UNIVERSITÀ DEGLI STUDI
DI MILANO

Procedural Content Generation

Part 2

A.I. for Video Games

Davie Gadia (gadia@di.unimi.it) and Dario Maggiorini (dario@di.unimi.it)

Creating a Landscapes

- Another useful application of PCG is the creation of landscapes
- In general, this is about generating an heightmap for a terrain
  - An heightmap is a black and white image, where black means lowest and white means maximum height
- A complete random terrain is easy to do … but also totally useless

Unless You Need This!

Random (Useless) Terrain in Unity

Source: RandomTerrain
Folder: PCG

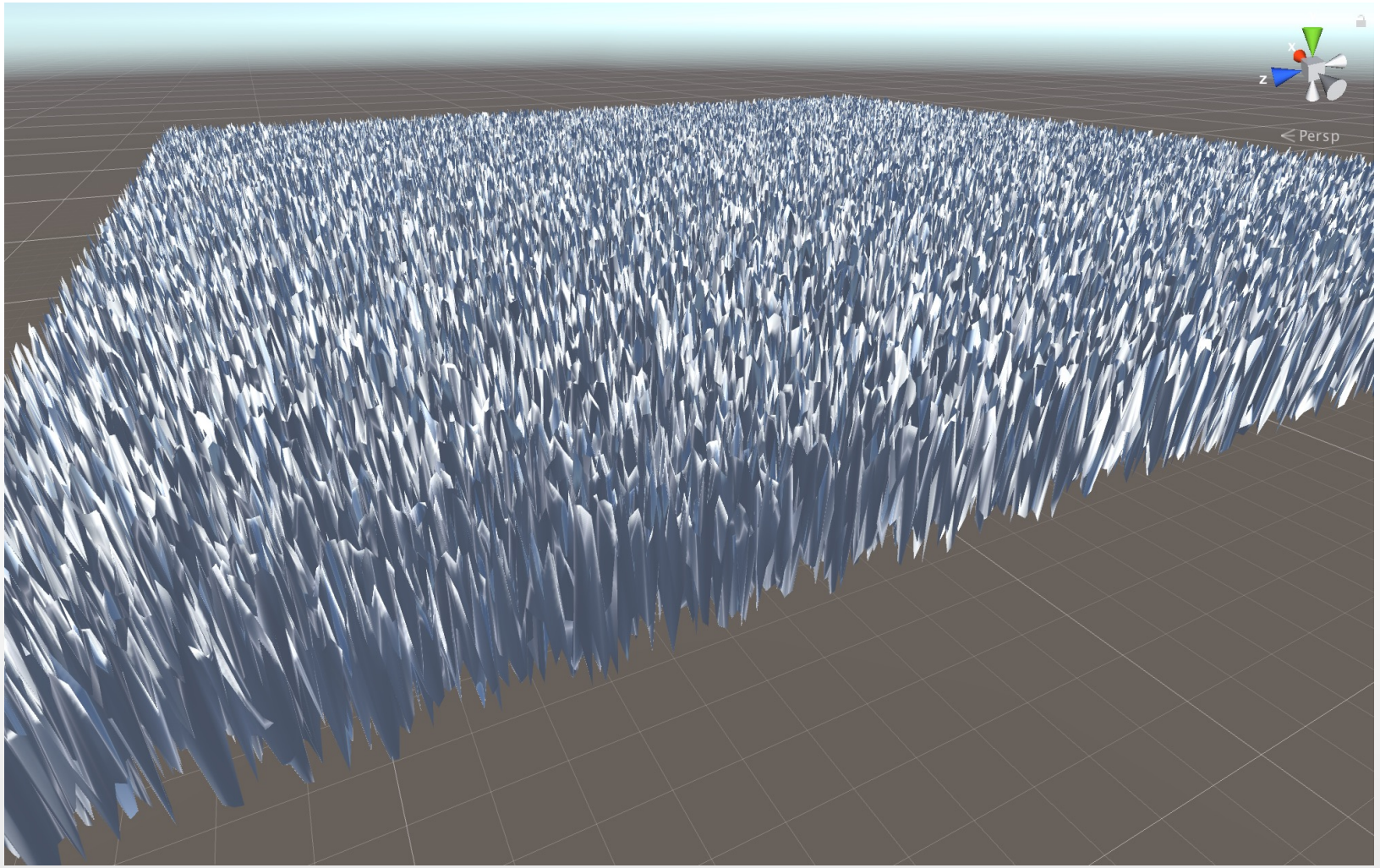
```
[RequireComponent(typeof(Terrain))]

public class RandomTerrain : MonoBehaviour {

    public bool makeItFlat = false;

    void Start () {
        Terrain t = GetComponent<Terrain> ();
        TerrainData td = t.terrainData;
        int x = td.heightmapWidth;
        int y = td.heightmapHeight;
        float [,] h = new float[y, x];

        for (int i = 0; i < x; i += 1) {
            for (int j = 0; j < y; j += 1) {
                h [j, i] = makeItFlat ? 0f : Random.Range (0f, 1f);
            }
        }

        td.SetHeights (0, 0, h);
    }

}
```

When this flag is set to true, the terrain will be made flat instead of random (more on this in the next slides)

A terrain gameobject is a matrix of points. Setting the height of each point will modify the shape of the canopy in the scene

And the Predictable Result is …

Scene: Random
Folder: PCG



Beware of the Terrain

- Terrain is some kind of anomalous gameobject

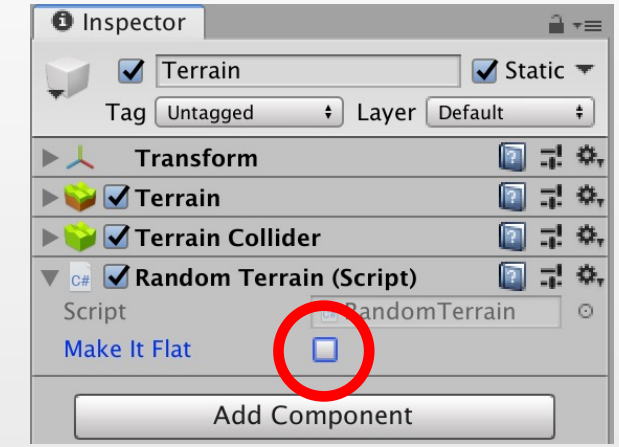
1. It is extremely huge
   - By default 500x500 meters

Not the actual word
I would like to use

2. It does not scale like other objects
   - Changing the scale value in the transform component is not doing anything

3. The geometric reference is in a corner

4. It is an asset
   - So, every change you make to its structure will be in the asset database ... and will persist between runs!
     - That is why I had to define the “make it flat” flag



A Better Random Generation

- The problem with our Devil's Golf Course is about randomizing each sample independently
  - In nature, this is statistically improbable

- In real terrain, heights at different points on the terrain are not independent of each other: the elevation in a specific point on the earth's surface is statistically related to the elevation in nearby points

Interpolated Random Generation

- We can randomly pick some equally spaced points to create a lattice over the terrain
  Then, all other points will be interpolated to get a gentle slope
  - Bilinear interpolation
    - We interpolate linearly in one direction and then in the other
    - Peaks will be spikes and slopes will be flat
  - Bicubic interpolation
    - Same as before (in the sense of “bi”) but we introduce a *slope function* to make it non-linear (“S” shaped, if possible)
    - A typical slope formula is $slope(x) = -2x^3 + 3x^2$

Bilinear Interpolation

Source: BilinearTerrain
Folder: PCG

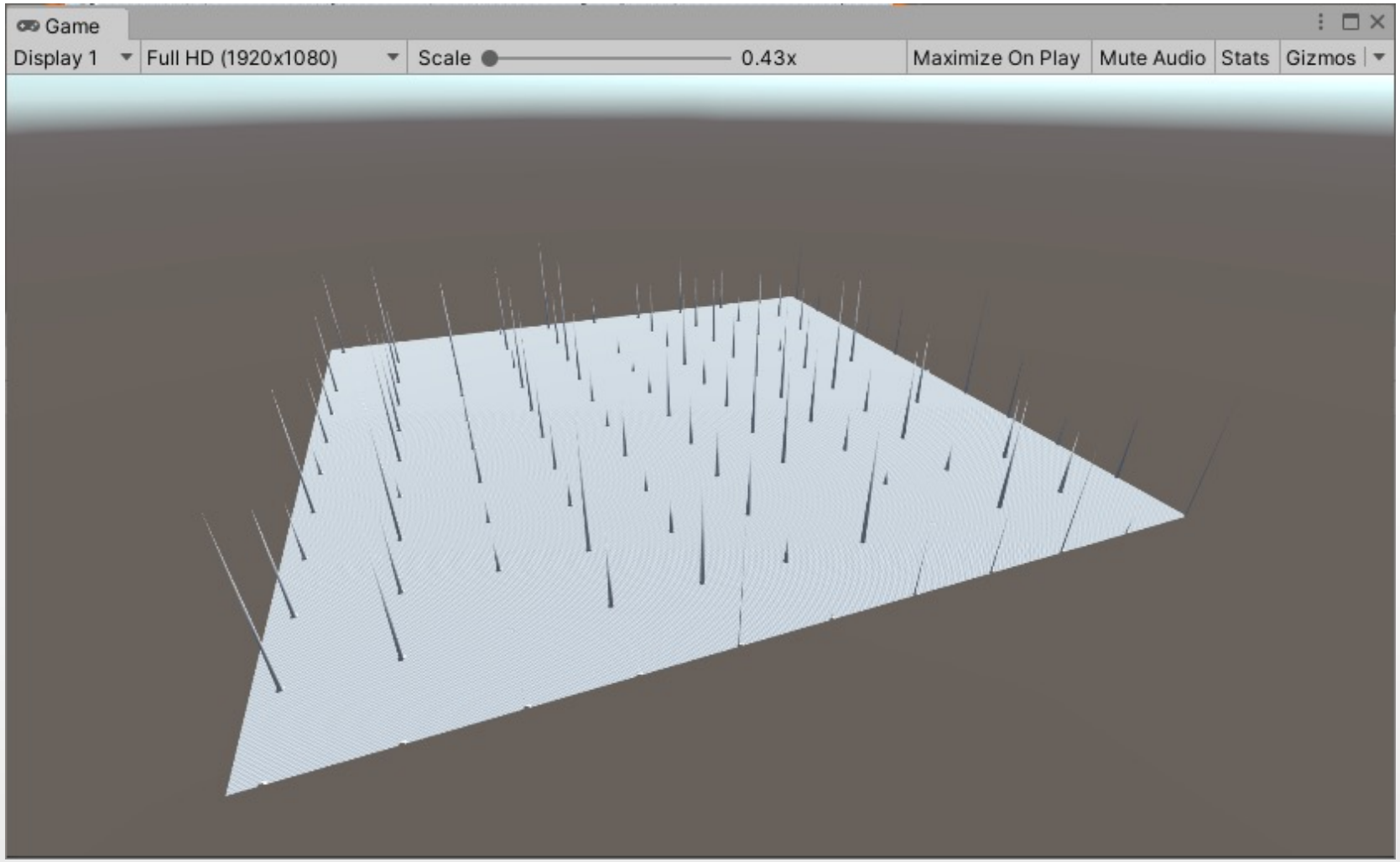
```
float [,] h = new float [y, x];

// build a lattice
for (int i = 0; i < x; i += subsampling) {
    for (int j = 0; j < y; j += subsampling) {
        h [j, i] = Random.Range (0f, 1f);
    }
}

// cut out border effects to make code simpler
int xCut = subsampling * ((int) Mathf.Floor (x / subsampling));
int yCut = subsampling * ((int) Mathf.Floor (y / subsampling));

// build bilinear interpolations
// first direction i only on lattice points
for (int i = 0; i < xCut; i += subsampling) { // avoid border effect
    for (int j = 0; j <= yCut; j += subsampling) {
        for (int k = 1; k < subsampling; k += 1) {
            h [j, i + k] = Mathf.Lerp (h [j, i], h [j, i + subsampling], (float)k / (float)subsampling);
        }
    }
}
// then direction j on all points to cover all grid
for (int i = 0; i <= xCut; i += 1) {
    for (int j = 0; j < yCut; j += subsampling) {
        for (int k = 1; k < subsampling; k += 1) {
            h [j + k, i] = Mathf.Lerp (h [j, i], h [j + subsampling, i], (float)k / (float)subsampling);
        }
    }
}

td.SetHeights (0, 0, h);
```

The version presented in this slide is available as a “standalone” script in the Random scene (you can manually activate this script or the RandomTerrain shown before)

Another version, slightly more complex, is present in the repository in the InterpolatedTerrain script, where you can select the interpolation from the interface

Setting the Samples

Scene: StepsTerrain
Folder: PCG



Bilinear Interpolation

Source: BilinearTerrain
Folder: PCG

```
float [,] h = new float [y, x];

// build a lattice
for (int i = 0; i < x; i += subsampling) {
    for (int j = 0; j < y; j += subsampling) {
        h [j, i] = Random.Range (0f, 1f);
    }
}

// cut out border effects to make code simpler
int xCut = subsampling * ((int) Mathf.Floor (x / subsampling));
int yCut = subsampling * ((int) Mathf.Floor (y / subsampling));

// build bilinear interpolations
// first direction i only on lattice points
for (int i = 0; i < xCut; i += subsampling) { // avoid border effect
    for (int j = 0; j <= yCut; j += subsampling) {
        for (int k = 1; k < subsampling; k += 1) {
            h [j, i + k] = Mathf.Lerp (h [j, i], h [j, i + subsampling], (float)k / (float)subsampling);
        }
    }
}
// then direction j on all points to cover all grid
for (int i = 0; i <= xCut; i += 1) {
    for (int j = 0; j < yCut; j += subsampling) {
        for (int k = 1; k < subsampling; k += 1) {
            h [j + k, i] = Mathf.Lerp (h [j, i], h [j + subsampling, i], (float)k / (float)subsampling);
        }
    }
}

td.SetHeights (0, 0, h);
```

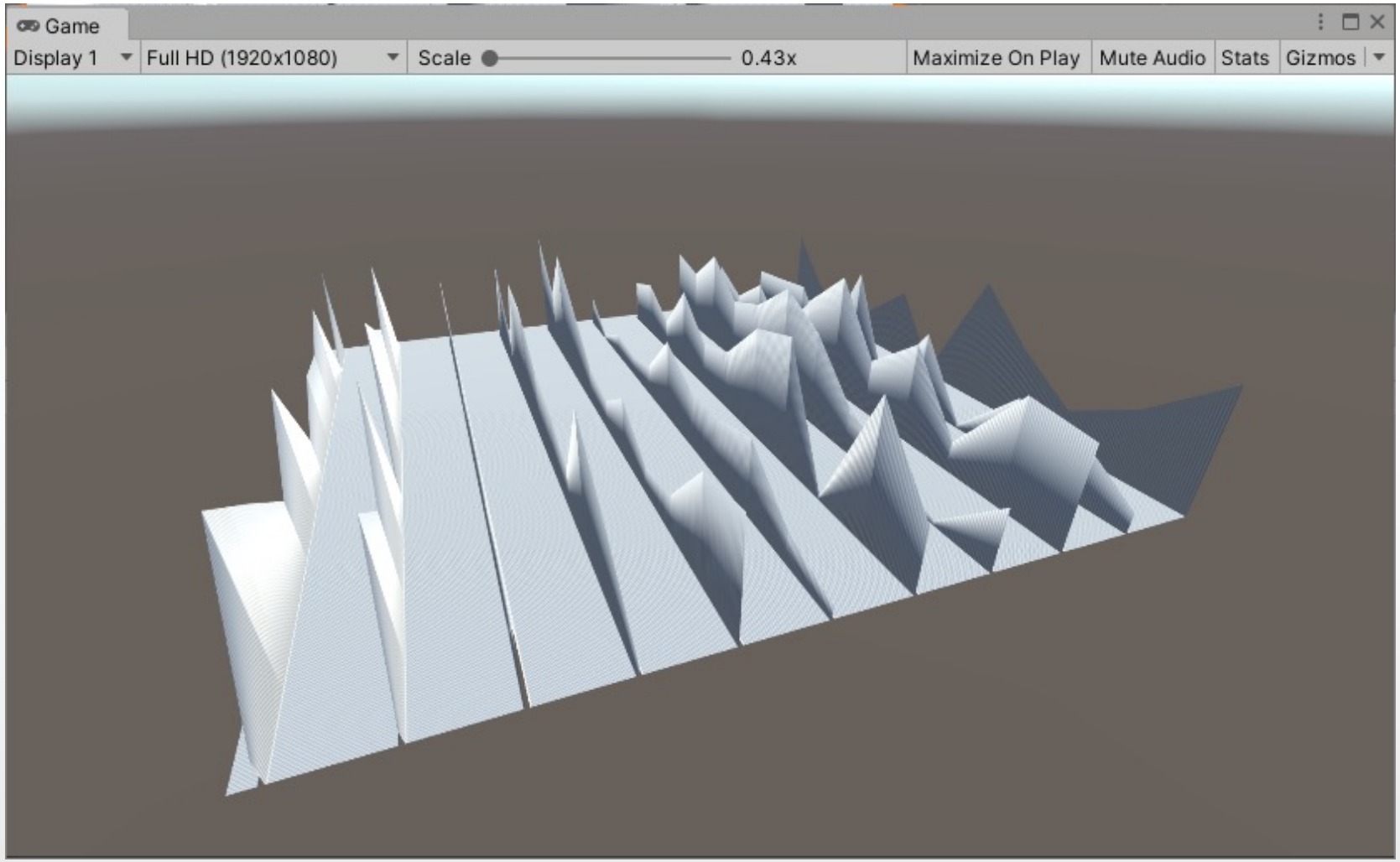
k will interpolate along the y axis all points belonging to lines passing over lattice points

The version presented in this slide is available as a “standalone” script in the Random scene (you can manually activate this script or the RandomTerrain shown before)

Another version, slightly more complex, is present in the repository in the InterpolatedTerrain script, where you can select the interpolation from the interface

Interpolating Along One Way

Scene: StepsTerrain
Folder: PCG



Bilinear Interpolation

Source: BilinearTerrain
Folder: PCG

```
float [,] h = new float [y, x];

// build a lattice
for (int i = 0; i < x; i += subsampling) {
    for (int j = 0; j < y; j += subsampling) {
        h [j, i] = Random.Range (0f, 1f);
    }
}

// cut out border effects to make code simpler
int xCut = subsampling * ((int) Mathf.Floor (x / subsampling));
int yCut = subsampling * ((int) Mathf.Floor (y / subsampling));

// build bilinear interpolations
// first direction i only on lattice points
for (int i = 0; i < xCut; i += subsampling) { // avoid border effect
    for (int j = 0; j <= yCut; j += subsampling) {
        for (int k = 1; k < subsampling; k += 1) {
            h [j, i + k] = Mathf.Lerp (h [j, i], h [j, i + subsampling], (float)k / (float)subsampling);
        }
    }
}
// then direction j on all points to cover all grid
for (int i = 0; i <= xCut; i += 1) {
    for (int j = 0; j < yCut; j += subsampling) {
        for (int k = 1; k < subsampling; k += 1) {
            h [j + k, i] = Mathf.Lerp (h [j, i], h [j + subsampling, i], (float)k / (float)subsampling);
        }
    }
}

td.SetHeights (0, 0, h);
```

k will interpolate along the y axis all points belonging to lines passing over lattice points

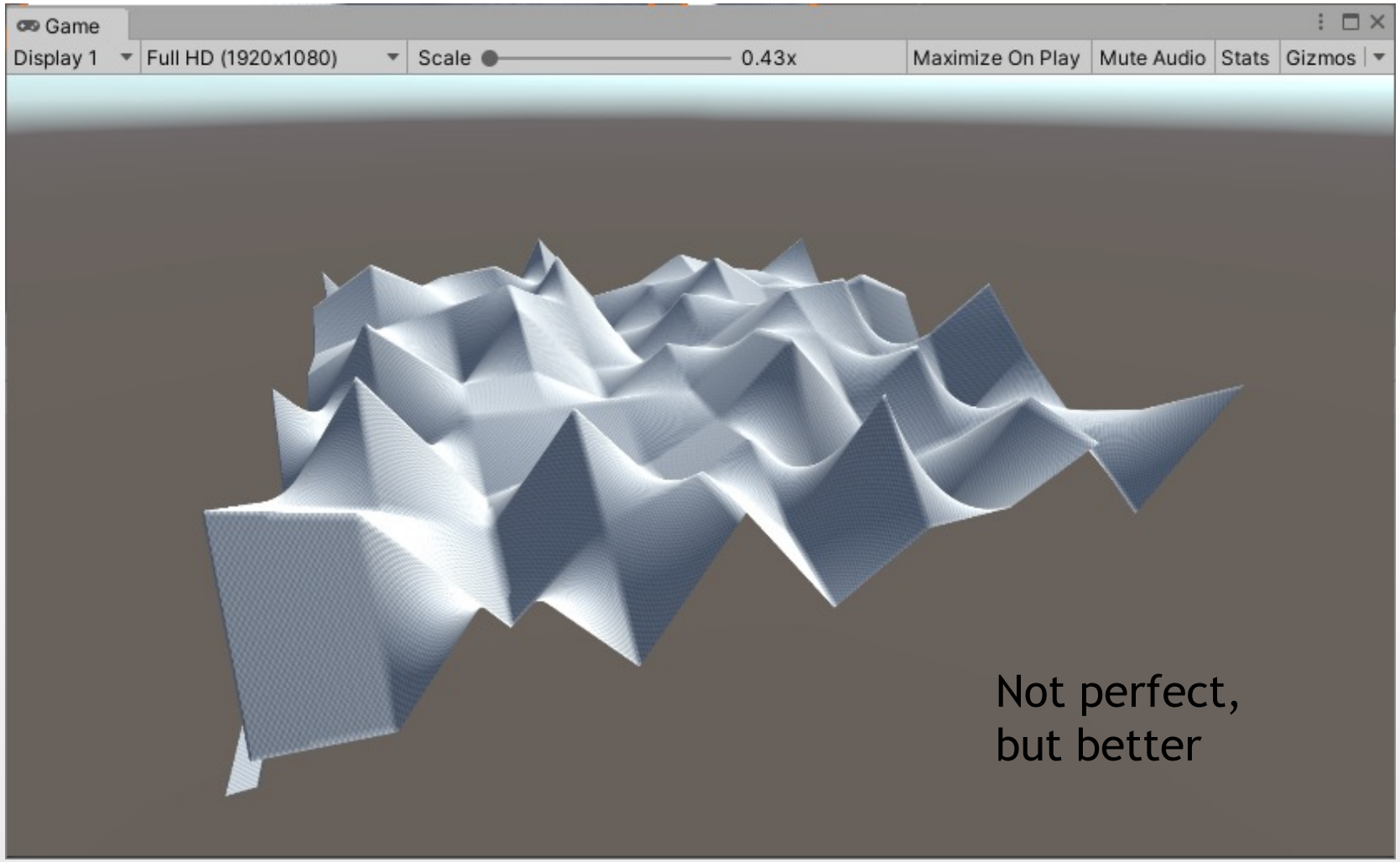
k will interpolate all points of the terrain along the x axis

The version presented in this slide is available as a “standalone” script in the Random scene (you can manually activate this script or the RandomTerrain shown before)

Another version, slightly more complex, is present in the repository in the InterpolatedTerrain script, where you can select the interpolation from the interface

Final Result

Scene: Interpolated
Folder: PCG



Bicubic Interpolation

Source: InterpolatedTerrain
Folder: PCG

```
private static float [,] BicubicInterpolator (int x, int y, int subsampling) {
    float [,] h = new float [y, x];

    // build a lattice
    for (int i = 0; i < x; i += subsampling) {
        for (int j = 0; j < y; j += subsampling) {
            h [j, i] = Random.Range (0f, 1f);
        }
    }

    // cut out border effects to make code simpler
    int xCut = subsampling * ((int)Mathf.Floor (x / subsampling));
    int yCut = subsampling * ((int)Mathf.Floor (y / subsampling));

    // build bilinear interpolations
    // first direction i only on lattice points
    for (int i = 0; i < xCut; i += subsampling) { // avoid border effect
        for (int j = 0; j <= yCut; j += subsampling) {
            for (int k = 1; k < subsampling; k += 1) {
                h [j, i + k] = Mathf.Lerp (h [j, i], h [j, i + subsampling], slope((float)k / (float)subsampling));
            }
        }
    }
    // then direction j on all points to cover all grid
    for (int i = 0; i <= xCut; i += 1) {
        for (int j = 0; j < yCut; j += subsampling) {
            for (int k = 1; k < subsampling; k += 1) {
                h [j + k, i] = Mathf.Lerp (h [j, i], h [j + subsampling, i], slope((float)k / (float)subsampling));
            }
        }
    }

    return h;
}
```

The only difference here is the introduction of a slope function

Bicubic Interpolation

- Using the standard formula

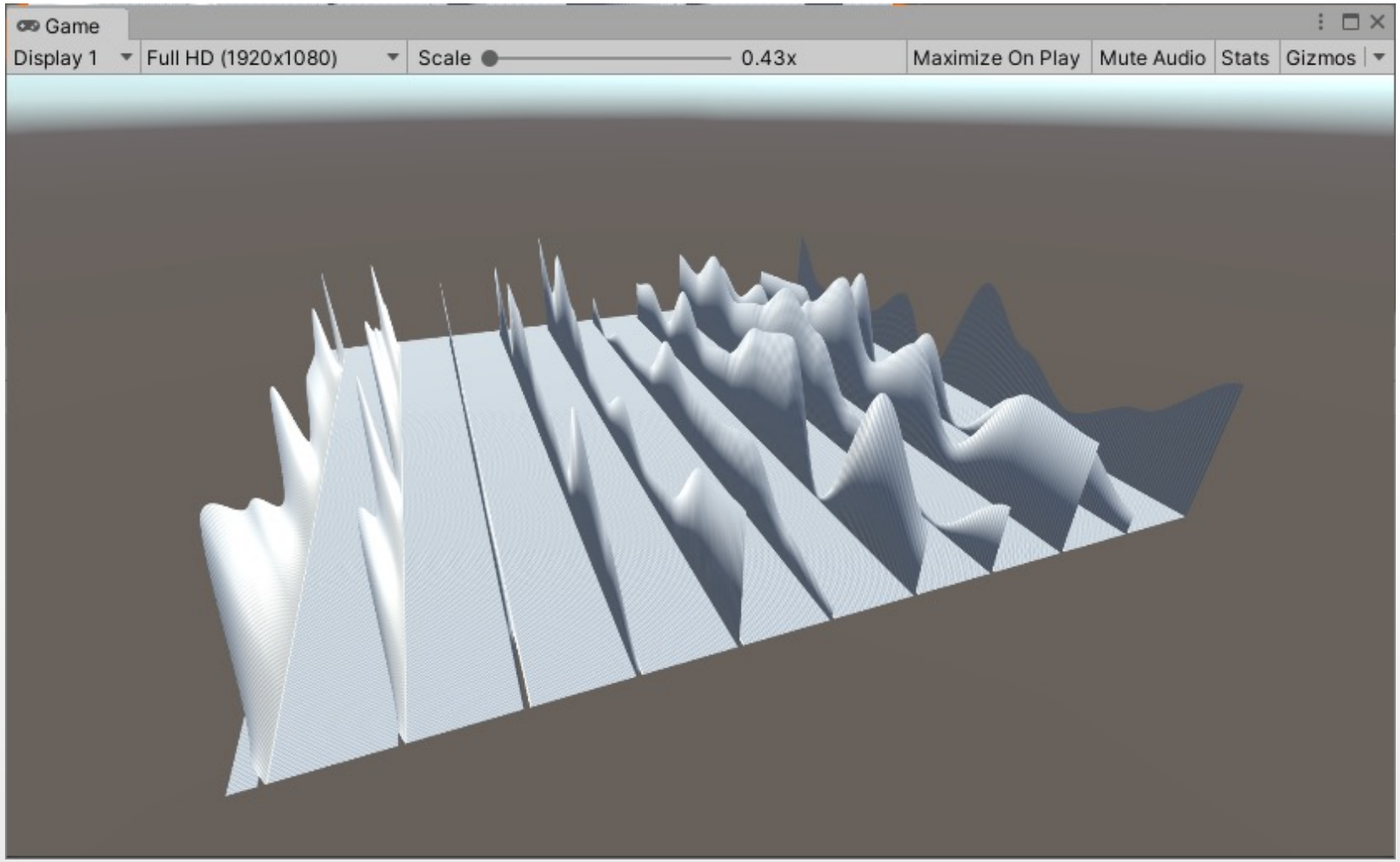
```
private static float slope (float x) {
    return -2f * Mathf.Pow (x, 3) + 3f * Mathf.Pow (x, 2);
}
```

- To revert to bilinear, you can use the same code with

```
private static float slope (float x) {
    return x;
}
```

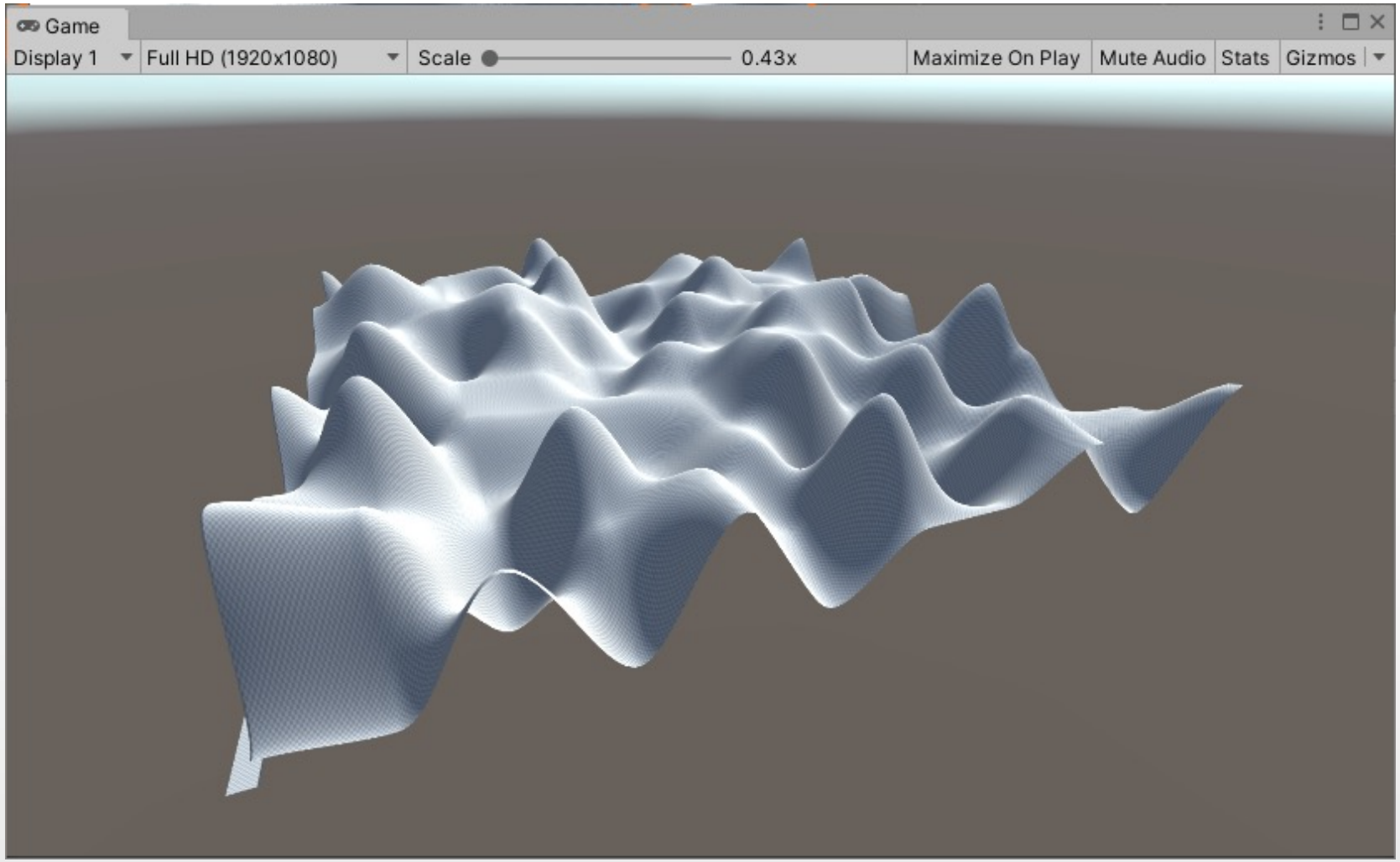
One Way Bicubic Interpolation

Scene: StepsTerrain
Folder: PCG

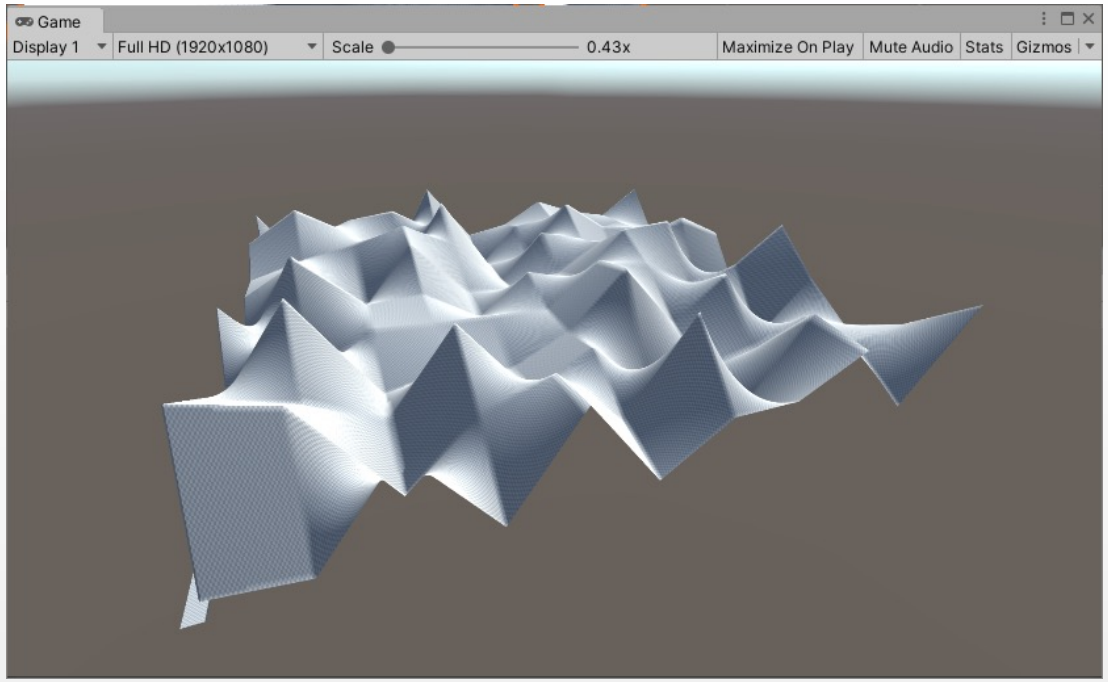


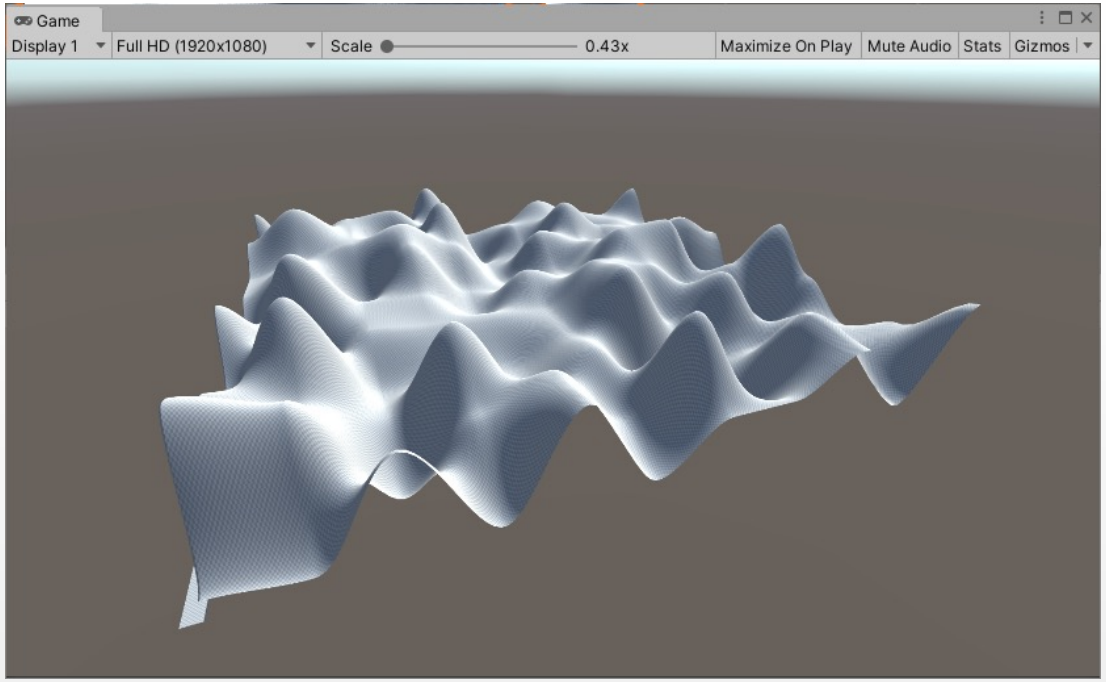
Bicubic Interpolation Results

Scene: Interpolated
Folder: PCG



Bilinear vs Bicubic





Starting samples are the same

What Interpolation is NOT Giving Us

Canyons, terraces, and crannies

Perlin Noise

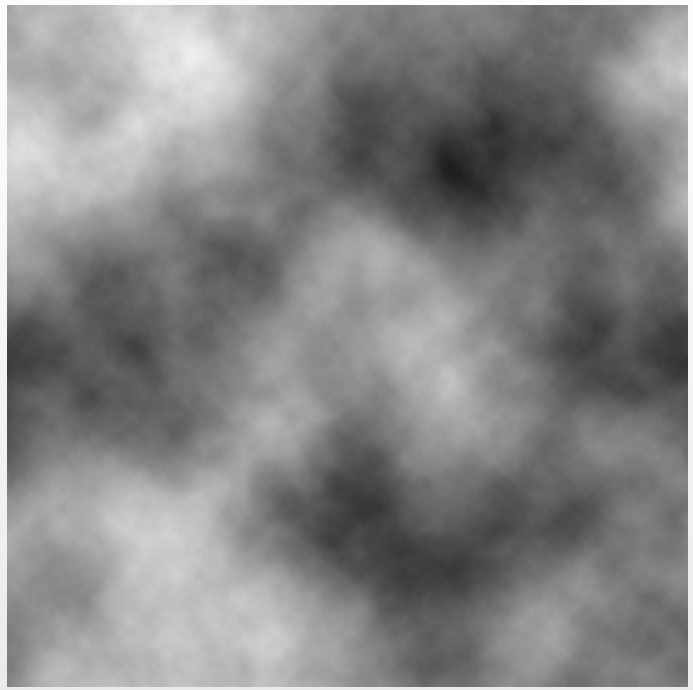
- Perlin noise can be used as another, more natural, way to generate a terrain

- **But perlin noise is NOT for terrain**, perlin noise is a swiss army knife to add noise (hence, the name) to a set of values in order to make them more natural

What IS Perlin Noise

- Perlin noise is a mathematical model associating to each point of a multi-dimensional space a numerical value. Neighboring numerical values inside the multi-dimensional space are bounded in gradient

- Perlin noise on a two-dimensional space is this:



Perlin Noise Can Be Everything

- The meaning that we give to the values inside the multi-dimensional space can be EVERYTHIG

- One dimension
  - Traffic distribution on a freeway
  - Distortion of audio along a propagation path
  - Side offset of a car while driving along a road

- Two dimensions
  - Density of population on a map
  - Water availability in the underground
  - A texture representing magma

- Three dimensions
  - Density of gas inside a chamber
  - A cloud formation in the sky
  - Shark encounter probability in the sea

Perlin Noise is a Noise

- Perlin noise is giving us values to be added to “perfect” information we already have, and make it feel more natural

- 1-dimension: make a sword blade dull
- 2-dimensions: make a regular surface bumpy
- 3-dimensions: add distortions in water transparency

This is why we can use it here

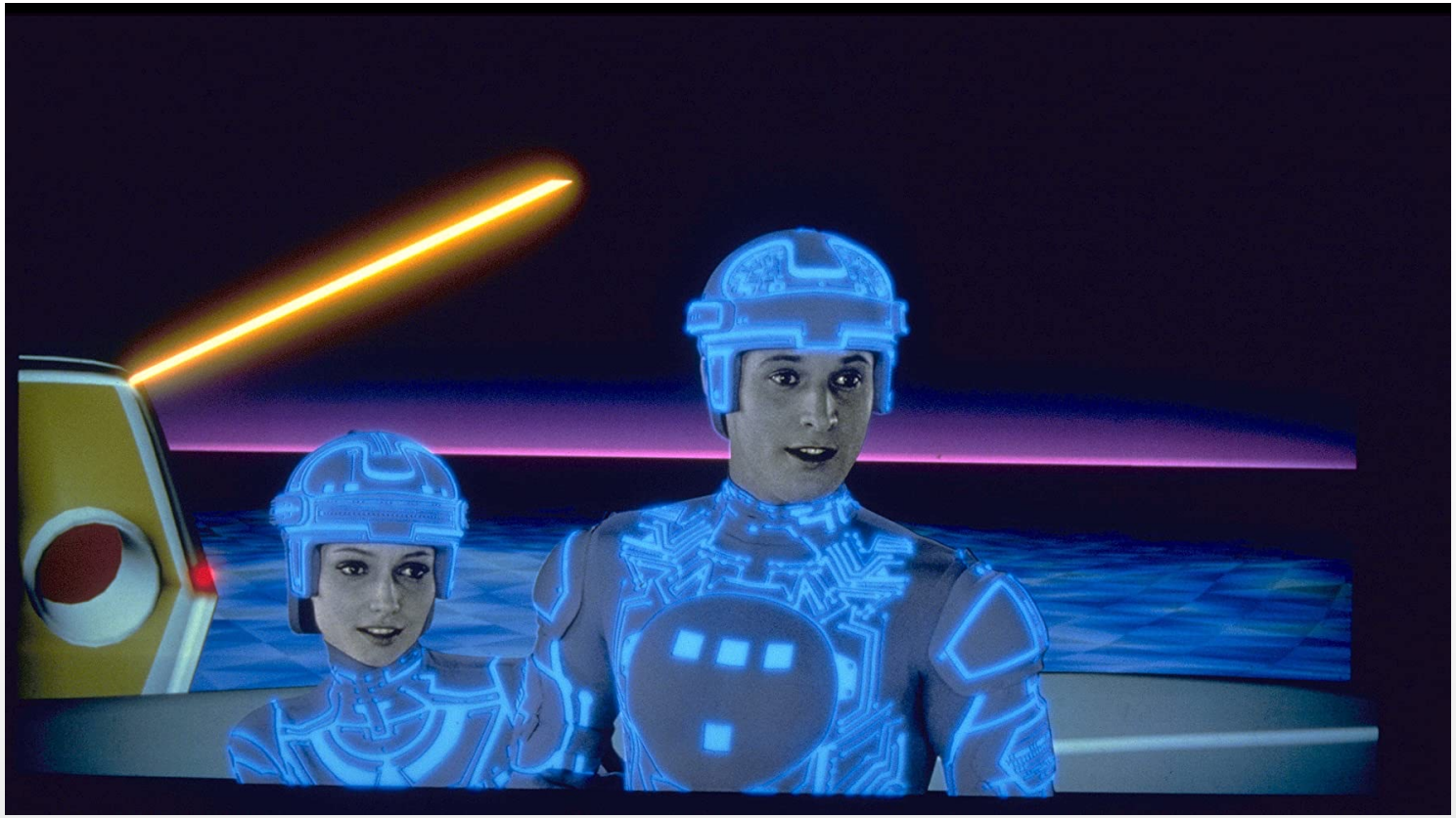
Gradient-Based Terrain Generation

- Interpolation methods generate heights (value noise) and calculate slopes

- Gradient-based methods generate slopes (gradient noise) and calculate heights
  - This method has been invented by Ken Perlin for a silly nerd movie in 1982, and is usually regarded as *Perlin Noise*

Mr. Perlin won an oscar for this!

Perlin Noise



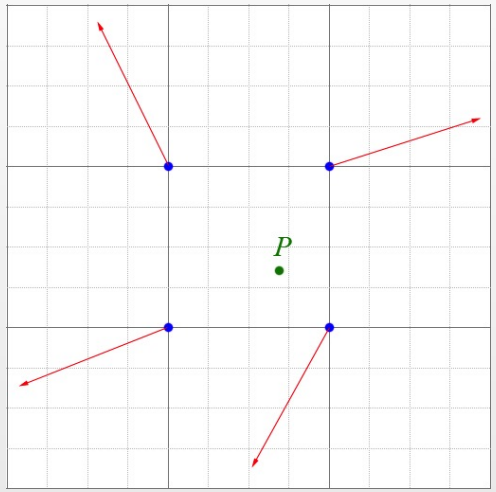
Perlin Noise

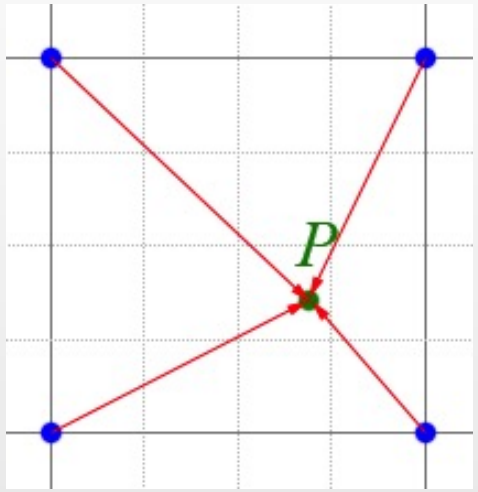
- We still use the lattice grid but instead of randomizing height we randomize gradient at the lattice points

- We set the height for all lattice points to zero
  - Kind of counter-intuitive, but remember the idea is to create a bumpy blanker to cover another shape

Perlin Noise

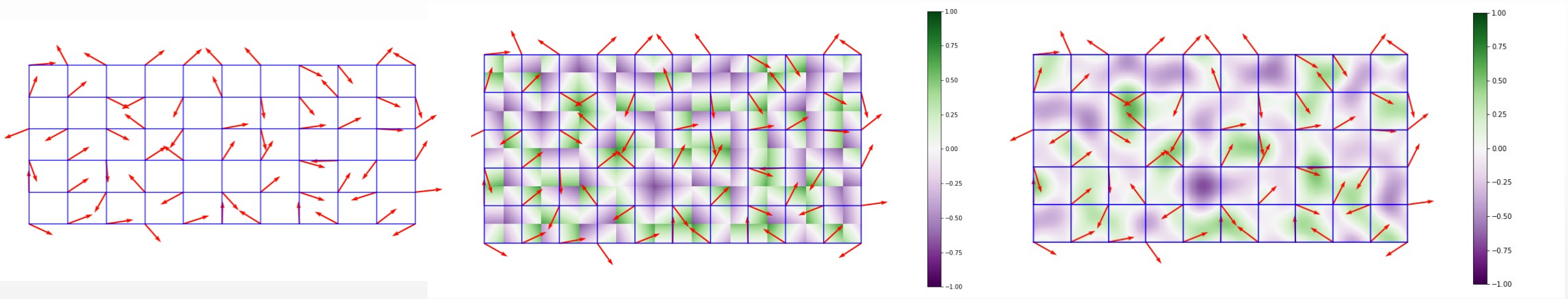
- To find the height values at non-lattice points, we look at the four neighboring lattice points
  - Based on the gradient at each lattice point, we can evaluate a contributed height using a dot product
    - dot product between the gradient vector and a vector drawn from the lattice point to our current point
  - We end up with four heights … to interpolate



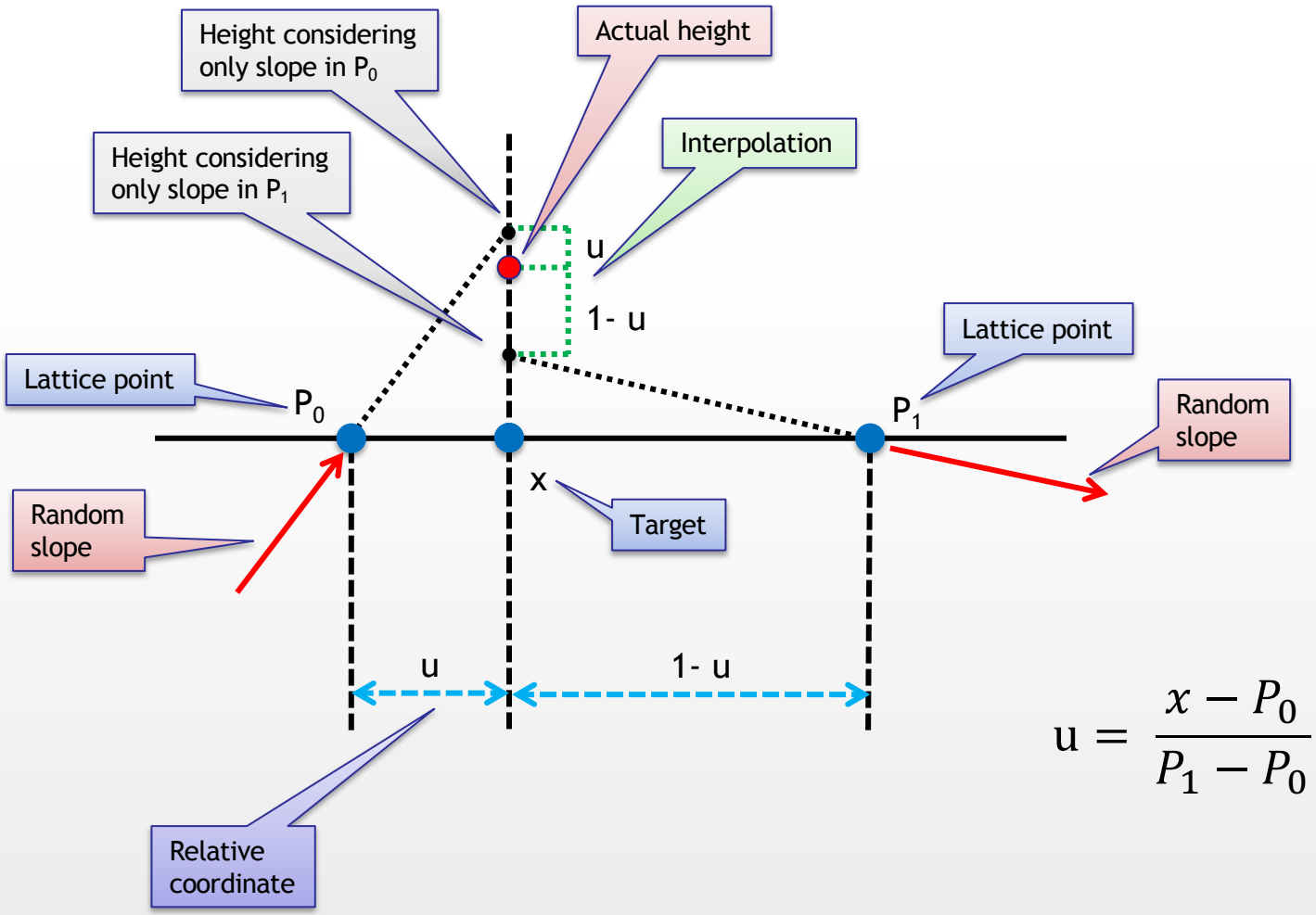


You are supposed to know what are and how to use dot and cross products!

Perlin Noise

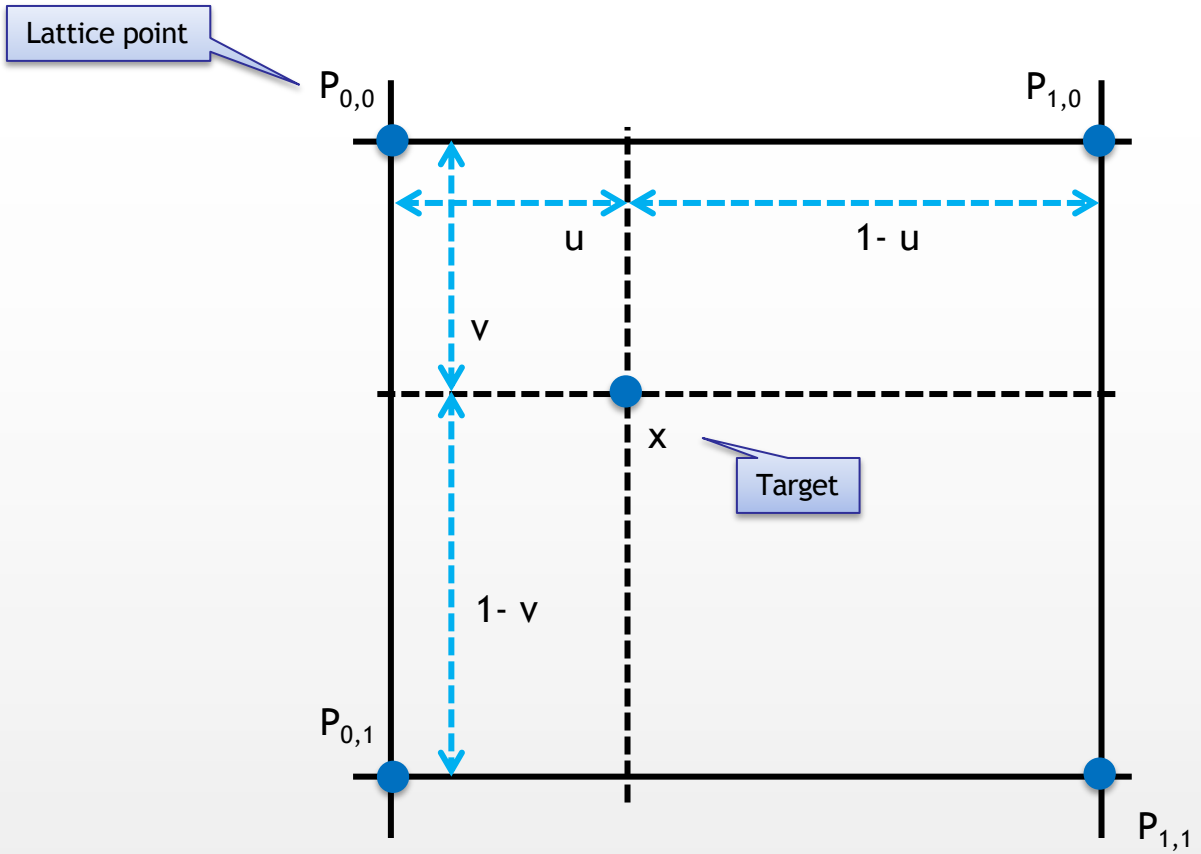


Perlin Interpolation … in 2D



$$u = \frac{x - P_0}{P_1 - P_0}$$

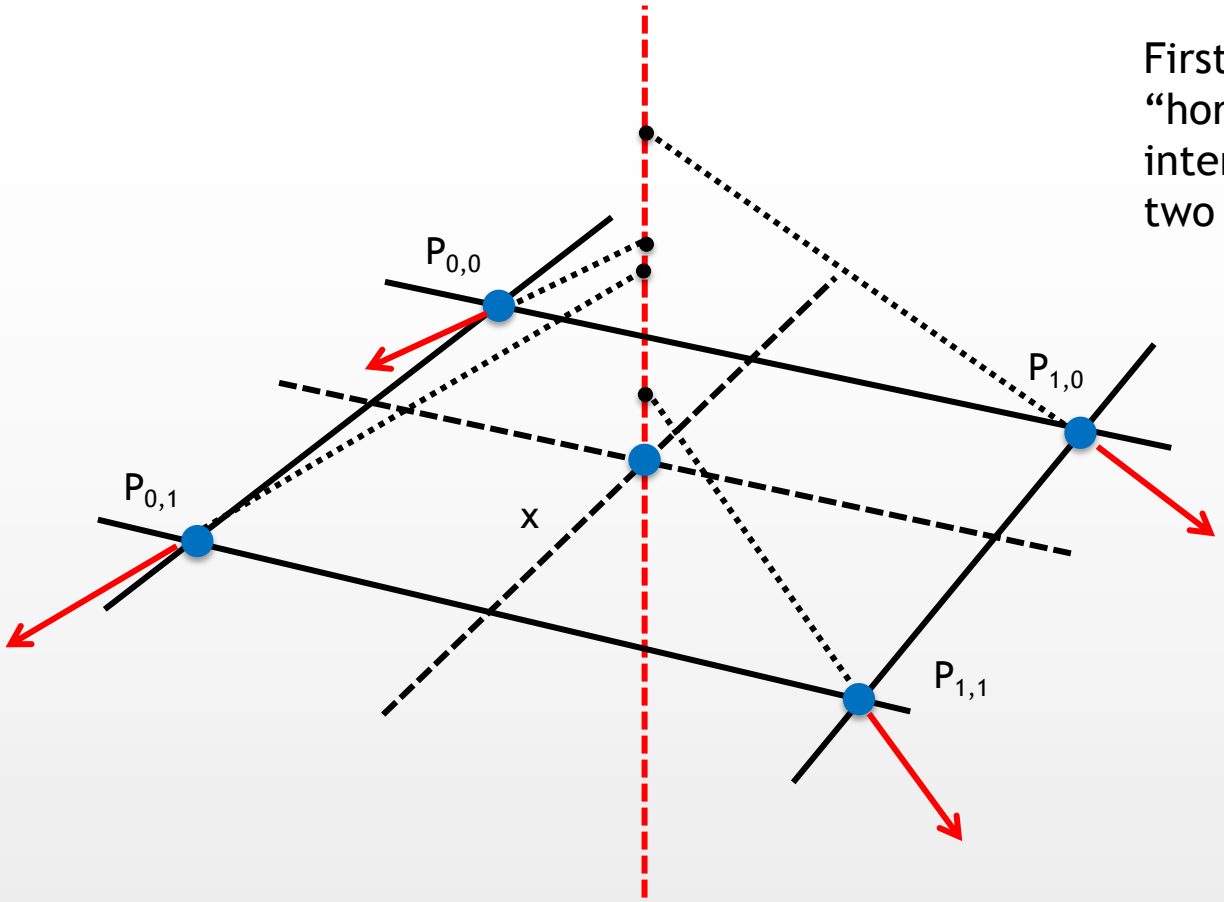
Perlin Interpolation in 3D



If you look to the same process from upside down, things are similar, but we have one relative coordinate for each axis.

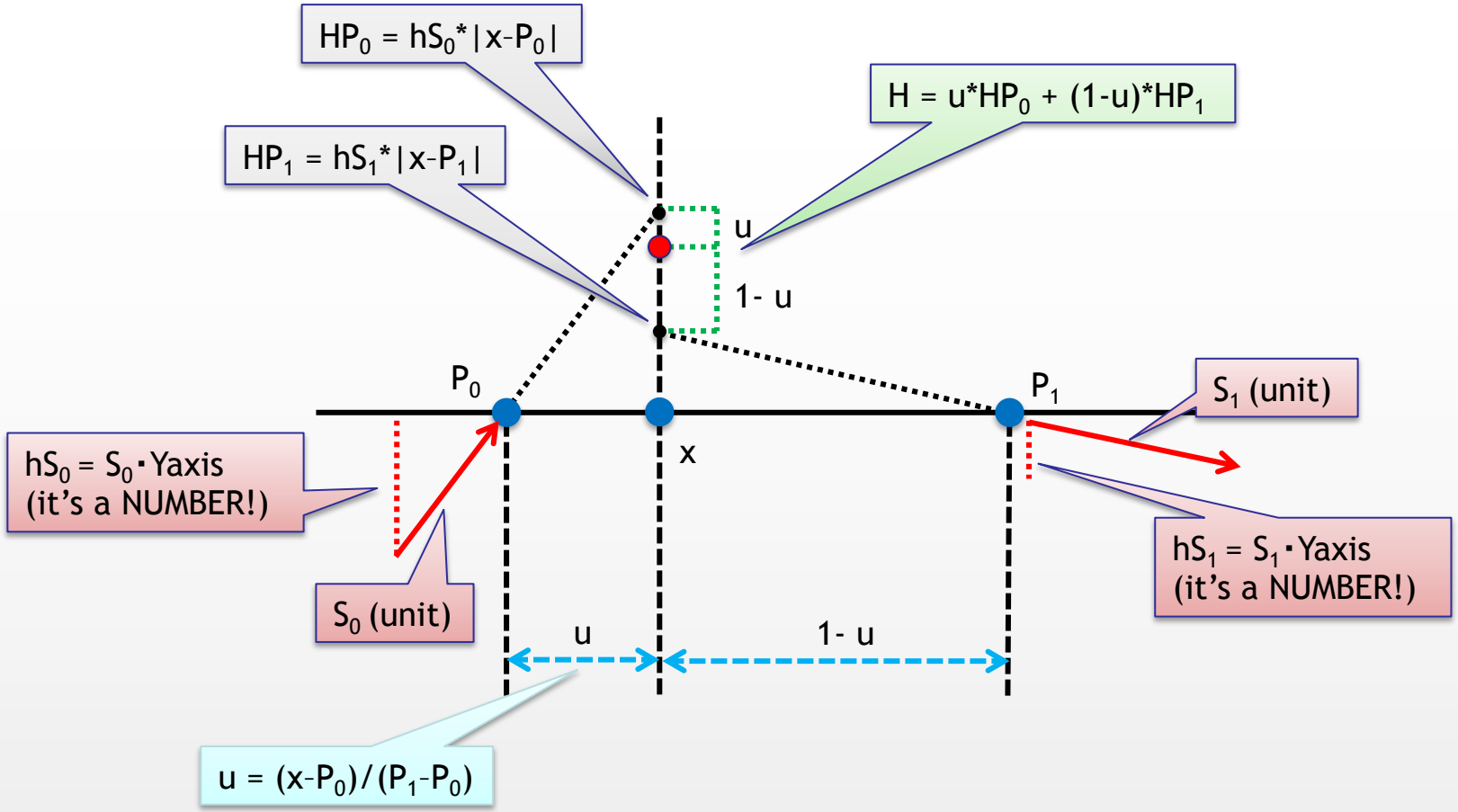
We call them u, and v, following the notation of UV coordinates for textures in Computer Graphics

Perlin Interpolation in (more) 3D



First interpolate "horizontally" using u, then interpolate "vertically" the two results using v

Now … with Math!



Perlin Noise Generation

Source: PerlinTerrain
Folder: PCG

```
float [,] h = new float [y, x];
Vector2 [,] slopes = new Vector2 [y, x];

// first, set up the slopes and height at lattice points
for (int i = 0; i <= xCut; i += subsampling) {
    for (int j = 0; j <= yCut; j += subsampling) {
        slopes [j, i] = Random.insideUnitCircle;
        h [j, i] = 0;
    }
}

// now let's start with the good stuff
// THIS CODE IS SUB-OPTIMAL!
for (int i = 0; i < xCut; i += 1) {
    for (int j = 0; j < yCut; j += 1) {

        // find the neighbouring lattice points
        int floorI = subsampling * ((int) Mathf.Floor ((float) i / subsampling));
        int floorJ = subsampling * ((int) Mathf.Floor ((float) j / subsampling));
        int ceilI  = subsampling * ((int) Mathf.Ceil  ((float) i / subsampling));
        int ceilJ  = subsampling * ((int) Mathf.Ceil  ((float) j / subsampling));

        // calculate the four contribution to height
        float h1 = Vector2.Dot (slopes [floorJ, floorI], new Vector2 (i, j) - new Vector2(floorI, floorJ));
        float h2 = Vector2.Dot (slopes [floorJ, ceilI], new Vector2 (i, j) - new Vector2 (ceilI, floorJ));
        float h3 = Vector2.Dot (slopes [ceilJ, floorI], new Vector2 (i, j) - new Vector2 (floorI, ceilJ));
        float h4 = Vector2.Dot (slopes [ceilJ, ceilI], new Vector2 (i, j) - new Vector2 (ceilI, ceilJ));

        // calculate relative position inside the square
        float u = ((float) i - floorI) / subsampling;
        float v = ((float) j - floorJ) / subsampling;

        // interpolate by lerping first horizontally (for each couple) and then vertically
        float l1 = Mathf.Lerp (h1, h2, slope(u));
        float l2 = Mathf.Lerp (h3, h4, slope(u));
        float finalH = Mathf.Lerp (l1, l2, slope(v));

        h [j, i] = finalH;
    }
}

td.SetHeights (0, 0, Normalize(h, x, y));
```

i and j are scanning all the terrain. Floor and ceiling functions are used to identify the 4 surrounding lattice points for every terrain point

NOTE: casts for i and j are extremely important!!!

Perlin Noise Generation

```
float [,] h = new float [y, x];
Vector2 [,] slopes = new Vector2 [y, x];

// first, set up the slopes and height at lattice points
for (int i = 0; i <= xCut; i += subsampling) {
    for (int j = 0; j <= yCut; j += subsampling) {
        slopes [j, i] = Random.insideUnitCircle;
        h [j, i] = 0;
    }
}

// now let's start with the good stuff
// THIS CODE IS SUB-OPTIMAL!
for (int i = 0; i < xCut; i += 1) {
    for (int j = 0; j < yCut; j += 1) {

        // find the neighbouring lattice points
        int floorI = subsampling * ((int) Mathf.Floor ((float) i / subsampling));
        int floorJ = subsampling * ((int) Mathf.Floor ((float) j / subsampling));
        int ceilI  = subsampling * ((int) Mathf.Ceil  ((float) i / subsampling));
        int ceilJ  = subsampling * ((int) Mathf.Ceil  ((float) j / subsampling));

        // calculate the four contribution to height
        float h1 = Vector2.Dot (slopes [floorJ, floorI], new Vector2 (i, j) - new Vector2(floorI, floorJ));
        float h2 = Vector2.Dot (slopes [floorJ, ceilI], new Vector2 (i, j) - new Vector2 (ceilI, floorJ));
        float h3 = Vector2.Dot (slopes [ceilJ, floorI], new Vector2 (i, j) - new Vector2 (floorI, ceilJ));
        float h4 = Vector2.Dot (slopes [ceilJ, ceilI], new Vector2 (i, j) - new Vector2 (ceilI, ceilJ));

        // calculate relative position inside the square
        float u = ((float) i - floorI) / subsampling;
        float v = ((float) j - floorJ) / subsampling;

        // interpolate by lerping first horizontally (for each couple) and then vertically
        float l1 = Mathf.Lerp (h1, h2, slope(u));
        float l2 = Mathf.Lerp (h3, h4, slope(u));
        float finalH = Mathf.Lerp (l1, l2, slope(v));

        h [j, i] = finalH;
    }
}

td.SetHeights (0, 0, Normalize(h, x, y));
```

i moves along x and j moves along y
unfortunately, the terrain uses inverted indexes.
This is why i and j are jumping around

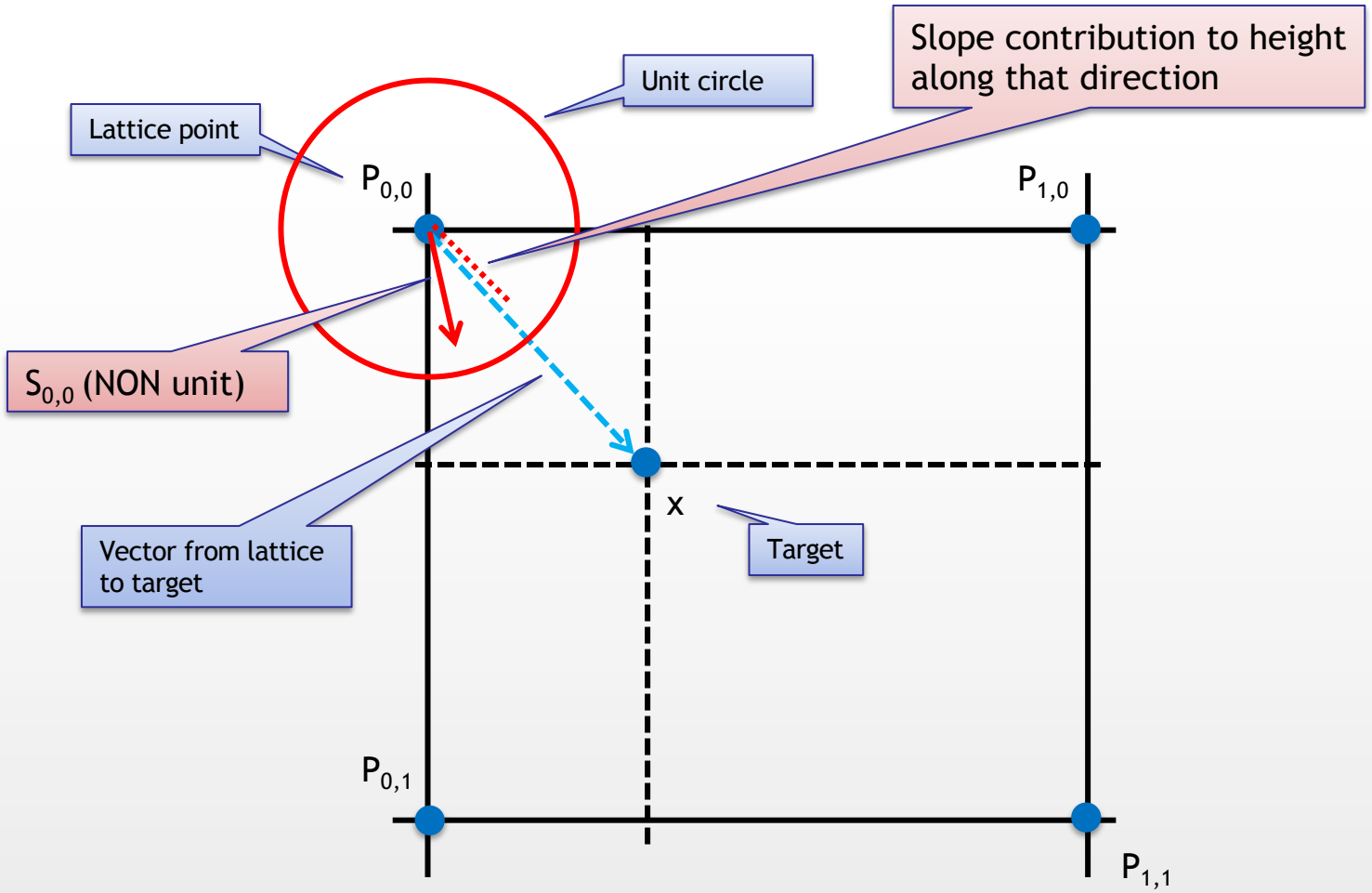
Slopes are actually 2D

NOTE: casts for i and j are extremely important!!!

But ...

- How comes slopes for terrain (3D) are 2D vectors?
  - Yes, we are changing the rules
- These 2D vectors are indicating in which direction the slope is “going up” and how strong is it (magnitude)
- The projection (dot product) of 2D slopes along the vector going from the lattice point to the evaluated point means “how strong” is the slope contribution while going in that direction

What is Happening?

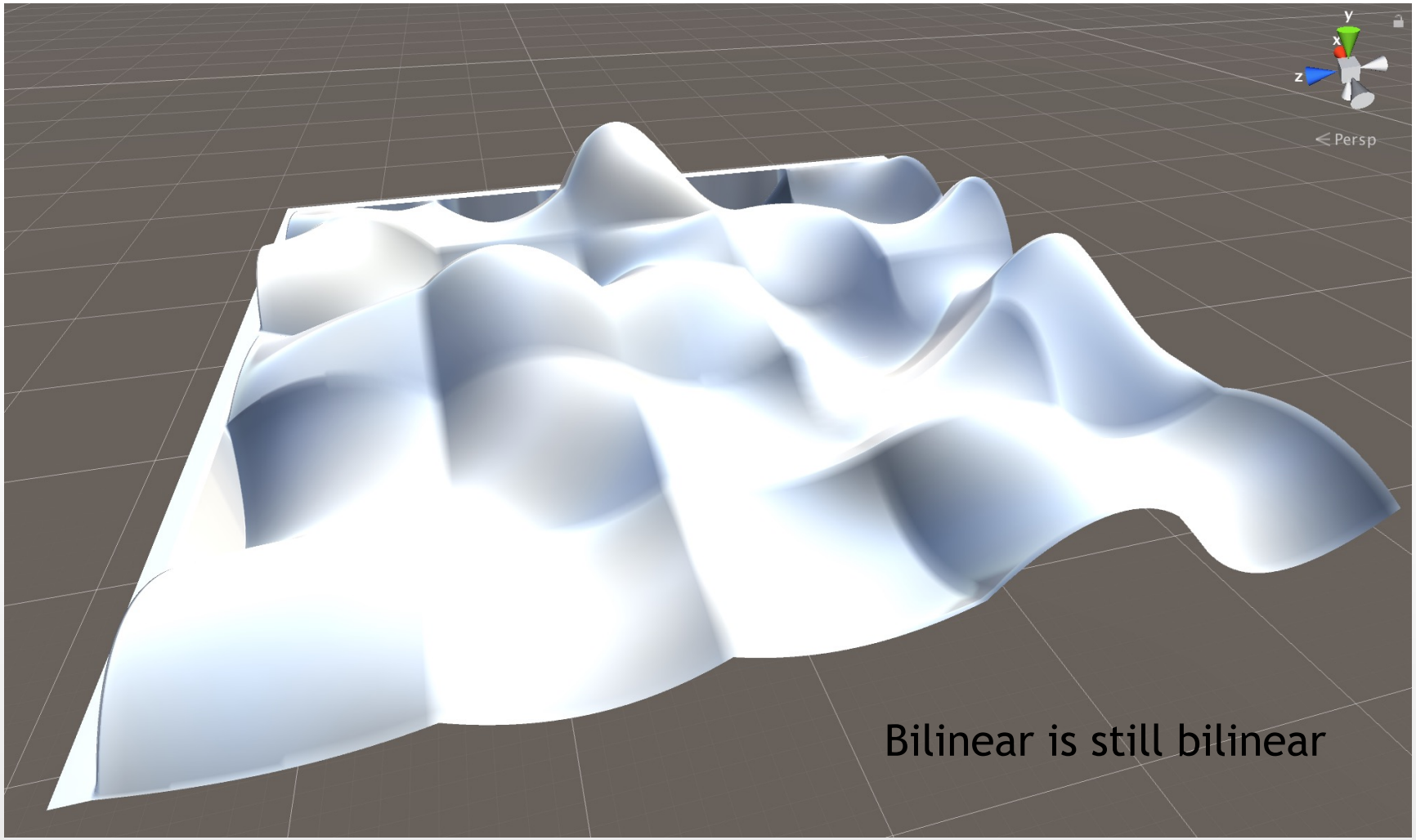


Perlin Noise Generation

```
private float [,] Normalize (float [,] m, int x, int y) {
    float max, min;
    max = float.MinValue;
    min = float.MaxValue;
    for (int i = 0; i < x; i += 1) {
        for (int j = 0; j < y; j += 1) {
            if (m [j, i] < min) min = m [j, i];
            if (m [j, i] > max) max = m [j, i];
        }
    }
    for (int i = 0; i < x; i += 1) {
        for (int j = 0; j < y; j += 1) {
            m [j, i] = (m [j, i] - min) / (max - min);
        }
    }
    return m;
}
```

Perlin Noise Results

Scene: PerlinNoise
Folder: PCG



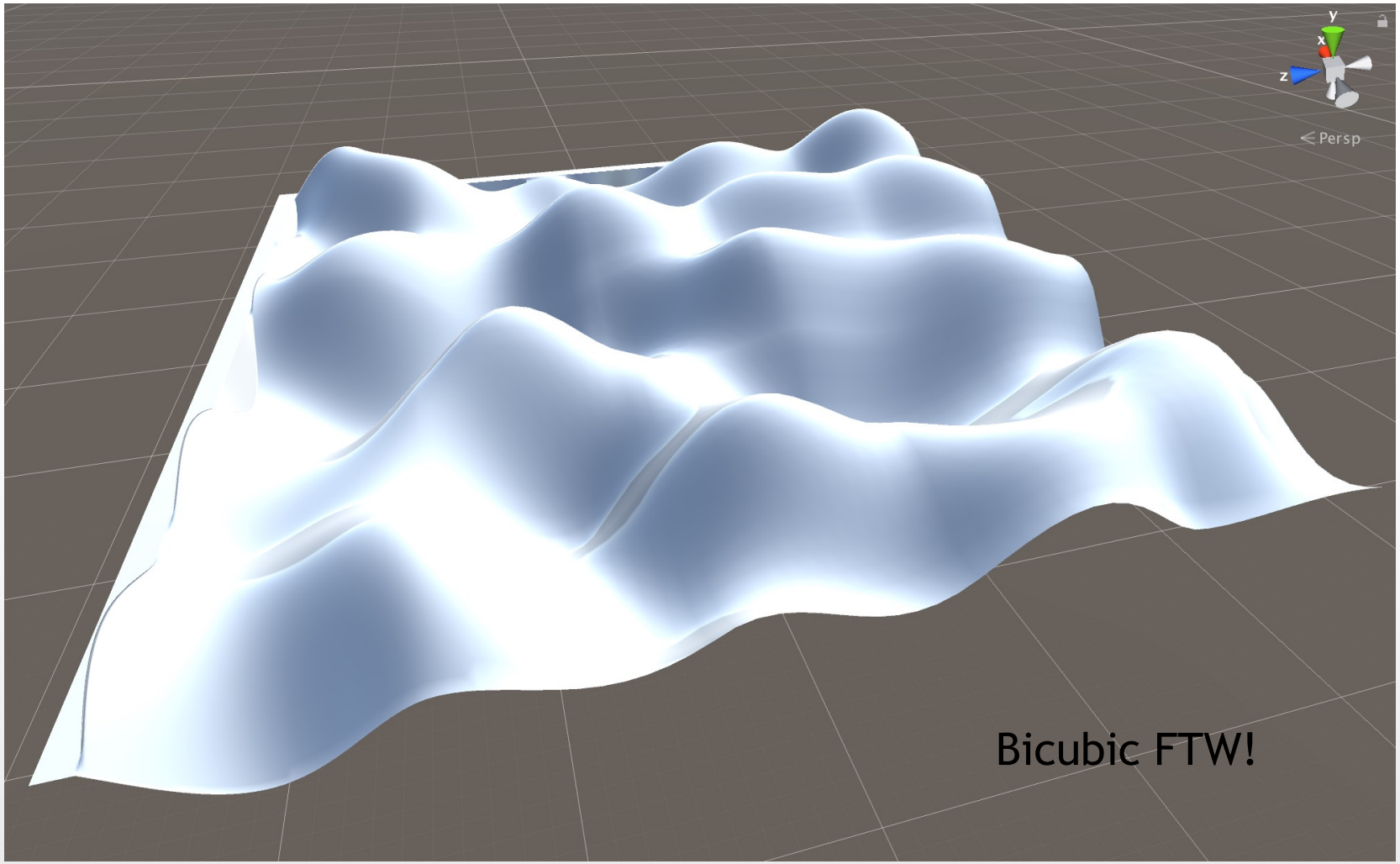
Perlin (Bicubic) Noise

```
// interpolate by lerping first horizontally (for each couple) and then vertically
float l1 = Mathf.Lerp (h1, h2, slope(u));
float l2 = Mathf.Lerp (h3, h4, slope(u));
float finalH = Mathf.Lerp (l1, l2, slope(v));
```

```
private static float slope (float x) {
    return -2f * Mathf.Pow (x, 3) + 3f * Mathf.Pow (x, 2);
}
```

$6x^5-15x^4+10x^3$
Has been found to be a better choice here.
Even if it is a bit more computationally intensive

Perlin (Bicubic) Noise Results

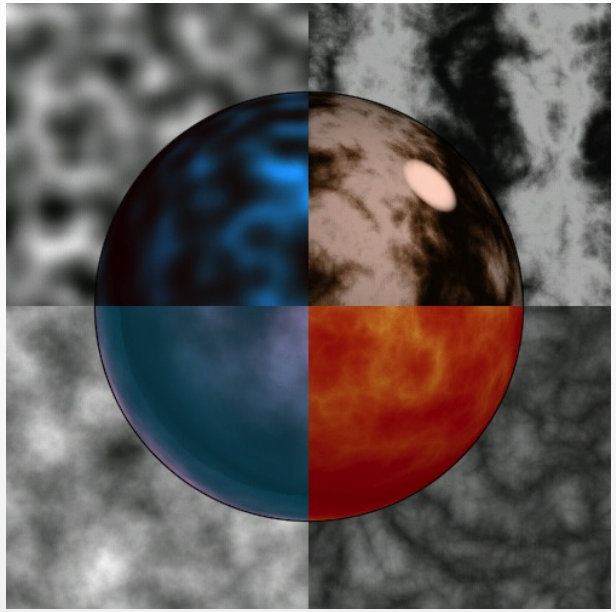


Variations on Perlin noise

- Perlin noise applications
  - alone
  - into mathematical expressions, to generate different effects
  - see https://bit.ly/3CPW8ZI for a presentation on Perlin noise by Perlin himself

$$sin\left(x + \sum\left(\frac{1}{f}|noise|\right)\right)$$

$noise$

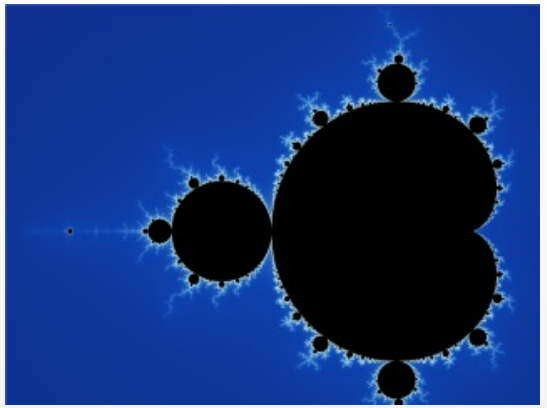


$$\sum\left(\frac{1}{f}noise\right) \qquad \sum\left(\frac{1}{f}|noise|\right)$$

Fractal Terrain

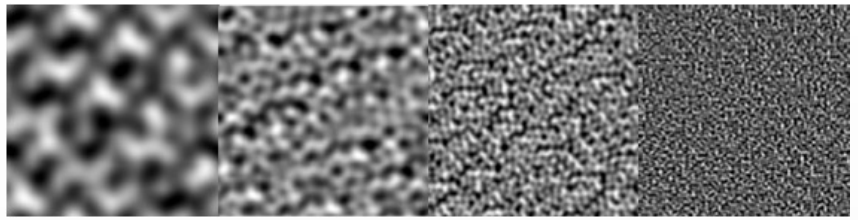
- Noises generated so far “undulates” around fixed (lattice) points
- Real (natural) landscapes offer similar variations over multiple scales
  - Ever tried to zoom a Mandelbrot set?

- A very easy way to produce fractal terrain is to take the single-scale random terrain methods and run them several times, at multiple scales and add all the scales together

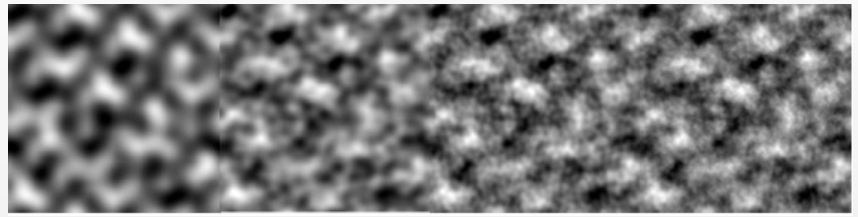


$$noise(f) + \frac{1}{2}noise(2f) + \frac{1}{4}noise(4f) + \ldots$$

Octaves



Noise at frequencies 4,8,16,32



Summed noise at 1,2,3,4 octaves

Agent-Based Terrain Creation

- In the same way we used an agent to dig dungeon, the same approach is possible when creating terrains in the open

- Nevertheless, we do not have one single operation in this case (dig). So, it is advisable to use multiple agents with different building purposes
  - Coastlines and beaches
  - Mountains and hills
  - Roads and rivers

- Dividing the task between multiple agents allows for a better control on the process from the level designer and simplifies the overall software architecture of the A.I.

Study Material

- Procedural Content Generation in Games
  ISBN 978-3-319-42716-4
  by N. Shaker, J. Togelius, M. J. Nelson
  - you can download it for free from Unimi IP addresses from:
    https://link.springer.com/book/10.1007/978-3-319-42716-4
  - an open access version of the book is available at:
    http://pcgbook.com/

- Chapter 4 up to § 4.4 included