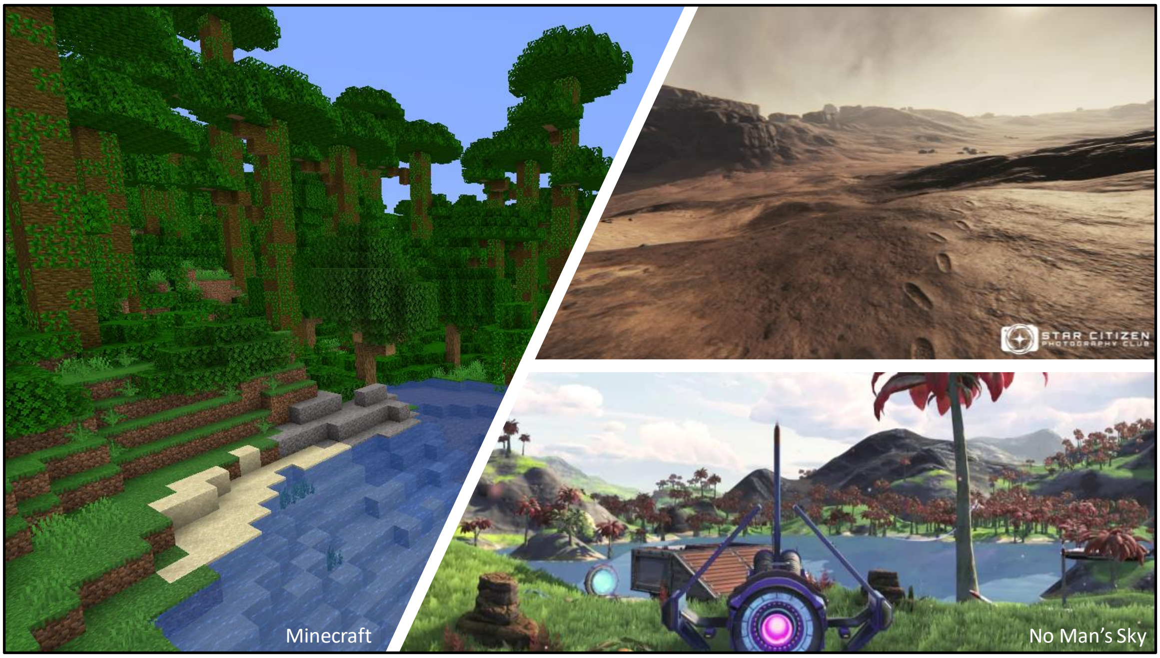
Procedural Content Generation for Computer Games

Lecture 1 – Terrain Generation

Vojtěch Černý
cerny@gamedev.cuni.cz

Terrain generation

- Arguably the most common form of PCG
- Why do it?
  - Cost-efficiency
  - Consistency
  - High-level of detail
  - Works well as mixed-initiative
  - Infinite worlds
  - Exploration as a game mechanic
  - It is fun



https://minecraft.gamepedia.com/Biome
https://www.voidu.com/en/no-mans-sky
https://robertsspaceindustries.com/community/citizen-spotlight/11876-Star-Citizen-Photography-Club-Wallpapers-1-42

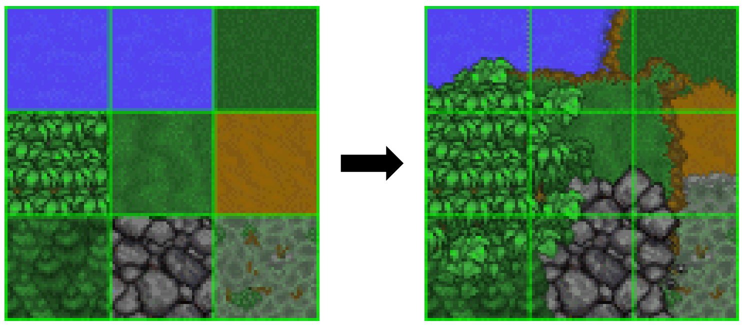
Different kinds of terrain generation

- By approach:
  - Teleological / Ontological
- By timing:
  - Design time / Runtime
- By representation:
  - Tilemaps
  - Heightmaps
  - Meshes
  - …

Representation is something you have to decide early on with any PCG.

TileMaps

- Quite simple for 2D games; assign a tile (integer) to each grid position

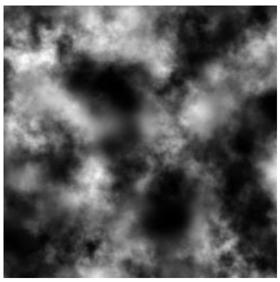


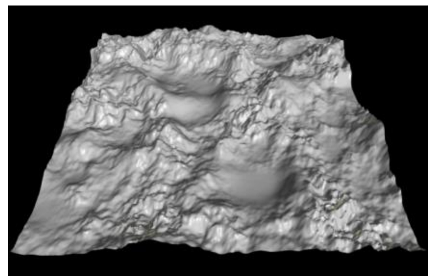
More natural look when overlaps are authored for given tile layers

http://archive.gamedev.net/archive/reference/articles/article934.html

HeightMaps

- Each point in a grid has an assigned height value
- Tilemaps can be just a discretization of height-maps


height visualized as brightness



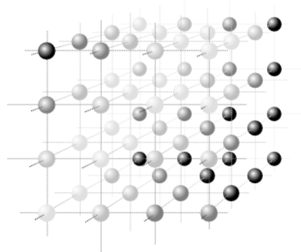
Pros: Quite simple to handle, good results

Cons: Cannot generate overhangs, caves, etc.

Visualization of height as black-white (lowest height – highest height)

Voxels

- Full 3D volumetric representation
- (a 3D grid of values)



height visualized as brightness

Pros: Can represent any 3D terrain

Cons: More complicated (a lot of data), harder to visualize

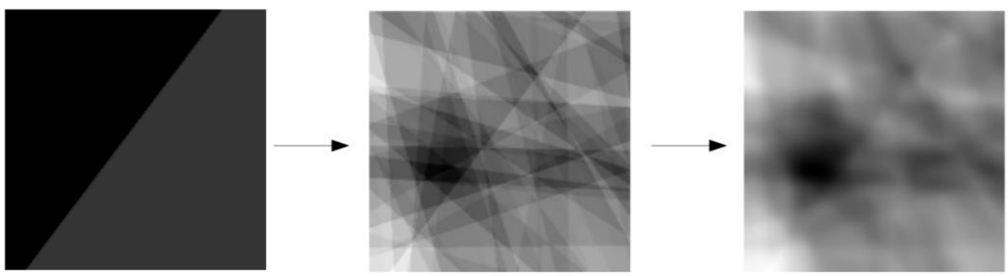
Visualization of height as black-white (lowest height – highest height)

HeightMaps
Simple approaches

A lot of PCG is actually just about hacking some stuff together, sort of this “anything-that-works” approach.
You may be encouraged to do this in your homeworks.

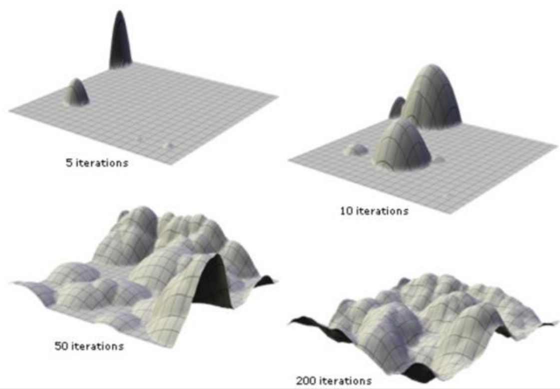
Fault formation

- Start with a grid of height 0
- Randomly create a line across the cell. Increase height of points on one side by an amount.
- Iterate previous step with continuously decreasing amount
- Apply filter in the end



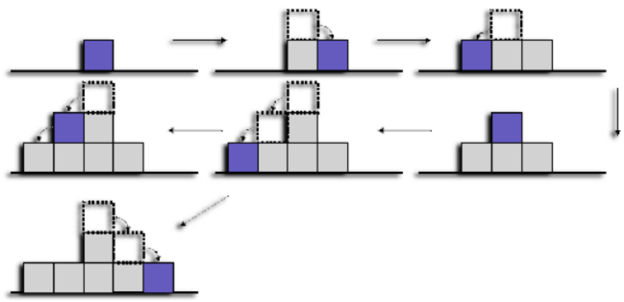
Hills-adding

- Add random hills (parabolas) to a flat terrain until ~~satisfied~~ hills cannot be distinquished



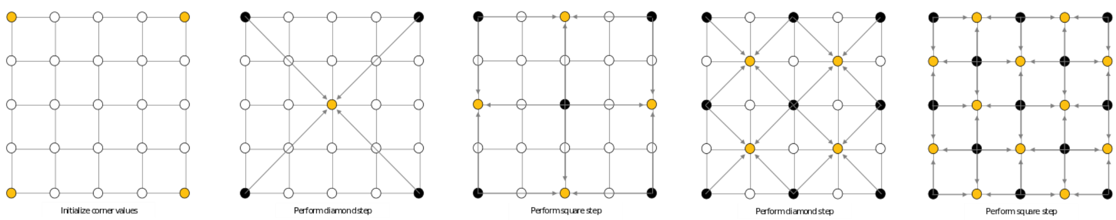
Particle deposition

- Add a unit of height on a random spot
- This unit will fall “down” on a neighboring square, if there is one that is at least two units of height lower



Diamond-square algorithm

- Improvement on midpoint-displacement algorithm
- Start with low-resolution height-map
- Alternate performing „diamond-step“ and „square-step“



Algorithm 1: Diamond-square

Data: mapSize (power of 2), $\alpha \in \mathbb{R}^+$

Main ()

- map $\leftarrow$ *blank heightmap of size* (mapSize + 1, mapSize + 1)
- *initialize corners of the map*
- size $\leftarrow$ mapSize
- **while** size $>$ 1 **do**
  - diamondStep(size)
  - squareStep(size)
  - size $\leftarrow$ size / 2
- **end**
- **return** map

Procedure `diamondStep()`

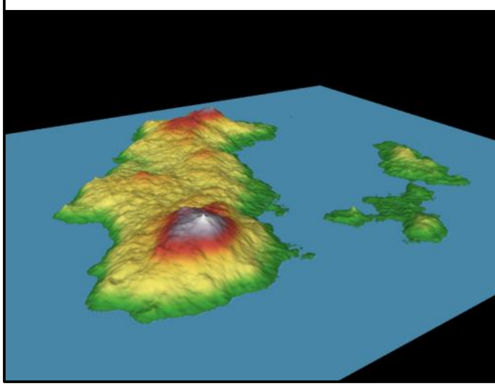
- **for** x in 0, size $\ldots$ (mapSize $-$ size) **do**
  - **for** y in 0, size $\ldots$ (mapSize $-$ size) **do**
    - corners $\leftarrow$ map[(x, y), (x + size, y), (x, y + size), (x + size, y + size)]
    - map[x + $\frac{\text{size}}{2}$, y + $\frac{\text{size}}{2}$] $\leftarrow$ mean(corners) + $\alpha \cdot$ size $\cdot$ uniform(-1, 1)
  - **end**
- **end**

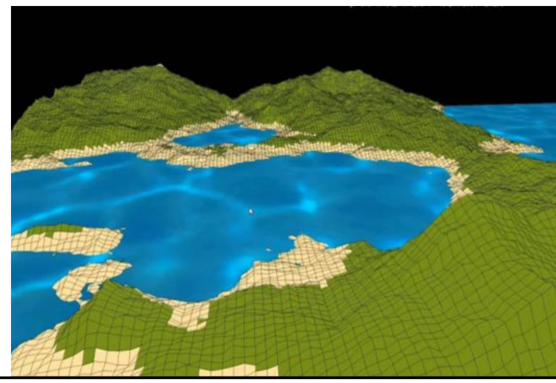
Procedure `squareStep()`

- **for** x in 0, size $\ldots$ mapSize **do**
  - **for** y in 0, size $\ldots$ mapSize **do**
    - coordsA $\leftarrow$ [(x, y), (x + size, y), (x + $\frac{\text{size}}{2}$, y + $\frac{\text{size}}{2}$), (x + $\frac{\text{size}}{2}$), (y $-$ $\frac{\text{size}}{2}$)]
    - coordsB $\leftarrow$ [(x, y), (x, y + size), (x + $\frac{\text{size}}{2}$, y + $\frac{\text{size}}{2}$), (x $-$ $\frac{\text{size}}{2}$), (y + $\frac{\text{size}}{2}$)]
    - midPointsA $\leftarrow$ *values of map at all coordsA which lie in map*
    - midPointsB $\leftarrow$ *values of map of all coordsB which lie in map*
    - map[x + $\frac{\text{size}}{2}$, y] $\leftarrow$ mean(midPointsA) + $\alpha \cdot$ size $\cdot$ uniform(-1, 1)
    - map[x, y + $\frac{\text{size}}{2}$] $\leftarrow$ mean(midPointsB) + $\alpha \cdot$ size $\cdot$ uniform(-1, 1)
  - **end**
- **end**

Diamond-square algorithm

- Results are impressive, given the simplicity of the algorithm
- Can be used as mixed-initative (smooth-out given heightmap)

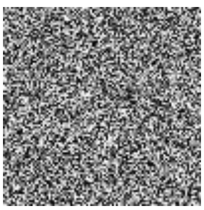




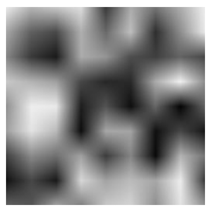
HeightMaps
Noise approaches

Noise functions

- White noise isn’t practical as a heightmap
- Can we make it smoother?
- Simple approach - interpolate

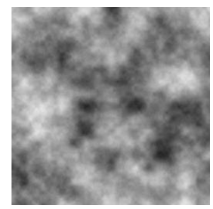


White noise



Bilinear interpolation

Smooth interpolation

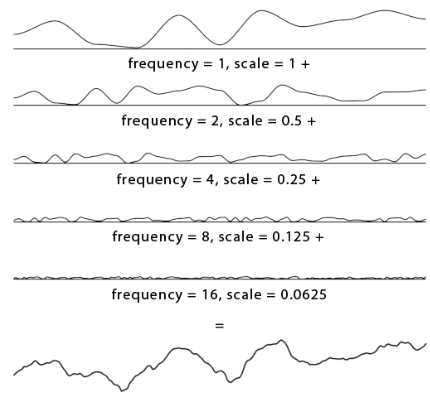


Octaved

This approach is called value noise

Side-note: Octaves

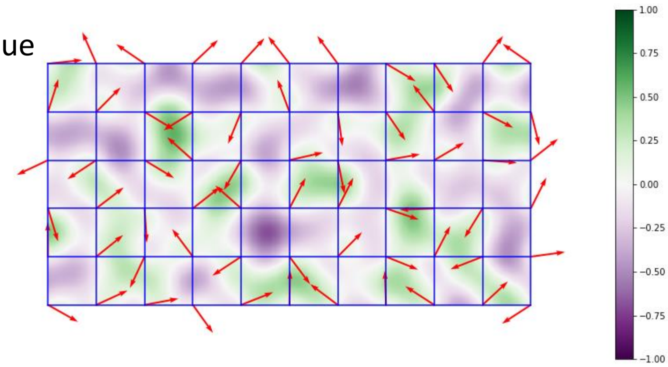
- A single noise function usually either:
  - Has no low-level details
  - Has no large-scale variations
- This can be solved by combining multiple instances with different frequencies and scales
- The parts are usually called „octaves“
- Can be used on many kinds of noise
  - (even value noise with octaves is usable)



Why are they called octaves – it is similar to music theory, where each octave means halving / doubling (it is also exponential)

Perlin noise

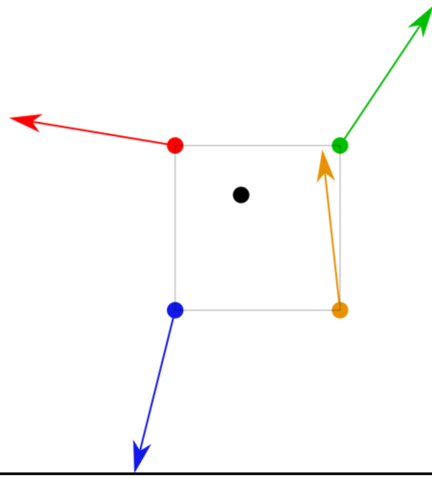
- (Ken Perlin, 1983)
- The first of gradient noise algorithms – generate random gradients and interpolate
- Fewer plateaus than value noise



https://en.wikipedia.org/wiki/Perlin_noise

Perlin noise

Gradient interpolation explained

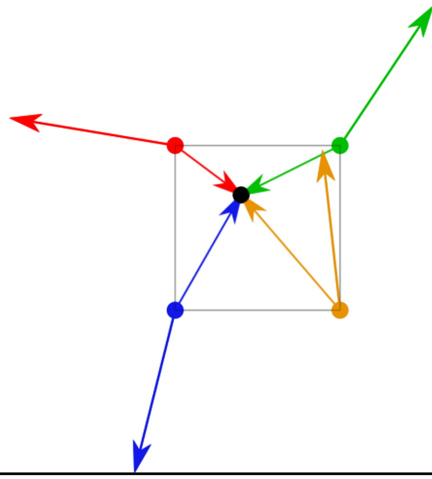


Algorithm 2: Perlin noise - gradient interpolation

Data: point $\in [0,1] \times [0,1]$, pseudorandom vectors in corners of unit square

Perlin noise

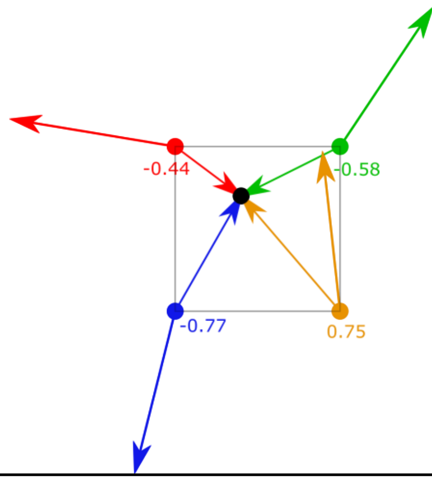
Gradient interpolation explained



```
Algorithm 2: Perlin noise - gradient interpolation
  Data: point ∈ [0,1] × [0,1], pseudorandom vectors in corners of unit square
1 construct vectors from corners to point
```

Perlin noise

Gradient interpolation explained



Algorithm 2: Perlin noise - gradient interpolation

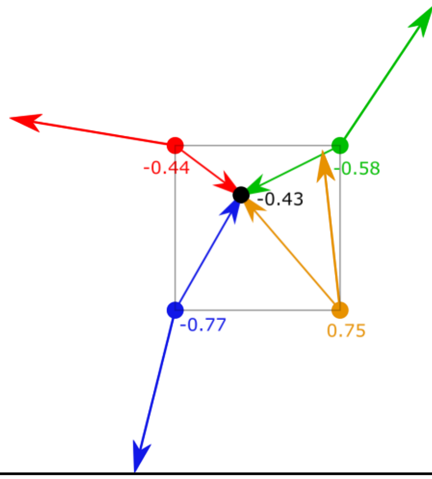
Data: point $\in [0,1] \times [0,1]$, pseudorandom vectors in corners of unit square

1 *construct vectors from corners to point*

2 *perform dot product of both vectors from each corner*

Perlin noise

Gradient interpolation explained



Algorithm 2: Perlin noise - gradient interpolation

Data: point $\in [0,1] \times [0,1]$, pseudorandom vectors in corners of unit square

1. *construct vectors from corners to point*
2. *perform dot product of both vectors from each corner*
3. *interpolate*

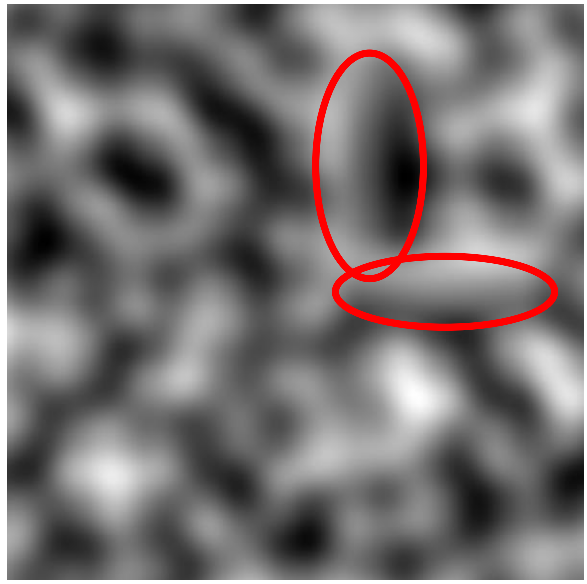
For interpolation, use smoothstep (Perlin, 1983) or smootherstep (Perlin, 2002)

https://mrl.nyu.edu/~perlin/paper445.pdf

Perlin noise

- Due to starting in 2D grid,

Perlin noise often has visible directional artifacts by both axes

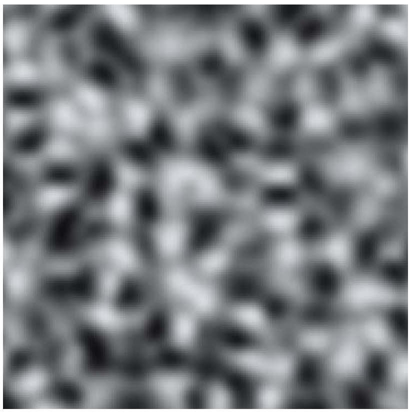


Simplex noise

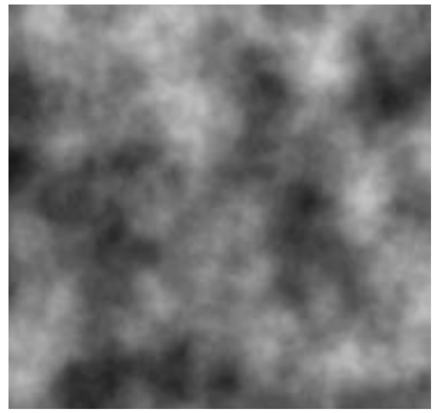
- (Ken Perlin, 2001)
- Improvement over the original algorithm, by:
  - Removing directional artifacts
  - Better performance, better scaling to higher dimensions (4D, 5D)
  - Simply computable gradients at any point
  - Easy hardware implementation

- The algorithm is a little more complex, so we won‘t go into detail here
- However, usually a very solid choice
- There exists an open source implementation called OpenSimplex

Simplex noise

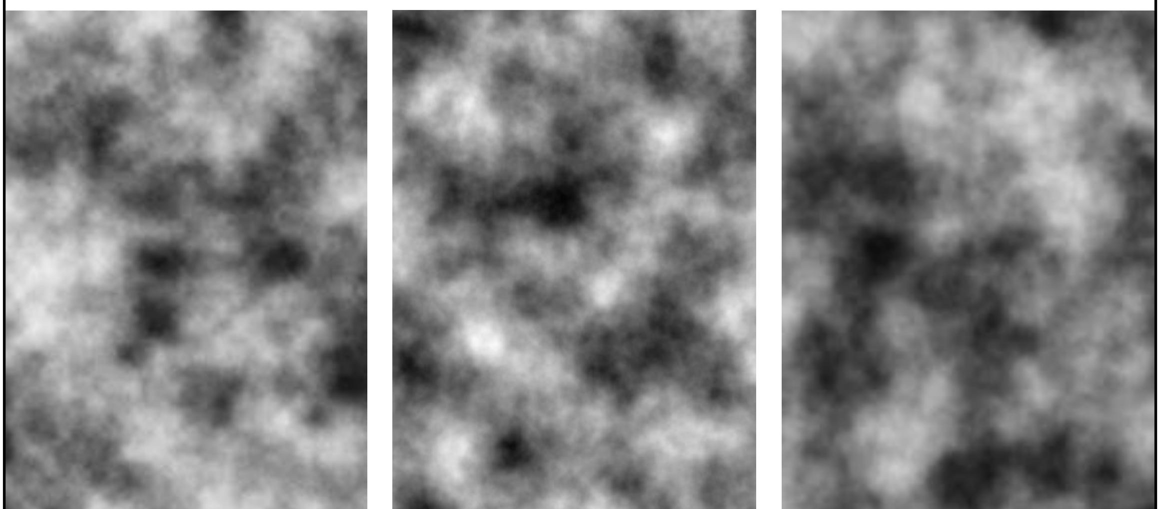


Single Octave



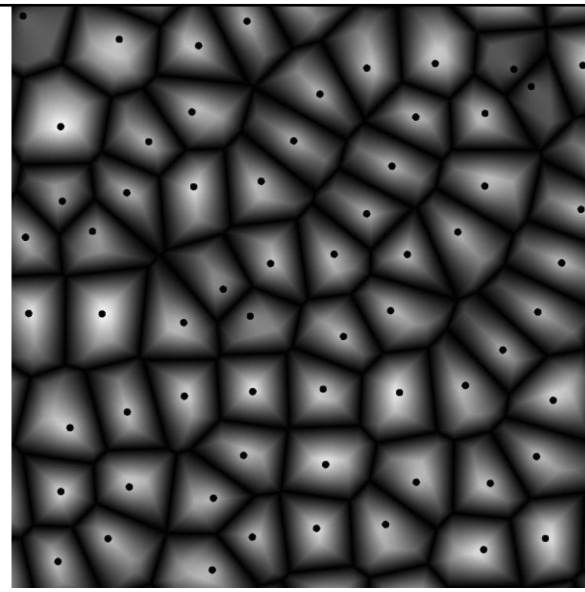
Multiple Octaves

Value noise Perlin noise Simplex noise



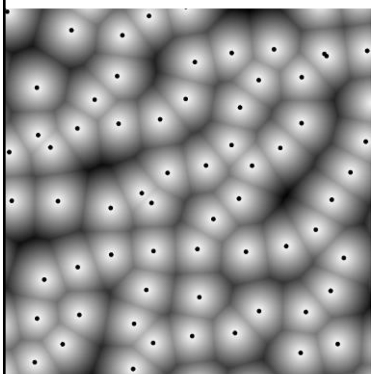
Worley noise

- Steven Worley, 1996
- Start with random points
  - Grid jittering
- Take distances to n-th closest
- With tweaking, can look like:
  - Water
  - Stone
  - Biological cells

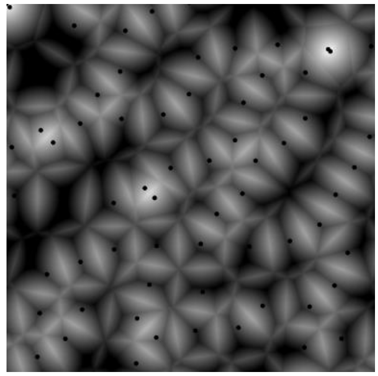


You put the points in a square grid. To get distance to nearest point, you only need to check squares neighboring the square your pixel lies in.

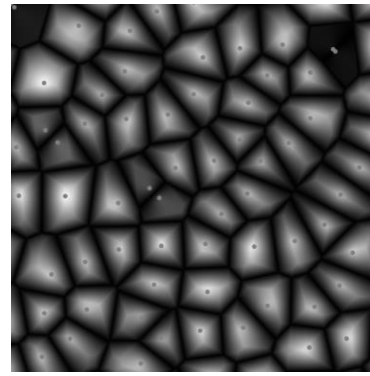
Different kinds of Worley noise



closest



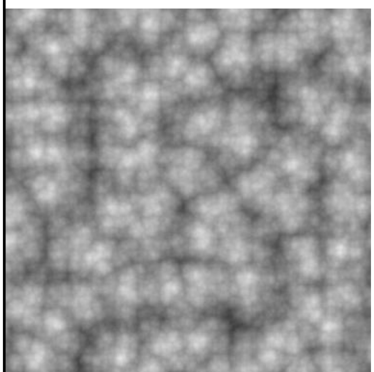
2nd closest



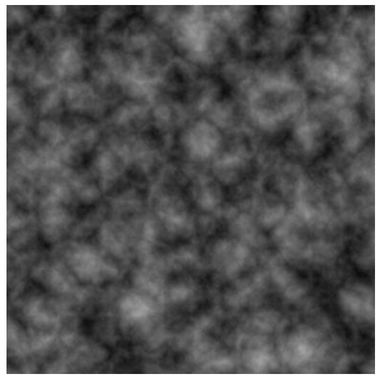
closest – 2nd closest

Notice that in the last one, the two components cancel each other in a whole “cell”.
This is due to two seed points being very close to each other.
In practice, you may want to “partial jitter” to avoid this.

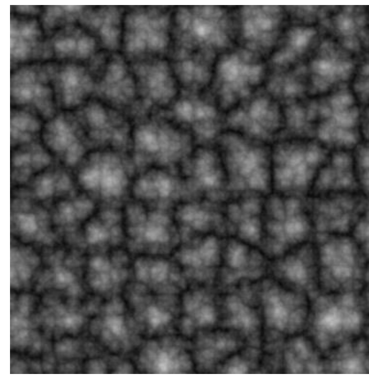
Worley noise with octaves



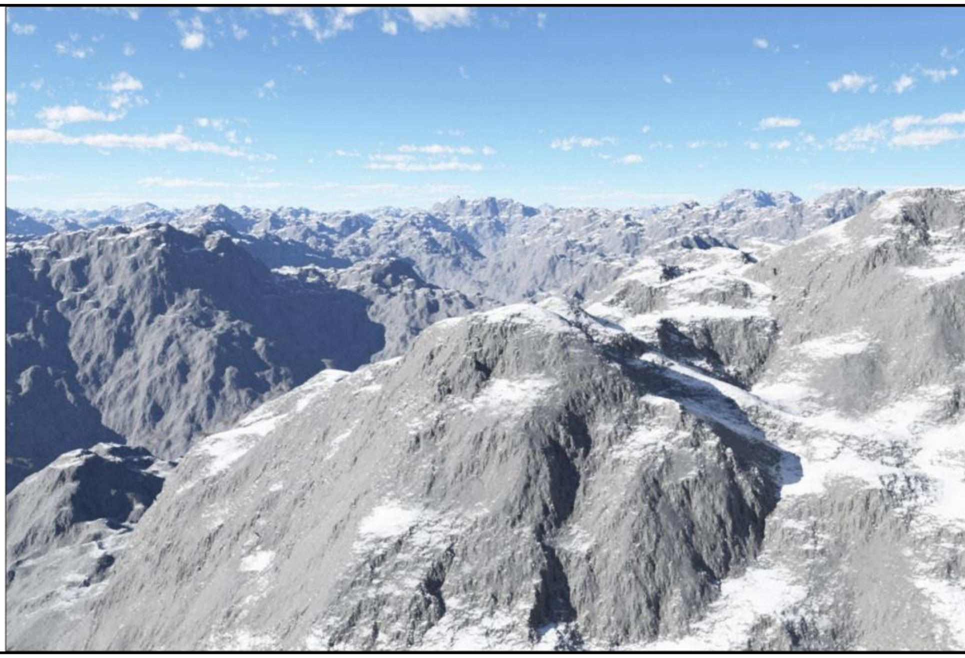
closest



2^{nd} closest

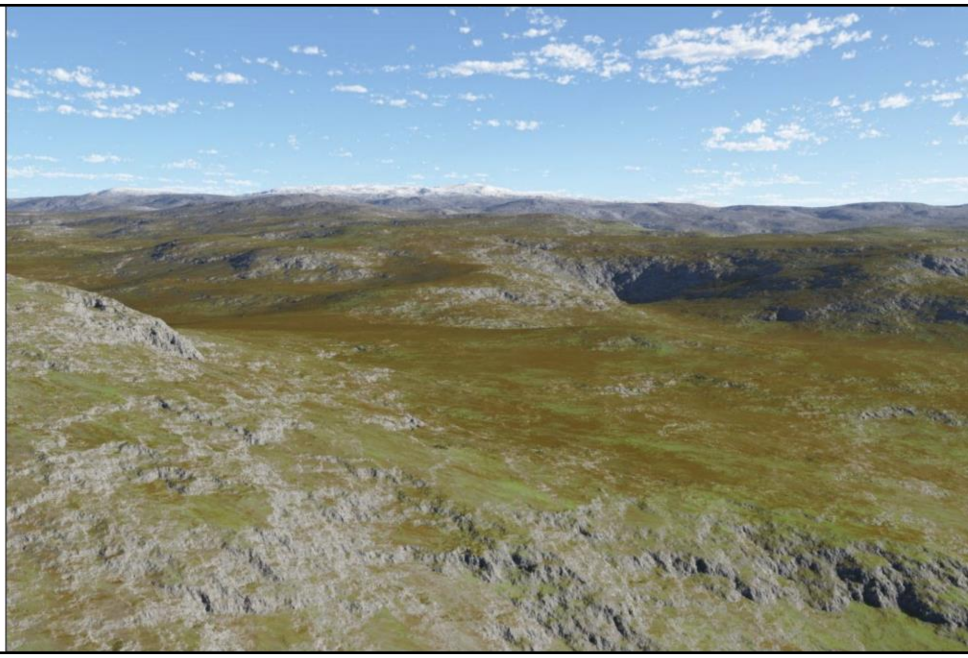


closest – 2^{nd} closest



Perlin noise (several octaves). Just using perlin noise is unrealistic – large gradients are almost everywhere. Can be somewhat offset by biomes.

Real-world photography of Totes Gebirge in Austria.



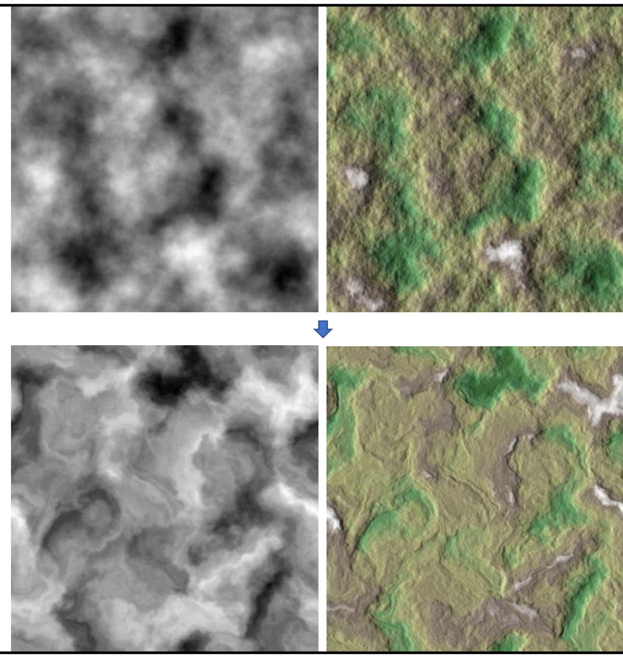
Ian Parberry, "Modeling Real-World Terrain with Exponentially Distributed Noise", *Journal of Computer Graphics Techniques*, Vol. 4, No. 2, pp. 1-9, 2015.
Much more realistic, but also more boring to explore (real world is boring).

Domain warping

- Inigo Quilez (iquilezles.org)
- Offset the positions by another noise function
- Better distribution of gradients

f(p) = noise(p + noise(p))

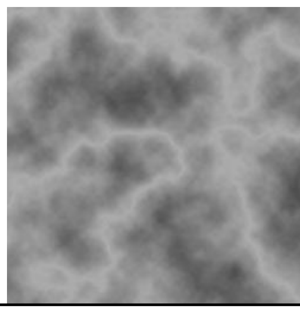
f(p) = noise(p + noise(p + noise(p)))

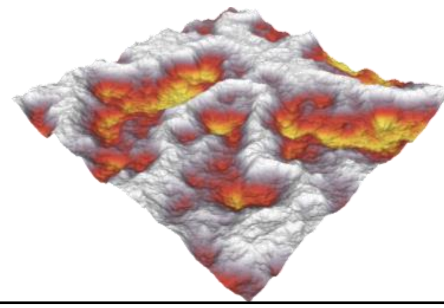


http://libnoise.sourceforge.net/tutorials/tutorial6.html
http://www.iquilezles.org/www/articles/warp/warp.htm
https://github.com/dandrino/terrain-erosion-3-ways

Ridged noise

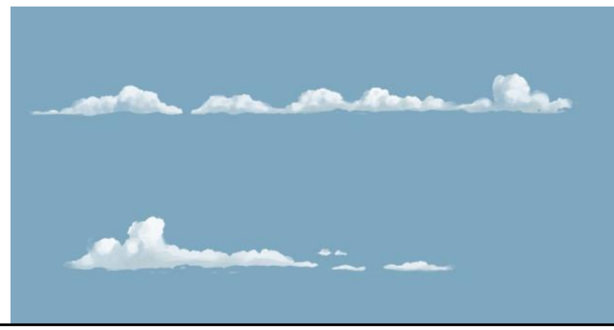
- Noise functions tend to generate smooth mountain tops, which isn't very realistic
- To combat, you can use an ABS function to noise in the range (-1, 1), and then invert (so you have ridged mountains instead of valleys)





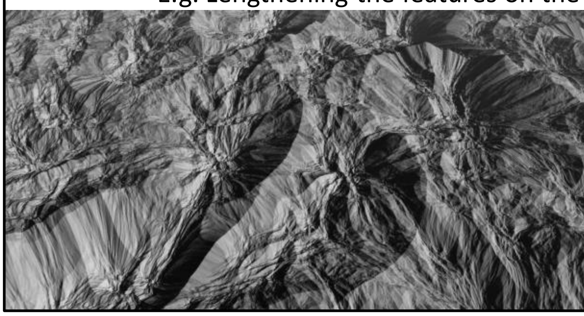
Billow noise

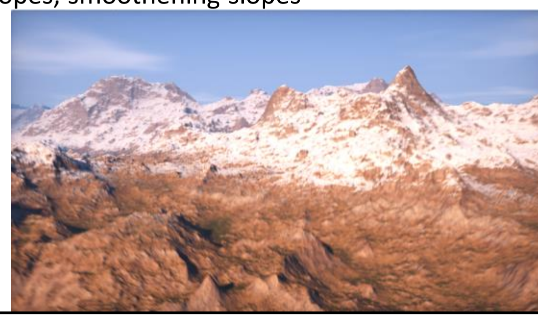
- Using just ABS is called billow noise
- Useful for rocks / clouds / …



Analytical derivatives

- Inigo Quilez
- Both value and gradient noise can be computed with derivatives
- Useful for lighting and even to modify noise in low-level octaves
  - E.g. Lengthening the features on the slopes, smoothening slopes





https://www.iquilezles.org/www/articles/morenoise/morenoise.htm
https://www.iquilezles.org/www/articles/gradientnoise/gradientnoise.htm
https://www.decarpentier.nl/scape-procedural-extensions

Advantages of noise functions

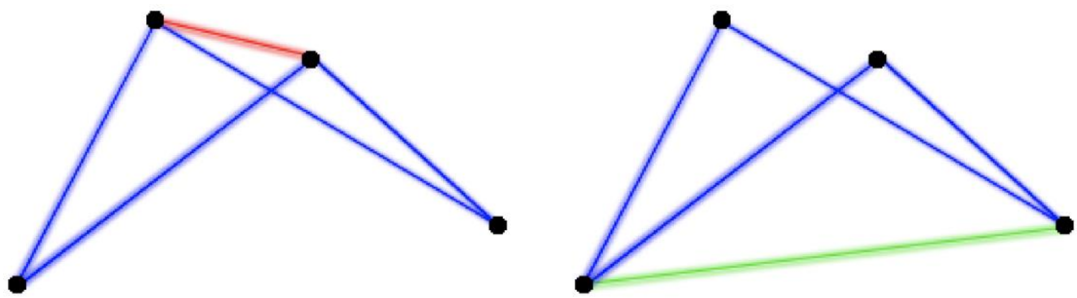
- Very low cost of resources
  - Usually nothing needs to be stored (!)
  - Computation is quite effective
- Modifiable, constrainable
- Combinable

To compute noise value without anything stored – hash the positions of points where you need to sample.

Limitations of heightmaps

- Cannot create caves, natural bridges, etc.
- Workarounds:
  - Only use heightmaps as a first step, then sculpt by other methods
    - May even treat “overhangs” as not terrain
  - Create e.g. three heightmaps, use one as bottom layer, second as ceiling, third as a new bottom layer
  - Don’t use heightmaps ☺

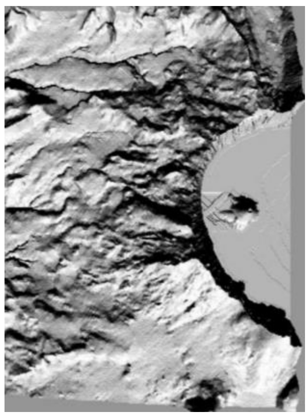
Limitations of heightmaps



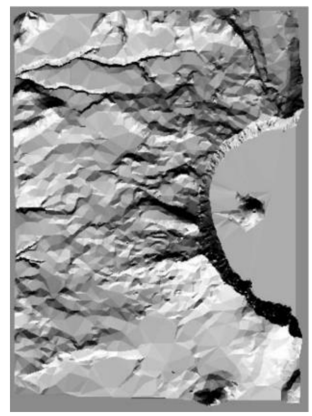
- Remember that there are two different triangulations of a square

Non-linear triangulations

- Michael Garland and Paul S. Heckbert (1995)



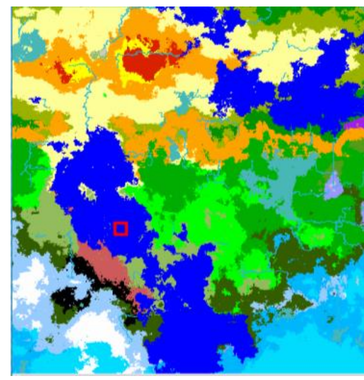
Original



Triangulated using 5% of points

Side-note: Biomes

- Large(r) scale partitions of game-world with different terrain properties
  - e.g. Mountains, Grasslands, Deserts, Sea, etc.
- Good to increase the variability of terrain

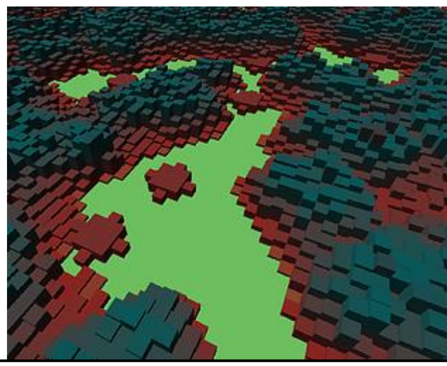

Biomes of Map from Outer Colony

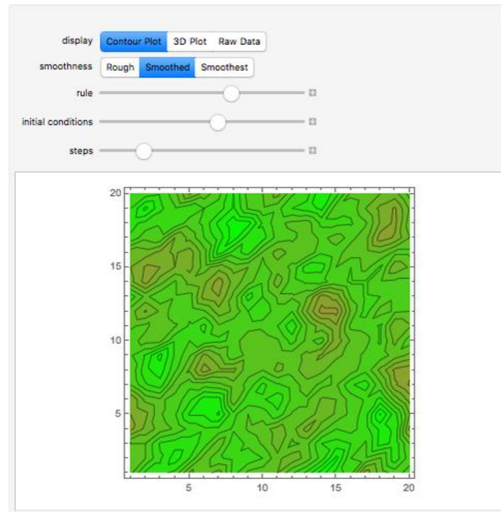
(Tagged) Map from Outer Colony

HeightMaps
Non-noise approaches

Terrain with Cellular Automata

- Jason Cawley, 2011
- 2018 ProcJam submission by twillacy



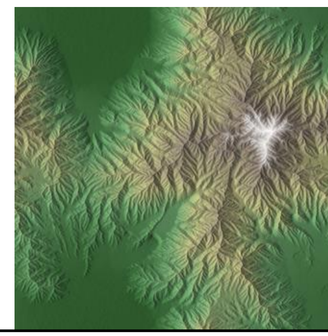


https://demonstrations.wolfram.com/AlgorithmicTerrainWithCellularAutomata/
https://itch.io/jam/procjam2018/rate/323026

Simulation

- Heightmaps generated by noise still don't look very real
- Real-life terrain is shaped by erosion

-> erosion simulation (wind / **water**)



Left picture – real world height map
Right picture – generated height map

Rarer approaches

- Geological simulation (erosion, tectonic uplift, …)
- Agent based simulation
- AI methods (e.g. GANs)
- Grammar based methods

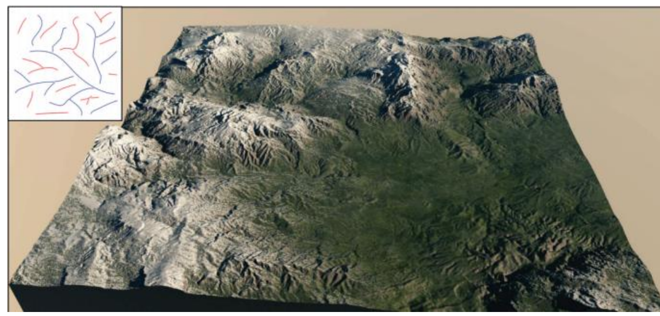


Figure 2 of *Interactive Example-Based Terrain Authoring with Conditional Generative Adversarial Networks* by Guérin et al.

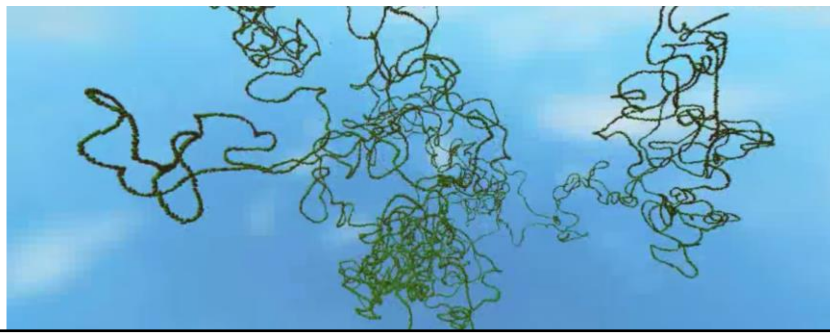
Non-heightmaps

Three-dimensional approaches

- Random walk
- Perlin worms
- Use 3D noise
- Cellular automata
- Agent-based approaches (miners)

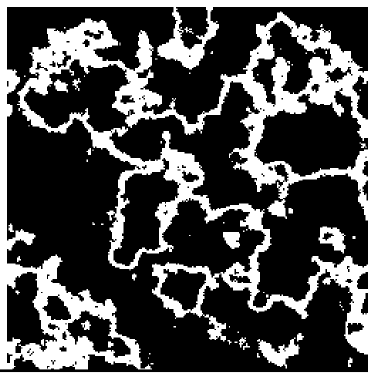
Random walks

- Perlin noise might be used to smoothen a random walk
- Map linear noise to 0-360º, and use it as a sequence of directions



Perlin worms

- Apparently what Minecraft and NMS use to generate caves
- (Small) threshold of noise

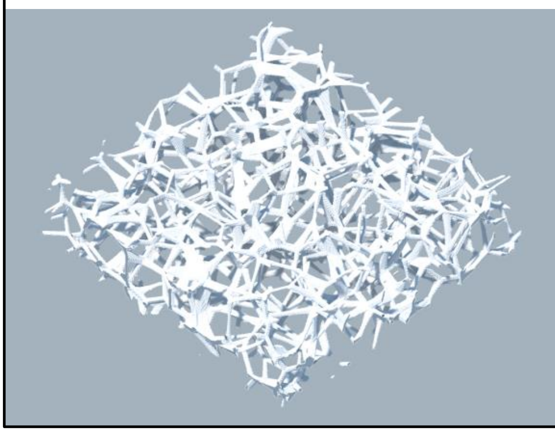


3D Noise

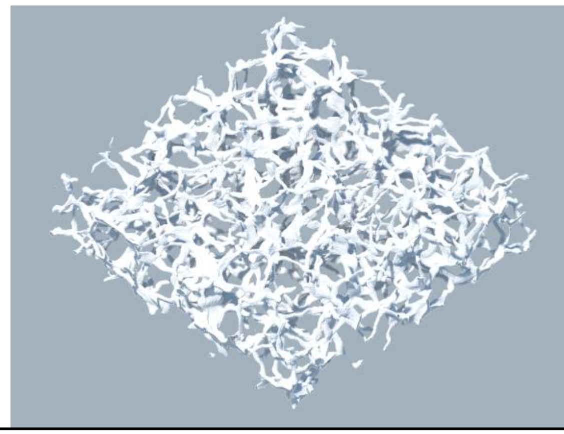
- Most of the previous notes on noise still apply
- The 3D noise is coupled with a threshold (or multiple) to select where there will be mass and where there won't
- Converting this 3D array to triangles is slightly more complicated
  - Marching Cubes algorithm
  - **Dual Contouring algorithm**
- Can be combined with 2D noise
  - No Man's Sky does this extensively

3D Worley noise

- Separation of planes into "cells"
- Use the edge of the cells



3D domain warped



https://www.reddit.com/r/proceduralgeneration/comments/94cwr0/procedural_caves_generated_using_worley_noise_in/

Summary

- Noise is your friend
- Modify, mix, experiment
  - 2D and 3D noise can work well together
  - Noise + other methods

Q & A

cerny@gamedev.cuni.cz
discord.gg/Zts98PGw6z