PRO_GEN

Procedural generation

Michael Gans
Tylor Lilley
Steve Moskal
Bob Tishma

Overview

Background

Art & Terrain

Noise

Game Generation

Applications in Unity

Summary

Questions

No Man’s Sky

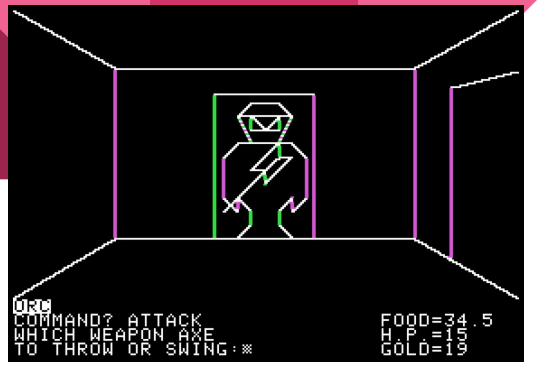
Background

- Method of creating data algorithmically rather than manually
- Computer graphics: commonly used for creating textures
- Video games: used for creating various other kinds of content
  - Examples: items, quests or level geometry.
- Common applications include (not limited to)
  - Smaller file sizes
  - Larger amounts of content than can be created manually
  - The inclusion of randomness for less predictable gameplay experiences
- Also been used for various other purposes and in other media

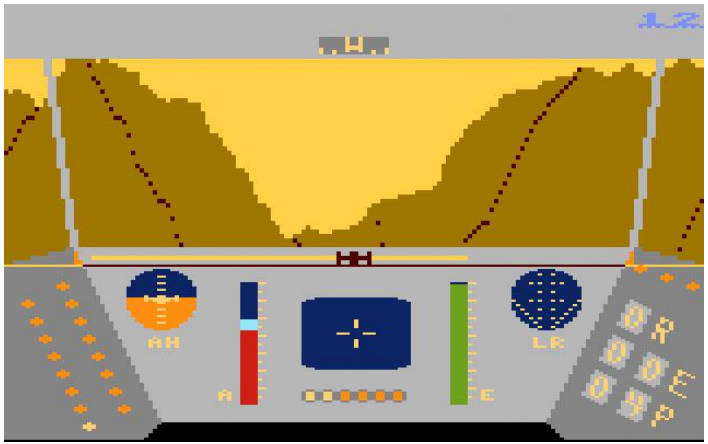
Past

- Procedural generation has been used in video games as early as the 70’s
  - **Roguelike** subgenre
  - Beneath Apple Manor
- Earliest Graphical Computer Game
  - limited by memory constraints forced content, such as maps, to be generated algorithmically on the fly
    - wasn’t enough space to store a large amount of pre-made levels and artwork
  - Pseudorandom number generators were often used with predefined seed values in order to create very large game worlds that appeared premade.

Past - Games

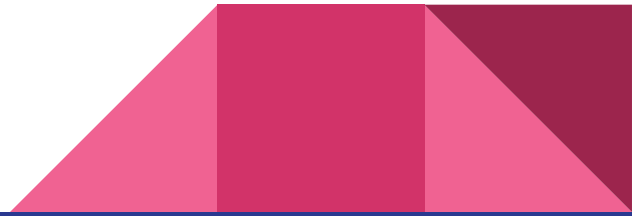


- Rouge (1980s)
- Akalabeth (1980)
- River Raid (1982)
- Elite (1984)
- Rescue on Fractalus (1985)
- The Sentinel (1986)



Moving Toward Present Day

- Hardware Advances
  - Able to store thousands of times as much data than was possible in the early 80s
- Content such as textures and character and environment models are created by artists beforehand
  - Keeps the quality of content consistently high.
  - Needs to be designed by hand (large games take hundreds of artists)
- Hybrid/Middleware
  - Pre-made with procedurally applied distortions
    - SpeedTree



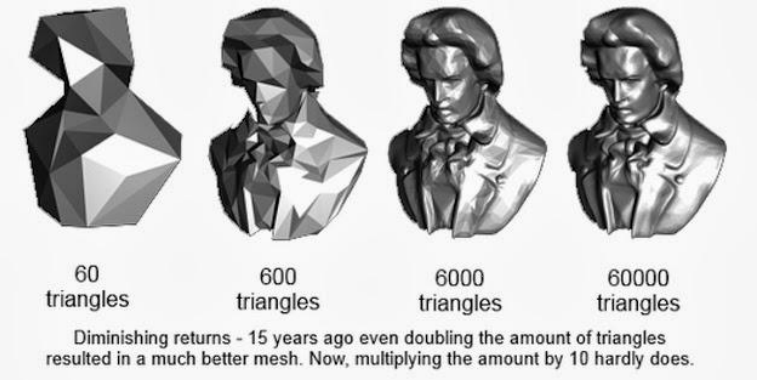
Present Day

- 2004
  - .kkrieger
  - RoboBlitz
  - Spore
- 2006
  - Dwarf Fortress
- 2008
  - Left 4 Dead
- 2009
  - Left 4 Dead 2
  - Minecraft
- 2014
  - Elite Dangerous

Moving Toward the Future

Handmade Vs. Procedural Generation

Detailed Vs. Freedom



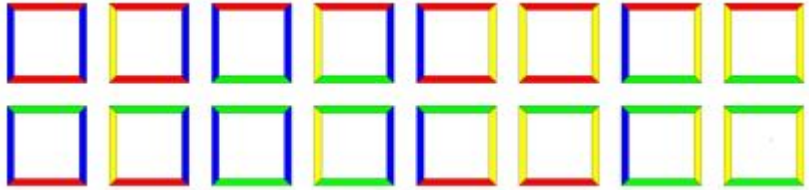
Future

- No Man's Sky
  - "a game about exploration and survival in an
  - infinite procedurally generated galaxy"
- New methods and applications are presented annually in conferences such as the IEEE Conference on Computational Intelligence and Games and Artificial Intelligence and Interactive Digital Entertainment

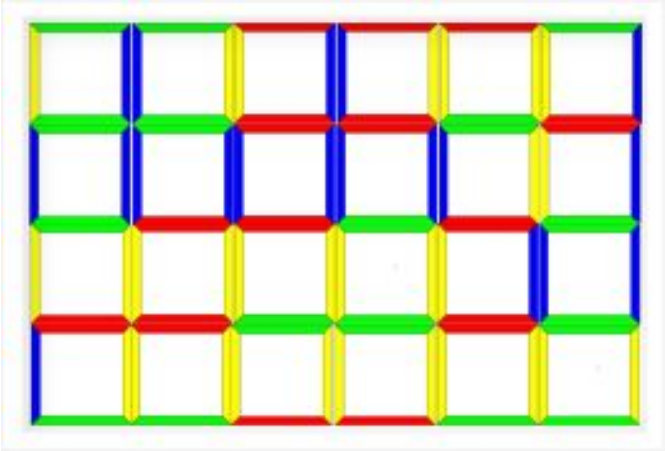
IEEE Computational Intelligence Society

Artificial Intelligence and Interactive Digital Entertainment

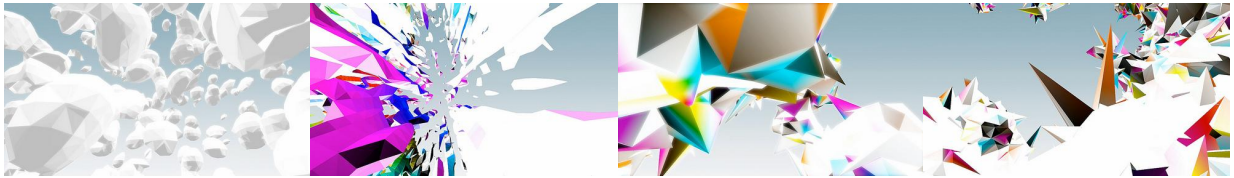
ART & Terrain



- Wang Tiles
  - N = Horizontal Borders & M = Vertical Borders
    - Complete Set: N^2M^2
    - Full Tiling: 2NM



- Fractals
  - Fractal Algorithms
    - Perlin Noise
    - Simplex Noise

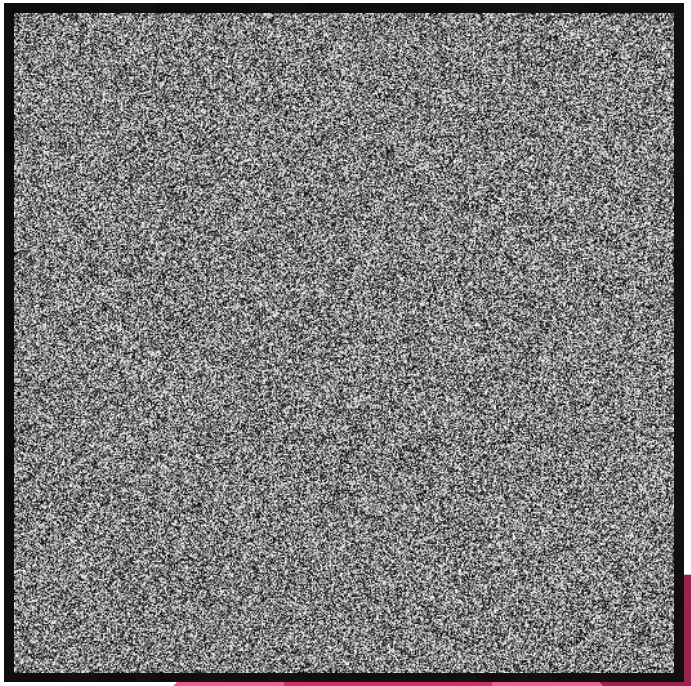


NOISE

- Noise is a building block for creating a variety of procedurally generated textures.
- Significantly, it can be used to simulate natural patterns from simple mathematical functions.
- Although noise can be used to generate patterns in any number of dimensions, I'll explain how 2D noise is used to create textures and heightmaps.

White Noise

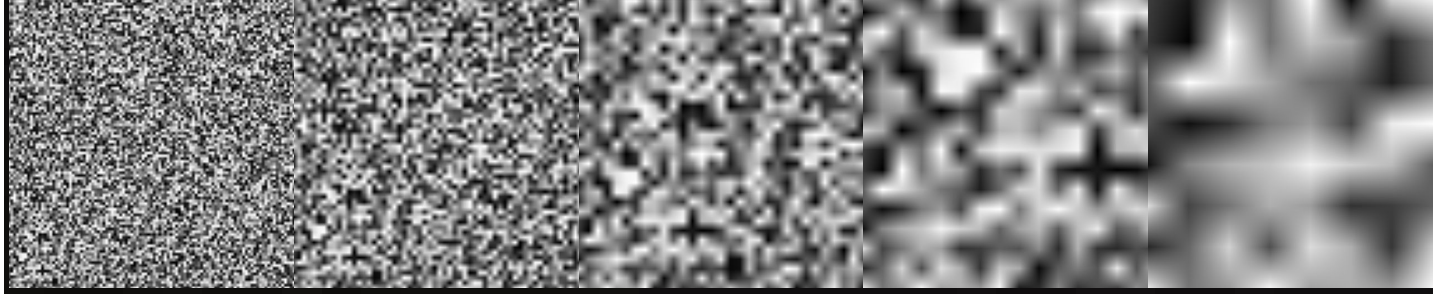
- The most basic type of noise is **white noise**.
- Generated by getting a random value [0,1] for each pixel.
- Ew, what an unrealistic texture.
- Unlike white noise, natural patterns should have smoothness as well as multiple levels of detail.



Interpolated Noise

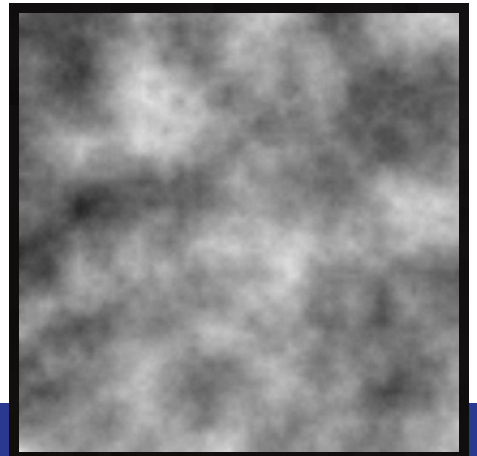
- Instead, let's make a 2D grid of points that has a lower resolution that the pixels of our image.
- Now we can interpolate between the four surrounding points, called lattices, for each pixel.
- Basic bilinear interpolation is a quick and simple way to do this.

Turbulence



- We can sum different sizes of our smooth noise to create a better texture.
- At each pixel, this sum is known as the turbulence at that point. Each pass is known as an octave.
- It actually looks pretty nice now. Unfortunately, you can still notice axis-aligned bias. To deal with this, we need to use a better smooth function than linear interpolation.

This texture is created by summing the above textures at varying scaled amplitudes.

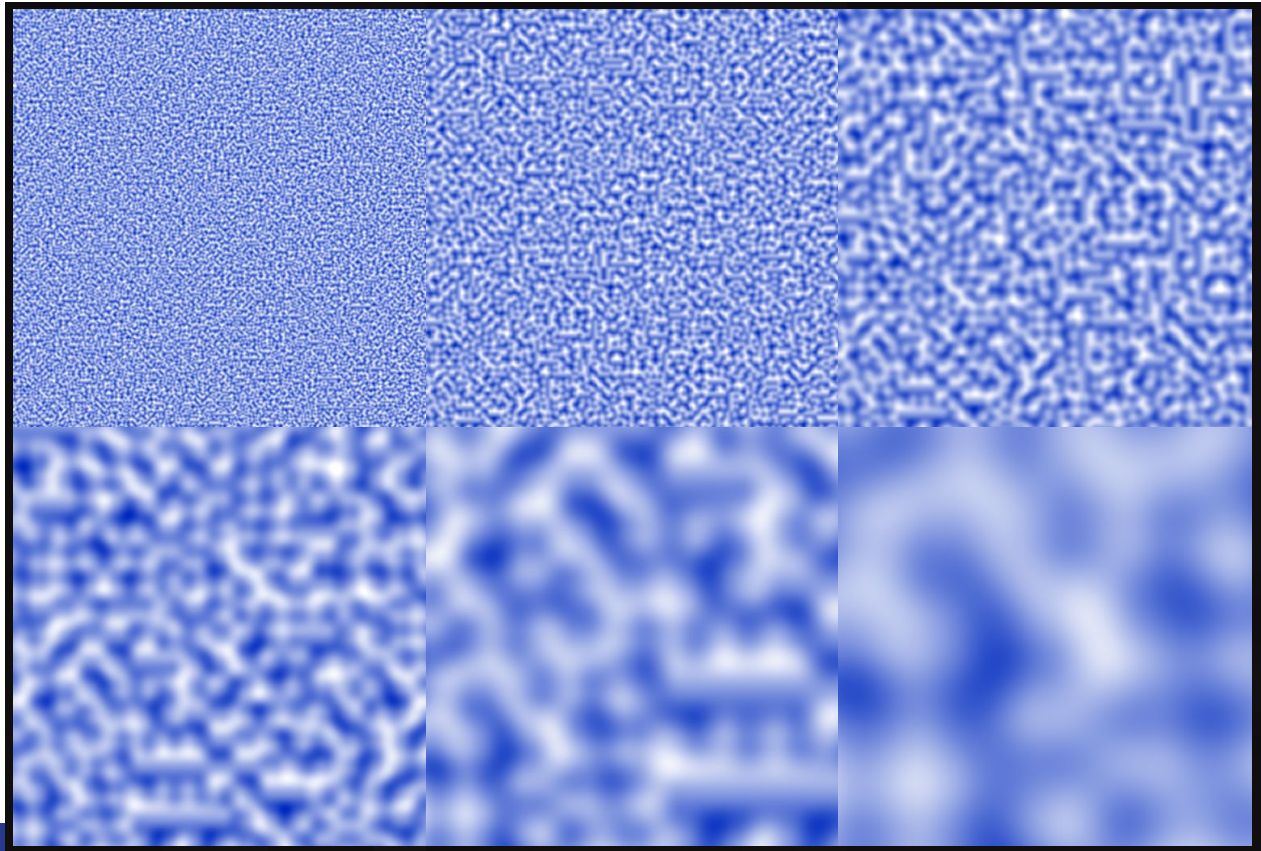


Perlin Noise

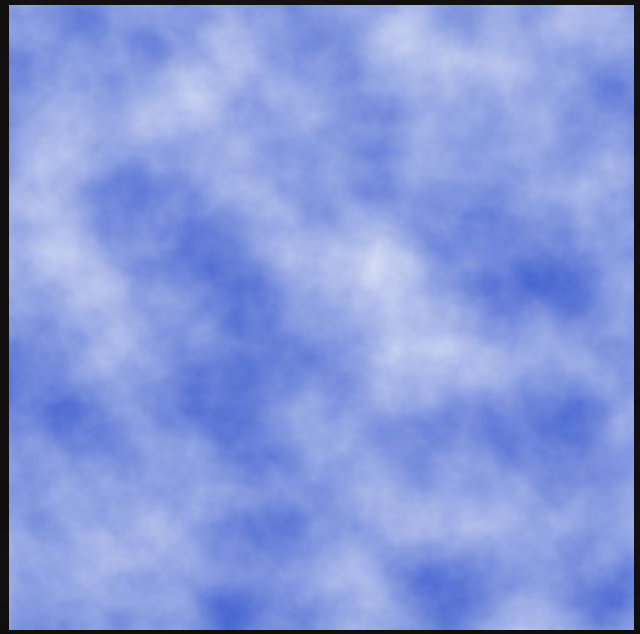
- Our previous technique used **value noise**. This means each lattice was assigned a single value [0,1].
- Perlin Noise, instead of using a single value, uses **gradient noise**. Each lattice is assigned a random N-dimensional unit vector.
- To interpolate a pixel to a gradient point, you use a vector pointing from the lattice to the pixel, and dot product it with the gradient vector.

Perlin Noise (above) looks a lot better and has less directional artefacts than the value noise with bilinear interpolation (below).

Clouds with Perlin Noise



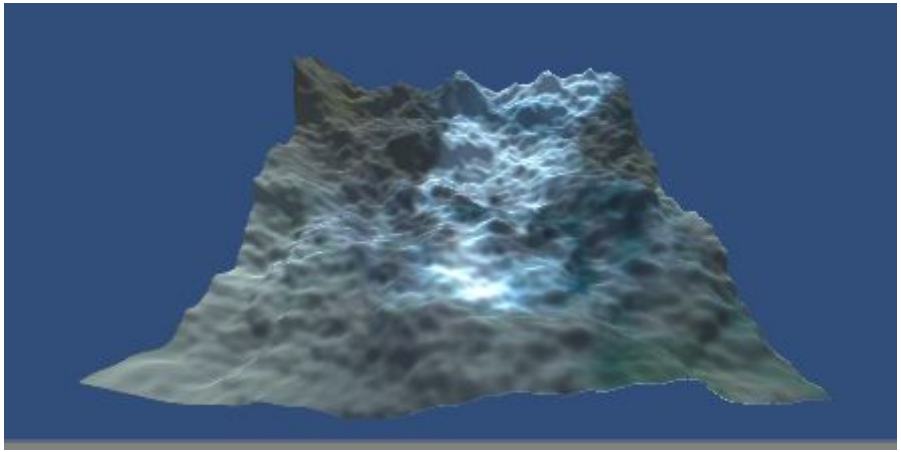
← These are summed together



to make this ↑

Terrain Generation with Noise

- It’s easy to apply our noise textures to heightmaps.
- Color at each pixel represents height.
- The cloud texture is an excellent representation of mountains.



Terrain Generator Code

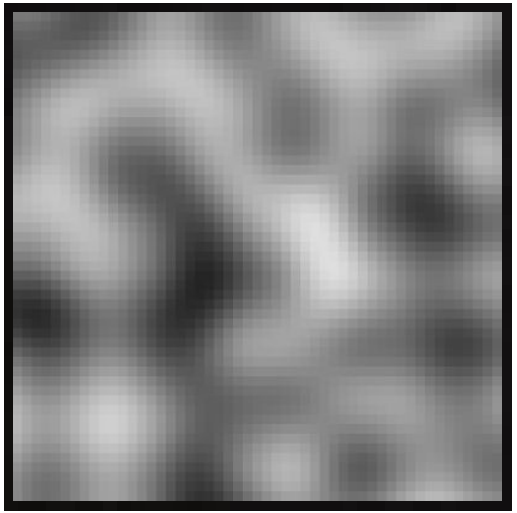
```
5  public class Noisey : MonoBehaviour {
6
7      int width = 250;
8      int length = 250;
9      int octaves = 6;
10     float horizontalScale = 6f;
11     float verticalScale = 200f;
12
13     void Start () {
14         Vector3[] vertices = new Vector3[width * length];
15         Vector2[] uv = new Vector2[width * length];
16         int[] triangles = new int[(width - 1) * (length - 1) * 6];
17
18         /* initialize vertices/uv/triangles */
19         for (int z = 0; z < width; z++) {
20             for (int x = 0; x < length; x++) {
21                 int index = x + z*width;
22                 vertices[index] = new Vector3(x,0f,z);
23                 uv[index] = new Vector2((float)x/(width-1),(float)z/(length-1));
24             }
25         }
26         for (int i = 0; i < triangles.Length; i+=6) {
27             int offset = i*width/(6*(width-1));
28             triangles[i]   = offset;
29             triangles[i+1] = offset+width;
30             triangles[i+2] = offset+1;
31             triangles[i+3] = offset+1;
32             triangles[i+4] = offset+width;
33             triangles[i+5] = offset+width+1;
34         }
```

Terrain Generator Code

```
35
36          /* do noise */
37          for (int i = 0; i < octaves; i++) {
38              for (int z = 0; z < width; z++) {
39                  for (int x = 0; x < length; x++) {
40                      float frequency = (1<<i)*horizontalScale;
41                      float amplitude = 1<<(octaves-i);
42                      float height = Mathf.PerlinNoise (x / frequency, z / frequency) / amplitude;
43                      vertices[x + z*width].y += height*verticalScale;
44                  }
45              }
46          }
47
48          /* apply to gameobject */
49          Mesh terrainMesh = new Mesh ();
50          terrainMesh.vertices = vertices;
51          terrainMesh.uv = uv;
52          terrainMesh.triangles = triangles;
53          terrainMesh.RecalculateNormals();
54          (this.GetComponent<MeshFilter>() as MeshFilter).mesh = terrainMesh;
55
56      }
57
```

Terrain Generation with Noise

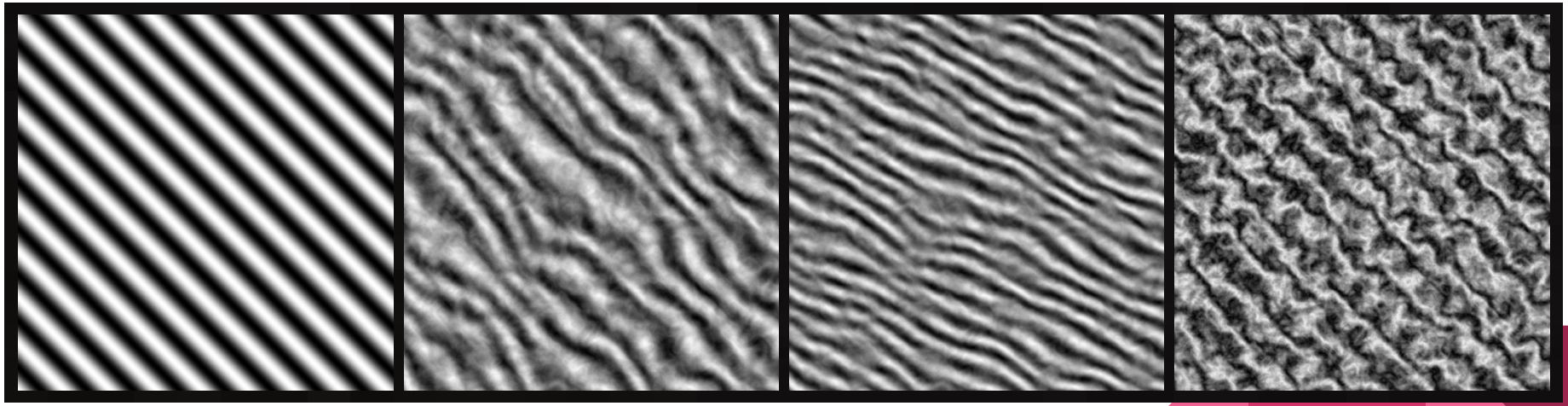
- Using a separate noise gradient, we can also define different areas of the terrain to have exclusive parameters.



In this low-resolution map, pixels with a color value <0.35 are where we place buildings.

Cool Texture Examples

Using math functions to generate simple textures as a base, we can add noise to make some nice looking stuff.



Pixel at x,y = sin(x+y)/scale

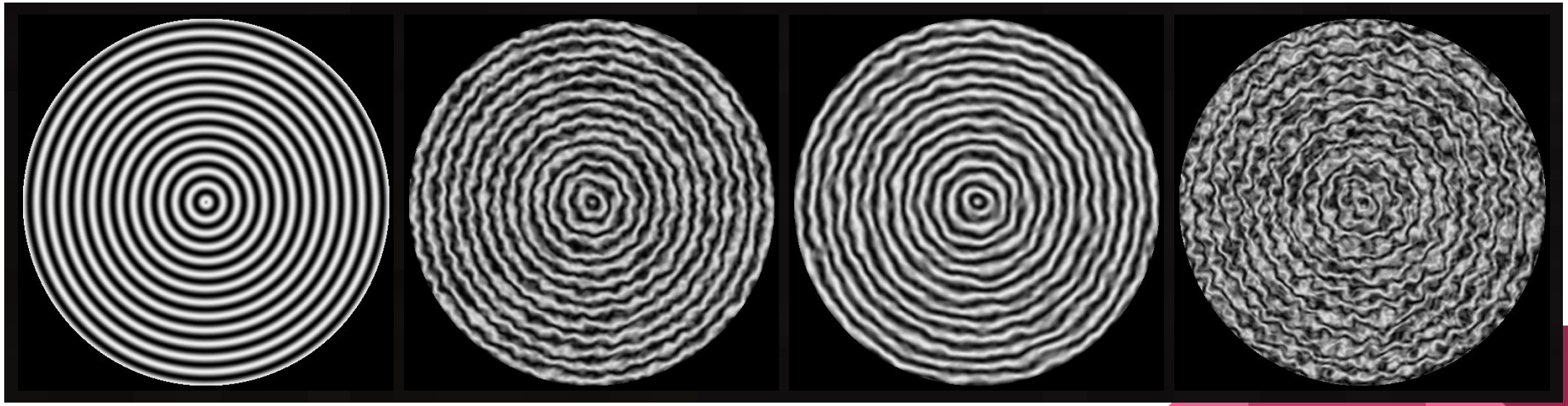
Marble-esque

Waves or sand dunes

carpet

Cool Texture Examples

Using math functions to generate simple textures as a base, we can add noise to make some nice looking stuff.



Pixel at x,y = sin(sqrt(x^2+y^2)/scale

Diseased-looking tree

Nice slice of wood

Even more diseased tree

Gaming

Content: Diablo, ESV: Skyrim, Borderlands

Terrain: ESII: Daggerfall, Scorched Earth

Both: Minecraft, Terraria, No Man’s Sky

Content

Skyrim - Dynamic Quest System

Diablo - Enemies, Items

Borderlands - Gun Creation

Terrain

The Elder Scrolls II: Daggerfall - 1996

Huge, mostly empty generated world

- Empty, meaningless

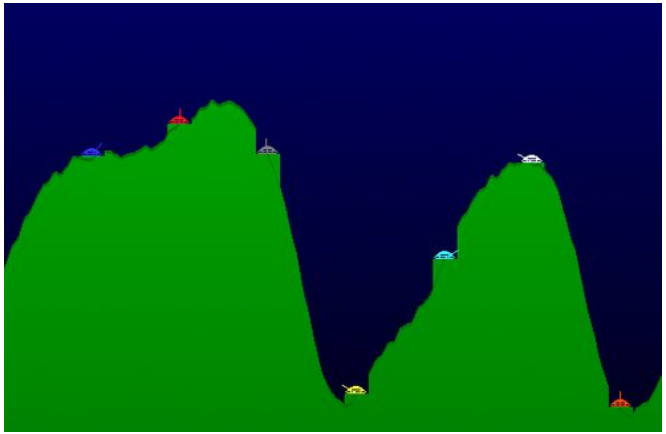
Massive maze like dungeons

- One is bigger than the world of ESIII: Morrowind

Terrain

Scorched Earth: 1991

1d noise (“Proper Function”)



Terraria: 2011

Terraria: 2D noise (“Polar”)

Minecraft: A Case Study

MineCraft : 2009

- Structures
  - Villages
  - Ruins
  - Strongholds
- Items
  - Chest-spawned
  - NPC-traded
  - Enemy-dropped

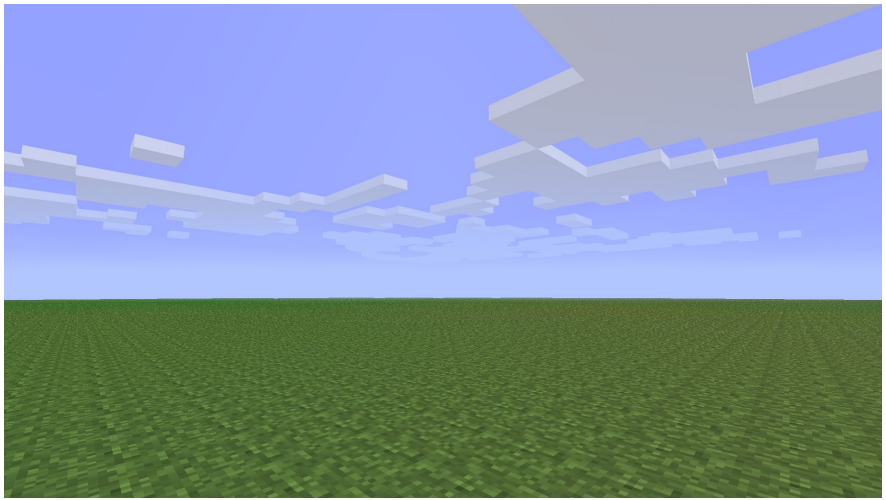
Terrain

Minecraft : 2009

- Uses 3d Perlin Noise and Interpolation
- Values based on “seed”
- Values < n represent land where Values >=n represent air
- Biomes assessed via graph
- Features added at end

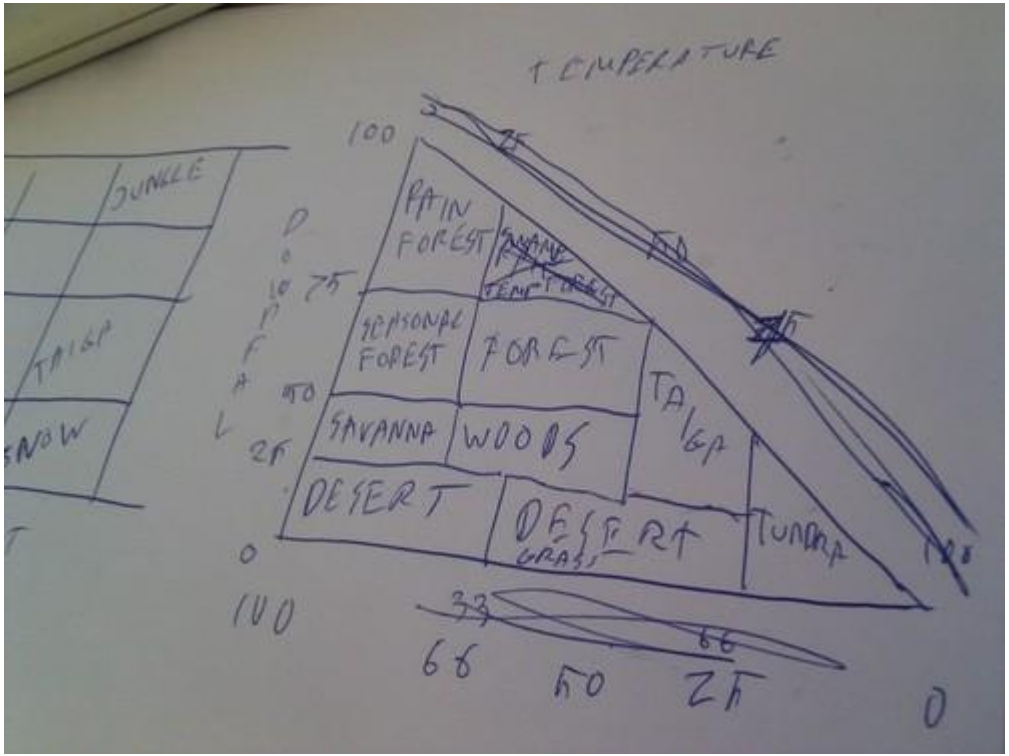
Minecraft - Sculpting a world

1. Generate landscape
2. Biome designation
3. World details
4. Add structures



Biomes

- Generated based on graph
  - bordering biomes are logical
  - temp vs rainfall
- Can alter elevation
  - deserts are flat, etc
- Can be separated by river
- Alters spaws



Post-Processing

Features: minecraft-generated caves, ravines, lakes, lava lakes

Ores: spawn based on parameters

Structures: Villages, strongholds, temples

Villages are created by expanding outward from a well

Custom World Settings

Perlin Noise Octaves

Noise 1 Octaves: 1	Noise 2 Octaves: 1	Noise 3 Octaves: 1
Noise 4 Octaves: 1	Noise 5 Octaves: 1	Noise 6 Octaves: 24

Terrain Stitching Parameters

X Lerp Factor: 0.00	Z Lerp Factor: 0.00
Y Lerp Factor: 0.00	Solid Cutoff Factor: 0.00

< Noise & Stitching Data [1/12] >

Done | Randomize | Defaults | Preview | Back

- One octave active
  - Gradual
  - Smooth
  - No “anomalies”
- No interpolation
  - “Blocky”
- Features
  - caves
  - lakes
  - biomes


Custom World Settings
Perlin Noise Octaves
Noise 1 Octaves: 1
Noise 2 Octaves: 1
Noise 3 Octaves: 1
Noise 4 Octaves: 1
Noise 5 Octaves: 20
Noise 6 Octaves: 20
Terrain Stitching Parameters
X Lerp Factor: 0.00
Z Lerp Factor: 0.00
Y Lerp Factor: 0.00
Solid Cutoff Factor: 0.00
<
Noise & Stitching Data [1/12]
>
Done
Randomize
Defaults
Preview
Back

- Two octaves active
  - more interesting boundaries
  - less predictable
  - more anomalies

Perlin Noise Octaves

Noise 1 Octaves: 1 | Noise 2 Octaves: 1 | Noise 3 Octaves: 1

Noise 4 Octaves: 1 | Noise 5 Octaves: 20 | Noise 6 Octaves: 20

Terrain Stitching Parameters

X Lerp Factor: 0.24 | Z Lerp Factor: 0.24

Y Lerp Factor: 0.00 | Solid Cutoff Factor: 0.00

< Noise & Stitching Data [1/12] >

Done | Randomize | Defaults | Preview | Back

- Same Perlin Noise
- Interpolation in x and z axes
- No y “lerp”
- Features remain generally uneffected

Perlin Noise Octaves

Noise 1 Octaves: 3 | Noise 2 Octaves: 3 | Noise 3 Octaves: 3

Noise 4 Octaves: 3 | Noise 5 Octaves: 3 | Noise 6 Octaves: 15

Terrain Stitching Parameters

X Lerp Factor: 0.24 | Z Lerp Factor: 0.24

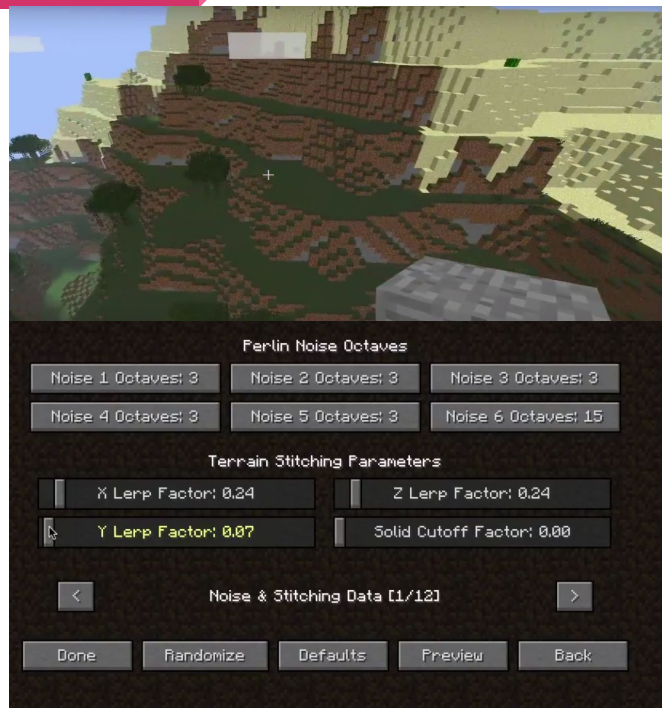
Y Lerp Factor: 0.00 | Solid Cutoff Factor: 0.00

< Noise & Stitching Data [1/12] >

Done | Randomize | Defaults | Preview | Back

- All 6 octaves active
  - large terrain features given highest magnitude
  - perturbed by lower octaves
  - Scaled up
- More “funny” Characteristics
  - boundaries “tucking” in on themselves
  - anomalies
  - differences more drastic

Y-Lerp

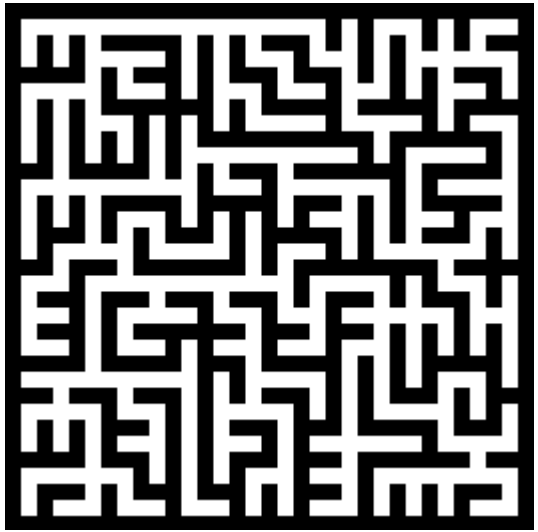

Perlin Noise Octaves
Noise 1 Octaves: 3
Noise 2 Octaves: 3
Noise 3 Octaves: 3
Noise 4 Octaves: 3
Noise 5 Octaves: 3
Noise 6 Octaves: 15
Terrain Stitching Parameters
X Lerp Factor: 0.24
Z Lerp Factor: 0.24
Y Lerp Factor: 0.07
Solid Cutoff Factor: 0.00
Noise & Stitching Data [1/12]
Done
Randomize
Defaults
Preview
Back


Perlin Noise Octaves
Noise 1 Octaves: 3
Noise 2 Octaves: 3
Noise 3 Octaves: 3
Noise 4 Octaves: 3
Noise 5 Octaves: 3
Noise 6 Octaves: 14
Terrain Stitching Parameters
X Lerp Factor: 0.24
Z Lerp Factor: 0.24
Y Lerp Factor: 0.24
Solid Cutoff Factor: 0.00
Noise & Stitching Data [1/12]
Done
Randomize
Defaults
Preview
Back

Final World

- All other options remained constant throughout

Applications in Unity



An Example Usage: Generating a Maze

- Depth First Search
- Kruskal’s Algorithm
- Animation of Prim's Algorithm at work

Basic Overview of Prim’s Algorithm

1. Start with a grid full of walls and unmarked cells.
2. Pick a cell, mark it as part of the maze. Add the walls of the cell to the wall list.
3. While there are walls in the list:
    - Pick a random wall from the list. If the cell beyond that wall isn't in the maze yet:
        - Destroy the wall and mark the cell on the opposite side of it
        - Add the neighboring walls of the cell to the wall list.
    - Remove the wall from the list.

Code Snippet

Cell Object Prefab:

```
using UnityEngine;
using System.Collections;

public class Cell : MonoBehaviour {

    public bool visited;
    public GameObject north;
    public GameObject east;
    public GameObject west;
    public GameObject south;
    public GameObject space;

}

```

Source Code

Let’s Look at the actual algorithm in Unity and see it work!

Other Resources

- SpeedTree Example
  - https://www.youtube.com/watch?v=Dh5DKrsXNc8
- Cave Generation Tutorial
  - https://www.youtube.com/watch?v=v7yyZZjF1z4
- Generating Procedural Dungeon
  - https://www.youtube.com/watch?v=ySTpjT6JYFU
- Rooms With Holes
  - http://procworld.blogspot.com/2012/03/building-rooms.html
- Procedural Texture Mapping Example
  - https://www.youtube.com/watch?v=LjotNeyFtOo

Summary

Background
Art & Terrain
Noise
Minecraft
Applications in Unity

Questions?

Thank You