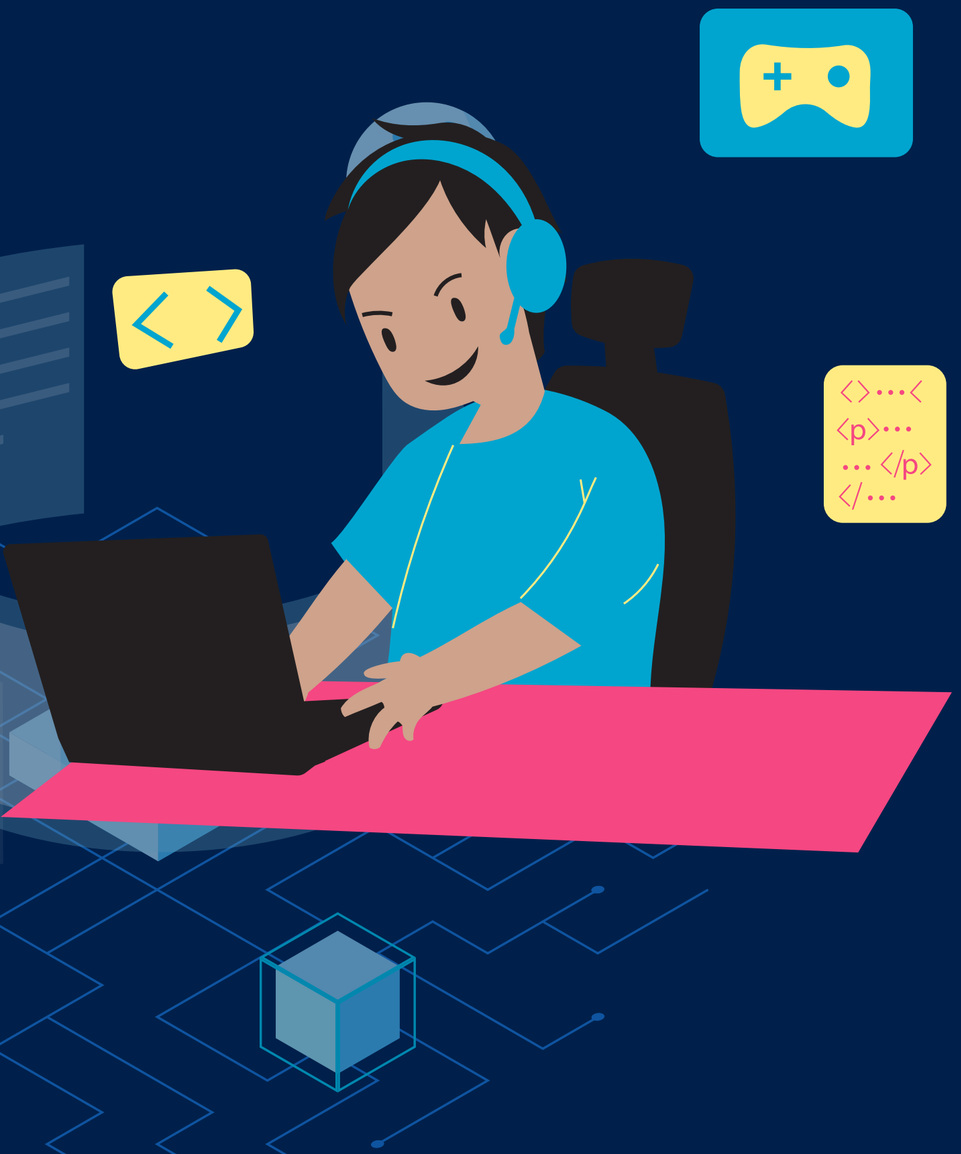


RANDOM WORLD GENERATION

EXPLORING ALGORITHMS, VISUAL STYLES, AND OPTIMIZATION TECHNIQUES

NAME:- SHUVRADEEP BERA
ROLL:- 24IE10042

INTRODUCTION

What is Procedural Generation?

Procedural generation is the practice of using algorithms to create game content (like maps, characters, items, etc.) automatically instead of manual creation.

Importance of Procedural Generation

- Enhances replayability and unpredictability in games.
- Procedural generation lowers manual labour, thus reducing development time and costs

Key Takeaways

- Discussing algorithms like Perlin noise, simplex noise, cellular automata, and Voronoi diagrams.
- Understanding their development, algorithm, applications, limitations and optimization techniques.

PERLIN NOISE

Algorithm

GRID GENERATION

- Perlin noise operates on n-dimensional grid
- Each grid point (or vertex) is assigned a random gradient vector

DOT PRODUCT

- For any point within a grid cell, calculate the offset vectors from each corner of the cell to the point.
- Dot product of each gradient and offset vector is computed

INTERPOLATION

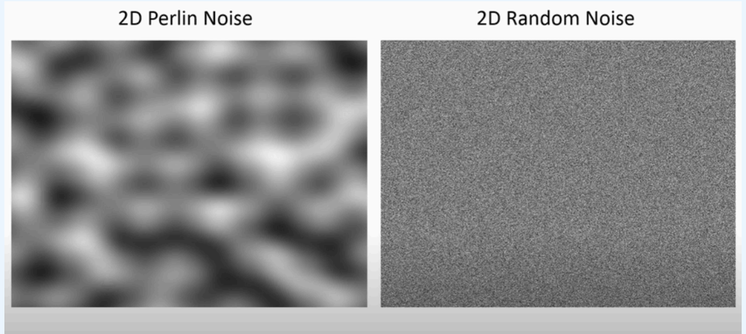
- Use these dot products to interpolate between the corners of the grid cell.
- First top left and top right, then bottom left and bottom right
- Then, interpolate these teo to get final noise at that point

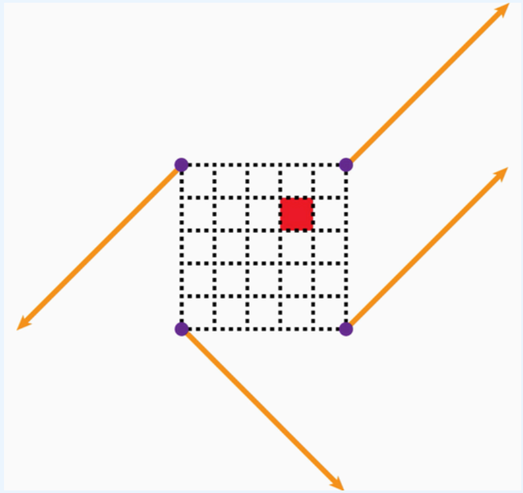
FADE FUNCTION (EASE CURVE)

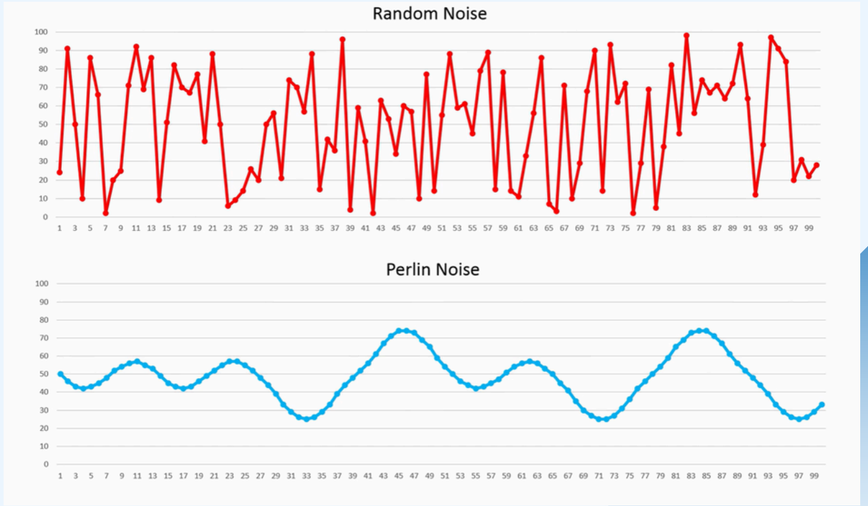
- To ensure smooth transitions,an ease curve is used.
- This function, typically 6t^5−15t^4+10t^3, is applied to interpolation parameters to make transitions more gradual and natural-looking

WHAT IS PERLIN NOISE?

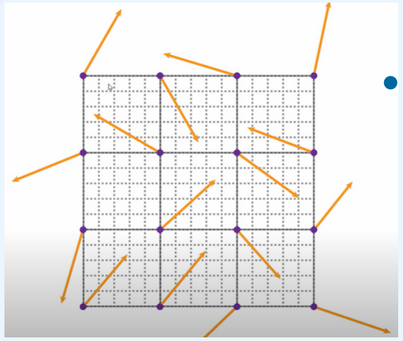
Perlin noise is a gradient-based procedural noise algorithm developed by Ken Perlin in 1983. It generates smooth, natural-looking randomness



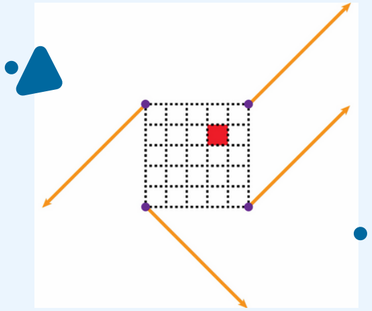




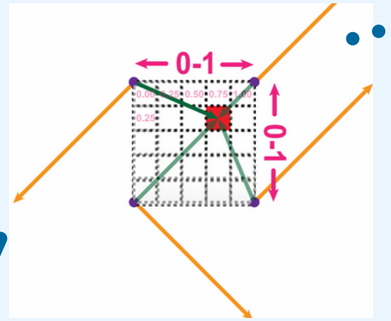
PERLIN NOISE ALGORITHM



Gradient vectrors assigned to various corners



One cell chosen and gradient vectors randomly generated



Diistance of each cell from different corners are measured and later used in dot product

Dot product computed gradient vector and pixel distance vector

$$D_1 = \vec{G}(-1,1) \cdot \vec{D}(0.75, 0.25) = -0.5.$$
$$D_2 = \vec{G}(1,-1) \cdot \vec{D}(-0.25, 0.25) = -0.5.$$
$$D_3 = \vec{G}(1,1) \cdot \vec{D}(-0.75, 0.75) = 0.$$
$$D_4 = \vec{G}(1,-1) \cdot \vec{D}(-0.25, -0.75) = 0.5.$$

Now these dot products are used along with fade function parameters
For eg.-

For (x, y) = (2.3, 4.7):

- The **bottom-left** corner of the grid cell is $(x_0, y_0) = (2, 4)$
- The **top-right** corner is $(x_0+1, y_0+1) = (3, 5)$
- The **fractional part** of (x, y) inside this grid cell is:

$$t_x = x - x_0 = 2.3 - 2 = 0.3$$
$$t_y = y - y_0 = 4.7 - 4 = 0.7$$

This means the point (2.3, 4.7) is **30% of the way from (2,4) to (3,4) in X** and **70% of the way from (2,4) to (2,5) in Y.**

$$\text{fade}(t) = 6t^5 - 15t^4 + 10t^3$$

fade(0.3) ≈ 0.163
fade(0.7) ≈ 0.837

Top Row Interpolation (y = 1, between dot00 and dot01)

$$N_x0 = (1-u) \cdot \text{dot00} + u \cdot \text{dot01}$$
$$= (1 - 0.163) \cdot (-0.5) + 0.163 \cdot (-0.5)$$
$$= (0.837 \times -0.5) + (0.163 \times -0.5)$$
$$= -0.4185 - 0.0815 = -0.5$$

Bottom Row Interpolation (y = 0, between dot10 and dot11)

$$N_x1 = (1-u) \cdot \text{dot10} + u \cdot \text{dot11}$$
$$= (1 - 0.163) \cdot (0) + 0.163 \cdot (0.5)$$
$$= (0.837 \times 0) + (0.163 \times 0.5)$$
$$= 0 + 0.0815 = 0.0815$$

Interpolate along x-direction

$$N = (1-v) \cdot N_x0 + v \cdot N_x1$$
$$= (1 - 0.683) \cdot (-0.5) + 0.683 \cdot (0.0815)$$
$$= (0.317 \times -0.5) + (0.683 \times 0.0815)$$
$$= -0.1585 + 0.0556$$
$$= -0.103$$

Interpolate along y-direction

Final Result

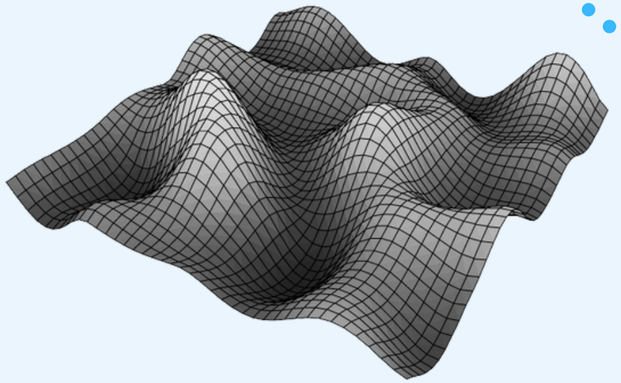
So, the final **Perlin noise value** at **(0.3, 0.6)** is **-0.103**.

This result tells us that at this point, the Perlin noise function is **negative**, meaning this location corresponds to a **low-intensity area** in a grayscale Perlin noise map.

PERLIN NOISE

KEY FEATURES

- Smooth, continuous gradient noise.
- Smooth and organic appearance that mimics natural textures and patterns

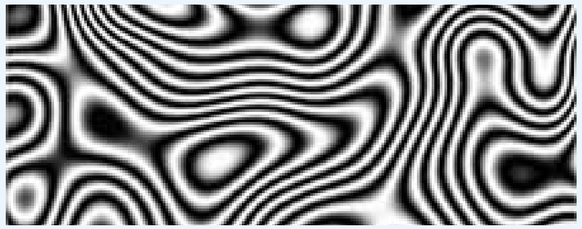


APPLICATIONS

- Terrain generation (mountains, clouds, water).
- Texture generation (wood, marble).
- Example: Rolling hills or cloud formations.

DISADVANTAGES

- Can appear "blobby" or lack sharp features.
- Computationally expensive for high dimensions.



OPTIMISATION

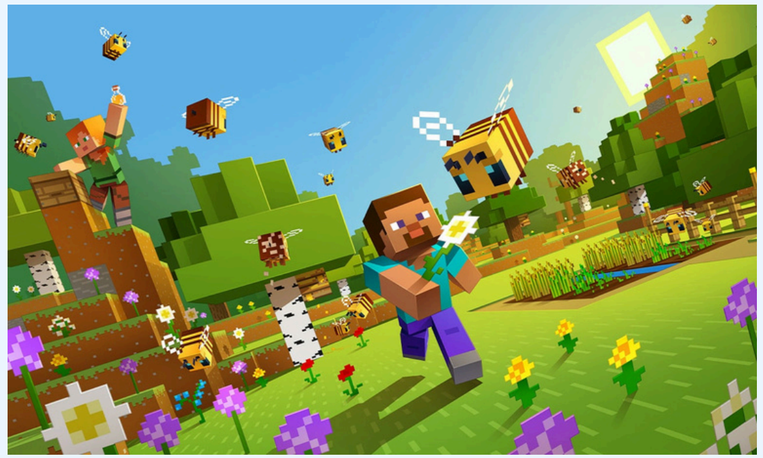
- .Use precomputed gradients.
- Limit the number of octaves in fractal noise.
- Use SIMPLEX NOISE.

Mother Nature doesn’t use a random world generator—she just smudges the pixels until they look right!" 😆

SIMPLEX NOISE

HOW IS IT DIFFERENT FROM CLASSIC PERLIN NOISE?

- It has a complexity of O(n^2) for n dimensions, compared to O(2^n) for Perlin noise
- Uses a simplex grid (triangles in 2D, tetrahedrons in 3D).
- Unlike Perlin noise, which can exhibit noticeable grid patterns, simplex noise is visually isotropic



ALGORITHM

- Coordinate Skewing: Transforming the input coordinates to align with a skewed grid, which helps in determining the simplex that contains the point.
- Simplicial Subdivision: The space is divided into simplices [triangles, tetrahedrons]
- Gradient vector selection and noise calculation by kernel summation.

WHY IS IT BETTER?

- Reduced computational complexity, hence suitable for higher dimensions.
- Minimized directional artifacts
- Smoother transitions

WHAT ARE DIMENSIONAL ARTIFACTS?

Dimensional artifacts are unwanted patterns or distortions that arise when generating procedural noise, particularly in multi-dimensional spaces

GAMES WHICH USE PERLIN AND SIMPLEX NOISE FOR RANDOM WORLD GENERATON:-

MINECRAFT, FACTORIO, TERRARIA

CELLULAR AUTOMATA

OVERVIEW

- Cells live on a 1D or multidimensional grid.
- Each cell has a state such as 0 or 1 or on and off.
- The on or off state of a cell depends on a set of rules considering the initial states of its neighbouring cells

VISUAL STYLE:-

- Blocky, grid-like patterns.
- Example: Jagged cave walls or maze-like dungeons.

APPLICATIONS:-

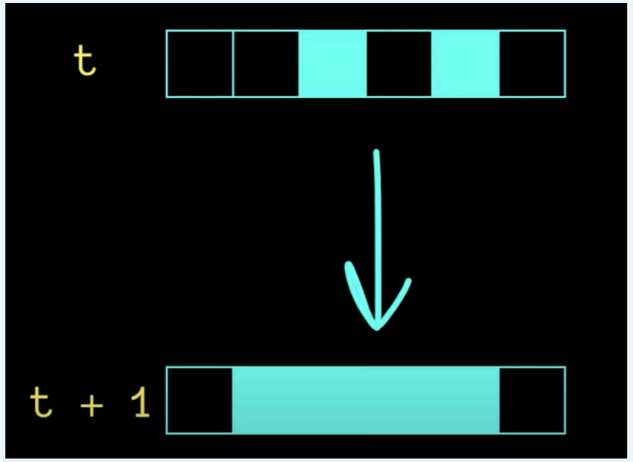
- Generate organic-looking structures like forests or coral reefs in games.
- Used in Parallel Processing, with each cell processing data individually

EXAMPLE TO DEMONSTRATE THE ALGORITHM:-

Let us take RULE 110:-

- If Inital states of cell and neighbours are 1-> Final state =0
- If Inital states of cell and neighbours are 0-> Final state =0
- If cell and right neighbour is 0 but left neighbour is 1-> Final state=0
- Else ->1

INITIAL STATE



FINAL STATE

GAMES WHICH USE CELLULAR AUTOMATA FOR RANDOM WORLD GENERATON:-

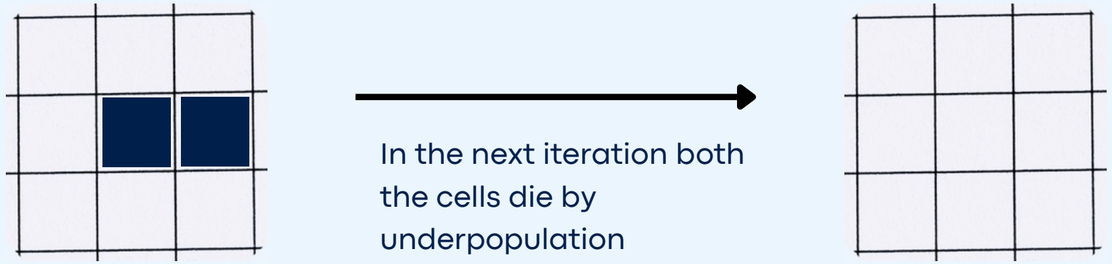
GAME OF LIFE, DWARF FORTRESS

THE GAME OF LIFE

It is a “ZERO PLAYER GAME”, devised by John Horton Conway in 1970. It is built on 2D cellular automaton in which its evolution is determined by its initial state **only**

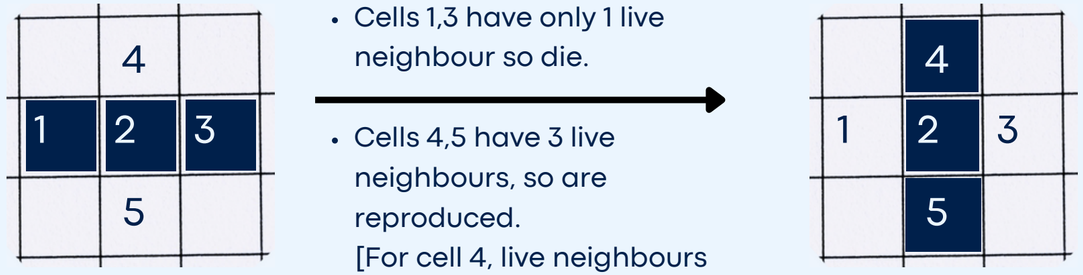
EXAMPLE SIMULATION

1.Suppose there are initially 2 live cells



In the next iteration both the cells die by underpopulation

2. If initially there are 3 horizontal live cells



- Cells 1,3 have only 1 live neighbour so die.
- Cells 4,5 have 3 live neighbours, so are reproduced.
  [For cell 4, live neighbours are 1,2,3]

RULES OF GAME OF LIFE:-

- **Any live cell with fewer than two live neighbours dies (underpopulation).**
- **Any live cell with two or three live neighbours lives on to the next generation.**
- **Any live cell with more than three live neighbours dies (overpopulation).**
- **Any dead cell with exactly three live neighbours becomes a live cell (reproduction).**

Cave Generation Example:

- Initial Randomness: Start with a grid of randomly assigned "wall" and "floor" tiles.
- Smoothing: Apply a rule that makes a cell more likely to be a "wall" if it has many "wall" neighbors, and more likely to be a "floor" if it has many "floor" neighbors.
- Iteration: Repeatedly apply the rule to smooth out the random pattern and create cave-like structures.

CAVE GENERATION USING CELLULAR AUTOMATA

1.Initialize the Grid with Random Noise:

```python
import random

# Step 1: Initialize the grid with random noise
def fill_cell_grid_with_noise(width, height, chance_of_wall):
    grid = []
    for y in range(height):  # Iterate over height first
        row = []
        for x in range(width):  # Iterate over width
            row.append(random.random() >= chance_of_wall)
        grid.append(row)
    return grid
```

Printing initial noise, where True values are printed with # and false ones with dot(.)

```
Initial Grid:
#.......#...#.....#####..
###...##...#......#.#....
......#....###..###...#..
#..#.####...#.#.#...##...
####.##.##.#.......#.#...
#.#.#..#.#..#..#.....#..#
#.....#.###..###.##.#..##
.....##.##.#######....#.#
.#..##..###....##......##
#....####..##.......##...
.....#..###.#..###.#....#
.#.##.####..#..#.#.#.#.##
..........#.#........##.#
.#..#..#.#.#.#..##.####..
.#####.....#.#...#..###..
```

2. Count no. of neighbors of every cell to apply automata rules

```python
# Step 2: Count the number of wall neighbors for a given cell
def count_wall_neighbors(cell_grid, x, y):
    width, height = len(cell_grid[0]), len(cell_grid)
    neighbors = 0
    for dy in [-1, 0, 1]:
        for dx in [-1, 0, 1]:
            if dx == 0 and dy == 0:
                continue
            x1, y1 = x + dx, y + dy
            if 0 <= x1 < width and 0 <= y1 < height and cell_grid[y1][x1]:
                neighbors += 1
    return neighbors
```

CAVE GENERATION USING CELLULAR AUTOMATA

3.Apply Cellular automata rules

```python
# Step 3: Apply the cellular automata rules to smooth the gri
def apply_rules(cell_grid):
    width, height = len(cell_grid[0]), len(cell_grid)
    num_wall_neighbors=[]
    for y in range(height):
        row = []
        for x in range(width):
            row.append(count_wall_neighbors(cell_grid, x, y))
        num_wall_neighbors.append(row)
    for y in range(height):
        for x in range(width):
            neighbors = num_wall_neighbors[y][x]
            if cell_grid[y][x] and neighbors < 4:
                cell_grid[y][x] = False
            elif not cell_grid[y][x] and neighbors >= 5:
                cell_grid[y][x] = True
    return cell_grid
```

4.Writing the print statements with dimension values

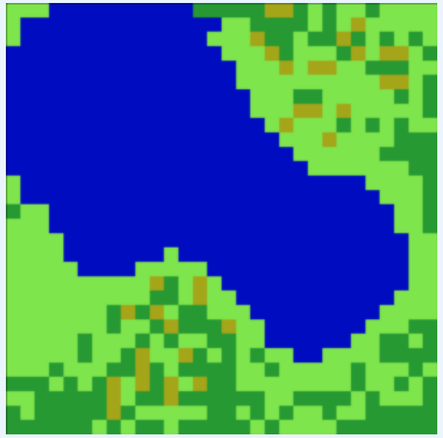
```python
# Example Usage:
width, height = 25, 15  # Grid dimensions
chance_of_wall = 0.45   # Probability of a cell starting as a wall

# Generate initial grid and apply smoothing rules
cell_grid = fill_cell_grid_with_noise(width, height, chance_of_wall)
# Apply rules multiple times to refine the cave structure
for _ in range(3):  # Adjust iterations for better smoothing
    cell_grid = apply_rules(cell_grid)

# Display the generated cave grid
print("\nFinal Grid:")
for row in cell_grid:
    line=""
    for i in row:
        if i==True:
            line+="#"
        else:
            line+="."
    print(line)
```

FINAL CAVE LAYOUT PRINTED.
AS SEEN FROM TOP(IN A MAP)

```
Final Grid:
..................##.....
................####....
..#...##........######..
.########.......#######.
..#######.......########.
.....#####.....########..
.....#####....#######....
.......###....#####......
.......##.......#........
........................
....#####...............
...########.............
...##..####.............
........###.............
........................
```



THESE MAPS CAN BE CONVERTED TO COLORED PIXELS MAPS BY APPLYING ALGORITHMS

VORONOI DIAGRAMS

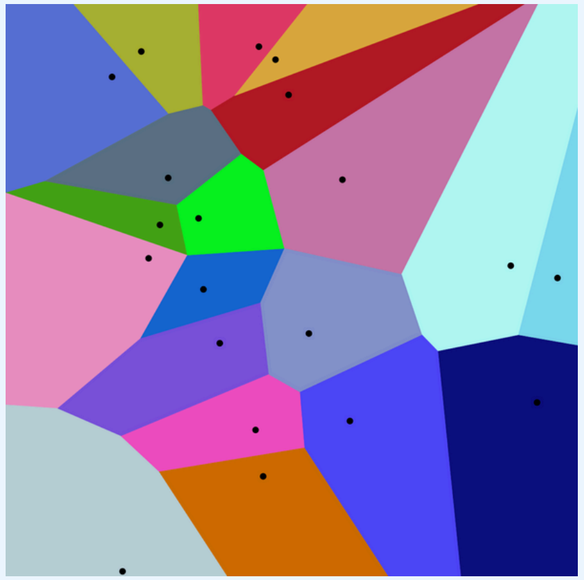
Voronoi diagram is a partitioning of a plane into regions based on the distance to a set of given points. These points, known as seeds, sites, or generators, define the structure of the diagram.

$$V(P_i) = \{x \in \mathbb{R}^2 \mid d(x, P_i) < d(x, P_j) \text{ for all } j \neq i\}$$

Each region consists of all points closer to one seed than to any other.

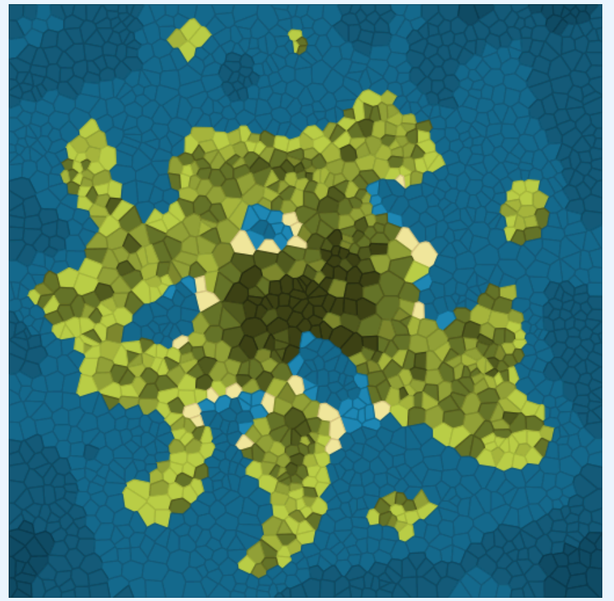
SOME OF THE COMMON ALGORITHMS:-

- Fortune's Algorithm: A plane-sweep algorithm that efficiently constructs Voronoi diagrams in time.
- Incremental Insertion: Adds points one by one, updating the diagram iteratively. Each new point is inserted, and affected regions are adjusted accordingly.
- Divide and Conquer: Recursively divides the set of points into smaller subsets, constructs Voronoi diagrams for each subset, and then merges them



VORONOI DIAGRAMS IN WORLD GENERATION:-

- Terrain Generation: Used to create diverse terrain types by assigning different features (e.g., mountains, forests) to each cell based on the seed point's properties.
- Natural patterns like river networks or geological formations
- Used in designing maps in games, with settlements, islands etc.



INCREMENTAL INSERTION

Algorithm

1. Initialize an Empty Diagram

- Begin with an empty plane where no points exist.

2. Insert the First Point

- The first point inserted occupies the entire plane as its Voronoi cell.
- Since there are no other points, no edges or boundaries are needed.

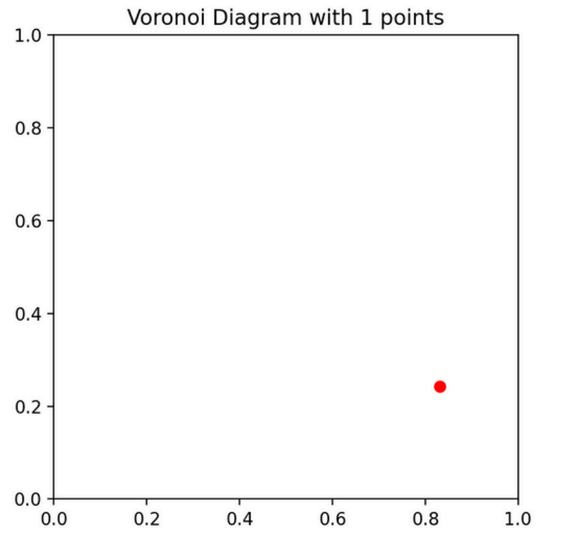
3. Insert the Second Point

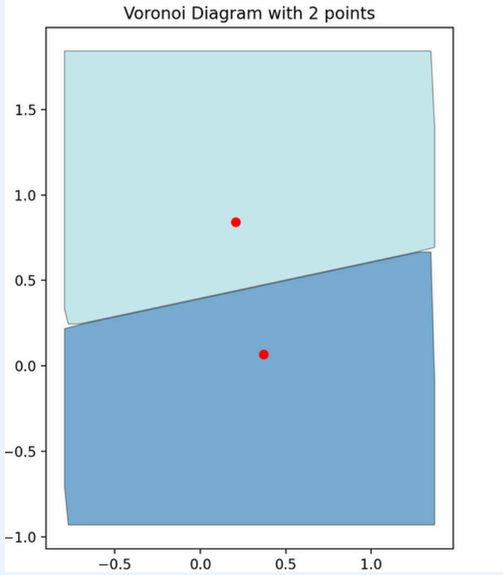
- Compute perpendicular bisector between the two points:
  a. Find the midpoint between the two points.
  b. Compute the perpendicular slope.
  c. Extend the bisector infinitely in both directions.
- This bisector divides the plane into two cells, one for each point.

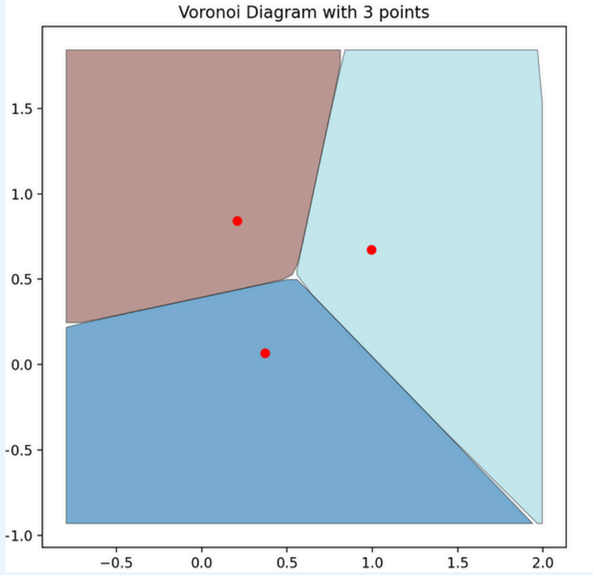
4. Insert the Third Point and Beyond

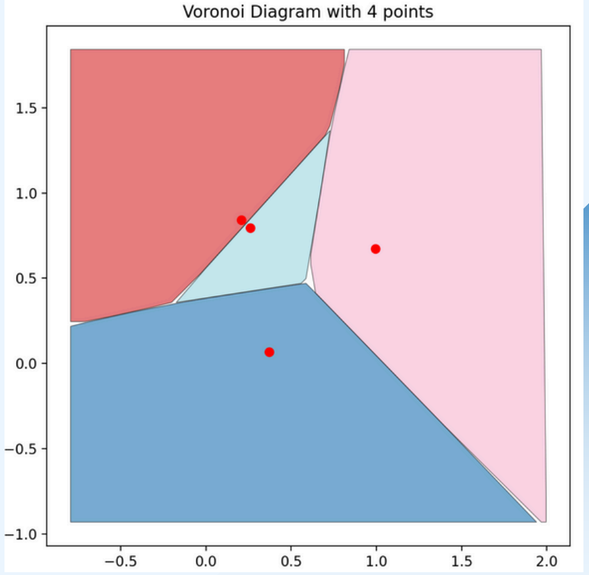
For each new point:

- Locate the nearest neighboring points.
- Compute the midpoint between the new point and each of its neighbors.
- Compute the perpendicular bisector.
- Extend the bisector until it intersects existing edges.
- Remove parts of old Voronoi cells that now belong to the new point.







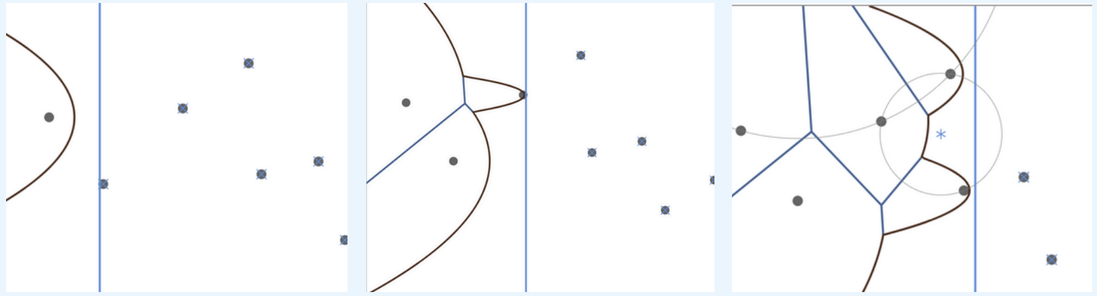


FORTUNE'S ALGORITHM

DRAWBACKS OF INCREMENTAL INSERTION

- High Cost of Updates:- Voronoi diagrams requires frequent updates to the structure as new sites are added.
- As the number of points increases, the computational overhead grows significantly.
- Special cases such as collinear points or closely spaced points can lead to numerical instability.

THUS, WE NEED A BETTER ALGORITHM
---->FORTUNE'S ALGORITHM



- Fortune's Algorithm efficiently constructs Voronoi diagrams in O(n log n) time using a sweep-line approach.
- It processes points (site events) from top to bottom while maintaining a beach line of parabolic arcs.
- When arcs disappear (circle events), Voronoi vertices are created, and edges are extended.
- The algorithm completes when all edges are formed and extended to infinity.

Advantages of Fortune's Algorithm

- Fast: Runs in O(n log n) time, making it more efficient than naive approaches.
- Handles Large Datasets: Can compute Voronoi diagrams for thousands of points efficiently.
- Robust: Works well for both finite and infinite Voronoi cells.

VISUAL CHARACTERISTICS OF VORONOI DIAGRAMS

- Consist of polygonal regions with sharp edges
- Settlement patterns, river and road network and other sharp features.

DIFFERENCES BETWEEN THE ALGORITHMS

FEATURE	PERLIN NOISE	SIMPLEX NOISE	CELLULAR AUTOMATA	VORONOI DIAGRAMS
TYPE	Gradient Noise	Gradient Noise	Rule-Based Grid Evolution	Space Partitioning
VISUAL STYLE	Smooth and continuous	Smoother and less aliasing	Discrete and abrupt changes	Sharp cell boundaries
EFFICIENCY	Moderate (computationally expensive in higher dimensions)	More efficient than Perlin Noise	Efficient for small grid sizes, can be slow for large worlds	Depends on chosen algorithm. Slow for large no. of seed pts.
OPTIMIZATION TECHNIQUES	• Precompute noise in chunks • Use octaves for smoother transitions	Use lookup tables for faster computation	• Use multi-threading • Apply early stopping	Optimize seed distribution to avoid clustering
USE CASES	Terrain generation, clouds, textures	Terrain generation	Cave systems, map generation	Biomes, city layouts, natural formations

CONCLUSION

OFTEN THESE ALGORITHMS ARE USED TOGETHER TO OBTAIN THE DESIRED WORLD IN GAMES

TERRAIN GENERATION IN GAMES (MINECRAFT-STYLE WORLDS)

- Perlin/Simplex Noise: Used for generating base terrain elevation (hills, mountains, valleys).
- Voronoi Diagrams: Used for dividing the world into distinct areas like forests, deserts,etc.
- Cellular Automata: Used for cave generation, ensuring caves are connected and have natural-looking shapes.

CITY GENERATION (SIMCITY-STYLE GAMES)

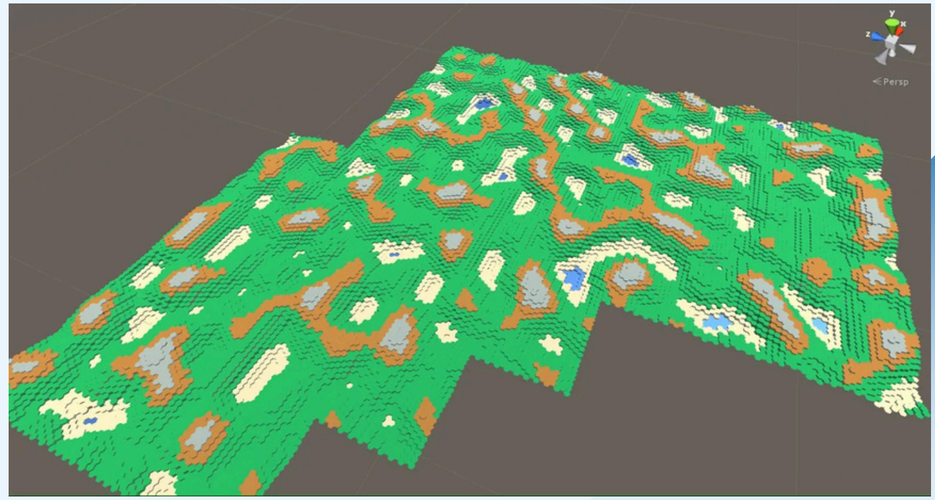
- VORONOI DIAGRAMS: HELPS CREATE ROAD NETWORKS AND DISTRICT LAYOUTS.
- PERLIN NOISE: USED FOR ELEVATION CHANGES AND TERRAIN SHAPING.
- CELLULAR AUTOMATA: SIMULATES URBAN EXPANSION AND BUILDING PLACEMENT.

Dungeons & Rogue-like Maps

- Perlin Noise: Used for large-scale dungeon layouts with smooth pathways.
- Cellular Automata: Creates cave-like rooms with natural formations.
- Voronoi Diagrams: Used to ensure varied room shapes and sizes.

SUMMING-UP

Procedural world generation is a powerful tool that blends multiple algorithms to create rich, dynamic environments. Techniques like Perlin noise, simplex noise, Voronoi diagrams, and cellular automata each bring unique characteristics, and when used together, they enable more realistic and diverse landscapes.



THANK YOU!