

Terrain Engine 2D

A 2D Block Engine for Unity

Out now on the Unity Asset Store

[BUY NOW!](#)

FEATURES

DOCUMENTATION

API

FAQ

DEMO

EXAMPLE PROJECT

Terrain Engine 2D

User Manual - V1.20

INTRO

GENERAL

MAIN PROPERTIES

Terrain Generation

This page explains how terrain is generated in the engine as well as including some in depth information on how to create a custom TerrainGenerator script for procedurally generating the world.

Table of Contents

- General
- The Framework
- Creating a custom TerrainGenerator script

General

Behind the scenes the terrain is simply an array of data where each element represents a block, and each block has a number of different properties that define how it will be rendered and interacted with in the world. This data is arranged in separate block layers. Each layer stores an array of [Block Types](#), [Bitmasks](#), [Variations](#), and [Render Blocks](#). These arrays index and store the information for every single grid unit/tile position of the world. All of this data is handled by the block engine, there are various levels of abstraction making it easy for the developer to make changes to the Terrain Data without worrying about all the extra complexity.

- Block Type** The type of block as defined in the world custom inspector
- Bitmask** The bitmask of the block for rendering
- Variation** The variation of the block texture
- Render Blocks** Whether this block should be rendered

In Terrain Engine 2D, the Terrain can be generated in the inspector, or at runtime. It can be dynamically modified in game using the included tools, or through any custom scripts which you may create. The terrain is generated using a custom TerrainGenerator script which is very powerful and allows you to procedurally generate complex terrain with ease.

Framework

Terrain Engine 2D contains a framework for procedurally generating the terrain. This framework will handle the generation of the bitmasking values, block variations, and the render blocks, but it requires a custom Terrain Generator script to set up the location of the block types, and any fluid data. You as the developer have full control over how all the blocks are placed in the terrain. When you create a custom Terrain Generator script must extend the base TerrainGenerator script which includes many helper functions so you do not have to access any variables directly.

TerrainGenerator Script

The TerrainGenerator script is a base MonoBehaviour script which is meant to act as an extension class for creating a custom Terrain Generator script. This class makes it very easy to setup the block data for the World. This script can be found in the asset package under:

TerrainEngine2D/Assets/Scripts/Terrain/TerrainGenerator.cs.

Helper Functions

- GenerateData** This is where all the block data is set
- PerlinNoise** Gives the perlin noise value at a specific coordinate
- SetBlock** Sets the block type of a specified layer to a specific block
- DoAddBlock** Determines if a block should be placed based on a probability factor
- RemoveBlock** Removes a block from a specified layer
- RemoveAllBlocks** Removes blocks from all layers at a specific coordinate
- IsBlockAt** Checks if there is a block at a specific coordinate or area
- AddFluid** Adds fluid to a Fluid Block
- RemoveFluid** Removes all fluid from a Fluid Block
- GeneratePool** Generates a pool of fluid
- ClearFluid** Removes all fluid from blocks below a threshold (floodfills the area, stops at terrain blocks)

Creating a custom Terrain Generator script

Script Setup

- Start by creating a new c# script or using the TerrainGeneratorTemplate script as a starting point. If you choose to use the template some steps are already completed for you.
- Inherit the TerrainEngine2D namespace by adding: `using TerrainEngine2D;` at the top of the script.
- Extend the TerrainGenerator script in the class declaration: `public class CustomTerrainGenerator : TerrainEngine2D.TerrainGenerator`
- Override the GenerateData function and call the base function:

```
public override void GenerateData()
{
    base.GenerateData();
}
```

- (Optional but recommended) Add a Layers enum containing the name of all your layers. For example:

```
public enum Layers { Background, Trees, Decoration, Main, Ore, Foreground}
```

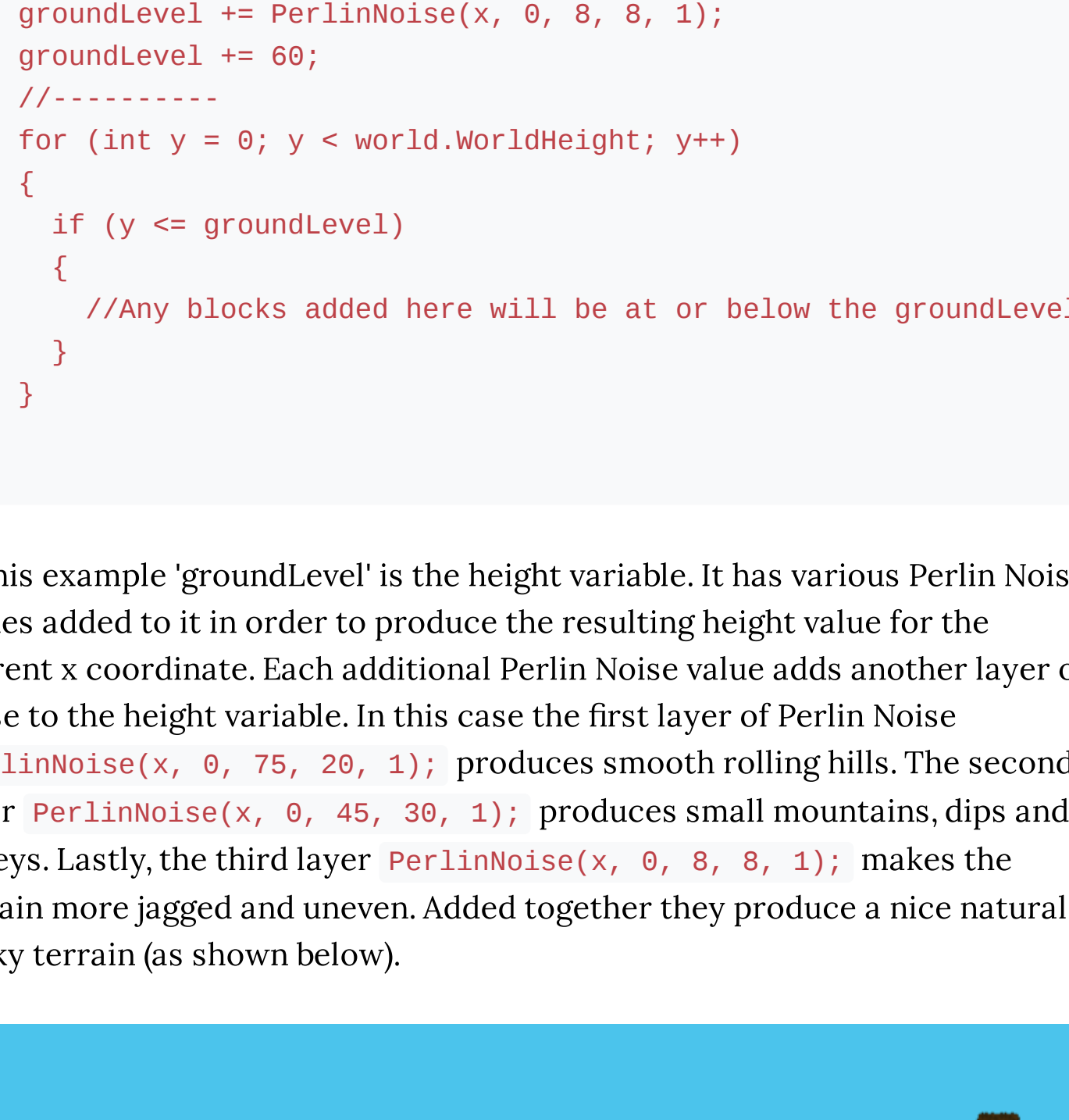
- (Optional but recommended) Add enums for all your Layers containing the name of all its Block Types. For example:

```
public enum Background { Dirt, Stone, Wood }
```

- (Optional but recommended) Add an enum for all your Fluid Types. For example:

```
public enum FluidType { Water, Poison, Lava }
```

- Now you can start adding to the GenerateData function and begin generating your block data ([next section](#)).
- When you are finished with your script, ensure everything is saved, and then add it as a component to the World GameObject (the Terrain Generator Script field will automatically populate).



Terrain Generator Component

Generating Block Data

As the developer you have full control over generating the block data for your terrain. This means there is no proper or correct way to code this section, it is up to you to figure out your own cool and unique ways of generating terrain. You can use any kind of noise you wish, algorithms, libraries, etc. That being said everything you need to know to get started can be found below and the included TerrainGeneratorExample script gives you further detail and examples.

Pass

Block data is generated in Passes, inside each Pass is where you can modify the block data. A Pass is where you loop through all of the blocks in the world to modify the block data. You can have as many passes as you wish inside the GenerateData function. A pass looks something like this:

```
//Pass
for (int x = 0; x < world.WorldWidth; x++)
{
    for (int y = 0; y < world.WorldHeight; y++)
    {
        //-----Set block data here-----

        //-----

    }
}
```

Modifying Block Data

Block data modification refers to adding and removing blocks and/or fluid. Block data is modified inside a Pass which loops through every block in the World. This means anything you do inside the Pass is applied to every single block in the world. Generally we do not want to set every block to the same type, so to avoid this we use conditional statements. These conditional statements make use of noise functions, random functions, and other values or functions in order to decide whether to add or remove a block.

To add or remove blocks there are two helper functions:

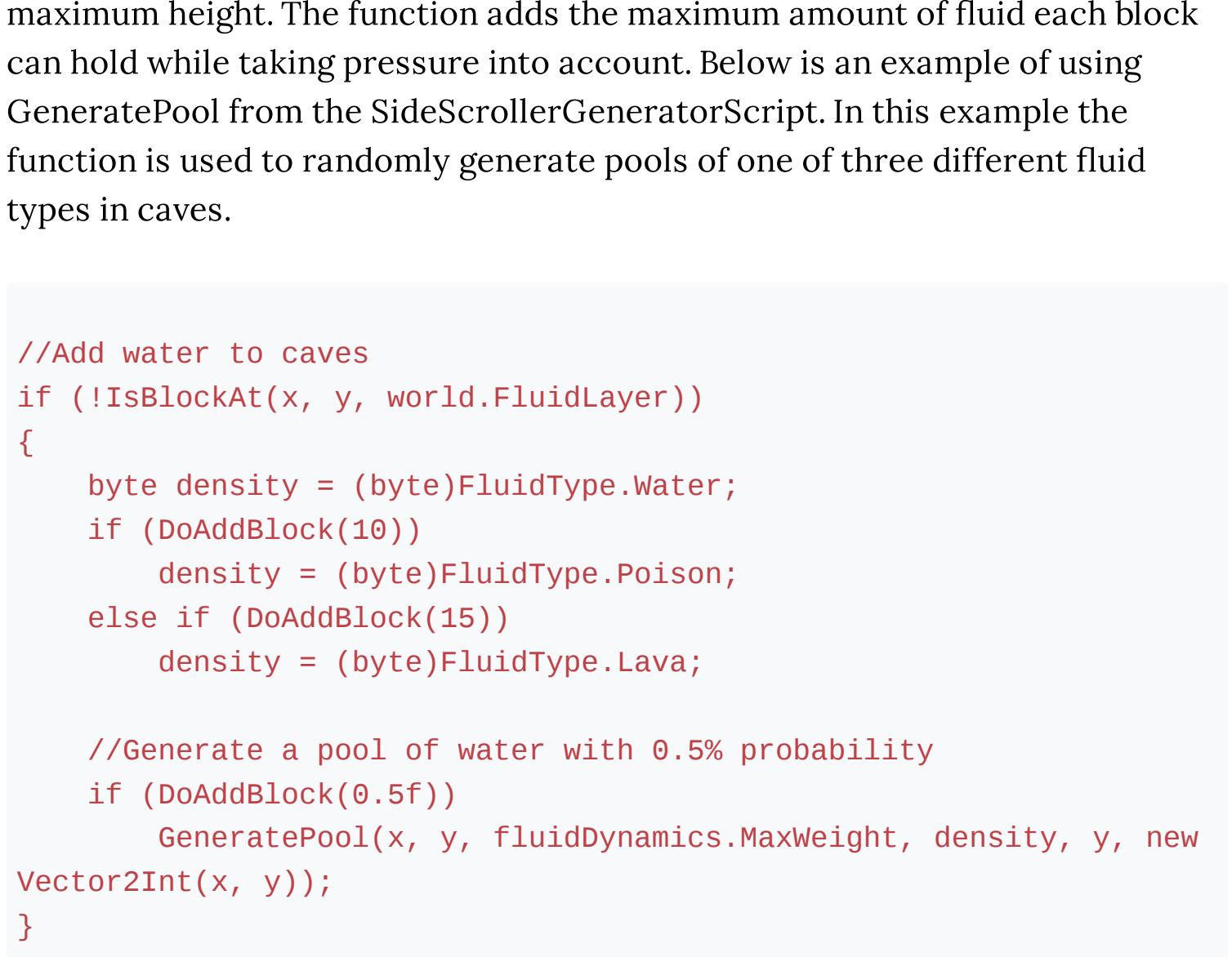
```
SetBlock(int x, int y, byte layer, byte blockType, float probability = 100f)
RemoveBlock(int x, int y, byte layer)
```

When calling these functions you need to specify which Layer and if you are adding a block which Block Type is being set. The functions take this data as a byte referring to the index of the element in the Layers/Block list of the World custom inspector. It is recommended to use Enums in your script to keep track of all your Layers and Block Types.

Example - Removing clumps of blocks to create caves

```
SetBlock(x, y, (byte)Layers.Background, (byte)Background.Dirt);
SetBlock(x, y, (byte)Layers.Main, (byte>Main.Dirt);

if (PerlinNoise(x, y, 10, 10, 1) > 5)
{
    RemoveBlock(x, y, (byte)Layers.Main);
}
```



Terrain Engine 2D Terrain Caves

Perlin Noise

Perlin Noise is a type of gradient noise generated through the Perlin Noise algorithm. It is a popular algorithm used in many different applications, one of those being procedural generation of graphics. Unity contains a built in function 'Mathf.PerlinNoise' which can be used to get values of Perlin Noise from a given x and y float value. The TerrainGeneratorExample scripts show how you can use Perlin Noise to procedurally generate data. There is also a helper function called PerlinNoise which makes it easier to manipulate and generate Perlin Noise values.

Perlin Noise can be used and manipulated to create natural pseudo-random block terrain. Some things Perlin Noise can be used for includes; calculating terrain height, generating clumps of blocks, and creating caves.

The PerlinNoise helper function contains a number of parameters for manipulating the output Perlin Noise value.

```
Perlin Noise Parameters PerlinNoise(int x, int y, float scale, float mag, float exp)
```

- x** The x coordinate of the block
- y** The y coordinate of the block
- scale** Scaling; lower values gives more varying height output
- mag** Magnitude; higher values produce a larger height output
- exp** Exponential; greatly increases height output

Example - Adding clumps of rock

```
SetBlock(x, y, (byte)Layers.Background, (byte)Background.Dirt);
SetBlock(x, y, (byte)Layers.Main, (byte>Main.Dirt);

if (PerlinNoise(x, y, 9, 12, 1) > 6)
{
    SetBlock(x, y, (byte)Layers.Main, (byte>Main.Rock);
}
```


Terrain Engine 2D Terrain Rocks

Height Variables

Height Variables are variables used to set the height of the terrain. They are initialized inside the first loop of the Pass, so that for every x value there is one consistent height value.

```
//Pass
for (int x = 0; x < world.WorldWidth; x++)
{
    //-----Set Height Variables here-----
    int groundLevel = PerlinNoise(x, 0, 75, 20, 1);
    groundLevel += PerlinNoise(x, 0, 45, 30, 1);
    groundLevel += PerlinNoise(x, 0, 8, 8, 1);
    groundLevel += 60;
    //-----
    for (int y = 0; y < world.WorldHeight; y++)
    {
        if (y <= groundLevel)
        {
            //Any blocks added here will be at or below the groundLevel
        }
    }
}
```

In this example 'groundLevel' is the height variable. It has various Perlin Noise values added to it in order to produce the resulting height value for the current x coordinate. Each additional Perlin Noise value adds another layer of noise to the height variable. In this case the first layer of Perlin Noise `PerlinNoise(x, 0, 75, 20, 1);` produces smooth rolling hills. The second layer `PerlinNoise(x, 0, 45, 30, 1);` produces small mountains, dips and valleys. Lastly, the third layer `PerlinNoise(x, 0, 8, 8, 1);` makes the terrain more jagged and uneven. Added together they produce a nice natural rocky terrain (as shown below).

Terrain Engine 2D Height Variable Example

Random

When generating terrain it is often necessary add a level of randomness to your blocks. In Terrain Engine 2D this can be done a number of ways. Unity contains the class 'UnityEngine.Random' which can be used for generating random values. Terrain Engine 2D has two functions which make use of this Random class: SetBlock and DoAddBlock. SetBlock can take an optional percent probability to determine whether the block should be added or not. DoAddBlock takes a percent probability and returns a boolean value representing the evaluation of the odds. Below are some examples making use of these functions.

Example - Randomly adding gems

```
SetBlock(x, y, (byte)Layers.Main, (byte>Main.Rock);
//Add Gems with a probability of 1%
SetBlock(x, y, (byte)Layers.Ore, (byte)Ore.Gems, 1);
```


Terrain Engine 2D Random Gems Example

Example - Random Table Sets

```
if(y < 50)
{
    SetBlock(x, y, (byte)Layers.Main, (byte>Main.Wood);
    //Add Table set with 10% probability if the space is free
    if (y == 50 && DoAddBlock(10) && !IsBlockAt(x, y, (byte)Layers.Decoration, 5, 2))
    {
        SetBlock(x, y, (byte)Layers.Decoration, (byte)Decoration.LeftChair);
        SetBlock(x + 1, y, (byte)Layers.Decoration, (byte)Decoration.Table);
        SetBlock(x + 4, y, (byte)Layers.Decoration, (byte)Decoration.RightChair);
    }
}
```


Terrain Engine 2D Random Table Sets Example

Generating Fluid Data

When manipulating fluid data for your terrain you have three main functions at your disposal: **AddFluid**, **RemoveFluid**, and **GeneratePool**. If you wish to add/remove fluid from a specific block you can use the AddFluid and RemoveFluid functions. With these functions you can set a specific fluid weight, color and density (if using the advanced fluid simulation).

Note that calling AddFluid on a block that contains either a solid block or a fluid of a different density will not work. You will have to remove the fluid in that block first.

Generate Pool is used when you want to generate a pool in an already open area free of blocks, such as in a cave underground. It works by using a flood-fill algorithm to recursively loop through all connected empty blocks below a set maximum height. The function adds the maximum amount of fluid each block can hold while taking pressure into account. Below is an example of using GeneratePool from the SideScrollerGeneratorScript. In this example the function is used to randomly generate pools of one of three different fluid types in caves.

```
//Add water to caves
if (!IsBlockAt(x, y, world.FluidLayer))
{
    byte density = (byte)FluidType.Water;
    if (DoAddBlock(10))
        density = (byte)FluidType.Poison;
    else if (DoAddBlock(15))
        density = (byte)FluidType.Lava;

    //Generate a pool of water with 0.5% probability
    if (DoAddBlock(0.5f))
        GeneratePool(x, y, fluidDynamics.MaxWeight, density, y, new Vector2Int(x, y));
}
```


Terrain Engine 2D Fluid Example

[DONATE](#)