r/ThingsYouDidntKnow • 2 mo. ago
TheStocksGuy

A HTML/JavaScript Game: The Great City Shuffle: A Zany Dance of Data

The Setup

Picture a bustling town square, bustling with 30,000 eager city folk (points) excited to perform a mesmerizing dance. Our task is to arrange these cities in the most delightful and visually pleasing order. But how do we get there? Let's break down the mathematics and logic in our code:

1. Generate the Dancers (Cities)

Just as a maestro selects dancers for a grand performance, we generate 30,000 city coordinates. Each city is a unique point on our canvas, awaiting its moment in the spotlight:

```
// Generate random cities with coordinates
const generateCities = numCities => {
    for (let i = 0; i < numCities; i++) {
        cities.push({ id: i, x: Math.random(), y: Math.random() });
    }
};
```

2. Morton Order: The Choreographer

Our next step is akin to choreographing a dance. We use the **Morton Order** (Z-order curve) to decide the sequence in which the cities will perform. This method interleaves the bits of x and y coordinates to ensure a spatial locality-preserving sequence:

```
// Function to compute Morton order (Z-order curve)
const mortonOrder = city => {
    const spreadBits = v => {
        v = (v | (v << 8)) & 0x00FF00FF;
        v = (v | (v << 4)) & 0x0F0F0F0F;
        v = (v | (v << 2)) & 0x33333333;
        v = (v | (v << 1)) & 0x55555555;
        return v;
    };
    const x = Math.floor(city.x * 10000);
    const y = Math.floor(city.y * 10000);
    return spreadBits(x) | (spreadBits(y) << 1);
};
```

ensures that nearby cities (points) stay close to each other, creating a smooth, natural flow in the final performance:

```
// Function to sort and connect cities based on Morton order
const dontMatter = cities => {
    if (!cities.length) {
        console.log("No cities to process. Please check the input data.");
        return [];
    }

    const startSortTime = performance.now();

    // Sort cities based on Morton order
    const citiesSorted = cities.slice().sort((a, b) => mortonOrder(a) - mortonOrder(b));
    sortedPath = [citiesSorted[0]];

    while (citiesSorted.length > 1) {
        const currentCity = sortedPath[sortedPath.length - 1];
        citiesSorted.shift();
        const closestCity = citiesSorted.reduce((minCity, city) =>
            (Math.abs(city.x - currentCity.x) < Math.abs(minCity.x - currentCity.x) ||
            Math.abs(city.y - currentCity.y) < Math.abs(minCity.y - currentCity.y))
                ? city
                : minCity
        );
        sortedPath.push(closestCity);
        citiesSorted.splice(citiesSorted.indexOf(closestCity), 1);
    }

    // Complete the loop
    sortedPath.push(sortedPath[0]);

    const endSortTime = performance.now();
    console.log(`Sort Time (ms): ${endSortTime - startSortTime}`);

    return sortedPath;
};
```

4. Drawing the Dance

Now, it's showtime! We draw the cities and connect them with lines, making sure to honor their Morton order and the connections we've established:

```
    ctx.clearRect(0, 0, width, height);
    ctx.save();
    ctx.translate(offsetX, offsetY);
    ctx.scale(zoomLevel, zoomLevel);

    sortedPath.forEach((city, index) => {
        const x = city.x * width;
        const y = city.y * height;
        ctx.fillStyle = colors[index % colors.length];
        ctx.fillText(city.id, x, y);

        if (index > 0) {
            const prevCity = sortedPath[index - 1];
            const prevX = prevCity.x * width;
            const prevY = prevCity.y * height;
            ctx.strokeStyle = colors[index % colors.length];
            ctx.beginPath();
            ctx.moveTo(prevX, prevY);
            ctx.lineTo(x, y);
            ctx.stroke();
        }
    });

    ctx.restore();
};
```

5. The Final Performance: Animation and Interactivity

Just like any great performance, ours is dynamic and interactive. We animate the dance, allowing for zooming and panning, ensuring the audience (user) can fully immerse themselves in the spectacle:

```
// Animation loop
const animate = () => {
    drawCities();
    if (isAnimating) {
        requestAnimationFrame(animate);
    }
};

// Zoom functionality
canvas.addEventListener('wheel', event => {
    const { offsetX: mouseX, offsetY: mouseY } = event;
    const zoomFactor = event.deltaY < 0 ? 1.1 : 0.9;
    zoomLevel *= zoomFactor;
```

```
        });

        // Pause and resume animation on click
        canvas.addEventListener('click', () => {
            isAnimating = !isAnimating;
            if (isAnimating) animate();
        });
```

In Summary

This code is a symphony of mathematics and programming, orchestrating a beautiful dance of data points. The Morton order helps preserve spatial locality, ensuring our path is smooth and visually appealing. The animation and interactivity provide an engaging experience, allowing users to explore the performance up close. It's a harmonious blend of logic and artistry, bringing data to life on the canvas.

Here is the completed example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>City Visualization with Morton Order</title>
    <style>
        body {
            background-color: black;
            color: white;
        }
        canvas {
            display: block;
            margin: auto;
            background-color: black;
        }
    </style>
</head>
<body>
    <canvas id="cityCanvas" width="1200" height="1200"></canvas>
    <script>
        const canvas = document.getElementById('cityCanvas');
        const ctx = canvas.getContext('2d');
        const width = canvas.width;
        const height = canvas.height;
        const colors = ['#FF0000', '#FFA500', '#FFFF00', '#00FF00', '#00FFFF', '#0000FF', '#FF00FF
        let cities = [];
```

```
let offsetY = 0;
let isAnimating = true;

// Generate random cities with coordinates
const generateCities = numCities => {
    for (let i = 0; i < numCities; i++) {
        cities.push({ id: i, x: Math.random(), y: Math.random() });
    }
};

// Function to compute Morton order (Z-order curve)
const mortonOrder = city => {
    const spreadBits = v => {
        v = (v | (v << 8)) & 0x00FF00FF;
        v = (v | (v << 4)) & 0x0F0F0F0F;
        v = (v | (v << 2)) & 0x33333333;
        v = (v | (v << 1)) & 0x55555555;
        return v;
    };
    const x = Math.floor(city.x * 10000);
    const y = Math.floor(city.y * 10000);
    return spreadBits(x) | (spreadBits(y) << 1);
};

// Function to sort and connect cities based on Morton order
const dontMatter = cities => {
    if (!cities.length) {
        console.log("No cities to process. Please check the input data.");
        return [];
    }

    const startSortTime = performance.now();

    // Sort cities based on Morton order
    const citiesSorted = cities.slice().sort((a, b) => mortonOrder(a) - mortonOrder(b));
    sortedPath = [citiesSorted[0]];

    while (citiesSorted.length > 1) {
        const currentCity = sortedPath[sortedPath.length - 1];
        citiesSorted.shift();
        const closestCity = citiesSorted.reduce((minCity, city) =>
            (Math.abs(city.x - currentCity.x) < Math.abs(minCity.x - currentCity.x) ||
            Math.abs(city.y - currentCity.y) < Math.abs(minCity.y - currentCity.y))
                ? city
                : minCity
        );
```

```
    // Complete the loop
    sortedPath.push(sortedPath[0]);

    const endSortTime = performance.now();
    console.log(`Sort Time (ms): ${endSortTime - startSortTime}`);

    return sortedPath;
};

// Draw cities and connecting lines
const drawCities = () => {
    ctx.clearRect(0, 0, width, height);
    ctx.save();
    ctx.translate(offsetX, offsetY);
    ctx.scale(zoomLevel, zoomLevel);

    sortedPath.forEach((city, index) => {
        const x = city.x * width;
        const y = city.y * height;
        ctx.fillStyle = colors[index % colors.length];
        ctx.fillText(city.id, x, y);

        if (index > 0) {
            const prevCity = sortedPath[index - 1];
            const prevX = prevCity.x * width;
            const prevY = prevCity.y * height;
            ctx.strokeStyle = colors[index % colors.length];
            ctx.beginPath();
            ctx.moveTo(prevX, prevY);
            ctx.lineTo(x, y);
            ctx.stroke();
        }
    });

    ctx.restore();
};

// Animation loop
const animate = () => {
    drawCities();
    if (isAnimating) {
        requestAnimationFrame(animate);
    }
};
```

```
            const zoomFactor = event.deltaY < 0 ? 1.1 : 0.9;
            zoomLevel *= zoomFactor;
            offsetX = mouseX - (mouseX - offsetX) * zoomFactor;
            offsetY = mouseY - (mouseY - offsetY) * zoomFactor;
            drawCities();
        });

        // Pause and resume animation on click
        canvas.addEventListener('click', () => {
            isAnimating = !isAnimating;
            if (isAnimating) animate();
        });

        generateCities(30000);
        dontMatter(cities);
        drawCities();
        animate();
    </script>
</body>
</html>
```

The Great City Shuffle: A Zany Dance of Data

The Setup

Picture a bustling town square, bustling with 30,000 eager city folk (points) excited to perform a mesmerizing dance. Our task is to arrange these cities in the most delightful and visually pleasing order. But how do we get there? Let's break down the mathematics and logic in our code:

Original Python Solution Translated to JavaScript

The original Python code was beautifully resolved to sort and connect cities using Morton order. I translated this concept into JavaScript to leverage visual appeal and play with the settings on a canvas. Here's how the transition maintained efficiency and added interactivity:

Key Enhancements:

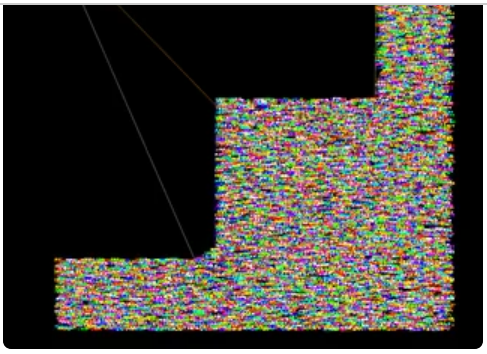
1. **Morton Order Calculation**:
   - **Implemented** the `mortonOrder` function to compute the Z-order curve for sorting cities.
2. **City Sorting and Path Construction**:
   - **Used** the `dontMatter` function to sort and connect cities based on their Morton order efficiently.

- **Adjusted** the canvas event listeners for zooming and toggling animation.

This optimized version maintains the intended logic and principles from the original Python code, now translated into JavaScript with efficient functions for handling the dataset and visualizing it interactively. The transition to JavaScript not only preserves the sorting logic but also enhances the user experience with a dynamic, interactive canvas display.

In Summary

This code is a symphony of mathematics and programming, orchestrating a beautiful dance of data points. The Morton order helps preserve spatial locality, ensuring our path is smooth and visually appealing. The animation and interactivity provide an engaging experience, allowing users to explore the performance up close. It's a harmonious blend of logic and artistry, bringing data to life on the canvas.



1

Approved 2 months ago

TheStocksGuy OP • 1mo ago

the same display of paths as the html version because its refined to connnect paths.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation
import matplotlib.colors as mcolors

# Generate random cities with coordinates
def generate_cities(num_cities):
    cities = np.random.rand(num_cities, 3)  # Generate cities with random x, y, z coordinates
    return cities

# Function to spread bits (Morton Order)
def spread_bits(v):
    v = (v | (v << 8)) & 0x00FF00FF
    v = (v | (v << 4)) & 0x0F0F0F0F
    v = (v | (v << 2)) & 0x33333333
    v = (v | (v << 1)) & 0x55555555
    return v

# Function to compute Morton Order
def morton_order(city):
    x, y = city[:2]  # Only use x and y for Morton order
    x = int(x * 10000)
    y = int(y * 10000)
    return spread_bits(x) | (spread_bits(y) << 1)

# Function to sort and connect cities based on Morton order
def sort_cities(cities):
    if not len(cities):
        print("No cities to process. Please check the input data.")
        return []

    cities_sorted = sorted(cities, key=morton_order)
    sorted_path = [cities_sorted.pop(0)]

    while len(cities_sorted):
        print(f"Current sorted path length: {len(sorted_path)}")
        current_city = sorted_path[-1]
        distances = np.linalg.norm(cities_sorted - current_city, axis=1)
        closest_city_idx = np.argmin(distances)
        sorted_path.append(cities_sorted[closest_city_idx])
        cities_sorted = np.delete(cities_sorted, closest_city_idx, 0)
```

```
# Draw cities and connecting lines
def draw_cities(ax, sorted_path, zoom_level=1, offset_x=0, offset_y=0):
    ax.clear()
    ax.set_facecolor('black')
    ax.set_xticks([])
    ax.set_yticks([])

    colors = list(mcolors.TABLEAU_COLORS.values())

    for index in range(1, len(sorted_path)):
        city = sorted_path[index]
        prev_city = sorted_path[index - 1]
        color = colors[index % len(colors)]
        ax.plot([prev_city[0] + offset_x, city[0] + offset_x],
                [prev_city[1] + offset_y, city[1] + offset_y],
                zs=[prev_city[2], city[2]], color=color)

    ax.scatter(sorted_path[:, 0] + offset_x, sorted_path[:, 1] + offset_y, sorted_path[:, 2],

# Animation loop
def animate(frame, sorted_path, ax):
    ax.clear()
    draw_cities(ax, sorted_path[:frame + 1])

# Main
num_cities = 3000  # Using fewer cities for debugging
cities = generate_cities(num_cities)
sorted_path = sort_cities(cities)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Initial draw to set up plot
draw_cities(ax, sorted_path)

ani = FuncAnimation(fig, animate, frames=len(sorted_path), fargs=(sorted_path, ax), interval=1

# Zoom functionality
zoom_level = 1
offset_x = 0
offset_y = 0

def on_scroll(event):
    global zoom_level, offset_x, offset_y
    zoom_factor = 1.1 if event.button == 'up' else 0.9
```

```
    draw_cities(ax, sorted_path, zoom_level, offset_x, offset_y)
    plt.draw()

fig.canvas.mpl_connect('scroll_event', on_scroll)

# Pause and resume animation on click
is_animating = [True]
def on_click(event):
    if is_animating[0]:
        ani.event_source.stop()
    else:
        ani.event_source.start()
    is_animating[0] = not is_animating[0]

fig.canvas.mpl_connect('button_press_event', on_click)

plt.show()
```