Linked List

Linked List

A list of nodes each node has 2 fields: data and pointer. The pointer points to another node, next or previous. So linked list can be single way, double way, circular or even spatial.
Link

Challenge:
Implement a generic linked list, means the data is a generic type. So you can create this list with any type you want.
Implementation consist of creating a node class and a linked list class collecting the nodes, as well as the ability to insert after/before nodes and remove nodes.

Nice application of the challenge:
Instantiate a linked list of type Action, so you can create a list of actions, means a precise ordered list of commands.

Binary Tree

Binary Tree

A list of nodes each node has 3 fields: data and 2 pointers. The pointers point to another nodes, left and right. Binary Tree makes searching data very fast. The speed of search come from the structured nature of the data.
How structured data is faster for search
How to traverse breadth first and depth first

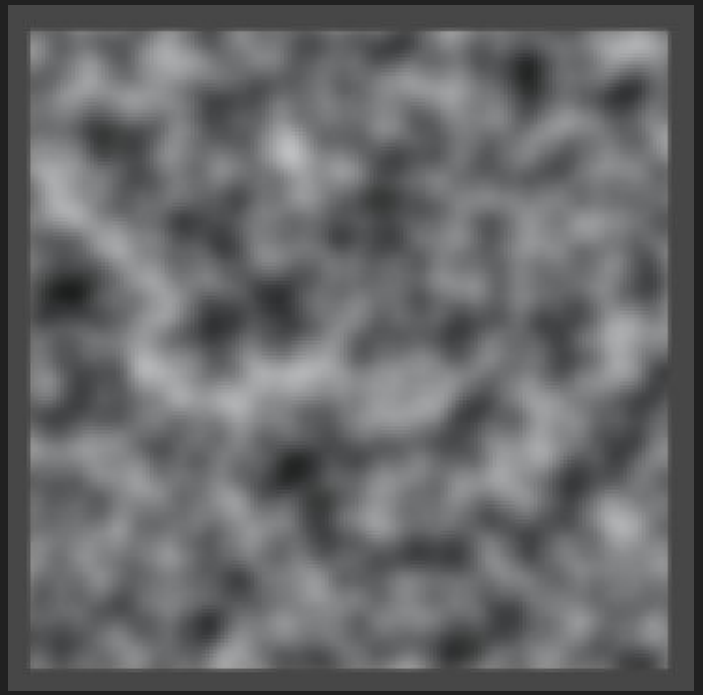
Challenge:
Implement a int binary tree with the rules smaller to the left and bigger to the right.
Implement as well a traverse breadth first and a travel depth first methods.
Search a number in the tree and count the number of iterations consumed to find the number.

Perlin Noise

Perlin Noise

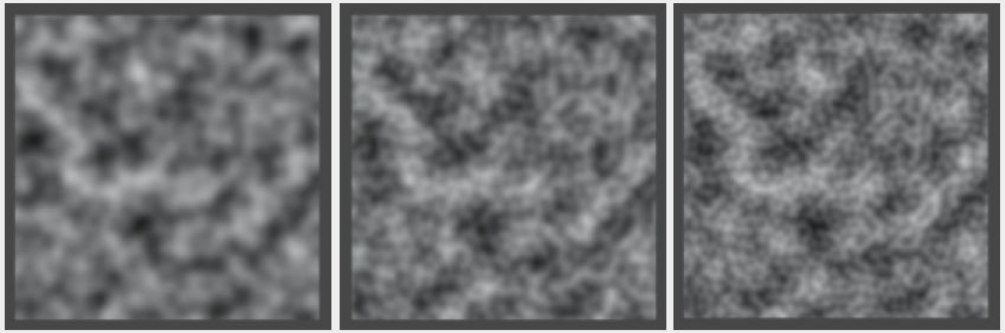
A gradient fractal noise, procedurally generates organic like textures.

Perlin build from scratch with CatLikeCoding.

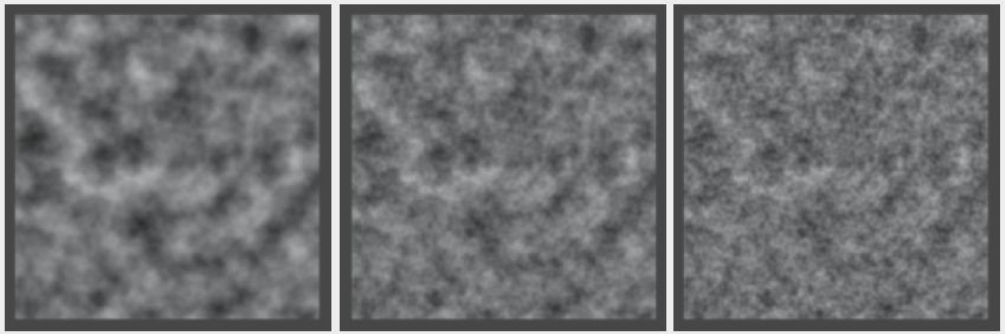


Perlin Noise

In our code we did not build it from scratch instead we used Sebastian Lague implementation. Where we apply a fractal logic on a Unity built-in simple Perlin noise:



Two octaves of Perlin noise with base frequency 8, using lacunarity 2, 3, and 4.



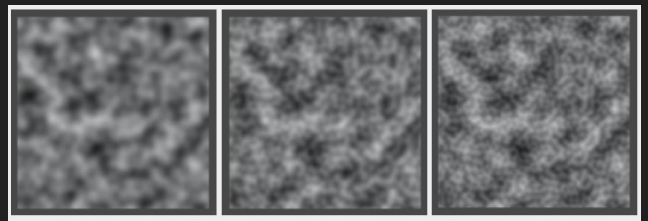
Three octaves of Perlin noise with base frequency 8, using persistence ½, ¾, and 1.

```
for (int y = 0; y < height; y++)
{
    for (int x = 0; x < width; x++)
    {
        float amplitude = 1f;
        float frequency = 1f;
        float noiseHeight = 0f;

        for (int o = 0; o < octaves; o++)
        {
            float sampleX = x / scale * frequency;
            float sampleY = y / scale * frequency;

            float perlinValue = Mathf.PerlinNoise(
                sampleX, sampleY) * 2f - 1f;
            noiseHeight += perlinValue * amplitude;

            amplitude *= persistance;
            frequency *= lacunarity;
        }
```

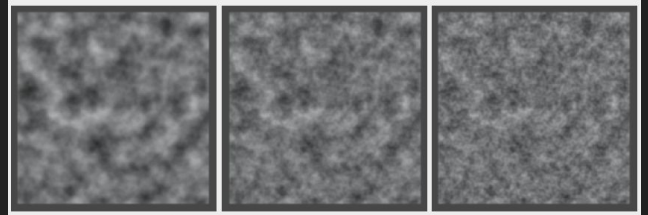
Perlin Noise

- Octaves: Number of loops to achieve a fractal effect.
- Persistance: Amount of “surviving” of higher frequencies inside higher octaves.
- Lacunarity: Amount of frequency growth for higher octaves.

Higher frequency means smaller details.



Two octaves of Perlin noise with base frequency 8, using lacunarity 2, 3, and 4.



Three octaves of Perlin noise with base frequency 8, using persistence ½, ¾, and 1.

```
for (int y = 0; y < height; y++)
{
    for (int x = 0; x < width; x++)
    {
        float amplitude = 1f;
        float frequency = 1f;
        float noiseHeight = 0f;

        for (int o = 0; o < octaves; o++)
        {
            float sampleX = x / scale * frequency;
            float sampleY = y / scale * frequency;

            float perlinValue = Mathf.PerlinNoise(
                sampleX, sampleY) * 2f - 1f;
            noiseHeight += perlinValue * amplitude;

            amplitude *= persistance;
            frequency *= lacunarity;
        }
```