ChatGPT

Meta-Pet Genome Core: Deterministic Encoding and Trait Derivation

Deterministic Genome Encoding

In the new **number-theory-based genome system**, we represent each base genome as an **immutable sequence of 60 digits** (0–9). These three base sequences – **Red60**, **Blue60**, and **Black60** – are fixed reference genomes (similar to the previous DNA `BLUE60`, `RED60`, `BLACK60` templates [1]). By using digits instead of nucleotide letters, we can apply numeric pattern analysis directly. Key components include:

- **Immutable digit arrays**: We define `Red60`, `Blue60`, `Black60` as `Readonly` arrays of length 60, each filled with digits. Marking them `as const` (or using `Object.freeze`) ensures they are not modified at runtime.
- `PetGenome` **interface**: This TypeScript interface holds a pet's genome data. It includes a `type` field (the genome lineage: `"red"`, `"blue"`, or `"black"`) and a `digits` array field containing the 60-digit sequence. This structure encapsulates which base genome is used and its actual digits.
- **Genome instantiation functions**: We provide functions to create a `PetGenome` easily:
- A `createGenome(type)` function that takes a specific type (`"red" | "blue" | "black"`) and returns a new `PetGenome` with the corresponding fixed sequence.
- A `createRandomGenome()` (or an optional parameter in `createGenome`) that randomly selects one of the three base sequences for the genome. This uses simple randomness (which can later be replaced with a seeded RNG for fairness) to pick a lineage.

Below is a TypeScript module snippet illustrating the **genome encoding** definitions and functions:

```ts
// GenomeTypes.ts – defines genome types and base sequences
export type GenomeType = "red" | "blue" | "black";

export interface PetGenome {
  type: GenomeType;
  digits: Readonly<number[]>;  // 60-digit genome sequence
}

// Define the base genome sequences as immutable arrays of 60 digits each:
export const RED60: Readonly<number[]> = [
  7, 7, 5, 4, 7, 7, 7, 4, 2, 7,
  9, 7, 6, 7, 7, 5, 7, 7, 8, 7,
  7, 9, 7, 7, 5, 8, 9, 9, 0, 7,
  7, 4, 6, 7, 2, 7, 7, 7, 7, 7,
  7, 6, 5, 1, 3, 2, 1, 7, 4, 7,
  7, 3, 1, 7, 7, 4, 1, 7, 6, 8
```

```
] as const;

export const BLUE60: Readonly<number[]> = [
  2, 6, 9, 4, 6, 6, 8, 2, 8, 4,
  3, 3, 8, 2, 9, 0, 6, 4, 9, 1,
  7, 0, 9, 3, 0, 5, 1, 1, 5, 3,
  7, 1, 3, 8, 3, 5, 4, 8, 6, 4,
  6, 2, 2, 7, 2, 5, 7, 4, 9, 5,
  7, 9, 0, 1, 1, 8, 7, 0, 0, 5
] as const;

export const BLACK60: Readonly<number[]> = [
  2, 9, 1, 4, 1, 7, 7, 7, 6, 3,
  0, 0, 8, 6, 9, 1, 4, 1, 3, 1,
  4, 5, 6, 2, 0, 8, 7, 0, 9, 1,
  1, 9, 0, 7, 7, 7, 1, 4, 1, 2,
  3, 6, 7, 7, 7, 1, 4, 1, 9, 2,
  0, 0, 8, 6, 9, 1, 4, 1, 3, 1
] as const;  // This Black genome is palindromic (reads same backwards)

// Helper to get base sequence by type:
const BASE_SEQUENCE: Record<GenomeType, Readonly<number[]>> = {
  red:  RED60,
  blue: BLUE60,
  black: BLACK60
};

// Create a PetGenome from a specified type:
export function createGenome(type: GenomeType): PetGenome {
  return { type, digits: BASE_SEQUENCE[type] };
}

// Create a PetGenome with a random base type:
export function createRandomGenome(): PetGenome {
  const types: GenomeType[] = ["red", "blue", "black"];
  const randomType = types[Math.floor(Math.random() * types.length)];
  return createGenome(randomType);
}
```

Notes: The `PetGenome` returned simply holds a reference to the chosen base sequence (no mutation applied yet). All functions are **pure** and have no side effects (aside from using `Math.random` for `createRandomGenome`). This deterministic encoding ensures that a given `type` always yields the exact same starting sequence every time, which is crucial for consistency. The use of immutable constants means any code using these genomes cannot accidentally alter the base sequences.

Basic Trait Expression

With the genome encoded as digit sequences, we derive pet traits **deterministically** by analyzing numeric patterns in the sequence. We establish clear **mapping rules** from genome patterns to high-level traits:

- **Digit Frequency → Temperament**: We count how often each digit 0–9 appears in the 60-digit genome. The digit that occurs most frequently indicates the pet's *temperament*. For example, a genome dominated by a high digit like 7 or 8 might signify a "bold" or "aggressive" temperament, whereas one dominated by a low digit like 0 or 1 might indicate a "calm" temperament. (The idea is that different dominant digits map to different personality archetypes in the game world.)
- *Implementation:* find the max-frequency digit in `genome.digits`. Then map that digit to a descriptive temperament string. This mapping can be as simple as splitting by ranges or a lookup table (e.g. {0,1,2} = "Calm", {3,4} = "Cautious", {5,6} = "Playful", {7,8} = "Bold", {9} = "Chaotic"} as an example).
- **Palindromic Runs → Symmetry Trait**: A **palindromic run** is a sequence of digits that reads the same forward and backward (e.g. `12321`). If the genome contains any substantial palindromic substring, it grants a symmetry-related trait. This could be a boolean trait like `symmetricalPattern` or a descriptive trait indicating mirror-like features.
- *Implementation:* scan the digit array for palindromes of length >= 3 (or any chosen threshold). For simplicity, we can check every possible substring or just specific lengths. If any palindrome is found, mark the pet as having the "Symmetrical" trait (perhaps reflecting in perfectly balanced markings or structure); if none, trait could be "Asymmetrical". (In our example above, the Black60 sequence is entirely palindromic, so a pet with a black genome would definitely get the symmetry trait.)
- **Distribution Variety (Entropy) → Complexity/Rarity**: We evaluate how **diverse** the genome's digit distribution is. If all digits are evenly represented, the sequence has high entropy (maximum uncertainty) which implies *complexity* and possibly a *rare* combination [2]. If the sequence is heavily skewed to a few digits, entropy is low, indicating a simpler or more common pattern. We translate this into a trait reflecting the pet's genetic complexity or rarity:
- *Implementation:* compute the frequency of each digit and calculate a normalized entropy value (Shannon entropy). High entropy (close to the theoretical max for 10 symbols) means high variety; low entropy means one digit dominates. We can categorize the result into tiers (e.g. "High Complexity" vs "Low Complexity") or even assign a rarity level ("Common", "Uncommon", "Rare") – for example, a genome with all 10 digits appearing roughly equally is *high complexity*, whereas a genome with one digit appearing 50% of the time is *low complexity*. **Note:** Entropy is maximized when all digits are equally likely [2], and minimized when the distribution is very uneven, so this gives a quantitative basis for the trait.

Using these rules, we implement a pure function `deriveTraits(genome: PetGenome): PetTraits` that analyzes a given genome and returns a structured traits object. This `PetTraits` could be an interface defining properties like `temperament`, `symmetry`, `complexity`, etc., each holding a descriptive value. All calculations are deterministic from the input genome.

Below is an example implementation of trait derivation in TypeScript:

```
// Traits.ts – derive traits from a PetGenome
export interface PetTraits {
```

```
  temperament: string;   // e.g. "Calm", "Bold", "Chaotic", etc.
  symmetry: string;      // e.g. "Symmetrical" or "Asymmetrical"
  complexity: string;    // e.g. "High Variety" / "Low Variety" or rarity tier
}

export function deriveTraits(genome: PetGenome): PetTraits {
  const digits = genome.digits;

  // 1. Determine dominant digit (most frequent)
  const freqMap: { [digit: number]: number } = {};
  for (const d of digits) {
    freqMap[d] = (freqMap[d] || 0) + 1;
  }
  const mostFreqDigit = Object.keys(freqMap).reduce((a, b) =>
    freqMap[Number(a)] > freqMap[Number(b)] ? a : b
  );
  const dominant = Number(mostFreqDigit);

  // Map the dominant digit to a temperament trait.
  // (Example mapping – this can be adjusted or made more granular)
  let temperamentDesc: string;
  if (dominant <= 2)       temperamentDesc = "Calm";
  else if (dominant <= 4)  temperamentDesc = "Reserved";
  else if (dominant <= 6)  temperamentDesc = "Playful";
  else if (dominant <= 8)  temperamentDesc = "Bold";
  else                     temperamentDesc = "Chaotic";

  // 2. Check for palindromic runs in the sequence
  let hasPalindrome = false;
  const length = digits.length;
  // Simple check: look for any palindrome of length 5 or more
  for (let runLength = 5; runLength <= length; runLength++) {
    for (let start = 0; start <= length - runLength; start++) {
      const end = start + runLength - 1;
      // Check if digits[start...end] forms a palindrome
      let isPalin = true;
      for (let i = 0; i < Math.floor(runLength/2); i++) {
        if (digits[start + i] !== digits[end - i]) {
          isPalin = false;
          break;
        }
      }
      if (isPalin) {
        hasPalindrome = true;
        break;
      }
    }
    if (hasPalindrome) break;
```

```
  }
  const symmetryDesc = hasPalindrome ? "Symmetrical" : "Asymmetrical";

  // 3. Compute distribution entropy or variety measure for complexity
  const total = digits.length;
  let entropy = 0;
  for (let d = 0; d < 10; d++) {
    const count = freqMap[d] || 0;
    if (count > 0) {
      const p = count / total;
      // Shannon entropy contribution: -p * log2(p)
      entropy -= p * Math.log2(p);
    }
  }
  // Normalize entropy to [0,1] range by dividing by log2(10) (max entropy for
10 symbols)
  const normalizedEntropy = entropy / Math.log2(10);
  let complexityDesc: string;
  if (normalizedEntropy > 0.8)       complexityDesc = "High Variety";   // very
diverse distribution
  else if (normalizedEntropy > 0.4)  complexityDesc = "Moderate Variety";
  else                               complexityDesc = "Low Variety";    //
highly skewed distribution

  return {
    temperament: temperamentDesc,
    symmetry: symmetryDesc,
    complexity: complexityDesc
  };
}
```

This function examines the genome's digits step by step: - It builds a frequency map and identifies the **dominant digit**. In our example definitions, for instance, the Red genome has digit `7` repeating the most, so that would yield a bold/aggressive temperament. The temperament categories are just illustrative – they ensure that small digits correspond to calmer traits and large digits to more energetic ones. - It then checks for **palindromic sequences** within the genome. We chose palindromes of length ≥ 5 for a significant symmetry pattern (this threshold can be tuned). If any such run exists, we label the pet as "Symmetrical". If none are found, it's "Asymmetrical". (In practice, even a single palindrome substring is enough to imply some symmetrical genetic trait.) Palindromic numbers are essentially sequences that read the same backwards and forwards [3], indicating a form of reflectional symmetry in the genome. - Next, it calculates the **entropy of the digit distribution**. Using the Shannon entropy formula, we sum `-p * log2(p)` for each digit (where p is the frequency of that digit divided by 60). We then normalize by `log2(10)` to get a 0–1 scale (0 means only one digit present, 1 would mean perfectly even distribution of all 10 digits). This gives an objective measure of variety in the genome. We map this to a complexity descriptor: e.g., if entropy is very high (close to 1), we label "High Variety" (indicating a complex, diverse genome); if very low, "Low Variety" (a simple or monomorphic genome). These could correspond to rarity tiers in gameplay, since a

highly uniform or an extremely diverse genome might both be less common than a moderately varied one [2].

The resulting `PetTraits` object might look like, for example:

```
{
  "temperament": "Bold",
  "symmetry": "Symmetrical",
  "complexity": "High Variety"
}
```

(for a genome that had a dominant high digit and contained a palindrome and a very even digit spread).

Integration and Testing

The above **TypeScript module** can be placed into the `engine/` folder of the MVP. It is self-contained and uses only basic language features (no external libraries or browser APIs), keeping dependencies minimal. All functions (`createGenome`, `createRandomGenome`, `deriveTraits`) are **pure logic**: - They perform calculations or selection based solely on input parameters (or internal constants) and produce outputs without side effects. - This makes them easy to unit test: for instance, you can feed a known `PetGenome` into `deriveTraits` and assert that the returned traits match expected values for that genome. Each component can be tested in isolation (e.g., test that a known palindrome in a sequence sets `symmetry` to "Symmetrical", or that a heavy frequency of one digit yields the correct temperament category).

Finally, this deterministic core is designed to be **extensible**. In the future, we can integrate *evolutionary changes* or *random mutations* by modifying the genome digits (with controlled randomness) and then re-running `deriveTraits` to update traits. Because the trait derivation is deterministic, the same genome state will always produce the same traits, fulfilling the requirement of consistency (important for fairness and predictability in gameplay). This forms a solid backbone for later adding *"fair randomness"* on top – for example, introducing mutations or cross-breeding will alter the digit sequences, but the trait computation remains a transparent, explainable mapping from those sequences to pet characteristics. The output trait values can directly feed into the UI (for displaying personality or rarity), into journal logs (text descriptions of the pet), or into gameplay logic (affecting the pet's behavior or evolution triggers), without any further randomization needed at the point of use.

In summary, we've established a **deterministic genome encoding and trait derivation system** that is lean, expressive, and ready to drop into the project: - **Genome definitions:** Three immutable 60-digit arrays define the genetic baseline for all pets. - **Genome instantiation:** Simple functions allow creating a genome by type or at random, enabling easy pet initialization. - **Trait derivation:** A pure function analyzes digit patterns (frequency, symmetry, diversity) to produce a rich `PetTraits` object. - **Ready for use:** This module can be tested and integrated immediately, providing consistent trait outcomes and a platform on which more complex genetic mechanics (mutations, breeding, etc.) can be built.

---

1 genetics.ts
https://github.com/Blackcockatoo/META-PET/blob/27da426611c08c8e812cffe10ca56ccf7809cc26/src/genetics.ts

2 Information, science and biology · Creation.com
https://creation.com/en/articles/information-science-and-biology

3 Palindromic number - Wikipedia
https://en.wikipedia.org/wiki/Palindromic_number