Terrain Tessellation Tools

Table of contents:

1: Introduction

This Toolkit provides tools to easily create and organize an efficient terrain for your game. Its main purpose is to create, and slice simple meshes from heightmaps, to use for CPU-side purposes. (Like for example, collisions). And also use them as base for tessellation, which is the final result displayed on screen.

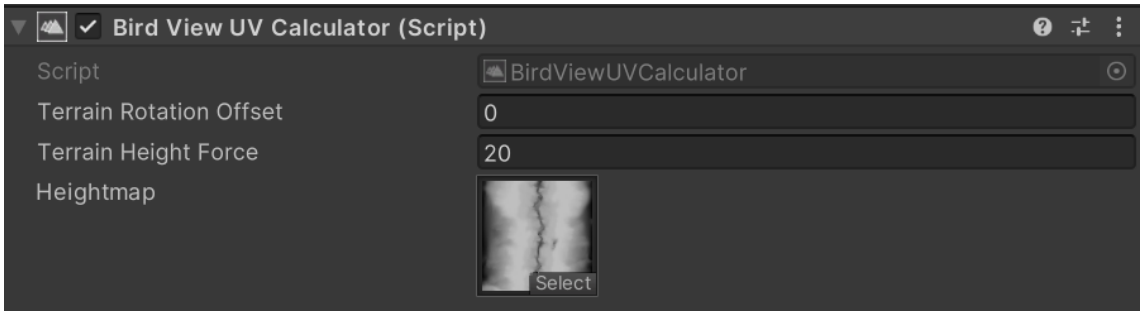
This workflow has many upsides, and this toolkit aims to provide an easy way to achieve it.

Between the many upsides, some of them are:

- No need to create complicated meshes in external programs. Making great terrains is hard in 3d modelling.
- Heightmaps are easier, and faster to make.
- Heightmaps can more accurately simulate real-world effects, like erosion, deposit or river flow.
- By tessellating in real time, with LOD, not only you save up in file size, but you also improve performance. (A LOD Tessellated Mesh has less vertices than the equivalent premade mesh, but looks exactly the same; since distant vertices don't need tessellation to appear the same on the screen)
- This workflow uses meshes instead of unity's terrain; Larger, open-world terrains are usually made with meshes for better performance.
- This workflow also tiles meshes, encapsulating your scenery in tiles. If you dislike how one tile looks, you can just edit that tile texture. Instead of remaking / editing the mesh in an external program.

2: Terrain Tessellation Tools Content

BirdViewUVCalculator



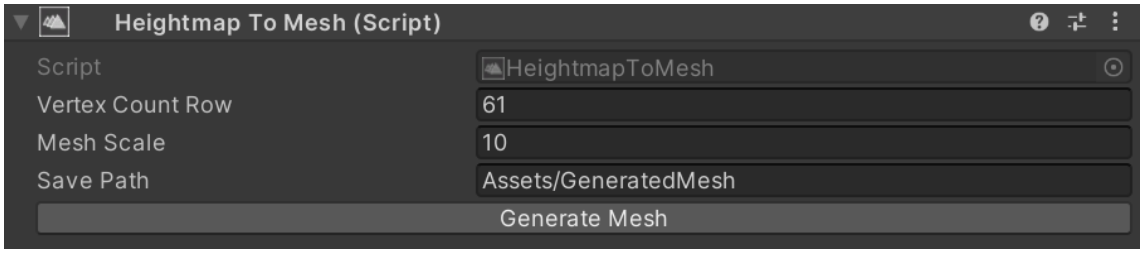
This script is used in all the tools. It reads the heightmap texture and mesh filter to give the proper height of the point.

-Rotation Offset: If the Heightmap orientation is not correct, you can offset it certain degrees. Works best at 90° intervals.

-Height Force: Heightmaps are values from 0 to 1. Since our terrain will probably will be higher than a single unit, we multiply the height with the force. With force 20, black is 0 Y, and white is 20 Y.

-Heightmap: The texture to read. It should be a square grayscale texture.

Mesh Generator (HeightmapToMesh)



This script creates a mesh asset with the help of Bird View UV Calculator. The resulting mesh is square.

-Vertex Count Row: How many vertices in a row to create the mesh with. Should line up with Tiling grid. If for example the grid is 10 tiles for this terrain, it should be 10*x+1. (Optional)

-Mesh Scale: The resulting mesh square side size. Height is defined by BirdViewUVCalculator height force.

-Save Path: Where will be the mesh saved, and the name to give it.

Terrain Grid Tiler

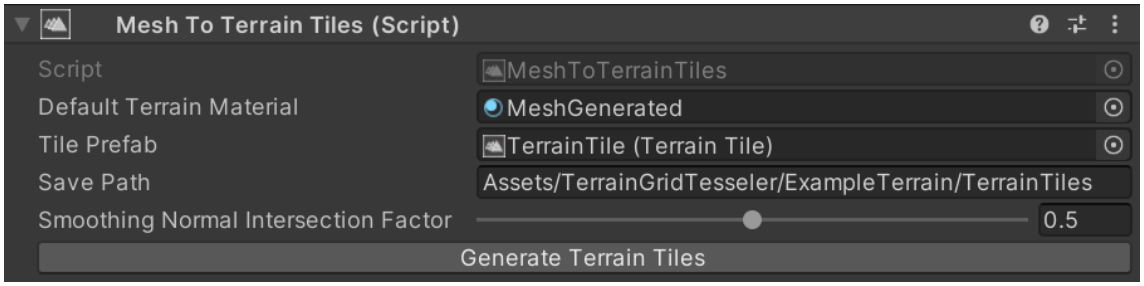
This is the grid of the Tiler. This script tells the slicer how to slice the mesh.

-Grid Size: Size in global units of the parent tile. It should be square for best results and organization.

-Tile Size: The size of a single tile, all the tiles in your project should have the same size ideally. This declares how many child tiles will be created. (You can perform operations in float fields. For example: type 10/8 and you get 1.25, for eight tiles) If the size isn't exact, it will ignore the exterior of the parent tile.

-Center Offset: You will most probably wont use this, it offsets the center of the grid a Vector2. In case some mesh you want to slice doesn't have a proper center, and you want to ignore a side of it.

Mesh Slicer Tiler(MeshToTerrainTiles)



This script slices a mesh, while also slicing the heightmap. Useful for big worlds. It helps if the mesh given has no triangles sliced up in half. This script will also slice them, and average the normals, but you will end up with more geometry in the borders. Refer to Vertex Count Row on the mesh generator.

-Default Terrain Material: The material to apply to all the tiles mesh renderer. You can modify it later for a single tile.

-Tile Prefab: The prefab to instantiate for each tile. It should be a Tile class prefab. This Class is empty, for your own behaviors. Each sliced tile has a BirdViewUVCalculator, so you can tesselate them later.

-Save Path: The parent folder where the terrain folder of this mesh will be created. This folder has: The sliced mesh total (the same as your mesh, except when you slice triangles in half.), the tile meshes, and the tile sliced heightmaps.

-Smoothing Normal Intersection Factor: How the normals are smoothed when inexact triangles are sliced. Works best at 0.5 (half-half) but might look better in some cases at other values (You should try to generate exact meshes anyways)

Tessellation Shaderlab library

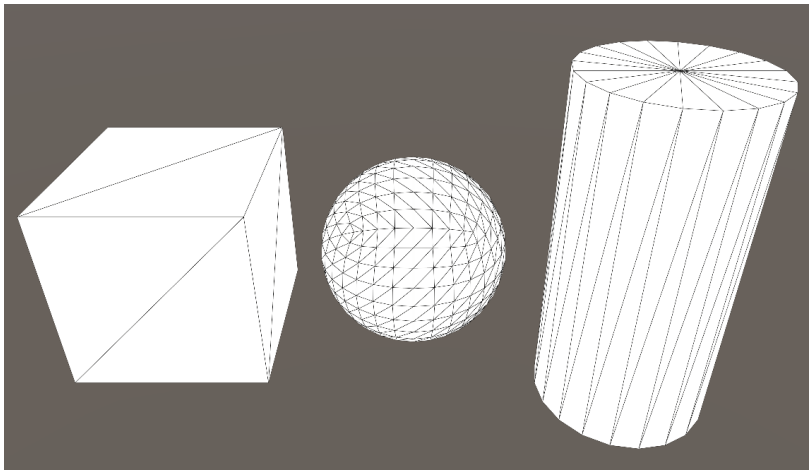
This toolkit also comes with a some Shaderlab libraries for tessellation. It uses both the hull and domain stages of the shader pipeline. These libraries are in both .cginc and. shader files. Since the shader pipeline uses the output from the stage from before, they cannot be called with just a function. And you must edit a bit of code to use them, refer to: Creating your own Tessellation shader

Every Library has a .shader, a .cginc; and in Tessellation, a .cginc input file. The input defines structures/first input, the regular .cginc has the actual stages code, and the .shader file serves as the material. Every Shader except wireframe also has culling (and tolerance value)

The library has code for:

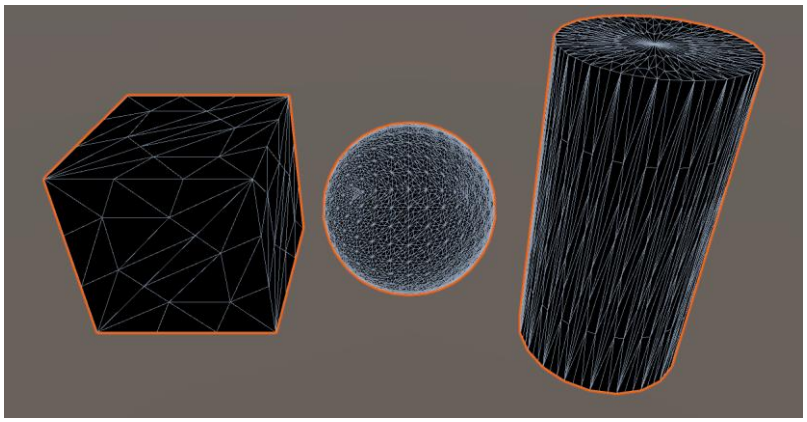
- Wireframe
- Tessellation Simple
- Tessellation
- Tessellation Smoothing
- Tessellation Vertex Displacement

Wireframe



Tessellation changes geometry at runtime. Before Unity would only display the initial mesh in the scene. Luckily, now it also displays the mesh tessellated by a shader. Nonetheless, its nice to check the mesh geometry in play mode, for better debugging. This library calculates the edges, and displays the result in another color than the rest of the mesh.

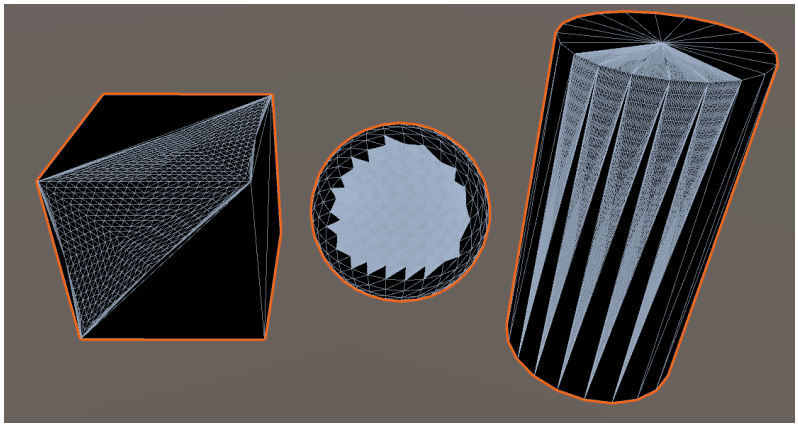
Tessellation Simple



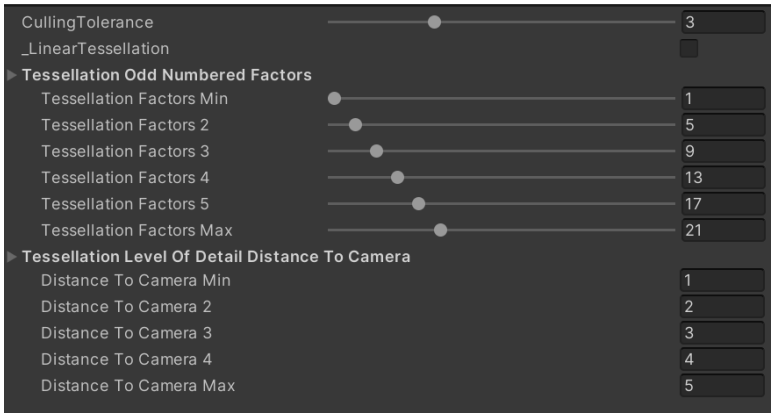
Tessellation is used to add more geometry to a mesh, and it can be used for various things. In case you want to add you own behaviors, there is also this simple version, in where you decide simply how much to tessellate the entire mesh. Without moving the mesh, it should look almost the same as before, so it's advised to move the vertex yourself for your behavior. Note that the Tessellation is odd-number-based, but can be easily changed.

`[UNITY_partitioning("fractional_odd")]` : Inside "Tessellation. Cginc"

Tessellation



This is just like Tessellation Simple, but with a Distance to camera algorithm to change the tessellation factor. This is useful for LOD Behaviors, and shouldn't be too hard to change it into distance to another object. This is still for you to make your own behaviors, as the vertex still don't move.



This shader has two modes. Depending on _LinearTesselation.

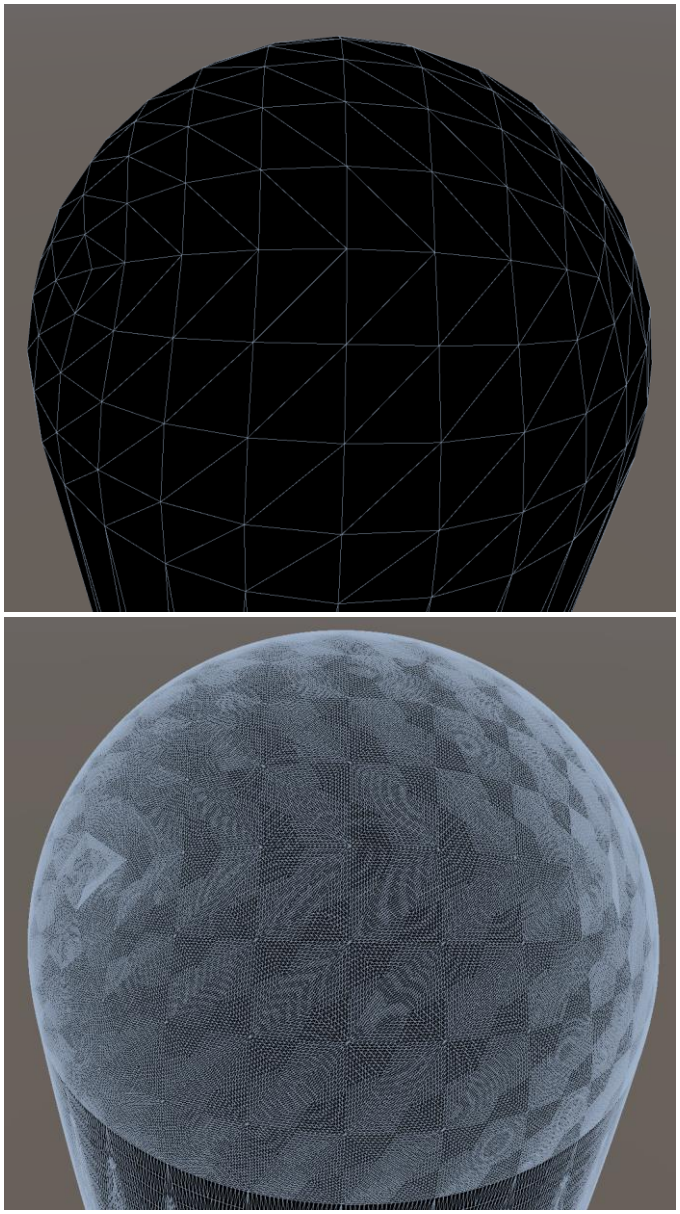
If its enabled, only the first and last value matter of both categories. Otherwise, it functions like a Floor function, and takes the greatest factor it can, based on distance.

For example: Tessellation Factors Min is always active, next level is Tessellation Factors 2, then 3. Etc.... until Max.

If you are at Max distance or more: you get the min Factor. Between 4 and max you get the factor 2. Etc.... if you are at min distance or less, you get the maximum factor.

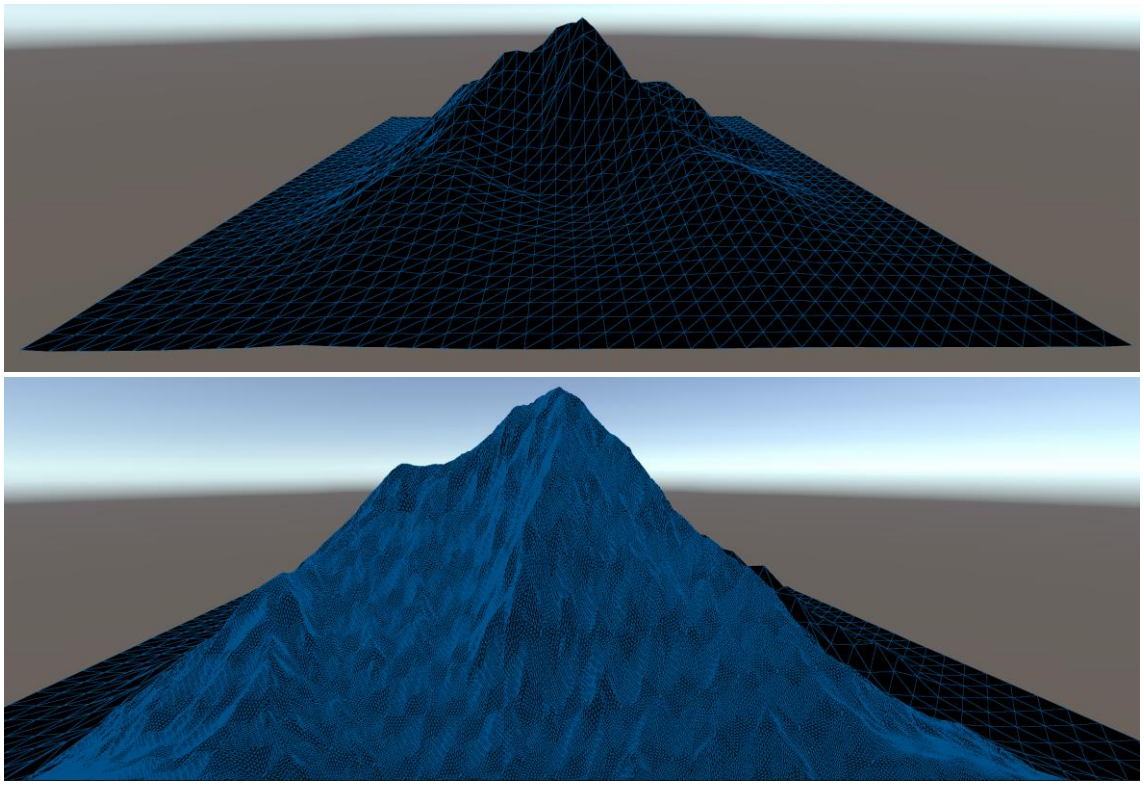
Turning _LinearTesselation makes it a linear function based on distance, which gives a smoother transition.

Tessellation Smoothing



The first of the two implemented behaviors, and the simplest one. This behavior, tessellates depending on distance, and moves the vertex with the Pong algorithm, to smooth out surfaces. This may make some objects smoother when you are near them, to make them more detailed or less sharp.

Tessellation Vertex Displacement



The second of the two implemented behaviors, and the most useful one. With the help of BirdViewUVCalculator, this shader displaces the vertex from tessellation matching the Heightmap texture, making so much easier having terrain with incredible detail.

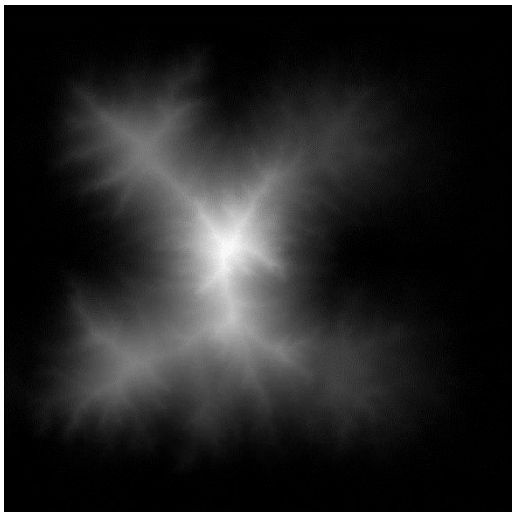
(This one is the example version, with wireframe implemented too. There is also a version with no wireframe and color.)

3: Getting started

What you will need

To create terrain for your game, you will need Heightmaps for it.

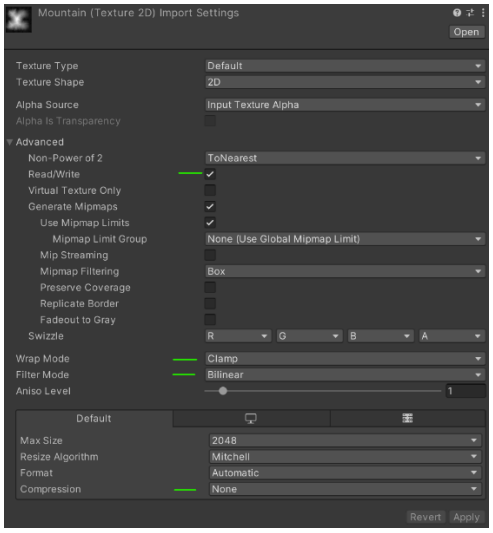
Heightmaps are square textures in grayscale, that are designed to depict in a value from black to white, the height of a certain point in the texture. Black is low, and white is high.



This makes it so that you can get much more definition with a big Texture, than a mesh with a lot of vertices.

For that, you need to generate all the heightmaps beforehand, making sure all the textures align properly. Refer to How to Create Heightmaps and Properly Slicing Textures Externally.

You also need to modify your texture's configuration. You need to enable Read/Write. Filter Mode should be Clamp and Bilinear. Compression can be done, but None makes terrain look best. (At the cost of space)

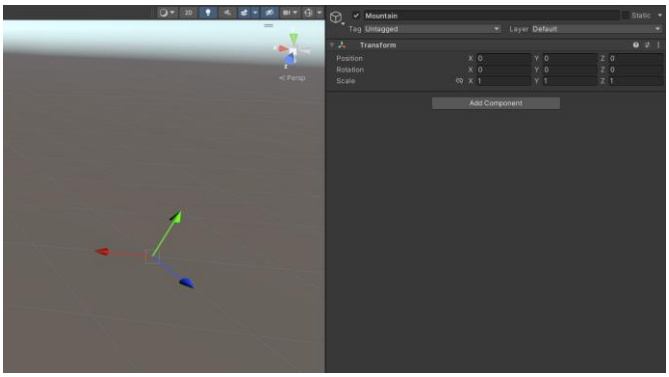


You will also probably need to create your own Tessellation Shader with the library that fits your terrain needs. Refer to Creating your own Tessellation shader.

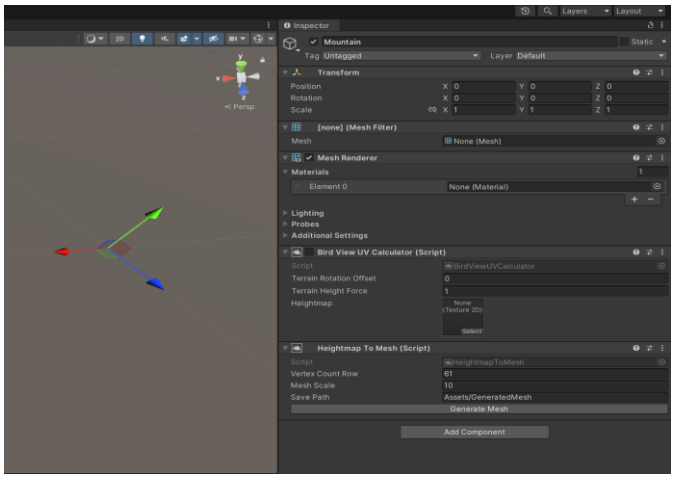
Quick start guide

For a quick example, you only need one heightmap. You can use one of the demos, or use another. The only requisite is a square and grayscale Texture, with read/write enabled.

First you need to decide the place where your terrain would be, and its scale.



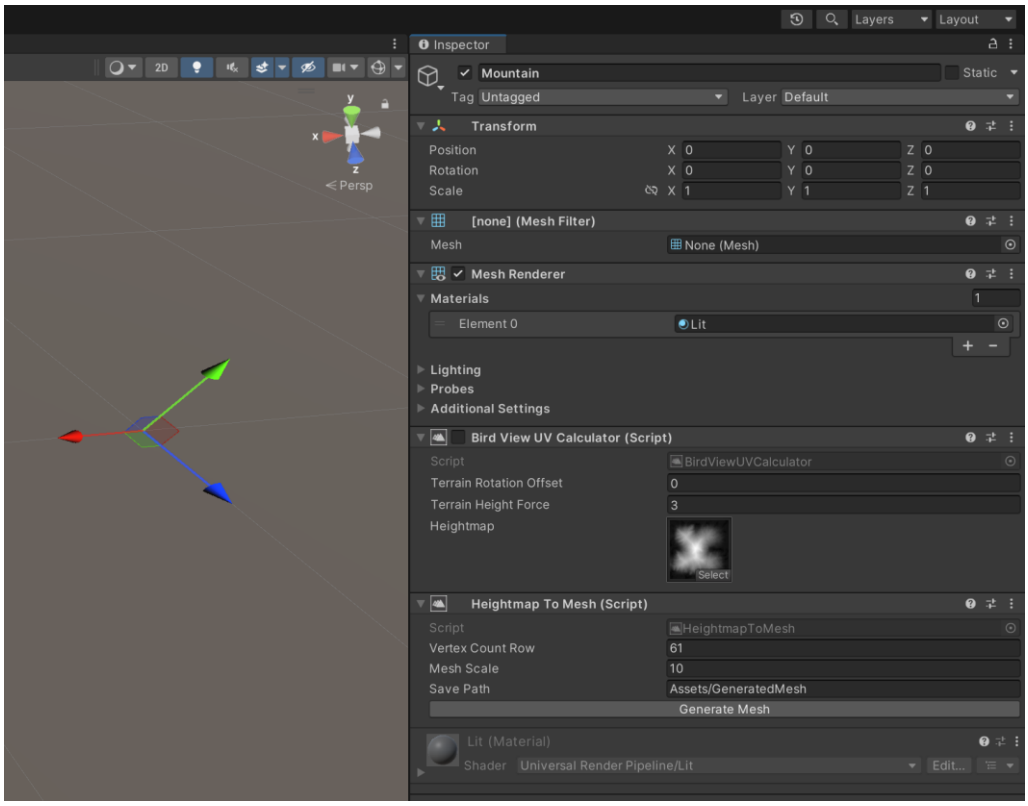
Then, you can attach HeightmapToMesh script...



Fill BirdViewUVCalculator with the data, and decide how many vertices per row your mesh would have, and if possible, try to match the grid tile edges to the mesh edges, so no triangle gets sliced in half. (resulting in useless geometry)

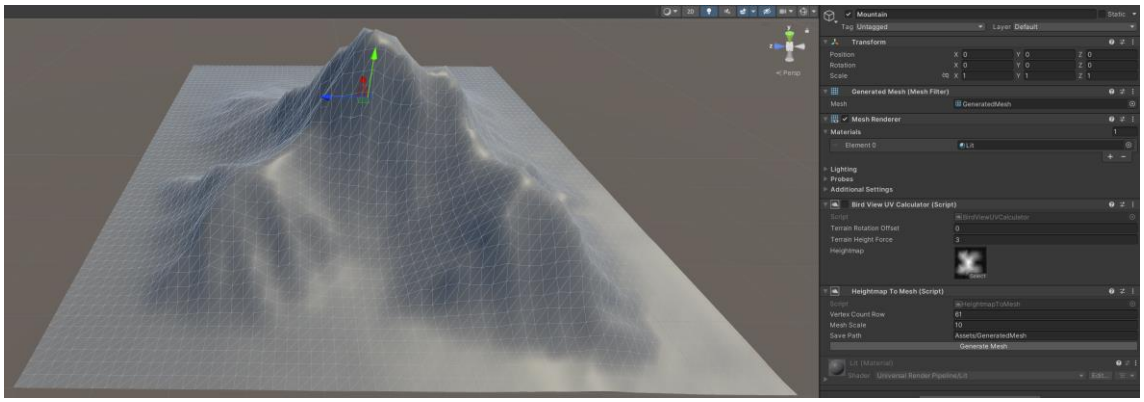
If you are not going to use a Tessellation Shader, you can use a lot of vertices per row to have more detail. And the bigger the scale, the more vertices you will need.

If you are going to use one, the resulting mesh will be used for CPU-side, like mesh colliders. You might not need a lot of vertices for that.



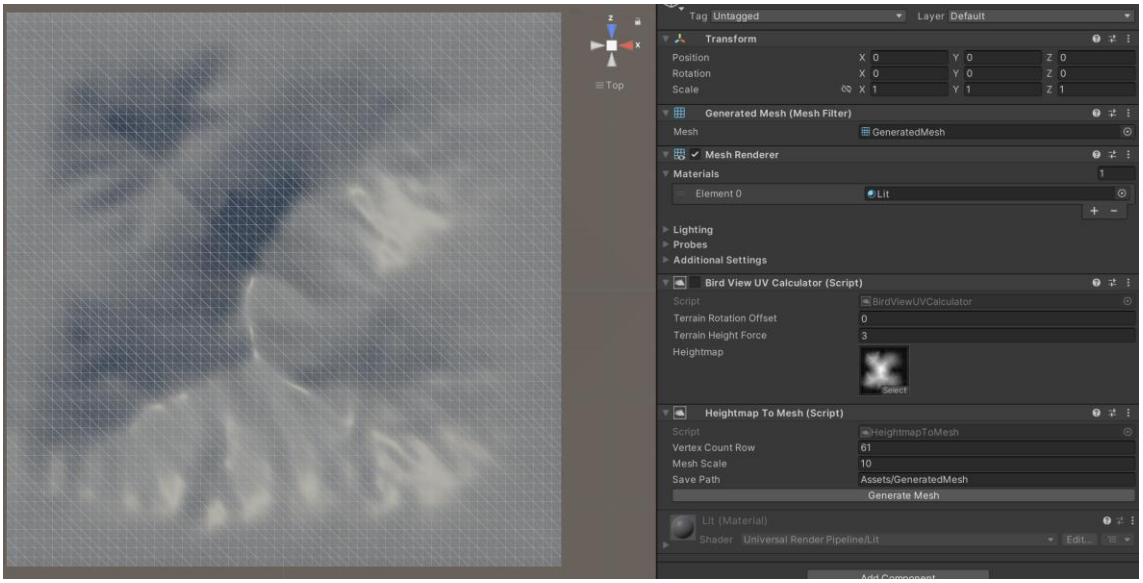
Add the Heightmap, change height force maximum height, and declare the asset path/name, if you are going to slice the mesh later, you may remove the mesh afterwards.

After all that, press “Generate Mesh” to generate your mesh with your configuration…



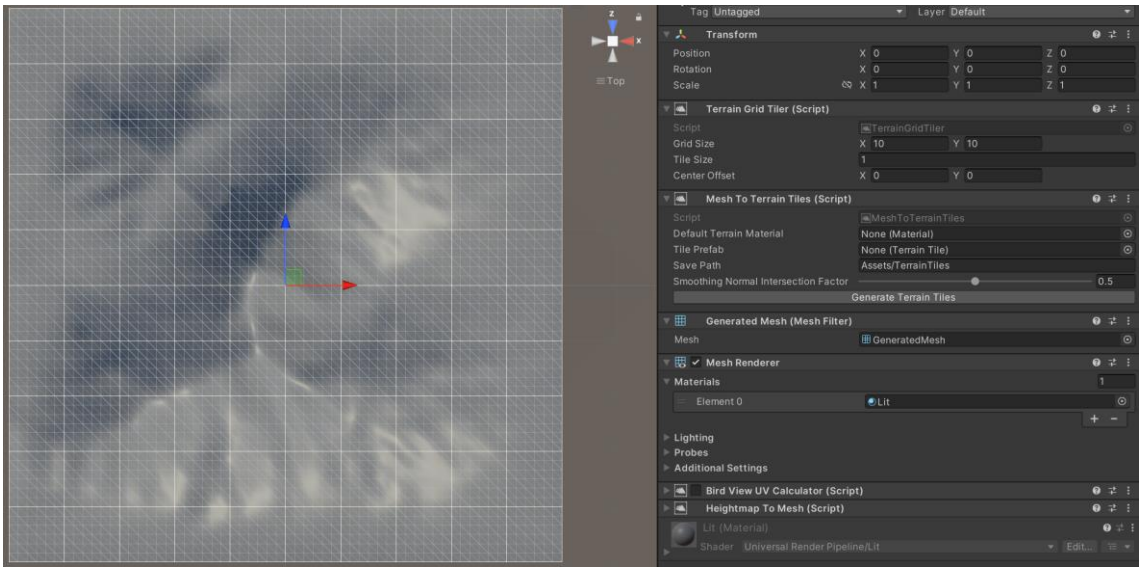
And you already have a working mesh.

As you can see, generated meshes are done with two triangles making a square, in XZ plane.



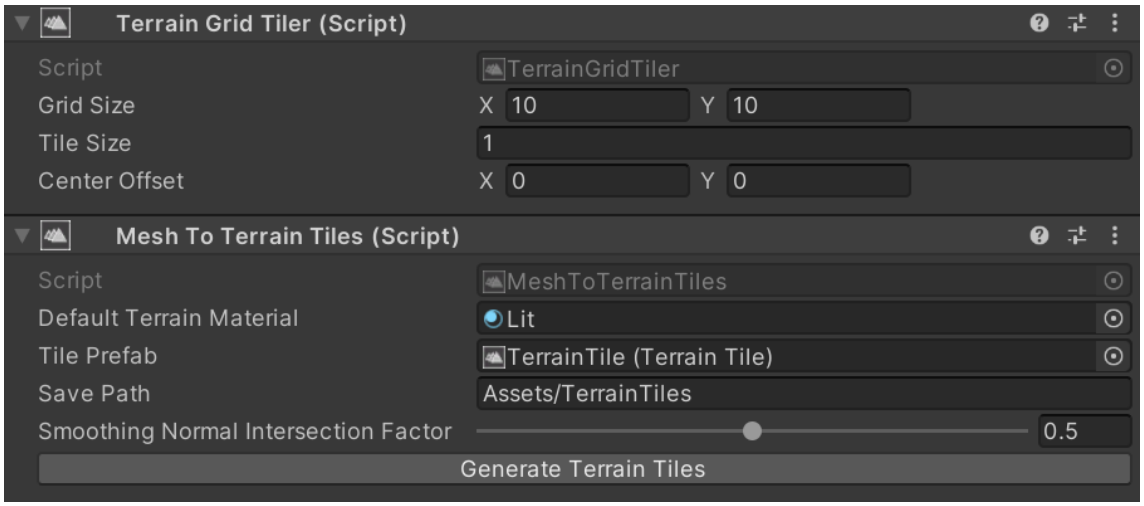
(View from above)

This is important when slicing the mesh for better organization.



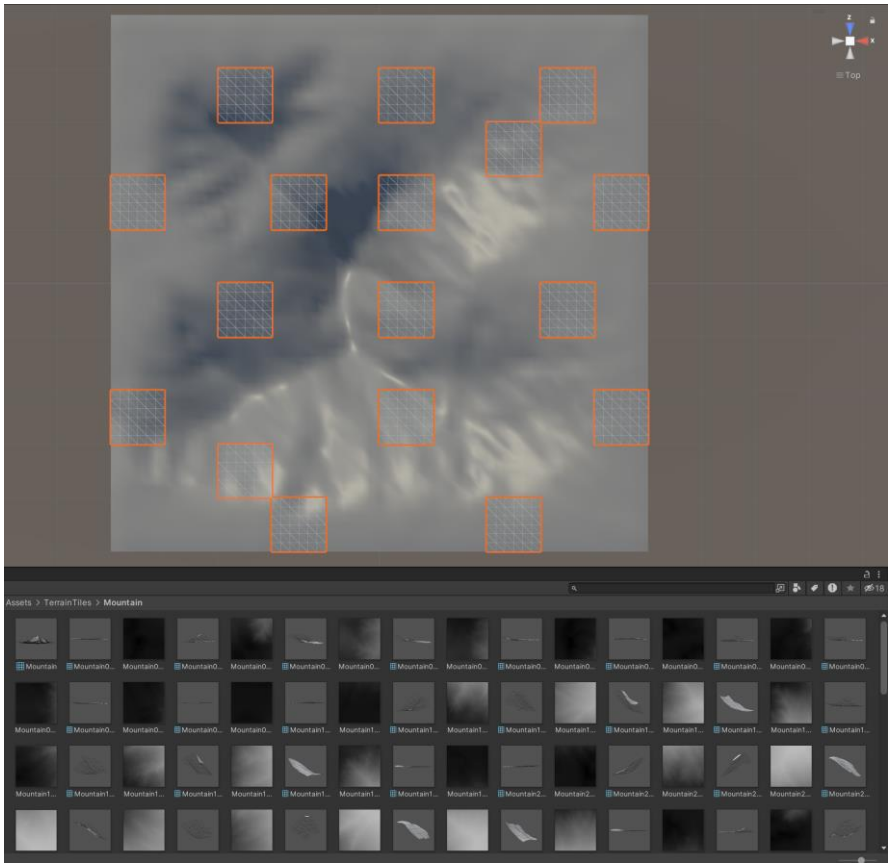
When adding the MeshToTerrainTiles script, you will get the grid for tiling. It is very recommended for the Grid to coincide with the mesh squares. That way you won't create extra geometry. Refer to Tile Vertex Count Row in mesh Generator to understand how to do it.

After that, you should match the Grid with your scale, Tile size, and fill out the fields from the MeshToTerrainTiles. (You can use the prefab from the demo, if you want. Or create one yourself)



Then, you only need to press Generate Terrain Tiles, and It will start slicing the mesh and Heightmap in tiles.

Be careful with either slicing very large meshes, or very small tiles. As it can take a while before it finishes. (To Do: Making this using GPU, for faster loading times. Right now, it's made in C#)



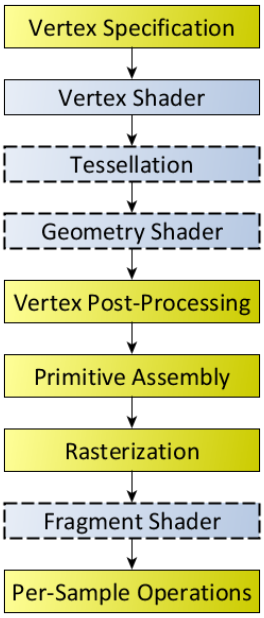
And your mesh is sliced.

Creating your own Tessellation shader

To use the Shaderlab library, first is recommended to know a bit of Shaderlab. It's in another language than c#, hlsl. And since it's a high-level language it's a bit harder to get around.

ShaderGraph is useful, but you are limited to nodes, and you can't use some of the stages of the shader pipeline directly, like the Hull and Domain stages (Tessellation stage).

When using ShaderGraph, you only really use Vertex and Fragment stages by default, And maybe Geometry stage. Either way Geometry stage is used less and less, as its inefficient.



The main problem with Shaderlab stages, is that every stage needs the output of the stage before, meaning you can't just call a function in your shader code to tessellate.

You must edit your code, so that the vertex stage outputs the data structure to Tessellation stage, and that your fragment stage receives the output of the tessellation stage.

Instructions how to exactly achieve that are in the example shaders in the demo, and shouldn't be too hard if you know your way around hlsl.

In written text it might be hard to understand, and I think it's easier if you just read the commented code.

That being said, instead of adapting a shader to tessellation, I recommend just creating a new shader by copying either the example, or the tessellation shader in the library. And creating your shader from there.

Either way, you must add in all 3 files (.shader, .cginc and Input.cginc) every new variable that you use in your shader. That way its interpolated properly, and things like normals, tangents, and vertex color are all gradual with tessellation.

4: Extra Observations

How to Create Heightmaps

Heightmaps are usually done in specialized programs made for creating them. As there are some realistic effects that can be calculated programmatically, instead of simply drawing them in an image editing software. That way, you can truly get more detail, and more realistic looking effects. Although you can just create the images yourself if you would like.

One software I recommend is Gaea, for terrain generation. The free version has a lot of features, and you can export until 1024x1024 pixels resolution freely. Including a lot of nodes that simulate complex effects.

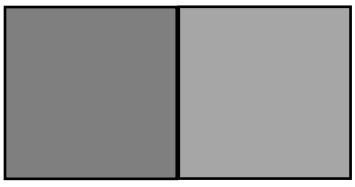
But that is just a software of many. You can research more programs If you would like.

Properly Slicing Textures Externally

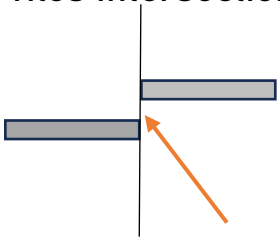
Apart from having to make tiles so that the terrain is continuous (meaning you shouldn't jump abruptly in height between tiles). There is also a small detail to take into account when slicing your terrain in tiles externally.

Take this example. Imagine a tile that ends with height 4.5 and the next tile begins with height 5. Since each tile is a separate mesh, there would be a gap between 4.5 and 5 in the tile. Something like this would occur.

Image 1 | Image 2

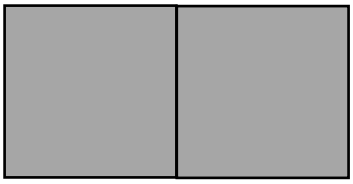


Tiles Intersection



For that reason, you must ensure both tiles ends and begins with the same color. That way, even though both tiles are different meshes, since the share the vertex height and position, the fact that they aren't connected is imperceptible.

Image 1 | Image 2



Tiles Intersection

That being said, this is only a problem when you generate different meshes with different mesh generators. As I average the value of the last pixel with each tile is connected to in the mesh slicer.

When slicing the images externally, you must ensure that the connecting pixels have the same color, to achieve that you can:

A) Average the border pixels with their connecting tiles
B) Repeat one pixel when slicing.
   For example: Image1: 4.3, 4.5 Image2: 5, 5.7
   Should be instead: Image1: 4.3, 4.5 Image2: 4.5, 5, 5.7

This applies to all directions of the tile. Not only from left to right.