: #222;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    user-select: none;
  }
  canvas {
    background: #111;
    border: 1px solid #444;
    cursor: crosshair;
  }
</style>
</head>
<body>
<canvas id="canvas" width="600" height="600"></canvas>
<script>
const canvas = document.getElementById('canvas');
```

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

const baseX = 300;
const baseY = 300;
const L1 = 150;
const L2 = 120;

let targetX = baseX + L1 + L2 - 10; // initial target
let targetY = baseY;

function lawOfCosinesIK(baseX, baseY, targetX, targetY, L1, L2) {
  const dx = targetX - baseX;
  const dy = targetY - baseY;
  const c = Math.hypot(dx, dy);

  const reach = L1 + L2;
  const dist = Math.min(c, reach);

  const cosTheta2 = (L1*L1 + L2*L2 - dist*dist) / (2 * L1 * L2);
  const theta2 = Math.acos(Math.min(Math.max(cosTheta2, -1), 1));

  const alpha = Math.atan2(dy, dx);

  const cosBeta = (L1*L1 + dist*dist - L2*L2) / (2 * L1 * dist);
  const beta = Math.acos(Math.min(Math.max(cosBeta, -1), 1));

  const theta1 = alpha - beta;

  return { theta1, theta2 };
}

canvas.addEventListener('mousemove', (e) => {
  const rect = canvas.getBoundingClientRect();
  targetX = e.clientX - rect.left;
  targetY = e.clientY - rect.top;
});

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

  // Compute joint angles
  const { theta1, theta2 } = lawOfCosinesIK(baseX, baseY, targetX, targetY, L1, L2);

  // Calculate joint positions
  const jointX = baseX + L1 * Math.cos(theta1);
  const jointY = baseY + L1 * Math.sin(theta1);

  const endX = jointX + L2 * Math.cos(theta1 + theta2);
  const endY = jointY + L2 * Math.sin(theta1 + theta2);

  // Draw base joint
  ctx.fillStyle = '#ffaa00';
  ctx.beginPath();
  ctx.arc(baseX, baseY, 12, 0, Math.PI * 2);
  ctx.fill();

  // Draw first segment
  ctx.strokeStyle = '#00aaff';
```

```
  ctx.lineWidth = 10;
  ctx.lineCap = 'round';
  ctx.beginPath();
  ctx.moveTo(baseX, baseY);
  ctx.lineTo(jointX, jointY);
  ctx.stroke();

  // Draw second segment
  ctx.beginPath();
  ctx.moveTo(jointX, jointY);
  ctx.lineTo(endX, endY);
  ctx.stroke();

  // Draw joints
  ctx.fillStyle = '#33ccff';
  ctx.beginPath();
  ctx.arc(jointX, jointY, 10, 0, Math.PI * 2);
  ctx.fill();

  ctx.fillStyle = '#ff4444';
  ctx.beginPath();
  ctx.arc(endX, endY, 10, 0, Math.PI * 2);
  ctx.fill();

  // Draw target
  ctx.strokeStyle = '#ff4444';
  ctx.lineWidth = 2;
  ctx.beginPath();
  ctx.arc(targetX, targetY, 14, 0, Math.PI * 2);
  ctx.stroke();
}

function animate() {
  draw();
  requestAnimationFrame(animate);
}

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

Summary

The Law of Cosines transforms the problem of finding joint angles into solving a triangle with known side lengths. This method is:

- **Robust** for two-segment arms,
- **Efficient** to implement,
- A **foundation** for more complex IK systems.

Understanding this law enables precise control of segment rotations, forming the core of many realistic animations and robotic simulations.

Next, we will review key IK formulas summarizing these computations for quick reference.

Chapter 21.

Introduction to 3D Animation

1. Understanding 3D Coordinates and Perspective
2. 3D Motion: Velocity, Bouncing, and Gravity
3. Z-Sorting and Depth Simulation
4. Easing, Springing, and Collision in 3D

21 Introduction to 3D Animation

21.1 Understanding 3D Coordinates and Perspective

In 2D animation, we manipulate objects along the X (horizontal) and Y (vertical) axes. When moving into 3D animation, an additional axis—Z—represents depth, enabling the creation of immersive, lifelike scenes. Understanding how 3D coordinates work and how they are projected onto a 2D screen is fundamental for crafting compelling animations.

The 3D Coordinate System

The 3D coordinate system extends the familiar 2D plane by adding a third axis:

- **X-axis**: Left to right (horizontal),
- **Y-axis**: Up and down (vertical),
- **Z-axis**: Forward and backward (depth).

This forms a three-dimensional space where every point is represented as (x, y, z). Imagine a room where:

- Moving left or right changes the **x** coordinate,
- Moving up or down changes the **y** coordinate,
- Moving closer or farther from you changes the **z** coordinate.

The **origin** $(0, 0, 0)$ is the center of this space, where all three axes intersect.

Visualizing 3D Points on a 2D Screen: Perspective Projection

Since our screens are flat surfaces, we need to convert these 3D points into 2D coordinates. This process is called **projection**.

The most common projection for 3D graphics is **perspective projection**, which mimics human vision. Objects farther away appear smaller, creating a realistic depth effect.

How Perspective Projection Works

Imagine a camera placed somewhere in 3D space, looking toward a scene. The camera has a **viewpoint** and a **viewing direction**. To project a 3D point onto the camera's 2D image plane (the screen), we use the following logic:

- The camera looks along the **Z-axis**,
- Each 3D point (x, y, z) is "projected" onto a 2D plane positioned at a certain distance (called the **focal length** or **projection plane distance**) from the camera,
- The farther the z value (depth), the smaller the projected x and y become.

Mathematically, the 2D coordinates (x', y') on the screen are calculated by scaling the x and y values based on their depth z:

$$x' = \frac{x \times d}{z + d}$$

$$y' = \frac{y \times d}{z + d}$$

Here, d is the distance from the camera to the projection plane (often called the focal length).

- When z is small (object close), the fraction is close to 1, so $x' \approx x$, $y' \approx y$,
- When z increases (object farther away), the fraction becomes smaller, shrinking the projected coordinates and making the object appear smaller.

This is why distant objects look smaller, giving a sense of depth.

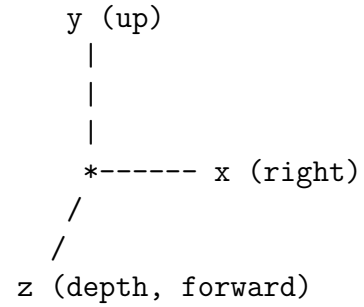
Camera Positioning and View Frustum

The **camera** in 3D animation is like a physical camera or the human eye. It defines:

- **Position**: Where the camera is located in the 3D space,
- **Orientation**: Which direction it is facing,
- **Field of View (FOV)**: The angle that determines how wide the camera can see,
- **Near and Far Clipping Planes**: Limits defining how close or far objects can be to be visible.

These parameters define the **view frustum**, a truncated pyramid-shaped volume representing the visible space. Only objects inside this frustum are rendered.

Diagram: Spatial Relationships




```
Camera at origin looking down positive z-axis.

Projection plane is perpendicular to z-axis at distance d.
3D point (x,y,z) projects to 2D point (x', y') on the projection plane.
```

Summary

- 3D coordinates (x, y, z) add depth to animation, enabling realistic spatial placement.
- Perspective projection scales points by depth to simulate how distant objects appear smaller.
- Camera settings determine the viewpoint, direction, and visible scene portion (view frustum).
- Understanding these basics is crucial to position, animate, and render objects in a believable 3D space.

In upcoming sections, we will explore how to animate 3D objects with velocity, simulate bouncing in 3D, and implement depth sorting for correct rendering order.

21.2 3D Motion: Velocity, Bouncing, and Gravity

Expanding motion from 2D into 3D introduces exciting complexity and realism to your animations. In three-dimensional space, objects move not only along the X and Y axes but also along the Z-axis (depth), allowing for rich, spatially dynamic scenes. This section explores how to apply velocity, simulate forces like gravity, and calculate bounces off surfaces in 3D using vectors and physics concepts.

Velocity in 3D Space

Velocity in 3D is a vector that describes both the speed and direction of an object along all three axes. Instead of a 2D vector $\vec{v} = (v_x, v_y)$, velocity is now:

$$\vec{v} = (v_x, v_y, v_z)$$

Where each component corresponds to the velocity along the X, Y, and Z axes, respectively.

To update an object's position $\vec{p} = (x, y, z)$ over time t, you simply add the velocity scaled by the time delta Δt:

$$\vec{p}_{new} = \vec{p}_{old} + \vec{v} \times \Delta t$$

This formula applies regardless of dimension but now accounts for motion in depth as well.

Applying Gravity in 3D

Gravity is a constant force pulling objects downward, typically along the negative Y-axis (assuming Y is vertical). Gravity can be represented as an acceleration vector:

$$\vec{g} = (0, -g, 0)$$

Where $g \approx 9.8\, m/s^2$ or a suitable scaled value for your animation.

To simulate gravity affecting an object's velocity, update the velocity by adding the acceleration times the time step:

$$\vec{v}_{new} = \vec{v}_{old} + \vec{g} \times \Delta t$$

Then update the position as before.

This simple model allows objects to accelerate downward realistically, simulating falling motion.

Bouncing Off Surfaces in 3D

When an object collides with a surface, it typically bounces off, reversing or redirecting part of its velocity. In 3D, surfaces are represented by **normal vectors** $\vec{n}$, which are perpendicular to the surface.

To compute the velocity after a bounce (perfectly elastic collision), reflect the velocity vector $\vec{v}$ about the surface normal $\vec{n}$:

$$\vec{v}_{reflected} = \vec{v} - 2(\vec{v} \cdot \vec{n})\vec{n}$$

Where:

- $\vec{v} \cdot \vec{n}$ is the dot product of velocity and the normal vector,
- Multiplying the normal by this scalar projects $\vec{v}$ onto $\vec{n}$,
- Subtracting twice this projection from $\vec{v}$ produces the reflected vector.

This reflection formula causes the velocity vector to "bounce" symmetrically off the surface, changing direction according to the surface's angle.

Simple 3D Particle Motion Example

Consider a particle moving inside a cubic space with boundaries at $x, y, z = \pm 100$. The particle has:

- Initial position $\vec{p} = (0, 0, 0)$,
- Initial velocity $\vec{v} = (2, 3, 1)$,
- Gravity acceleration $\vec{g} = (0, -0.1, 0)$.

Each animation frame updates velocity and position:

```
const position = {x: 0, y: 0, z: 0};
const velocity = {x: 2, y: 3, z: 1};
const gravity = {x: 0, y: -0.1, z: 0};
const bounds = 100;
const elasticity = 0.8; // energy loss on bounce

function update() {
  // Apply gravity
  velocity.y += gravity.y;

  // Update position
  position.x += velocity.x;
  position.y += velocity.y;
  position.z += velocity.z;

  // Check and handle bouncing on each boundary
  ['x', 'y', 'z'].forEach(axis => {
    if (position[axis] > bounds) {
      position[axis] = bounds;
      velocity[axis] = -velocity[axis] * elasticity;
```

```
    }
    if (position[axis] < -bounds) {
      position[axis] = -bounds;
      velocity[axis] = -velocity[axis] * elasticity;
    }
  });
}
```

This code models gravity pulling the particle down (negative Y), and bouncing off the cubic container's walls with some energy loss (elasticity).

Visualizing Bounce with Normals

For arbitrary angled surfaces, you calculate the normal vector $\vec{n}$ of the surface at the collision point, then reflect velocity accordingly. For example, if a ball hits a sloped plane, the bounce direction changes naturally.

Summary

- **3D velocity vectors** extend movement along three axes, updated every frame.
- **Gravity** adds downward acceleration, affecting vertical velocity.
- **Bouncing** is handled via vector reflection against surface normals, generalizing collision response to any surface angle.
- Together, these principles allow simulating rich, realistic 3D particle motions, from bouncing balls to orbiting objects.

This foundation enables you to create dynamic 3D scenes with believable physical behaviors, enhancing immersion and interaction in your animations.

21.3 Z-Sorting and Depth Simulation

In 3D animation, simulating depth correctly is crucial for creating believable scenes. One of the fundamental challenges is rendering objects in the right order so that closer objects properly obscure those farther away. This section explains **z-sorting**, a common technique for depth simulation, explores common issues like the painter's algorithm, and discusses ways to handle them.

What is Z-Sorting?

Z-sorting is the process of ordering objects based on their distance (depth) from the camera before rendering. Since the canvas and many rendering contexts draw shapes sequentially, drawing distant objects first and nearer objects later ensures that closer elements visually appear on top, correctly simulating occlusion.

Typically, each object in the scene has a **z-value** representing its depth relative to the camera:

- Smaller or more negative z-values often mean objects are **closer** to the viewer.

- Larger or more positive z-values mean objects are **farther** away.

Sorting objects by their z-value from farthest to nearest and drawing them in that order is called the **Painter's Algorithm** because it mimics how painters layer background elements first, then add foreground details.

Implementing Basic Z-Sorting

A simple approach is:

1. Calculate each object's distance from the camera or its z-coordinate.
2. Sort the array of objects by their z-values in ascending order (farthest first).
3. Render objects in this sorted order.

Here is a conceptual example in JavaScript:

```
// Assume objects is an array with {x, y, z, draw()} for each object
objects.sort((a, b) => a.z - b.z); // Sort by depth: farthest to nearest

objects.forEach(obj => obj.draw());
```

By drawing the farthest objects first, nearer objects naturally overlay them, simulating depth.

Common Pitfalls: The Painters Algorithm Limitations

While z-sorting works well for many cases, it faces challenges:

- **Intersecting or overlapping objects:** When objects intersect or partially overlap in complex ways, a single linear sort by depth may fail. For example, two polygons might overlap differently depending on the viewing angle, causing visual artifacts.
- **Cycles and Sorting Ambiguities:** If Object A appears in front of B in some areas but behind it in others (cyclic overlap), the painter's algorithm cannot order them correctly because no consistent depth order exists.
- **Transparency Issues:** Transparent objects require special handling. Simply drawing transparent distant objects before nearer ones can produce incorrect blending results.

Solutions and Alternatives

To overcome these issues, several techniques can be applied:

1. **Z-buffering (Depth Buffer):** This technique stores depth information for every pixel. When rendering each pixel, the algorithm compares its depth with the stored depth to decide if the pixel should be drawn. This allows correct overlap handling regardless of object complexity. However, z-buffering requires low-level graphics APIs or WebGL, not available in basic 2D canvas rendering.
2. **Splitting Polygons:** Breaking intersecting polygons into smaller parts so they can be correctly sorted. This is complex and rarely done manually in simple 3D engines.
3. **Binary Space Partitioning (BSP) Trees:** A data structure that organizes objects spatially to determine draw order dynamically. Useful in complex scenes with static

geometry.

4. **Painter's Algorithm with Manual Overrides:** In simple cases, sorting objects by depth and manually adjusting draw order for problematic objects can suffice.

Practical Tip: Sorting by Average or Closest Vertex

When objects have volume, deciding which z-value to use for sorting can vary:

- **Average z-value:** Average depth of all vertices.
- **Closest vertex z-value:** Minimum z among vertices.
- **Centroid z-value:** Depth of the object's center.

Using average or centroid z-values is most common, providing a good balance between accuracy and simplicity.

Summary

- **Z-sorting** orders objects by their depth to simulate correct occlusion in 3D scenes.
- The **Painter's Algorithm** draws from farthest to nearest but struggles with intersecting or cyclic overlaps.
- More robust solutions include **z-buffering**, **BSP trees**, or polygon splitting, typically used in advanced graphics engines.
- For canvas animations, sorting objects by their centroid or average z-value is a practical and effective method for most cases.

Understanding these techniques lets you create convincing 3D animations with proper depth cues, enhancing realism and immersion in your projects.

21.4 Easing, Springing, and Collision in 3D

In 3D animation, the principles of easing, springing, and collision handling play a vital role in producing smooth, natural motion and realistic interactions between objects. While these concepts originate from 2D animation, extending them into 3D space requires accounting for the extra dimension, vector operations, and more complex spatial relationships. This section explores how easing and springing are adapted for 3D, how collisions are detected and resolved in three dimensions, and includes practical implementation guidance.

Easing in 3D Animations

Easing functions control the rate of change of motion to avoid abrupt starts and stops, creating smooth transitions. In 3D, easing is applied to each spatial coordinate (x, y, z) or, more efficiently, to the position and orientation vectors of objects.

For example, if an object moves from position $\mathbf{P}$ to a target $\mathbf{P}$, easing interpolates between these points over time using functions like linear, ease-in, ease-out, or custom cubic-bezier curves.

Vector-based easing example:

```
// currentPos and targetPos are {x, y, z} objects
const easingFactor = 0.1;

function easePosition(currentPos, targetPos) {
  return {
    x: currentPos.x + (targetPos.x - currentPos.x) * easingFactor,
    y: currentPos.y + (targetPos.y - currentPos.y) * easingFactor,
    z: currentPos.z + (targetPos.z - currentPos.z) * easingFactor,
  };
}
```

Applying easing uniformly to all three coordinates results in smooth motion through 3D space.

Springing in 3D: Natural Oscillations and Settling

Springing simulates forces that pull an object toward a target position with an oscillatory, "bouncy" motion. In 3D, spring physics involves vectors for position, velocity, and acceleration, with forces applied in all three axes.

The spring force is proportional to the displacement from the target, following Hooke's law:

$$\mathbf{F} = -k(\mathbf{x} - \mathbf{x}_{target}) - c\mathbf{v}$$

- k: spring stiffness constant
- c: damping coefficient (friction)
- $\mathbf{x}$: current position vector
- $\mathbf{v}$: current velocity vector

This produces smooth, natural settling behavior.

Basic 3D spring code snippet:

```
function springUpdate(pos, vel, target, k, c, dt) {
  // Calculate spring force vector
  const force = {
    x: -k * (pos.x - target.x) - c * vel.x,
    y: -k * (pos.y - target.y) - c * vel.y,
    z: -k * (pos.z - target.z) - c * vel.z,
  };

  // Update velocity and position
  vel.x += force.x * dt;
  vel.y += force.y * dt;
  vel.z += force.z * dt;

  pos.x += vel.x * dt;
  pos.y += vel.y * dt;
  pos.z += vel.z * dt;
}
```

This approach allows objects to "spring" toward targets while realistically overshooting and settling.

Handling Collisions in 3D Space

Collision detection and response in 3D require determining when objects intersect and calculating how their velocities should change post-collision.

- **Collision detection** often uses bounding volumes like spheres or axis-aligned bounding boxes (AABB) in 3D. For spheres, the test checks if the distance between centers is less than the sum of radii.
- **Collision response** computes new velocities using vector reflections. The reflection formula for velocity $\mathbf{v}$ off a surface with normal $\mathbf{n}$ is:

$$\mathbf{v}' = \mathbf{v} - 2(\mathbf{v} \cdot \mathbf{n})\mathbf{n}$$

This flips the velocity component along the surface normal, creating a bounce.

Example: 3D Particle Bouncing

Here's a simplified example demonstrating 3D particle bouncing inside a box:

```
// Particle properties
let pos = { x: 50, y: 50, z: 50 };
let vel = { x: 2, y: -3, z: 1 };
const radius = 5;
const bounds = { xMin: 0, xMax: 100, yMin: 0, yMax: 100, zMin: 0, zMax: 100 };
const restitution = 0.8; // energy loss factor on bounce

function update(dt) {
  // Update position
  pos.x += vel.x * dt;
  pos.y += vel.y * dt;
  pos.z += vel.z * dt;

  // Bounce on x-axis walls
  if (pos.x - radius < bounds.xMin) {
    pos.x = bounds.xMin + radius;
    vel.x = -vel.x * restitution;
  } else if (pos.x + radius > bounds.xMax) {
    pos.x = bounds.xMax - radius;
    vel.x = -vel.x * restitution;
  }

  // Bounce on y-axis walls
  if (pos.y - radius < bounds.yMin) {
    pos.y = bounds.yMin + radius;
    vel.y = -vel.y * restitution;
  } else if (pos.y + radius > bounds.yMax) {
    pos.y = bounds.yMax - radius;
    vel.y = -vel.y * restitution;
  }

  // Bounce on z-axis walls
  if (pos.z - radius < bounds.zMin) {
    pos.z = bounds.zMin + radius;
    vel.z = -vel.z * restitution;
  } else if (pos.z + radius > bounds.zMax) {
```

```
    pos.z = bounds.zMax - radius;
    vel.z = -vel.z * restitution;
  }
}
```

This logic can be combined with easing or springing to create objects that smoothly bounce and settle inside 3D spaces.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>3D Particle Bouncing Visualization</title>
<style>
  body {
    margin: 0;
    background: #111;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    user-select: none;
    color: white;
    font-family: sans-serif;
    flex-direction: column;
  }
  canvas {
    background: #222;
    border: 1px solid #444;
    cursor: default;
  }
  #info {
    margin-top: 10px;
    font-size: 14px;
  }
</style>
</head>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
<div id="info">3D Particle Bouncing inside a Box (projected to 2D)</div>

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

// Particle properties
let pos = { x: 50, y: 50, z: 50 };
let vel = { x: 60, y: -90, z: 30 }; // velocity in units per second
const radius = 10;
const bounds = { xMin: 0, xMax: 100, yMin: 0, yMax: 100, zMin: 0, zMax: 100 };
const restitution = 0.8; // energy loss factor on bounce

// Simple perspective projection parameters
const cameraZ = 300;  // distance of camera from origin along z axis
const centerX = canvas.width / 2;
```

```
const centerY = canvas.height / 2;

let lastTime = performance.now();

function project3D(x, y, z) {
  // Simple perspective projection onto 2D plane
  const scale = cameraZ / (cameraZ + z);
  return {
    x: centerX + x * scale,
    y: centerY - y * scale, // invert y for canvas coords
    scale: scale
  };
}

function update(dt) {
  // Update position with velocity and delta time
  pos.x += vel.x * dt;
  pos.y += vel.y * dt;
  pos.z += vel.z * dt;

  // Bounce on x-axis walls
  if (pos.x - radius < bounds.xMin) {
    pos.x = bounds.xMin + radius;
    vel.x = -vel.x * restitution;
  } else if (pos.x + radius > bounds.xMax) {
    pos.x = bounds.xMax - radius;
    vel.x = -vel.x * restitution;
  }

  // Bounce on y-axis walls
  if (pos.y - radius < bounds.yMin) {
    pos.y = bounds.yMin + radius;
    vel.y = -vel.y * restitution;
  } else if (pos.y + radius > bounds.yMax) {
    pos.y = bounds.yMax - radius;
    vel.y = -vel.y * restitution;
  }

  // Bounce on z-axis walls
  if (pos.z - radius < bounds.zMin) {
    pos.z = bounds.zMin + radius;
    vel.z = -vel.z * restitution;
  } else if (pos.z + radius > bounds.zMax) {
    pos.z = bounds.zMax - radius;
    vel.z = -vel.z * restitution;
  }
}

function drawBox() {
  // Draw wireframe cube edges projected in 2D

  const corners = [
    {x: bounds.xMin, y: bounds.yMin, z: bounds.zMin},
    {x: bounds.xMax, y: bounds.yMin, z: bounds.zMin},
    {x: bounds.xMax, y: bounds.yMax, z: bounds.zMin},
    {x: bounds.xMin, y: bounds.yMax, z: bounds.zMin},

    {x: bounds.xMin, y: bounds.yMin, z: bounds.zMax},
```

```
    {x: bounds.xMax, y: bounds.yMin, z: bounds.zMax},
    {x: bounds.xMax, y: bounds.yMax, z: bounds.zMax},
    {x: bounds.xMin, y: bounds.yMax, z: bounds.zMax},
  ];

  const projected = corners.map(c => project3D(c.x, c.y, c.z));

  ctx.strokeStyle = '#888';
  ctx.lineWidth = 1;
  ctx.beginPath();

  // Bottom face
  ctx.moveTo(projected[0].x, projected[0].y);
  ctx.lineTo(projected[1].x, projected[1].y);
  ctx.lineTo(projected[2].x, projected[2].y);
  ctx.lineTo(projected[3].x, projected[3].y);
  ctx.lineTo(projected[0].x, projected[0].y);

  // Top face
  ctx.moveTo(projected[4].x, projected[4].y);
  ctx.lineTo(projected[5].x, projected[5].y);
  ctx.lineTo(projected[6].x, projected[6].y);
  ctx.lineTo(projected[7].x, projected[7].y);
  ctx.lineTo(projected[4].x, projected[4].y);

  // Vertical edges
  for (let i = 0; i < 4; i++) {
    ctx.moveTo(projected[i].x, projected[i].y);
    ctx.lineTo(projected[i + 4].x, projected[i + 4].y);
  }

  ctx.stroke();
}

function drawParticle() {
  const p = project3D(pos.x, pos.y, pos.z);
  const scaledRadius = radius * p.scale;

  // Draw shadow (simple)
  ctx.fillStyle = 'rgba(0,0,0,0.3)';
  ctx.beginPath();
  // Project shadow onto bottom face z = bounds.zMin
  const shadowScale = cameraZ / (cameraZ + bounds.zMin);
  const shadowX = centerX + pos.x * shadowScale;
  const shadowY = centerY - pos.y * shadowScale;
  ctx.ellipse(shadowX, shadowY, scaledRadius * 1.2, scaledRadius * 0.5, 0, 0, Math.PI * 2);
  ctx.fill();

  // Draw particle (circle)
  const gradient = ctx.createRadialGradient(p.x - scaledRadius/3, p.y - scaledRadius/3, scaledRadius/4,
  gradient.addColorStop(0, '#66ccff');
  gradient.addColorStop(1, '#004466');
  ctx.fillStyle = gradient;

  ctx.beginPath();
  ctx.arc(p.x, p.y, scaledRadius, 0, Math.PI * 2);
  ctx.fill();
  ctx.strokeStyle = '#0088cc';
```

```
  ctx.lineWidth = 1;
  ctx.stroke();
}

function animate(time = 0) {
  const dt = (time - lastTime) / 1000; // seconds
  lastTime = time;

  update(dt);

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

  drawBox();
  drawParticle();

  requestAnimationFrame(animate);
}

requestAnimationFrame(animate);
</script>
</body>
</html>
```

Integrating All Together

In complex 3D animations, easing and springing often control position or rotation transitions, while collision detection ensures realistic physical interactions. For instance, a bouncing ball might use springing to soften its bounce, and easing to settle gradually after impacts.

To integrate:

1. Calculate motion updates with easing or springing formulas.
2. Detect collisions and compute new velocities using reflection vectors.
3. Adjust positions and velocities accordingly before the next frame.

Summary

- Easing and springing extend naturally into 3D by operating on vector components, enabling smooth and dynamic motion.
- Collision detection in 3D often uses bounding volumes and vector math to detect and resolve impacts.
- Reflection of velocity vectors off surfaces produces realistic bounce effects.
- Combining these techniques creates engaging and believable 3D animations.

Mastering easing, springing, and collision in 3D is key to bringing your interactive animations and simulations to life with realism and polish.

Chapter 22.

3D Shapes and Rendering

1. Creating Lines, Points, and 3D Shapes
2. Modeling Solids and Moving 3D Objects
3. Using Triangles for 3D Fills

22 3D Shapes and Rendering

22.1 Creating Lines, Points, and 3D Shapes

When working with 3D animation on the web, creating and rendering primitive elements like points, lines, and polygons is the foundation for building complex shapes and models. This section introduces the basics of drawing these primitives in 3D space, primarily focusing on how to plot coordinates, connect points to form shapes, and render them interactively using either the HTML5 canvas or WebGL contexts.

Drawing Points in 3D Space

A **point** in 3D space is defined by its coordinates (x, y, z). To visualize a point on a 2D canvas or WebGL viewport, we must project the 3D coordinate into 2D screen space, typically using a perspective projection (covered in earlier chapters).

For a simple canvas 2D context demo, we assume a projection function `project3Dto2D(x, y, z)` that converts 3D coordinates to 2D:

```
function drawPoint(ctx, x, y, z) {
  const { px, py } = project3Dto2D(x, y, z);
  ctx.beginPath();
  ctx.arc(px, py, 4, 0, Math.PI * 2); // Draw small circle for the point
  ctx.fill();
}
```

Connecting Points with Lines

Lines are fundamental for outlining shapes. In 3D, a line is defined by two endpoints (x_1, y_1, z_1) and (x_2, y_2, z_2). To draw the line, both points are projected to 2D, and then connected:

```
function drawLine(ctx, x1, y1, z1, x2, y2, z2) {
  const p1 = project3Dto2D(x1, y1, z1);
  const p2 = project3Dto2D(x2, y2, z2);
  ctx.beginPath();
  ctx.moveTo(p1.px, p1.py);
  ctx.lineTo(p2.px, p2.py);
  ctx.stroke();
}
```

By connecting multiple points in sequence, you can create polygons or wireframe models.

Creating Polygons and 3D Shapes

Polygons are closed shapes defined by multiple vertices connected by edges. In 3D, these polygons form faces of solid objects. A polygon can be drawn by connecting its vertices and filling the enclosed area.

Example of drawing a triangle (the simplest polygon):

```
function drawTriangle(ctx, v1, v2, v3) {
  const p1 = project3Dto2D(v1.x, v1.y, v1.z);
  const p2 = project3Dto2D(v2.x, v2.y, v2.z);
```

```
  const p3 = project3Dto2D(v3.x, v3.y, v3.z);

  ctx.beginPath();
  ctx.moveTo(p1.px, p1.py);
  ctx.lineTo(p2.px, p2.py);
  ctx.lineTo(p3.px, p3.py);
  ctx.closePath();
  ctx.fill();
  ctx.stroke();
}
```

Complex 3D shapes like cubes, pyramids, or more elaborate polyhedra are constructed by defining their vertices and connecting them into triangular or polygonal faces.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>3D Points, Lines, and Triangles Projection</title>
<style>
  body {
    margin: 0;
    background: #222;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    color: white;
    font-family: monospace;
  }
  canvas {
    background: #111;
    border: 1px solid #444;
  }
</style>
</head>
<body>
<canvas id="canvas" width="500" height="500"></canvas>
<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const centerX = canvas.width / 2;
const centerY = canvas.height / 2;
const cameraZ = 500; // distance from camera to origin on Z axis

// Simple perspective projection from 3D (x,y,z) to 2D (px, py)
function project3Dto2D(x, y, z) {
  const scale = cameraZ / (cameraZ + z);
  return {
    px: centerX + x * scale,
    py: centerY - y * scale // invert y for canvas coords
  };
}
```

```
// Draw a 3D point as a small circle on canvas
function drawPoint(ctx, x, y, z, color = 'white') {
  const { px, py } = project3Dto2D(x, y, z);
  ctx.fillStyle = color;
  ctx.beginPath();
  ctx.arc(px, py, 5, 0, Math.PI * 2);
  ctx.fill();
}

// Draw a 3D line between two 3D points
function drawLine(ctx, x1, y1, z1, x2, y2, z2, color = 'white') {
  const p1 = project3Dto2D(x1, y1, z1);
  const p2 = project3Dto2D(x2, y2, z2);
  ctx.strokeStyle = color;
  ctx.lineWidth = 2;
  ctx.beginPath();
  ctx.moveTo(p1.px, p1.py);
  ctx.lineTo(p2.px, p2.py);
  ctx.stroke();
}

// Draw a filled and stroked triangle from 3 vertices in 3D
function drawTriangle(ctx, v1, v2, v3, fillColor = 'rgba(100, 150, 250, 0.6)', strokeColor = 'white') {
  const p1 = project3Dto2D(v1.x, v1.y, v1.z);
  const p2 = project3Dto2D(v2.x, v2.y, v2.z);
  const p3 = project3Dto2D(v3.x, v3.y, v3.z);

  ctx.fillStyle = fillColor;
  ctx.strokeStyle = strokeColor;
  ctx.lineWidth = 2;
  ctx.beginPath();
  ctx.moveTo(p1.px, p1.py);
  ctx.lineTo(p2.px, p2.py);
  ctx.lineTo(p3.px, p3.py);
  ctx.closePath();
  ctx.fill();
  ctx.stroke();
}

// Example vertices in 3D space
const points3D = [
  { x: -100, y: 100, z: 50 },
  { x: 100, y: 100, z: 100 },
  { x: 50, y: -100, z: 0 },
  { x: -50, y: -50, z: 150 }
];

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

  // Draw points
  points3D.forEach((p, i) => {
    drawPoint(ctx, p.x, p.y, p.z, 'yellow');
    // Label points
    const { px, py } = project3Dto2D(p.x, p.y, p.z);
    ctx.fillStyle = 'white';
    ctx.font = '14px monospace';
```

```
    ctx.fillText(`P${i}`, px + 6, py - 6);
  });

  // Draw lines connecting points in order to form a shape
  for (let i = 0; i < points3D.length; i++) {
    let next = (i + 1) % points3D.length;
    const p1 = points3D[i];
    const p2 = points3D[next];
    drawLine(ctx, p1.x, p1.y, p1.z, p2.x, p2.y, p2.z, 'cyan');
  }

  // Draw a triangle (first 3 points)
  drawTriangle(ctx, points3D[0], points3D[1], points3D[2], 'rgba(50,200,50,0.5)', 'lime');
}

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

Interactive Rotation of 3D Shapes

One powerful way to visualize 3D objects on a flat canvas is by rotating them interactively. Rotation modifies the 3D coordinates before projection, allowing the object to spin and reveal its structure.

Here's a simplified rotation around the Y-axis for a point (x, y, z) by angle θ:

$$\begin{cases} x' = x\cos\theta - z\sin\theta \\ y' = y \\ z' = x\sin\theta + z\cos\theta \end{cases}$$

Applying this rotation to every vertex of a shape before projection simulates 3D rotation.

Runnable Example: Rotating Wireframe Cube

Below is a basic example of drawing a rotating wireframe cube using canvas 2D and simple 3D projection:

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

// Cube vertices
const vertices = [
  {x: -1, y: -1, z: -1},
  {x:  1, y: -1, z: -1},
  {x:  1, y:  1, z: -1},
  {x: -1, y:  1, z: -1},
  {x: -1, y: -1, z:  1},
  {x:  1, y: -1, z:  1},
```

```
  {x:  1, y:  1, z:  1},
  {x: -1, y:  1, z:  1},
];

// Cube edges (pairs of vertex indices)
const edges = [
  [0,1],[1,2],[2,3],[3,0], // back face
  [4,5],[5,6],[6,7],[7,4], // front face
  [0,4],[1,5],[2,6],[3,7]  // connecting edges
];

// Perspective projection function
function project3Dto2D(x, y, z) {
  const distance = 3;
  const scale = distance / (distance + z);
  return {
    px: width / 2 + x * scale * 100,
    py: height / 2 - y * scale * 100,
  };
}

// Rotate around Y axis
function rotateY(point, angle) {
  const cosA = Math.cos(angle);
  const sinA = Math.sin(angle);
  return {
    x: point.x * cosA - point.z * sinA,
    y: point.y,
    z: point.x * sinA + point.z * cosA,
  };
}

let angle = 0;
function animate() {
  ctx.clearRect(0, 0, width, height);

  // Rotate vertices
  const rotated = vertices.map(v => rotateY(v, angle));

  // Draw edges
  ctx.strokeStyle = 'black';
  edges.forEach(([i, j]) => {
    const p1 = project3Dto2D(rotated[i].x, rotated[i].y, rotated[i].z);
    const p2 = project3Dto2D(rotated[j].x, rotated[j].y, rotated[j].z);
    ctx.beginPath();
    ctx.moveTo(p1.px, p1.py);
    ctx.lineTo(p2.px, p2.py);
    ctx.stroke();
  });

  angle += 0.01;
  requestAnimationFrame(animate);
}

animate();
</script>
```

This code defines a cube's vertices and edges, rotates it around the Y-axis, projects the 3D

points to 2D, and draws the wireframe each frame, producing a smooth spinning cube.

Full runnable code:

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

// Cube vertices
const vertices = [
  {x: -1, y: -1, z: -1},
  {x:  1, y: -1, z: -1},
  {x:  1, y:  1, z: -1},
  {x: -1, y:  1, z: -1},
  {x: -1, y: -1, z:  1},
  {x:  1, y: -1, z:  1},
  {x:  1, y:  1, z:  1},
  {x: -1, y:  1, z:  1},
];

// Cube edges (pairs of vertex indices)
const edges = [
  [0,1],[1,2],[2,3],[3,0], // back face
  [4,5],[5,6],[6,7],[7,4], // front face
  [0,4],[1,5],[2,6],[3,7]  // connecting edges
];

// Perspective projection function
function project3Dto2D(x, y, z) {
  const distance = 3;
  const scale = distance / (distance + z);
  return {
    px: width / 2 + x * scale * 100,
    py: height / 2 - y * scale * 100,
  };
}

// Rotate around Y axis
function rotateY(point, angle) {
  const cosA = Math.cos(angle);
  const sinA = Math.sin(angle);
  return {
    x: point.x * cosA - point.z * sinA,
    y: point.y,
    z: point.x * sinA + point.z * cosA,
  };
}

let angle = 0;
function animate() {
  ctx.clearRect(0, 0, width, height);
```

```
  // Rotate vertices
  const rotated = vertices.map(v => rotateY(v, angle));

  // Draw edges
  ctx.strokeStyle = 'black';
  edges.forEach(([i, j]) => {
    const p1 = project3Dto2D(rotated[i].x, rotated[i].y, rotated[i].z);
    const p2 = project3Dto2D(rotated[j].x, rotated[j].y, rotated[j].z);
    ctx.beginPath();
    ctx.moveTo(p1.px, p1.py);
    ctx.lineTo(p2.px, p2.py);
    ctx.stroke();
  });

  angle += 0.01;
  requestAnimationFrame(animate);
}

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

Summary

- **Points** in 3D are coordinates (x, y, z) projected onto 2D space for display.
- **Lines** connect pairs of points and form edges of shapes.
- **Polygons**, especially triangles, are the building blocks of 3D solids.
- Interactive rotation involves applying rotation transformations to points before projection.
- Using canvas or WebGL, these primitives can be combined to create rich 3D visuals with smooth user interactions.

Mastering these basics paves the way for modeling complex objects and scenes in 3D animation, providing a foundation for advanced rendering techniques like shading, lighting, and texture mapping covered in later chapters.

22.2 Modeling Solids and Moving 3D Objects

Once you've mastered rendering points, lines, and basic shapes in 3D, the next step is building more **complex 3D models**—known as solids—by combining primitive elements like cubes, pyramids, and spheres. This section introduces techniques for constructing these solids and animating them in space using 3D transformations: **translation**, **rotation**, and **scaling**. We'll also provide runnable code examples to illustrate how these concepts apply in JavaScript.

Building Solids from Primitives

A **3D solid** is composed of connected faces (usually triangles or quadrilaterals) that enclose a volume. You can build complex models by defining their **vertices** and **faces**. For example, a cube has 8 vertices and 6 faces (each face being a square, made from two triangles).

Here's a structure to define a 3D mesh:

```
const cube = {
  vertices: [
    {x:-1, y:-1, z:-1}, {x:1, y:-1, z:-1},
    {x:1, y:1, z:-1},  {x:-1, y:1, z:-1},
    {x:-1, y:-1, z:1}, {x:1, y:-1, z:1},
    {x:1, y:1, z:1},   {x:-1, y:1, z:1}
  ],
  faces: [
    [0, 1, 2, 3], // back
    [4, 5, 6, 7], // front
    [0, 1, 5, 4], // bottom
    [2, 3, 7, 6], // top
    [1, 2, 6, 5], // right
    [0, 3, 7, 4]  // left
  ]
};
```

Each face references vertex indices. Later, we'll transform these vertices before projecting them to 2D space and drawing each face as a polygon.

3D Transformations: Translation, Rotation, Scaling

To animate 3D objects, you manipulate their vertices using transformation matrices or simpler vector math.

1. **Translation (Moving Position)**

Translation moves the entire model in space by adding an offset to each vertex:

```
function translate(vertex, tx, ty, tz) {
  return {
    x: vertex.x + tx,
    y: vertex.y + ty,
    z: vertex.z + tz
  };
}
```

2. **Scaling (Resizing)**

Scaling multiplies the size of each vertex from a fixed point (usually the origin):

```
function scale(vertex, sx, sy, sz) {
  return {
    x: vertex.x * sx,
    y: vertex.y * sy,
    z: vertex.z * sz
  };
}
```

3. **Rotation**

Here's how to rotate a point around the Y-axis:

```
function rotateY(vertex, angle) {
  const cos = Math.cos(angle);
  const sin = Math.sin(angle);
  return {
    x: vertex.x * cos - vertex.z * sin,
    y: vertex.y,
    z: vertex.x * sin + vertex.z * cos
  };
}
```

Similar formulas apply for rotating around the X or Z axes.

Rendering and Animating Solids

Once your object is transformed, you can project each vertex into 2D and draw the faces. Here's an example that animates a cube spinning and moving across the canvas:

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

const cube = {
  vertices: [
    {x:-1, y:-1, z:-1}, {x:1, y:-1, z:-1},
    {x:1, y:1, z:-1},  {x:-1, y:1, z:-1},
    {x:-1, y:-1, z:1}, {x:1, y:-1, z:1},
    {x:1, y:1, z:1},   {x:-1, y:1, z:1}
  ],
  faces: [
    [0, 1, 2, 3], [4, 5, 6, 7],
    [0, 1, 5, 4], [2, 3, 7, 6],
    [1, 2, 6, 5], [0, 3, 7, 4]
  ]
};

function project(vertex) {
  const scale = 300 / (vertex.z + 5);
  return {
    x: w/2 + vertex.x * scale,
    y: h/2 - vertex.y * scale
  };
}

let angle = 0;
function animate() {
  ctx.clearRect(0, 0, w, h);
  angle += 0.01;

  const transformed = cube.vertices.map(v => {
    let r = rotateY(v, angle);
    r = translate(r, Math.sin(angle) * 2, 0, 0); // Move left-right
    return r;
  });
```

```
  ctx.strokeStyle = "#000";
  ctx.fillStyle = "#89C";
  for (let face of cube.faces) {
    ctx.beginPath();
    const p0 = project(transformed[face[0]]);
    ctx.moveTo(p0.x, p0.y);
    for (let i = 1; i < face.length; i++) {
      const pi = project(transformed[face[i]]);
      ctx.lineTo(pi.x, pi.y);
    }
    ctx.closePath();
    ctx.fill();
    ctx.stroke();
  }

  requestAnimationFrame(animate);
}

animate();
</script>
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
const w = canvas.width;
const h = canvas.height;

const cube = {
  vertices: [
    {x:-1, y:-1, z:-1}, {x:1, y:-1, z:-1},
    {x:1, y:1, z:-1},  {x:-1, y:1, z:-1},
    {x:-1, y:-1, z:1}, {x:1, y:-1, z:1},
    {x:1, y:1, z:1},   {x:-1, y:1, z:1}
  ],
  faces: [
    [0, 1, 2, 3], [4, 5, 6, 7],
    [0, 1, 5, 4], [2, 3, 7, 6],
    [1, 2, 6, 5], [0, 3, 7, 4]
  ]
};

// Rotate around Y axis
function rotateY(v, angle) {
  const cosA = Math.cos(angle);
  const sinA = Math.sin(angle);
  return {
    x: v.x * cosA + v.z * sinA,
    y: v.y,
    z: -v.x * sinA + v.z * cosA
  };
}
```

```
// Translate vertex by dx, dy, dz
function translate(v, dx, dy, dz) {
  return {
    x: v.x + dx,
    y: v.y + dy,
    z: v.z + dz
  };
}

function project(vertex) {
  // Simple perspective projection
  const scale = 300 / (vertex.z + 5);
  return {
    x: w/2 + vertex.x * scale,
    y: h/2 - vertex.y * scale
  };
}

let angle = 0;
function animate() {
  ctx.clearRect(0, 0, w, h);
  angle += 0.01;

  const transformed = cube.vertices.map(v => {
    let r = rotateY(v, angle);
    r = translate(r, Math.sin(angle) * 2, 0, 0); // Move left-right
    return r;
  });

  ctx.strokeStyle = "#000";
  ctx.fillStyle = "#89C";
  for (let face of cube.faces) {
    ctx.beginPath();
    const p0 = project(transformed[face[0]]);
    ctx.moveTo(p0.x, p0.y);
    for (let i = 1; i < face.length; i++) {
      const pi = project(transformed[face[i]]);
      ctx.lineTo(pi.x, pi.y);
    }
    ctx.closePath();
    ctx.fill();
    ctx.stroke();
  }

  requestAnimationFrame(animate);
}

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

Summary

- Solids are composed of **vertices** and **faces**, often built from basic shapes.
- Use **translation**, **rotation**, and **scaling** to manipulate objects in 3D space.
- Animations involve continuously transforming vertices before rendering.

- Projection translates 3D points to 2D space so they can be drawn on a canvas.
- This technique forms the foundation for building interactive, animated 3D scenes with JavaScript and HTML5.

22.3 Using Triangles for 3D Fills

Triangles are the **fundamental building blocks** of 3D rendering. From simple models to complex animated characters, nearly all 3D graphics use triangles to define surfaces. Why? Triangles are mathematically stable, always planar (a triangle is always flat), and easy to render using both software and hardware rendering pipelines. In this section, we'll explore why triangles dominate 3D graphics, how to **rasterize** and fill them, and how basic **shading** can simulate depth and light. We'll conclude with a runnable demonstration using the HTML5 Canvas.

Why Triangles?

Unlike quads or polygons with more than three sides, **a triangle is always convex** and lies on a single plane. This makes it predictable and easy to break complex shapes into manageable pieces. Any polygon can be **triangulated**—split into non-overlapping triangles that cover the same area.

Benefits of triangles in 3D:

- They never warp or fold unpredictably.
- Graphics cards are optimized for triangle rasterization.
- Triangles are the simplest polygon that can form arbitrary shapes when combined.

Triangle Rasterization and Filling

Rasterization is the process of converting geometric data (like triangles) into pixels on a screen. In canvas-based rendering, we typically:

1. Project 3D vertices to 2D screen space.
2. Use `beginPath()`, `moveTo()`, and `lineTo()` to define triangle edges.
3. Fill the triangle using `fill()`.

Here's an example of drawing a single triangle from 3D space onto a 2D canvas:

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

function project(v) {
  const scale = 300 / (v.z + 5);
  return {
    x: w/2 + v.x * scale,
```

```
    y: h/2 - v.y * scale
  };
}

const triangle3D = [
  {x: -1, y: -1, z: 0},
  {x:  1, y: -1, z: 0},
  {x:  0, y:  1, z: 0}
];

function draw() {
  ctx.clearRect(0, 0, w, h);
  ctx.beginPath();
  const [a, b, c] = triangle3D.map(project);
  ctx.moveTo(a.x, a.y);
  ctx.lineTo(b.x, b.y);
  ctx.lineTo(c.x, c.y);
  ctx.closePath();
  ctx.fillStyle = '#49A';
  ctx.fill();
  ctx.strokeStyle = '#000';
  ctx.stroke();
}
draw();
</script>
```

This code projects a triangle in 3D space onto the 2D canvas and fills it with color.

Full runnable code:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
const w = canvas.width;
const h = canvas.height;

function project(v) {
  const scale = 300 / (v.z + 5);
  return {
    x: w/2 + v.x * scale,
    y: h/2 - v.y * scale
  };
}

const triangle3D = [
  {x: -1, y: -1, z: 0},
  {x:  1, y: -1, z: 0},
  {x:  0, y:  1, z: 0}
];

function draw() {
  ctx.clearRect(0, 0, w, h);
  ctx.beginPath();
```

```
  const [a, b, c] = triangle3D.map(project);
  ctx.moveTo(a.x, a.y);
  ctx.lineTo(b.x, b.y);
  ctx.lineTo(c.x, c.y);
  ctx.closePath();
  ctx.fillStyle = '#49A';
  ctx.fill();
  ctx.strokeStyle = '#000';
  ctx.stroke();
}
draw();
</script>
</body>
</html>
```

Basic Shading with Triangles

Although our triangle fill above uses a flat color, you can simulate light and depth using **simple shading**. One technique is **face-based shading**, where the triangle's brightness is based on its **surface normal**—a vector perpendicular to the triangle's face.

To compute a basic shading factor:

1. Compute the **normal vector** of the triangle using a cross product.
2. Compare it with a **light direction vector** using the dot product.
3. Use the result to darken or lighten the fill color.

Here's a simplified pseudo-code version of that logic:

```
// Assume v1, v2, v3 are triangle vertices
const edge1 = subtract(v2, v1);
const edge2 = subtract(v3, v1);
const normal = cross(edge1, edge2);
normalize(normal);

const lightDir = normalize({x: 0, y: 0, z: -1});
const brightness = Math.max(0, dot(normal, lightDir));
```

You can then use `brightness` to adjust the fill color dynamically.

Triangles in Practice

Modern 3D engines and even WebGL rely entirely on triangles. Complex models, like a human face or an animated dragon, are nothing more than thousands of triangles stitched together and shaded appropriately. By mastering triangle rasterization and shading—even at the canvas level—you gain the foundation for more advanced rendering systems.

Summary

- **Triangles are essential** in 3D rendering due to their simplicity, reliability, and efficiency.
- Using `canvas`, we can project and fill triangles to simulate 3D surfaces.
- **Shading techniques** add realism by simulating how light interacts with surfaces.
- Triangle-based rendering scales from simple canvas demos to high-performance 3D

engines.

In the next chapter, we'll explore how these triangles form the basis of rendering pipelines in **WebGL** and other hardware-accelerated contexts, opening the door to high-performance, real-time 3D animation.

Chapter 23.

Animation Tips and Tricks

1. Random and Brownian Motion
2. Distribution Techniques (Square, Circular, Biased)
3. Timer-Based vs Time-Based Animation
4. Sound Integration in Animation

23 Animation Tips and Tricks

23.1 Random and Brownian Motion

In animation, **random motion** adds an organic, lifelike unpredictability to the movement of particles, characters, or environmental effects. **Brownian motion**—named after botanist Robert Brown—is a model of chaotic, small-scale motion observed in particles suspended in fluid, and it is widely used in animation to simulate natural jitter, subtle drifts, and organic noise.

In this section, we'll cover basic and advanced forms of random motion, including Brownian motion, and demonstrate how to apply these techniques in JavaScript using the HTML5 canvas.

23.1.1 Random Motion Basics

Random motion involves assigning unpredictable changes to an object's position or velocity over time. A basic particle system might use random numbers to give each particle a direction and speed.

```
let particle = {
  x: 200,
  y: 200,
  vx: Math.random() * 4 - 2, // random velocity between -2 and 2
  vy: Math.random() * 4 - 2
};
```

In an animation loop, we move the particle like so:

```
function update() {
  particle.x += particle.vx;
  particle.y += particle.vy;
}
```

Each frame, the particle moves in a straight line based on its randomly initialized velocity. This results in **uniform linear random motion**, but it lacks the subtle, jittery feel of natural motion.

23.1.2 Simulating Brownian Motion

Brownian motion builds upon random motion by **continuously modifying** the velocity at each step. Instead of keeping velocity constant, we change it slightly every frame:

```
function update() {
  particle.vx += Math.random() * 0.4 - 0.2;
  particle.vy += Math.random() * 0.4 - 0.2;
```

```
  particle.x += particle.vx;
  particle.y += particle.vy;
}
```

Here, `vx` and `vy` are "nudged" randomly each frame. The result is an erratic yet smooth path—ideal for simulating the motion of dust, smoke, or tiny insects.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Brownian Motion Simulation</title>
<style>
  body {
    margin: 0;
    background: #111;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
  }
  canvas {
    background: #222;
    border: 1px solid #444;
  }
</style>
</head>
<body>
<canvas id="canvas" width="600" height="400"></canvas>
<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const particle = {
  x: canvas.width / 2,
  y: canvas.height / 2,
  vx: 0,
  vy: 0,
  radius: 8,
};

function update() {
  // Randomly jitter velocity (Brownian motion)
  particle.vx += Math.random() * 0.4 - 0.2;
  particle.vy += Math.random() * 0.4 - 0.2;

  // Update position by velocity
  particle.x += particle.vx;
  particle.y += particle.vy;

  // Keep particle inside canvas bounds (bounce)
  if (particle.x < particle.radius) {
    particle.x = particle.radius;
    particle.vx *= -1;
  } else if (particle.x > canvas.width - particle.radius) {
```

```
    particle.x = canvas.width - particle.radius;
    particle.vx *= -1;
  }

  if (particle.y < particle.radius) {
    particle.y = particle.radius;
    particle.vy *= -1;
  } else if (particle.y > canvas.height - particle.radius) {
    particle.y = canvas.height - particle.radius;
    particle.vy *= -1;
  }
}

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

  // Draw particle
  ctx.beginPath();
  ctx.arc(particle.x, particle.y, particle.radius, 0, Math.PI * 2);
  ctx.fillStyle = '#00ffcc';
  ctx.fill();
}

function animate() {
  update();
  draw();
  requestAnimationFrame(animate);
}

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

23.1.3 Example: Brownian Motion in Canvas

Here's a runnable example demonstrating Brownian motion:

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

const particles = Array.from({length: 50}, () => ({
  x: Math.random() * canvas.width,
  y: Math.random() * canvas.height,
  vx: 0,
  vy: 0
}));

function update() {
  for (const p of particles) {
    p.vx += Math.random() * 0.4 - 0.2;
    p.vy += Math.random() * 0.4 - 0.2;
```

```
    p.x += p.vx;
    p.y += p.vy;

    // keep particles inside bounds
    if (p.x < 0 || p.x > canvas.width) p.vx *= -1;
    if (p.y < 0 || p.y > canvas.height) p.vy *= -1;
  }
}

function draw() {
  ctx.fillStyle = "rgba(255,255,255,0.1)";
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  ctx.fillStyle = "black";
  for (const p of particles) {
    ctx.beginPath();
    ctx.arc(p.x, p.y, 2, 0, Math.PI * 2);
    ctx.fill();
  }
}

function loop() {
  update();
  draw();
  requestAnimationFrame(loop);
}
loop();
</script>
```

In this animation, each particle “jitters” across the screen, creating a believable and fluid simulation of Brownian motion.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Brownian Motion Example</title>
<style>
  body {
    margin: 0;
    background: #eee;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
  }
  canvas {
    background: white;
    border: 1px solid #ccc;
  }
</style>
</head>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
```

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

const particles = Array.from({ length: 50 }, () => ({
  x: Math.random() * canvas.width,
  y: Math.random() * canvas.height,
  vx: 0,
  vy: 0
}));

function update() {
  for (const p of particles) {
    // Brownian velocity jitter
    p.vx += Math.random() * 0.4 - 0.2;
    p.vy += Math.random() * 0.4 - 0.2;

    p.x += p.vx;
    p.y += p.vy;

    // Keep particles inside bounds with simple bounce
    if (p.x < 0) {
      p.x = 0;
      p.vx *= -1;
    } else if (p.x > canvas.width) {
      p.x = canvas.width;
      p.vx *= -1;
    }

    if (p.y < 0) {
      p.y = 0;
      p.vy *= -1;
    } else if (p.y > canvas.height) {
      p.y = canvas.height;
      p.vy *= -1;
    }
  }
}

function draw() {
  // Slight fade to create trailing effect
  ctx.fillStyle = "rgba(255, 255, 255, 0.1)";
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  // Draw particles as black dots
  ctx.fillStyle = "black";
  for (const p of particles) {
    ctx.beginPath();
    ctx.arc(p.x, p.y, 2, 0, Math.PI * 2);
    ctx.fill();
  }
}

function loop() {
  update();
  draw();
  requestAnimationFrame(loop);
}
```

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

23.1.4 Adding Smooth Noise with Perlin or Simplex Noise

While Brownian motion gives erratic randomness, **Perlin noise** and **Simplex noise** provide **coherent randomness**, which is ideal for smooth, natural-looking drift—like clouds or underwater currents.

Libraries like SimplexNoise.js can generate noise values that you can use as velocity components:

```
const noise = new SimplexNoise();

function updateWithNoise(t) {
  for (let i = 0; i < particles.length; i++) {
    const p = particles[i];
    p.vx = noise.noise2D(i, t) * 2;
    p.vy = noise.noise2D(i + 1000, t) * 2;

    p.x += p.vx;
    p.y += p.vy;
  }
}
```

This creates a **smooth, wave-like** movement over time rather than jitter.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Smooth Noise Motion with Simplex Noise</title>
<style>
  body {
    margin: 0;
    background: #222;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
  }
  canvas {
    background: #eee;
    border: 1px solid #ccc;
  }
</style>
</head>
<body>
```

```
<canvas id="canvas" width="500" height="500"></canvas>

<!-- Include SimplexNoise from CDN -->
<script src="https://cdn.jsdelivr.net/npm/simplex-noise@2.4.0/simplex-noise.min.js"></script>
<script>
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

const noise = new SimplexNoise();

const particles = Array.from({ length: 100 }, () => ({
  x: Math.random() * canvas.width,
  y: Math.random() * canvas.height,
  vx: 0,
  vy: 0
}));

function update(t) {
  // t in seconds for smooth noise progression
  const time = t * 0.001;

  for (let i = 0; i < particles.length; i++) {
    const p = particles[i];
    // Noise based velocities; different offsets for x and y
    p.vx = noise.noise2D(i, time) * 1.5;
    p.vy = noise.noise2D(i + 1000, time) * 1.5;

    p.x += p.vx;
    p.y += p.vy;

    // Wrap around edges for continuous motion
    if (p.x < 0) p.x += canvas.width;
    if (p.x > canvas.width) p.x -= canvas.width;
    if (p.y < 0) p.y += canvas.height;
    if (p.y > canvas.height) p.y -= canvas.height;
  }
}

function draw() {
  // Clear with slight opacity for trailing effect
  ctx.fillStyle = "rgba(255,255,255,0.1)";
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  ctx.fillStyle = "#003366";
  for (const p of particles) {
    ctx.beginPath();
    ctx.arc(p.x, p.y, 3, 0, Math.PI * 2);
    ctx.fill();
  }
}

function loop(time) {
  update(time);
  draw();
  requestAnimationFrame(loop);
}

loop();
```

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

23.1.5 Jitter Effects

Jitter refers to small, rapid, and unpredictable shifts—often added on top of other animations to make them feel alive.

For example, to jitter a button slightly on hover:

```
button.x += Math.random() * 2 - 1;
button.y += Math.random() * 2 - 1;
```

This creates a subtle, energetic vibration, often seen in UI elements, creature movement, or energetic particles.

23.1.6 Summary

- **Random motion** gives particles or objects unpredictability.
- **Brownian motion** enhances realism by randomly modifying velocity, not just position.
- You can simulate **chaotic behavior** with simple loops and random number generation.
- For smoother randomness, use **Perlin or Simplex noise**.
- **Jitter effects** make animations feel more organic and reactive.

Understanding and mastering random and Brownian motion enables animators to inject subtle lifelike motion into simulations, from particle systems to UI feedback. In the next section, we'll look at how **distribution techniques** shape random behaviors spatially—creating more control over where and how randomness appears.

23.2 Distribution Techniques (Square, Circular, Biased)

In animation and interactive graphics, distributing elements like particles, icons, or visual effects evenly or artistically across a canvas is a common and powerful technique. Whether you're creating a starfield, visualizing a data cluster, or animating a swarm of particles, **how** those elements are positioned greatly affects the visual output.

This section explores **square**, **circular**, and **biased/randomized** distribution methods. We'll cover their mathematical foundations and provide JavaScript examples to demonstrate how each type of distribution can be implemented for visual animations.

23.2.1 Square Grid Distribution

Square grids are commonly used for laying out elements in a uniform, predictable fashion—great for tiling, pixel art, or spatial patterns like matrices or dot fields.

Code Example: Square Grid of Particles

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

const spacing = 20;
const cols = canvas.width / spacing;
const rows = canvas.height / spacing;

function drawGrid() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.fillStyle = "black";
  for (let i = 0; i < cols; i++) {
    for (let j = 0; j < rows; j++) {
      let x = i * spacing + spacing / 2;
      let y = j * spacing + spacing / 2;
      ctx.beginPath();
      ctx.arc(x, y, 2, 0, Math.PI * 2);
      ctx.fill();
    }
  }
}
drawGrid();
```

Each point is positioned using simple multiplication of its grid index by a fixed spacing. This ensures clean and repeatable placement.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Square Grid of Particles</title>
<style>
  body {
    margin: 0;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    background: #f0f0f0;
  }
  canvas {
    border: 1px solid #ccc;
    background: white;
  }
</style>
</head>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
```

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

const spacing = 20;
const cols = Math.floor(canvas.width / spacing);
const rows = Math.floor(canvas.height / spacing);

function drawGrid() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.fillStyle = "black";
  for (let i = 0; i < cols; i++) {
    for (let j = 0; j < rows; j++) {
      let x = i * spacing + spacing / 2;
      let y = j * spacing + spacing / 2;
      ctx.beginPath();
      ctx.arc(x, y, 3, 0, Math.PI * 2);
      ctx.fill();
    }
  }
}

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

23.2.2 Circular Distribution

In circular distributions, elements are placed around or inside a circle. This is useful for visualizations like clocks, radar charts, or radial explosions.

Uniform Circle Perimeter Distribution

For evenly spaced objects along the **circumference** of a circle:

```
const centerX = canvas.width / 2;
const centerY = canvas.height / 2;
const radius = 100;
const count = 20;

function drawCircleRing() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.fillStyle = "blue";
  for (let i = 0; i < count; i++) {
    let angle = (Math.PI * 2 / count) * i;
    let x = centerX + radius * Math.cos(angle);
    let y = centerY + radius * Math.sin(angle);
    ctx.beginPath();
    ctx.arc(x, y, 4, 0, Math.PI * 2);
    ctx.fill();
  }
}
drawCircleRing();
```

By rotating evenly around 360 degrees (2Pi radians), this method produces symmetric and aesthetic layouts.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Circular Distribution</title>
<style>
  body {
    margin: 0;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    background: #fafafa;
  }
  canvas {
    border: 1px solid #ccc;
    background: white;
  }
</style>
</head>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

const centerX = canvas.width / 2;
const centerY = canvas.height / 2;
const radius = 100;
const count = 20;

function drawCircleRing() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.fillStyle = "blue";
  for (let i = 0; i < count; i++) {
    let angle = (Math.PI * 2 / count) * i;
    let x = centerX + radius * Math.cos(angle);
    let y = centerY + radius * Math.sin(angle);
    ctx.beginPath();
    ctx.arc(x, y, 6, 0, Math.PI * 2);
    ctx.fill();
  }
}

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

Uniform Circle Area Distribution

Randomly placing points **inside** a circle requires correction for density. Without care, points will cluster at the center. The fix is to scale radius with the **square root** of a random number:

```
function randomCirclePoint(cx, cy, r) {
  const angle = Math.random() * Math.PI * 2;
  const dist = Math.sqrt(Math.random()) * r;
  return {
    x: cx + dist * Math.cos(angle),
    y: cy + dist * Math.sin(angle)
  };
}
```

This ensures a **uniform area distribution**, making the points evenly fill the circle rather than concentrating in the middle.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Uniform Circle Area Distribution</title>
<style>
  body {
    margin: 0;
    background: #fafafa;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
  }
  canvas {
    border: 1px solid #ccc;
    background: white;
  }
</style>
</head>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

const centerX = canvas.width / 2;
const centerY = canvas.height / 2;
const radius = 150;
const count = 200;

function randomCirclePoint(cx, cy, r) {
  const angle = Math.random() * Math.PI * 2;
  const dist = Math.sqrt(Math.random()) * r;
  return {
    x: cx + dist * Math.cos(angle),
    y: cy + dist * Math.sin(angle)
```

```
  };
}

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

  // Draw circle boundary for reference
  ctx.strokeStyle = "gray";
  ctx.beginPath();
  ctx.arc(centerX, centerY, radius, 0, Math.PI * 2);
  ctx.stroke();

  ctx.fillStyle = "tomato";
  for (let i = 0; i < count; i++) {
    const p = randomCirclePoint(centerX, centerY, radius);
    ctx.beginPath();
    ctx.arc(p.x, p.y, 3, 0, Math.PI * 2);
    ctx.fill();
  }
}

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

23.2.3 Biased and Randomized Distributions

Sometimes uniformity is too mechanical. **Biased** distributions allow for more natural or purposeful layouts.

Gaussian (Normal) Distribution

Instead of `Math.random()`, you can use a Gaussian (bell curve) function to bias points toward the center:

```
function gaussianRandom(mean = 0, stddev = 1) {
  let u = 1 - Math.random();
  let v = 1 - Math.random();
  return mean + stddev * Math.sqrt(-2 * Math.log(u)) * Math.cos(2 * Math.PI * v);
}
```

You can use this to bias the placement of particles, making them cluster around a center point—ideal for simulating things like stars, clouds, or populations.

Example: Radial Clustered Particles

```
function drawGaussianCluster(cx, cy, stddev, count) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.fillStyle = "red";
  for (let i = 0; i < count; i++) {
    let x = cx + gaussianRandom(0, stddev);
```

```
    let y = cy + gaussianRandom(0, stddev);
    ctx.beginPath();
    ctx.arc(x, y, 2, 0, Math.PI * 2);
    ctx.fill();
  }
}
drawGaussianCluster(canvas.width / 2, canvas.height / 2, 50, 200);
```

This results in a dense center and a gradual falloff—a much more **naturalistic** pattern than flat randomness.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Radial Clustered Particles</title>
<style>
  body {
    margin: 0;
    background: #fafafa;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
  }
  canvas {
    border: 1px solid #ccc;
    background: white;
  }
</style>
</head>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
<script>
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

// Box-Muller transform for Gaussian random numbers
function gaussianRandom(mean = 0, stddev = 1) {
  let u = 0, v = 0;
  while(u === 0) u = Math.random(); // Converting [0,1) to (0,1)
  while(v === 0) v = Math.random();
  let num = Math.sqrt(-2.0 * Math.log(u)) * Math.cos(2.0 * Math.PI * v);
  return num * stddev + mean;
}

function drawGaussianCluster(cx, cy, stddev, count) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.fillStyle = "red";
  for (let i = 0; i < count; i++) {
    let x = cx + gaussianRandom(0, stddev);
    let y = cy + gaussianRandom(0, stddev);
    ctx.beginPath();
    ctx.arc(x, y, 2, 0, Math.PI * 2);
    ctx.fill();
```

```
  }
}

drawGaussianCluster(canvas.width / 2, canvas.height / 2, 50, 200);
</script>
</body>
</html>
```

23.2.4 Summary

Distribution	Use Case	Characteristics
Square Grid	Tiling, matrices	Predictable, rigid
Circular (Perimeter)	Dials, rings	Radial symmetry
Circular (Area)	Radial scatter	Uniform density
Biased (Gaussian)	Natural clustering	Organic randomness

Combining distribution techniques with animation unlocks an enormous range of effects—from particles spiraling out of a circular explosion to UI elements smoothly populating a grid or forming organic swarms.

In the next section, we'll explore **time-based vs timer-based animation**, a critical topic when synchronizing visual effects across frames or devices.

23.3 Timer-Based vs Time-Based Animation

When animating objects in JavaScript, choosing the right approach for updating movement and rendering is crucial for performance, consistency, and smoothness. Two main methods exist: **timer-based** and **time-based** animation. Understanding their differences—and when to use each—will help you create robust and fluid animations that perform reliably across devices and browsers.

23.3.1 Timer-Based Animation

Timer-based animation uses fixed time intervals to update the scene. The most common method is `setInterval()` or `setTimeout()`, where you define how often the update function should run.

Example:

```
setInterval(() => {
  x += 2; // Move 2 units every frame
  draw();
}, 1000 / 60); // Aim for 60 frames per second
```

In this example, the animation updates at a consistent 60 FPS—**in theory**. However, in practice:

Pros:

- Simple to implement.
- Easy to reason about for static frame rates.

Cons:

- **Frame rate drops or spikes** can cause animation stutter or speed inconsistencies.
- `setInterval()` doesn't align with the screen's refresh rate.
- If the system lags or loses focus, timer events can **stack or skip**, causing jittery movement.

In short, timer-based animation assumes a fixed frame rate, which is rarely guaranteed in the real world.

23.3.2 Time-Based Animation

Time-based animation, in contrast, calculates movement based on the actual **elapsed time** between frames, making it resilient to variations in frame rate. This technique uses `requestAnimationFrame()`, which syncs animation with the display refresh cycle (typically 60Hz, or 60 frames per second).

Example:

```
let lastTime = 0;
let x = 0;

function animate(time) {
  const delta = (time - lastTime) / 1000; // Convert ms to seconds
  lastTime = time;

  x += 100 * delta; // 100 units per second
  draw(x);

  requestAnimationFrame(animate);
}
requestAnimationFrame(animate);
```

In this example, we update `x` based on how much **real time** has passed (`delta`). Whether the frame took 16ms or 33ms, the motion will adjust proportionally, maintaining **smooth**

speed.

Pros:

- Motion stays consistent across high- and low-refresh monitors.
- Pauses or slow frames won't break the animation logic.
- Energy-efficient: `requestAnimationFrame()` pauses in background tabs.

Cons:

- Slightly more complex to set up.
- Requires managing timestamps and deltas.

23.3.3 Comparing Both Approaches

Feature	Timer-Based	Time-Based (`requestAnimationFrame`)
Frame Rate Sync	NO Not guaranteed	YES Matches display refresh rate
Smoothness Across Devices	NO Inconsistent	YES High consistency
CPU Efficiency	NO Always running	YES Pauses in inactive tabs
Jitter/Stutter Risk	NO High on slow devices	YES Adapts to slowdowns
Time-Based Motion	NO Needs extra handling	YES Built-in via timestamp

23.3.4 Implementing a Time-Based Game Loop

Here's a reusable template you can drop into most animation projects:

```
let lastTime = 0;

function loop(currentTime) {
  const deltaTime = (currentTime - lastTime) / 1000; // seconds
  lastTime = currentTime;

  update(deltaTime);
  draw();

  requestAnimationFrame(loop);
}
requestAnimationFrame(loop);

function update(dt) {
  // dt is the time since last frame (in seconds)
```

```
  object.x += object.vx * dt;
  object.y += object.vy * dt;
}
```

In this pattern:

- `update(dt)` scales movement to real time.
- Velocity is measured in units per second.
- Everything remains smooth regardless of actual frame rate.

23.3.5 Bonus: Mixing Techniques

In some cases, you may use both. For example:

- Use `requestAnimationFrame()` for **drawing and movement**.
- Use `setInterval()` for **logic or AI ticks** (e.g., once every second).

However, for **real-time visual updates**, time-based animation should be your default.

23.3.6 Summary

Timer-based animation is quick to implement but fragile in dynamic environments. Time-based animation, though slightly more complex, results in much smoother and more reliable performance—especially on modern devices with variable frame rates. By switching to `requestAnimationFrame()` and measuring `deltaTime`, you gain better control over animation timing and unlock a more professional, polished user experience.

In the next section, we'll explore how to synchronize animations with **sound**, creating even more immersive and dynamic web experiences.

23.4 Sound Integration in Animation

Sound is a powerful enhancer in web animation, adding emotion, depth, and interactivity to visual experiences. Whether you're creating a game, interactive infographic, or playful UI, integrating audio with motion can transform a static animation into an engaging, multisensory experience. This section covers the fundamentals of synchronizing audio with animations in JavaScript using event-triggered sounds, timed cues, and the powerful **Web Audio API**.

23.4.1 Playing Sounds on Events

The most straightforward way to integrate sound is to trigger playback when something happens—like a button click, collision, or animation completion.

Here's a basic example using HTML5's `<audio>` element:

```
<audio id="clickSound" src="click.mp3" preload="auto"></audio>
<button onclick="playSound()">Click Me</button>

<script>
  function playSound() {
    document.getElementById("clickSound").play();
  }
</script>
```

This works well for simple, non-overlapping effects. However, for richer interactions—such as dynamically altering pitch or synchronizing sounds with continuous motion—we need more control. That's where the **Web Audio API** shines.

23.4.2 Introducing the Web Audio API

The Web Audio API allows fine-grained control over audio playback, timing, spatialization, volume, and effects. To begin, we create an `AudioContext`, which serves as the sound engine:

```
const audioCtx = new (window.AudioContext || window.webkitAudioContext)();
```

To play a sound file:

```
async function loadAndPlay(url) {
  const response = await fetch(url);
  const arrayBuffer = await response.arrayBuffer();
  const audioBuffer = await audioCtx.decodeAudioData(arrayBuffer);

  const source = audioCtx.createBufferSource();
  source.buffer = audioBuffer;
  source.connect(audioCtx.destination);
  source.start();
}
```

You could now call `loadAndPlay('beep.wav')` whenever your animation reaches a key moment (e.g., a bounce or impact).

23.4.3 Timing Animations to Beats or Cues

Animations can also be driven by audio—not just react to it. For instance, in music visualizers or rhythm-based games, animation must be synchronized with audio cues. You can use the `currentTime` property of the `AudioContext` to align visuals precisely with audio playback.

Example: Beat-Synced Flash

```
let nextBeat = 0;
const bpm = 120;
const interval = 60 / bpm; // seconds per beat

function animate(time) {
  const ctxTime = audioCtx.currentTime;

  if (ctxTime >= nextBeat) {
    flash(); // Animate something synced to the beat
    nextBeat += interval;
  }

  requestAnimationFrame(animate);
}
```

This structure ensures a visual update occurs exactly every beat, even if the frame rate fluctuates.

23.4.4 Controlling Motion with Sound

Beyond syncing animations to beats, you can use sound data to dynamically modify visuals—great for equalizer effects or dancing particles. The Web Audio API provides an `AnalyserNode` that delivers real-time frequency and amplitude data.

Example: Audio-Driven Bouncing Circle

```
const analyser = audioCtx.createAnalyser();
source.connect(analyser);
analyser.connect(audioCtx.destination);

const dataArray = new Uint8Array(analyser.frequencyBinCount);

function render() {
  analyser.getByteFrequencyData(dataArray);
  const average = dataArray.reduce((a, b) => a + b, 0) / dataArray.length;

  const radius = 10 + average / 10;

  drawCircle(radius); // Scale based on audio level
  requestAnimationFrame(render);
}
```

This setup dynamically adjusts the size of a circle based on the intensity of the audio signal.

23.4.5 Sound Design Tips

- **Keep latency low**: `AudioBufferSourceNode` provides near-zero latency if decoded ahead of time.
- **Avoid overlapping**: If sound overlaps feel messy, stop and restart or limit playback frequency.
- **Mobile restrictions**: Many browsers block autoplay of sound until the user interacts with the page.
- **Add variation**: Slightly alter pitch, volume, or timing to keep repeated sounds from feeling robotic.

23.4.6 Putting It All Together

Here's a complete example: a ball bounces, triggering a drum sound each time it hits the ground.

```
<canvas id="canvas" width="400" height="300"></canvas>
<audio id="drum" src="drum-hit.mp3" preload="auto"></audio>
```

```
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");
let y = 0, vy = 0, gravity = 0.5;

function playHit() {
  const drum = document.getElementById("drum");
  drum.currentTime = 0;
  drum.play();
}

function loop() {
  vy += gravity;
  y += vy;

  if (y > canvas.height - 20) {
    y = canvas.height - 20;
    vy *= -0.8;
    playHit();
  }

  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.beginPath();
  ctx.arc(200, y, 20, 0, Math.PI * 2);
  ctx.fill();

  requestAnimationFrame(loop);
}
loop();
```

23.4.7 Summary

Sound integration adds a vital layer of expressiveness to HTML5 animations. Whether you're playing sounds on events, syncing visuals to audio beats, or generating animations directly from audio data, JavaScript and the Web Audio API give you powerful tools to create immersive experiences. With careful design, you can bring your animations to life in ways that not only look good—but sound amazing too.