ChatGPT

Implementing a Fractal-Holographic Cloud System in Three.js

Fractal-Holographic Cloud Design in Three.js

Designing **fractal-holographic clouds** means creating cloud forms that exhibit organic, self-similar detail (via fractal noise) and a semi-transparent, "holographic" depth. These cloud objects will serve as dynamic UI backdrops and containers in the scene, positioned around the central **Donut of Attention** (a 3D torus). The clouds should feel alive – gently shifting shape, reacting to user movement, and blending with the **Language of Light** (LoL) background. To achieve this, we propose modular cloud components with fractal noise shaders to produce realistic, non-cartoonish detail [1] [2]. Each cloud will be an independent module (e.g. a Three.js Object3D or custom class) that can be positioned behind/around the donut. They can be moved or rotated in the scene to respond to camera position (for example, subtly orbiting the donut as the user pans, reinforcing an interactive "attention field"). The following approaches outline **2–4 implementations** for these clouds, from a true volumetric solution to simpler sprite-based tricks, along with methods for **realistic shadows** and integration with the LoL geometry.

Approach 1: Volumetric Ray-Marched Fractal Clouds

Description: This approach uses **volumetric raymarching** in a shader to render fully 3D cloud volumes. Each cloud is essentially a cube or spherical volume in which the fragment shader steps through a 3D fractal noise field to determine cloud density and color [3]. The noise can be a precomputed 3D texture (e.g. Perlin-Worley or fractal Brownian motion) or a procedural function. For example, one can load a **tileable 3D noise texture** (128^3 voxels or larger) and sample it at multiple frequencies to get fractal detail [4]. The shader accumulates opacity as it marches through the volume, creating the appearance of fluffy depth. We can leverage known techniques from games and demos: Inigo Quilez's "Clouds" shader or libraries like `THREE.Cloud` (which implements a fractal noise raymarch on a `THREE.BoxGeometry`) [5]. Crucially, the shader should also compute lighting – e.g. a single sun/sky light direction – to shade the cloud. This involves casting a secondary ray toward the light for each sample to account for light attenuation (self-shadowing) inside the cloud [6]. While heavy, this yields beautiful results: soft internal shading and a dynamic, billowing look.

Shadows with Volumetric Clouds: A volumetric cloud can **receive and cast shadows** realistically. Since the cloud shader already considers light, the donut's shadow can be integrated by treating the donut as an occluder in the light ray step. One method is to use the engine's shadow map: pass the **light's depth map** into the cloud shader and, at each sample, check if that sample point lies in shadow (comparing its light-space depth to the shadow map) – if so, reduce light contribution (darkening the cloud at that point). This essentially combines standard light-space shadow mapping with the volumetric march. Another method is to explicitly **raymarch from each sample to the donut** (expensive) or approximate the donut as a simple occluder region. Additionally, the cloud itself can cast shadows onto other objects: e.g., on the background plane or donut. This can be done by enabling shadows on the cloud mesh (if using Three.js shadowMap, though a custom shader might need tweaking to write depth properly). Because volumetric objects are tricky for built-in shadows, an alternative is using a **projected texture** approach: render the cloud's density to a texture (as seen from light) and project it as a shadow. For instance, Three.js's `SpotLight` can project a texture like a slide projector [7] – one could render a "cloud shadow" texture and assign it to a light to cast soft cloud-shaped shadows on the donut or background.

Integration & Performance: In code, this approach requires writing GLSL within Three.js. For example, one could create a custom ShaderMaterial for a cloud:

```
// Pseudocode for volumetric cloud material (fragment shader outline)
uniform sampler3D noiseTex;
uniform vec3 lightDir;
uniform vec3 cloudColor;
uniform float density; // base density
void main() {
  vec3 rayDir = normalize(vPosition - cameraPosition);
  vec3 samplePos = vPosition;
  float accumulatedAlpha = 0.0;
  vec3 accumulatedColor = vec3(0.0);
  // Ray-march inside the cube volume
  for(float t = 0.0; t < 1.0; t += 0.02) {
    vec3 p = vPosition + rayDir * t * volumeDepth;
    float dens = texture(noiseTex, p).r * density;
    if(dens > 0.01) {
      // Simple light attenuation via shadow map (p_light is p in light space)
      float shadow = readShadowMap(p);
      vec3 localColor = cloudColor * dens * shadow;
      accumulatedColor += (1.0 - accumulatedAlpha) * localColor;
      accumulatedAlpha += (1.0 - accumulatedAlpha) * dens;
      if(accumulatedAlpha > 0.95) break; // opaque enough
    }
  }
  gl_FragColor = vec4(accumulatedColor, accumulatedAlpha);
}
```

The above is a simplification – in practice you'd also add ambient light scattering, and use a proper step size and noise function. You would enable `renderer.shadowMap.enabled = true` and set `donut.castShadow = true`, `cloud.receiveShadow = true`. This approach produces the highest visual fidelity (truly volumetric fractal clouds with internal lighting and soft edges), aligning with the fractal-holographic vision. **Pros:** Very rich, 3D appearance from any angle; can achieve realistic light scattering (silver linings, etc.) and dynamic self-shadowing for a "holographic" feel. **Cons:** Computationally expensive (raymarching many steps per pixel); requires custom shaders and careful optimization (fewer samples, jittering, lower resolution render target) to maintain frame rate [8] [9]. It may need WebGL2 and tricks like downsampling or temporal accumulation to be smooth. This is likely best for powerful systems or if clouds are few and not fullscreen.

Approach 2: Procedural Sprite/Plane Clouds

Description: This technique uses **billboard sprites or textured planes** with procedural shaders to fake volumetric clouds at far lower cost. Instead of a true 3D volume, we render a cloud as a flat sprite always facing the camera [10]. A custom SpriteMaterial or ShaderMaterial on a PlaneGeometry can be employed to draw a cloud texture that evolves over time. For instance, we start with a base **cloud shape texture** (an image or generated pattern that defines the general outline of a fluffy cloud) and then apply **fractal noise distortion in the fragment shader** to give it a wispy, dynamic edge [2]. The Codrops tutorial by Borghesi demonstrates this: it combines a base shape mask with moving **Simplex noise** layers to simulate a shifting interior pattern [11] [12]. Essentially, two scrolling noise textures (offset in UV over time) are blended to create a turbulent cloud density texture, and **FBM (fractal Brownian motion)** is used to perturb the sprite's UV coordinates, producing a vaporous border that changes shape over time [2]. The result is a cloud sprite that appears to boil and move like a real cloud, despite being a flat object. We can create multiple such sprites of varying sizes/opacities to represent an array of clouds (e.g., "10 Clouds" around the donut).

Enhancing 3D Illusion: To avoid the clouds feeling paper-thin, we can employ a few tricks. First, use **depth-tested, semi-transparent materials** so that multiple clouds overlap convincingly (far clouds can be partially obscured by nearer ones). Second, consider layering **crossed planes** or **clusters of sprites** for each cloud: for example, three planes at 90° angles intersecting (an **X, Y, Z billboard trio**). From most angles, the viewer sees at least one plane full-on, giving volume, and the others add thickness. These planes all share the same procedural texture but maybe cut at different thresholds to represent the cloud's different cross-sections. Another trick is **fading the sprite's opacity based on viewing angle** – when edge-on, we could make it fade so it's less noticeable as a flat object. The sprite approach leverages the fact that the cloud always faces the camera, so the user never sees it edgewise [10]; this keeps the illusion of volume as long as the camera isn't too free-roaming.

Shadows with Sprite Clouds: Standard shadow mapping can work with some care. Each cloud sprite (Plane or Sprite in Three.js) can receive shadows from the donut if `receiveShadow=true` on its material. However, a pure SpriteMaterial in Three.js might not support shadow receiving out of the box. If using a custom ShaderMaterial, we can integrate the shadow map similar to any surface: the sprite is essentially a quad in space, so when the donut is between the light and the sprite, parts of the sprite should darken. This requires computing the sprite fragment's position in light space and sampling the shadow map. Alternatively, a **projected texture** method can be simpler here: imagine a **SpotLight at the donut** projecting a fuzzy round shadow texture onto the cloud sprites behind it (the texture could be dynamically generated or a simple radial gradient representing the donut's shadow). Three.js's new spotlight texture feature allows projecting any image [7]. We could generate a circular "donut shadow" cookie and let it cast onto the sprites for a stylized effect (the shadow would naturally blur out as the light's angle widens). Another approach is **screen-space contact shadows** for small-scale softness: e.g. use a post-processing pass to darken pixels of the background or cloud that lie directly behind the donut in screen space [13]. This could connect the donut and cloud visually with a gentle shadow without heavy calculation – essentially an AO-like darkening right behind the donut's silhouette.

Pros & Cons: Pros: Sprite/plane clouds are much faster to render than volumetric – only one (or a few) quads per cloud. They are easier to code: you can even use **existing noise libraries in GLSL** (as shown in Codrops) to get the fractal edges without computing a full 3D volume. They integrate well with Three.js materials and can be manipulated like any mesh (positioned, parented, etc.). **Cons:** They are inherently 2D illusions, so if the user's perspective changes drastically (e.g., orbiting behind a cloud), the cloud always turning to face may feel unnatural. Also, shadows and lighting are less physically correct – e.g., a flat cloud can't self-shadow convincingly. The clouds might appear a bit like "painted backdrops" if not layered. Additionally, without volumetric depth, clouds cannot occlude the donut gradually (either the donut is in front or behind a sprite – there's no partial inside). However, careful design (multiple layers, slight 3D offsets) can mitigate many of these issues while keeping performance high.

Implementation: In code, one could define a **CloudSprite class** that creates a plane mesh with a ShaderMaterial. Use `THREE.TextureLoader` to load a base shape alpha texture (or generate one), and perhaps another for noise. Then in the fragment shader, sample the noise textures with moving UVs and combine with the shape mask to output `gl_FragColor.a`. For example:

```
uniform sampler2D shapeTex;
uniform sampler2D noiseTex;
uniform float time;
varying vec2 vUv;
void main(){
  // Base cloud silhouette
  float shape = texture2D(shapeTex, vUv).r;
  // Two moving noise layers
  float n1 = texture2D(noiseTex, vUv + vec2(time*0.0001, -time*0.00014)).r;
  float n2 = texture2D(noiseTex, vUv + vec2(-time*0.00005, time*0.00008)).r;
  float noisePattern = (n1 + n2)*0.5;
  // Fractal perturbation (could add fbm noise here for edges)
  // Combine shape mask and noise pattern to form alpha
  float alpha = shape * smoothstep(0.4, 0.7, noisePattern);
  vec3 cloudColor = vec3(0.95);
  gl_FragColor = vec4(cloudColor, alpha);
}
```

This shader creates a soft cloud texture that animates over time. The material should set `transparent=true` and use alpha blending. We could also modulate `cloudColor` slightly per vertex to add depth hints (darker at bottom, for example). Finally, ensure `cloudMesh.renderOrder` is managed so clouds render behind the donut (or use depth sorting).

Approach 3: Hybrid Layered Clouds and Particle Effects

Description: A hybrid approach combines **2D and 3D methods** to balance realism and performance. One option is using **layered cloud planes in 3D**: rather than a single sprite, stack multiple semi-transparent slices through the cloud's volume. For instance, you can create 5–10 parallel planes (say horizontal layers) each with a cloud texture slice. When viewed together, they approximate a volumetric density. This has been used historically but tends to look misty and can be costlier to draw many layers [14] . A more clever hybrid is to use **billboard particles or impostor spheres** to fill a cloud volume. For example, you generate a cloud as a collection of small **sprites or puffballs** (impostor spheres with faded edges) distributed within a region. Each particle samples a noise field to decide if it's part of the cloud (so the overall shape is still defined by fractal noise). This effectively creates a **point-based volumetric cloud** – the eye sees a cluster of tiny cloud bits forming a larger shape. You can then light those particles with simple techniques (e.g., a lambertian light so one side of the cloud appears brighter). The trade-off is number of particles vs. smoothness. If done sparingly (a few dozen sprites per cloud), it can look like a stylized puffy cloud and still run fast. Three.js could handle this via **InstancedMesh** or Points; each instance given a random rotation and slight scale variation for natural look.

2D Canvas & WebGL Compositing: Another hybrid concept is leveraging the existing **2D LoL canvas** for clouds. For example, clouds could be first drawn or composited in a 2D layer (perhaps using Canvas2D or a 2D context with a noise filter) and then integrated with the 3D scene via texture or overlay. One could **render the 3D donut and other objects on top of a pre-rendered 2D cloud background**, but allow some interaction by updating the 2D canvas based on 3D positions (for instance, draw a shadow or highlight on the 2D canvas where the donut overlaps). This is a form of **hybrid compositing**: the 2D layer provides rich, soft clouds and the 3D layer has the solid objects; they sync through shared data (like the donut's screen position for shadow). For instance, when the donut rotates, your code could draw a faint circular shadow on the 2D background at the donut's projected position, connecting it visually. Conversely, you could draw bright areas on the canvas where a 3D light shines through cloud gaps. This approach uses the strength of 2D (easy soft drawing) and 3D (interactive depth) combined, but it requires careful layering and z-index handling (ensuring the WebGL canvas is transparent and overlays the 2D canvas or vice versa).

Shadow Techniques (Projected, Light-Space, Screen-Space): In a hybrid scenario, multiple shadow strategies can mix: - *Projected textures:* As described, using a Spotlight with a texture (cookie) is a straightforward way to throw **stylized shadows**. For example, project an **annulus (donut-shaped) shadow texture** from the donut onto any cloud planes or even onto a background plane. The projection will naturally scale and blur with distance [7]. This could be used in any approach that has planar receivers. - *Light-space (shadow map):* This is Three.js's built-in shadow mapping. It's physically accurate and will work if clouds are implemented as standard meshes. Ensure `renderer.shadowMap.enabled=true`, add a directional or spot Light with `castShadow=true`, and for each cloud mesh set `receiveShadow=true`. If the cloud material is custom, you might need to write depth to the shadow map (for volumetric shaders, an alternative is to approximate a solid shape for shadow casting or use the shadow camera to render a simplified cloud). - *Hybrid compositing for shadows:* In cases where the background is a 2D canvas, you can manually composite shadows. For instance, draw a blurred blob under the donut's position on the canvas to simulate a **soft contact shadow** (common trick in UI/AR applications to "ground" an object). If the LoL background has geometry lines, you could darken the lines behind the donut subtly when it's overhead, as if the donut is blocking light. - *Screen-space shadows:* Using screen-space techniques (similar to SSAO or contact shadows) can add small shadow detail. Three.js even has a **Screen-Space Shadows (SSS)** pass that works alongside normal shadow maps to refine contact areas [13]. This could be enabled if your project uses post-processing; it will detect depth differences (like between donut and background) and darken the intersection region, creating a soft shadow where the donut meets the cloud or background in view.

Pros & Cons: Pros: The hybrid approach is very flexible – it can achieve good visual results with moderate performance cost by combining techniques. For example, using a few cloud layers (not too many) gives depth, while using a 2D canvas for broad sky color and light can reduce the 3D work. Particle-based clouds are highly dynamic (you could even physics-drive them if needed) and can be made to swirl or respond to forces, enhancing the interactive feel. **Cons:** It can be complex to implement and maintain, as it mixes rendering contexts. Layered planes still don't self-shadow well [14], and particle clouds might require tuning to avoid looking like noise. Also, compositing 2D and 3D can lead to **integration issues** (e.g., mismatched color profiles or z-order glitches). One must ensure the aesthetic remains consistent across layers (using similar color and lighting cues in both canvas and WebGL).

Anchoring Clouds to the Language of Light Geometry

The **Language of Light (LoL)** background is composed of circles, grids, and sacred geometry patterns. We want the 3D clouds to harmonize and possibly derive their positioning or form from this geometry, so that the scene feels unified and intentional (every cloud "knows" about the underlying pattern). Here are a few strategies to anchor and align clouds with the LoL design:

- **Geometric Positioning:** Identify key features of the LoL pattern (for example, the centers of circles, intersection points of the grid, perimeter of a "flower of life" pattern, etc.) and place cloud objects at those coordinates (projected into 3D space). For instance, if LoL has a big circle behind the donut, a cloud could be arranged to sit along that circle's arc, effectively outlining it in 3D. You might position 10 clouds corresponding to ten significant nodes in the sacred geometry (which resonates with having "10 clouds"). This ensures clouds aren't random but sit in **meaningful visual positions** relative to the pattern.

- **Anchoring & Constraint:** We can make clouds partially **follow the LoL geometry** as the user or donut moves. For example, if the LoL has concentric circles, a cloud can be constrained to move along one circle. As the camera angle changes, the cloud could slide around the circumference of that invisible circle, staying aligned. This creates a harmonious motion – the cloud is essentially "orbiting" on a sacred geometry track. We could use simple math for this: e.g., cloud.position = center + radius * (cosθ, sinθ) on the plane of the LoL pattern, updating θ over time or with interaction.

- **Geometry-Seeding Clouds:** Use the LoL shapes to **seed the cloud shapes themselves**. One idea is to initialize the fractal noise or particle distribution of a cloud using the LoL pattern. For instance, take an image of the LoL background (or a procedural representation of it) and use it as a **density map** for clouds – areas where the LoL pattern is bright or intersects could correspond to higher probability of cloud presence. In practice, you could sample points on the LoL geometry (imagine picking random points along the sacred geometry lines) and use those as centers for cloud particle emitters. This way, the **sacred geometry "births" the clouds**. A cloud might literally take the shape of a circle or flower if the seed points outline that shape, giving a holographic layering (the cloud reveals the pattern within it). For fractal noise-based clouds, you might modulate the noise function by a low-frequency shape from LoL – e.g., add a term to the density field that raises density near a LoL circle curve. Then a volumetric cloud might subtly form along that curve.

- **Alignment & Orientation:** Align cloud orientations to the geometry. If the LoL grid has, say, a diagonal orientation or certain symmetry axes, we could orient elongated cloud forms parallel to those axes. For example, if there are radial lines emanating from the center, perhaps stretch the cloud's shape slightly along those lines. This can be done by scaling the cloud object or in shader making noise slightly anisotropic following a direction from center. The goal is that at a glance the clouds seem "woven" into the background lattice, reinforcing the fractal-holographic idea that the part (cloud) reflects structure of the whole (the LoL field).

- **Dynamic Modulation:** The LoL system might not be static – if it has animations (pulsing geometry, color changes), the clouds can respond. For example, if a particular geometric pattern lights up or rotates (language of light might have cyclic patterns), a nearby cloud could change color or shape in sync. This could be implemented by linking a uniform in the cloud shader to some global LoL state (like an oscillation that matches the sacred geometry frequency). Clouds could even serve as "portals" highlighting parts of the LoL design: when a cloud container opens (see next section), it might highlight the corresponding part of the background (e.g., the cloud's position corresponds to a circle that starts glowing). Such coordination makes the **attention system feel cohesive and meaningful**, as each cloud is not arbitrary but grounded in the LoL blueprint.

In summary, treat the LoL geometry as a **scaffold or blueprint** for the clouds. This not only preserves the aesthetic vision of sacred geometry guiding the experience, but also helps the user intuitively sense that the clouds and donut are part of one integrated, symbolic system.

Interactive Cloud Membranes and UI Panels

A major goal is for each cloud to function as an **interactive container** – when the user engages with a cloud, it can reveal a "sub-membrane" UI panel or trigger a mode in the application. This adds an **experiential layer**: the clouds aren't just visuals, they are entry points or menus in the interface, consistent with the idea of fractal attention (each cloud contains deeper information). Here are strategies to implement this:

- **Cloud as a Group with Hidden Panel:** Construct each cloud as a `THREE.Group` that contains the cloud visual (volume or sprite) **plus** a child mesh for the hidden panel (the "membrane"). The panel could be a simple PlaneGeometry with some texture or HTML rendered onto it (e.g., via Canvas or CSS3D). Initially, this panel is invisible or very small. When interaction happens, we can animate it to appear. For example, if using Three.js directly, the panel mesh could have `material.opacity=0` and scale near zero, and on trigger we tween it to opacity 1 and a reasonable size, making it look like it **emerges from within the cloud**. The panel content might show controls (sliders, buttons) or information. If using HTML/CSS, one can use `CSS3DObject` to place a DOM element at the cloud's position in 3D space [15] – that DOM element could be a div containing interactive HTML UI. This way, the user can click actual HTML buttons that appear to float in the cloud. The CSS3DRenderer approach requires managing two scenes (WebGL for 3D and CSS for overlay), but it's powerful for complex UI. Alternatively, using `<canvas>` or text on a plane via Three.js might suffice if the UI is simple (like just an icon or label).

- **Interaction Triggers:** We need to decide how the user "touches" a cloud. If this is a standard browser 3D scene, the user might click or tap on a cloud. We can use **raycasting** (Three.js Raycaster) to detect a click intersection with the cloud mesh. Each cloud group can have an `.userData.id` or callback so we know which was selected. If using VR or gaze, then gazing could be detected by the reticle intersecting the cloud. Another trigger is **proximity/rotation**: for example, if the user rotates the donut such that it faces a particular cloud, that could auto-activate the cloud (since the donut's "attention" is on it). This could be done by monitoring the donut's orientation/quaternion or a target vector and checking if it aligns with the vector to a cloud. When conditions meet (angle below threshold), trigger the cloud's panel. This "rotating into a cloud" mechanism aligns with a fluid, exploratory UI – the user tilts the donut or camera and suddenly a hidden interface fades in.

- **Visual Feedback:** To make interaction intuitive, give subtle feedback when a cloud is interactive or focused. For example, on hover or gaze, the cloud could **glow or ripple**. This can be done by altering the cloud shader (e.g., briefly increase brightness or animate the noise faster) or overlaying a slight halo sprite. The cloud might also **move slightly** – e.g., float forward or enlarge – as if inviting the user in. Because the clouds are holographic, one could even let the camera move *through* a cloud to fully reveal the membrane (e.g., as the user moves closer, the cloud dissipates to unveil the UI panel entirely).

- **Use Cases:** Imagine the user sees the donut at center with several clouds around. Each cloud might correspond to a category or mode (for example, one cloud = "Focus Mode Settings", another = "Notifications", etc., in a conceptual sense). **Use Case 1:** The user clicks on the cloud to the right of the donut. Immediately, that cloud's misty form swirls apart slightly and a **circular control panel** becomes visible inside it – perhaps showing a few icons (drawn with LoL-style geometry) that the user can now click. The donut could simultaneously respond, e.g., changing color to indicate it's now in the mode represented by that cloud. **Use Case 2:** The user slowly rotates the donut object. As it turns to point at a cloud above it, that cloud begins to shimmer. When aligned, the cloud opens like a gate, and the user's view **zooms into a micro-scene** inside the cloud – the "membrane" could even be a mini 3D scene (since conceptually fractal-holographic: a smaller torus or interactive diagram could live inside). The user tweaks some settings there, then pulls back out; the cloud closes back into its fluffy form. **Use Case 3:** Simply hovering the mouse over a cloud causes a tooltip-like panel to fade in, giving a textual hint (e.g., "Analytics Controls") with a stylized design consistent with the LoL background, and looking away hides it.

To implement these, we rely on standard UI patterns applied in 3D: event listeners for clicks (e.g., using `renderer.domElement.addEventListener('pointerdown', ...)` and Raycaster intersection tests), and animations (using GSAP or Three.js's built-in tween libraries, or manual `requestAnimationFrame` updates). We also ensure that these interactions do not break the immersion – panels should be semi-transparent or smoothly integrated so they feel like part of the scene (a **membrane** is a great metaphor – it should feel like a thin film within the cloud that reveals information when energized).

Code Integration and Architecture Considerations

Given the existing project structure (with files like `torus.js`, `app.js`, `main.js`, `style.js`), integrating the cloud system should be done cleanly and modularly:

- **Cloud Module/Class:** Develop a dedicated module (e.g., `CloudSystem.js` or `Cloud.js`) that encapsulates cloud creation and management. For example, a `Cloud` class could take parameters (size, type of cloud implementation, maybe an anchor geometry point from LoL) and internally create the Three.js objects (as per Approach 1, 2, or 3). This class can also hold the logic for updating the cloud (animating noise, etc., each frame) and for triggering its UI membrane. By keeping this separate, the main app can instantiate multiple Cloud objects easily. For instance:

```
import { Cloud } from './Cloud.js';
// ... in app initialization:
const clouds = [];
LoLGeometryPoints.forEach(pt => {
  let cloud = new Cloud({ position: pt, type: 'sprite', size: 5 });
  scene.add(cloud.object3D);
  clouds.push(cloud);
});
```

Here `cloud.object3D` might be the root group containing the visual and UI panel. The Cloud class can also expose methods like `cloud.showPanel()` that the main app can call if needed (e.g., when donut rotates to it).

- **Main Scene Integration:** If the clouds are part of the main Three.js scene with the donut, ensure to configure the renderer and lighting accordingly. Enable shadows once and set the appropriate shadow map type (PCFSoftShadowMap for smoother shadows). Add a global light (sunlight or a few spotlights) that will cast shadows from the donut to clouds. In `app.js` or `main.js`, after setting up the donut (torus) and LoL background, call the cloud module to add clouds. The LoL background might currently be a 2D canvas (`style.js` might handle it). If so, consider one of two architectural approaches:

- **Overlay Canvases:** Keep the LoL canvas as a separate layer behind the WebGL canvas. In this case, the clouds and donut are drawn on the Three.js canvas with an alpha background, overlaying the LoL canvas. This requires `renderer.setClearColor(0x000000, 0)` to make background transparent. The advantage is you can keep the existing 2D drawing code intact. The challenge is syncing interactions: e.g., if a cloud needs to cast a shadow on the LoL canvas, you have to compute that and draw it in the 2D context. This can be done by capturing cloud positions and drawing shadows in `style.js` (e.g., draw a blurred circle under each cloud's 2D coordinates).
- **Unified Three.js Scene:** Alternatively, bring the LoL geometry into Three.js. You could recreate the 2D background using Three.js lines, circles, and planes (perhaps using an orthographic camera or a large flat plane textured with the LoL design). If the LoL pattern is static, you could even export it as an image and use it as a backdrop texture on a giant plane behind the donut. Then all elements (donut, clouds, geometry) live in one 3D scene and Three.js can handle occlusion and shadows automatically. For example, a directional light will cast the donut's shadow onto that background plane (which has the LoL texture), achieving a realistic connection. The downside is re-implementing the background might be time-consuming if it's complex or animated in ways easier with canvas.

- **Shader and Code Reuse:** The existing `torus.js` likely sets up the donut's material and animation. The cloud implementations we propose should not conflict with that. If using Approach 1 or 2, you will write custom shaders. You can keep these in separate files (e.g., `cloudVertex.glsl` and `cloudFragment.glsl`) and import as strings, or inline in your JS. Reuse Three.js's built-in chunks if possible (for lighting, fog, log depth buffer etc., to maintain compatibility with the rest of the scene). For example, if the scene uses logarithmic depth (common for big scenes to avoid z-fighting), include `#include <logdepthbuf_pars_vertex>` in the cloud vertex shader [16] and related lines to ensure the clouds render at correct depth.

- **Performance Tuning:** Whichever approach, build in options to toggle complexity. Perhaps have a quality setting where you can switch between volumetric and sprite clouds easily (since you've modularized it). For instance, on high-end systems use Approach 1 clouds, on low-end or mobile, instantiate Approach 2 clouds instead. The architecture could allow this by abstracting a common interface for clouds (they all have a `.update(deltaTime)` method, for example, to animate noise or move). Then the main loop (likely in `app.js` animation frame) just does `clouds.forEach(c => c.update(clock.getDelta()))`. This keeps the cloud logic self-contained.

- **Extensibility:** We should ensure that adding new cloud behaviors (or even other "attractor" objects in the attention system) is straightforward. Following good Three.js structure, we separate **scene setup**, **objects creation**, and **animation loop**. The cloud system might also listen for certain events (like `onCloudActivated` or a custom EventEmitter) to communicate back to the application (e.g., telling torus.js that "Cloud 3 activated, switch donut mode").

By adhering to these patterns, we integrate the clouds without monolithic code. The experimental cloud code in the side folder (the alternate `main.js` and `index.html`) can likely be merged by extracting the essential parts (shader, movement logic) into the new Cloud module and then creating instances in the main index. Pay attention to any global effects in that experimental code (like CSS styles or full-screen canvas behavior) and reconcile them with the main app's structure.

Conclusion and Conceptual Alignment

All the above proposals align with the **Donut of Attention's** conceptual framework by treating attention as a dynamic, fractal field. The **clouds are fractal-holographic modules**: each cloud contains interactive information yet mirrors the larger system's patterns (holographic principle). By using fractal noise and sacred geometry anchoring, the visuals themselves carry symbolic meaning (the clouds' forms echo the Language of Light, creating a sense that the UI is built from the same "code of life" the background represents [17]). Realistic, soft shadows connect the donut and clouds, symbolizing influence and focus of attention (e.g., the donut "shining" on a cloud or vice versa). The interactive membranes make the user's **engagement tangible** – you literally reach into a cloud (a nebulous thought) and find a coherent interface or idea, which is a powerful metaphor for focused attention emerging from a field of possibilities.

By combining the technical approaches above, you can achieve a compelling balance: clouds that are **visually rich, responsive, and meaningful**. For example, you might use volumetric clouds for large backdrop elements and sprite clouds for smaller interactive ones, all while using the LoL geometry to place them and guide their shapes. Each approach has trade-offs, but together they offer a toolkit to realize the **"10 Clouds" interactive backdrop** in harmony with the 2D/3D Language of Light system. The end result will be an immersive 3D browser experience where the user's attention (the donut) and the field of awareness (the clouds/LoL) are in constant, beautiful dialogue.

Sources: The techniques and ideas here draw on WebGL shading practices and Three.js capabilities. For instance, volumetric cloud rendering uses fractal noise and raymarching as discussed in Three.js community resources [5] [1], while sprite-based clouds leverage procedurally animated textures [10] [2]. Real-time shadow strategies like texture projection and screen-space shadows are informed by Three.js documentation and forums [7] [13]. These approaches are adapted and expanded to fit the unique **Donut of Attention** context, ensuring both technical feasibility and conceptual fidelity.

---

[1] [3] [4] [6] [8] [9] Efficient volumetric clouds - Resources - three.js forum
https://discourse.threejs.org/t/efficient-volumetric-clouds/66067

2 10 11 12 16 How to Create Procedural Clouds Using Three.js Sprites | Codrops
https://tympanus.net/codrops/2020/01/28/how-to-create-procedural-clouds-using-three-js-sprites/

5 GitHub - mattatz/THREE.Cloud: Ray tracing based cloud object for three.js.
https://github.com/mattatz/THREE.Cloud

7 What does the textures map do for a SpotLight and do I need one? - Questions - three.js forum
https://discourse.threejs.org/t/what-does-the-textures-map-do-for-a-spotlight-and-do-i-need-one/44357

13 What does "Screen-Space Shadows" do? - Unity Discussions
https://discussions.unity.com/t/what-does-screen-space-shadows-do/887938

14 Natural looking Volumetric Clouds Tutorial
https://3d-synthesis.com/tutorials/VolumetricCloudsTutorial.pdf

15 CSS3DRenderer – three.js docs
https://threejs.org/docs/pages/CSS3DRenderer.html

17 Art is 't - Sacred Geometry & Graphics
https://www.art-is-t.net/understand-geometry