3D Perlin noise for clouds generation

Giuseppe Massimiliano Carugno

Contents

1 Abstract

Perlin noise is a popular computer graphics algorithm used to procedurally generate realistic landscapes. The focus of this project is to implement a three-dimensional version of the Perlin noise to produce a realistic cloud shape structure. The tools used for this project are Unity and VisualStudio. The coding language is C# and all simulation's assets are basic Unity 3D objects. The following document is an in depth explanation of the Perlin noise algorithm and its implementation.

2 Index

2.1 Inputs

We provide some starting values to define the size, the cloudiness and the regularity of the cloud.

2.1.1 Random seed

Random seed used to set the initial state of the Unity random data generator, that otherwise is randomly initialized, so that each time it generates different random vertices.

```
//Random seed
public int randomSeed = 0;
```

2.1.2 Dimensions

Variables x, y and z define the three-dimensional space that we are going to use as sky.

```
//3D dimensions of the sky
[Min(0)]
public int x = 0;
[Min(0)]
public int y = 0;
[Min(0)]
public int z = 0;
```

2.1.3 Cloudiness

Cloudiness represents the probability of a cloud to spawn. The lower bound 0 is used to generate a clear sky, meanwhile the upper bound 1 means that each cloud spawns, until a total of $x * y * z$ clouds, defined previously as maximum sky size, resulting in a completely clouded sky.

```
//Cloudiness
[Range(0f, 1f)]
public float cloudiness = 1f;
```

2.1.4 Normalization and scaling

Since the Perlin noise algorithm returns noise values that are weighted combined contributions of the arrays on the vertices of each unit cube to a gradient, these values may be outside the constraints provided by the dimensions x, y and z. Setting both flags to true normalizes noise values in the range between the minimum and maximum noise values recorded and then scales them between 0 and the maximum value allowed by the y dimension, where y in Unity is the vertical axis. We can then spawn a cloud using this value as height, while also ensuring to remain inside the defined sky. Below is the complete mathematical formula:

$$noise_{scaled} = \frac{(noise - noise_{min})}{(noise_{max} - noise_{min})} \times (y_{max} - y_{min}) + y_{min}$$

```
//Restrict to sky
public bool normalize = true;
public bool scale = true;
```

2.1.5 Side offset

Since we use noise values to randomize only the y coordinate, which is the height, all clouds spawned have x and z sides lying on integer values of coordinates of the horizontal matrix [x, z]. This is most relevant along the horizontal surfaces where the final cloud will appear flat. Setting this flag to true adds a random offset between 0 and 1 to the x and z coordinates of each cloud so that the horizontal surfaces become rough and the final cloud is more realistic.

```
//Roughen lateral sides
public bool sideOffset = false;
```

2.1.6 Cloud

Unity asset to use as cloud.

```
//Cloud asset
public GameObject cloud;
```

2.2 Functions

Here are listed and described the different functions used.

2.2.1 Initialize random cube vertices

Receives an empty three-dimensional matrix of three-dimensional arrays and iterates over all its cells. For each cell the function generates and stores a new random three-dimensional array with the tail on space coordinates equal the indexes of the cell and the head within a sphere whose center is the tail and whose radius is 1.

```
//Initialize random vertices on sky's unit cubes vertices
private Vector3[,,] initRandomVertices(Vector3[,,] perlinGrid)
    {
        for (int i = 0; i < perlinGrid.GetLength(0); i++)
        {
            for (int j = 0; j < perlinGrid.GetLength(1); j++)
            {
                for (int k = 0; k < perlinGrid.GetLength(2); k++)
                {
                    perlinGrid[i, j, k] = perlinGrid[i, j, k] +
                        Random.insideUnitSphere;
                }
```

```
            }
        }

        return perlinGrid;

    }
```

2.2.2 Perlin noise algorithm

This function is the Perlin noise algorithm. It receives as inputs the 3D coordinates of a point in the available space and the three-dimensional matrix with random generated three-dimensional arrays. For each one of the three coordinates the smallest nearest integer and the biggest nearest integer are stored. We end up with 6 values, 2 for each dimension, and we use them to identify the three-dimensional unit cube surrounding the input point and its 8 vertices. Then we compute the dot product between the array saved in the matrix at the cell with indexes equal to the coordinates of 1 of the 8 vertices and a new array going from the same vertex to the input coordinates. We repeat this formula for each one of the 8 vertices. Then we start to linearly interpolate this 8 values in ordered pairs. We linearly interpolate two values together if and only if they are from two different dot products whose first arrays were sharing two identical coordinates of the unit cube and as interpolation value we use the result of the Perlin noise interpolation function applied to the only coordinate that they're not sharing. We iterate this process for all 8 dot products results until we get one final value. This final value is the Perlin noise value for the input position, which accounts the combined weighted contributions to the gradient of the random arrays on the vertices of the current unit cube.

```
//Compute Perlin noise
private float perlinNoise(int i, int j, int k, Vector3[,,]
    perlinGrid)
    {

        //Phase 1
        //Find surrounding unit cube vertices
        int X0 = Mathf.Clamp(Mathf.FloorToInt(perlinGrid[i, j,
            k].x), 0, x);
        int X1 = Mathf.Clamp(Mathf.CeilToInt(perlinGrid[i, j,
            k].x), 0, x);

        int Y0 = Mathf.Clamp(Mathf.FloorToInt(perlinGrid[i, j,
            k].y), 0, y);
        int Y1 = Mathf.Clamp(Mathf.CeilToInt(perlinGrid[i, j,
            k].y), 0, y);

        int Z0 = Mathf.Clamp(Mathf.FloorToInt(perlinGrid[i, j,
            k].z), 0, z);
        int Z1 = Mathf.Clamp(Mathf.CeilToInt(perlinGrid[i, j,
            k].z), 0, z);
```

```
//Phase 2
//Start permutations

//Fix X0 and permute Y and Z
float X0Y0Z0 = Vector3.Dot(perlinGrid[X0, Y0, Z0], new
    Vector3(perlinGrid[i, j, k].x, perlinGrid[i, j, k].y,
    perlinGrid[i, j, k].z) - new Vector3(X0, Z0, Y0));
float X0Y0Z1 = Vector3.Dot(perlinGrid[X0, Y0, Z1], new
    Vector3(perlinGrid[i, j, k].x, perlinGrid[i, j, k].y,
    perlinGrid[i, j, k].z) - new Vector3(X0, Z1, Y0));
float X0Y1Z0 = Vector3.Dot(perlinGrid[X0, Y1, Z0], new
    Vector3(perlinGrid[i, j, k].x, perlinGrid[i, j, k].y,
    perlinGrid[i, j, k].z) - new Vector3(X0, Z0, Y1));
float X0Y1Z1 = Vector3.Dot(perlinGrid[X0, Y1, Z1], new
    Vector3(perlinGrid[i, j, k].x, perlinGrid[i, j, k].y,
    perlinGrid[i, j, k].z) - new Vector3(X0, Z1, Y1));

//Fix X1 and permute Y and Z
float X1Y0Z0 = Vector3.Dot(perlinGrid[X1, Y0, Z0], new
    Vector3(perlinGrid[i, j, k].x, perlinGrid[i, j, k].y,
    perlinGrid[i, j, k].z) - new Vector3(X1, Z0, Y0));
float X1Y0Z1 = Vector3.Dot(perlinGrid[X1, Y0, Z1], new
    Vector3(perlinGrid[i, j, k].x, perlinGrid[i, j, k].y,
    perlinGrid[i, j, k].z) - new Vector3(X1, Z1, Y0));
float X1Y1Z0 = Vector3.Dot(perlinGrid[X1, Y1, Z0], new
    Vector3(perlinGrid[i, j, k].x, perlinGrid[i, j, k].y,
    perlinGrid[i, j, k].z) - new Vector3(X1, Z0, Y1));
float X1Y1Z1 = Vector3.Dot(perlinGrid[X1, Y1, Z1], new
    Vector3(perlinGrid[i, j, k].x, perlinGrid[i, j, k].y,
    perlinGrid[i, j, k].z) - new Vector3(X1, Z1, Y1));

//Phase 3
//Use Perlin function to get interpolation values
float perlinX = fadeByPerlin((perlinGrid[i, j, k].x - X0));
float perlinY = fadeByPerlin((perlinGrid[i, j, k].y - Y0));
float perlinZ = fadeByPerlin((perlinGrid[i, j, k].z - Z0));

//Phase 4
//Start interpolating to get the vertices weighted
    contributions

//Fix X0 and Y then interpolate along Z
float X0Y0lerpedZ = Mathf.Lerp(X0Y0Z0, X0Y0Z1, perlinZ);
float X0Y1lerpedZ = Mathf.Lerp(X0Y1Z0, X0Y1Z1, perlinZ);
//Fix X0 then lerp along Y
float X0lerpedYZ = Mathf.Lerp(X0Y0lerpedZ, X0Y1lerpedZ,
    perlinY);

//Fix X1 and Y and interpolate along Z
```

```
        float X1Y0lerpedZ = Mathf.Lerp(X1Y0Z0, X1Y0Z1, perlinZ);
        float X1Y1lerpedZ = Mathf.Lerp(X1Y1Z0, X1Y1Z1, perlinZ);
        //Then X1 then interpolate along Y
        float X1lerpedYZ = Mathf.Lerp(X1Y0lerpedZ, X1Y1lerpedZ,
            perlinY);

        //Finally interpolate along X
        float lerpedXYZ = Mathf.Lerp(X0lerpedYZ, X1lerpedYZ,
            perlinX);
        //This is the resulting value -> Perlin noise

        //Return noise value
        return lerpedXYZ;

    }
```

2.2.3 Perlin noise interpolation function

This is the improved version of the Perlin noise interpolation function used to smooth the different noise values. It was presented in the research paper "Improving Noise" of July 2002 by Kenneth Perlin himself:

$$f(t) = 6t^5 - 15t^4 + 10t^3$$

```
//Perlin noise interpolation function by Perlin himself
private float fadeByPerlin(float t)
    {
        return 6 * Mathf.Pow(t, 5) - 15 * Mathf.Pow(t, 4) + 10 *
            Mathf.Pow(t, 3);
    }
```

2.2.4 Normalization function

Function that normalizes a value inside an interval described by a minimum value and a maximum value. Below is the normalization formula:

$$noise_{normalized} = \frac{(noise - noise_{min})}{(noise_{max} - noise_{min})}$$

```
//Normalization function
private float normalizeNoise(float noiseToNormalize, float min,
    float max)
    {
        return (noiseToNormalize - min) / (max - min);
    }
```

2.2.5 Scaling function

Function that scales a value inside an interval described by a minimum value and a maximum value. Below is the scaling formula:

$$noise_{scaled} = noise_{normalized} \times (y_{max} - y_{min}) + y_{min}$$

```
//Scaling function
private float scaleNoise(float noiseToScale, float min, float max)
    {
        return (noiseToScale * (max - min) + min);
    }
```

2.2.6 Start function

This is the main function and it is run only once so the cloud generated is static. It uses two three-dimensional matrices initialized with same size defined by the three dimensional variables: the first one is filled with random generated three-dimensional arrays and the second one will be used to store noise values. The two float variables track the minimum and the maximum noise values recorded, later maybe used to normalize and scale noise values. The first loop fills the second matrix with the noise values generated using the Perlin noise algorithm and updates the minimum and maximum noise values. The second loop handles the spawn of the clouds, according to the chosen starting variables. When the function is over the full cloud is spawned.

```
//Main function
void Start()
    {

        if (randomSeed == 0) randomSeed =
            (int)System.DateTime.Now.Ticks;
        Random.InitState(randomSeed);

        Vector3[,,] perlinGrid = new Vector3[x, y, z];
        float[,,] noiseValues = new float[x, y, z];

        perlinGrid = initRandomVertices(perlinGrid);

        float maxNoise = 0f;
        float minNoise = 0f;

        for (int i = 0; i < x; i++)
        {
            for (int j = 0; j < y; j++)
            {
                for (int k = 0; k < z; k++)
                {

                    float noise = perlinNoise(i, j, k, perlinGrid);

                    if (noise < minNoise) minNoise = noise;
                    if (noise > maxNoise) maxNoise = noise;
```

```
                //Noise is the new height!
                noiseValues[i, j, k] = noise;

            }
        }
    }

    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < y; j++)
        {
            for (int k = 0; k < z; k++)
            {

                if (Random.value > cloudiness) continue;

                float finalNoise = noiseValues[i, j, k];

                if (normalize == true) finalNoise =
                    normalizeNoise(finalNoise, minNoise,
                    (maxNoise));

                if (scale == true) finalNoise =
                    scaleNoise(finalNoise, 0, y);

                if (sideOffset == true && (i == 0 || i == x - 1
                    || k == 0 || k == z - 1))
                {
                    Vector2 offset = Random.insideUnitCircle;
                    Instantiate(cloud, new Vector3(i +
                        offset.x, finalNoise, k + offset.y),
                        Quaternion.identity);
                } else
                    Instantiate(cloud, new Vector3(i,
                        finalNoise, k), Quaternion.identity);

            }
        }
    }

}
```