ChatGPT

Dynamic 3D Fractal Torus with Neon Shaders on the Web

Example of neon-colored fractal rings, illustrating the kind of "hyper-color, neon punk" aesthetic targeted in our dynamic torus visualization.

Introduction

Creating a **dynamic 3D fractal nesting of toruses** ("donut" shapes) in a web environment is an exciting intersection of math, art, and technology. Such a project falls under *"scientific art"* – blending mathematical fractals with creative design – and aims for a **"modern multi-dimensional hyper-color, sweet fairy neon punk rock"** style. In practical terms, this means vivid neon colors, glowing materials, and fluid, almost psychedelic motion. Achieving this in the browser involves harnessing **WebGL** for 3D rendering (via libraries like *Three.js* or *Babylon.js*), writing custom **shaders** for glow and gradients, using recursive geometry for **fractal** structure, and layering in CSS/JS effects and animation libraries for polish and interactivity. The result should be an interactive visual that feels like a neon-lit fractal in motion, with multiple nested torus shapes moving or morphing in a captivating way.

In this report, we'll explore the tools and techniques needed to realize this vision. We'll compare popular **WebGL frameworks** (Three.js, Babylon.js, and raw GLSL/WebGL) and discuss their suitability. Then we'll delve into **shader effects** for neon glows and color gradients, methods for constructing **fractal torus geometry** recursively, and **CSS/JS tricks** (glows, blurs, etc.) to enhance the visual. We'll also cover **motion design** patterns and libraries (like GSAP and Anime.js) to achieve smooth, trippy animations. A comparison table of these technologies is provided to summarize their ease of use, flexibility, performance, and aesthetic potential.

WebGL Technologies for 3D Fractal Visualizations

To create any 3D graphics in the browser, **WebGL** is the underlying engine – but you'll likely use a high-level library on top of it. The main options are **Three.js** and **Babylon.js**, or you can go *"bare metal"* with raw WebGL/GLSL shaders (or use small frameworks like OGL or even platforms like Shadertoy for prototyping). Each approach has pros and cons:

- **Three.js** – a widely-used, open-source WebGL library known for its large community and abundance of examples. Three.js is relatively lightweight and **"favors hackability over structure"** [1]. This makes it popular in creative coding and digital art circles. Developers often use Three.js as a flexible foundation, plugging in third-party add-ons or custom shaders for special effects. Three.js *"integrates easily with other web frameworks"* and gives a lot of low-level control [2]. However, many advanced features (physics, post-processing, GUI controls) require extra libraries or manual coding. In the context of our neon fractal torus, Three.js would allow us to freely combine custom geometry and shaders for the fractal and glow effects, at the cost of more custom setup. Three.js's popularity in artistic demos (CodePen, Shadertoy, etc.) means there are plenty of inspiring examples and community-made shaders to draw on [3] [4]. Overall, Three.js offers high flexibility and aesthetic potential, with a moderate learning curve.

- **Babylon.js** – a comprehensive, open-source 3D *engine* for the web. In contrast to Three's minimalism, Babylon comes as a **full-featured engine** with many built-in systems (physics, animation, GUI, VR support, etc.) out of the box [5]. It's written in TypeScript and emphasizes a structured, component-based approach. Babylon might be slightly heavier to load, but it can be easier to get complex features working quickly since so much is included. It has a visual **Node Material Editor** for building shaders graphically and a GlowLayer system for neon effects (more on that later). Babylon's philosophy is to be *"a complete 3D engine"* providing everything you need, whereas *"Three.js is a lightweight rendering engine"* that you assemble into what you need [2]. For our fractal torus, Babylon.js could handle rendering and glow with somewhat less custom code (e.g. just apply an emissive material and add a glow layer), but it might be less flexible than Three.js for wildly custom shader tricks. Babylon is often used in enterprise, games, and XR applications [6] – areas which value stability and built-in features – but it's fully capable of artistic visuals too. The community and ecosystem, while slightly smaller than Three's, are very active and focused. Babylon's ease of use is high if you follow its patterns, and performance is on par with Three (differences are minor in most cases [7] [8]).

- **Raw WebGL / GLSL** – for ultimate control, one can write directly in WebGL API and GLSL shaders without a framework. This is the most flexible route (literally anything supported by the GPU can be done) but also the most complex. Raw WebGL is **"super low-level; it's like writing assembly for 3D graphics"** [9]. You manage buffers, shader programs, and draw calls manually. In the context of a fractal torus, using raw WebGL would let you implement advanced techniques like **ray marching shaders** to render fractals that would be hard to model with geometry. In fact, many stunning fractal visuals (e.g. Mandelbulbs or distance-estimated 3D fractals) are done in pure GLSL on platforms like Shadertoy. The **aesthetic potential is limitless** with custom GLSL – one could write a shader that produces an infinite nested torus fractal by formula, with color glowing patterns, all on the GPU. Performance can be excellent for pixel shaders, but the development time is much greater. For example, the *Fractos* project uses a "specialized ray marcher [in GLSL] to render fractals in real time" [10] on top of Three.js. If you have strong shader programming skills or need a truly unique visual, raw WebGL is an option – otherwise, Three.js or Babylon.js will significantly speed up development.

In practice, many developers choose **Three.js** for creative coding projects like this neon fractal torus. Three.js's large creative community and collection of examples for visual effects (glow, post-processing, etc.) make it attractive [3] [4]. Babylon.js is equally capable, and if you prefer a more structured engine with built-in support (for example, Babylon's **GlowLayer** can add bloom to emissive materials with one call), it could reduce the amount of custom code. Both frameworks ultimately use WebGL under the hood and have similar performance characteristics – a recent comparison notes that *"performance differences are often negligible for beginners. In expert hands, Three.js can be optimized... [while] Babylon provides consistent, predictable performance"* [11]. It may come down to familiarity and the development style you prefer. If time allows, you could even prototype the fractal shader on **Shadertoy** (pure GLSL) and then integrate it into a Three.js ShaderMaterial.

Shader Effects: Glowing Neon Materials and Color Gradients

A hallmark of our desired aesthetic is **glowing neon colors** – think of the bright neon lights in a cyberpunk city or a black-light poster, where colors bloom and bleed light. In 3D rendering, achieving a glow or bloom effect requires some special handling since typical renderers won't blur or bloom on their own. We have a few techniques at our disposal, often used in combination:

Emissive Materials + Bloom Post-Processing: The simplest approach in Three.js or Babylon.js is to use emissive materials (materials that **emit light** on their own) and then apply a bloom filter so that bright areas bleed outward. For example, in Three.js you might use a `MeshStandardMaterial` or `MeshBasicMaterial` with a strong `emissive` color (even above the normal [0,1] intensity range), and then add the **UnrealBloomPass** in an EffectComposer. Bloom is a *screen-space* effect that adds a fuzzy glow around bright spots. A key trick is to allow intensities above 1.0 for truly bright neon surfaces. As one Three.js core developer advises: *"Make emissive surfaces brighter — much brighter! — and use bloom post-processing… don't limit yourself to [0,1] values. Set the bloom threshold to something > 1. Then only genuinely emissive parts will glow."* [12] In practice, you'd configure `UnrealBloomPass` with parameters like threshold, strength, and radius to get the desired halo. Babylon.js similarly has a **GlowLayer** – by default it will make any material with an emissive color glow, and you can tune its intensity or which objects it affects [13]. Using these high-level bloom effects is often the fastest way to get a neon look. For example, to make a torus glow neon pink in Three.js, you could set `mesh.material.emissive.set('#ff00ff')` and crank up the bloom strength; in Babylon, you'd ensure the torus material's emissiveColor is pink and add `new BABYLON.GlowLayer("glow", scene)`.

Custom Glow Shaders (Fresnel Glow): Another technique (if post-processing is not an option or for a more stylized effect) is the **expanded geometry glow shader**. This involves rendering a second copy of the object slightly scaled up, with a special shader that makes its edges fade out. One known approach is a Fresnel term based on the **dot product of view angle and surface normal** [14] [15]. In this method, the shader computes `intensity = pow(dot(normalize(viewVector), normal), power)` – this gives a value that is highest at faces directly facing the camera and lowest at edges. By coloring the scaled copy with an intense neon color multiplied by this intensity, you get a glowing outline around the object. Kade Keith describes this for Three.js: *"for each glowing object, use the base geometry and another copy, scaled 1.1× larger… the vertex shader calculates the dot product of view vector and normal… the fragment shader multiplies the color by that intensity. That's all it takes for a 'good-enough' glow shader."* [14] [15] This produces a halo effect that can simulate glow without post-processing. The advantage is it's just another mesh in the scene (perhaps using `AdditiveBlending` so it adds light), but the drawback is it won't illuminate other objects and might not look as natural as a true bloom. Still, for our fractal toruses, a Fresnel glow shader could accentuate their silhouettes in neon colors.

Color Gradients in Shaders: Beyond just solid neon colors, we likely want **gradient transitions** – e.g. the torus might fade from electric blue to hot pink along its form, or have rainbow rings. This can be done in the fragment shader by using some coordinate as a gradient parameter. For instance, one could use the torus's texture UVs, its world-space coordinates, or even mathematical functions to assign color. A simple approach: compute color as a function of the angle around the torus. Since a torus is symmetric, you could map its angular coordinate (the parametric angle along the tube or around the ring) to a color ramp. A fragment shader might do: `float t = uv.x; // use U coordinate` and then mix between colors (or use a sine wave to oscillate colors). Alternatively, a 1D texture (gradient texture) can be applied so that it automatically gives a neon gradient across the mesh. Many Three.js examples use this trick of applying a rainbow gradient texture to a mesh and then enabling emissive or bloom to make it glow. In a purely procedural shader, you could define a palette of neon colors and index into it. For example, one could use a smoothstep or mix function to blend from purple to teal to orange along the torus. The key is to ensure the colors are **bright** (often near fully-saturated and high value in HSV) so that they bloom nicely. Also consider using **tone mapping** if using intensities beyond 1.0, so colors don't wash out – e.g. Three.js's ACESFilmicToneMapping can preserve detail in bright neon scenes [16] .

Shader Libraries and Tools: Both Three.js and Babylon.js allow writing custom shaders or modifying materials. Three.js has `ShaderMaterial` and `onBeforeCompile` hooks to tweak built-in materials, and Babylon.js offers its Node Material Editor or `ShaderMaterial` as well. If writing GLSL from scratch, one can also prototype on **Shadertoy**. For instance, to achieve a wild multi-dimensional fractal feel, one might incorporate noise functions or time-based color shifting in the shader. Imagine the torus material pulsing colors over time (you can pass an animated uniform `u_time` to the shader and use sin/cos to cycle hues). Another aspect is **HDR rendering** – by rendering the scene in high dynamic range and then blooming, you get more intense glows. In Three.js, ensure your renderer's output encoding and toneMapping are set correctly for HDR bloom (e.g. use `WebGLRenderer.toneMappingExposure` along with UnrealBloomPass).

In summary, to get the *"sweet fairy neon punk"* look, we'll combine **bright emissive materials** with either a bloom post-process or a custom glow shader. We'll use **gradient colors** in the shader for a rich palette (potentially cycling over time). Both Three.js and Babylon.js support these: Three.js has built-in postprocessing (via `three/examples/jsm/postprocessing` or the **pmndrs/postprocessing** library which offers advanced bloom [17]), and Babylon.js has GlowLayer and lens effects. We should also note that **animated shaders** (shaders that change over time) can add a lot – e.g. a slight pulsation in glow intensity or hue shifting can contribute to that psychedelic motion.

Techniques for Fractal Nesting of Toruses

The core geometric concept here is a **fractal**: a shape that exhibits self-similarity at multiple scales. Formally, *"a fractal is a fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole"* [18] . In our case, the base shape is a torus, and fractal *"nesting"* means we want torus shapes repeated within torus shapes, recursively. There are a few ways to construct such a structure:

- **Recursive Scene Graph:** The straightforward approach is to use a recursive algorithm to add smaller torus meshes as children of larger ones. For example, you could start with a large torus mesh, then create a slightly smaller torus and position it somehow relative to the first (perhaps in the center or along the surface), then repeat for an even smaller one, and so on. Each torus could be oriented differently or offset, creating a nested effect. Pseudocode:

```
function addNestedTorus(parent, scale, depth) {
  if (depth <= 0) return;
  const torus = new THREE.Mesh(torusGeometry, neonMaterial);
  torus.scale.set(scale, scale, scale);
  parent.add(torus);
  // position torus, e.g., at the center or rotated
```

```
    torus.position.set(0, 0, 0); // could be adjusted
    addNestedTorus(torus, scale * 0.5, depth - 1);
  }
  addNestedTorus(scene, 0.5, 4);
```

This simple example attaches each smaller torus to the previous one. Because each child inherits transformations from its parent, you can achieve interesting spiraling or twisting fractals by rotating or offsetting at each level. For instance, rotate the child torus 90 degrees relative to the parent, or position multiple children around the parent's ring. A more sophisticated recursive pattern might place several toruses around the ring of a larger torus (imagine 4 little tori spaced evenly around the big torus's circumference, each of those holding even smaller tori on them, and so on). This begins to resemble a 3D fractal called a *trefoil or torus fractal*. The key is to ensure some self-similarity: each torus might carry miniature versions of itself.

- **Instancing for Performance:** One challenge with fractals is the potentially large number of objects (if each torus contains N smaller tori, and each of those contains N more, the count grows exponentially with depth). To keep performance smooth, especially in JavaScript, you'd want to limit the recursion depth and possibly use **instanced rendering**. In Three.js, `InstancedMesh` allows one geometry to be drawn many times in one draw call [19]. *"The usage of InstancedMesh will help you to reduce the number of draw calls and thus improve… performance"* [19]. So, if your fractal calls for, say, 100 small torus copies at various positions, you could use an InstancedMesh instead of 100 separate mesh objects. You'd calculate the matrices for each instance (positioned and rotated appropriately on the parent torus surface). Babylon.js has a similar instancing capability (`mesh.createInstance`). Using instancing or merging geometry will ensure the fractal complexity doesn't choke the GPU with too many draw calls.

- **Mathematical Fractals (Ray Marching):** For the most extreme fractal visuals, one could go beyond placing discrete objects and instead define the fractal shape mathematically in a shader. For example, a famous 3D fractal is the **Menger Sponge** or the **Mandelbulb**. There are formulas for volumetric fractals or distance-estimated fractals that could, for instance, produce a torus that contains ripples of smaller tori on its surface recursively. Implementing these typically means writing a fragment shader that performs a raymarch through a signed distance function (SDF). This is advanced but can yield true *infinite* self-similarity. A quaternion Julia fractal or kaleidoscopic IFS might even produce toroidal structures. The *Fractos* library we mentioned uses an SDF approach: it lets you describe fractals with transformations (translate, scale, rotate, mirror operations applied repeatedly) and then renders them via ray marching [20] [21]. In Fractos's custom syntax, one could call `torus({ radius: R, tube: r })` inside a loop with `scale(1/3)` and `mirror(...)` to produce a fractal torus structure. While very powerful, writing such a shader from scratch is a project in itself. If the goal is a real-time interactive piece, a moderate approach (limited recursion with actual geometry) might be sufficient, and you can fake deeper recursion with clever visuals (like reflections or shaders that draw patterns on the surface).

Designing a **fractal torus** likely involves deciding how the smaller tori are arranged. One idea: place a smaller torus *inside* the hole of a larger torus (concentric, each one rotated 90° from the next). Another idea: attach small tori along the outer surface like spikes. Because a torus's surface is essentially a circle swept around an axis, you could attach mini toruses at intervals around the big torus's ring (like a chain of linked rings). This could create a *loop of loops* visual. Each of those mini-tori could have even tinier ones on them, and so forth – which indeed becomes complex! It might be wise to keep the depth to maybe 2 or 3 levels for clarity and performance. Even a **two-level fractal** (a torus with smaller tori around it) can look striking, especially with neon shader effects.

Keep in mind that fractals don't have to be exact self-copies at every scale; *approximate self-similarity* still creates that fractal feel [22]. So you have artistic license in how you arrange the nested toruses. The overall goal is that viewers perceive a repeating motif. With the glow and motion added, the fractal structure will feel alive.

A rendered example of a nested torus fractal (two toroidal shapes, one inside the other). The inner torus has a lattice-like structure and glows within the outer torus. Such geometry demonstrates self-similarity at different scales, which is key to fractal design. [23]

Finally, note that fractal generation benefits from interactivity – you could let the user adjust the "depth" of the fractal or rotate the structure to appreciate its complexity. Both Three.js and Babylon.js support interactive controls (like Three's OrbitControls or Babylon's ArcRotateCamera) so the user can orbit around the fractal. This enhances the *"multi-dimensional"* aspect of the art, letting one view it from various angles (almost like a scientific visualization of a higher-dimensional object).

CSS and JavaScript Tricks for Glow, Blur, and Fluid Motion

While the heavy lifting of 3D rendering is done in WebGL, **CSS** and standard JS can complement the visual with additional effects. We can use CSS for things like background gradients, overlay glows, or UI elements in a matching neon style. We can also use vanilla JavaScript (outside of the WebGL libraries) for certain animations or event handling that give the scene a polished feel.

CSS Glow and Neon Styling: If your page includes text or HTML elements that accompany the canvas (e.g. a title or instructions in the "punk rock fairy" style), you can use pure CSS to give them a neon glow. CSS `text-shadow` and `box-shadow` properties allow creating a colored blur around elements, effectively a glow. For example, setting `text-shadow: 0 0 10px #0ff, 0 0 20px #0ff;` on a heading will make it glow cyan. There are many CSS tricks for neon: one can layer multiple text-shadows of different intensities to build a halo. CSS filters (`filter: blur(5px)`) can also blur an element; sometimes people duplicate an element, blur it, and place it behind the original to act as a glow. In our context, the 3D canvas itself could be given a CSS filter or blend mode. For instance, applying a slight `filter: blur(1px)` to the canvas (or a duplicate of it) could smooth out jagged edges of the glow. You might also use CSS animations on pseudo-elements to create flickering or pulsing borders around the canvas for extra vibe.

Background Gradients: A subtle (or not so subtle!) moving background behind the fractal can amplify the psychedelic feel. This can be done with CSS keyframes animating a gradient. Modern CSS even allows **animating gradients** smoothly, especially with the help of custom properties. One developer joked, *"Finally I can launch my* 'acid trip rave party' *website without relying on JavaScript to provide the psychedelic background!"* now that CSS supports animated gradients [24]. For example, you could have a `background: linear-gradient(45deg, magenta, cyan, lime)` on the body, and then animate the gradient's angles or color stops with `@keyframes`. This yields a constantly shifting aurora of color. Pairing a bright moving background with a semi-transparent canvas (using `renderer.setClearColor(0x000000, 0)` to make the WebGL background transparent) could make the torus fractal appear in a sea of neon colors. Alternatively, a CSS `backdrop-filter: blur()` on a background image can give a diffused light effect. Just be cautious that too much background action might distract from the fractal itself; you'll want to find a balance.

Blur and Bloom Overlays: Another trick is using an HTML canvas or SVG overlay to simulate effects like motion blur or light streaks. For example, one technique for *motion trails* is to use CSS or a second canvas to draw a translucent afterimage of the last frame, creating a feedback loop. Some creative coders overlay a `<canvas>` on top of Three.js scene, where the canvas continuously draws a faded copy of the Three.js output, causing bright moving objects to leave trails. This can deepen the *"super position cool motion"* aesthetic by visually layering frames.

UI and Interaction: CSS comes into play for any on-screen controls or info. If you have buttons to toggle animation modes or sliders for color, styling them in neon theme (glowing borders, neon hover effects) keeps the immersion. CSS `:hover` can be combined with transitions to smoothly glow a button when the user points at it. Also consider cursor effects – e.g. a custom cursor that's a glowing circle (using CSS `cursor: url(...)`) or simply an HTML element following mouse position with a glow can add a *fairy dust* feel.

JavaScript Timing and Other Libraries: Even outside of GSAP/Anime (next section), you can use the Web Animations API or simple `setInterval`/`requestAnimationFrame` to animate CSS properties or DOM elements. For example, you might use JS to periodically change a CSS variable controlling the background gradient or to synchronize a CSS animation with the WebGL animation (so that, say, the background flashes when the fractal torus pulses). Standard JS can also listen to user inputs (mouse, keyboard) to make the scene interactive – e.g. pressing a key could toggle fractal depth or start a wild color cycle.

Combining CSS with WebGL: One powerful but simple effect is using **mix-blend-mode** on the WebGL canvas. If you place the canvas over a background and set `canvas { mix-blend-mode: screen; }` or `difference`, you can get interesting compositing that can make the glow pop. For example, `mix-blend-mode: screen` will cause the canvas to blend additively with the background – this can intensify bright colors. A mode like `difference` could create an inverted ghosting effect for crazy results. These techniques venture into experimental territory, but since the theme is multi-dimensional and punk, such visual experiments might be welcome.

In short, **CSS** can add that extra 2D polish: glowing text, animated backgrounds, and stylish UI elements, while **plain JS** can orchestrate page-level effects and handle interactivity. Together, they ensure that the entire presentation – not just the 3D object – fits the neon fractal theme.

Motion Design and Animation Libraries for Psychedelic Effects

Dynamic visuals are as much about *how things move* as how they look. A fractal torus that just sits static wouldn't feel "alive." We want **fluid, trippy motion**: perhaps the torus rotates in multiple axes, the smaller tori orbit or oscillate, colors cycle, and everything pulses to create a hypnotic vibe. To achieve this, we can use high-level JavaScript animation libraries like **GSAP (GreenSock Animation Platform)** or **Anime.js**, which excel at creating smooth, complex animations easily.

GSAP (GreenSock): GSAP is a professional-grade animation library known for its high performance and rich features. It can animate any numeric property of an object, including Three.js object properties (such as mesh positions, rotations, scales, material uniforms, etc.). GSAP's syntax allows you to create timelines of animations with precise control. For example, to continuously rotate a torus, you could do: `gsap.to(torus.rotation, { duration: 10, y: 2*Math.PI, repeat: -1, ease: "none" });` – this would spin the torus around the Y-axis indefinitely. GSAP is very **flexible**; it supports sequencing, easings, repeats, yoyo, delays, and can synchronize multiple objects. A recent overview states *"GSAP is a high-performance and robust JavaScript animation library that allows you to animate anything written in JavaScript… Instead of writing complicated CSS, GSAP lets you animate an object with a single function call"* [25] . With GSAP, we could animate the fractal's components in a choreographed way: for instance, animate the parent torus rotating slowly in one direction, the child toruses rotating the opposite way faster, and perhaps pulsate the scale of the entire fractal slightly (a subtle breathing motion). GSAP's timelines would allow us to chain these so they can start, overlap, or sync as desired. If we want more interactive motion (like reacting to scroll or mouse), GSAP has plugins like ScrollTrigger, but for a self-contained art piece, a looping timeline or triggered animations on user events (click to explode the fractal, etc.) can be set up. GSAP is a bit heavier (~60KB) but very well-optimized – it can animate thousands of properties at 60fps. Given our scene likely has <100 objects and a few materials to animate, GSAP would handle it with ease.

Anime.js: Anime.js is another popular animation library, somewhat lighter-weight and with a very clean API. Anime.js can also target properties of Three.js objects or CSS or anything. In fact, the author of Anime.js demonstrated its power by animating a complex Three.js 3D scene purely with Anime.js (even though Three.js has its own modest animation system) [26] . Anime.js supports keyframes, timelines, staggered animations, and promises. Its syntax uses a single `anime({...})` call with targets and properties to animate. For example:

```
anime({
  targets: torus.rotation,
  x: [0, Math.PI * 2],
  duration: 5000,
  easing: 'easeInOutSine',
  loop: true
});
```

This would continuously rotate a torus around X. One can animate multiple properties and multiple targets together easily. Anime.js is quite capable of the *"fluid motion"* we want – it can animate along paths, do fancy easings, and coordinate multiple elements. The Anime.js v4 website itself combined Three.js for rendering with Anime.js for all animations, showing that *"you can animate pretty much anything with the lib"* [27] . The choice between GSAP and Anime.js might come down to personal preference or specific features needed. GSAP has more plugins (e.g. physics or 3D specific plugins) and is widely used in industry, whereas Anime.js is minimalist and popular in the open-source community.

Motion Design Patterns: Regardless of library, think about the patterns of motion that fit *"psychedelic"*: these often include oscillation, looped rotations, non-linear easing (smooth in-out or elastic movements), and perhaps *synchronization with music* (if you want to add an audio-reactive element). For example, a cool effect might be to have the fractal torus slowly rotate on one axis while also undulating (scaling up and down slightly like a breathing organism). This could be done by animating `torus.scale` between 0.9 and

1.1 periodically. Using an *easing function* like sine or a custom wave ensures the motion is smooth and continuous (GSAP and Anime both allow custom easing or even integration of GSAP's RoughEase for jittery punk-like motion if desired). If you want a real trippy effect, you could animate the camera in a slow orbit around the object, or even use Perlin noise to make the movement less predictable (e.g. update positions each frame with noise functions).

Another pattern is **hierarchical motion** – because we have a fractal hierarchy, a small rotation at the parent can trickle down into complex movement at the leaves. For instance, if the top-level torus rotates at 20°/s and a second-level torus rotates relative to it at 40°/s, the outer parts trace intricate paths. You might animate phase offsets so that at some moments all layers align and then they diverge – creating a visually engaging rhythm.

Combining with Shaders: Note that we can animate not just object transforms but also shader uniforms. This means things like the color gradient can be cycled over time by animation. If our shader has a uniform `u_colorPhase` or something, we could increment it smoothly to rotate the color palette. Libraries can animate these just by treating them as objects: e.g. `gsap.to(material.uniforms.u_colorPhase, { value: 2*Math.PI, duration: 10, repeat:-1, ease: "none" })` to endlessly cycle a hue shift.

Both GSAP and Anime.js will ensure the animations run smoothly using `requestAnimationFrame` under the hood. They can also be paused/resumed easily (so if you want, say, the fractal to stop pulsing when not in view or after a certain interaction, you can manage that).

For something truly *"punk rock"*, you could introduce more erratic motion momentarily – like a sudden shake or flash. GSAP's timelines allow adding a small burst of randomness (even using the **GSAP ScrambleText or RoughEase** for chaotic motion, originally meant for text but applicable to numeric animations). Anime.js can do timeline control where you trigger segments sequentially.

Physics-Based Motion: Though not explicitly requested, another layer could be physics or particle systems (imagine sparks or glow particles emanating from the torus). If one uses Babylon, its built-in particle system or physics could make parts of the fractal drift or collide. In Three.js, one would use add-ons (like using Cannon.js or tweening libraries to simulate springy motion). GSAP actually has a plugin for a simple physics tween (gravity, attraction, etc.), and Anime.js can simulate springs via its spring easing or a custom function.

In summary, **use GSAP or Anime.js to orchestrate the motion** – these libraries make it far easier to get silky animations than writing manual loops with incremental updates. They also handle cross-browser issues and timing well. As a plus, these libraries can animate CSS properties and other DOM elements too, which means you could synchronize the movement of HTML elements (like a glowing border or an SVG overlay) with the 3D scene animations for a cohesive effect.

Below is a comparison table summarizing the discussed tools and libraries, focusing on ease of use, flexibility, performance, and aesthetic potential for this fractal neon torus project:

Tool/Library	Ease of Use	Flexibility	Performance	Aesthetic Potential
Three.js	Moderate – Many examples and straightforward setup; some learning curve for 3D concepts.	High – Allows custom shaders, plugins, and integration with many tools. You hand-craft many effects (needs add-ons for physics, etc.) [2].	High – Lightweight core; efficient for most scenes. Can require optimizations for very large object counts.	Excellent – Widely used in creative coding. Known for artistic demos with custom visuals [28]. Anything from basic shapes to complex shader art is possible.
Babylon.js	Moderate/High – Comes with lots of features out-of-box (e.g. Editor, GUI) which simplifies some tasks. Strong documentation.	High – Full engine with many built-in systems. Slightly more structured approach (less free-form than Three) [29] [30]. Still extensible with shaders and plugins.	High – Comparable to Three.js. Slight overhead from rich features, but optimized core. Handles large scenes well (used in enterprise apps).	Excellent – Capable of neon visuals (GlowLayer, PBR materials). Community focuses on polished results; fewer wild “hacks,” but anything doable in Three is doable here with effort.
Raw WebGL/ GLSL	Low – Steep learning curve (must manage WebGL API, write all shaders). No abstraction.	**Very High** – Total control over rendering and shaders. Can implement unique fractal algorithms directly.	Very High – Can be extremely optimized since you tailor everything. But easy to make mistakes that hurt performance.	Unlimited – Only bounded by shader logic and your math. Can achieve truly novel fractal visuals (e.g. raymarched infinite detail) unattainable with stock libraries. Requires artistic and technical mastery to unlock.

Tool/Library	Ease of Use	Flexibility	Performance	Aesthetic Potential
GSAP (Animation)	High – API is easy to use for common Tweens/Timelines. Great documentation. Quickly get smooth animations [31].	Very High – Can animate any JS property, sequence complex timelines, and use plugins (scroll, physics, etc.). Integrates with Three.js objects seamlessly.	High – Engine is highly optimized for minimal GC and jank. Handles numerous simultaneous tweens at 60fps.	Very High – Enables sophisticated motion design: precise control allows syncing of multi-part animations, easing, and creative effects (e.g. jiggles, bounces) for dramatic neon pulses or rotations.
Anime.js (Animation)	High – Simple, intuitive API. Slightly fewer built-in extras than GSAP but very easy for basic sequences.	High – Supports timelines, keyframes, SVG/CSS animation, and can target Three.js object properties as well. Lacks some advanced plugins but covers most needs.	High – Lightweight and efficient. Can run many animations smoothly; powering complex site animations (e.g. Anime.js site itself).	Very High – Allows creative animations with minimal code. Its keyframe system can produce intricate motion patterns. Great for synchronized or staggering effects to give a playful, trippy feel.

References: The information above draws on official documentation and community insights. For instance, the Three.js vs Babylon.js differences are highlighted by LogRocket's comparison (Three.js as a lightweight, more manual engine vs Babylon.js as a feature-rich engine) [2] and a Babylon forum discussion noting Three.js's dominance in creative art demos [28]. Techniques for glow effects come from Three.js forums and tutorials, such as using bloom passes with high emissive values [12] and custom Fresnel shaders [14] [15]. The fractal definition is quoted from Mandelbrot via *Nature of Code* [18] to explain self-similarity, and fractal rendering approaches are exemplified by projects like Fractos [32]. Animation best practices are supported by a GSAP tutorial emphasizing its ease for smooth Three.js animations [31] and comments from the Anime.js creator about animating "pretty much anything" with it [27]. These sources (and more cited in-line) can be consulted for deeper guidance on each aspect of building this dynamic neon fractal torus scene.

1 3 4 6 28 29 30 Why Do Three.js Demos Often Look More Visually Impressive Than Babylon.js? - Questions - Babylon.js
https://forum.babylonjs.com/t/why-do-three-js-demos-often-look-more-visually-impressive-than-babylon-js/57974

2 5 9 Three.js vs. Babylon.js: Which is better for 3D web development? - LogRocket Blog
https://blog.logrocket.com/three-js-vs-babylon-js/

7 Three.js vs Babylon.js - Marble IT
https://marbleitsoftware.com/blog/three-js-vs-babylon-js

8 [PDF] Performance and Ease of Use in 3D on the Web - DiVA portal
https://www.diva-portal.org/smash/get/diva2:1523176/FULLTEXT01.pdf

10 20 21 32 GitHub - Martinusius/fractos: 3D fractal renderer written in TypeScript
https://github.com/Martinusius/fractos

11 Babylon.js vs Three.js: Which Should You Choose? | by Devin Rosario
https://javascript.plainenglish.io/babylon-js-vs-three-js-which-should-you-choose-14faef9f7d78

12 16 17 Whats the best way to achieve a glow effect? - Questions - three.js forum
https://discourse.threejs.org/t/whats-the-best-way-to-achieve-a-glow-effect/59724

13 GlowLayer - Babylon.js Documentation
https://doc.babylonjs.com/typedoc/classes/BABYLON.GlowLayer

14 15 three.js glow shader
https://kadekeith.me/2017/09/12/three-glow.html

18 22 8. Fractals / Nature of Code
https://natureofcode.com/fractals/

19 〔React Three Fiber〕InstancedMeshにShaderを適用する #TypeScript - Qiita
https://qiita.com/nemutas/items/5b72de3ab7870cf2f414

23 Abstract Fractal Torus Illuminated by Neon Light Stock Illustration ...
https://www.dreamstime.com/abstract-fractal-torus-illuminated-neon-light-mesmerizing-d-render-displays-two-nested-toroidal-shapes-dark-void-outer-image413156119

24 We can finally animate CSS gradient - DEV Community
https://dev.to/afif/we-can-finally-animate-css-gradient-kdk

25 31 Create Sick Web Animations in Three.js with GSAP
https://spin.atomicobject.com/animations-threejs-gsap/

26 27 The website for (newly-released) Anime.js v4 is just incredible. : r/webdev
https://www.reddit.com/r/webdev/comments/1jqmqkn/the_website_for_newlyreleased_animejs_v4_is_just/