UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

Procedural Textures Generation
Adaptation into a Unity tool

Treball de Final de Grau

Grau en Disseny i Desenvolupament de Videojocs

Alumne: Busquets Duran, Roger
Pla 2014

Director: Zuñiga Zarate, Anna Gabriela

Summary

This is a description document that explains the procedure and development of the *"Procedural Texture Generator"* tool for Unity. It is a node-based editor that allows the user to create textures by generating different types of noises, combining them, and using several filters to generate the different textures needed for PBR materials.

In this document there are described the techniques and procedures used for procedural texturing and the adaptation into a Unity tool, as well as all the difficulties encountered during the development.

Keywords

Procedural generation, Textures, Unity, Videogames, Programming, C#

Links

Source code:
https://github.com/rogerbusquets97/ProceduralTextureGenerator
Github Release (Import it as a Unity package):
https://github.com/rogerbusquets97/ProceduralTextureGenerator/releases/tag/1.0

Tables

Figures

Glossary

GUI - Graphical User Interface. Visual elements through which the user interacts with the application.

PBR - Physically based rendering. Computer graphic discipline that attempts to simulate lightning by using physics.

PCG - Any content created by a computer through an algorithm.

FDD - Feature Driven Development. Agile development methodology.

GPU - Graphics Processing Unit. Computer component specialized in graphics rendering.

RGB - Red, Green, Blue channels of a pixel color.

1. Introduction

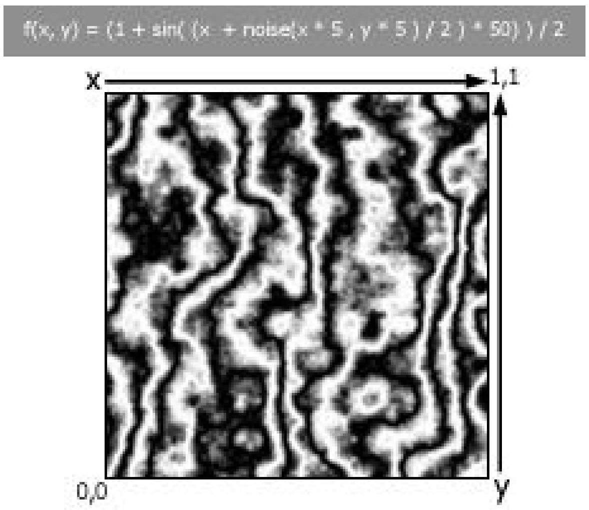
In this document I will analyze how procedural textures are generated, their pros and cons, as well as how to use Unity's in-editor GUI system to create tools, in this case how to create a node based editor. Regardless of its implementation in Unity, the theory applied could be used in any implementation.

Procedural content is any content that has been generated by a computer using an algorithm. Procedural generation is being used in so many areas of video games, from procedural terrain generation to procedural animations. Its main strength is that using procedural content, we can speed up the production process. One of its main applications is, of course, procedural texture generation.

As we know textures are used in video games to apply color to our models (diffuse texture), and to add extra detail (normal maps, occlusion maps, roughness maps, ...). Traditionally all this textures were hand-made by artists or taken from photos. Even though using, those methodologies we can achieve good results, and we have all the control over the look of each texture, the process to generate them is quite long.
To speed up the texture creation process we will use procedural generation. Procedural generated textures are those textures that are created by a computer through an algorithm. (different algorithms will be explained later on).

Even though procedural textures can generate good looking results, we do not have much control over it so most likely most of the textures generated will end up looking alike. So, the next step is to allow some user interaction over the procedural creation process. The idea is to use different procedurally created textures as a base and apply transformations and filters to them so the user can get to the desired result, and that is the main purpose of this tool, to allow users to generate procedural textures and tweek them as they want to fill their needs.

In the image below that function has been used to calculate the grayscale color for each pixel.


f(x, y) = (1 + sin((x + noise(x * 5 , y * 5) / 2) * 50)) / 2
x
1,1
0,0
y

F1.1- Procedural texture sample

1.1. Motivation

I am not really skilled when it comes to art, so whenever I have tried to make a game one of the most difficult tasks that I have encountered has been the texture creation process. Even though we can get textures for free from different websites, most of the time you do not find exactly what you were looking for, so I end up discovering procedural texture generation.

There are several software applications specialized in procedural texture creation, most of them use a node based editor interface. The problem with those applications, of course, is that they are not free, or cheap.

Then is when I decided that making my own application for creating those textures would be ideal, not just to share it with other indie developers, but to learn and understand in depth all the technologies involved.

Why Unity? Because is the engine that I have been using the most, and learning how to develop my own tools for it is a critical skill, that, for sure, will be helpful in the future.

1.2. Problem Formulation

As I have explained procedural texture generation is being used in the industry, but the main problem is that it is not accessible for indie developers, even though there are several alternatives.

Another problem that I have experienced is that whenever I have tried to make textures in a procedural texture software, the resulting look will vary when you try to export those texture and use them inside your game engine. Since those programs are optimized and thought to look as realistic as possible unlike game engines. Also the exporting/importing process sometimes can become tedious.

So with my implementation I am trying to solve those issues, to make it accessible to everyone, to allow the user to see directly how it would look inside the game (since everything will be running in the same engine), and to speed up the exporting/importing process.

1.3. General Objectives

The main objective is to create a free Unity tool that allow users to create all the textures needed for PBR [1] texturing procedurally.

Objectives:

- Develop a Unity tool to generate procedural textures.
- Develop an easy-to-use and intuitive GUI [2].
- Provide different filters and noises to give users as much control as possible over the look of textures.
- Publish it in the Unity's Asset Store.
- Create a sample scene that uses only textures created with this tool.

1.4. Specific Objectives

In order to achieve my general objectives, the following technologies and objectives will be studied in a more specific way:

- **GUI:**
    - **Unity in-editor GUI**: Learn how Unity's in-editor GUI works in order to be able to develop custom tools, in this case the node based editor, but this knowledge can be applied in any tool that I may want to develop in the future.
    - **Node based editor:** Understand and develop a node based editor interface that allow the creation of different nodes that can be connected between them, so that the concatenation of different nodes will derive into a specific result. Develop it in such a way that it could be applied not only for procedural texture generation.
- **Texturing:**
    - **Noises**: Apply and understand several noise algorithms used for procedural texture generation.
    - **Filters**: Apply and understand image filtering and processing techniques that could be applied as well for procedural texture creation.
    - **PBR**: Understand how each texture needed for PBR texturing works in order to be able to generate them.
- **Optimizations:**
    - **Multithreading:** In order not to freeze the Unity editor, since most of the nodes will perform heavy computations, the tool should use multithreading.
    - **GPU[3] computations:** As said before some algorithms are heavy to compute and in addition to multithreading some computations could be done in the GPU using **compute shaders.**
- **Usage:**
    - Use this tool to generate textures for a sample level.

1.5. Project Scope

Who am I targeting with this project?

The main target of this project are indie Unity game developers, who cannot afford buying licenses for procedural texture generation software and want fast and good results, regarding texture generation, and don’t have the time or resources to spend in developing their own procedural generation tools. Of course I will not be offering the same amount of features that most of professional texturing software offer, but I will offer enough to do simple textures, that could be used in any game. Furthermore, it will also let them see the results directly inside the engine.

It may also interest coders, interested in procedural generation and image processing, since the code will be available at github. The idea is that in a future, anyone could add extra features to the tool.

2. State of the Art

2.1. PBR

Before talking about procedural textures it is important to understand which kind of textures are we going to generate, in this case we are going to generate all the textures needed for PBR (Physically Based Rendering).

According to *Wikipedia*, PBR is a philosophy in computer graphics that seeks to render graphics in a way that more accurately models the flow of light in the real world, having as objective photorealism.

In order to do so several textures are used, each one interpreted in a different way inside the shader.

2.1.1. Diffuse texture

The diffuse texture was also used in non-PBR texturing. Is the texture that holds raw color data of 3D objects. Basically it is a 2D file that is wrapped around a 3D model using UV mapping.

The following texture is a diffuse texture for a rock wall:

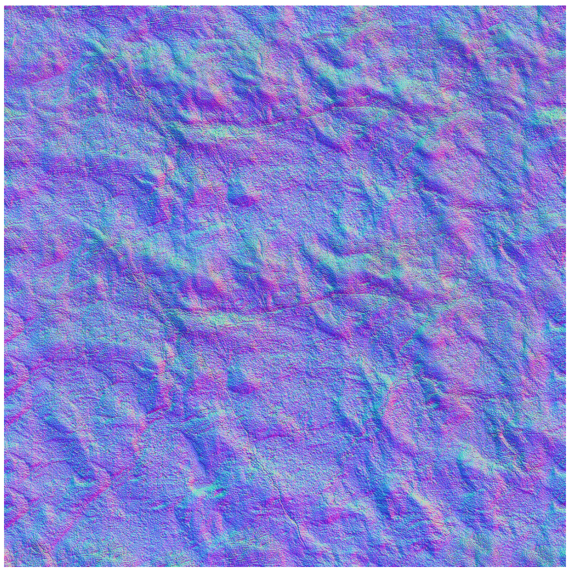
F2.1 Diffuse Texture sample

2.1.2. Normal maps

Normal maps are used to simulate lighting of bumps over a surface of a low-poly model giving it the appearance of a high-poly model. Normally they are baked from a high-poly into a low-poly transfering the detail of the first into the other.

Normal maps use the RGB[4] channels of the image to give information about the X, Y and Z spatial coordinates. So for calculating lighting, instead of using a normal per-vertex, with normal maps we can use a normal per-pixel, this way we can increase detail by a lot.

The following texture holds the normal information of each pixel from the previous rock wall.



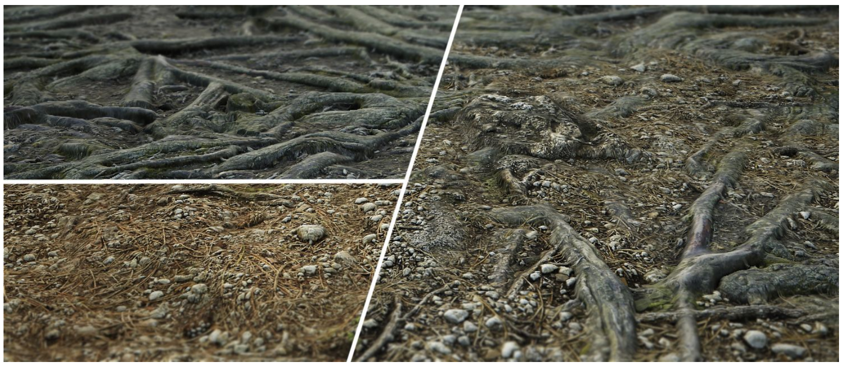
F 2.2 Normal map sample

2.1.3. Height map

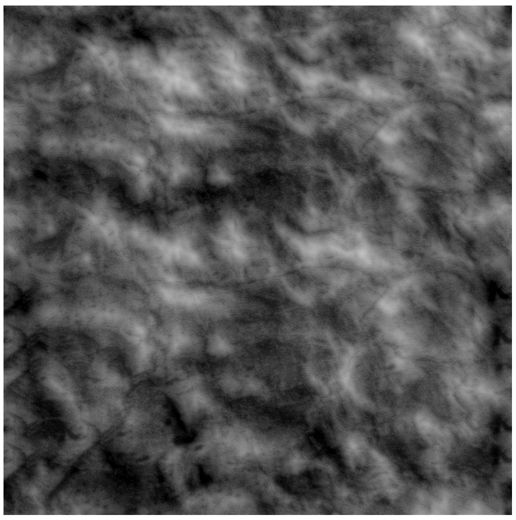
Height maps hold information of each vertex height. Their grayscale value is used to determine the height. From a 0 (black) to 1 (white) value it is specified how high the vertex should be. The greater the number of vertices of the model the better will look the result, reason why, most of the time, they are used over tessellated surfaces, so they can provoke performance issues.

They are most commonly used to make terrains, this way the terrain creation process is pretty much automatic once you have your height map defined.

They can be also used to blend in a natural way between different materials. In the following image a heightmap has been used to determine which of both materials should be displayed, normally it is calculated in the shader.


F 2.3 Height blending sample

The following heightmap is the one used for the previous rock wall example:


F 2.4 Heightmap sample

2.1.4. Specular Map

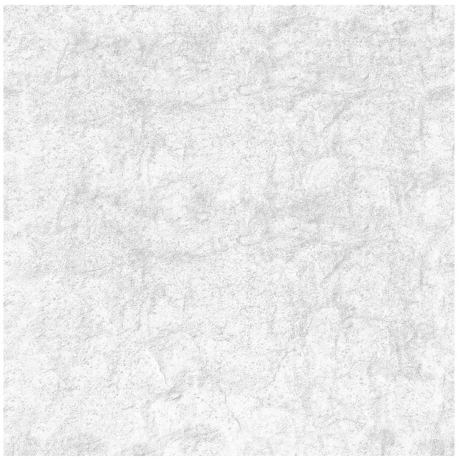
Specular maps are grayscale images that determine how shiny and reflective a surface is. The higher the value the brighter and reflective will be the pixel.

In the case of the rock that we have been seeing, the specular map would be just black, since rocks are not reflective. Normally a texture to specify that each pixel is black won't be used, we would just specify that the specular value for the whole material is 0.

2.1.5. Ambient Occlusion

Ambient occlusion is a grayscale texture that specifies how exposed each pixel is to ambient lighting. The darker the value the less light it receives. It is a quite simple technique and tends to give great results.

The following texture is the ambient occlusion map for the rock wall example:



F 2.5 Ambient occlusion sample

As we can see in the rock cracks the occlusion value is darker, so light will be less effective there.

2.1.6. Roughness/ Glossiness map

Roughness maps are grayscale images that specify the irregularities of each pixel, so determines the light deviation that certain pixel would have. The greater the value the rougher the surface will be. In the case of the rock the whole texture will be white, since rocks are rough.

2.1.7. Metallic map

Metallic textures are grayscale images that specify the metalness of each pixel, the greater the value the more metallic it will be. In the example of the rock the texture would be filled with black, again the good practice in this case is to use a single value to specify the metalness, this way we avoid having too many textures in the GPU.

2.1.8. Grayscale Images

A grayscale image is such image where the Red, Green,Blue and alpha channels hold the same number. So theoretically we do not need all the channels to hold that information, one single channel is able to carry all the information needed. We can take advantage of this situation by combining different grayscale images in a single one. For example, we can join the metallic map, the ambient occlusion map, the heightmap and the specular map in a single image giving to each channel the value of each one of the maps. This way we can have only one texture loaded instead of four.

Recently Unity released a new rendering pipeline and changed most of the materials that were used previously. Most of them uses this technique to optimize the usage of the GPU.

2.2. Procedural Content Generation

2.2.1. Introduction to PCG

Procedural content generation(PCG)[5] in video games is any content that has been produced by the computer through an algorithm. It has many applications besides video games, but the following are some of the applications and benefits in video games:

- **Replayability**: Using procedural generation we can create infinite levels, a game that generate dungeons procedurally will offer a different experience any time we play. Games like “Diablo” or “Rogue” started using this techniques.
- **Terrain creation**: As explained previously we can generate heightmaps procedurally and use them to create terrains, this way the process is automatic. We can also compute the height value of the terrain at runtime and create an infinite terrain. Games like “No man’s sky” uses this technique.
- **Optimize memory usage:** Following the previous example, computing things like terrain data at runtime, it will also minimize the memory usage since we won’t need to store that anywhere.
- **Speed up design process**: We can also place assets procedurally, for example grass, trees, houses, … Or even generate cities procedurally. There is no need to mention the time that could be saved using this techniques.
- **Texture generation:** We can generate textures procedurally by combining noises and filters to generate PBR textures. One of its main strength is that since it is calculated through an algorithm, procedural textures are resolution independent. They will not lose detail when scaling them. They are also really customizable since a change of one parameter can dramatically change the final result.

2.2.2. Procedural Texture Generation

As we have seen procedural content is such content generated through an algorithm, so procedural textures are defined by an algorithm. I like to think of it as a mathematical formula that determines the final output of the texture's pixel. For example we can use the pixel position to determine its grayscale value, so that the greater the X position of the pixel the darker the value will be:

```
f(x,y) = x
```

This simple formula applied to every pixel of a texture will give darker values to pixels in the left side of the texture. We could do the same but with its Y position, so upper pixels will look darker.

```
f(x,y) = y
```

As we can see it does not matter the resolution of the image we want to generate since its values will depend on this formula. Of course algorithms used are way more complex.

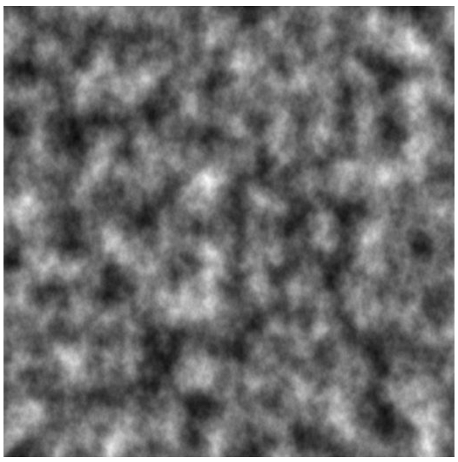
We can combine different formulas to get a more complex result.

2.2.3. Noise Generation

To achieve organic results there are different pseudo-random algorithms able to generate natural looking patterns, as well as variants of the same noise. This is a list of the most common noises used in procedural textures:

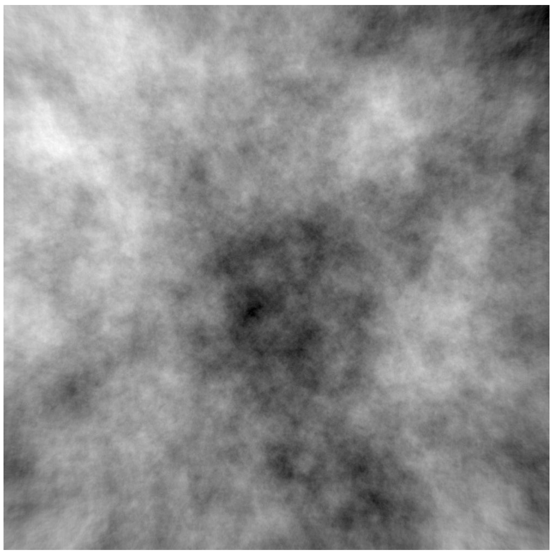
Perlin Noise:
It was developed by Ken Perlin in 1983 in an attempt to give computer graphics a natural looking feel. It is a gradient noise that involve three main steps:

- Defines a grid with randomly generated gradient vectors.
- Calculates the dot product between the distance-gradient vectors.
- Interpolates between those values.


F 2.6 Perlin noise sample

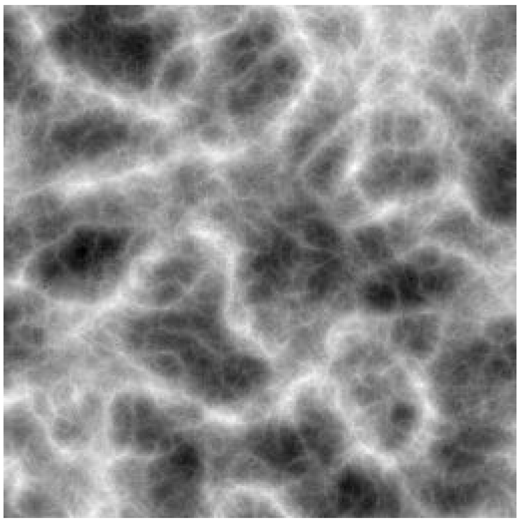
Fractal Perlin Noise:

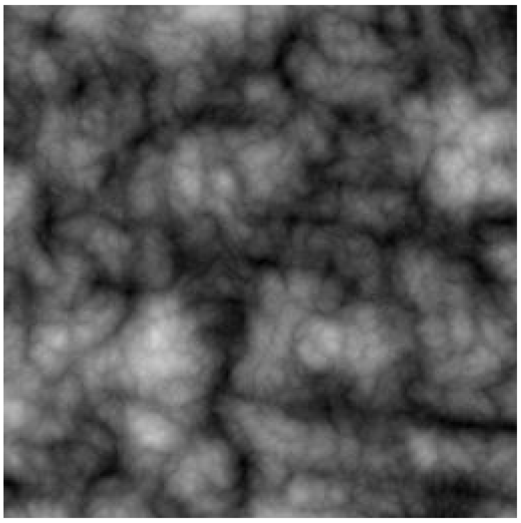
In order to get more detail we can combine several layers of perlin noise (octaves) with different intensities and sizes.


F 2.7 Fractal noise sample

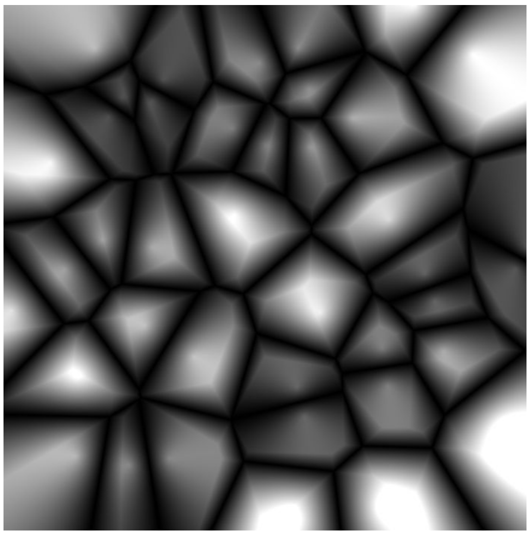
This kind of noise is one of the most used in procedural texture generation, as well as for terrain generation and clouds rendering (we can find this noise also referred as “Clouds”).

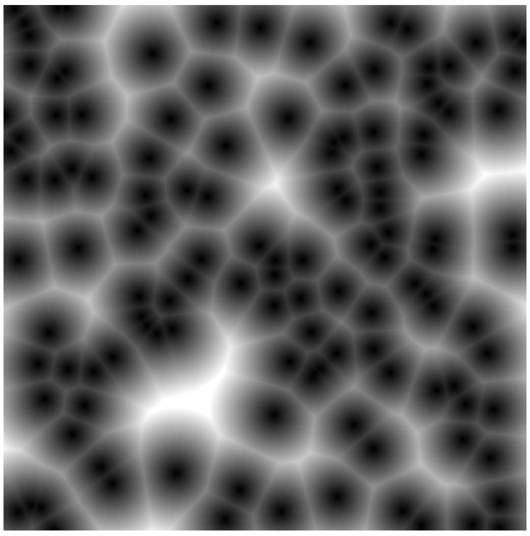
Applying variation to the result of fractals and perlin noise we can get different results like **Ridged perlin noise** or **Billow perlin noise.**


F 2.8 Ridged Perlin noise sample


F 2.9 Billow Perlin noise sample

Worley Voronoi Noise:
It is a noise function that was introduced in 1996 by Steven Worley. The main idea is that given a set of random points, the color of each pixel is decided taking into account the distance to the closest point. This noise is mostly used combined with fractals for terrain generation, for water simulations and cell noise.


F 2.10 Voronoi noise sample


F 2.11 Voronoi noise sample 2

All the previous noises work as a basis, normally we use different filters to combine and tweek them in order to give the desired shape to the texture.

2.2.4. Common filters

This is just a brief introduction to the filters used, they will be explained in detail later on, in the development section.
Blending:
Blending is a well known concept for everyone in computer graphics, or used to image editing software. For example, layers in Photoshop uses blending modes to determine how colors will look one on top of the other. There are several blending modes important for procedural texture generation:

- Multiply: As we can deduct from the name, this blending mode multiplies the pixel value of the first image with the value in the second one. Values are normalized in the 0 - 1 range, so it will give darker results. It is used to add detail from one image onto the other. Pixel values will be calculated as follows:

  ```
  f(a,b) = a * b
  ```

- Screen: This will give the opposite effect to multiply, it gives brighter results. Values from each layer are inverted, multiplied and then inverted again. As follow:

  ```
  f(a,b) = 1 - (1 - a) * (1 - b)
  ```

- Addition: Simply make a sum of the values from each layer, and clamp them between black (0) and white (1):

  ```
  f(a,b) = a + b
  ```

- Subtraction: Simply subtract values from one layer to the other.

  ```
  f(a,b) = a - b
  ```

Leveling

A level filter allows to modify the image histogram by specifying the value of pure black and pure white, it is a well known filter used in Photoshop. It is used to add contrast and adjust brightness of images.

In this case it is used to adjust the values of a heightmap, for example, if we get a heightmap with a lot of pure white values (it will not give good looking results), we can use leveling to lower its brightness.

Blur

There are a lot of different approaches to blurring, but the main idea is the same, it reduces the detail of an image, by calculating the value of a pixel taking in account the value of its neighbours and using a kernel, which is an array of N x N that travels along the image.

This is used mainly to reduce a bit the detail of heightmaps when used to generate normal maps.

Warping

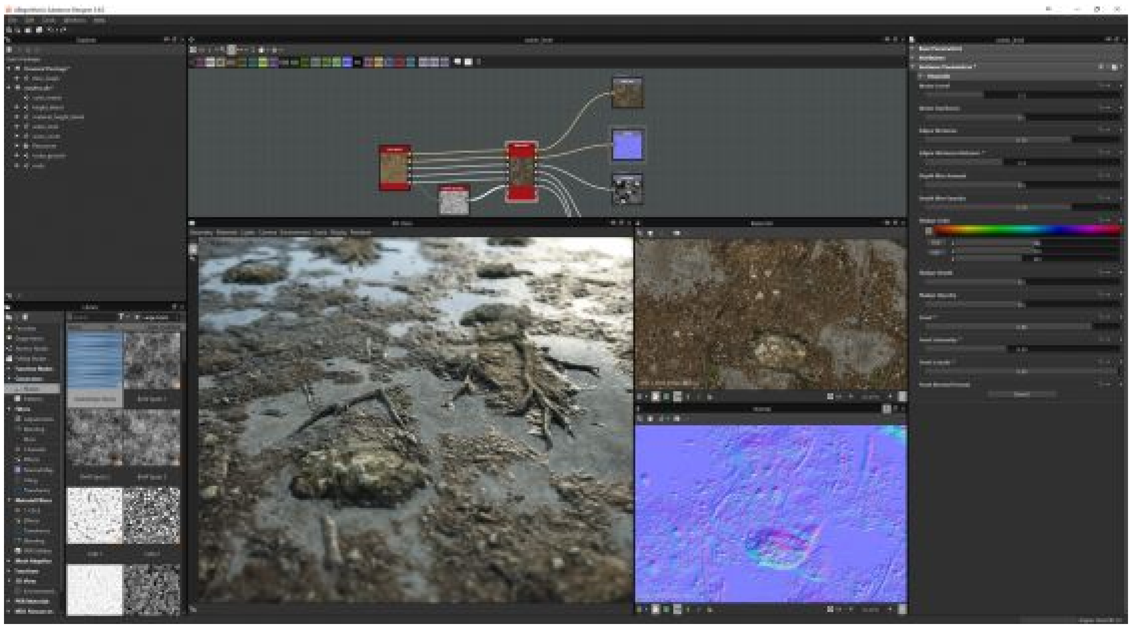
Warping, also called domain distortion is a filter used to add deformations, it takes an input which is used to calculate the deformation. In procedural texture generation we normally use noises to warp images, this way we give a natural looking feel.

2.3 Similar applications

There are several application that allow procedural texture generation, most of them are standalone applications, that generate textures that can be imported into game engines or any other texturing application.

2.3.1 Substance designer

Substance designer is probably the most similar approach to what I am trying to do. It is a procedural texture generation software that allows to create all the textures needed for PBR through a node based editor, it provides a lot of different nodes, from noise generators to any kind of filter needed.



F 2.12 Substance designer editor

It is used alongside Substance painter, which allows you to use materials created in Designer to paint meshes.

It introduces the concept of “Substance”, which is a file that contains all the texture from a material as well as some variables that can be used to change the material at runtime. This is a really powerful feature that most game engines are supporting now.

I am trying to emulate this software in Unity to make it accessible to everyone, except from the “Substance” file, which is out of the scope of this project.

There are other programs similar to Substance designer, but most of them are not free as well, and do not offer as much features and quality as Substance designer does.

2.3.2. Libraries

There are some libraries for several languages (C++, C#, Python, ...) that allows to create different noises and filters for procedural textures:

- libnoise: http://libnoise.sourceforge.net/examples/textures/index.html
- Accidental noise library: http://accidentalnoise.sourceforge.net/
- Fast noise: https://github.com/Auburns/FastNoise

Some engines also provides methods that allows you to create those noises and filters.

Those libraries are accessible for coders, since they are free, but game artists and designers would not be able to use them. I want to provide the source code, so it can be used by coders, and an interface that allows to create textures without any coding experience needed.

2.4. Game Engines

Even though textures are not only used in games, I want to focus on its applications in game development.

Game engines were not accessible back then, and were only propertie of each videogame company. Nowadays We have engines like Unity or Unreal Engines, that has made game development accessible to everyone, and this has increased the number of video game companies. That’s why I decided to make this project as a tool for an engine.

I decide to use Unity since it is the most used by indie developers.

2.4.1 Unity3D

Unity is a multiplatform game engine developed by Unity Technologies, it supports both 2D and 3D. It was first released in 2005, with the goal to democratize game development, and it has become since then one of the most popular game engines, allowing the growth of indie development.

It also provides a Store, the Asset, Store that allow developers to share their content with other developers. This project is intent to be released there.

Games like Angry Birds, Hearthstone and Aragami were developed with Unity.

Unity will provide me with the base classes and methods to create and manipulate textures as I want to generate procedural textures, as well as to modify and add extra features to the editor, in order to develop the tool.

Some of Unity features also use a node-based interface, so Unity users are used to it, in fact, this kind of interface is becoming popular within designers and artists, and most engines and modelling and texturing software are using it.

2.5. Market Study

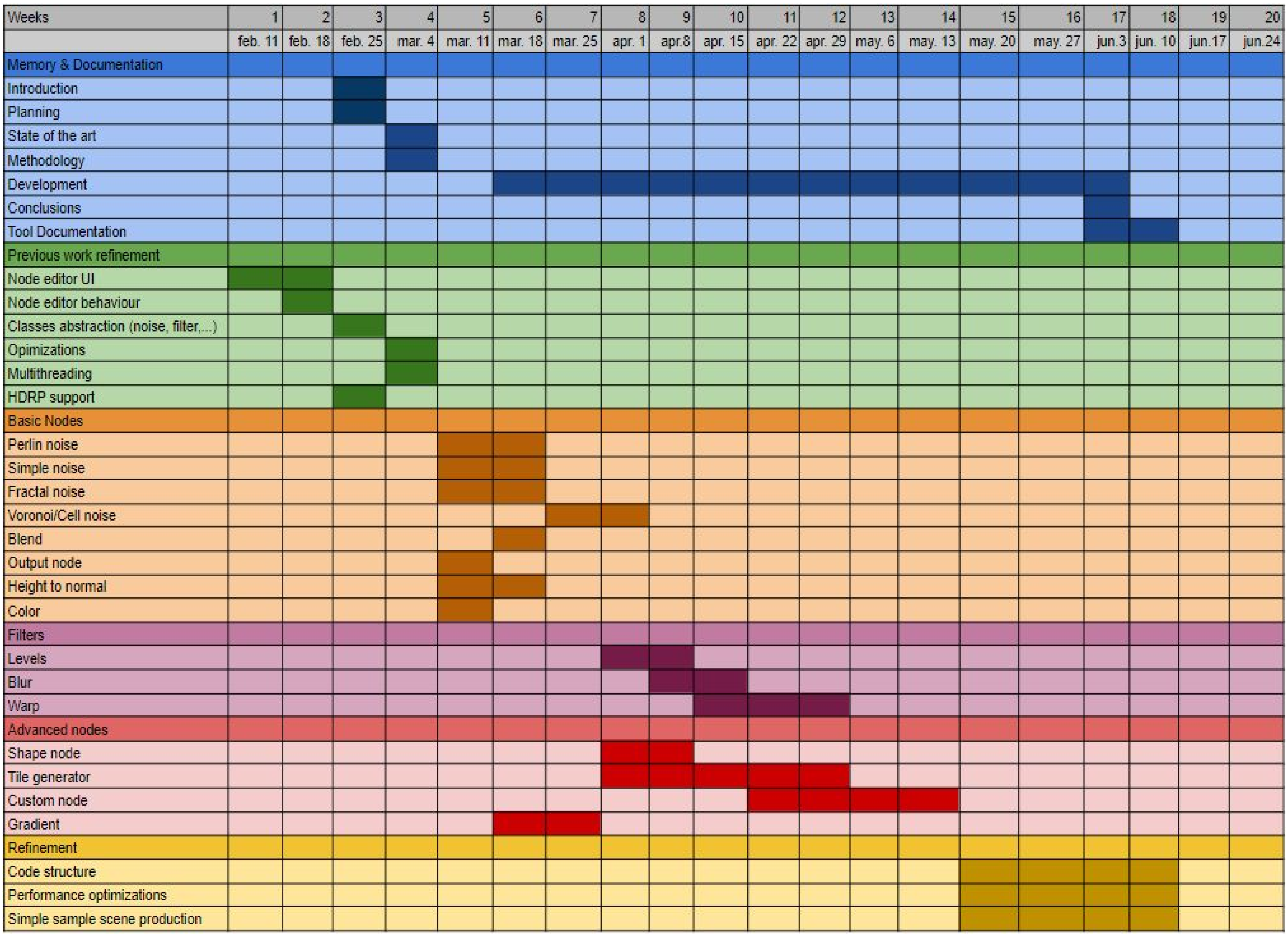
As stated before, Unity is the engine that I am going to use. Unity is one of the most used game engines in the world. In April 2015, the number of reported registered developers reached 4.5 million, with 1 million monthly active users. This means that I have the chance to reach all those users by publishing it at the Asset Store.

The asset store offers more than 15,000 pieces of content from nearly 3800 creators, and top sellers make around 30,000$ a month.

Developers save a lot of time by using assets in the Asset store, and is one of the main reasons some developers choose Unity over other engines, since most of the assets are free.

3. Project Management

In order to distribute tasks along time I have used a Gantt diagram, that shows the time estimated for each tasks and when will I start implementing it. This diagram will be used to measure how is the project performing in order to be able to deliver it at the proper date.



Weeks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	feb. 11	feb. 18	feb. 25	mar. 4	mar. 11	mar. 18	mar. 25	apr. 1	apr.8	apr. 15	apr. 22	apr. 29	may. 6	may. 13	may. 20	may. 27	jun.3	jun. 10	jun.17	jun.24
Memory & Documentation																				
Introduction			■																	
Planning			■																	
State of the art				■																
Methodology				■																
Development						■	■	■	■	■	■	■	■	■	■	■	■			
Conclusions																	■			
Tool Documentation																	■	■		
Previous work refinement																				
Node editor UI	■	■																		
Node editor behaviour		■																		
Classes abstraction (noise, filter,...)			■																	
Opimizations				■																
Multithreading				■																
HDRP support			■																	
Basic Nodes																				
Perlin noise					■	■														
Simple noise					■	■														
Fractal noise					■	■														
Voronoi/Cell noise							■	■												
Blend						■														
Output node					■															
Height to normal					■	■														
Color					■															
Filters																				
Levels								■	■											
Blur									■	■										
Warp										■	■	■								
Advanced nodes																				
Shape node								■	■											
Tile generator								■	■	■	■	■								
Custom node											■	■	■	■						
Gradient						■	■													
Refinement																				
Code structure															■	■	■	■		
Performance optimizations															■	■	■	■		
Simple sample scene production															■	■	■	■		

T 3.1 Gantt diagram

3.1. Planification changes

During the development of the project some changes has been made to the original planification.

Since textures were taking too much time to compute, I decided to start working on *performance optimizations* before it was planned since it required some major changes in the core structure of the application. In order to make it possible I extended the time needed for the *Gradient node* and the *Tile generator.*

This decision was positive for the project, since it is better to add major changes at early stages, so I did not had to redo all the nodes, just the ones that were already done.

Regarding save and loading, it was taking too much time to have it done properly, so I had to decide whether to keep working on that and cut some of the other features, or discard save and loading. I finally decide to stop working on save and loading and focus on implementing other features, since I considered it makes no sense to save something that is not complete enough. Even though serialization is something extremely necessary for a tool of this kind, since one of the objectives was to develop a tool capable of producing good quality textures, I believe this was a good decision. Save and loading will be the first thing to be done as future work after delivering this project.

I also realized that it was due to a bad planning that I was not able to develop save and loading. It should have been one of the first things to take in consideration, if I would have developed the tool in such a way that was taking into account serialization I would have not had so many problems implementing it.

3.1. Tools and Procedures

In order to manage short-term tasks, maintain code, test results and develop the project I will use several tools:

- **Trello**: Trello is a project management tool that will allow me to define the different tasks, as well as to classify them by priorities and stages until they are done. It is used mostly to follow the scrum methodology.
- **Github:** Github is hosting service for version control using Git. This will allow me to post publicly my code, and keep track of small changes over time.
- **Unity3D:** Unity is the engine I will be using to develop this tool, although it also provides version control features, I decided to use Github since it is more powerful and accessible.

3.2. Validation Tools

In order to validate each of the tasks, a classmate, David Franco, will act as a tester, and he will be the one with the authority to decide if a task could be considered "done". We will use trello, where he will be able to add comments to the tasks and move them to the "Done" column. So whenever I finish a tasks I will tell David he has a task to review, and he will give me feedback and return the task back to development or consider it "Done".

3.3. SWOT

Strengths	Weaknesses
• Flexibility and chance to take risks and try innative things • Previous experience with image processing	• Project develop as a part-time • Self-thought previous knowledge in the subject. • As I am an student I have less experience coding so code might be not fully optimal.
Opportunities	**Threads**
• It is a student project, so I can take risks, and learn from the experience. • General growth of interest in procedural content generation.	• There are existing tools that offer similar features, and better results.

T 3.2 SWOT table

3.4. Risks and Contingency Plan

During the project I might face problems, and obstacles that will stop me to get to the delivery date, this is list of the possible issue I think could be encountered and how I plan to deal with them.

Issue	Solution
Complex math involved in the noise generation algorithm that I could not understand.	Even though I pretend to code all the noises by myself, I know of several libraries that provides some functions to create fractal and voronoi noises as well as some methods to operate with them.
Time constraints	It is possible that due to too much work from other subject plus the time I spend as an intern programmer i do not have physical time to finish some of the features. ● I have planned it leaving to weeks free before delivering to put there hours that will come from other tasks. ● I also have decided not to go for GPU optimizations if I do not have the time since it is a topic completely new for me and will require a lot of previous research.

T 3.3 Risks and contingency plan

3.5. Initial Costs Analysis

The following tables show the estimation of costs over time. The first one shows the costs regarding tasks, and the second table regarding fixed costs.

	Estimated Hours	**Potential deviation**	**Planned Hours**	**Cost**
Memory & Documentation				
Introduction	1h	None	1h	9€
State of the art	6h	Low	7h	63€
Planning	5h	Low	6h	45€
Methodology	1h	None	1h	9€
Development	20h	Average	30h	180€
Conclusions	2h	Low	2,5h	18€
Editor UI				
Node-based editor UI	5h	Low	7h	45€
UI interaction	5h	Low	7h	45€
Code base				
Main classes	5h	Low	7h	45€
Basic functionalities	5h	Low	7h	45€
Noises	30h	High	35h	270€
Filters	25h	Average	30h	225€
Advanced Nodes	25h	Average	30h	225€
Optimizations	20h	High	25h	180€
Sample scene	15h	Low	20h	135€
				Total: 1.584€

T 3.4 Hour costs

Type	Subject	Price	Type	Years of amortization	Total Price
Personal	Salary	1.584€	Total		1.584€
Equipment	Desk	200€	Amortization	5	12,5€
	Chair	100€	Amortization	5	8,33€
	Computer	1000€	Amortization	3	138,89€
	Screen	150€	Amortization	3	15,28€
	Mouse	30€	Amortization	3	4,17€
	Keyboard	40€	Amortization	3	5,56€
Consumables	Sheets of paper	3€	Unique		3€
	Pens and pencils	3€	Unique		3€
Software	Visual Studio	0€	Monthly		
	Github	7€	Monthly		7€
	Trello	0€	Monthly		0€
	Unity	125€	Monthly		500€ (4 months)
Maintenance	Electricity	20€	Monthly		80€ (4 months)
	Water	12€	Monthly		48€ (4 months)
	Food	50€	Monthly		200€ (4 months)
	Internet connection	17€	Monthly		68€ (4 months)
					Total: 2667,73€

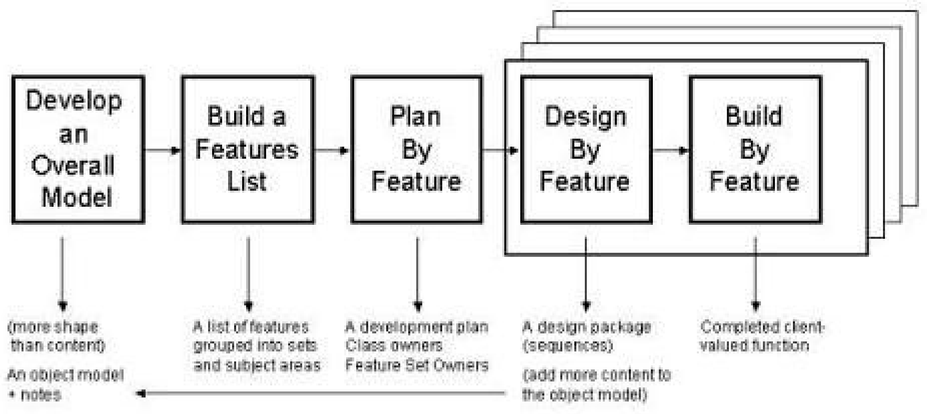
T 3.5 Costs

4. Methodology

For this project I have used the Agile methodology known as “Feature-driven development” (FDD)[6] to control the progress and steps to follow during development.

This method focuses on “Features” and separate them as different units, and sub classifying them in tasks. Dependencies and blocking features can be considered as well.

4.1. Feature-Driven Development


Develop an Overall Model
Build a Features List
Plan By Feature
Design By Feature
Build By Feature
(more shape than content)
An object model + notes
A list of features grouped into sets and subject areas
A development plan Class owners Feature Set Owners
A design package (sequences)
(add more content to the object model)
Completed client-valued function

F 4.1 FDD sample

This figure represents an scheme of the differents steps involved in this methodology. They divided in several stages, but they can be grouped in two main phases:

4.1.1. Project Model

In this first stage, an overall picture of the project is designed and planned. We do not develop any software yet, but we prepare and plan all the material that will be needed.

- The first step is to develop an Overall Model. Make clear the final goal, and all the requisites that must be accomplished.
- The second step is to build a “Feature List”. Basically we divide the main idea of the software into small tasks or “Features”, so different functionalities our application will have. This is a very important stage, that helps a lot to organize and divide work (in development teams).
- The third step is to “Plan by Features”. Once we have the feature list, now we decide when each task will be done, and what priorities each task have. We must here establish dependencies.

4.1.2 Feature Development

Once all the previous steps are done and we have a well defined feature list and correctly ordered the tasks we can start developing them.

This stage will be repeated for each of the features we need to develop.

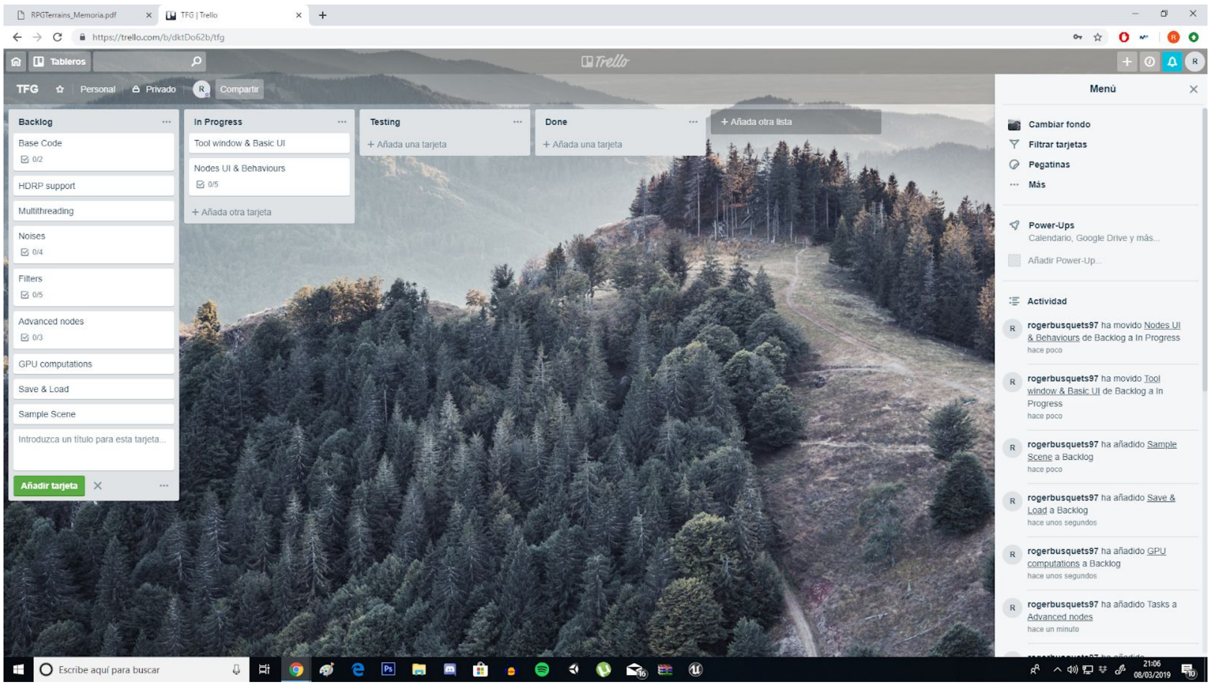
During this phase we work on each feature individually and adding them to the final product. Ideally, if we have planned correctly our project, when we finish all the features we would have a finished product. During this iterative stage we will follow three steps:

- "Design by feature": Establish different subtasks, and steps needed for the development of this single feature.
- "Build by feature": Develop and integrate the feature .
- "Test by feature": We cannot considered a feature done until we haven't test it, once it is tested and considered done, we can proceed with the following feature.

4.2. Trello

To be able to follow this methodology I have used an online tool called Trello, that allows to control tasks in a fast and easy way and supports everything needed to use this method.

The following picture is the basic interface of the board I will be using during this project.



Backlog
- Base Code 0/2
- HDRP support
- Multithreading
- Noises 0/4
- Filters 0/5
- Advanced nodes 0/3
- GPU computations
- Save & Load
- Sample Scene
- Introduzca un título para esta tarjeta...

Añadir tarjeta

In Progress
- Tool window & Basic UI
- Nodes UI & Behaviours 0/5

+ Añada otra tarjeta

Testing

+ Añada una tarjeta

Done

+ Añada una tarjeta

+ Añada otra lista

rogerbusquets97 ha movido Nodes UI & Behaviours de Backlog a In Progress
hace poco

rogerbusquets97 ha movido Tool window & Basic UI de Backlog a In Progress
hace poco

rogerbusquets97 ha añadido Sample Scene a Backlog
hace poco

rogerbusquets97 ha añadido Save & Load a Backlog
hace unos segundos

rogerbusquets97 ha añadido GPU computations a Backlog
hace unos segundos

rogerbusquets97 ha añadido Tasks a Advanced nodes
hace un minuto

F 4.2 Trello interface

5. Development

5.1. Basic Interface

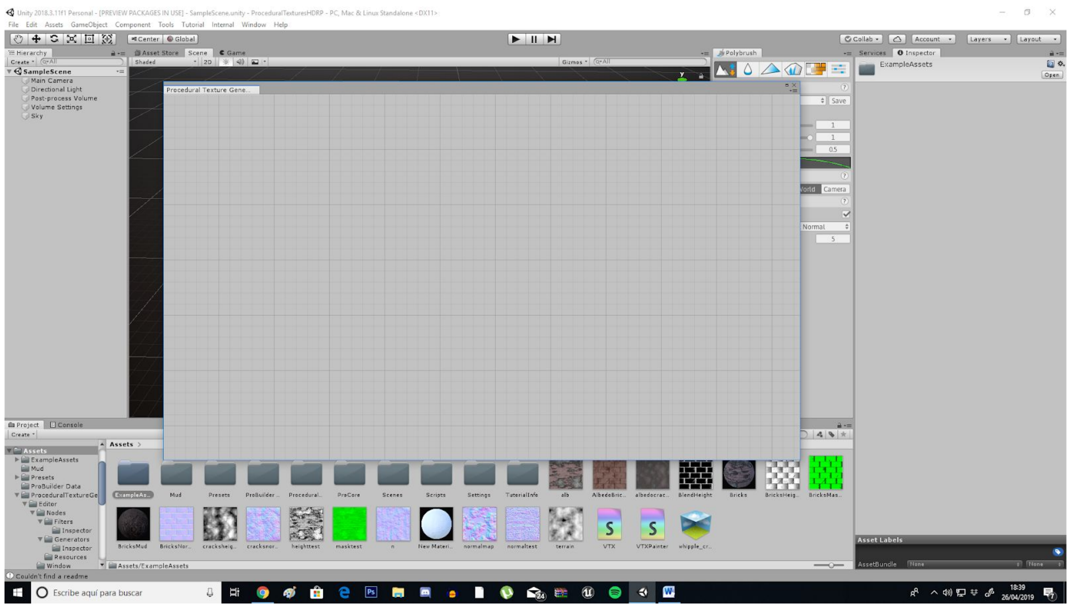
Before explaining how I have managed the procedural texture creation it is necessary to explain how the final users will be able to interact with this tool, after all it is meant to be a tool for non-programmers to ease the texture creation process within Unity.

As mentioned before I have chosen to follow the node based workflow, as many other similar applications do, and because I think it is the most intuitive and easy way to work when creating textures.

Unity give me the tools I need to develop in-editor GUI for this tool. I've used several classes that Unity provides. All the classes and code needed for developing editor GUI can be found in the Unity official web page : https://docs.unity3d.com/ScriptReference/GUI.html.

The Unity GUI system follows an immediate mode GUI also known as IMGUI. It basically renders the GUI every frame, making it to more dynamic and flexible and allowing the programmer to focus on the main purpose of the tool and not having to worry too much about UI.

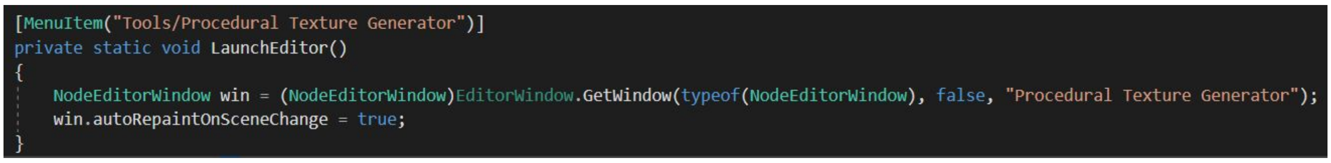
The first thing I created was a window inside the editor where the entire tool will reside. This window is similar as any other window inside the editor (the inspector, the hierarchy, ...) and it can be docked as well.



F 5.1 Empty editor window inside the Unity Editor

5.1.1. Editor Window

Creating this kind of windows inside Unity is pretty easy. The following is a simple code snippet that creates a simple window.



```
[MenuItem("Tools/Procedural Texture Generator")]
private static void LaunchEditor()
{
    NodeEditorWindow win = (NodeEditorWindow)EditorWindow.GetWindow(typeof(NodeEditorWindow), false, "Procedural Texture Generator");
    win.autoRepaintOnSceneChange = true;
}
```

F 5.2 C# code for creating Unity editor window

The first line before defining the function is saying to Unity that in tha main editor bar there must be a tab called “Tools”, that will contain a subtab called “Procedural Texture Generator”, once that tab is pressed the function specified below will be called. So the window from this tool will be accessed this way.

NodeEditorWindow is the class that I’ve created that will basically contain the tool itself. This class must inherit from *EditorWindow* which is a built-in Unity class, all windows inside the editor inherits from this class. This way we have access to the function *GetWindow*. Calling *GetWindow* will return an *EditorWindow* reference, if it is already created it will return a reference to the existing window, otherwise it will create a new one and return its reference.

Once we have the reference of our new window we can set several parameters that will specify how it will behave, in this case we are only using the *autoRepaintOnSceneChange* that is saying the window to repaint itself if something has changed (windows are not behaving as IMGUI).

5.1.2. Node Editor

Once the window is ready we can start filling it with UI elements overriding its method *OnGUI*. There we will write all the code related to UI as well as some logic. It is basically the *Update* function of the editor GUI. In my case I want to fill it with nodes, with individual functionalities each, and connect and disconnect them.

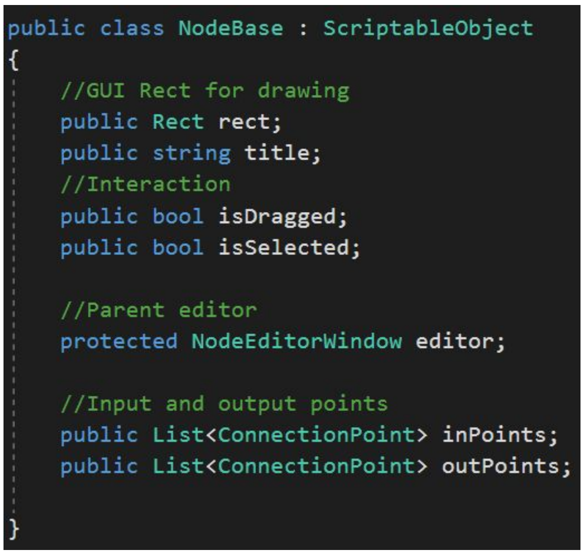
First of all I needed to create a base class for the node, *NodeBase.* All different nodes created from now on will inherit from this class, that will contain the common behaviour of all the nodes (dragging, drawing, ...).

At the end the only difference between nodes will be how they will compute the final texture that they will output.

Each node will contain at least a texture and an output point, in charge of communicating with other nodes.

This is how a basic *BaseNode* class looks like, it contains the variables needed to be drawn and to interact with, as well as a reference to the editor that contains him.

In this example we have a list of in and out points, since I'm considering that a node may need from more than one input and will be able to return several things at once. The rest of variables are pretty self explanatory. A *Rect* that specifies the position and size of this node inside the window, a title for the node, and two flags that specify if the node is being dragged or if it is selected.



```
public class NodeBase : ScriptableObject
{
    //GUI Rect for drawing
    public Rect rect;
    public string title;
    //Interaction
    public bool isDragged;
    public bool isSelected;

    //Parent editor
    protected NodeEditorWindow editor;

    //Input and output points
    public List<ConnectionPoint> inPoints;
    public List<ConnectionPoint> outPoints;

}
```

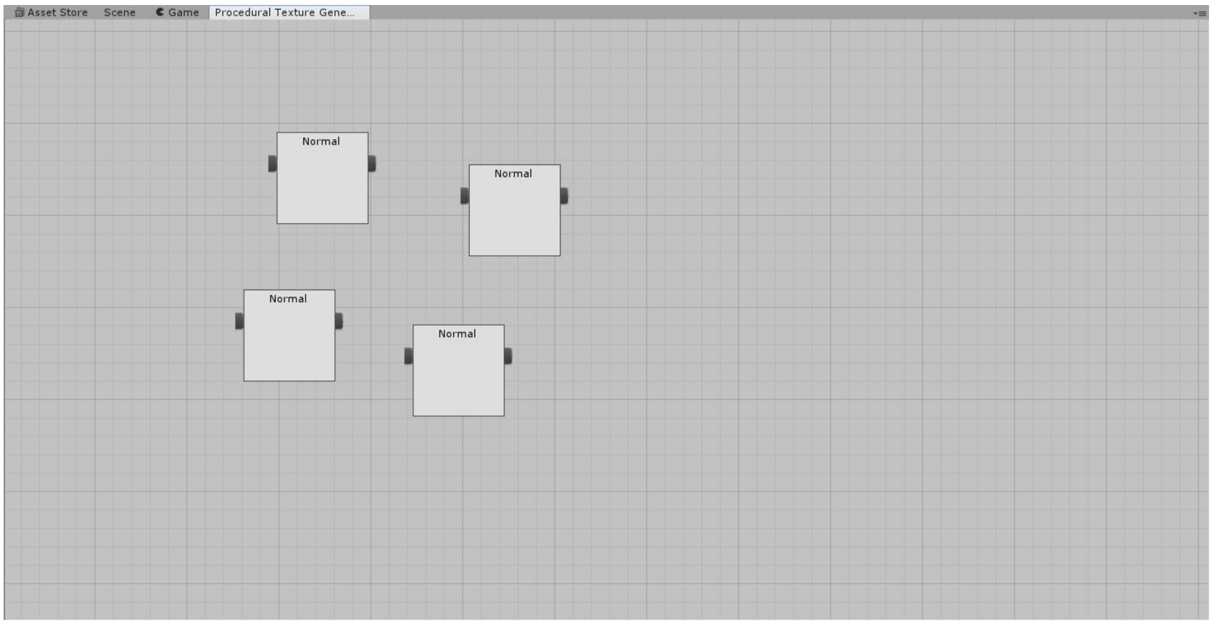
F 5.3 Node Base C# class

As we can see this class inherits from *ScriptableObject ,* it is a unity class, they are data containers that can be treated as assets in the project but they are not intended to be attached to a *GameObject*.

Once we have a *NodeBase* defined we can use them inside the editor window created previously.
Assuming that we have created a list of *NodeBase* objects inside the *EditorWindow* class we can now iterate through all of them and call their Draw function, inside the OnGUI method that we are overriding.

To draw nodes I have used a Unity function that draws a box. *GUI.Box()* will ask for a label and a rectangle to be drawn, which are the *title* and the *rect* variables from our node, that are initialized when the node is created(it will be explained later on), so the NodeBase will have a Draw method that all its childs will share, that will basically draw a box.

Ignoring the in and out points, this is how GUI boxes look like:

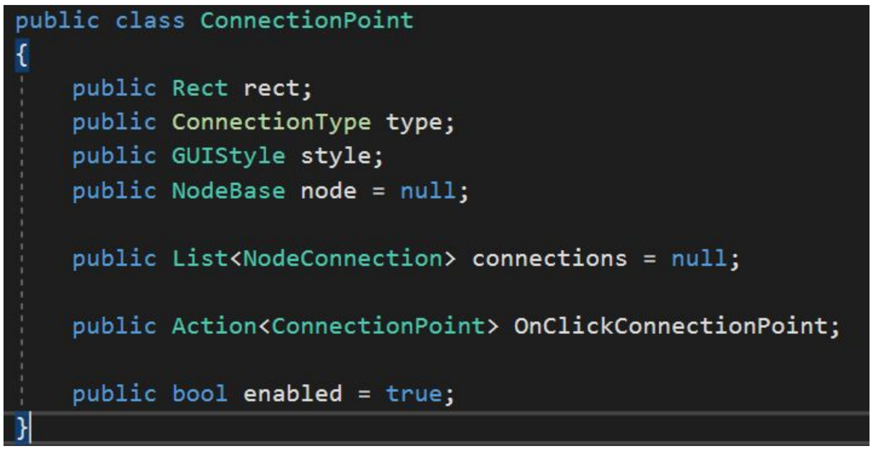


F 5.4 GUI boxes drawn inside a Unity Editor Window

This looks kind of boring, but since I didn't want to waste too much time on this I decided to leave it like this and start working on generating textures.

The next step now that we have the node ready and drawn, is to draw and create the connection points. As said before those are the responsibles for communicating nodes, and we have seen each node has two lists of *ConnectionPoints.* One for input points and the other for output points. Those points will be created in each node, so they are not initialized in the *NodeBase,* since the NodeBase has no way to know how many *ConnectionPoints* its children need. In order to work, connections need from two different classes, the *ConnectionPoint* and the *NodeConnection*.

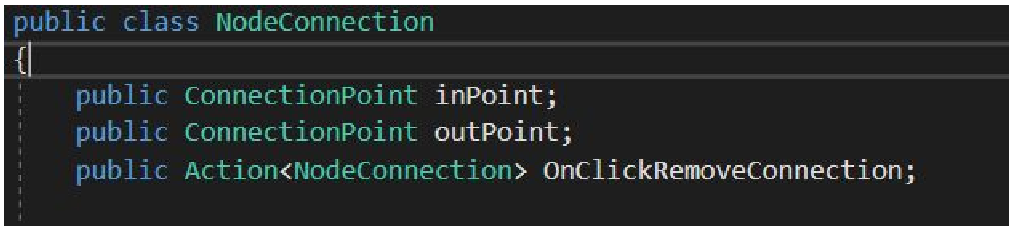
ConnectionPoint looks as follow, it contains a rect to be drawn, the type (output or input), a *GUIStyle* that will specify its look, a reference to the node that contains him and a list of connections (to other *ConnectionPoints*), an action to be called when it is clicked, and a flag that specifies if it is enabled or not.



```
public class ConnectionPoint
{
    public Rect rect;
    public ConnectionType type;
    public GUIStyle style;
    public NodeBase node = null;

    public List<NodeConnection> connections = null;

    public Action<ConnectionPoint> OnClickConnectionPoint;

    public bool enabled = true;
}
```

F 5.5 Connection point C# class

The NodeConnection class looks as follows:



```
public class NodeConnection
{
    public ConnectionPoint inPoint;
    public ConnectionPoint outPoint;
    public Action<NodeConnection> OnClickRemoveConnection;
```

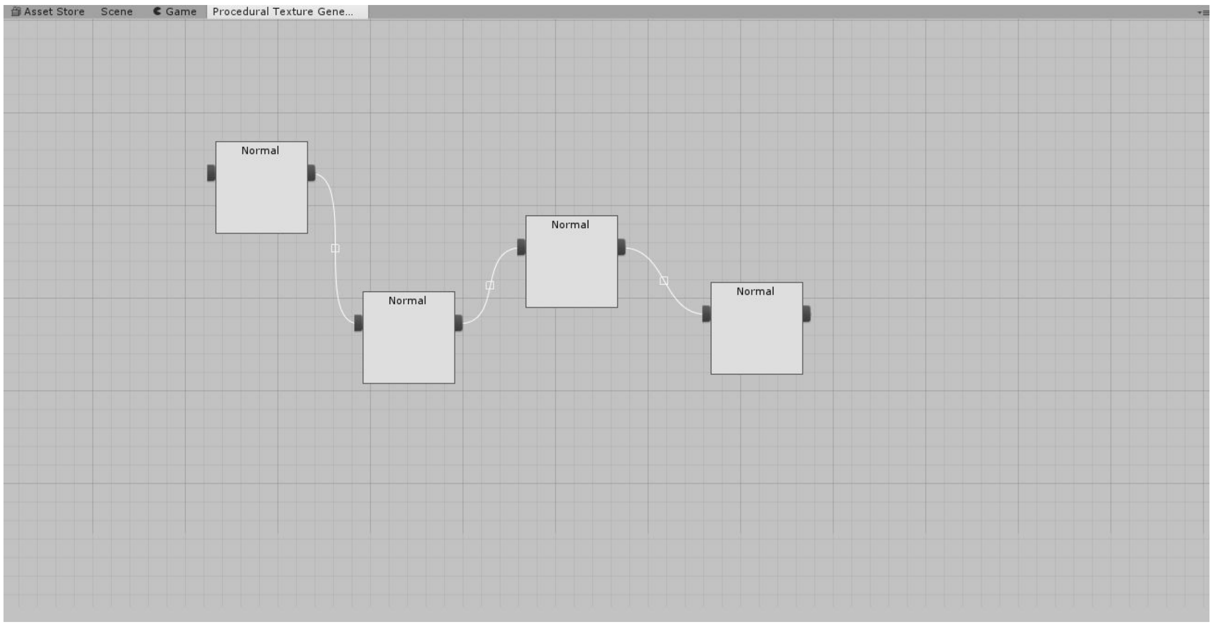
F 5.6 Node Connection C# class

It keeps a reference to the both *ConnectionPoints* it is connecting, and a action to be called when a connection is removed.

This class will be in charge of drawing the connections and to allow nodes to know to whom they are connected, so they can retrieve the input they need.

To draw connections I have used a built-in function from unity to draw Bezier Curves. *Handles.DrawBezier()*, will ask for several parameters that we can get from each *ConnectionPoint* rect. Drawing a simple line would have done the job but curves are better for visualizing things inside the editor, otherwise it can become really messy with so many lines.

This is how connection drawn with Bezier curves look like. Now we can draw and connect nodes.



F 5.7 Drawing bezier curves to represent Nodes Connections

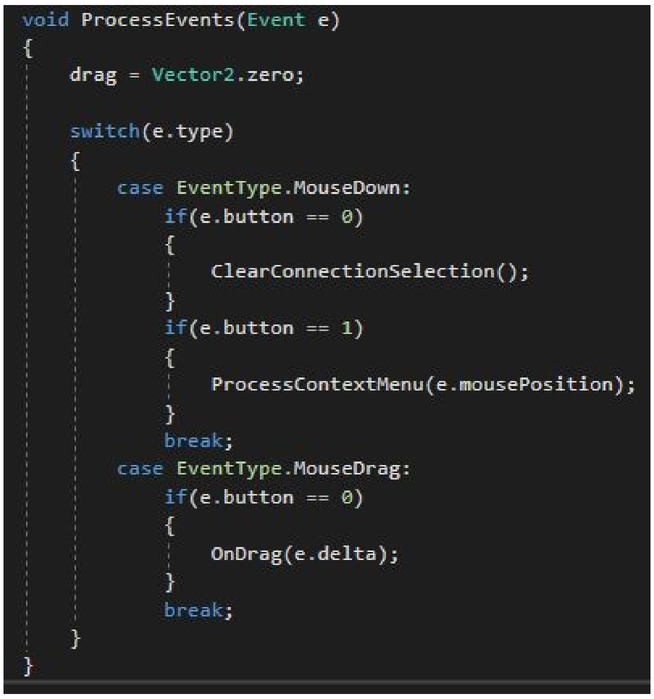
5.1.3. User Interaction

Now that we can draw things inside the editor, we need a way to allow the user to actually create the nodes and connect them.

In the OnGUI method from the editor, besides drawing nodes and connections we can also process any event that we are expecting. Unity grants access to events triggered by the user such us a mouse click.

So when the user right clicks we can perform some action that we want to do, in this case I opened a context menu at the right click position that shows a list of the nodes that can be created, when certain node is clicked I create and instantiate the desired node.

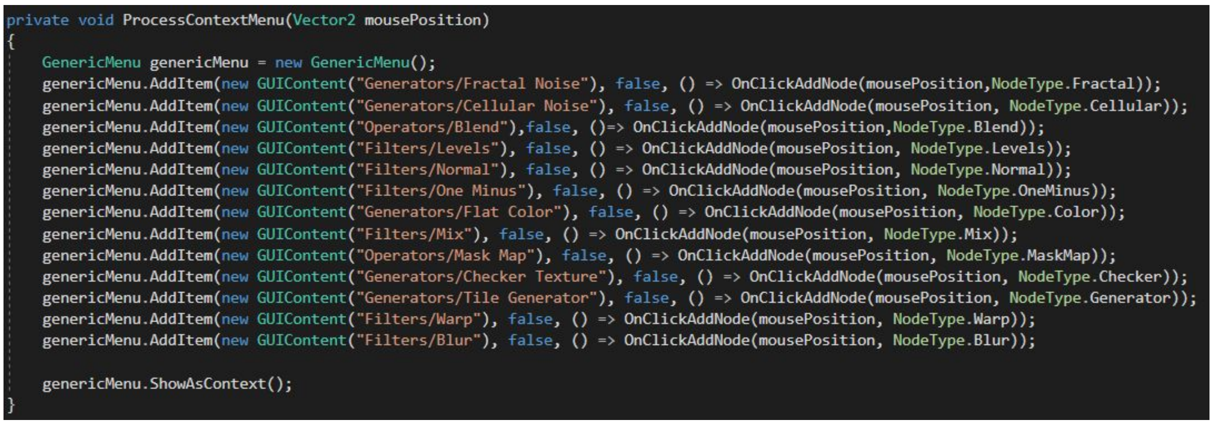
The following code receives the current event that unity is sending and process it depending on its type. In this case if it is of the type *MouseDown* (a mouse click), and the mouse button pressed is the right one (e.button == 1), I call the context menu function.



```
void ProcessEvents(Event e)
{
    drag = Vector2.zero;

    switch(e.type)
    {
        case EventType.MouseDown:
            if(e.button == 0)
            {
                ClearConnectionSelection();
            }
            if(e.button == 1)
            {
                ProcessContextMenu(e.mousePosition);
            }
            break;
        case EventType.MouseDrag:
            if(e.button == 0)
            {
                OnDrag(e.delta);
            }
            break;
    }
}
```

F 5.8 C# Processing Events code

This function creates as mentioned a context menu at the position clicked, it looks like this:



```
private void ProcessContextMenu(Vector2 mousePosition)
{
    GenericMenu genericMenu = new GenericMenu();
    genericMenu.AddItem(new GUIContent("Generators/Fractal Noise"), false, () => OnClickAddNode(mousePosition,NodeType.Fractal));
    genericMenu.AddItem(new GUIContent("Generators/Cellular Noise"), false, () => OnClickAddNode(mousePosition, NodeType.Cellular));
    genericMenu.AddItem(new GUIContent("Operators/Blend"),false, ()=> OnClickAddNode(mousePosition,NodeType.Blend));
    genericMenu.AddItem(new GUIContent("Filters/Levels"), false, () => OnClickAddNode(mousePosition, NodeType.Levels));
    genericMenu.AddItem(new GUIContent("Filters/Normal"), false, () => OnClickAddNode(mousePosition, NodeType.Normal));
    genericMenu.AddItem(new GUIContent("Filters/One Minus"), false, () => OnClickAddNode(mousePosition, NodeType.OneMinus));
    genericMenu.AddItem(new GUIContent("Generators/Flat Color"), false, () => OnClickAddNode(mousePosition, NodeType.Color));
    genericMenu.AddItem(new GUIContent("Filters/Mix"), false, () => OnClickAddNode(mousePosition, NodeType.Mix));
    genericMenu.AddItem(new GUIContent("Operators/Mask Map"), false, () => OnClickAddNode(mousePosition, NodeType.MaskMap));
    genericMenu.AddItem(new GUIContent("Generators/Checker Texture"), false, () => OnClickAddNode(mousePosition, NodeType.Checker));
    genericMenu.AddItem(new GUIContent("Generators/Tile Generator"), false, () => OnClickAddNode(mousePosition, NodeType.Generator));
    genericMenu.AddItem(new GUIContent("Filters/Warp"), false, () => OnClickAddNode(mousePosition, NodeType.Warp));
    genericMenu.AddItem(new GUIContent("Filters/Blur"), false, () => OnClickAddNode(mousePosition, NodeType.Blur));

    genericMenu.ShowAsContext();
}
```

F 5.9 Context Menu C# code

It simply creates a menu and add items to it (the nodes). We can see it is connected to *OnClickAddNode*, it is a function reference, whenever that item is pressed, this method will be called, and we send them the mouse position (that will be used to determine the rectangle of the created node), and the *NodeType*, so we know which node to instantiate.

Any kind of user interaction is done the same way, processing Unity events and calling custom functions to perform whatever we want. (Connecting nodes, deleting nodes, ...)

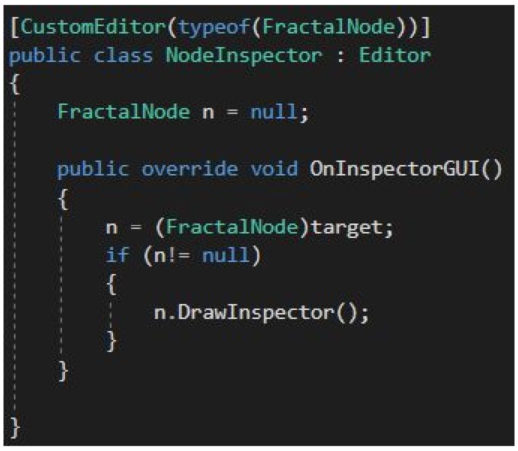
5.1.4 Node Inspector

Nodes must somehow display information about their status and variables so users are aware of what they can actually do with that node. I decided to use the inspector window, just like regular *GameObjects* inside Unity do. Also each node will render a preview of the texture inside the box.

In order to use the inspector, Unity allow us to redefine how the inspector of an object should look like, overriding the *OnInspector* method.

This would not be possible if we were not using *ScriptableObjects*, since Unity only calls *OnInspector* whenever a GameObject or ScriptableObject has been selected.

To override this method we need to create a separate



```
[CustomEditor(typeof(FractalNode))]
public class NodeInspector : Editor
{
    FractalNode n = null;

    public override void OnInspectorGUI()
    {
        n = (FractalNode)target;
        if (n!= null)
        {
            n.DrawInspector();
        }
    }

}
```

F 5.10 Custom inspector for Nodes

class for each node, that will handle the inspector drawing of that node, and it must inherit from *Editor*. At the right we can see the Inspector class for the fractal node. We are specifying that that classes of type *FractalNode* will override this method, then we are getting the target (the fractal node that has been selected) and calling its *DrawInspector* method.

5.2. Texture filtering

Once I had the basic interface working I was able to move on and start creating and editing textures.

As mentioned before each node has its own texture, that will be outputted. So each node has its own unique *Compute* method.

5.2.1 GPU vs CPU

As explained in the *State of the art,* procedural textures are defined by mathematical formulas that we define, so the final color of a pixel is determined by such formula. There are several ways to compute them, I tried different approaches but I ended up computing them on the GPU.

The firsts textures I created were entirely computed in the CPU, since most of the examples and libraries used the CPU to do that. I quickly realized that CPU was too slow to perform this kind of tasks.

To create or filter a texture we must at least make an entire iteration through all the pixels of an image. It may be fine to compute small textures like 256x256 pixels, but since this textures are intended to be used for PBR rendering a minimum quality is required, at least 1024x1024 pixels, in order to be able to get detailed textures. This means that the compute method for every node will, at least, iterate over 1048576 pixels.

Calculating such large textures in the main thread of the CPU will probably freeze the editor for a while, and make the experience quite slow. Another approach would be to use a separate thread to calculate the texture. Threads are used to be able to perform parallel computation. This way I was able not to freeze the editor but the core problem remained, textures were taking too long to compute, the editor was still functional, but textures were not ready after a few seconds.

A solution to that would have been to separate the texture in chunks and compute each one of them in separate threads. I never get to program this since multithreaded applications are complex to get working, and Unity didn't allow me to access some of its built-in classes, like the *Texture2D* class from outside the main thread.

I finally realized that what I was trying to do with this multithreaded approach was already happening elsewhere, the GPU.

GPUs are designed to rapidly manipulate memory usage and work with mathematical functions, and they highly parallelize computations, so they are perfect for image processing and rendering. Traditionally GPUs were exclusively used for rendering since they needed to output everything to a display device.

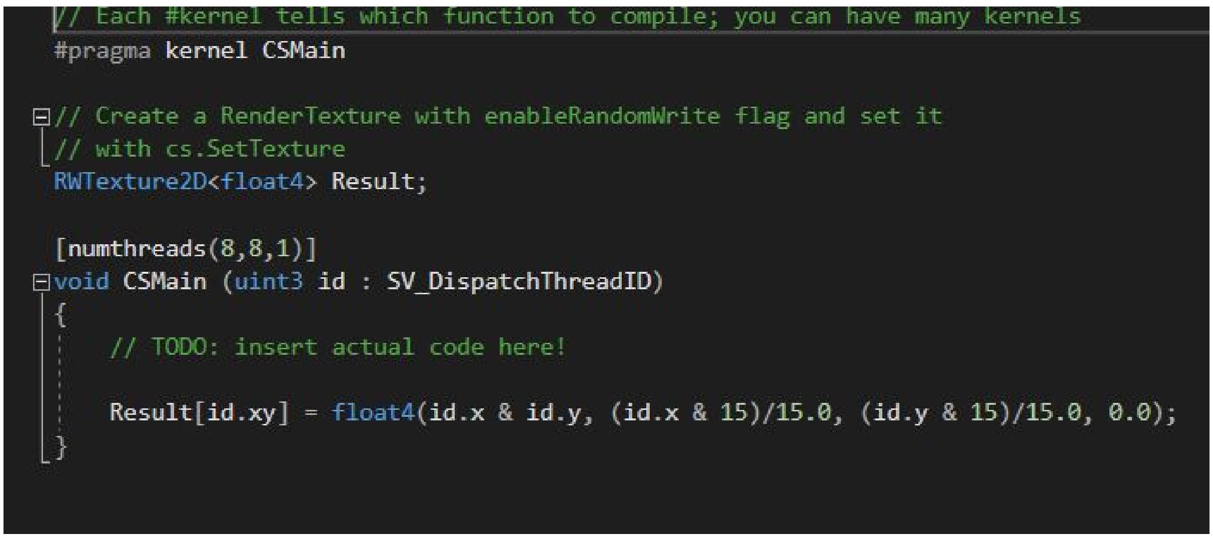
Modern GPU also allow to compute non-rendering related tasks, such us physics calculations.

The GPU uses shaders, small programs executed there. Traditionally they used vertex and fragment shaders. *Vertex shaders* are programs that are executed for every vertex of a mesh, and *fragment shaders* are computed for every pixel, this way we can calculate the final color of the pixel in the screen. As I mentioned nowadays we can also use them to compute other tasks, to do that we use *Compute shaders.*

I used compute shaders to compute each node's texture, and it resulted in a massive improvement of performance, as expected. But it also had some drawbacks. I had to rethink and redo the entire computing pipeline that I had already programmed. And also had to learn HLSL, the shading language used in compute shaders within Unity, since GPUs do not understand C#. Also, people with old GPUs won't be able to use this tool.

5.2.1.1. Unity Compute Shaders

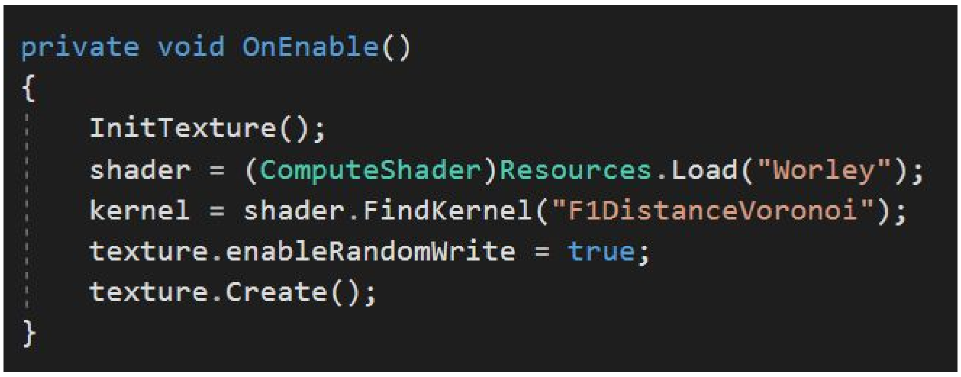
Unity allowed me to easily create and use compute shaders. They are treated as files inside the project, the only thing I had to do was to create a *.compute* file. The code below is the first thing that appears when we open a *.compute* file from Unity. Here a simple kernel is defined. A kernel is a function inside the shader that we will be able to call from our scripts. "Result" is a parameter that must be sent to the shader to compute that kernel, in this case it is a texture (float4 stands for the 4 channel color). On the upper line before defining the kernel we are specifying how many threads we are going to use to compute this, in this case 8, we could have gone higher, but 8 is more than enough for the tasks I needed to perform.



```
// Each #kernel tells which function to compile; you can have many kernels
#pragma kernel CSMain

// Create a RenderTexture with enableRandomWrite flag and set it
// with cs.SetTexture
RWTexture2D<float4> Result;

[numthreads(8,8,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    // TODO: insert actual code here!

    Result[id.xy] = float4(id.x & id.y, (id.x & 15)/15.0, (id.y & 15)/15.0, 0.0);
}
```

F 5.11 Basic Unity Compute shader

From this point I had to add a couple of tweaks to the *NodeBase* class, now each node will have a *ComputeShader* and a kernel. The shader is just the shader that it will use to be computed, and the kernel is the ID of the function inside the shader to be called.

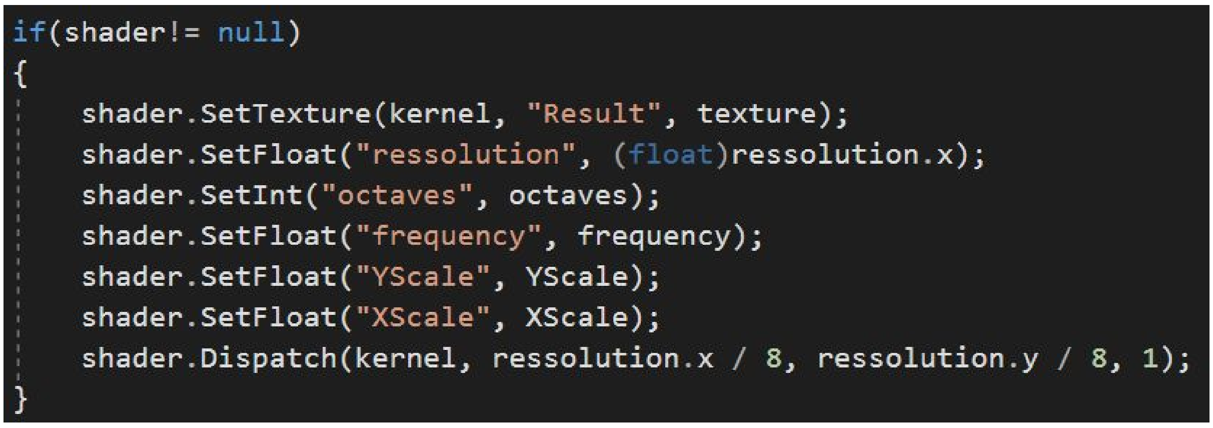


```
private void OnEnable()
{
    InitTexture();
    shader = (ComputeShader)Resources.Load("Worley");
    kernel = shader.FindKernel("F1DistanceVoronoi");
    texture.enableRandomWrite = true;
    texture.Create();
}
```

F 5.12 Initializing compute shaders

This is the code needed to initialize a node, in this case the node that produces worley noise. First I am initializing the textures(it is a inherited function from *NodeBase*), then I am loading the shader that this node will use, that is located inside a Resources folder, this way I can easily load it using the built-in Unity class *Resources*. Once I have the shader loaded I can ask to it for the ID of the kernel function “F1DistanceVoronoi”, where the actual code for computing the noise is. Finally I just create the texture.

Once I have the shader loaded and ready, before computing it I must send to it the required parameters that I know it will need to compute that kernel.



```
if(shader!= null)
{
    shader.SetTexture(kernel, "Result", texture);
    shader.SetFloat("ressolution", (float)ressolution.x);
    shader.SetInt("octaves", octaves);
    shader.SetFloat("frequency", frequency);
    shader.SetFloat("YScale", YScale);
    shader.SetFloat("XScale", XScale);
    shader.Dispatch(kernel, ressolution.x / 8, ressolution.y / 8, 1);
}
```

F 5.13 Sending data to compute shaders

Here I am setting all the data that it will need, and at the end, once all the data has been sent, I "Dispatch" it, sending which kernel I want to compute, and more parameters needed to know how many iterations it will do in each thread. Since I know that it uses 8 threads, I divide the resolution of the image by 8.

5.3. Noises

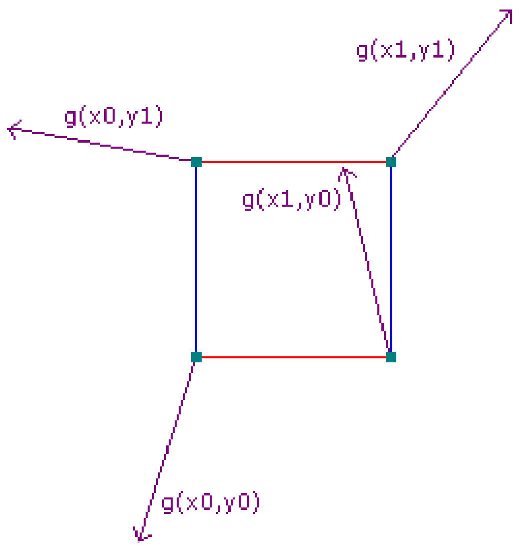
Since noises are the basis for most of procedural textures, noise nodes were the first ones I decided to make. The main noises I needed to understand were Perlin noise and Worley noise. All other noises are based on this two main noises, like ridged and billow noise, they are just variations of the same concept.

5.3.1 Perlin Noise

Perlin noise is probably the most known and used noise in computer graphics. It was originally developed in 1983 by Ken Perlin, for the movie Tron, he was, in fact, awarded an Academy Award for Technical Achievement for creating the algorithm.

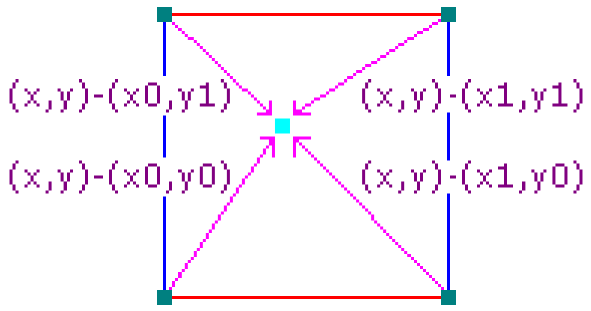
Perlin noise is a gradient noise that can be calculated for 1, 2, 3 and even 4 dimensions.

The first thing perlin noise does is to calculate pseudo random gradient vector for each unit coordinates. The quad represents the texture, and we will access to pixels inside it with values from 0 to 1 (uv coordinates), that we can get dividing the pixel number by the resolution of the texture. Gradient vectors can also be stored in an array and accessed with a hash algorithm in order to speed up the computation. Since I was using shaders, generating random numbers was not an easy task, so I decided not to use completely random numbers.



F 5.14 Perlin noise pseudo random Gradient Vectors

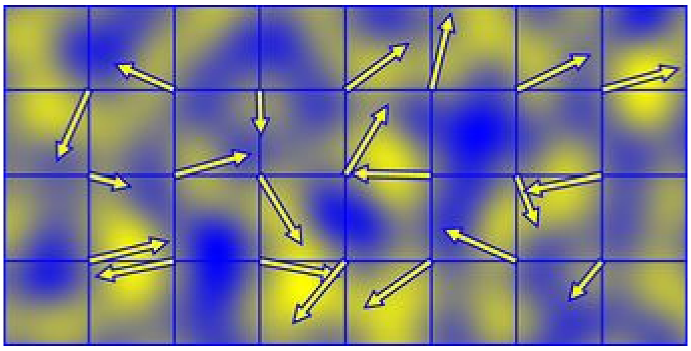
Next, it calculates the 4 vectors from the given point to the 4 surrounding points on the grid.



F 5.15 Perlin noise Distance Vectors

Then it calculates the influence value by taking the dot product between the gradient and the distance vectors.

The result of the dot product will be positive if both vectors are pointing in the same direction, negative if pointing in opposite directions or zero if they are perpendicular.



F 5.16 Perlin direction values

Then it only need to make a linear interpolation between the four influence values, this will return a value between -1 and 1 that I used as the grayscale value for that pixel. It results in a texture like this. (this texture was actually computed with this tool).

F 5.17 Generated perlin texture

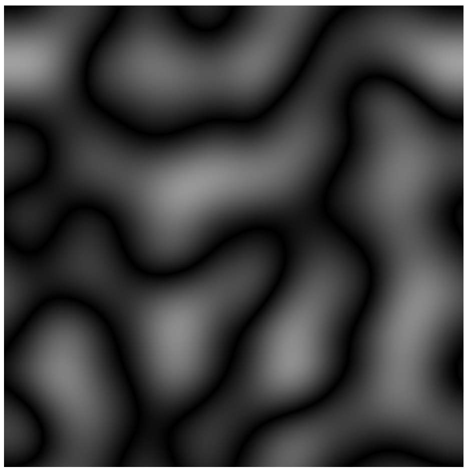
5.3.1.1. Perlin Variations

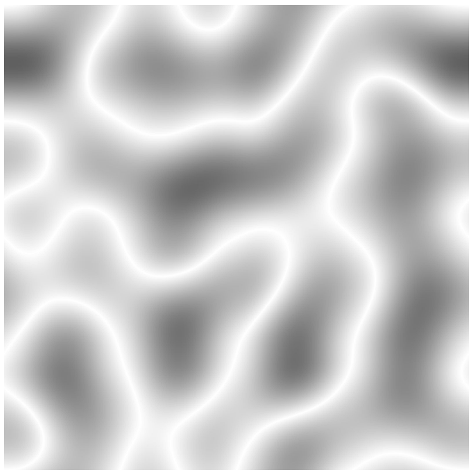
Having a perlin noise method defined that outputs a value between -1 and 1 we can now extend its applications and get different results. If we want to use the simple perlin noise we need to transform the value into the 0 to 1 range, it can be done by multiplying the result by 0.5 and then adding 0.5, then we can use this value as the grayscale for the textures (color values must be between 0 and 1).

If we take the absolute value of the outputted result we can get a completely different looking texture, and in this case it is not necessary to do any extra math to get a value between 0 and 1, since the absolute value will be already a positive number between 0 and 1, this technique is also known as billow perlin noise.

By inverting the billow result we get what is known as ridged perlin noise, it is widely used for procedural terrains, since it defines creases pretty effectively. The inverted result is calculated by subtracting the billow result to one. This way the parts that were dark before now become brighter.

By doing so, we get the following results. (They have been also generated with this tool).


F 5.18 Generated billow texture

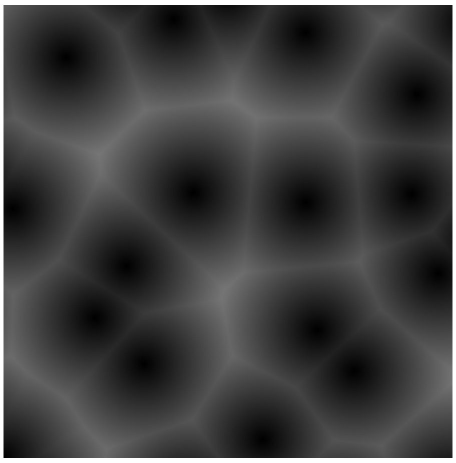

F 5.19 Generated ridged texture

5.3.2. Worley Noise

Worley noise is a distance field based noised proposed by Steven Worley in 1996. It is easier to understand than perlin noise, in fact the concept is quite simple and easy to visualize.

Given a set of points in space (texture space in my case), it calculates the value of each pixel by computing the distance between the current pixel and the closest point, so each point in the set has an influence area. This way it can simulate voronoi patterns.

Ideally the set of points should be randomly generated, but as in perlin noise with gradient vectors, in most implementations this points are accessed with a hash algorithm from an already defined array of points.



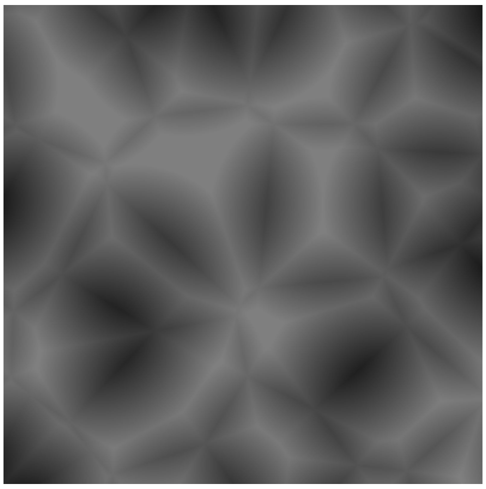
F 5.20 Generated Worley texture

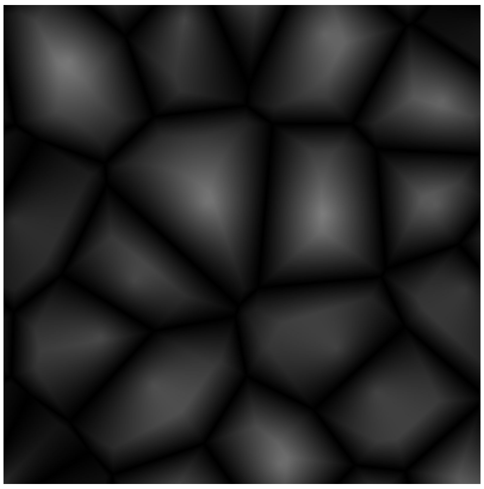
On the above image it can clearly be seen where the points are, the darker parts are the pixels closest to the points, so the center of each of this small cells is where the points are located.

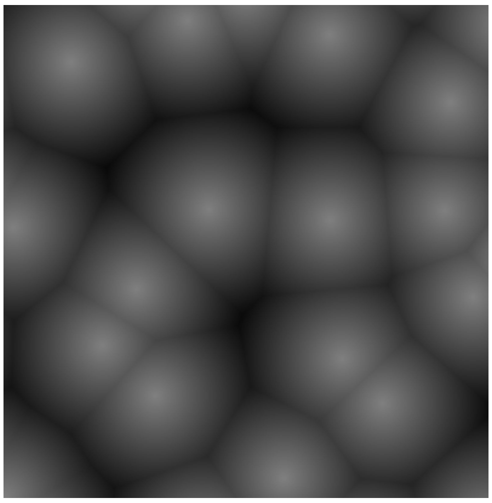
As in perlin noise some tweaks can be done to the output of this function in order to get different results.

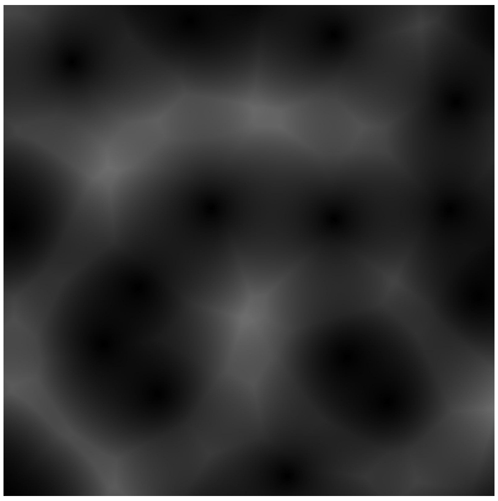
If instead of returning the distance between the pixel and the closest point, it returns a vector with the distance to the closest point and the distance to the second closest point, we can generate various interesting results. Those two values are referenced as F1 and F2.

Having those two values, by performing some operations between them we get the following results:


F 5.21 Worley: F2 (Second closest point


F 5.22 Worley: F2 - F1


F 5.23 Worley: 1 - F1

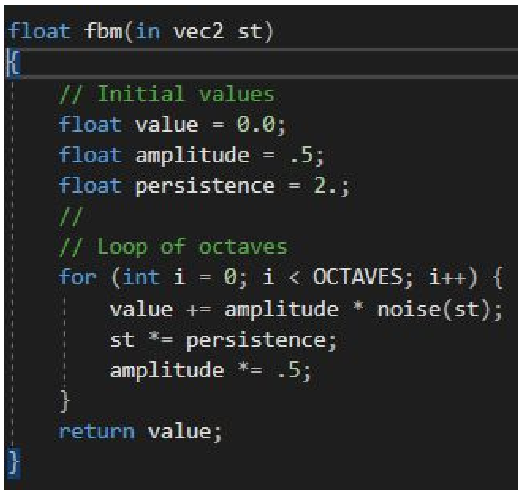

F 5.24 Worley: F1 * F2

5.3.3. Fractals

A fractal is a geometrical object which basic structure is repeated at different scales. This concept was proposed by Benoît Mandelbrot in 1975, and it is widely applied in procedural texture generation since.

If we think about the algorithms explained before, results were too simple and boring, without too much detail, but if we apply the principle of fractals to them the result will be richer in details.

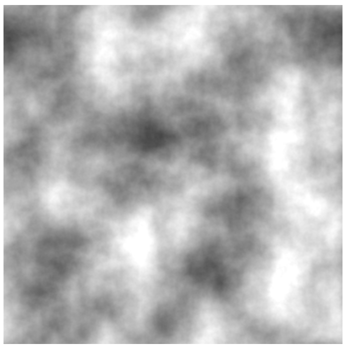
To calculate fractals it is needed the mathematical algorithm that defines the geometry of the object, the number of repetitions, also known as octaves, and the change in scale at each repetition, also known as persistence.



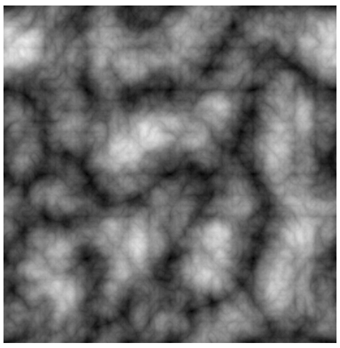
```
float fbm(in vec2 st)
{
    // Initial values
    float value = 0.0;
    float amplitude = .5;
    float persistence = 2.;
    //
    // Loop of octaves
    for (int i = 0; i < OCTAVES; i++) {
        value += amplitude * noise(st);
        st *= persistence;
        amplitude *= .5;
    }
    return value;
}
```

F 5.25 HLSL Fractal code

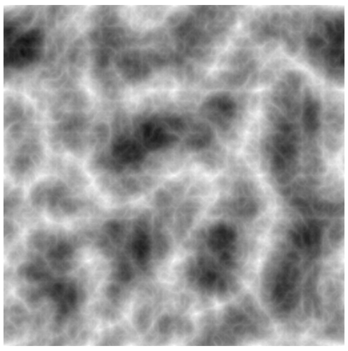
Having implemented the perlin noise method we can now use it as the formula to iterate over. In the code above we can see that we are iterating over the number of octaves and modifying the values that we send to the noise function at every iteration by the persistence value. Depending on the number of octaves the result will be better, but also slower. We can apply this concept to all the noises functions discussed before. Images below are 8 octaves fractals.



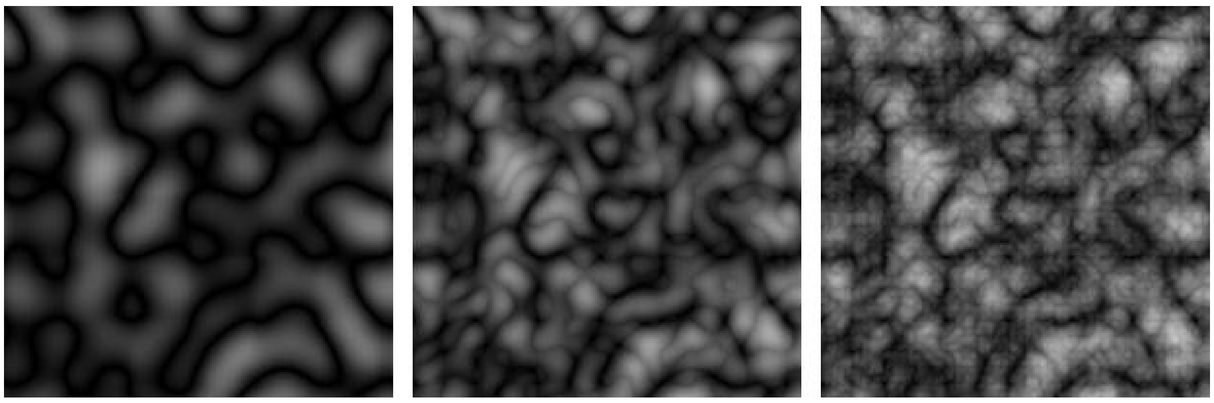
F 5.26 Perlin fractal



F 5.27 Billow fractal



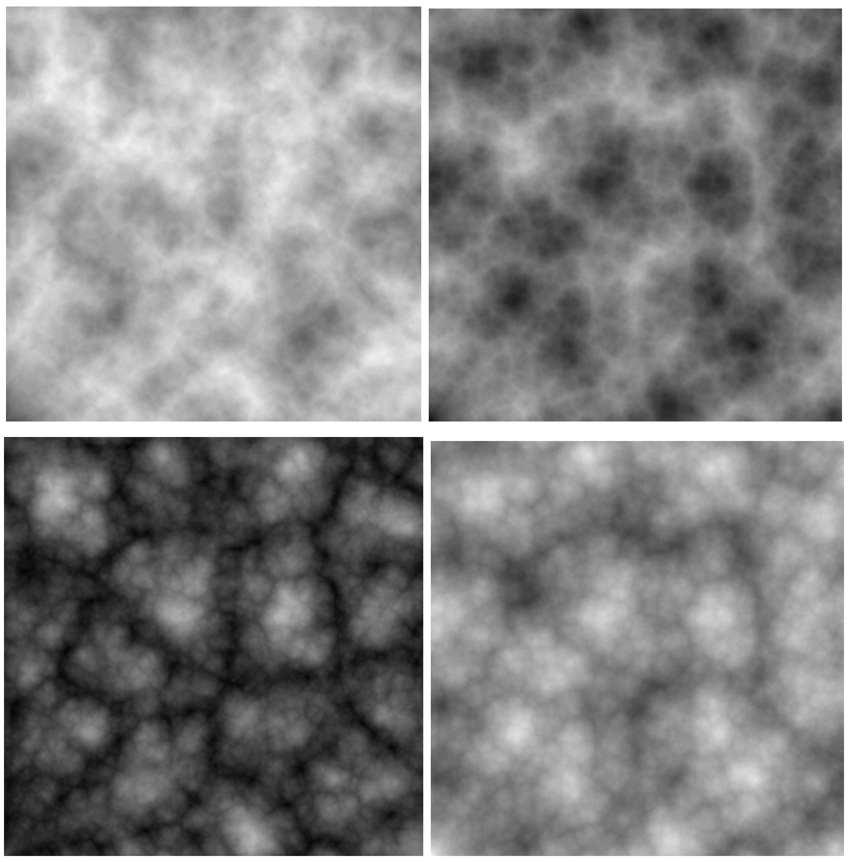
F 5.28 Ridged fractal



F 5.29 Increasing octaves to add more detail

In the image above it is clearly seen the difference in quality when increasing octaves.

The following are the worley variations as fractals.

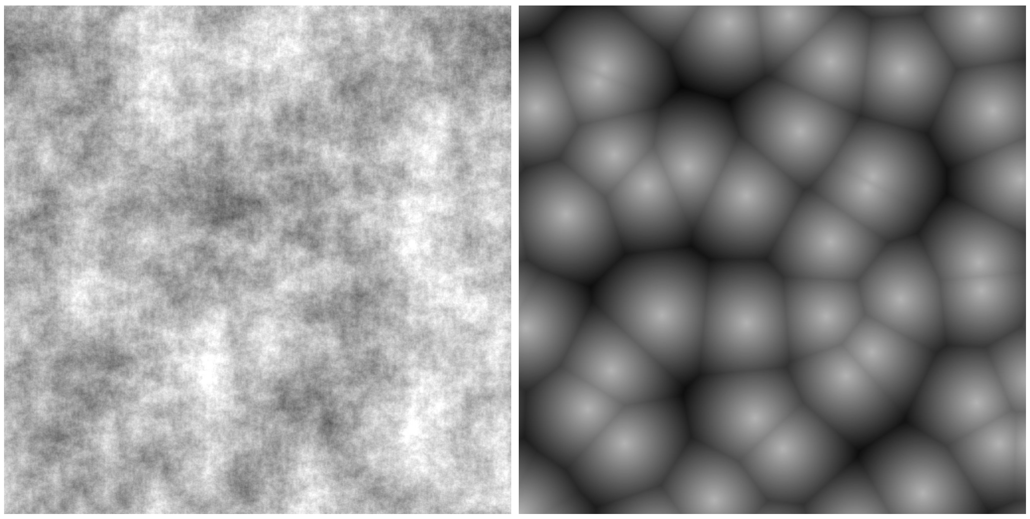


F 5.30 Worley fractals

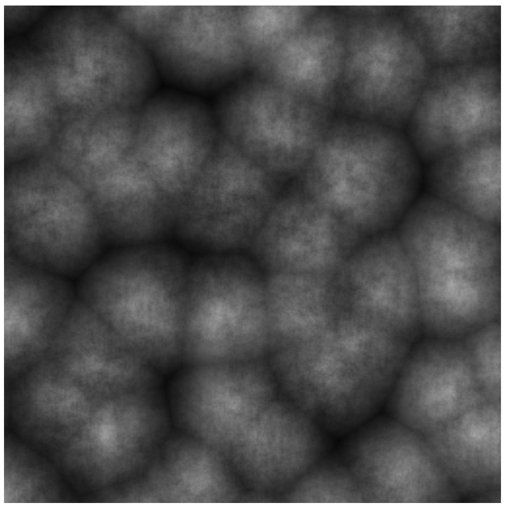
5.4. Blending

Having fractal noises I was able to get detailed noise textures, but working always with the same concept will always give us similar results, and end up repetitive once a couple of textures are generated. To fix that we can concatenate several filters and image operations to modify the look of it.

One easy and powerful way to modify textures is to blend them with other textures, for example take a worley noise and blend it with a perlin noise.



If we multiply each pixel of one image with the pixel in the same position of the other image, detail from one texture will be added to the other while keeping the basic shape, this way we can get even more detailed and varied textures. The result is the following:



F 5.31 Worley noise blended with fractal perlin noises

There are lots of different ways to blend images, the idea is to perform some sort of operation between pixels in the same position of two images.

Photoshop layers work that way, we can change the blending mode of one layer, and it will affect how that layer treats pixels from layers below. In the upper example we are multiplying pixels, but as said we can do much more than that. The most used blending modes for procedural texturing are the following:

- **Copy blend mode**: This will just place the foreground on top of the background. (As default layers in photoshop). It may seem useless, but it can become handy if we play with different opacities or if the foreground has alpha values, less than 1.
- **Addition blend mode**: As the name says it will add pixel values from both images, it will result into brighter images, a good practice is to add some sort of image leveling after blending.
- **Subtraction blend mode**: It subtracts pixel values from both images, values below 0 are clamped to be 0 (black).
- **Multiply blend mode**: As explained in the example before, it multiplies pixel values from both images, allowing to transfer detail from one image to the other. It will darken the image.
- **Add Sub blending mode**: It is a combination of addition and subtraction. Foreground pixel values higher than 0.5 are added to their respective background, and foreground pixel values lower than 0.5 are subtracted instead.
- **Max (Lighten):** It will pick the higher value between the foreground and background pixels.
- **Min (Darken):** As opposite to Max blending mode, it will pick the lower value between foreground and background.
- **Switch**: Having a control parameter (opacity), it will blend both images according it. It is basically a linear interpolation having opacity as the time parameter.
- **Divide**: The opposite to multiply, it will divide the background pixels values with the corresponding foreground pixel.
- **Screen**: Values from each layer are inverted, multiplied, and then inverted again. It gives the opposite effect to multiply, and will always give brighter results than the original images.
- **Overlay**: It combines Multiply and Screen. If the value of the lower layer is below 0.5 then it will multiply, otherwise it will use the screen blend mode.

Depending on the kind of effect we are looking for we will use different blending modes, but this is up to the artist that will be creating the textures.

5.5. Domain Warping

Another useful trick to modify the look of an image is to distort its domain in order to deform its shape. In order to get a more organic and natural looking result, the distortion applied can be taken from another image, normally a noise image.

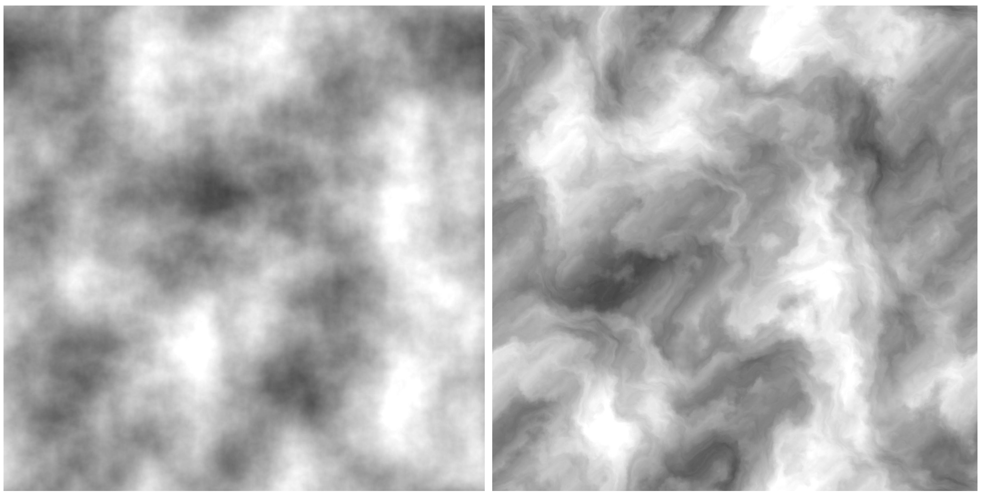
This will only work if we have our image defined as a function of space, which we have, since we are working with noises.

We know we calculate the final color of a pixel with a function f(x,y), or to make it simpler f(p), where p are the pixel coordinates.

Warping means we distort the domain with another function g(p) before evaluating f, so our f(p) becomes f(g(p)). Doing so the distortion would be too extreme, so what we really want to do is to slightly modify the domain, so instead we will add a small warping amount to the identity of the original function, if we multiply the added value with another value (strength) we can also control how much warping amount we want.

Now g(p) becomes (p + h(p)) * *strength*.

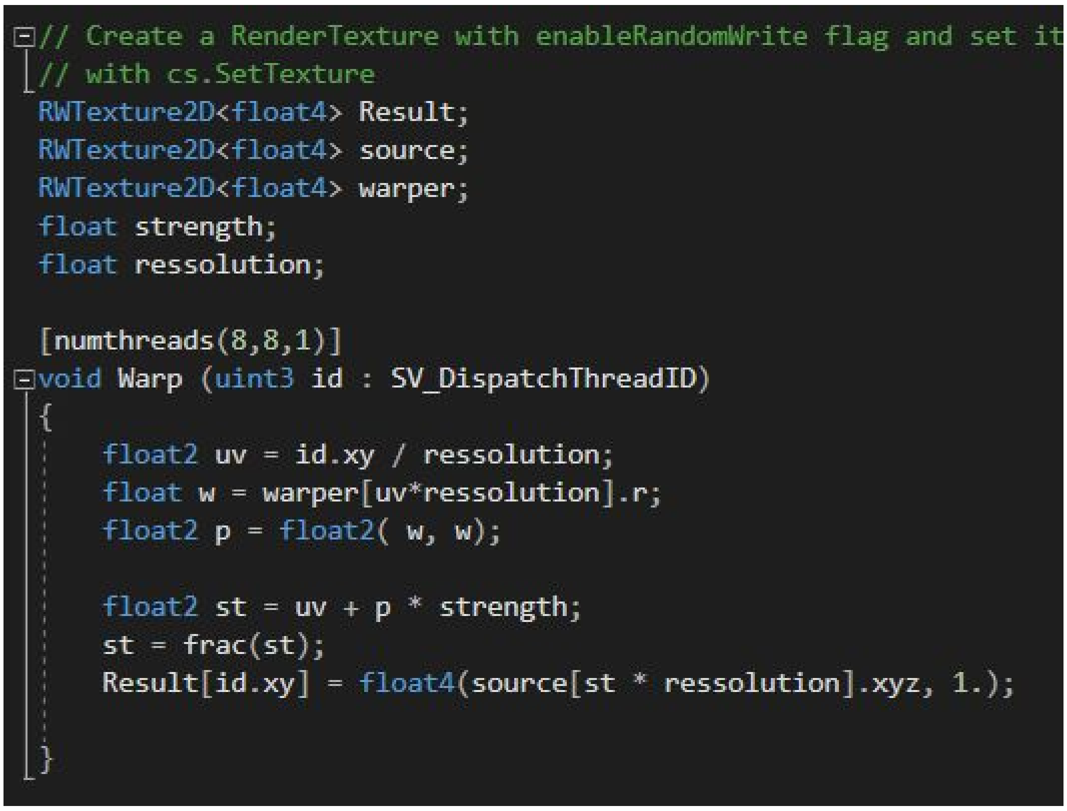
So if we take a fractal noise and warp it according to the values of another fractal image, and a strength of 0.25 the result will be the following.


F 5.32 Domain distortion of a perlin fractal

We are generating static textures, but if we were applying this principle inside a fragment shader in real time we could use time to warp differently at every frame giving a movement look. This shader is a perfect example of the power of warping: https://www.shadertoy.com/view/4s23zz

In fact it is quite easy to implement, and only a small piece of code is needed to warp an image based on another. In the code below we can see we have two textures, the “source” texture which is the image we are going to warp and a “warper” texture from which we will calculate the warping direction. The “Result” texture is the actual texture of the node where we are outputting the pixel values calculated.

First we calculate *w* which is the value that we are going to use as the addition to the domain, we get this from the *warper* image. Then with that value we construct a vector.
With that we can calculate the resulting texture coordinates, adding the warping vector *p* to the original texture coordinates. We use the *frac* function before using the new texture coordinates. *Frac* is an HLSL function that will return the right part of a floating number. Texture coordinates must be between 0 and 1, but when we add *p* to the domain we can surpass 1, taking the fractional part we assure we will be always in the 0 to 1 range.



```
// Create a RenderTexture with enableRandomWrite flag and set it
// with cs.SetTexture
RWTexture2D<float4> Result;
RWTexture2D<float4> source;
RWTexture2D<float4> warper;
float strength;
float ressolution;

[numthreads(8,8,1)]
void Warp (uint3 id : SV_DispatchThreadID)
{
    float2 uv = id.xy / ressolution;
    float w = warper[uv*ressolution].r;
    float2 p = float2( w, w);

    float2 st = uv + p * strength;
    st = frac(st);
    Result[id.xy] = float4(source[st * ressolution].xyz, 1.);

}
```

F 5.33 Warp HLSL code

5.6. Generating Normal Maps

So far I've been explaining how to generate grayscale images, noises mainly, and how to combine and manipulate them, but how are we going to use those textures?

Grayscale images are normally used as heightmaps. Heightmaps have different applications, in the case of procedural textures, they attempt to simulate the "geometry" of the material we are trying to create, in other words, they define the shape of it.

As seen in the *State of the art*, for PBR rendering we need from several textures, so far we have the heightmap that may be used as a displacement map combined with tessellation. Another important map for PBR rendering are *Normal Maps,* they are a cheaper and effective way to fake bumpiness on the mesh surface. Having normal maps and height maps we can use *parallax occlusion mapping* to also simulate depth (this is a bit more expensive).

To generate normal maps we only need from our already created height map, to convert its height information into normal information. In order to do this we can use the *sobel operator*.

The sobel operator is an image filter widely used in image processing for edge detection. At each point in the image, the result of the Sobel operator is either the corresponding gradient vector or the norm of this vector.

This operator uses two 3x3 matrices that calculates approximations of the derivatives, for horizontal and vertical changes. Being A our source image, we can apply the two kernels.

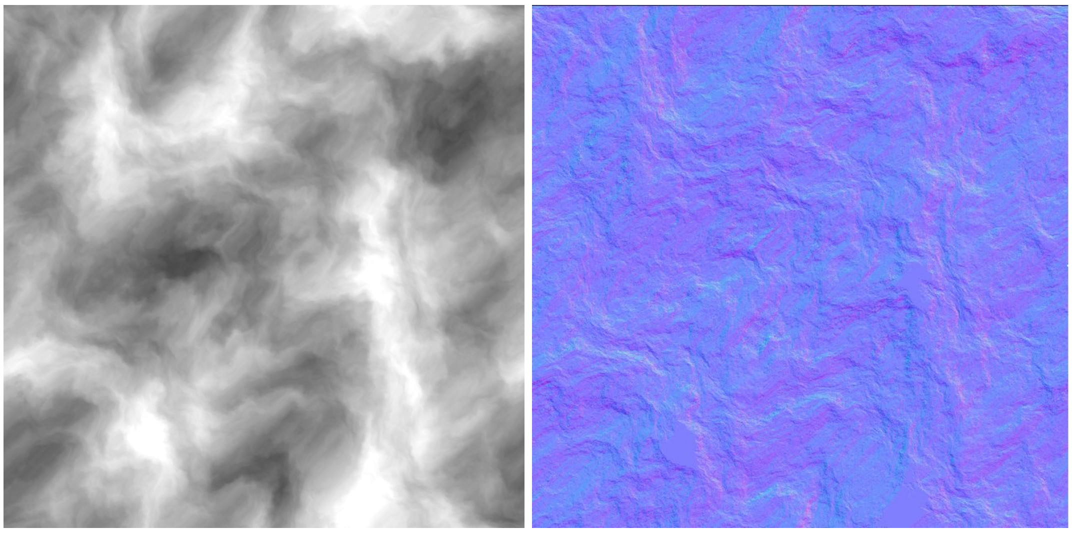
$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

Having G_x and G_y as horizontal and vertical changes, we can now combine both to get changes in all directions together as follows:

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$

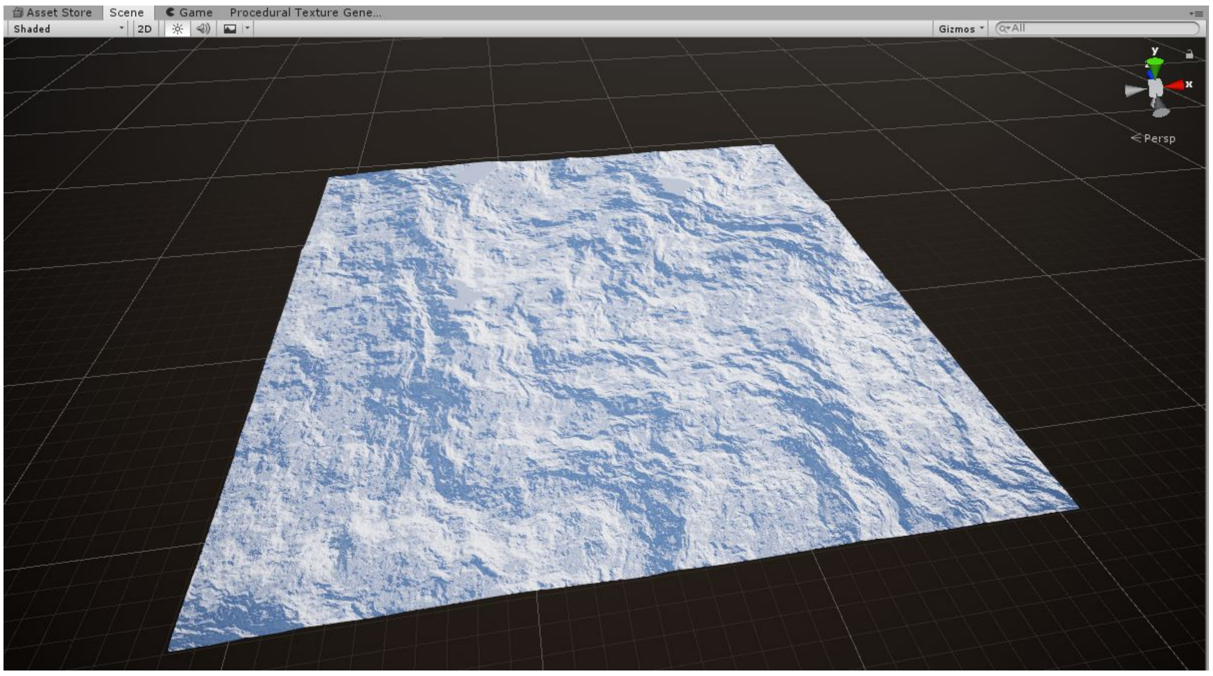
Having this we can get an image with emphasis on the edges. But this is not what we want. (Although it can become useful for other purposes). What we want is to generate a normal map. To do so we can use the grayscale pixel value of G_x as the red channel of our final color, and the grayscale pixel value of G_y as the green channel value. To determine the strength of the normal map we can use the blue channel, having a *strength* value, that may be defined by the user, we can use it to calculate the blue channel value, in this case, by dividing 1/*strength,* this way if the strength is 1 it will have full strength, and if it is less it will decrease the amount of bumpiness.

If we take the previously warped fractal, and we apply this technique we get the following result:



F 5.34 Converting Height information into normal information

If we now use both maps inside Unity within a material and apply that material to a plane it will look as follows:

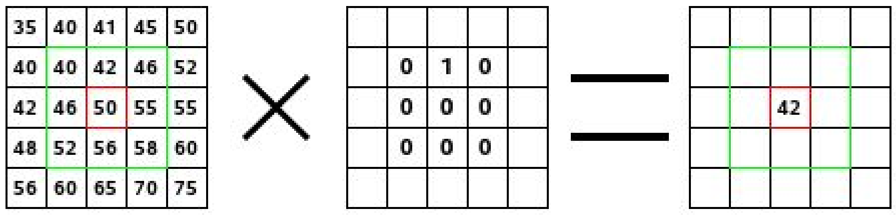


F 5.35 Normal map applied into a Unity material

5.7. Blur

When generating normal maps if the source height map has too much detail we can get artifacts when applying it to a material. Even though we can play around with the *strength* value and decrease a bit the detail we can end up with too flat materials, at the end what we really want is to decrease a bit the difference in the normal vectors of each pixel, a useful trick is to blur the source height map before converting it to a normal map.

Code-wise applying a blur filter to an image is similar to applying a sobel filter. We have a kernel that we must convolve into an image. This kernel can be whatever we want, but the results will be better depending on the values or the size of this kernel. A very popular blur is Gaussian blur.
Blur filters, as well as so many other filters use a technique called *kernel convolution*. It consists in taking a small grid of numbers (the kernel) and make it travel through the entire image.



35	40	41	45	50
40	40	42	46	52
42	46	50	55	55
48	52	56	58	60
56	60	65	70	75

	0	1	0	
	0	0	0	
	0	0	0	

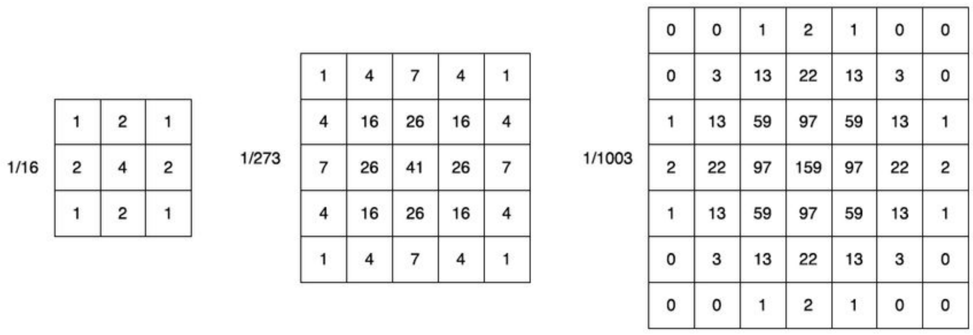
		42		

F 5.36 Kernel Convolution

In the upper example we have an image and a 3x3 kernel, for each pixel in the source image we are applying this 3x3 matrix. The idea is to add the multiplication of all the pixels in the image with the value on the kernel. In the upper case the final output is 42 since the rest of pixels are being discarded by multiplying by 0. For the value at that pixel what we are really doing is:

$$40 \times 0 + 42 \times 1 + 46 \times 0 + 46 \times 0 + 50 \times 0 + 55 \times 0 + 52 \times 0 + 56 \times 0 + 58 \times 0 = 42$$

This is a silly example that won't give meaningful results, but by changing the kernel we can get different results. A Gaussian kernel is made up with numbers that give more importance to those pixels in the middle. Gaussian kernels are the following.



1/16

1	2	1
2	4	2
1	2	1

1/273

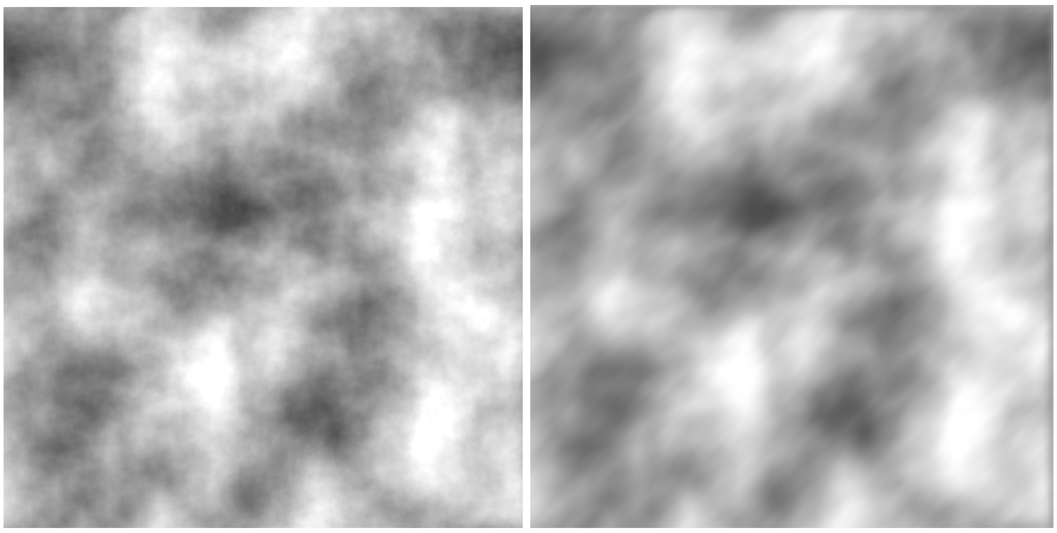
1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

1/1003

0	0	1	2	1	0	0
0	3	13	22	13	3	0
1	13	59	97	59	13	1
2	22	97	159	97	22	2
1	13	59	97	59	13	1
0	3	13	22	13	3	0
0	0	1	2	1	0	0

F 5.37 Gaussian Kernel

By applying this filter into a fractal noise created with this tool we get the following result:



F 5.38 Blurred Fractal

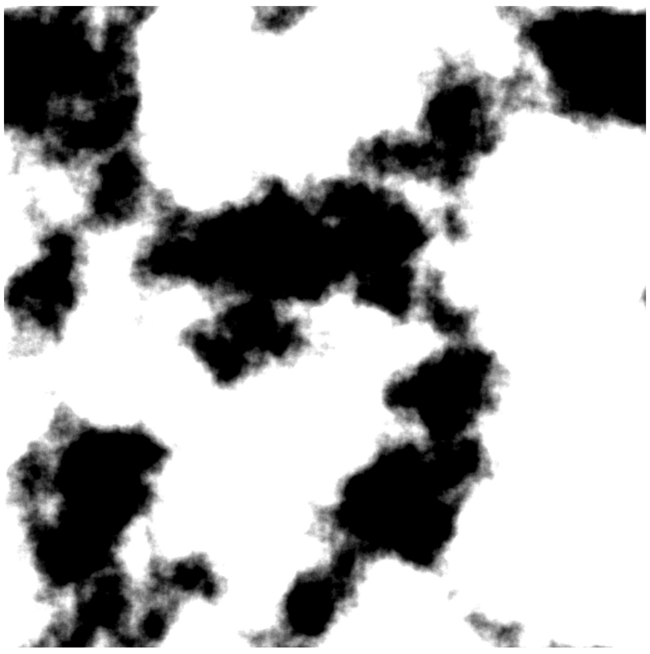
We can observe how the original image has lost some of its detail, it may be useful in so many different case, in procedural texture generation for avoiding artifacts when applying normal maps, but it is also used frequently in post-processing effects for video games, for blur motion, depth of field or after Screen Space Ambient Occlusion(SSAO).

Since this is a rather performance expensive technique and it is meant to be used in real-time application a good way to optimize it is to divide the process into two passes. In the first pass, a one-dimensional kernel is used to blur the image in the horizontal or vertical direction. In the second pass, the same one-dimensional kernel is used to blur in the remaining direction. The resulting effect is the same as convolving with a two-dimensional kernel in a single pass, but requires fewer calculations.

5.8. Image Levels

When working with images, after applying different filters and operations we can lose some of the original brightness and contrasts. Or maybe we don't like the contrast of a resulting noise texture. It is always handy to have a way to modify contrast and brightness, and it is very useful for procedural textures.

Let's say we want to blend two textures based on a mask, the blending algorithm will linearly interpolate between the pixel on one image and the pixel of another image based on the pixel value of the mask image. To do this we want high contrast images to be used as masks, otherwise the final output of the blend would be a strange mix between both sources. (In this case it is not what we want). Normally a mask would look like this:



F 5.39 Mask generated by modifying the levels of a fractal noise

Believe it or not this masks was taken by changing the contrast levels of the previous fractal image. In an hypothetical case where we were blending two textures, black areas will be assigned to the background texture and white areas to the foreground texture.

This kind of masks can also be used as opacity masks, if we were using some kind of transparent material, dark areas will be transparent and white areas will be opaque.

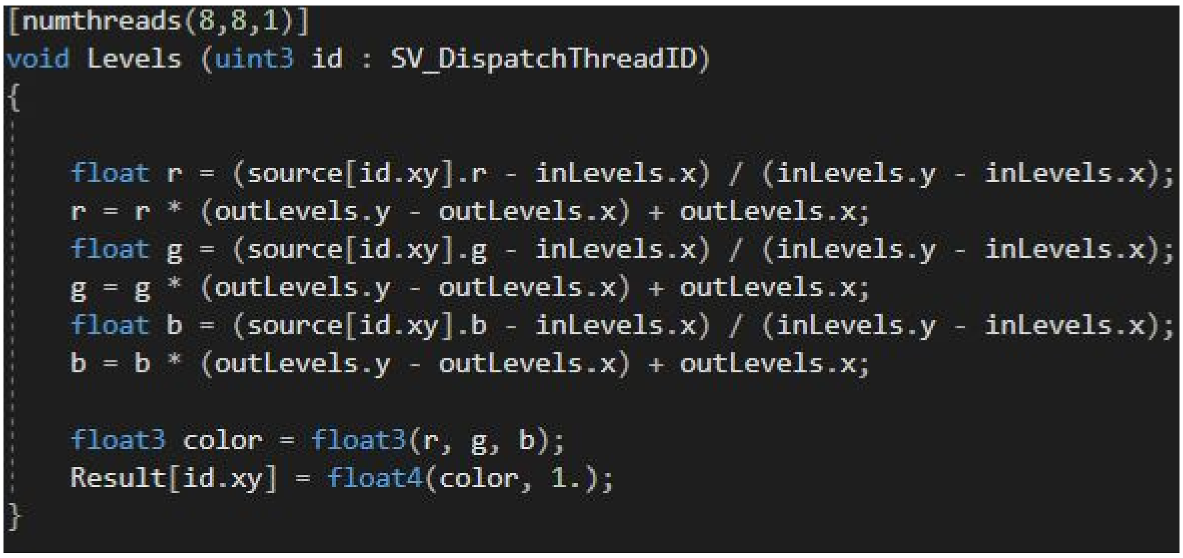
To obtain textures like this we use a technique known as image leveling. It is a tool present in most, if not all, image editing software. They also support gamma correction, in my case I did not implemented it.

With leveling we can move and stretch brightness and contrast having two input levels (level of black and level of white) and two output levels (black and white also).

By changing the values of the input levels and then mapping it to the output levels we redefine the histogram values of that image. For example if we set the black input level to 50 instead of zero we are saying to the image that it will have less grayscale values, hence we will increment the contrast of the image, since it will have less grayscale values to interpolate. The same applies to white values. The less difference between black and white values the more contrasted would be the image.

By changing the output levels we change the brightness of the image, they act like a threshold, if we say that the output black level is 50, the darker value we will get in the image will be 50, and the same applies for white values.

Since the code is quite simple and short, here is the shader code that produces this effect:



```
[numthreads(8,8,1)]
void Levels (uint3 id : SV_DispatchThreadID)
{

    float r = (source[id.xy].r - inLevels.x) / (inLevels.y - inLevels.x);
    r = r * (outLevels.y - outLevels.x) + outLevels.x;
    float g = (source[id.xy].g - inLevels.x) / (inLevels.y - inLevels.x);
    g = g * (outLevels.y - outLevels.x) + outLevels.x;
    float b = (source[id.xy].b - inLevels.x) / (inLevels.y - inLevels.x);
    b = b * (outLevels.y - outLevels.x) + outLevels.x;

    float3 color = float3(r, g, b);
    Result[id.xy] = float4(color, 1.);
}
```

F 5.40 Image leveling HLSL code

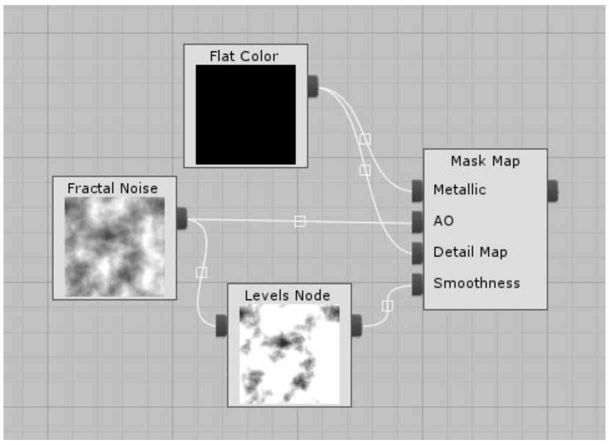
As we can see we are simply mapping the value of each of the texture channels to the input and output levels defined by the user. (*outLevels* and *inLevels* are two *float2* variables that are being sent to the shader before computing it).

5.9. Unity Mask Map

In order to optimize GPU memory usage, some of the built-in Unity shaders for materials requires from a map called *Mask Map*.

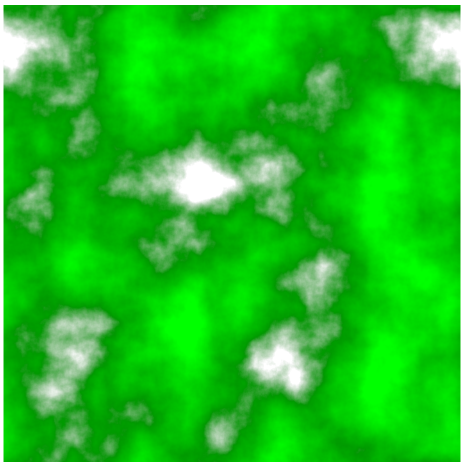
It is basically a map that joins three different grayscale images into one. In this case it joins the metallic, ambient occlusion, detail and smoothness map into a single texture. Since this maps are grayscale maps they only need from a single value to define its color, hence we can assign the value of each map into each channel of the generated texture.

This technique can be used to join any grayscale images we want, and in fact it is used in so many different applications, but since this tool is meant to be for Unity a decided to create a node to generate a *Mask Map* given four different textures. Masks maps in this tool are generated as follow, once we have all the maps ready, we use the Mask Map node to join them all into a single texture.



F 5.41 Mask Map Node

The output image looks as follows:



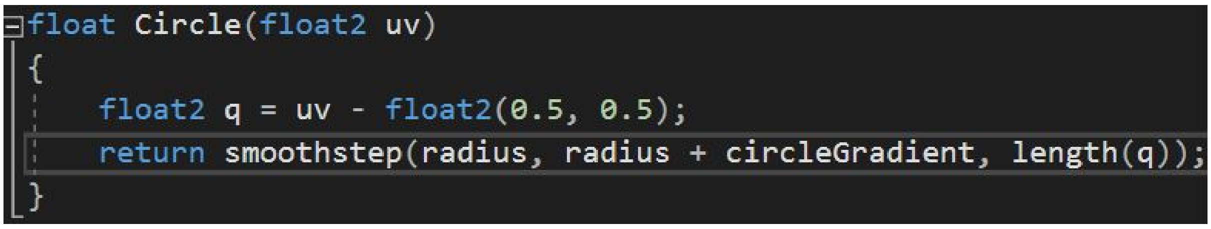
F 5.42 Mask Map

5.10. Procedural Shapes

We've seen noises are really useful to give a texture an organic and natural look, but sometimes can be difficult to give a texture a certain shape by just working with noises.

We can also generate shapes procedurally by using just a few shader functions and a bit of maths. The most common technique when generating shapes is called *Distance Fields.* By computing the distance of a the pixel to a given position we determine its color. The best example to understand this is by generating a circle. The central point would be at the coordinates (0.5, 0.5), the center of the image. Knowing the radius of the sphere and by using the *smoothstep* function, we can determine the final color (black or white).

The smoothstep function performs smooth Hermite interpolation between two values, and it is mainly used when we want smooth transition between both values. We just have to send to the function both edges and the current value.



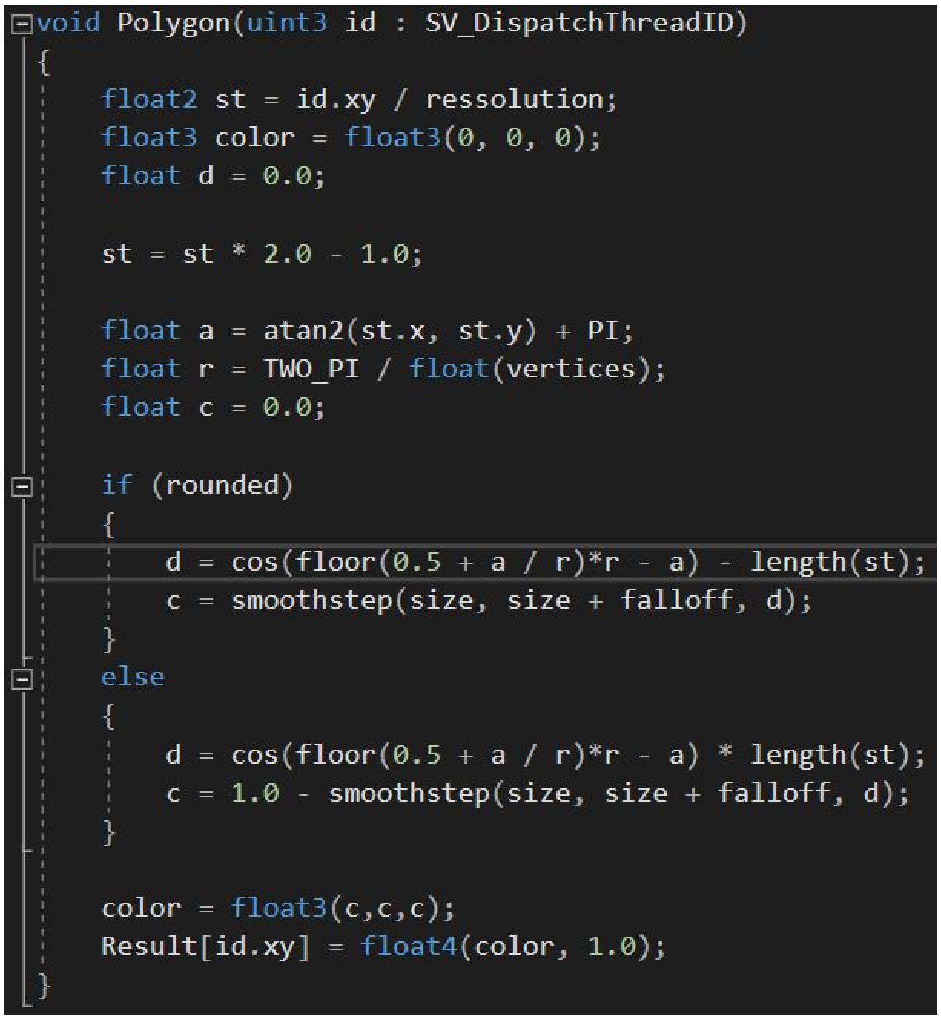
```
float Circle(float2 uv)
{
    float2 q = uv - float2(0.5, 0.5);
    return smoothstep(radius, radius + circleGradient, length(q));
}
```

F 5.43 Procedural circle HLSL code

This simple function applied to every pixel of an image will produce a black sphere at the middle, with the given radius, since the smoothstep function is given us 0 when we are inside this radius and an interpolated value when where are between *radius* and *radius* + *circleGradient*. If we then assign the pixel value as 1.0 - Circle(uv), we get the following result.

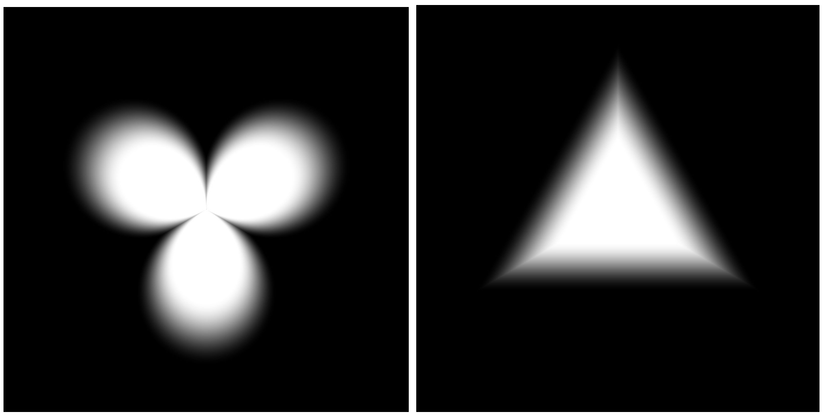
F 5.44. Procedural circle computed with distance fields

By using the arctangent (atan() in HLSL) combined with distance fields we can modify the space to use polar coordinates. This way we can, for instance, define a polygon with a given amount of vertices by constructing the distance field using polar coordinates.



```
void Polygon(uint3 id : SV_DispatchThreadID)
{
    float2 st = id.xy / ressolution;
    float3 color = float3(0, 0, 0);
    float d = 0.0;

    st = st * 2.0 - 1.0;

    float a = atan2(st.x, st.y) + PI;
    float r = TWO_PI / float(vertices);
    float c = 0.0;

    if (rounded)
    {
        d = cos(floor(0.5 + a / r)*r - a) - length(st);
        c = smoothstep(size, size + falloff, d);
    }
    else
    {
        d = cos(floor(0.5 + a / r)*r - a) * length(st);
        c = 1.0 - smoothstep(size, size + falloff, d);
    }

    color = float3(c,c,c);
    Result[id.xy] = float4(color, 1.0);
}
```

F 5.45 SIded polygons HLSL code



F 5.46 sided polygons computed with the code above

5.11. Tileable Patterns

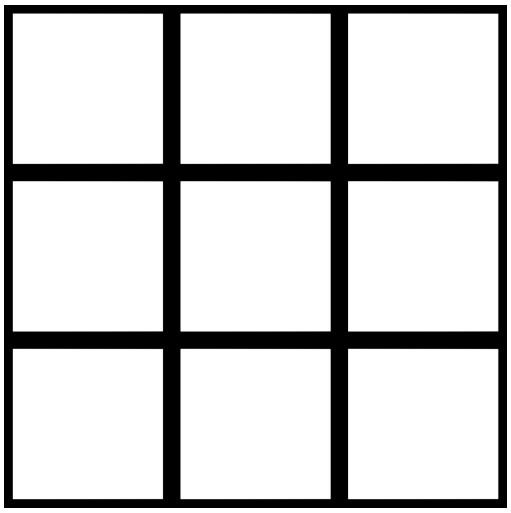
Although one of the initial purposes of procedural textures is to avoid the human-made look of most of hand painted textures, depending on which kind of textures we want to generate, sometimes we need to break that rule. For example a brick texture.

In many elements made by humans we can see repeated patterns and clear geometry. We can procedurally simulate those patterns and add extra detail on top with other procedural noises. Given a definition for any shape in space, we can repeat it as we want throughout the texture.

In order to repeat any pattern the main thing we have to understand is the *frac()* function. It has been pointed before, the *frac()* function return the right part of a floating number. Inside the shader we are working with texture coordinates (UVs), that goes from 0 to 1, what we want is to repeat the pattern inside this space many times, so what we need to do is just multiply it by any number, that will determine the scale. If we multiply the UV coordinates by 3, we will get the pattern repeated in a 3x3 grid.

But this is dangerous since by multiplying by a number bigger than 1 we will get, of course, a number bigger than one, and that is not what we want, since as said we must work with coordinates between 0 and 1. Here is when the *frac()* function becomes handy. Whenever we get to the coordinate 1.1 we know that its value will be the same as the value at 0.1 since it is a repeated pattern, the same applies for 2.1, 3.1, 4.1, 5.1, … All the values at those coordinates will be the same, so by making *frac(uv*) we will always get 0.1, we won't get out of boundaries and we will get the expected result.

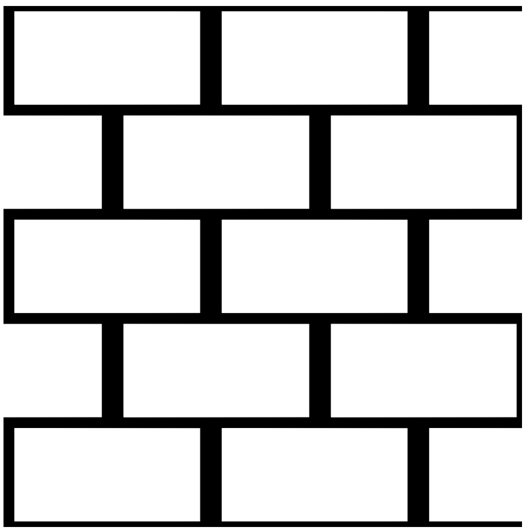
Knowing that and having the function that defines the shape of the box we can repeat the pattern as many times as we want:



F 5.47 Tileable pattern

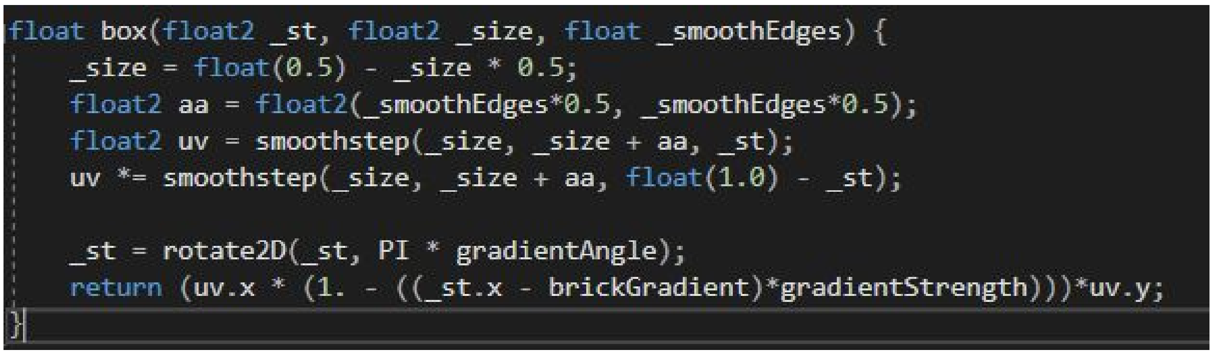
But this doesn't look like a brick wall. To do that we need to modify how the pattern is repeated. It can easily be done with the step() function. The step() interpolation receives two parameters. The first one is the limit or threshold, while the second one is the value we want to check or pass. Any value under the limit will return 0.0 while everything above the limit will return 1.0. Knowing that we can use the result of this function to modify the x position of each tile based on its y position, so we can translate tiles from pair rows.

If we apply this and modify the box shape function to be also larger in the x (so it looks more like a brick) we get the following result.



F 5.48 Brick Pattern

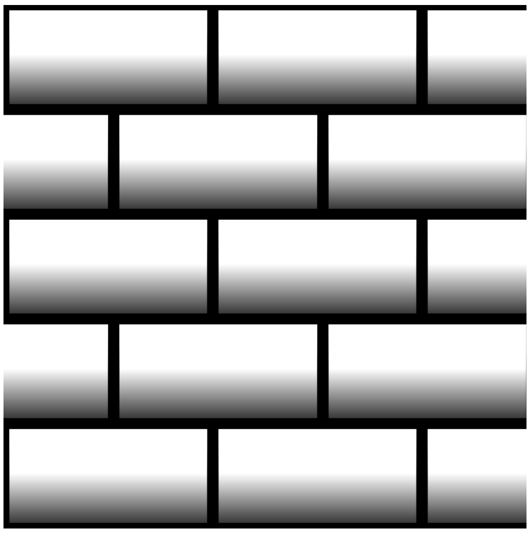
This is a rather flat texture, we can modify the function that defines a simple box and add some extra functionalities, in my case I added the ability to add a gradient for each individual brick as well as apply some rotation to the bricks. The code of the box is based on the example here : https://thebookofshaders.com/09/ the final code looks like this:



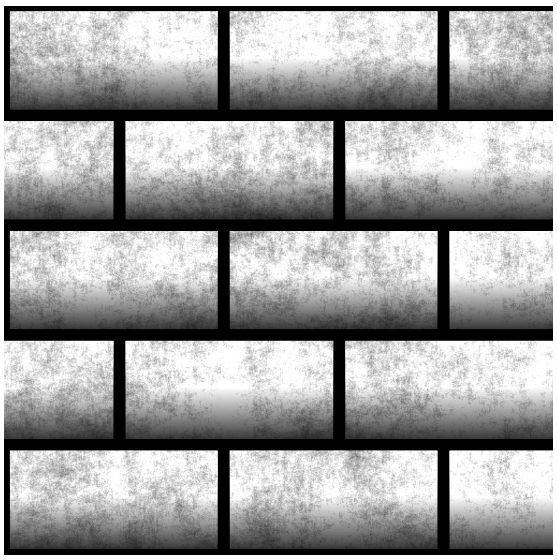
```
float box(float2 _st, float2 _size, float _smoothEdges) {
    _size = float(0.5) - _size * 0.5;
    float2 aa = float2(_smoothEdges*0.5, _smoothEdges*0.5);
    float2 uv = smoothstep(_size, _size + aa, _st);
    uv *= smoothstep(_size, _size + aa, float(1.0) - _st);

    _st = rotate2D(_st, PI * gradientAngle);
    return (uv.x * (1. - ((_st.x - brickGradient)*gradientStrength)))*uv.y;
}
```

F 5.49 Procedural box HLSL code

By using the gradient at each individual tile we can get more interesting results:

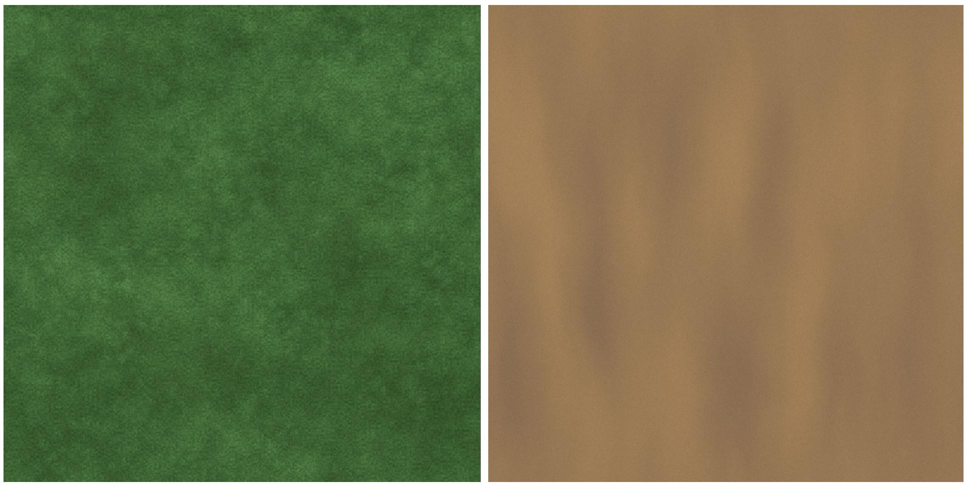

F 5.50 Gradient Brick Pattern

If we want to add even more detail we can blend this texture with a noise fractal, this way we get micro details for each brick, since bricks are not completely flat.


F 5.51 Detailed Brick Pattern

The image above has been obtained by using a multiply blend mode between the previous brick pattern and a large scale fractal noise, with some brightness and contrast adjustments (using the levels node).

This technique as pointed before can be applied to any shape. In the previous chapter I have explained how we can compute procedural shapes, so now we are able to tile them as we want. By tilling and blending a lot of times with different rotations and offsets a circle pattern I've been able to generate micro details for the following grass and sand textures. This technique could be applied to make skin textures as well as any material with micro details.

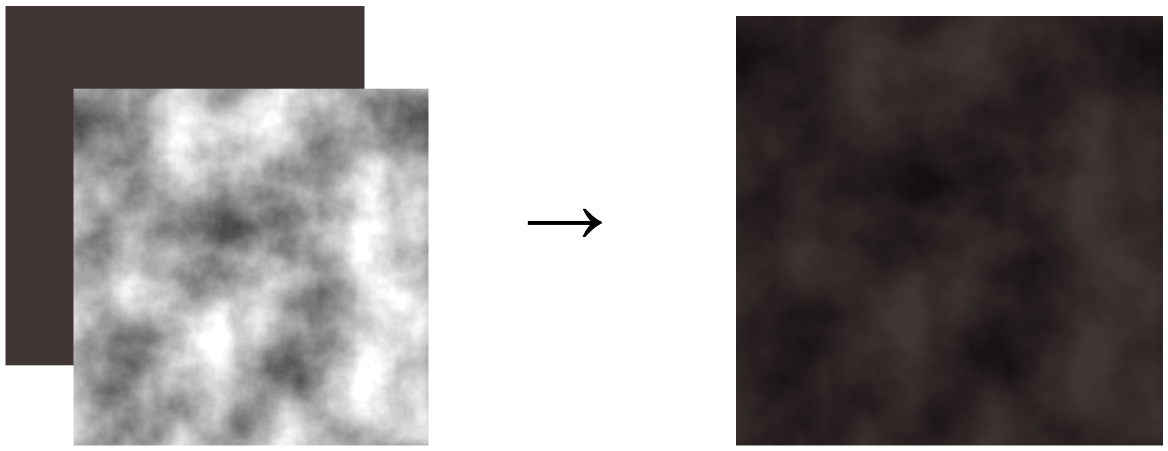


F 5.52 Micro detailed textures using tiled patterns

5.12. Color Gradients

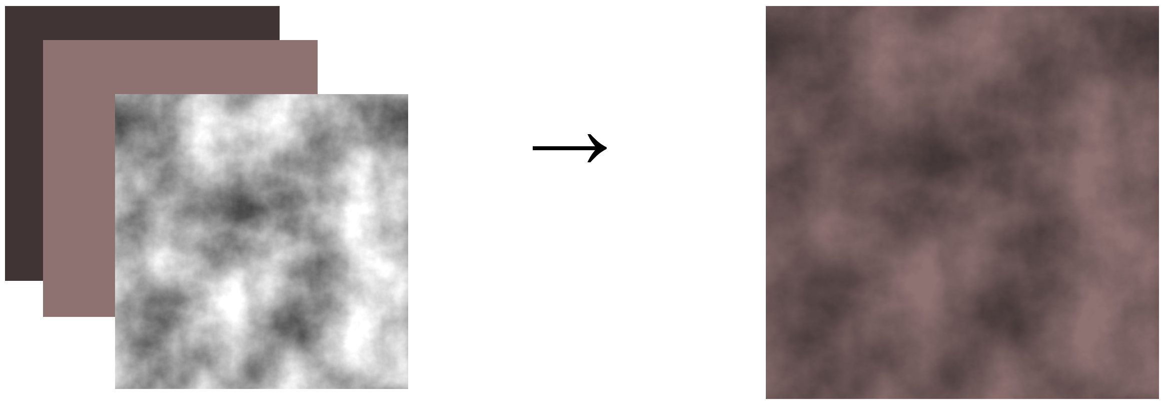
So far I've been only creating grayscale images (besides the normal map). But obviously we need of some color information to generate the diffuse texture for the final material where this texture will be applied.

The simplest way to generate color information is to blend the height map created with some flat color. By doing so we will get the following result.



F 5.53 Color blending

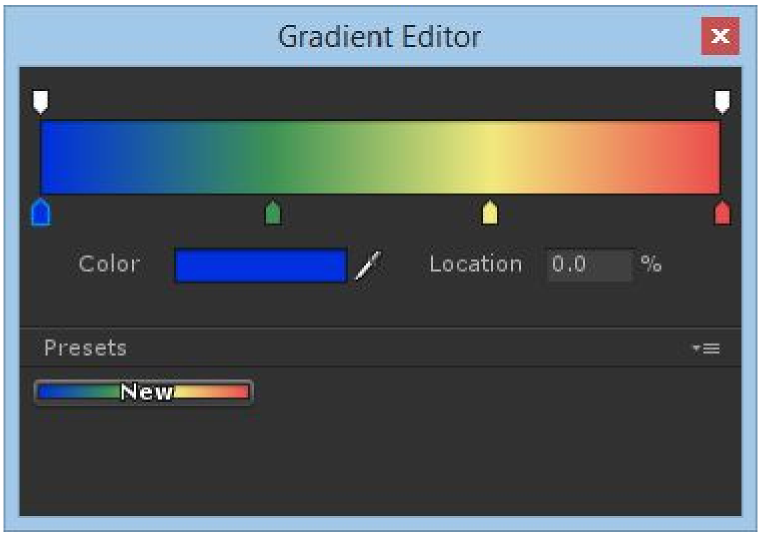
This approach doesn't give us much color information though. Another option would be to linearly interpolate between two colors based on the grayscale value of the height map, by using the HLSL *lerp()* function.



F 5.54 Color interpolation

Interpolating give a better result, but not enough, in order to get richer colors we must use more than one color to interpolate with. As in many other image editing software in procedural texture generation we use color gradients to define which color to pick given a value.

A gradient is just an array of colors, where each color has a key value associated to it (between 0 and 1). To get a color from that gradient we just have to tell him at which point on we need the color, it will look between which colors that value is and linearly interpolate between them. The user must be able to generate key points on the gradient and associate a color to it.



F 5.55 Color Gradient

In the example above if we ask the gradient for the color at 0.5 it will return a interpolation between green and yellow. In this case we will use the heightmap values to evaluate on the color gradient.

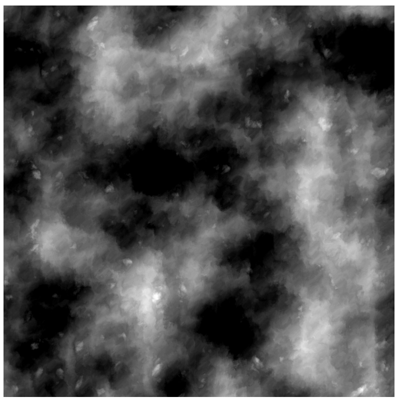
5.13. Generating Unity Materials

So far I have explained how this tool works and how it generates all the textures and nodes, but the final goal of this tool is to generate all the textures needed for a PBR material in Unity. Now I'm going to explain hot to generate the required textures and use them inside of Unity.

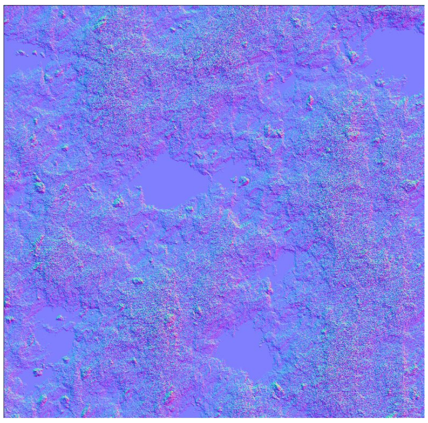
The first thing to do, obviously is to generate the textures we are going to use. I'm going to generate a mud material.

When generating any texture (or even 3D models), the first thing to do is to define the shape it is going to have. To do so with this tool I have everything I need to start.

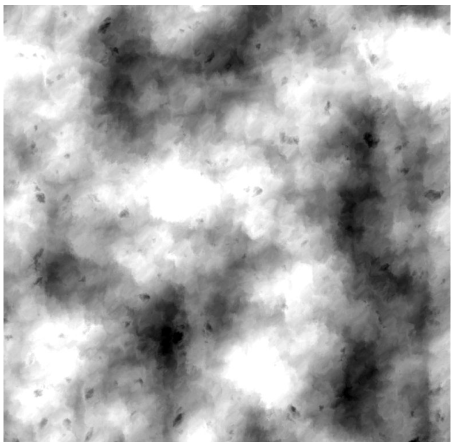
- The first thing I did was to generate a simple fractal that will serve as a base. On top of that fractal, by blending with other noises (tweaking their respective parameters), warping and blurring I defined the shape of the texture. So essentially what I did was to generate the height map.


F 5.56 Mud Height Map

- Once I had the height map I was able to generate the normal map.


F 5.57 Mud Normal Map

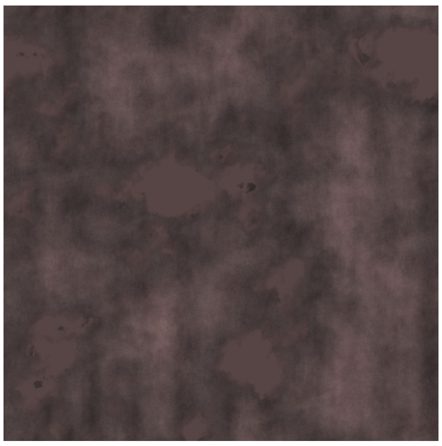
- Since mud is not completely rough I had to generate a smoothness map to define smoother areas (puddles), that will reflect more light. To do so I took the generated heightmap and by leveling and blurring the image I create the smoothness map:



F 5.58 Mud Smoothness Map

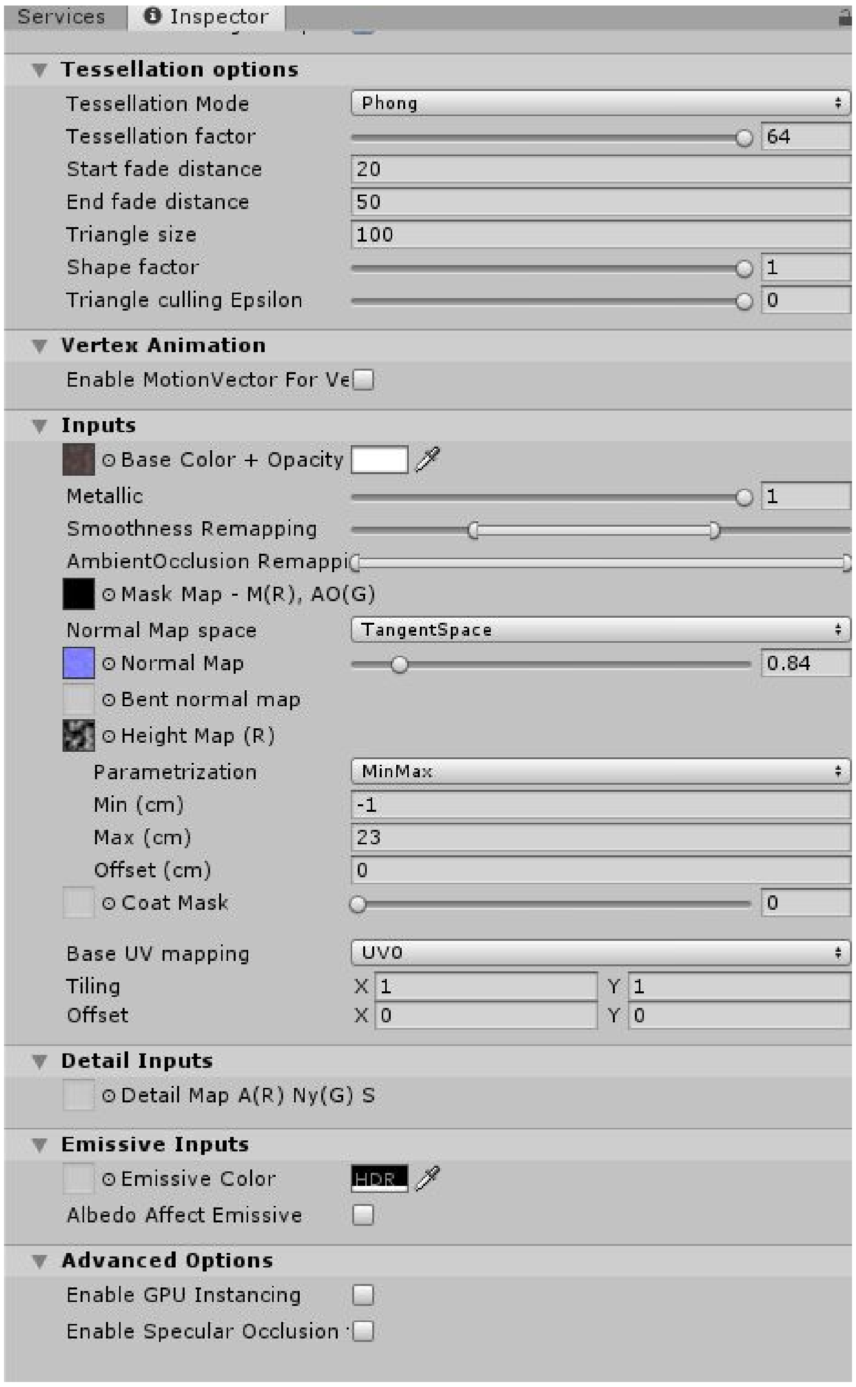
This texture was plugged into the alpha of the *Mask Map* node, the rest of channels were just a black texture since this material has no metalness or detail textures.

- The final step is to add color to it. Mud has not a lot of color richness, only a few brown tones were added to generate a muddy look. By lerping and blending colors onto the heightmap I obtained the diffuse texture:



F 5.59 Mud Diffuse Map

- Then I needed to use those textures inside a material. Unity provides us with some shaders that we can use. They have also recently released a new rendering pipeline, with some graphical improvements and introduced some new shaders. I used the tesseletation PBR shader provided by Unity. This is the inspector for the material inside Unity, as we can see it has so many different options, I only used the tessellation variables and plugged the textures created before.



Tessellation options

Tessellation Mode	Phong
Tessellation factor	64
Start fade distance	20
End fade distance	50
Triangle size	100
Shape factor	1
Triangle culling Epsilon	0

Vertex Animation

Enable MotionVector For Ve	☐

Inputs

Base Color + Opacity	
Metallic	1
Smoothness Remapping	
AmbientOcclusion Remappi	
Mask Map - M(R), AO(G)	
Normal Map space	TangentSpace
Normal Map	0.84
Bent normal map	
Height Map (R)	
Parametrization	MinMax
Min (cm)	-1
Max (cm)	23
Offset (cm)	0
Coat Mask	0
Base UV mapping	UV0
Tiling	X 1 Y 1
Offset	X 0 Y 0

Detail Inputs

Detail Map A(R) Ny(G) S	

Emissive Inputs

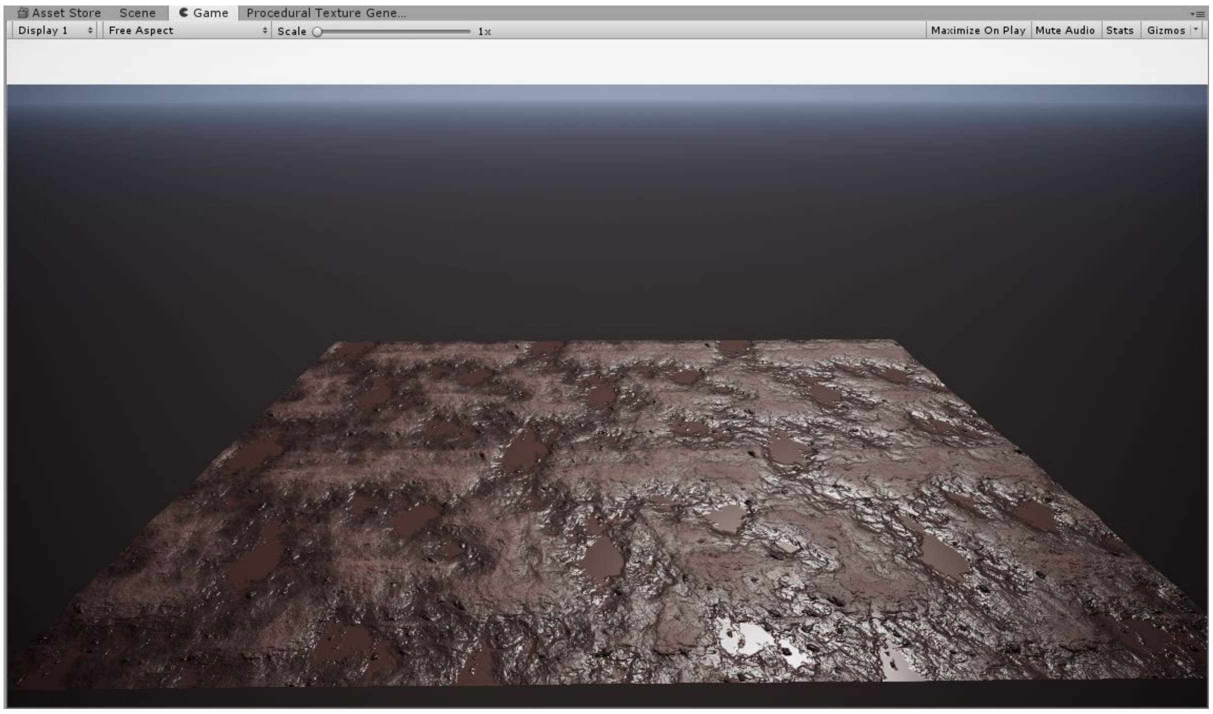
Emissive Color	HDR
Albedo Affect Emissive	☐

Advanced Options

Enable GPU Instancing	☐
Enable Specular Occlusion	☐

F 5.60 Unity Material Inspector

The final result of applying the texture created into the material is this:



F 5.61 Unity Final Material

As it can be seen for the results obtained, we can create high quality materials by generating textures with this tool, I also would like to point out that I am not really good at artistic tasks like this, so I suppose someone with more experience and talent than me will be able to get way better results than this, however, I am pretty satisfied with what I was able to obtain.

5.14. Testing

In order to test the usefulness and usability of this tool I have made several tests with external people. Although, due to time limitations, those tests were not as professional as I would have wanted, they were useful and worth to get documented.

5.14.1. Main Purpose

The main purpose of this tests was not to test the quality of the results that can be obtained, since I was not able to find someone experienced with texture creation, but to test the usability of the tool, to check if someone that would download the tool for the first time will be able to start using it intuitively.

5.14.2. Test Goal

The first thing I did was to give testers a goal, something to do with the tool. I gave them a reference of a material to replicate. I also explained them the basics of the tool (create nodes, connect nodes, save textures, ...). I didn’t care much about the quality of the results, since they were not experts in texturing, but it was useful to see how they were interacting with the tool. The testing session was done online, with shared screen, I was able to see what he was doing at any moment as well as try to answer any question they would have. Although the ideal is not to say anything.

5.14.3. Tests Personal Conclusions

As I was watching how they were interacting with it I concluded that the tool was very intuitive and easy to use once the main features were explained, so I decided that the tool will need a good documentation for future users to check whenever they are lost. They also didn’t know what many of the nodes did, they are common nodes in procedural texturing software, but for someone new they can be confusing, so I decided to display some information or explanation about all the nodes in the editor. They also didn’t know what were all the input points of nodes asking for (in the case they had more than one). This was something that I was aware of, and I was looking for a way to do, without overloading the UI with labels and text. They also complain about zooming, user cannot zoom in and out inside the editor, but this is something common in many tools for Unity, since Unity GUI is not meant to be zoomed, there are different ways to make it possible but I was still working on them at that time.

5.14.4. Tests Feedback

Of course after the test I asked for some feedback, their response was pretty much the same in all the cases. Besides what I already noticed by watching them they pointed several issues that I was not aware of. The suggested that there must be a way to distinguish between node types (generators, filters, ...), which would have made the graph clearer.

Also, when working with textures they were not able to see the final result inside a material until they saved the texture and applied it into a material, they suggested that to be able to link that graph onto a mesh and see the results directly there would save a lot of time.

They also pointed that graphs were not able to be saved and loaded, and if they closed the editor window the graph was lost. This was something that I was planning to do by the end of the project, but they insist on this particular case.

5.14.5. Overall Test Conclusions

Gathering everything they pointed out and what I already noticed I made a list of possible tasks that can be done in a future to fix those issues. If there is no time to include them inside this project I will try to fix them after the project anyway, since I intend to keep working and updating it.

This are the tasks, that I decided that were doable:

- Implement visual distinction between nodes of different types.
- Implement zoom inside the editor window.
- Display node information inside the UI without overloading it.
- Specify what each input point for nodes requires.
- Link the graph created to a mesh to be able to see results directly without having to save and apply textures.
- Allow to change texture resolution. Right now they are all 2048x2048 by default but cannot be changed.

6. Conclusions

The first achievement during this project has been to analyze and understand in depth the characteristics of procedural textures and everything that must be taken into account when implementing them, as well as how to develop a tool for an engine such us Unity, specially a node based editor, now I feel capable of developing a similar interface for any kind of engine or platform.

I've been able to experiment and prove to myself the usefulness of procedural textures (or procedural generation in general). Even though procedural textures are quite a challenge to generate, or at least understand how they are generated, once implemented their purpose becomes clearer. I'm now able to generate connecting a few nodes high quality PBR textures really fast, and with the ability to modify parameters to change the overall look of the texture immediately, this is incredibly powerful since using this approach we can generate, for instance, a library of rock materials by generating a base material and applying modifications to it. Traditionally this would have taken a lot more.

Obviously not everything is perfect and procedural textures have also some drawbacks. Even though applying the algorithms explained in this document we can get pretty good and natural results, it is not as accurate as a taking real world photos and generating photo scanned textures with photogrammetry. But in my opinion the quality difference is not high enough considering the amount of time that photoscanned materials requires.
Also, by using procedural texturing we can generate textures that are impossible to scan (sci-fi, fantasy, ...).

One of the goals was to implement this tool inside of Unity, since most external applications require the application to be installed, as well as importing/exporting. I've tried to generate textures using Substance Designer in comparison with my tool to use within Unity and I have realized that it is way more comfortable to have the interface and the results within the Unity editor, I can directly save any texture at any moment when generating textures and also, what I consider very important, that I'm able to see directly how the material looks inside of Unity, because external applications such as Substance Designer have their own renderer which is highly optimized for rendering their textures, and of course considering they only have to render a model with the textures created applied, they can afford more costly lighting computations and other graphical techniques like tessellation or parallax occlusion, which can also be used in Unity, but are barely used for games. When I tried to import the textures created in substance I noticed that they looked worse than I was expecting (although they still look really good, considering Substance Designer is one of the standards for texturing in the industry). But by using my tool I had no doubt that my textures were going to look as I expected since I was already visualizing them in Unity.

Since this tool is meant to be used for real Unity users I've had to approach it in such a way that is usable and friendly, I realized that node based editors are extremely useful and I had not to put too much effort to make it usable since by nature node based interfaces are intuitive and easy to use for us, since we are used to think and work this way, not only in software applications.

Even though performance is not critical for a tool like this, since it is not going to be used during gameplay, I realized image processing is quite a tough topic regarding speed, and since I was not using compiled languages like C or C++ which are known for their capabilities of performance I had to find another way. I knew GPUs were excellent for parallel computations but never realized about their potential until I used them together with compute shaders. I used them for computing textures, but they are capable of so much more, we can compute anything we want inside a compute shader, obviously they have some limitations, but I the change of performance I experimented when switching to compute shaders was massive, and probably one of the best choices I could have made.

7. Future Work

To be honest I think in the future I will redo this tool from scratch. As explained in this document I encountered some issues during development that were provoked by a bad planning and architecture of the tool. (Although thanks to this I was able to get more nodes and techniques implemented). So I would say this project has been a way for me to learn a lot about procedural texturing and how not to plan a tool like this. Although I'm happy with the overall outcome of this project, now I feel more capable of developing a tool like this.

I would do several things differently:

- The first and most important would be to take serialization into account from the very beginning. It is a tough and, at least for me, tedious work but it must be done. My mistake was to start to code everything and worry about saving and loading after that. Again, I think it was positive for me since I was able to learn so much more about procedural texturing, but now that I know all the techniques involved, I would start with serialization.
- Develop nodes in a more modular and dynamic fashion, in such a way that allows the user to create new nodes. When implementing some of the shaders I realized that most of the nodes had some operations in common, and the final output of a node was the concatenation of different simple operations (that were common among nodes). Having those operations defined as nodes itself, final users would have way more possibilities to generate their textures. This way we could define a node as a subgraph inside the final graph. It is a complex concept, but this would suppose a huge improvement and give total freedom to the final users.
- I would still use compute shaders to compute everything since it is the fastest way to do it, but it would be nice to give the user the chance to choose between CPU and GPU considering right now users with older GPUs are not able to use this tool.

- Finally I would polish the GUI look, since due to time constrictions this has never been part of this project and a good looking GUI is always more appealing to users.

All this changes of course will require more time, that the time that I have had to develop this project, but thanks to it I’ve been able to learn a lot about this subject and I feel positive about how this tool will end up being after applying everything I’ve learnt during the development, the how to and how not to do things.

8. Bibliography

PBR Theory Part 1 [Online]. Web Page, URL
<https://academy.allegorithmic.com/courses/the-pbr-guide-part-1> [Consult March 2nd 2019]

PBR Theory Part 2 [Online]. Web Page, URL
<https://academy.allegorithmic.com/courses/the-pbr-guide-part-2> [Consult March 2nd 2019]

PCG Book [Online]. Web Page URL
<http://pcgbook.com/> [Consult March 4th 2019]

Computer Graphics [Online]. Blog, URL
<http://iquilezles.org/www/index.htm> [Consult March 5th 2019]

Texture filtering [Online].
<http://emulation.gametechwiki.com/index.php/Texture_filtering> [Frequently Consulted]

Substance Designer Tutorial [Online]. Video, URL
<https://www.youtube.com/watch?v=i_q_JaCg7hk&list=PLB0wXHrWAmCwWfVVurGIQO_tMVWCFhnqE> [Consult March 6th 2019]

Agile Methodologies [Online]. Web Page, URL
<http://agilemodeling.com/essays/fdd.htm> [Consult March 10th 2019]

Procedural Textures [Online]. Blog, URL
<http://www.upvector.com/?section=Tutorials&subsection=Intro%20to%20Procedural%20Textures> [Consult April 10th 2019]

Perlin Noise [Online]. Blog, URL
<https://flafla2.github.io/2014/08/09/perlinnoise.html>[Consult April 9th]

Worley Noise [Online]. Web Page, URL
<https://www.gamedev.net/forums/topic/576541-voronoi--worley-noise-knowlege-dump/>
[Consulted April 10th 2019]

Seamless Texturing [Online]. Web Page, URL
<https://www.gamedev.net/forums/topic/688992-making-textures-on-a-cube-seamless/>
[Consulted April 15th 2019]

Unity Documentation[Online]. Web Page, URL
<https://docs.unity3d.com/ScriptReference/GUI.html.>[Frequently Consulted]

The Book of Shaders [Online]. Web Page, URL <https://thebookofshaders.com/00/>[Frequently consulted]

David S. Ebert (ed.) (2003). *Texturing & Modeling. A procedural Approach.* 3rd ed. Morgan Kaufmann Publishers. [Frequently consulted]

Matt Pharr. and Wenzel Jakob (ed.)(2010). *Physically Based Rendering: From Theory to Implementation.* 2nd ed. Morgan Kaufmann Publishers. [Consult April 23rd 2019]