ChatGPT

HTML5 Overlay Effects in Weebly: Enhancing Games and Websites

Introduction:
HTML5 overlay effects are visual layers placed on top of a webpage or game canvas to create atmosphere, feedback, or stylistic enhancements. In a Weebly context, these effects can be embedded via the **Embed Code** element, allowing custom HTML/CSS/JS overlays even in a drag-and-drop site. This report explores a wide range of overlay effects – from realistic weather (rain, snow) to screen distortions (glitches, static) – and discusses how to implement them for both web games and general websites. We'll cover the types of effects achievable, techniques (CSS3, Canvas, WebGL, SVG, etc.), asset and performance considerations, Weebly-specific embedding tips, and code examples for key effects.

Types of Visual Overlay Effects

Weather & Environmental Overlays: These effects simulate natural phenomena to make a scene more immersive. *Rain* overlays can include falling raindrops and water streaks that resemble rain hitting a camera lens or screen. For example, **RainyDay.js** is a library that renders realistic raindrops on a "glass" (screen) using an HTML5 Canvas [1] [2]. This produces droplets that splash and drip, making it look like the viewer is peering through a rainy window. *Snow* overlays involve many slowly falling flakes; one can draw snowflakes with canvas or position `<div>` elements with CSS animations. A canvas approach with JavaScript can create an animated snowfall effect to add holiday flair to a site [3]. *Fog and mist* can be overlaid as semi-transparent moving clouds. This might be done by drawing translucent noise patterns on a canvas (simulating fog via Perlin noise) [4] or by layering a PNG image of fog with CSS animation. *Smoke* can be rendered with particle effects that drift upward, useful in games (e.g. smoke from fires or explosions). Libraries like **smoke.js** produce interactive, realistic smoke using an HTML5 canvas [5]. *Fire* overlays (flames, fireballs) often use animated particles or sprite sheets. A simple flame effect can be achieved by emitting colored particles with gradients and using additive blending on canvas [6]. These environmental overlays enrich game scenes (for weather or ambiance) and can also be used on websites (for example, a light snowfall effect on a winter sale page).

Liquid & Water Effects: These overlays simulate water on the screen or lens. For instance, raindrop effects often include *water droplets* that distort the content behind them and *dripping* trails as they roll down. The RainyDay.js library mentioned above handles droplet dynamics (gravity, splashes, trails) to make rain look convincing [7] [8]. You might also overlay *water ripples* or splashes when needed (e.g. a ripple effect on a background image to simulate water). Such effects typically use either canvas (manipulating pixels or using WebGL shaders for refraction) or simple animated GIF/PNG overlays for splashes. A water splash could be an animated sprite that plays on an event (like a click or a game action).

Screen Distortion & Glitch Effects: These overlays intentionally degrade or alter the view to create a stylized effect. *Screen glitch* overlays mimic digital interference – tearing, offset colors, or flickering bands – often seen in horror or cyberpunk designs. A common technique is layering multiple copies of the same element (text or image) and shifting or clipping them with animations [9]. For example, one layer might be tinted red and nudged a few pixels left, another tinted blue and nudged right, creating an RGB split glitch. Using CSS, you can apply animations that randomly clip portions of these layers to produce jittery "broken signal" slices. The snippet below illustrates a simple image glitch overlay using an extra pseudo-element layer:

```
<!-- Glitch effect: an image with an overlaid, offset duplicate -->
<div class="glitch-img">
  <img src="photo.jpg" alt="Glitch example">
</div>

<style>
.glitch-img { position: relative; display: inline-block; }
.glitch-img::after {
  content: "";
  position: absolute; top: 0; left: 0; width: 100%; height: 100%;
  background: url('photo.jpg') no-repeat center/cover;
  /* Apply a colored tint and blending to the duplicate layer */
  background-color: magenta; mix-blend-mode: screen;
  opacity: 0.8;
  transform: translate(2px, 0);
  animation: glitchShift 0.5s infinite alternate;
}
@keyframes glitchShift { to { transform: translate(-2px, -2px); } }
</style>
```

In the above code, a pseudo-element overlays the image with a magenta-tinted copy. The `mix-blend-mode: screen` makes the overlay interact with the original, and the `glitchShift` animation jitters the layer to create a subtle glitch. More advanced glitches might use additional layers and faster, random clip changes [10] – for example, using CSS `clip-path` or SVG filters – but even this basic approach gives a glitchy vibe. *Screen static* (TV noise) overlays are another related effect. These involve a moving grain or snow pattern on the screen, reminiscent of a detuned analog TV. This can be achieved by absolutely positioning a `<div>` over the content with a noisy static texture. A small noise image (or an SVG filter) can be tiled and its opacity and position rapidly adjusted to simulate flickering static. For instance, one can use a CSS animation to quickly toggle the opacity or shift the background of a noise layer:

```
.static-overlay {
  pointer-events: none;               /* allow clicks through */
  position: absolute; top: 0; left: 0; width: 100%; height: 100%;
  background: url('static-noise.png') repeat;
  opacity: 0.3;
  animation: flicker 0.2s steps(2) infinite;
}
@keyframes flicker { to { opacity: 0.4; } }
```

Here, a semi-transparent noise image covers the screen and the `flicker` animation causes slight changes in opacity every few frames (using `steps(2)` for a jumpy transition), producing a convincing static effect. Modern CSS even allows pure procedural noise via fragment shaders or quirky gradient hacks (e.g. animating a subpixel radial gradient to generate static) – an advanced technique that can achieve 60fps noise purely in CSS [11] [12], but using an image is simpler and more widely compatible.

Other distortion overlays include *CRT scanlines* (thin horizontal lines across the screen to emulate old monitors) and *signal distortion* (wavy or shaky visuals). Scanlines can be done with a semi-transparent PNG of horizontal lines, or using CSS `repeating-linear-gradient` to draw lines, overlaid with `mix-blend-mode: multiply` or overlay. A subtle *heat shimmer* (like a hot road mirage) can be considered a distortion overlay: it slightly warps the underlying image. This is harder to do with pure CSS but can be achieved with WebGL shaders or an SVG **feDisplacementMap** filter. For example, a WebGL fragment shader can displace pixels based on a sine wave to simulate rising heat haze [13] [14]. There are tutorials demonstrating realistic heat distortion using fragment shaders in WebGL [15], which can be applied to a canvas or WebGL background for a game scene. For simpler cases, one could overlay a wavy translucent texture and animate it to fake a heat ripple.

Vision Filter Overlays: These effects replicate specialized camera or vision modes by altering colors and clarity. *Night vision* overlay is typically a green monochrome view with scanlines and a slight glow or noise. This can be achieved by layering a semi-transparent green overlay and a pattern of lines or grain on top of your content, then adjusting brightness/contrast. For example, one CSS approach uses multiple backgrounds: the original content, a radial gradient (green center fading to black at edges for a vignette), and a repeating semi-transparent black line pattern for scanlines, all blended together [16]. The snippet below (in SCSS-like form) shows the concept as taken from an image-effects library:

```
.night-vision-effect {
  background-image:
    url('photo.jpg'),                                   /* original image */
    radial-gradient(#0f0, #000),                        /* green tint with
vignette */
    repeating-linear-gradient(transparent 0, rgba(0,0,0,0.1) 2.5px, transparent
5px); /* scanlines */
  background-blend-mode: overlay;
  background-size: cover;
  background-position: center;
}
```

In the above, the radial gradient #0F0→#000 tints the image green (bright green center, dark edges), and the repeating linear gradient creates 5px-spaced horizontal lines with a slight transparency (to simulate the scanline glow). Using `background-blend-mode: overlay` mixes these layers to produce a composite night-vision look [16]. Additional CSS `filter: brightness(1.5) contrast(2)` could boost the contrast, making bright areas glow. A bit of grain noise (as discussed for static) can be added for authenticity. *Thermal vision* (infrared) overlays, in contrast, apply a false-color palette (e.g. black for cold, through blue/green, up to red/white for hot). Implementing this precisely usually requires pixel-by-pixel processing (e.g. via WebGL shader or canvas manipulation) because you have to map intensity to colors. However, an approximation can be done by heavy color filtering: for example, using CSS `hue-rotate` and `invert` or blending in colored gradients. One hack is to overlay multiple color-tinted copies of the content using mix-blend-mode *color* or *difference*. Real thermal views might be out of scope for pure CSS, but libraries or shader techniques can do it (e.g. using WebGL to sample a grayscale image and apply a colormap gradient).

More straightforward **color overlays** include things like damage indicators in games (e.g. a red translucent flash when the player is hurt) or mood filters on a website banner (e.g. an orange overlay for a warm tone). These are simply semi-transparent colored `<div>`s or CSS pseudo-elements positioned over the content. For instance, a full-screen `<div style="background: rgba(255,0,0,0.5); pointer-events:none;">` will tint the screen red without blocking interaction. You can trigger its appearance via JS/CSS (like a brief flash with CSS animations for a "take damage" effect). **Blur or focus overlays** can also be achieved as a filter on the content (CSS `filter: blur(5px)` on the underlying element) or by overlaying a blurred clone of the content for a stylized depth-of-field effect.

HUD & Camera Overlays: These refer to graphic elements placed on the screen for stylistic or UI purposes, often in games. Examples include *lens flares* (bright glares from light sources) or *dirt on camera lens* (spots, scratches) in first-person games, and heads-up display (*HUD*) elements like crosshairs, scopes, and targeting reticles. Implementing these is usually as simple as overlaying PNG images with transparency. For instance, a crosshair can be a PNG with a transparent center, absolutely positioned at the center of the screen (and `pointer-events: none` so it doesn't interfere). A scope or binocular overlay might be a semi-opaque vignette with a circular cutout. **Lens flare** effects can be static images (sun glare with additive blending) or dynamic (animated streaks and orbs) – advanced flares might use WebGL for particle-based glow, but a basic approach is to use an image of a flare and perhaps animate its opacity or rotation slightly. Because Weebly allows custom HTML/CSS, you can simply upload these overlay images to Weebly's asset folder and then place an `<img>` or a CSS `background-image` in an absolute positioned container. For example:

```
<div style="position: relative;">
  <img src="lensflare.png" style="position:absolute; top:50%; left:50%;
transform: translate(-50%, -50%); pointer-events:none;" alt="flare">
  <img src="crosshair.png" style="position:absolute; top:50%; left:50%;
transform: translate(-50%, -50%); pointer-events:none;" alt="crosshair">
  <canvas id="gameCanvas">…game…</canvas>
</div>
```

In the snippet above, the lens flare and crosshair images are stacked on top of a hypothetical game canvas. They are both centered via CSS transforms and have `pointer-events: none` so that mouse clicks pass through to the game canvas. HUD overlays for websites (outside of games) are less common, but one might add decorative frames or an on-screen watermark in a similar fashion. The key is that these overlays use standard HTML elements (images, SVGs, or canvases) absolutely positioned within a container.

Techniques for Implementation

Achieving the above effects involves several core web technologies. Often the same effect can be implemented via multiple techniques – for example, falling snow could be done with pure CSS animations, with a canvas particle system, or even WebGL. Here we compare the main techniques:

- **CSS3 Animations and Filters:** Many simple overlay effects can be created with just HTML elements and CSS. CSS **keyframe animations** or transitions can move, rotate, or fade elements (e.g. animating a `<div>` to shake for a camera vibration effect). CSS is particularly convenient for overlays that consist of repeated elements (like multiple snowflake `<span>`s falling) or for applying visual filters to images/videos. Modern CSS filters include `blur()`, `brightness()`, `contrast()`, `hue-rotate()`, etc., which can produce effects like night-vision color shifts or blurred backgrounds without any custom pixel manipulation. For instance, one could apply `filter: grayscale(1) hue-rotate(90deg) brightness(1.5)` to an image to approximate a greenish night vision look. CSS **blend modes** (`mix-blend-mode` and `background-blend-mode`) allow overlay layers to interact in complex ways (adding, multiplying colors, etc.), which is how some purely-CSS image effects libraries achieve their results [17] [18]. The advantage of CSS is that it's hardware-accelerated for transforms and many filters, and it requires no external libraries. However, CSS animations work at the element level – if you need per-pixel control or thousands of particles, CSS alone can become unwieldy or less performant. Also, certain advanced filters (e.g. SVG filters or clip-path animations) might not work in older browsers. As a progressive enhancement though, CSS techniques are powerful and easy to integrate into Weebly (just drop the styles in the page or site header).

- **HTML5 Canvas (2D Context):** The `<canvas>` element with the 2D drawing context (Canvas API) is a cornerstone for custom graphics. Canvas allows you to draw shapes, text, images, and even manipulate pixels, all with JavaScript. This is ideal for many overlay effects:

- You can implement particle systems (snow, rain, sparks) by animating and drawing many small shapes each frame.
- Canvas lets you draw semi-transparent shapes and images, enabling effects like smoke (drawing many translucent circles that fade out) or fire (drawing particles with `globalCompositeOperation = "lighter"` to add their colors) [6].
- It also allows procedural visuals like generating a noise pattern for fog – e.g., by setting a low global alpha and drawing a noise image or algorithmic noise across the canvas [4].
- The Canvas API is imperative (you manually redraw the scene each frame), which gives a lot of control. For example, to do rain without a library, you could use canvas to draw blue lines (raindrops) falling and puddle ripples; to do a glitch, you could copy fragments of an image and redraw them offset on the canvas.

The performance of canvas is generally good for a moderate number of objects (hundreds to a few thousand) but will degrade if you attempt tens of thousands of drawing ops or heavy pixel-by-pixel computations in pure JS. For most overlay needs (like a few hundred snowflakes), canvas can easily sustain 60 FPS on modern browsers. It's supported on all modern devices (including mobile). One thing to note is that canvas content is not DOM, so you can't apply CSS filters to what's drawn inside a canvas (but you *can* apply filters to the canvas element as a whole). If you need to combine canvas with DOM (for example, drawing an effect over a video element), you might overlap a canvas on top of the video in the layout.

Remember to make the canvas transparent (the default background is transparent if nothing is drawn) or use `clearRect` each frame to only draw moving particles without clearing the background. You should also size the canvas appropriately to cover the area (and update its size on window resize for responsiveness). We'll see examples of canvas usage (like the RainyDay library) in the code section. Canvas code can be included in Weebly by placing the `<canvas>` tag and associated `<script>` inside an Embed Code block (see Weebly embedding tips below).

- **WebGL and Fragment Shaders:** For the most advanced or performance-intensive effects, WebGL is the tool of choice. WebGL is the web's interface to GPU-accelerated 3D (and 2D) graphics, allowing use of **shaders** – programs that run on the GPU to render pixels. With WebGL, you can create effects that would be too slow in pure JavaScript, like real-time image distortions, complex particle systems with thousands of particles, or fullscreen filters (blur, bloom, CRT distortion, etc.) at 60 FPS. A fragment shader can manipulate each pixel of an image or canvas on the GPU in parallel, which is how one can achieve effects like the heat haze distortion in real time [15]. For example, a fragment shader could be written to sample an underlying texture (say the current frame of a game or a background image) and offset the texture coordinates based on a noise function, yielding a wavy distortion – something difficult to do per-pixel in JS but trivial for a shader. Using WebGL directly is complex, but there are libraries that simplify it. **Three.js** is a popular 3D library that also supports post-processing effects and shaders, and **PixiJS** is a 2D WebGL library great for rendering particles and filters in a simpler way [19]. In fact, PixiJS has built-in filter effects and a particle emitter system – you could use Pixi to easily create, say, a firework particle overlay or apply a CRT filter to a sprite. Pixi's `ParticleContainer` is highly optimized for many sprites [20], allowing "millions of particles" with good performance by batching draws. WebGL can also be used for offscreen rendering: for instance, capturing the scene into a texture and applying a shader (common in game engines to do full-screen effects like night vision or thermal vision). The downside of WebGL is the added complexity and larger libraries; however, for demanding overlays (like a full-screen shader effect), it's often the only way to achieve the effect smoothly. When using WebGL in Weebly, you'd include the library (Three.js, Pixi.js, etc.) via script tags and then run your shader or particle code in the embed. Keep in mind compatibility: WebGL is supported by all modern browsers, but very old devices or IE11 might not support it (IE11 supports a legacy subset WebGL1). Also, always test on mobile – complex shaders can drain battery or run slowly on low-end phones, so you may want to provide a fallback or toggle for mobile users.

- **SVG Filters and Blend Modes:** SVG offers a powerful filter system that can be applied to HTML content via CSS. SVG filters like **<feTurbulence>** (to generate noise) and **<feDisplacementMap>** (to distort images) can produce overlay effects without external images. For instance, you can define an SVG filter that generates fractal noise and use it to fill a `<rect>` as a translucent layer, achieving a grainy overlay [21]. This technique was demonstrated in a blog where an inline SVG with `<feTurbulence type="fractalNoise">` was used to create a noise texture for backgrounds [21]. The SVG was embedded as a data URI and applied with `background-image`, combined with `mix-blend-mode: soft-light` to overlay subtle grain. Similarly, an SVG filter could produce scanline effects or blur combined with color shifts. The advantage is you don't need image assets for the texture; the browser generates it. The downside is that SVG filters can be heavy on performance if applied to large areas continuously (they are not GPU-accelerated in all browsers). They also have spotty support on very old browsers. But for small, static overlays (like a one-time noise texture) or moderate animations, they can be useful. CSS `mix-blend-mode` (as mentioned earlier) is another tool to combine overlay layers in creative ways (e.g. using *screen* blend to simulate additive light for lens flares, or *multiply* for darkening shadows). One implementation trick is using pseudo-elements (`::before` / `::after`) on an element to add overlay layers that blend with the element's content below [22]. This avoids extra markup and can be triggered on hover or other interactions for cool effects (like the CSS-only glitch example that used layered images and blend modes). In summary, SVG filters and blend modes are more declarative ways to get certain overlay looks and can often be combined with CSS animations for dynamic behavior.

- **DOM Element Overlays vs. Canvas Overlays:** A design decision when implementing overlays is whether to use HTML elements (DOM) for the effect or draw the effect on a canvas. **DOM overlays** (e.g. using many `<div>` or `<img>` elements) are easy to style with CSS and to position in a responsive layout. For example, if you want 50 snowflakes falling, you could create 50 `<div class="snowflake">❄</div>` and use CSS keyframes to animate their `top` and perhaps a side-to-side drift. This has the benefit of being straightforward and allows each snowflake to be an element you could style or even make interactive. However, too many DOM nodes or very frequent DOM updates can hurt performance; animating via CSS is efficient, but if you try to manually move elements via JS every frame, it's slower than doing it in a single canvas. **Canvas overlays** condense all the effect rendering into one element (the canvas) and leverage the 2D graphics API, which can handle a lot of drawing operations efficiently in a single bitmap. Canvas is better suited for particle effects where elements are numerous or short-lived (sparks, raindrop splashes) because you don't pay the cost of maintaining a DOM node for each particle. Also, some effects (like drawing a complex perlin noise pattern, or applying a distortion) are much easier in canvas where you can set pixel colors or use image processing techniques. The trade-off is you need to manage the drawing logic in JS and ensure you clear/redraw properly. In some cases, a **hybrid** approach can work: use CSS for what it's good at (e.g. simple animations of a few elements) and canvas for heavy lifting. A good example is a game: you might have the game world drawn on a canvas, but use DOM for UI overlays (scores, HUD text) and maybe a DOM-based damage flash (since a simple red overlay might be easier as a `<div>`). You can layer a canvas above or below DOM elements by using CSS `position` and `z-index`. It's worth noting that an HTML `<video>` element can be treated like a DOM element for overlays (e.g. you can overlay captions or effects on top of a video by simply stacking HTML elements in a container). The **pointer-events** CSS property is very helpful when layering overlays: by setting `pointer-events: none` on overlay elements, you ensure they don't block mouse/touch inputs to underlying elements [23]. This is crucial for game overlays – for example, if you put an HTML element on top of a canvas game to show static or blood splatter, you'll want `pointer-events: none` on that overlay so that clicks still register on the game canvas underneath. In summary, use DOM/CSS overlays for simpler or UI-focused effects (or when leveraging CSS filters), and use canvas/WebGL overlays when you need a custom drawn effect or lots of moving pieces.

Asset Strategies for Overlay Effects

Implementing visual effects often requires assets – images, sprite sheets, or code libraries. Here we discuss how to manage assets for overlays:

- **Spritesheets & Transparent PNGs:** A spritesheet is a single image file containing a grid of animation frames. It's a classic game dev technique to animate something by quickly cycling through frames. For overlays, you might use spritesheets for things like an *explosion sequence*, a *fire animation*, or even an animated *raindrop splat*. In a web context, you can animate spritesheets with

CSS (using `@keyframes` and the `steps()` timing function to jump between frame positions) [24] or via canvas (drawing a portion of the sheet per frame). For example, if you have a 8-frame fire animation on a horizontal sprite strip, you could use CSS like `background-position: -Npx 0` changing over time combined with `animation-timing-function: steps(8)` to display it frame by frame [24]. Spritesheets reduce HTTP requests (multiple frames in one file) and keep frames perfectly aligned. Weebly allows uploading images to the site (either via the editor or the theme assets), so you can upload your sprite PNG and reference it in CSS or draw it with canvas. Transparent PNGs are essential for overlays – e.g. a PNG of a smoke puff or a lens flare with alpha transparency can be drawn or overlaid without obscuring the scene behind. You might have a set of PNG textures for particles (like different shaped snowflakes or embers) which a particle system can randomly use. When adding these to Weebly, you can put them in the **Theme > Assets** folder (via Weebly's code editor) or host them externally on a CDN and link by URL. Remember to use appropriate resolutions; for instance, a full-screen fog image might need to be large to avoid pixelation on big screens, but large images can impact load performance – consider tiling smaller seamless textures.

- **Particle Libraries and Engines:** There are many JS libraries focused on particle effects and animations that can simplify overlay implementation. **Particles.js** (and its modern variant tsparticles) is a lightweight library for creating customizable particle backgrounds like bubbles, confetti, snow, etc., by just including a script and a JSON config [25]. It can be integrated in Weebly by including the library script (from CDN or uploaded file) and initializing it on a `<canvas>` or `<div>`. Using a library saves you from writing physics and animation loops for common effects. **PixiJS** was mentioned earlier – it's essentially a rendering engine that can handle sprites and has add-ons for particles. For example, Pixi can produce a fountain of particles with various properties like velocity, acceleration, lifespan, using a texture for each particle (e.g. a small glow image for sparks). It offloads much of this to the GPU and is highly performant. **GSAP (GreenSock Animation Platform)** is another tool: while not specialized for particles, GSAP can animate large numbers of DOM elements efficiently and has utilities for randomization, bursts, and timeline control. For instance, GSAP could animate 100 `.snowflake` elements with staggered delays and looping y-translations, all with smooth performance. GSAP can also control canvas-based animations if you integrate it (or control CSS filters to create a glitch flicker timing, etc.). **Anime.js** is a similar animation library that can animate CSS, DOM, or even numeric JS properties – one could use it to animate parameters of a canvas animation (like oscillating an intensity value for an effect). For interactive games, sometimes using a full game engine (like Phaser or Unity WebGL) is an option, but within Weebly you're more likely to stick to self-contained overlays. Choose a library if it significantly reduces complexity (e.g., include Particles.js to get a snowfall with 20 lines of config, instead of writing it from scratch). Just be mindful of file size and load time: adding multiple large libraries for small effects might not be wise. Pixi.js is powerful but around ~500KB; a simpler library might be only 50KB. Weebly embed allows external scripts, so you could use CDN links for these (which might be cached across sites). Be sure to initialize the library in the Weebly Embed Code after including it. For example, after adding a `<script src="https://cdn.jsdelivr.net/particles.js/2.0.0/particles.min.js"></script>`, you would include a script block with something like `particlesJS.load('background-div', configObject);`. We'll show code using a library (RainyDay) in the examples section.

- **Procedural Generation & Noise Textures:** Some effects require randomness or natural patterns, which can be achieved through procedural algorithms. **Perlin noise** (and its faster variant Simplex noise) is commonly used to generate natural-looking randomness – useful for fog, fire, clouds, etc. In a game engine, you might generate a noise field and use it to move particles or distort an image. In JavaScript, you can include a noise function (several implementations are available) and then use it in a canvas pixel loop or to move elements around. For example, to create a dynamic fog, one approach is: generate a 2D noise grid (Perlin noise) and draw it to a canvas as semi-transparent gray pixels, then shift the noise slightly each frame (to look like moving fog) [4]. Another approach is using noise to modulate the opacity of fog sprites or the displacement of a background image (with a displacement filter). Procedural generation also encompasses simpler random distributions – e.g., randomly generate positions for 100 stars in a starfield overlay, or randomize the timing of a flicker effect. **Random seed control** is important if you want reproducible patterns (for consistent effect) versus fresh randomness each load. If procedural math is too complex to implement live, a shortcut is to use **pre-made textures**: e.g., use an image of noise (as we did for static) or a cloud texture for fog. You can even generate such an image in an art program or find free resources (some developers use Photoshop filters or Perlin noise generators to make tiling noise textures). These images can then be animated (scrolled, rotated, faded) to create the illusion of dynamic effect without realtime computation. For instance, a single translucent smoke PNG, if slowly scaled up and faded out, can mimic a puff of smoke. Many libraries or code examples use sprite images for particles (like a single spark graphic drawn hundreds of times). When embedding in Weebly, if you rely on external assets (images or JSON config), you should either host them on your site (upload to Weebly and use the file URL) or on a reliable CDN.

In summary, plan your asset strategy by deciding whether you can use existing libraries or need custom code, and gather any image assets (noise textures, overlays, sprite frames) needed. Always optimize images (e.g. compress PNGs) and only load libraries you actually use. Weebly's editor lets you manage files under **Settings > SEO > Footer Code** or the theme editor, where you can add files to the **Assets** folder and reference them by path (e.g. `/files/theme/yourfile.png` or the given Weebly file URL).

Embedding & Performance Considerations in Weebly

Embedding custom code on Weebly requires using the Embed Code element or the site's Code Editor for theme modifications. Here are strategies specific to Weebly:

- **Using the Embed Code Element:** Weebly provides a drag-and-drop **Embed Code** block where you can insert HTML, CSS, and JS. This is the simplest way to add an overlay effect to a specific page or section. You can write your custom HTML markup (like a `<canvas>` or `<div class="overlay">` structure) and inline CSS/JS right there. For instance, to add a canvas with an effect, drag an Embed Code to the page, click to edit HTML, and then paste something like: `<canvas id="effectCanvas" width="800" height="600"></canvas><script>/* JS code here */</script>`. Weebly will include that in the page output. According to Weebly experts, you can indeed include `<style>` and `<script>` tags inside the embed block along with your HTML [26]. It's often recommended to put large CSS in the site's header or theme stylesheet for reuse, but for one-off effects, a `<style>` tag in the embed code is fine. One tip is to place your `<script>` *after* the HTML in the embed (so the elements exist when the script runs) [27]. Alternatively, you can add your JS in the Footer Code (in site settings) so it runs after page load. Weebly might strip certain tags in the editor preview, but in published mode it should work. Make sure not to conflict with existing site scripts; use unique IDs and perhaps wrap your JS in a `window.onload` or DOMContentLoaded handler if needed to ensure the page is ready.

- **Hosting and Linking Assets in Weebly:** If your overlay uses external files (images, scripts, CSS), you have a few options:

- *Weebly Asset Upload:* Weebly allows adding files via the Theme Code Editor (accessible under Theme > Edit HTML/CSS). There is an **Assets** folder where you can hit "+" to upload files (JS, CSS, images) [28]. Uploaded files get a path like `/files/theme/filename.ext`. You can reference them in your code. For example, if you uploaded `rainyday.js`, you could add `<script src="/files/theme/rainyday.js"></script>` in your Embed Code or in the theme's HTML (perhaps in the footer) [29]. This way, the file is served from your site.
- *CDN/External Links:* For libraries like GSAP, Pixi, or others available on CDNs, you can link directly via their CDN URL in a `<script src="..."></script>`. This is often easier and ensures you get the latest or a cached copy. Weebly supports external scripts unless they violate content security, which is rare for common libraries.
- *Inline Code:* Small scripts or CSS can just be embedded inline in a `<script>` or `<style>` tag. Weebly's embed block will keep them. As a rule, test after publishing because sometimes the editor preview might not execute scripts for safety.

If you need to include CSS site-wide (for example, CSS for an overlay that appears on every page, like a background static effect), you can go to **Settings > SEO > Header Code** or **Footer Code** in Weebly and put a `<style>...</style>` block there. That will inject into every page. Similarly, site-wide JS can go in Footer Code (which is placed just above `</body>` of every page). However, exercise caution with site-wide includes; only use them if the overlay is needed on all pages, otherwise it's extra load where not needed.

- **Performance Best Practices:** Overlay effects can be graphics-intensive, so consider performance especially on mobile devices:
- **Optimize frame rate and loops:** Use `requestAnimationFrame` for any JS animation loops (instead of `setInterval`), as it's more efficient and syncs to screen refresh. For example, if you create a canvas animation, structure it as `function animate(){ ...update and draw... requestAnimationFrame(animate); }` rather than a fixed timer, for smoother results [30] [31]. This also pauses when the tab is not active, saving CPU.
- **Limit particle counts and sizes:** If using a particle system, find the sweet spot for number of particles. E.g., 100 snowflakes might look good; 1000 might start to lag on older devices. Many libraries let you configure this. Start with moderate values and test. Also consider capping effects on mobile – you could use CSS media queries or JS to detect screen width and reduce effect density (for example, fewer snowflakes or turning off a heavy shader on phones).
- **Offload heavy computations:** If you are doing expensive calculations (like generating noise arrays or complex image data), do it once and reuse if possible. For instance, generate one noise canvas and reuse it as a texture rather than recalculating every frame. For continuous effects, keep calculations simple each frame (e.g., update positions with basic arithmetic rather than running nested loops over big arrays). Web Workers could be used for heavy computations off the main thread, but this is advanced and usually not needed for typical overlays.
- **Use GPU where possible:** Leverage CSS transforms (which are GPU-accelerated) instead of top/left CSS changes for smooth animation. For example, for a shake effect, toggling a CSS class that applies `transform: translate()` with a CSS animation will be smoother than using JS to change element.style.left in a loop. Similarly, CSS `filter` can be GPU-accelerated (browsers often accelerate blur, etc.).

- **Responsive design:** Ensure your overlay scales or positions correctly on different screen sizes. If you use a canvas, you might set its width/height via JS to `window.innerWidth` and `window.innerHeight` (or the containing element's size) so it covers the area. You may need to redraw or adjust on resize events. If using CSS overlays, consider using `vw`/`vh` units or percentages so they stretch appropriately. For example, an overlay `<div class="fog">` with `width:100%; height:100%` inside a position:relative container will automatically fill that container regardless of device size.

- **Testing on devices:** Because Weebly sites are often accessed on mobile, test your effect on an actual phone or tablet. Check for frame rate (does it feel choppy?), touch responsiveness (does the overlay interfere with scrolling or clicking? It shouldn't if pointer-events are none), and memory (does the site become slow or crash after a while?). If an effect is too heavy, you might implement a toggle (e.g., a button to turn off animations) or automatically disable it on very small screens. Some websites do this by checking `navigator.userAgent` for older devices or using `matchMedia` to see if `prefers-reduced-motion` is set – for example, a user who prefers reduced motion might be served a static background instead of an animated one.

- **Weebly Specific Considerations:** Weebly sites load jQuery by default, which you can utilize if needed (for example, to easily select elements or on `$(document).ready` events). However, for performance-critical animations, it's better to use raw JavaScript or requestAnimationFrame than jQuery's `animate`, as the latter is not meant for high-frequency frame updates (jQuery is fine for simple fades or slides though). If adding multiple embed code sections, be careful about scope – a script in one embed block won't directly "see" elements in another if they are separated (once published it's one DOM, but if you add in editor, ensure the script comes *after* the corresponding HTML). Often it's easiest to put your HTML and its script together in one embed. If you have styles that should apply site-wide, put them in the theme or main CSS instead of each embed block.

Finally, note that Weebly's environment might sanitize some code in the editor preview. If you encounter issues (like your script tags disappearing in the editor), a workaround is to put the JS in the Footer Code or reference an external file. Once published, the code should execute. The Weebly Community and documentation confirm that adding custom HTML5 animations is possible via the embed element and asset uploads [32] [29] – many users have embedded Canvas and even WebGL projects successfully. Just remember that if you ever change your Weebly theme or use their site editor's design options, it might override some CSS, so use specific selectors or add `!important` on critical overlay styles if needed.

Cross-Browser and Mobile Support: In 2025, all modern browsers (Chrome, Firefox, Safari, Edge) support the HTML5/CSS3 features discussed. Internet Explorer (now deprecated) is the main outlier – IE11, for instance, does not support CSS `mix-blend-mode` or `clip-path` on HTML elements, and had limited WebGL. It's likely not a significant portion of users now. But if you do need to support older browsers, consider providing graceful degradation: e.g., a static image fallback if a CSS effect isn't supported (the Bennett Feely image-effects library uses `@supports` to only apply the fancy effects if supported, otherwise the image just displays normally [33]). On mobile, iOS and Android browsers are generally good with these standards. One thing to watch: CSS fixed backgrounds and some filters had performance issues on older mobile devices. Test any heavy effect on iOS Safari – if it's sluggish, you might tone it down or disable it for that UA. Also, very high-resolution canvases (like full HD on a phone) can be memory-intensive – you can consider lowering canvas resolution a bit on mobile by scaling it via CSS (draw at half resolution but stretch it, if the slight quality loss is acceptable). Always ensure overlays do not make the site unusable; provide a balance between visual appeal and performance.

Practical Examples and Code Snippets

Let's showcase a few key overlay effects with example code, demonstrating how one could implement them in a Weebly embed. These examples assume you add them via an Embed Code element in Weebly, but they can be adapted elsewhere. (For brevity, we use simple implementations – in practice, you might refine sizes, event handling, etc.)

Figure: Example of a raindrop overlay effect simulated with RainyDay.js (raindrops appear on the image, with some blurring and dripping).

1. Rain Overlay (Canvas + Library): We'll use the RainyDay.js library to create a realistic rain-on-glass effect over an image. First, upload the `rainyday.js` script to Weebly (or use the CDN) and have an image ready (the image will be the backdrop seen through the rain). In an Embed Code block, we could put:

```
<div style="position: relative; overflow:hidden;">
  <!-- Background image over which rain will appear: -->
  <img id="rain-bg" src="YOUR_IMAGE_URL.jpg" style="width:100%; display:block;">
  <!-- Canvas for rain overlay: -->
  <canvas id="rain-canvas" style="position:absolute; top:0; left:0;"></canvas>
</div>

<script src="/files/theme/rainyday.js"></script>  <!-- include the RainyDay
library -->
<script>
  function startRain() {
    const img = document.getElementById('rain-bg');
    const canvas = document.getElementById('rain-canvas');
    // Size the canvas to match the image dimensions:
    canvas.width = img.clientWidth;
    canvas.height = img.clientHeight;
    // Initialize RainyDay engine with the image as background
    const engine = new RainyDay({ image: img, canvas: canvas });
    // Start rain with drop size and count presets:
    engine.rain([[3, 2, 0.5]], 100);
    // The array [3,2,0.5] and 100 means small drops of radius ~3, trail length
2, and a rain intensity of 100.
  }
  // Start after image loads:
  document.getElementById('rain-bg').onload = startRain;
</script>
```

In this snippet, we place an image and a canvas in a container. The canvas is absolutely positioned over the image (covering it). When the image loads, we set the canvas size and initialize RainyDay. The `engine.rain()` call takes an array of presets (drop size configurations) and a droplets count (or intensity) [34]. For more complex rain, RainyDay supports different presets for big/small drops and even collision effects. The result is water droplets falling and sliding on the screen, with the background image visible beneath them, creating a rain-on-window illusion. This kind of effect is great for a game scene (e.g. a driving game with raindrops on the "camera") or a website header to add dynamic weather. If you want just the overlay on top of site content, you could even set the background image to transparent or to the page snapshot, but typically you use it with a background image. RainyDay uses canvas for rendering and will utilize some blur effect (it can simulate the refraction of droplets). It's optimized and runs well on modern browsers, though having many droplets can be CPU-intensive – you might lower the count or use simpler droplets for mobile.

Note: Ensure the container has `overflow:hidden` if the canvas might draw beyond image bounds. The code above uses the image's natural size; if you want full-screen rain, you could use a full-window image or set canvas width/height to `window.innerWidth/innerHeight`. In Weebly, if using in a section background, you might adapt accordingly.

2. Static TV Noise Overlay (CSS): To create a retro TV static effect over a page, we can use a CSS-only approach with a noise image. Suppose we have a small PNG (e.g. 20x20px) that is a random black-and-white noise pattern (you can generate one or find a free noise texture). We'll tile that across an overlay and animate it. We can do this entirely in CSS, and just need to insert a div for the overlay. For example, in an Embed Code block:

```
<div class="static-noise"></div>

<style>
.static-noise {
  pointer-events: none;
  position: fixed; top: 0; left: 0;
  width: 100vw; height: 100vh;
  background: url('noise-small.png') repeat;
  opacity: 0.2;
  animation: tv-flicker 0.15s steps(2) infinite;
  z-index: 9999;  /* ensure it's on top of everything */
}
@keyframes tv-flicker {
  50% { opacity: 0.3; transform: translateY(-1px); }
}
</style>
```

This will cover the whole viewport with a noise texture. The `steps(2)` animation toggles the opacity between 0.2 and 0.3 rapidly, which makes the static pattern flicker. I also added a tiny `translateY(-1px)` at 50% to jiggle it. The effect is a subtle, constantly changing static. The overlay is fixed position, so it stays even if the page scrolls (like an old TV screen overlaying the content). Because we set `pointer-events: none`, all page links and inputs remain clickable through it. You can adjust the opacity for heavier or lighter static, and animation speed if you want a faster flicker. If you prefer, you could animate `background-position` randomly to shift the noise tile instead of opacity. Another approach is using an animated GIF for noise – simply set it as the background image and it will play. But a CSS animation might be more efficient and allows control of intensity. This static overlay is great for a vintage or hacker aesthetic on a website, or could be toggled on in a game as a "lost signal" effect.

Browser support for this is broad (even older browsers will just see a static overlay, the only possibly unsupported part is `pointer-events` in very old IE, but that's not an issue today). On mobile, fixed positioning might not cover address bars etc., but since Weebly sites are mostly standard, it should be fine.

3. Glitch Effect on Text (CSS): For a quick text glitch, CSS can do a lot. There are many ways; here we'll duplicate text in pseudo-elements and animate cuts. Consider a title that we want to glitch. We can write HTML: `<h1 class="glitch-text" data-text="GLITCHY">GLITCHY</h1>`. We use the `data-text` attribute to hold the same text. Then CSS:

```
.glitch-text {
  position: relative;
  color: #fff;
  font-size: 3rem;
}
.glitch-text::before, .glitch-text::after {
  content: attr(data-text);
  position: absolute; left: 0; top: 0;
  /* mimic broken RGB channels */
  color: #f00;
  overflow: hidden;
}
.glitch-text::after {
  color: #0ff;
}
@keyframes glitchTop {
  0% { clip: rect(0, 0, 0, 0); }
  10% { clip: rect(10px, 999px, 40px, 0); transform: translate(-2px, -2px); }
  20% { clip: rect(5px, 999px, 15px, 0); transform: translate(2px, 0); }
  30% { clip: rect(0, 0, 0, 0); transform: translate(0); }
}
@keyframes glitchBot {
  0% { clip: rect(0, 0, 0, 0); }
  10% { clip: rect(60px, 999px, 90px, 0); transform: translate(0, 2px); }
  20% { clip: rect(40px, 999px, 80px, 0); transform: translate(-1px, 0); }
  30% { clip: rect(0, 0, 0, 0); transform: translate(0); }
}
.glitch-text::before {
  animation: glitchTop 2s infinite linear alternate-reverse;
}
.glitch-text::after {
  animation: glitchBot 2.5s infinite linear alternate-reverse;
}
```

This is a bit code-heavy, but what it does is: it creates two extra layers of the text (`::before` and `::after`), one tinted red and one cyan. Using the CSS `clip` property (rectangular clipping), each layer is periodically constrained to certain slices of the text (the `rect(top, right, bottom, left)` coordinates). We animate these clips and slight translations, causing slices of the text to offset at different times [10]. The main text remains white and stable, while the pseudo-elements give that glitchy offset look in red/cyan. The result is a multi-color glitch effect where bits of the text jump. The `overflow: hidden` on the pseudo-element ensures only the clipped slice shows. We alternate the animation direction to make it unpredictable. This technique is adapted from known glitch examples (like the 404 conference example that inspired a Codrops tutorial [35]). You can adjust the `clip` values (which are pixel values for slice positions) depending on font size (here I guessed some). In practice you'd tweak to look good. Even if one doesn't fully understand the code, it can be reused as a chunk. In Weebly, you'd put that CSS in an Embed Code (or site CSS) and just apply `class="glitch-text"` to any text element. If the site builder doesn't allow adding `data-text` easily, you could hardcode content in `::before` via CSS (some do `content: "GLITCHY";` directly, but that's less flexible). This works best on non-wrapping text (one line). For images, a similar approach of layering and shifting with blend modes can be used (as we did earlier with the `.glitch-img::after` example). For a quick image glitch, you could even use an SVG filter (there's an SVG filters combination that offsets color channels and adds noise). But CSS+pseudo-elements is often sufficient. One more tip: adding a tiny CSS `text-shadow` with an inverted color at a different offset can give a subtle ghosting too. Experimentation is key with glitch effects.

4. Fog/Mist Overlay (CSS Image or Canvas): To create a creeping fog effect, we can overlay a semi-transparent fog texture and move it slowly. We briefly mentioned this in techniques, now here's a concrete example using CSS. Suppose you have a PNG image of fog (e.g. a gray-white cloudy form with transparency). You want it to continuously drift. You can tile it or use one big image. For seamless movement, tiling a smaller seamless noise is easiest. We'll use CSS animations:

```
<div class="fog-layer"></div>

<style>
.fog-layer {
  pointer-events: none;
  position: absolute; top: 0; left: 0;
  width: 100%; height: 100%;
  background: url('fog.png') repeat;
  opacity: 0.5;
  animation: drift 30s linear infinite;
}
@keyframes drift {
  from { background-position: 0px 0px; }
  to   { background-position: 500px 500px; }
}
</style>
```

Place that `.fog-layer` as a child of a container that has `position: relative` (for example, wrap it over a section or image). The CSS above will cause the fog texture to move diagonally over 30 seconds, then loop. With 50% opacity, it creates a light mist. You can stack multiple fog layers with different speeds/

opacities to add depth (e.g., one moves faster (small, nearby mist) and one slower (far background mist)). This technique is very performance-friendly since it's just moving a background image with the GPU. No JavaScript needed, and it works on mobile (though on older phones 30s translate might get chunky if the image is huge – but usually it's fine). The `pointer-events: none` ensures any underlying links can be clicked. If you want the fog to cover the entire screen, you might put it in a fixed position container. In Weebly, you could overlay it on a banner image or in a full-screen header.

An alternative approach: use canvas to generate fog via noise. For example, using the Canvas API, you could procedural draw semi-transparent circles or perlin noise every frame with slight shifts. But that can be a bit heavy and frankly the CSS approach often looks just as good. If you wanted interactive fog (like reacting to mouse, e.g., clearing where the mouse moves to simulate a "fog of war"), then canvas or SVG would be needed. But for simple atmospheric mist, the moving background image trick is a go-to.

5. Combining Multiple Effects: Often, you might want to use several overlays together. This is possible – just ensure you handle layering via z-index and container structure. For example, you might have a game with a *rain* overlay and also want a *HUD* overlay (like crosshair). You can simply have two canvas or div overlays on top of the game canvas. E.g.:

```
<div id="game-container" style="position: relative;">
  <canvas id="gameCanvas">…</canvas>
  <canvas id="rainCanvas" style="position:absolute; top:0; left:0;"></canvas>
  <img src="crosshair.png" id="crosshair" style="position:absolute; top:50%;
left:50%; transform:translate(-50%,-50%);">
</div>
```

This uses one container, stacking the elements. The rainCanvas would be sized same as gameCanvas and drawn on (with pointer-events: none), the crosshair sits on top centered. This demonstrates that overlays are stackable – just manage z-index if using different container siblings. In this case, since all are in one parent, later elements naturally overlay earlier ones (unless styled otherwise). If you run into any interactivity issues (e.g., clicks not passing), double-check that `pointer-events` is off for purely visual layers.

6. Code Integration Example for Weebly: Let's finalize with an example scenario relevant to Weebly website enhancement: say you want an interactive hero section background with floating particles (like a subtle constellation of dots that connect on hover). Instead of coding from scratch, you could use **Particles.js**. Here's how one might embed that:

- Include the Particles.js script.
- Add a `<div id="hero-particles"></div>` in the hero section.
- Call the library to load a configuration for dot particles.

Minimal example:

```
<div id="hero-particles" style="position: absolute; width:100%; height:100%;"></
div>
```

```
<script src="https://cdn.jsdelivr.net/npm/particles.js@2.0.0/
particles.min.js"></script>
<script>
  /* Configuration for particles.js */
  particlesJS('hero-particles', {
    "particles": {
      "number": { "value": 50 },
      "color": { "value": "#ffffff" },
      "shape": { "type": "circle" },
      "opacity": { "value": 0.5 },
      "size": { "value": 3 },
      "line_linked": { "enable": true, "distance": 120, "color": "#ffffff",
"opacity": 0.4, "width": 1 },
      "move": { "speed": 1 }
    },
    "interactivity": {
      "events": { "onhover": { "enable": true, "mode": "grab" } }
    },
    "retina_detect": true
  });
</script>
```

This would create 50 white dots that float and link with lines when hovered (the grab mode). The library handles the canvas creation in the `hero-particles` div. Such an effect is visually appealing and achieved with a few lines of config. Just ensure the container (`hero-particles`) covers the area you want (you might need to set parent to position: relative and this div to absolute as shown). Weebly's builder may require some adjustments to insert such custom divs in a section (possibly using an Embed Code element that spans the full width of the section).

Through these examples, you can see the pattern: **include necessary assets** (scripts, images), **create the overlay element(s)** (canvas or div), **apply styling** (CSS or via JS), and **initialize the effect** (either manually drawing, or calling library functions). All of this is possible within Weebly's framework by using the Embed Code element and the Theme Editor for file uploads. Users have successfully added HTML5 animations to Weebly pages this way [32], and Weebly's flexibility with custom code is one of its strengths for advanced users.

Conclusion

HTML5 overlay effects can greatly enhance both game experiences and general websites by adding dynamic visuals—from pouring rain and drifting snow to glitching screens and simulated HUDs. On Weebly, you can implement these effects by embedding custom HTML/CSS/JS, leveraging techniques like CSS animations for simplicity or canvas/WebGL for more complex visuals. We've covered the gamut of effect types (weather, distortions, filters, HUDs) and shown that often multiple approaches exist (e.g. CSS vs canvas) each with pros and cons. Implementation involves careful layering (using absolute positioning and pointer-events settings), performance considerations (using GPU-accelerated features and controlling particle counts/frame rates), and sometimes using external libraries to speed up development. By following best practices – such as using Weebly's asset manager for files, placing scripts appropriately, and optimizing for mobile – you can achieve sophisticated overlay effects even on a Weebly site, which traditionally is a no-code builder but becomes very powerful with a bit of custom code.

As you experiment, refer to libraries' documentation and existing tutorials for specific effects (many open-source demos exist for things like canvas rain or CSS glitches). Always test your site after adding an effect to ensure it doesn't hamper usability. With the right effect, you can create a more engaging atmosphere: imagine a Weebly portfolio page with a gentle snowfall on a winter photograph, or a game embedded on a Weebly site with UI overlays and post-processing effects rivaling native apps. All this is attainable with HTML5 overlays – a creative toolkit at your disposal.

Sources: The information and examples above draw on a variety of web development resources and tutorials, including the RainyDay.js documentation [1] [34], canvas animation techniques for snow and fire [3] [6], CSS glitch and static effect guides [9] [16], as well as community advice on integrating custom code into Weebly [26] [29]. These references illustrate the state-of-the-art methods for overlay effects and confirm their applicability in the Weebly environment as of 2025.

---

[1] [7] [8] Realistic Raindrops Effect with Canvas and Rainyday.js | Free jQuery Plugins
https://www.jqueryscript.net/other/Realistic-Raindrops-Effect-with-Canvas-Rainyday-js.html

[2] [34] How to create Realistic Raindrops Effect With Javascript Canvas ? - GeeksforGeeks
https://www.geeksforgeeks.org/javascript/how-to-create-realistic-raindrops-effect-with-javascript-canvas/

[3] How to Implement an Animated Snow Effect using HTML5 Canvas and Javascript
https://designers.hubspot.com/blog/how-to-implement-an-animated-snow-effect-using-html5-canvas-and-javascript

[4] javascript - how to make a fog effect or blur effect in canvas with js? - Stack Overflow
https://stackoverflow.com/questions/8938965/how-to-make-a-fog-effect-or-blur-effect-in-canvas-with-js

[5] Interactive Smoke Effect With JavaScript and Canvas - smoke.js | CSS Script
https://www.cssscript.com/interactive-smoke-effect-javascript-canvas-smoke-js/

[6] HTML5 Canvas Experiment: A cool flame/fire effect using particles
https://thecodeplayer.com/walkthrough/html5-canvas-experiment-a-cool-flame-fire-effect-using-particles

[9] [10] [35] CSS Glitch Effect | Codrops
https://tympanus.net/codrops/2017/12/21/css-glitch-effect/

[11] [12] Make some hacky noise with CSS gradients · January 22, 2024
https://nerdy.dev/hacky-css-noise-with-repeating-gradients

[13] [14] [15] [19] [30] [31] Animated Heat Distortion Effects with WebGL | Codrops
https://tympanus.net/codrops/2016/05/03/animated-heat-distortion-effects-webgl/

[16] [17] [18] [22] [33] Image Effects with CSS
https://bennettfeely.com/image-effects/

[20] ParticleContainer - The New Speed Demon in PixiJS v8
https://pixijs.com/blog/particlecontainer-v8

[21] Noisy/Grainy backgrounds and gradients in CSS
https://www.bstefanski.com/blog/noisygrainy-backgrounds-and-gradients-in-css

[23] Creating a Canvas Overlay | Matt McKenna
https://www.mtmckenna.com/posts/2018/02/02/creating-a-canvas-overlay

[24] How to Create a CSS Sprite Animation With steps() | Treehouse Blog
https://blog.teamtreehouse.com/css-sprite-sheet-animations-steps

[25] particles.js - A lightweight JavaScript library for creating particles
https://vincentgarreau.com/particles.js/

[26] [27] [32] javascript - Adding code into weebly - Stack Overflow
https://stackoverflow.com/questions/16804776/adding-code-into-weebly

[28] Customizing Your Theme | Weebly Support - US
https://www.weebly.com/app/help/us/en/topics/customizing-your-theme

[29] Adding Elements to Weebly via Javascript Not Working - Stack Overflow
https://stackoverflow.com/questions/42615262/adding-elements-to-weebly-via-javascript-not-working