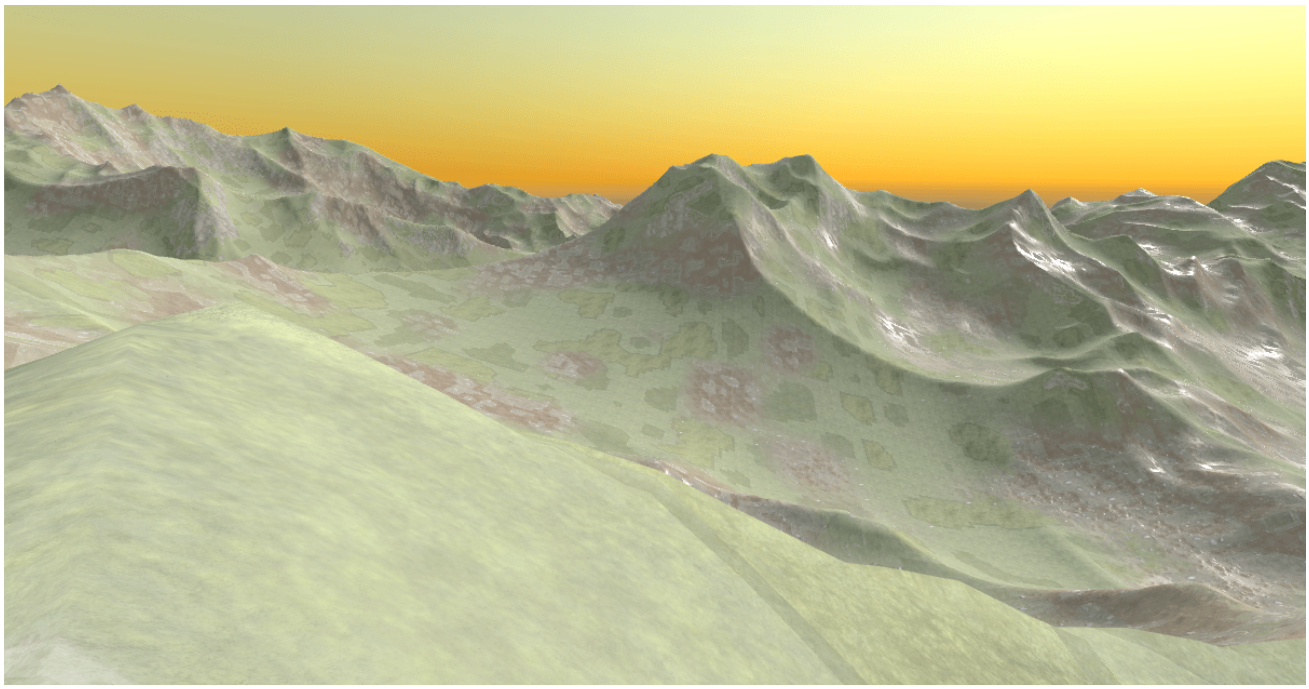
# Terra Documentation

Walker Christie

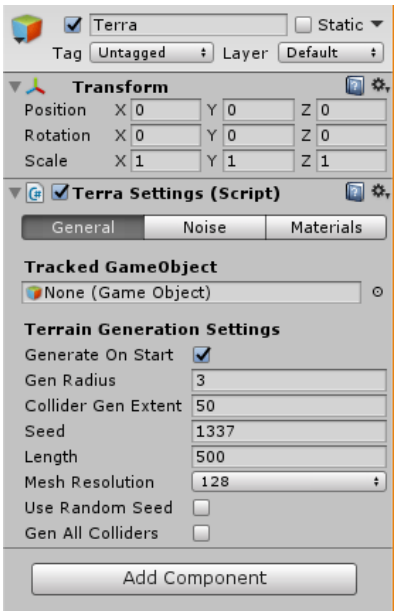


Thank you for purchasing Terra! Terra is a procedural terrain generator that uses a node-based system for an infinite amount of generation options. The following documentation explains how to use Terra and is broken up into the following sections:

## Contents

# 1 Getting Started

Getting up and running with Terra is really quite simple. First, create an empty gameobject; then click *Add Component* and add the *Terra Settings* script. Your inspector should look something like the image to the right. Below is a list of parameters and what they do:




- **Tracked GameObject:** allows you to specify a gameobject in the scene that the terrain will generate around. Because the terrain is infinitely generated, attaching a gameobject like the *FirstPersonController* will cause the terrain to generate around the moving player. If no gameobject is specified, the terrain will generate at <0,0,0> in world space.
- **Generate On Start:** toggles whether or not the terrain will start generating when play mode is entered. Disabling this allows for terrain that is generated programmatically rather than automatically. When enabled, a node graph is not needed for generation, instead, a generator can be specified programmatically. Programmatic generation is discussed in detail in section 5.
- **Gen Radius:** stands for generation radius and is the radius of tiles that will generate around the tracked gameobject. A live preview of this can be viewed in the scene when your Terra gameobject is selected. Keeping this number between 1-4 is suggested as generating new tiles is costly (especially on lower-end computers).
- **Collider Gen Extent:** The collider generation extent is a bit more complicated to explain, but important to the speed of the terrain generator. Terra, in order to speed up the time it takes to generate a terrain tile, allows you to (optionally) only generate mesh colliders around the tracked gameobject. In other words, the tile you're standing on has a collider, but the one you see in the distance doesn't. This parameter specifies how close the tracked gameobject has to be before the collider for a nearby tile is calculated. A live preview of this can be viewed in the scene, and is the blue square that moves with the tracked gameobject.
- **Seed:** is the seed used by the internal pseudo-random number generator. You can read up on random seeds on Wikipedia. This gives you the option to generate the same terrain across different computers using the same seed.
- **Length:** is the length of one side of a tile in Unity3D units (meters by default).
- **Mesh Resolution:** like the name, is the resolution of the generated mesh. The higher the resolution, the more vertices are sampled leading to a possibly longer generation time. The resolution can be explained as: how many squares will fit within one column of the tile? 128, for example, fits 128 squares in a single column and 16384 squares total since $128^2 = 16384$.
- **Random Seed:** Sets the internal pseudo-random number generator to a random seed. When this is enabled terrain will look different every time play mode is entered. Because of this, the preview in the noise tab is not entirely accurate.
- **Gen All Colliders:** coincides with the *Collider Gen Extent* setting. When generate all colliders is enabled, collider generation is no longer deferred. Instead, a tile's collider is generated after the mesh has been generated.

As a final note, I want to point out that changing the Length and Mesh Resolution parameters together can drastically change the way Terra works. A small length with a high resolution can lead to very fine terrain that looks great and runs quickly. You can also increase the Gen Radius to have many smaller meshes rather than fewer bigger meshes.

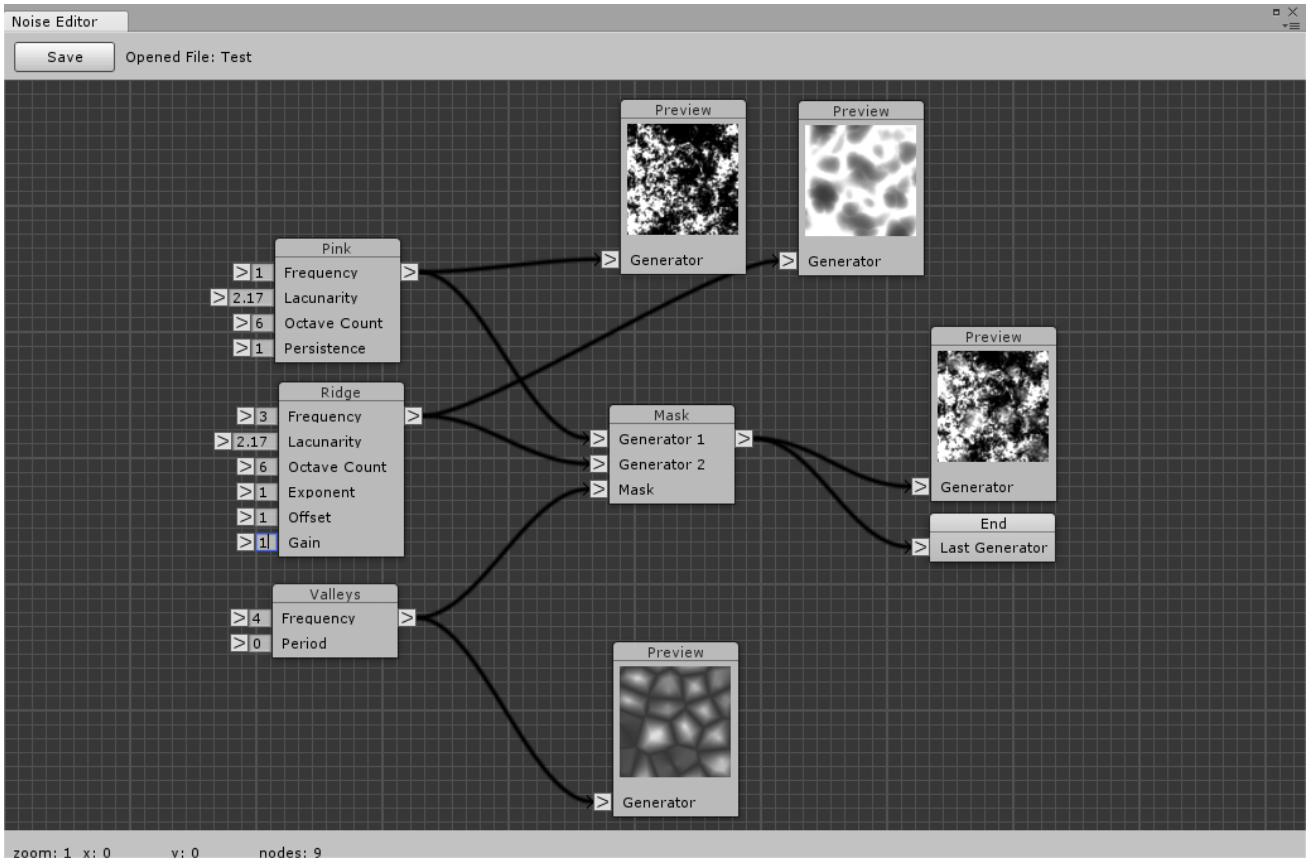



Figure 1: An example Noise Editor configuration

# 2 Noise Tab

The noise tab is responsible for handling user-created "graphs" that your terrain generates around. If you're not familiar with noise-based terrain generation, take a look at this website. To create a graph, select **Create New Graph** and choose a location to save the associated JSON file. This file contains all information about your node graph and can be saved, loaded, modified via text editor, and even shared online for others to use. Once you've created your graph, the noise editor will appear.

The Noise Editor, as shown in Figure 1, works by connecting a series of nodes together via input/output sockets. New nodes can be added by right clicking the editor window and choosing either a noise, modifier, or preview node. Every type of node is covered in the subsections below.

## 2.1 Noise Nodes

- **End:** The end node is what makes the terrain generator come to life. After you've constructed all of your nodes, connect your last node to the end node. This node tells Terra where to look when applying noise values to the mesh. The end node is required for generation to work correctly. Failure to include it will result in a warning from the noise tab.
- **Billow:** Billow noise is a derivation of Perlin noise. A unique aspect of billow noise is the cloud-like billows it produces. This type of noise can be helpful in adding extra bumpiness to the surface or your terrain.
- **Pink:** Pink noise is a fractal-based noise function. It produces a series of high and bumpy areas mixed with lower flat areas. Pink noise is perfect as a base for your terrain.
- **Ridge:** Ridge noise works by creating cells of tall, repeating "ridges" which look similar to mountains with peaks. A preview of ridge noise can be seen in figure 1. By default, ridge noise has a *gain* setting of 2, making figure 1's example a little different from what may be expected.

- **Voronoi Pits:** Voronoi pits come from the Voronoi Diagrams algorithm which divides a plane (in this case an infinite one) into similarly-sized cells. Though it's not shown in the figure 1, Voronoi pits look very similar to valleys with the exception that values are lower towards the center of the cell rather than higher.
- **Voronoi Valleys:** Voronoi valleys are very similar to pits except that values raise towards the center of a cell rather than lower. A good use for Voronoi pits and valleys is masking, which is explained in the next subsection.

## 2.2 Modifier Nodes

Because noise functions are just that— functions, mathematical operations can be applied to them. Modifier nodes do not take in generators and output a scalar value. Instead, a new generator is constructed and can be used in different nodes. For example: adding the generators $f(x)$ and $f(y)$ will produce the generator $f(z) = f(x) + f(y)$

- **Add:** Adds two generators together and outputs a summed generator.
- **Divide:** Divides the first input generator by the second generator and outputs a quotient generator.
- **Mask:** Blends the first two generators together with the third generator acting as the weight. If the weight generator outputs 1 as a value, the first generator will entirely appear. Conversely, if the weight generator outputs 0 as a value, the second generator will appear instead. If the weight is 0.5, both the first and second generator values will be averaged together 50/50. This effect can be seen in Figure 1 where the Voronoi node acts as the weight.
- **Max:** Takes the higher of the two supplied generators.
- **Min:** Takes the lower of the two supplied generators.
- **Multiply:** Multiplies the two generators together and outputs a product generator.
- **Subtract:** Subtracts the first generator from the second one and outputs a difference generator.

## 2.3 Preview Node

The preview node is a special type of node that acts differently from the those previously talked about. Instead of outputting a generator, the preview node displays a preview of whatever node it takes in as an input. This is helpful in checking out how your noise looks at different points in your graph. In Figure 1, 4 different preview nodes are used at different stages in the graph. Here are some things to keep in mind:

- Although adding many preview nodes may slow down how fast an image is updated in the graph, it does not make a difference at runtime. At runtime the preview nodes are not processed.
- If an error has occurred while processing nodes, a preview will not appear.

## 2.4 Spread & Amplitude

Unlike the previous subsections, the spread and amplitude parameters are not set in your graph. At runtime spread and amplitude are applied *after* a point has been calculated via terrain graph. If you find your terrain looking too spiky or too flat, try playing with these values.

- **Spread:** Either squashes or stretches your terrain depending on the input value. You can visualize this by imagining grabbing the corner of a generated tile and pulling it outwards or pushing it inwards. Values ranging from $0 - 1$ squash the tile inwards while values greater than 1 stretch the tile outwards. Internally, the *End Node* generator's x and z positions are multiplied by the spread value.
- **Amplitude:** Raises or lowers your terrain depending on the input value. Like the spread parameter, values ranging from $0 - 1$ lower the terrain while values greater than one raise it. Internally, the *End Node* generator's y position is multiplied by the amplitude value.

# 3 Materials Tab

Generated terrain would look pretty boring if there were no way to apply textures to it. Thankfully, Terra provides a way to apply splatted textures to your terrain using the materials tab. The way procedural texturing works in Terra is by having different textures appear at different heights. To get started, click "Add Material". Below is a description of each parameter.

- **Edit Material:** Opens up a material editor where a diffuse and normal texture can be set along with the tiling, offset, and how metallic or smooth a material is. This works similarly to the terrain editor's texture window.
- **Blend Amount:** Specifies how far this texture should blend with the texture below it and above it in Unity units.
- **Placement Type:**
  - **Elevation Range:** Places textures by their specified height range [min - max]
  - **Angle:** Places textures according to their specified angle range [0 - 90]
- **Max Height:** Sets what height this texture should stop displaying.
- **Min Height:** Sets what height this texture should start displaying.
- **Is Highest Material:** If enabled, this material becomes the highest one. This is equivalent to setting the *Max Height* to positive infinity.
- **Is Lowest Material:** If enabled, this material becomes the lowest one. This is equivalent to setting the *Min Height* to negative infinity.
- **Angle Min/Max Range:** When an *Angle* Placement Type is chosen, this setting appears. The range slider can be moved to set a minimum angle that a material will appear and the maximum angle at which the material will appear.

Each elevation material should have a max height that is greater than the minimum height. Also, each material should "line up" with the next one. For example, if you had a grass material that had a min height of negative infinity, and a max height of 100, the next material (ie dirt) should have a min height of 100.

Optionally, you may want to use a **custom material** instead, such as a MegaSplat material. In this case, check the "Custom Material" box at the top and choose which material you'd like to use. This material will be applied to each generated tile at runtime. The terrain pictured at the beginning of this document was created using a custom MegaSplat material.

# 4 Under the Hood

The following section explains what Terra is actually doing at runtime. Terra starts computations by creating a tile pool. This *TilePool* instance calculates where to place new tiles, whether or not to update old ones, and which colliders need to be generated.

Internally, TilePool contains a *TileCache* instance which maintains a list of tiles that have already been computed so that they don't need to be recalculated if requested. This means that if your player is walking between a few of the same tiles over and over again, less lag will occur. The *TerrainTile* class actually derives from *MonoBehavior* and is added as a component to each generated tile. This means properties from terrain tiles can be accessed easily at runtime. After TerrainTile calls the noise function for each vertex, the terrain is ready to be textured.

*TerrainPaint* is responsible for splatting the terrain and calculating what texture to place where based on the height of each vertex. Textures are applied using the built in terrain shaders that Unity provides. Individual textures and their associated splatmaps are assigned to the FirstPass and AddPass shaders, then applied to the MeshRenderer for each tile. Finally, many of these functionalities can be attached to delegate functions in your own code for customization.

This is where *TerraEvent* comes in. TerraEvent is a class that contains static events which are fired at different points in Terra's execution. These events can be received in your own code to perform custom operations. The following events exist in Terra.

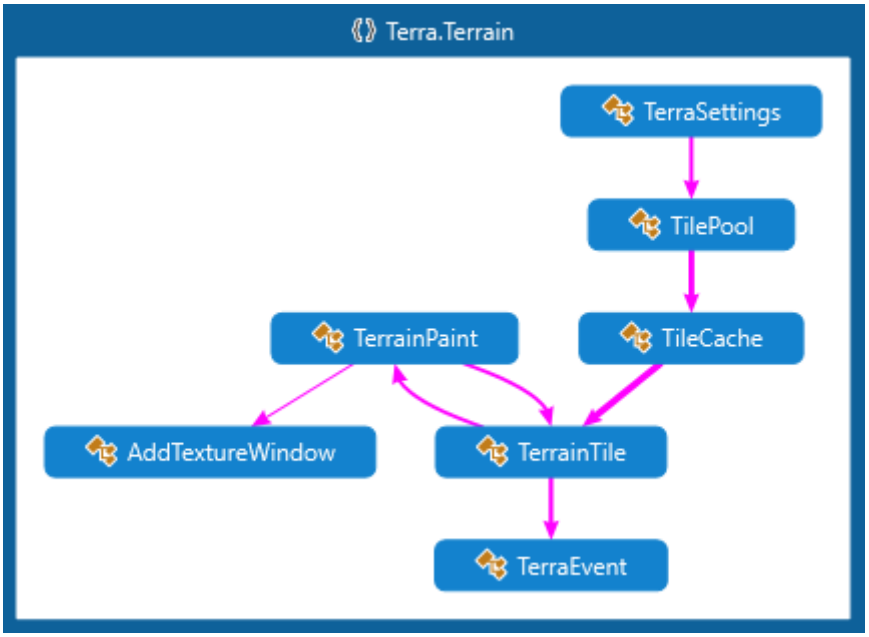



Figure 2: Code diagram for the Terra.Terrain namespace

- **OnMeshWillForm** and **OnMeshDidForm**, like their names, are fired when the mesh is about to be created or after it has been created. This is not the same as applying noise; rather, a flat Mesh is created before calculating and applying noise values.
- **OnSplatmapWillCalculate** and **OnSplatmapDidCalculate** are fired when the splatmap (which you can read more about here) is about to be or has been calculated.
- **OnCustomMaterialWillApply** and **OnCustomMaterialDidApply** are fired when the user-supplied custom material is about to be or has been applied to the MeshRenderer for a particular tile.

Here is an example of registering an OnMeshDidForm event handler.

```
public void RegisterEventHandler() {
        TerraEvent.OnMeshDidForm += HandleOnMeshDidForm;
}

public void HandleOnMeshDidForm(GameObject go, Mesh mesh) {
        Debug.Log("The Mesh attached to " + go.name + " has been created.");
}
```

# 5 Programmatic Generation

One of Terra's best features is its ability to generate terrain both visually and programmatically. If you're looking to control the terrain generation process as much as possible or modify terrain the way you see fit, try creating your terrain programmatically.

In this section, I'll walk through the steps needed to generate the terrain tiles shown in Figure 3. For convenience, a TerraSettings component is still required for generation. Just as was done in the beginning of this document, the TerraSettings component can be added to any gameobject you see fit. Throughout the generation process, Terra will reference the TerraSettings component to get information like a tile's length, resolution, amplitude, etc.

To get started, make sure the setting "Generate on Start" is disabled. This tells Terra to avoid generating terrain when play mode is entered. We do this because we want to tell Terra when, where, and how to generate our tiles. Next, create or enter a script where you want the generation to happen. I named my script TerrainTest.

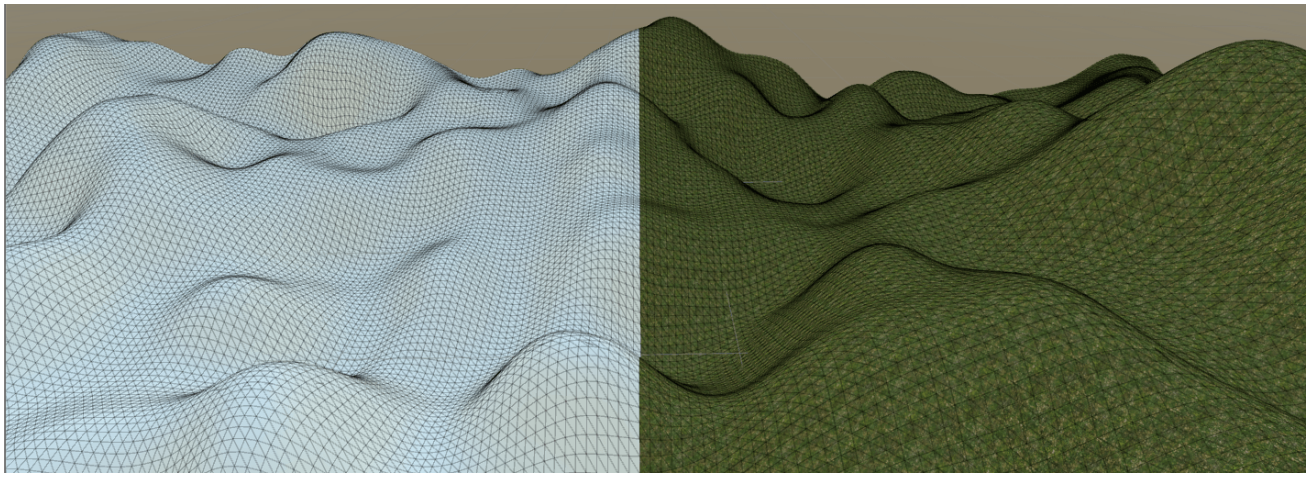


Figure 3: An example of programmatically generated terrain

```
using Terra.CoherentNoise;
using Terra.CoherentNoise.Generation.Combination;
using Terra.CoherentNoise.Generation.Fractal;
using Terra.Terrain;
using UnityEngine;

public class TerrainTest : MonoBehaviour {
        void Start () {
                //Change settings at runtime
                TerraSettings settings = FindObjectOfType<TerraSettings>();
                settings.Amplitude = 50f;
                settings.Spread = 50f;

                //Create gameobjects with attached TerrainTiles
                TerrainTile tile1 = TerrainTile.CreateTileGameobject("Custom Tile 1");
                TerrainTile tile2 = TerrainTile.CreateTileGameobject("Custom Tile 2");

                //Programmatically create generators
                PinkNoise pink = new PinkNoise(123);
                pink.Frequency = 2f;
                pink.Persistence = 0.9f;

                RidgeNoise ridge = new RidgeNoise(456);
                ridge.Frequency = 0.8f;

                Generator added = new Add(pink, ridge);

                //Create mesh and apply noise values
                tile1.CreateMesh(new Vector2(0, 0), added);
                tile2.CreateMesh(new Vector2(1, 0), added);

                //Create colliders
                tile1.GenerateCollider();
                tile2.GenerateCollider();

                //Paint one of the tiles using the settings from TerraSettings
                TerrainPaint paint = new TerrainPaint(tile1.gameObject, settings.SplatSettings);
                paint.GenerateSplatmaps();
        }
```

```
}
```

In this example, generation begins by referencing the TerraSettings component and changing some settings. Though they're not listed here, virtually every other setting can also be changed (i.e. length, resolution, splat settings, etc.) Next, two TerrainTiles are created using the convenience method *CreateTileGameObject* which creates an empty gameobject, attaches a TerrainTile component, and returns the component. After this, I create my custom generator configuration.

*PinkNoise* and *RidgeNoise* are two different types of fractal noise. These types are available both as node types in a node graph and as their generators (as seen above). Just as you would in a node graph, generators can be added together using the *Add* class which takes two generators as input. Terra uses the open-source library CoherentNoise for its noise algorithms. The creator of CoherentNoise, Unity forums user Nevermind, has put together an excellent documentation on the library which you can view here if you want more information regarding different generators.

After setting up the generators, meshes are created using the *CreateMesh* method which takes a Vector2 and Generator as input. The Vector2 parameter specifies where on the grid of tiles to place the gameobject, while the passed Generator is the CoherentNoise generator to use. The CreateMesh method does not, however, automatically generate a collider. Because generating a collider is costly, the functionality has been moved to the *GenerateCollider* method (which is called afterwards).

Finally, *TerrainPaint* is used on one of the tiles to texture it. The TerrainPaint constructor takes the gameobject of the tile to paint and a list of *SplatSetting*. A SplatSetting can be instantiated to change the way different textures display on the terrain. In the example code, SplatSetting is simply referenced from TerraSettings, which pulls the results from the Materials tab in the inspector. A SplatSetting can be created programmatically if needed and passed into TerrainPaint's constructor. After TerrainPaint has been created, *GenerateSplatmaps* is called to apply the passed splat settings to the terrain.

# 6 Support

As the creator of this asset, I want to provide as much support to the purchaser as I can. If any bugs are found or there is confusion regarding how to use Terra, please feel free to send me an email and I will get back to you as soon as possible. Please use the following email: **support@walkerc.me**. In your email include any necessary information such as logged debug messages, screenshots, and the version of Unity that you're using.