ChatGPT

Generative Canvas System Architecture

To build this complex generative system in native JavaScript and HTML5 canvas, we propose a modular, object-oriented design. Each feature (differential growth, tensor fields, audio, etc.) is encapsulated in its own class/module, and a top-level controller manages updates and rendering. Below we outline each component, their algorithms, and how they interconnect, citing relevant references.

1. Differential Growth Engine

Concept: Differential growth simulates chains of nodes (points) connected in series. Each node experiences forces from its neighbors: *attraction* (keeping the chain together), *repulsion* (avoiding collisions), and *alignment* (staying midway between neighbors). Periodically, if adjacent nodes are too far apart, new nodes are inserted ("growth"), causing the chain to buckle into organic forms [1] [2].

Implementation: Define a `Node` class (with position, velocity, neighbors) and a `World` or `Chain` class to manage a collection of nodes/edges. In each simulation tick, for each node, compute forces: - **Attraction:** pull toward its immediate neighbors (preceding and following nodes). - **Repulsion:** push away from any node (including non-neighbors) that is closer than a threshold. - **Alignment:** pull toward the midpoint of its two neighbors.

Update positions using a numerical integrator. A 4th-order Runge-Kutta (RK4) solver can be used for stability and accuracy (especially if forces change over time) [3]. The RK4 algorithm computes intermediate slopes and yields smoother movement than simple Euler integration [3]. For example:

```
class Node {
  constructor(x,y) {
    this.pos = new Vector(x,y);
    this.vel = new Vector(0,0);
    this.neighbors = [prevNode, nextNode];
  }
  applyForces() {
    let f = new Vector(0,0);
    // Attraction to neighbors
    for (let nb of this.neighbors) {
      f.add(Vector.sub(nb.pos, this.pos).scale(attractionStrength));
    }
    // Repulsion from nearby nodes
    for (let other of world.nodes) {
      let d = this.pos.distanceTo(other.pos);
      if (other!==this && d < repulsionRadius) {
        f.sub(Vector.sub(other.pos, this.pos).scale(repulsionStrength/d));
      }
```

```
    }
    // Alignment with midpoint of neighbors
    if (this.neighbors.length == 2) {
      let mid = Vector.midpoint(this.neighbors[0].pos, this.neighbors[1].pos);
      f.add(Vector.sub(mid, this.pos).scale(alignmentStrength));
    }
    return f;
  }
}
```

During each update, use RK4 to integrate `pos` and `vel` under the net force. After updating, check all connected pairs: if two neighboring nodes are farther than a maximum distance, insert a new node between them (adaptive subdivision) [2]. Also, periodically force growth by adding new nodes to create over-constrained bends [2]. This produces the rippling, space-filling patterns characteristic of differential growth.

Citations: The core rules of differential growth (nodes, attraction/repulsion/alignment, insertion of nodes) are described by Webb and others [1] [2]. For numerical integration, RK4 is a standard high-order integrator suitable for physics simulations [3].

2. Tensor Field Flow

Concept: We define a *tensor field* over the canvas: at each grid point `(x,y)`, compute a 2×2 symmetric matrix **M(x,y)** derived from noise. The eigenvectors of this matrix give preferred local flow directions (like principal stretch directions). Particles or points move along these principal directions, producing anisotropic flow and alignment.

Matrix Construction: Using a 2D simplex noise generator (e.g. Simplex Noise JS), create smoothly varying values for the matrix elements. For example, let:

- `a = noise.simplex2(x/scale, y/scale)`,
- `c = noise.simplex2((x+offset)/scale, (y+offset)/scale)`,
- `b = noise.simplex2((x+offset2)/scale, (y+offset2)/scale)`.

Construct **M** = [[a, b], [b, c]]. Ensure positive definiteness by adjusting `a,c,b` if necessary (e.g. adding a constant to make eigenvalues positive).

Eigen Decomposition: For each **M**, compute its eigenvalues and eigenvectors. A symmetric 2×2 matrix's eigenvectors are orthogonal and easily computed from the quadratic formula. The eigenvector corresponding to the larger eigenvalue is the **principal direction** of the tensor at that point. Intuitively, applying **M** to a vector **v** yields stretching by a factor of the eigenvalues along the eigenvectors [4]. Thus, the eigenvectors define directions of maximal and minimal effect.

In code, one can solve:

```
// Given symmetric M = [[a,b],[b,c]]:
let trace = a + c;
let det = a*c - b*b;
let term = Math.sqrt(trace*trace - 4*det);
let eig1 = (trace + term)/2;
let eig2 = (trace - term)/2;
// Eigenvectors (unnormalized):
let v1 = new Vector(eig1 - c, b); // for eig1
let v2 = new Vector(eig2 - c, b); // for eig2
v1.normalize(); v2.normalize();
```

The principal eigenvector `v1` gives the flow direction.

Particle Advection: Seed particles across the canvas. For each particle, sample the local tensor eigenvector (interpolating between grid points). Advance the particle by a small step in that eigen-direction (optionally using RK4 integration for smooth streamlines). Over time, particles trace out flow lines guided by the evolving tensor field. This aligns elements along coherent structures (akin to "anisotropic diffusion"). By remapping eigenvalues (e.g. boosting one relative to the other), one can intensify alignment along chosen directions [5].

Citations: A positive-definite matrix has all positive eigenvalues, meaning it stretches space without inversion [4]. The eigen-decomposition is used to find principal directions. Rice et al. describe using anisotropic metrics to align points with prescribed directions [5].

3. Color Mapping

Hue Gradients: Map geometric or vectorial properties to colors. For instance, take the direction (heading) of a vector or the normal of a curve, compute its angle (0–360°), and convert to a hue. Using a library like **Chroma.js**, we can create smooth color scales. Example:

```
let hue = (vector.heading() + Math.PI) / (2*Math.PI) * 360;  // map angle to
[0,360]
let color = chroma.hsl(hue, 0.7, 0.5).hex();
```

Alternatively, pre-define a multi-hue scale with `chroma.scale([...colors...])` and sample it by a parameter. Chroma.js supports various color spaces (RGB, HSL, Lab, etc.) and interpolations [6] [7].

FFT-Aware Intensity: Analyze the audio via an FFT (using `Tone.FFT` or `Tone.Analyser`). Retrieve amplitude values for frequency bands. Use the amplitude to modulate visual intensity or saturation. For instance, let the average bass-band amplitude drive the brightness of a node or the thickness of strokes. Tone.js provides FFT data: "Tone.FFT returns the amplitude of the incoming signal at different frequencies" [8].

Thus, map audio to color by, e.g.: `lightness = baseLight + gain * fftValue`, or directly map an FFT band to hue shifts. Combining heading-based hue with audio-driven luminosity/saturation yields a dynamic, music-responsive palette.

Citations: Chroma.js can interpolate colors in many spaces for smooth gradients [6]. Tone.js's FFT analyzer gives frequency amplitudes for visual mapping [8].

4. TB-303–Style Audio with Tone.js

Synth Design: Emulate a Roland TB-303 bassline by using Tone.js synthesizers. A classic acid sound uses a sawtooth or square oscillator with a snappy envelope and resonant lowpass filter. In Tone.js, one might use an `FMSynth` or `MonoSynth`:

```
// Create a voice with filter envelope for acid bass
const synth = new Tone.MonoSynth({
  oscillator: { type: "sawtooth" },
  envelope: { attack: 0.01, decay: 0.2, sustain: 0.0, release: 0.1 },
  filter: { Q: 6, type: "lowpass", rolloff: -24 }
}).toDestination();
```

Schedule bass notes on the Tone.Transport (e.g. a `Tone.Loop`) to play a repeating pattern. For pads and drones, use `FMSynth` or `AMSynth` with slower envelopes and lower frequencies.

Effects: Chain effects to shape the sound:
- **Filter:** Use `Tone.Filter` for low-pass/high-pass filters. Tone.Filter supports types like `"lowpass"`, `"highpass"`, etc. [9]. Automate cutoff for resonance sweeps.
- **Delay:** Use `Tone.PingPongDelay` for stereo echo ("echo is heard first in one channel then the opposite") [10]. E.g.: `const pingPong = new Tone.PingPongDelay("8n", 0.3).toDestination(); synth.connect(pingPong);`.
- **Reverb:** Use `Tone.Reverb`, a convolution reverb. (After creating it, call `generate()` once to initialize.) Tone.Reverb "creates an Impulse Response with decaying noise" [11].
- **Filters/EQ:** Add `Tone.Filter`, `Tone.Chebyshev`, or `Tone.Distortion` as desired. For classic acid resonance, automate the lowpass cutoff or resonance.

Audio Modulation: You can further modulate audio parameters via JavaScript or even tie them to visuals. For example, map an abstract "intensity" concept to the wetness of the delay or the cutoff frequency.

Citations: Tone.js provides built-in synths and effects. PingPongDelay is documented as a stereo feedback delay [10]. Reverb is a decaying convolution reverb module [11]. Tone.Filter supports common filter types for tonal shaping [9]. Tone.FFT (and Tone.Analyser) give frequency data for syncing visuals [8].

5. Symbolic Concept Mappings

Abstract Parameters: We include a system to link high-level "concepts" (e.g. emotions, ambition) to parameter values. For instance, define a JSON or map:

```
const symbolMap = {
  "joy":   { hueShift: +20,  brightness: 0.8,  forceMag: 0.5, synthEnv: 1.2 },
  "anger": { hueShift: -10, brightness: 1.2,  forceMag: 1.0, synthEnv: 0.8 },
  // etc.
};
```

When a concept is active, apply its adjustments: shift color hue, scale node attraction forces, adjust filter cutoff or envelope. This semantic layer interprets concepts into aesthetic rules.

If phig.pdf and cosmicegg.html define specific symbol-color or force mappings, load those definitions (e.g. from embedded data). In absence of specific doc references, this is a custom mapping step: associate each abstract idea with visual/audio parameters.

Citations: (No direct sources found for "phig.pdf" or "cosmicegg.html" references. We implement the symbolic mapping as user-defined parameter mappings.)

6. Merged Geometry & Audio Synchronization

Geometry Growth: Combine the above: run multiple chains (or "exosphere" nodes) growing in parallel with randomness (random growth/mutation). Implement **mitosis/merging**: if two chains approach, optionally merge them into one network; if a chain segment grows long, split it into two (simulated cell division). Introduce a *curvature force* by adding a higher-order spring between next-nearest neighbors to smooth bends (or by inserting nodes to minimize curvature).

Audio-Responsive Motion: Use the FFT data to modulate movement. For instance, use the amplitude of a low-frequency band to alter the strength of repulsion or attract nodes more strongly (making the structure pulse to the beat). This creates synchronized motion: when the bass hits, curves might tighten or colors brighten.

Citations: These combined behaviors are an extension of the differential growth rules [1], augmented with audio feedback (a common generative approach). (Specific "exosphere.html" references were not found, but the described features align with mixed morphology systems.)

7. UI Overlay Panel

Controls: Implement an HTML/CSS overlay (e.g. a `<div>` or canvas with transparent background) containing toggle switches or checkboxes for each module: - **Forces:** Toggle gravity, electromagnetic, or noise fields (if any). - **Differential Growth:** On/off. - **Tensor Field Visualization:** Show eigenvectors or principal directions. - **Audio Synth:** Enable/disable tone generation. - **FFT Animation:** Turn on/off visual response to audio. - **Symbolic Mapping:** Toggle specific concept influences. - **Visual Layers:** Toggle drawing of nodes, paths, field vectors, etc.

Use JavaScript to listen for UI input and enable/disable features in the animation loop. For example, only update differential growth when its checkbox is checked. This modular approach keeps the single-page demo organized.

Citations: No specific sources; standard UI programming.

Performance and Modern Support

Use ES6 classes and modules, and the `<canvas>` 2D context for drawing. Avoid blocking operations in the animation loop. If needed, use `requestAnimationFrame` for smooth updates. Tone.js handles audio in a separate thread via the Web Audio API, but remember to call `Tone.start()` on a user gesture (e.g. a Play button click) to enable sound.

Overall, this single HTML/JS demo will have clearly separated classes (e.g. `World`, `Node`, `TensorField`, `Synth`, etc.) and a main script that ties them together. By precomputing fields on a grid and using efficient data structures (like spatial hashing for node neighbor searches), the system can run interactively in modern browsers.

---

1 2 GitHub - jasonwebb/morphogenesis-resources: Resources on the topic of digital morphogenesis (creating form with code). Includes links to major articles, code repos, creative projects, books, software, and more.
https://github.com/jasonwebb/morphogenesis-resources

3 math - Runge-Kutta (RK4) integration for game physics - Stack Overflow
https://stackoverflow.com/questions/1668098/runge-kutta-rk4-integration-for-game-physics

4 5 Metric Tensors for Artists | Jake Rice - Technical Artist
https://jakerice.design/2023/10/23/MetricTutorial/

6 7 chroma.js api docs!
https://gka.github.io/chroma.js/

8 Analyser
https://tonejs.github.io/examples/analysis

9 Filter
https://tonejs.github.io/docs/14.7.38/Filter

10 PingPongDelay | Tone.js
https://tonejs.github.io/docs/15.0.4/classes/PingPongDelay.html

11 Reverb
https://tonejs.github.io/docs/r13/Reverb