Catlike Coding

NumberFlow Documentation

Getting Started

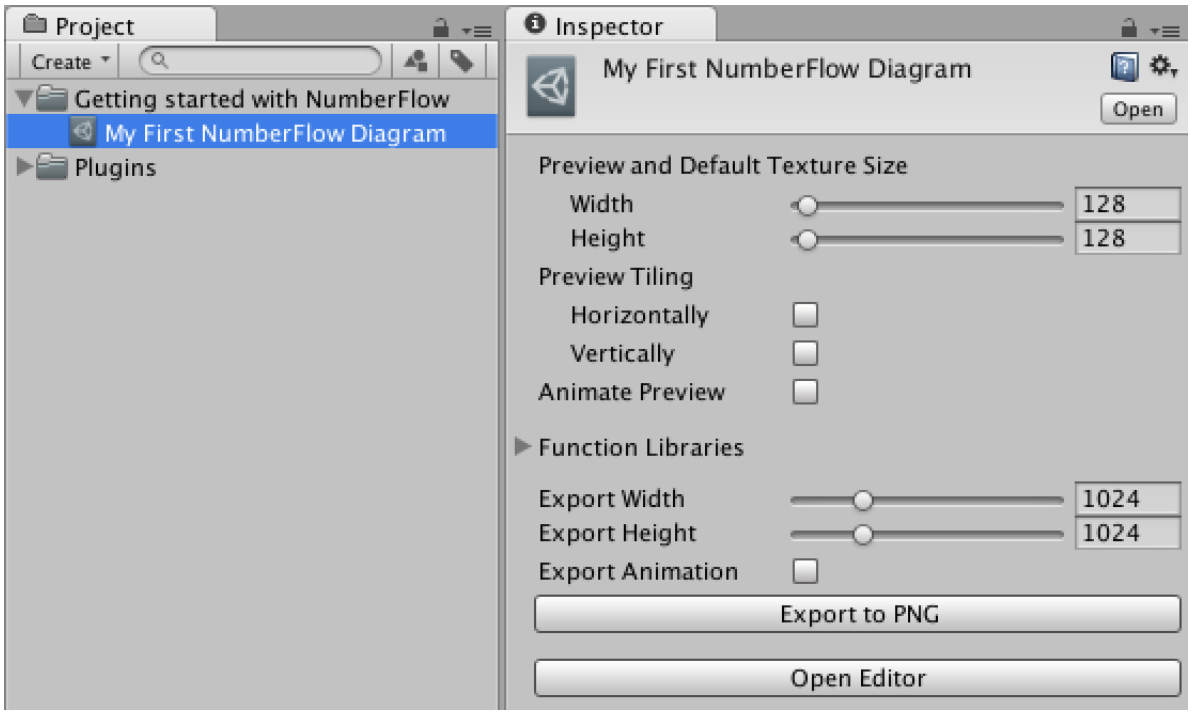
NumberFlow is a visual editor extension to Unity that allows you to create your own procedural textures.

You design texture diagrams in Unity and use a manager object to link them to materials in your scenes. You can also export to PNG images or generate textures and animations at runtime with scripts.

The store package also includes the Noise and Utilities libraries, which you can use in your own projects as well.

1 Creating Your First Diagram

After importing the NumberFlow package, you can create a new diagram asset via *Assets / Create / NumberFlow Diagram* in the menu, or via *Create / NumberFlow Diagram* in the project view.

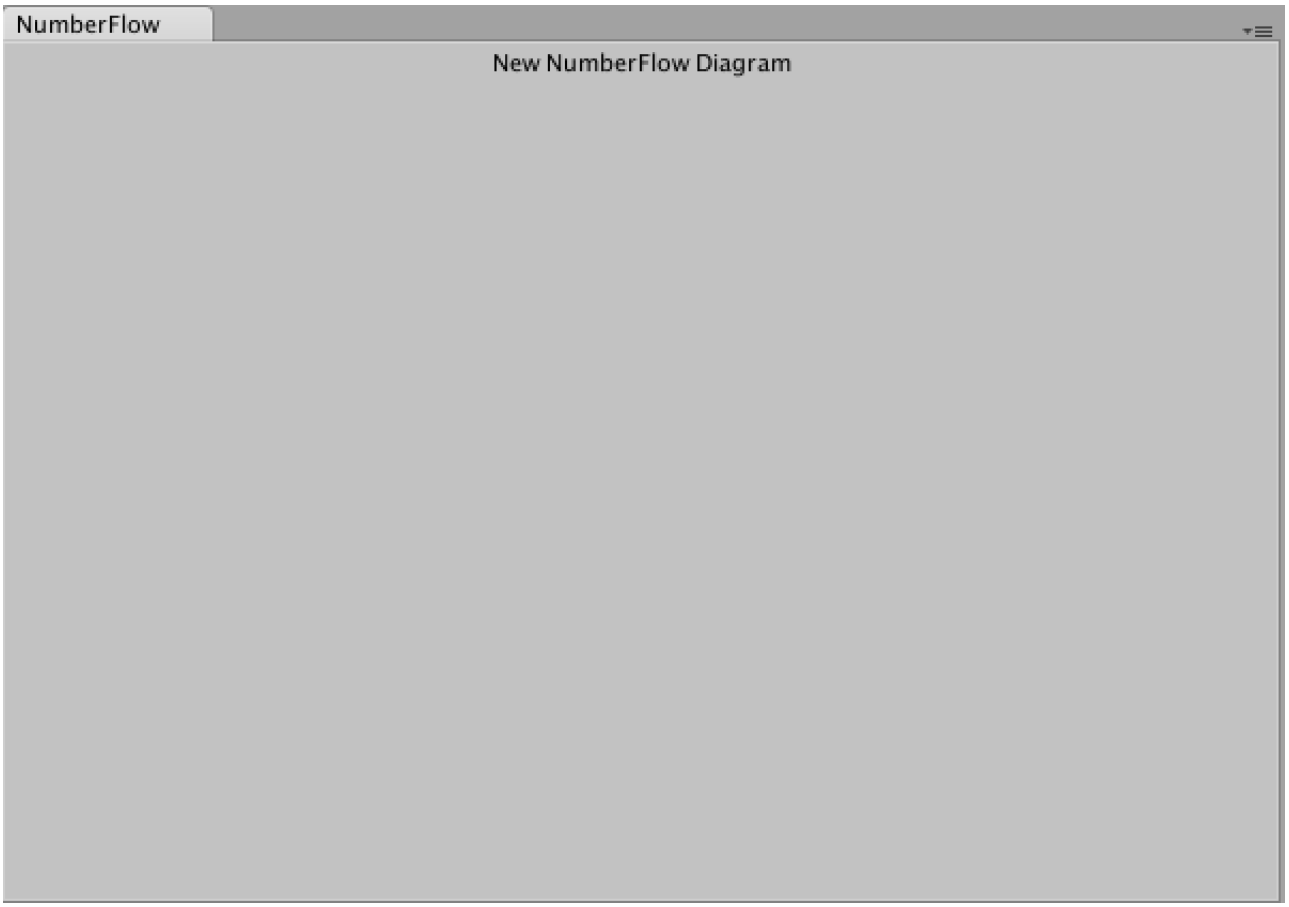


A new diagram.

1.1 Editing the Diagram

The inspector of the diagram contains various configuration options for previewing and exporting, but to edit the diagram you have to use the diagram editor window. You can open this window via *Window / NumberFlow* in the menu or by pressing the *Open Editor* button. You can leave it floating or dock it somewhere.

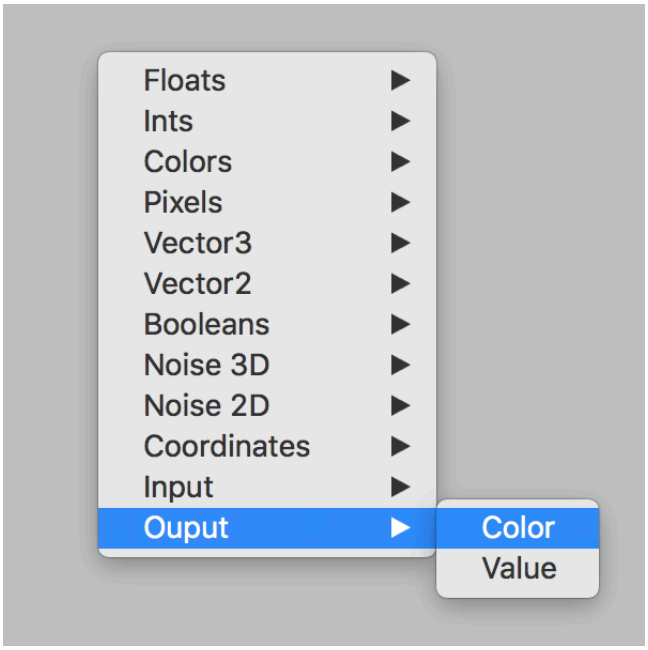
The editor window will display the last selected NumberFlow diagram, even if you currently have something else selected.



Editor window showing an empty diagram.

1.2 Adding Nodes

A new diagram is completely empty and does not generate any output. You can add nodes to it by right-clicking anywhere inside the editor window and selecting a node type from the context menu. A color output node can be added via *Output / Color*.



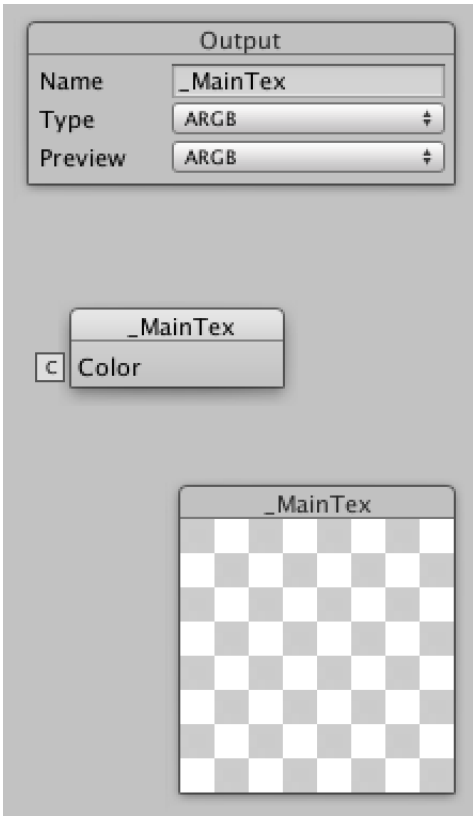
Adding an output node.

1.3 Configuring the Output

After adding the output node it will be automatically selected. You will now see an additional panel at the top right of the window, which contains information and configuration options for the selected node.

Our first output node doesn't have a name, so we set it to *_MainTex*. This matches the name of the shader parameter used for the main texture of Unity's default shaders, which will come in handy later.

A preview panel has also shown up at the bottom right of the window, but it doesn't show anything useful yet.

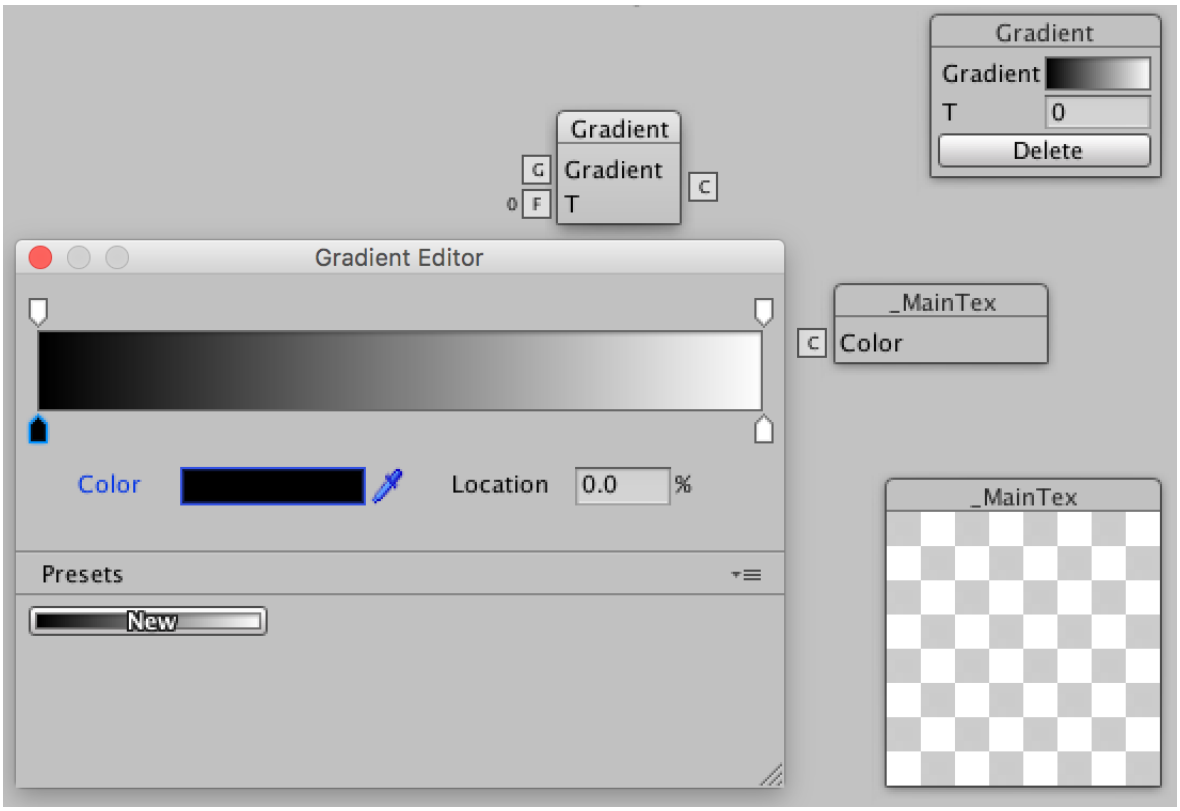


An empty main texture.

1.4 Adding a Gradient

You can add a gradient node via *Colors / Gradient* in the context menu. This node has two properties. The first is a gradient named *Gradient*, which defaults to solid white. The second is an interpolator value named *T*, which defaults to zero.

You can change the node's values via the panel at the top right. When clicking the gradient, a gradient editor will pop up. You can use it to change the gradient to whatever you wish, like setting its first color to black.

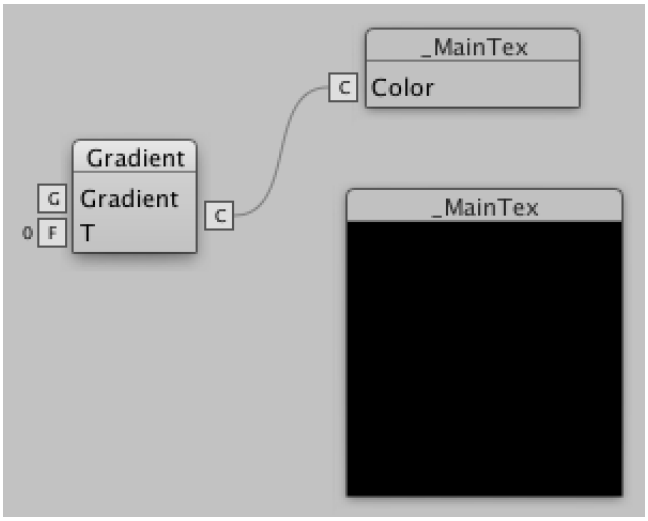


A grayscale gradient.

1.5 Connecting Nodes

To link the output to the gradient, drag from the output node's "C" square to the gradient node's "C" square. You can also drag to anywhere inside the gradient node, you need not exactly hit its "C" square.

The small squares are connectors. Connectors on the left of a node belong to its properties, while the connector on the right of a node represents its output. Only connectors of the same type can be linked.

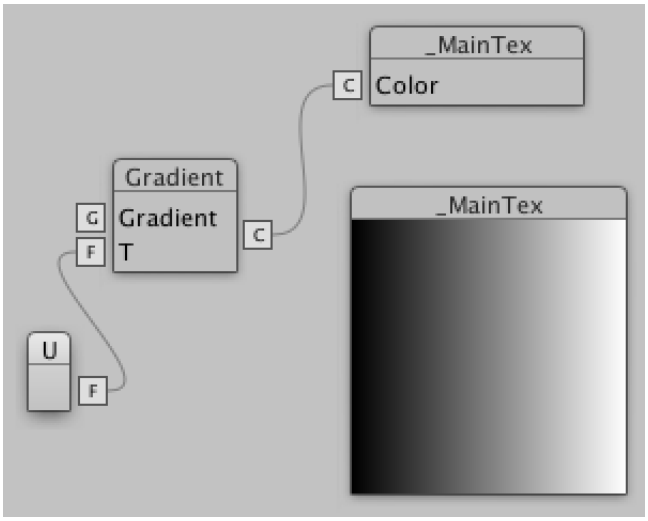


Connected nodes.

Once the nodes are connected, the preview will show whatever color comes first in the gradient. You can adjust this by changing the *T* parameter of the Gradient node. Zero corresponds with the left of the gradient, while one corresponds to its right side.

1.6 Using Coordinates

Instead of changing the gradient's *T* by hand, you can link it to another node that produces a float value. You can use the image's horizontal texture coordinate for this. You can add its node via *Coordinates / U*, and then you can connect the Gradient's *T* property to it.

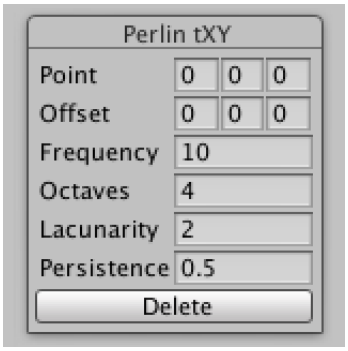


Using U to create a gradient.

1.7 Adding Noise

Noise nodes are useful for creating intricate pseudorandom patterns. You can create a tiling 3D Perlin noise pattern with the node found under *Noise 3D / Perlin / Tiled XY*. This version of the noise tiles in two dimensions, which is useful when you want to create tiling textures.

Noise nodes have a lot of properties, which are explained in the Noise documentation. For this example you can leave *Offset* untouched, set *Frequency* to 10, *Octaves* to 4, *Lacunarity* to 2, and *Persistence* to 0.5.

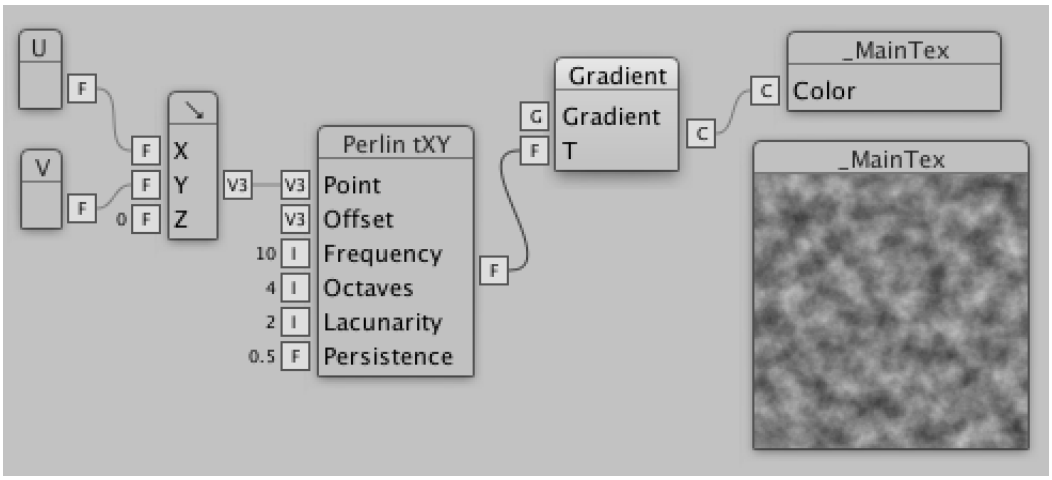


Noise parameters.

1.8 Sampling Noise

The 3D noise nodes need a Vector3 sample point. You can create one via *Vector3 / From Floats* and then connect its *X* and *Y* properties to the U and V coordinate. You can add a V coordinate node via *Coordinates / V*. The vector's *Z* property can remain at zero, or you can set it to a different value, which will produce a different result.

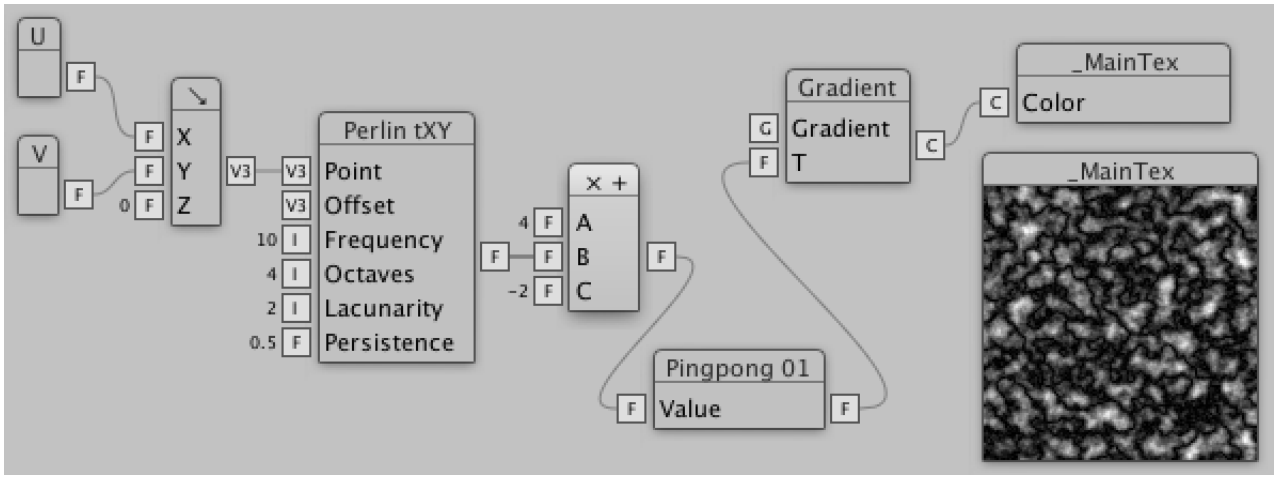
To finally see the noise, connect the gradient node to the noise node.



Feeding noise into a gradient.

1.9 Adjusting Noise Output

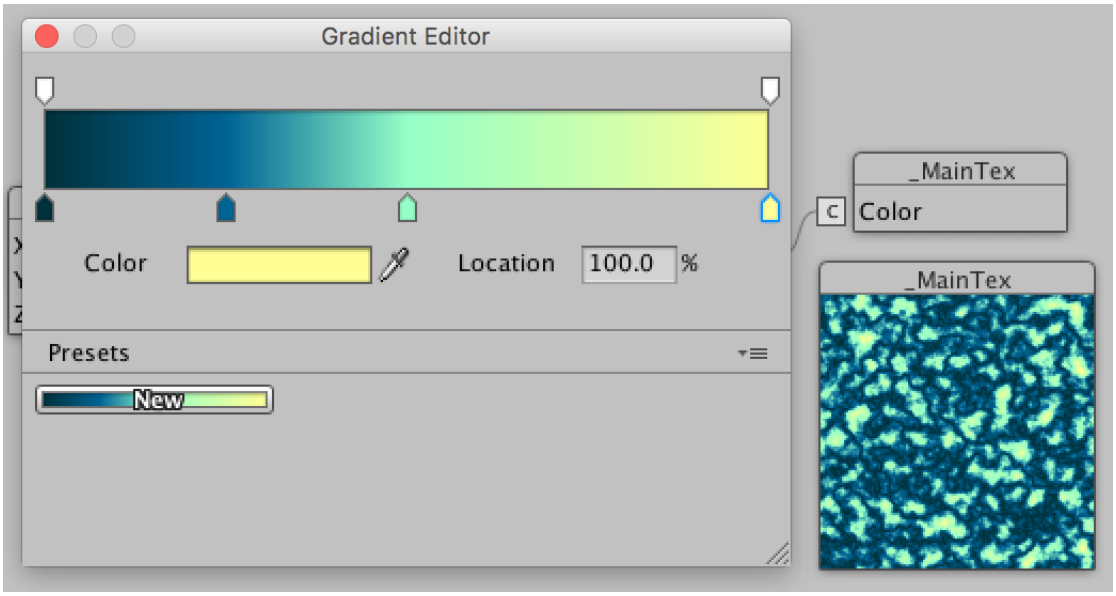
While you can directly use the noise samples, you can create different patterns by manipulating its output. For example, use a *Floats / × + Multiply Add* node to change the noise range from 0–1 to −2–2. Then feed that into a *Floats / Range / Pingpong 01* node.



Adjusted noise.

1.10 Coloring the Pattern

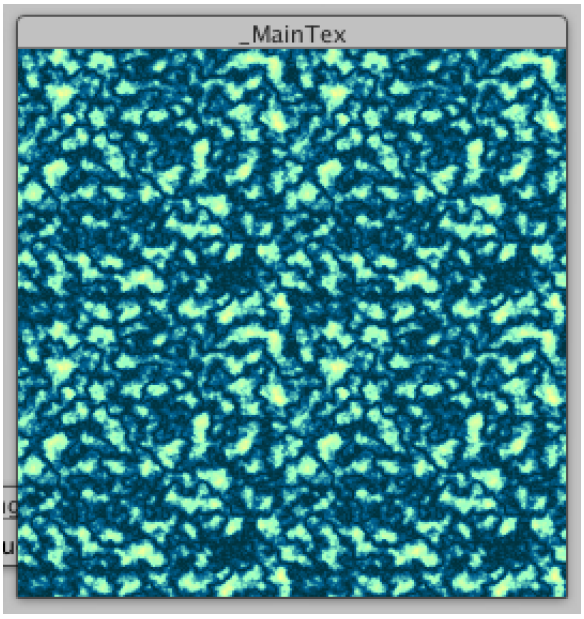
Instead of sticking to the monochrome pattern, you can adjust the gradient for a more colorful result. For example, you can set the colors to (0,50,60) at 0%, (0,100,150) at 25%, (150,255,200) at 50%, and (255,255,150) at 100%.



Colored noise.

1.11 Preview Tiling

You have now created a nice tiling pattern. You can verify that it tiles by checking both options under *Preview Tiling* in the inspector.



Preview tiling.

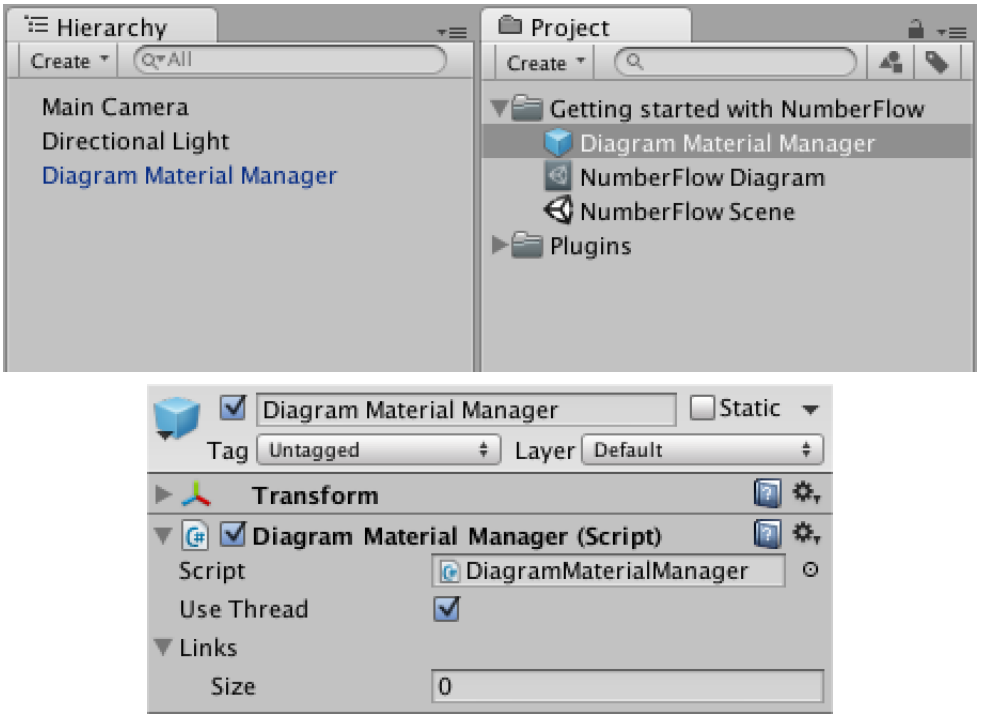
2 Using a Diagram Material Manager

How do you use your diagram to texture materials in your scenes? You could export to PNG files, or use scripting to fill textures manually. But the most convenient way is to use a diagram material manager.

A diagram material manager is a component that allows you to link diagrams and materials together. It will take care of generating the textures for you, both in play mode and in edit mode.

2.1 Creating a Prefab

Start with a new scene. The best way to work with a manager is through a prefab. Create a new empty game object via *GameObject / Create Empty*. Name it *Diagram Material Manager* and drag it into the project view to turn it into a prefab. Then select the prefab and add the manager component to it via *Components / Scripts / CatlikeCoding.NumberFlow / Diagram Material Manager*.

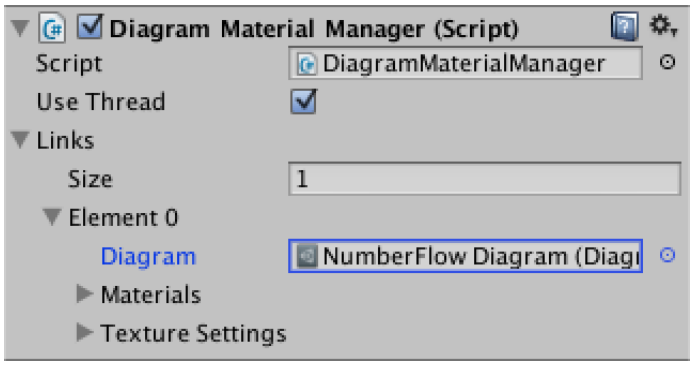


Empty diagam material manager.

Note that the inspector of the manager is quite basic. This is because it is still an experimental feature.

2.2 Adding a Link

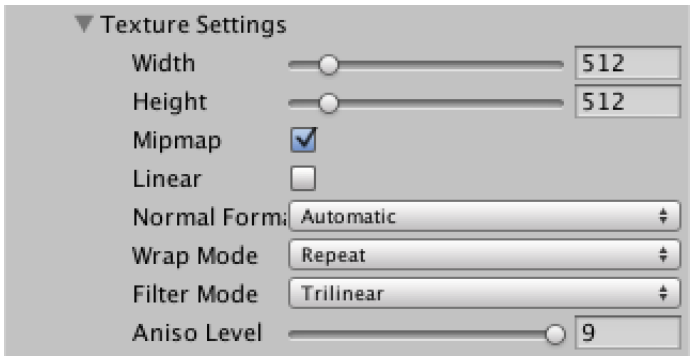
The manager prefab contains a list of links between diagrams and materials. Set its size to one and expand the link that was added, then make its *Diagram* property reference your diagram.



Adding a link.

2.3 Configuring the Texture Settings

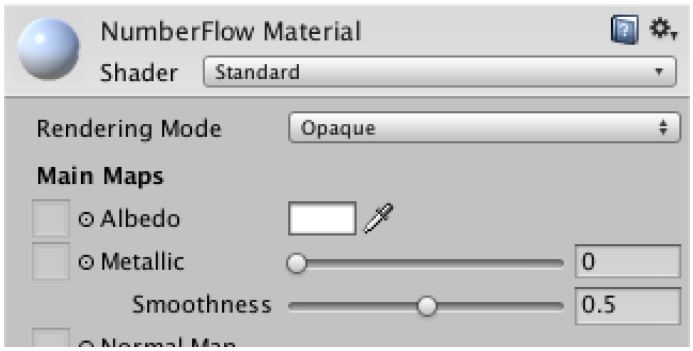
The texture settings are used for all textures generated by a diagram. For a nice texture you can set *Width* and *Height* to 512, enable *Mipmap*, set *Filter Mode* to *Trilinear*, and set *Aniso Level* to 9.



Texture settings.

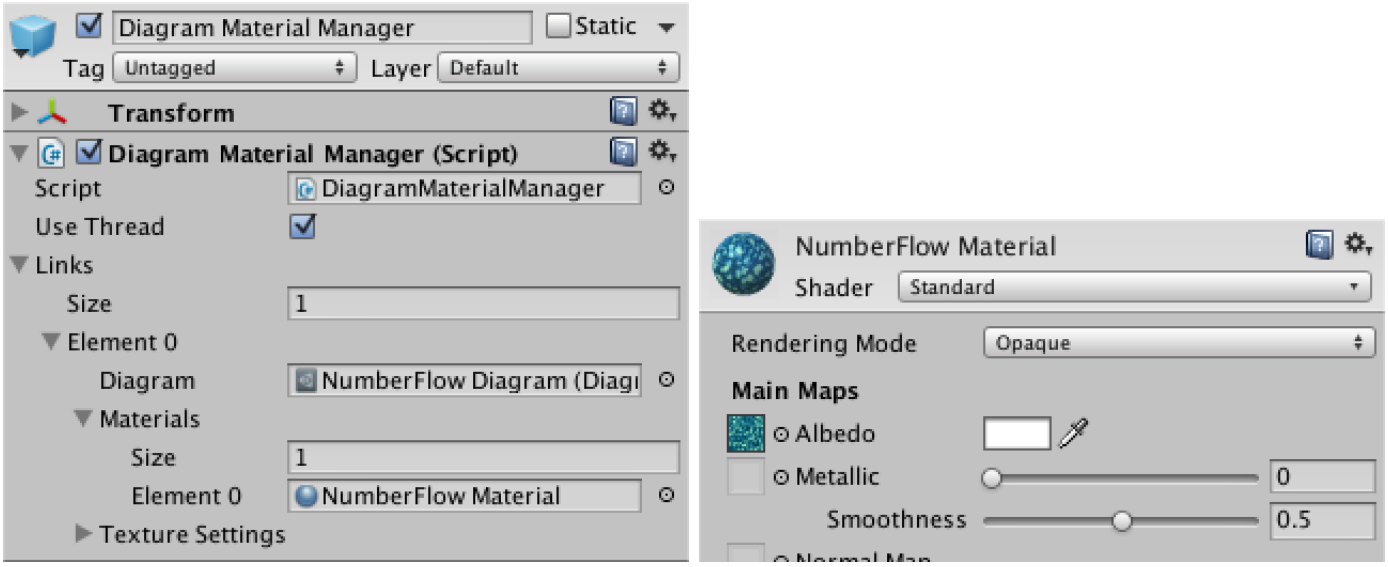
2.4 Linking a Material

You need a material, so create one via *Assets / Create / Material*. The default material has the Standard physically-based shader, which is fine.



Default material.

You can now link your new material to the diagram by dragging it onto the Materials list of the link. After doing so, you will see that the material has gained a texture.

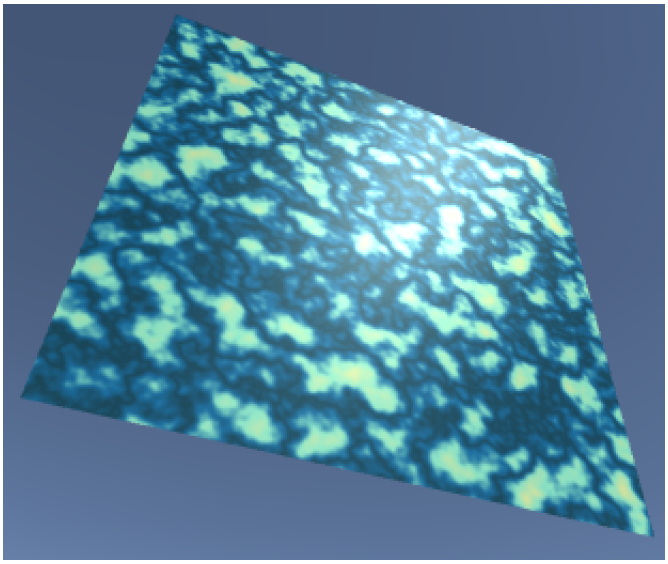


Linked material with a texture.

Note that the texture is not stored on the file system, it only exists in memory. If you were to save the scene, you might see it disappear for a moment, after which it returns. This is because the textures are automatically generated and destroyed at various moments by the manager instance. Deactivating the manager instance will also destroy the textures, while enabling it will generate them again. Any change to the prefab will also cause the textures to be created anew, which is convenient when editing the manager.

2.5 Using the Material

You can use your material in your scene like normal. For example, put a quad at the origin via *GameObject / Create Other / Quad* and drag your material onto it.



Quad with material.

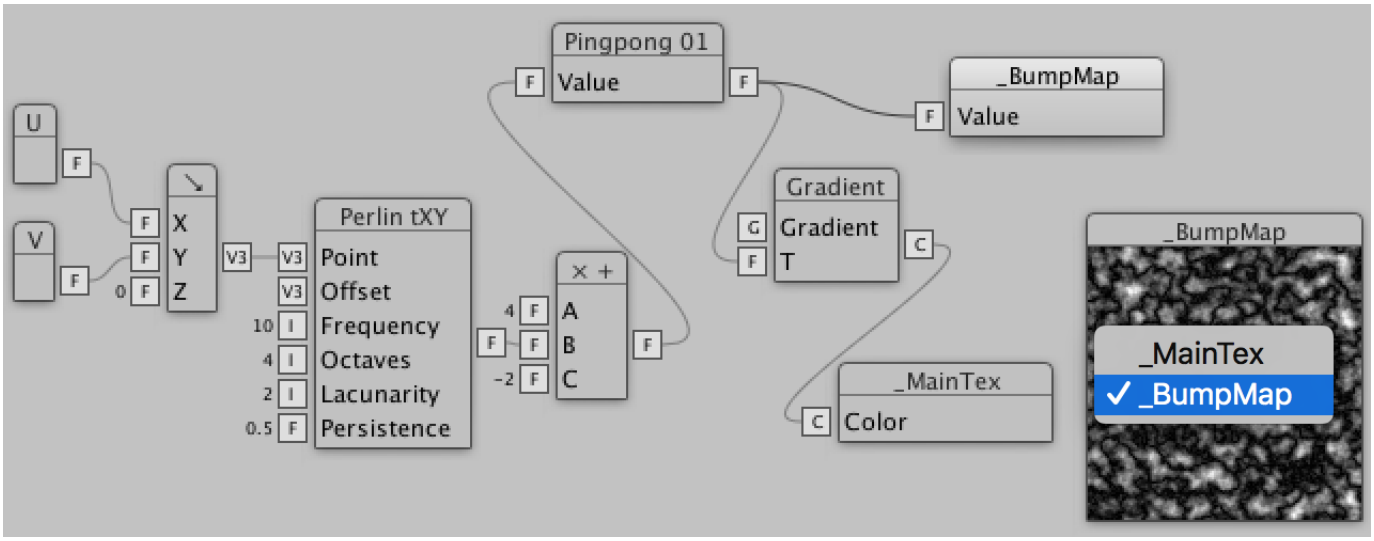
When entering play mode, you will notice that the texture will temporarily become white. This is because the textures that existed in edit mode are destroyed when play mode is entered. The material will fall back to its default white texture. The texture will reappear once the manager instance has generated it while in play mode. The same happens when exiting play mode.

The default mode of the manager is to use a separate thread to generate its textures. This means that your application will not stall, but the textures won't be immediately ready. You can change this by disabling *Use Thread*, which will make your application freeze until the manager has generated all its textures.

2.6 Adding Bumps

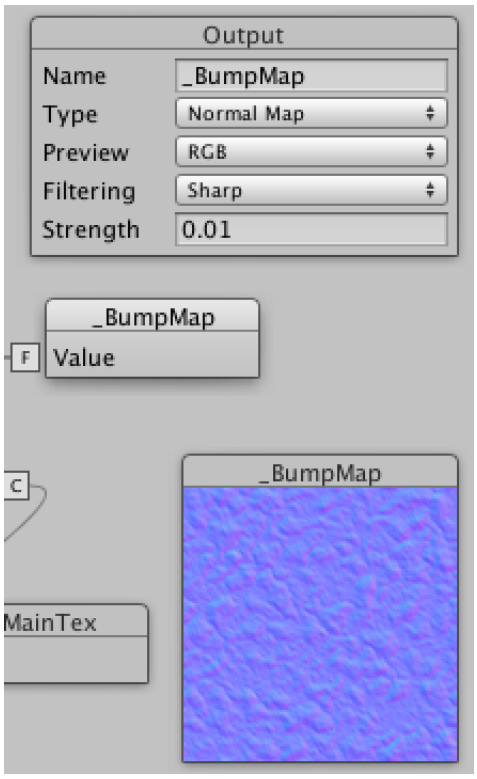
The material currently appears perfectly flat. You can change that by including a normal map. Add a second output to your diagram. This time you need only a single value instead of an entire color, so create one via *Ouput / Value* and connect it to the *Pingpong 01* node.

The new output will start with the same settings as the previous one. Rename it to *_BumpMap*, which is the name that the shader uses for its normal map. Also set its Preview mode to RGB. Then switch the preview to the new output, by right-clicking the preview window and selecting *_BumpMap*.



A second output node.

Change the *Type* of your new output to *NormalMap*. This will change the preview to a flat normal map. Increase its *Strength* and you will see the bumps appear. A value of 0.01 produces a subtle result in this case.



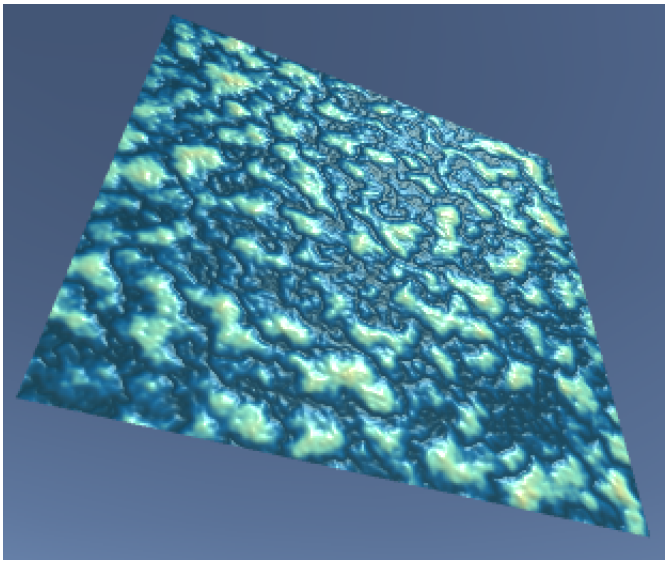
Normal map output.

By the way, as the main texture only uses the RGB channels, you can save memory by changing its ouput node's type to RGB.

2.7 Updating the Material

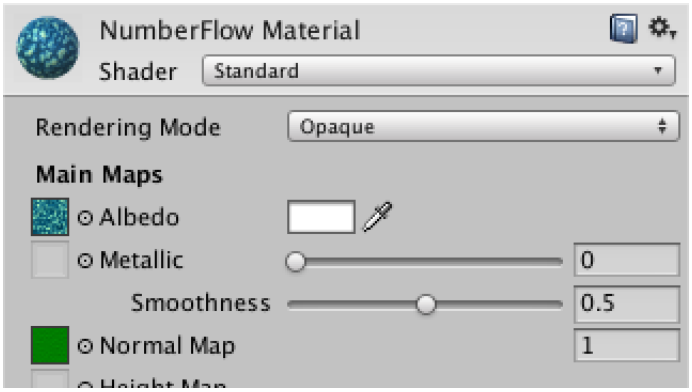
Changes to diagrams are not immediately picked up by the diagram material manager. The easiest way to update the manager without toggling play mode is to select the manager instance and hit the *Revert* button. You can lock an inspector window so you don't have to reselect the manager instance all the time.

For complex shaders such as the Standard one, it might also be necessary to manually select the material after adding or removing a texture. This is necessary when the shader's inspector sets shader keywords based on its parameters.



Quad with bumps.

You might have noticed that the normal map looks green in the material's inspector. This is because Unity internally uses the DXT5nm format for normal maps, except for some platform builds. Unity hides this from you for your imported normal maps, but the generated textures show their true form. You can manually set which normal format to use, or leave it up to the manager.

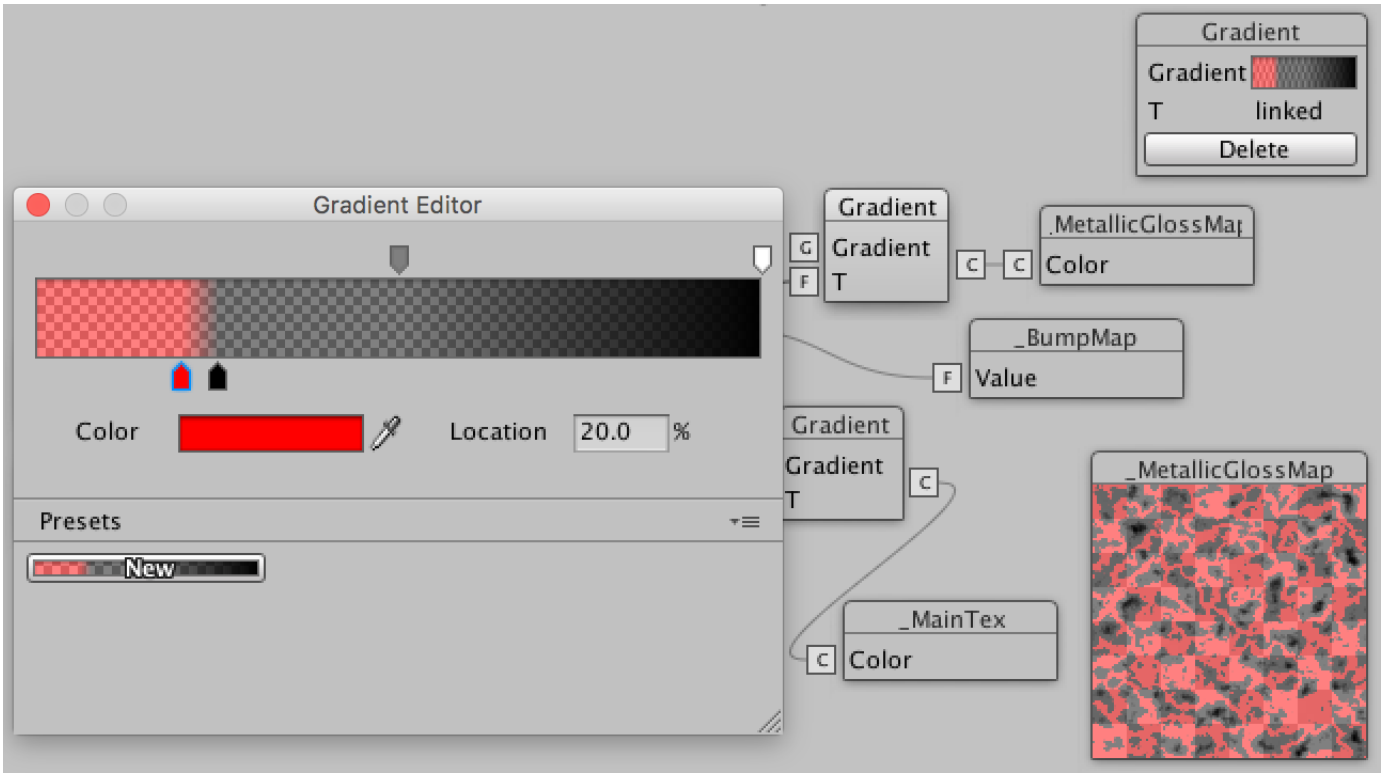


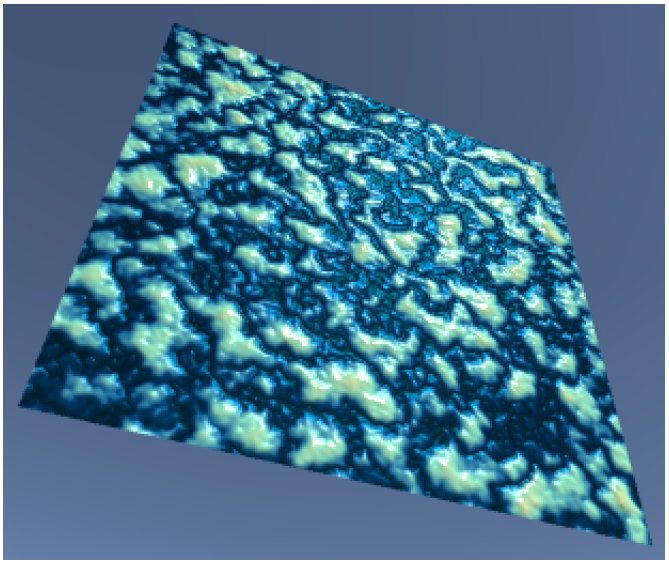
Material with a normal map.

2.8 Controlling Metallic and Smoothness Levels

The Standard shader supports even more maps, for example the metallic and smoothness map. You can support it by adding another output named *_MetallicGlossMap* to the diagram. This has to be an ARGB texture that stores the metallic value in its R channel and the smoothness value in its A channel. You could control these values with a gradient.

For example, make the bottom 20% of the gradient metallic and fade to nonmetallic at 25%. And increase smoothness from 0.5 – that's 127 alpha – to 1.0 between 50% and 100%.





Using a metallic and smoothness map.

3 Managing Multiple Scenes

Applications often consist of multiple scenes. You will need to include an instance of your diagram material manager prefab in each scene, otherwise textures won't be generated for it. To try this out, duplicate your scene and make some change to the new one so you can tell them apart.

When switching between those scenes, you will see that the textures are recreated for each one. So it works in the editor, but how does it behave in play mode?

3.1 Switching Scenes in Play Mode

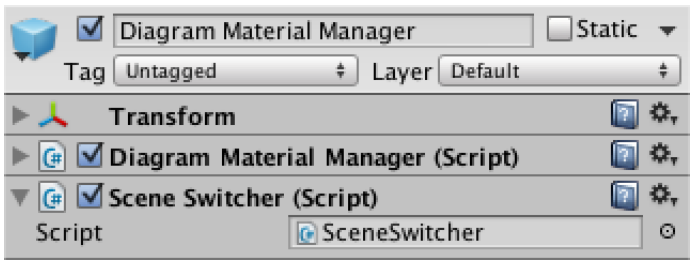
To switch between scenes in play mode, you need a script to do so. Here is a very simple C# script that alternates between the first and second scene when pressing the space bar.

```
using UnityEngine;

public class SceneSwitcher : MonoBehaviour {

    void Update () {
        if (Input.GetKeyDown(KeyCode.Space)) {
            Application.LoadLevel(1 - Application.loadedLevel);
        }
    }
}
```

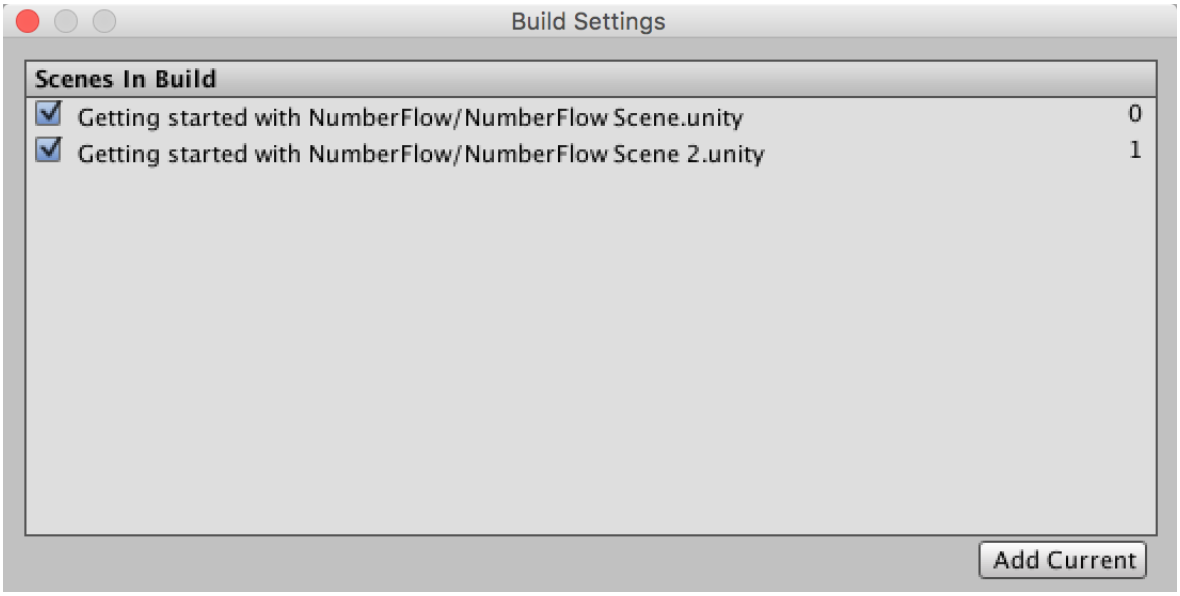
As you need this script in both scenes, it is easiest to simply add it to your manager prefab.



Manager with a scene switcher component.

3.2 Adding the Scenes

Next, you need to configure your application via **File** / **Build Settings...** and add both scenes. Then you can enter play mode and switch between them by hitting space.



Adding scenes via build settings.

You will notice that it behaves just like opening another scene while in edit mode. Everything in the current scene is destroyed, then the objects in the next scene are created. Unfortunately, this means that all the generated textures are destroyed and are then generated again.

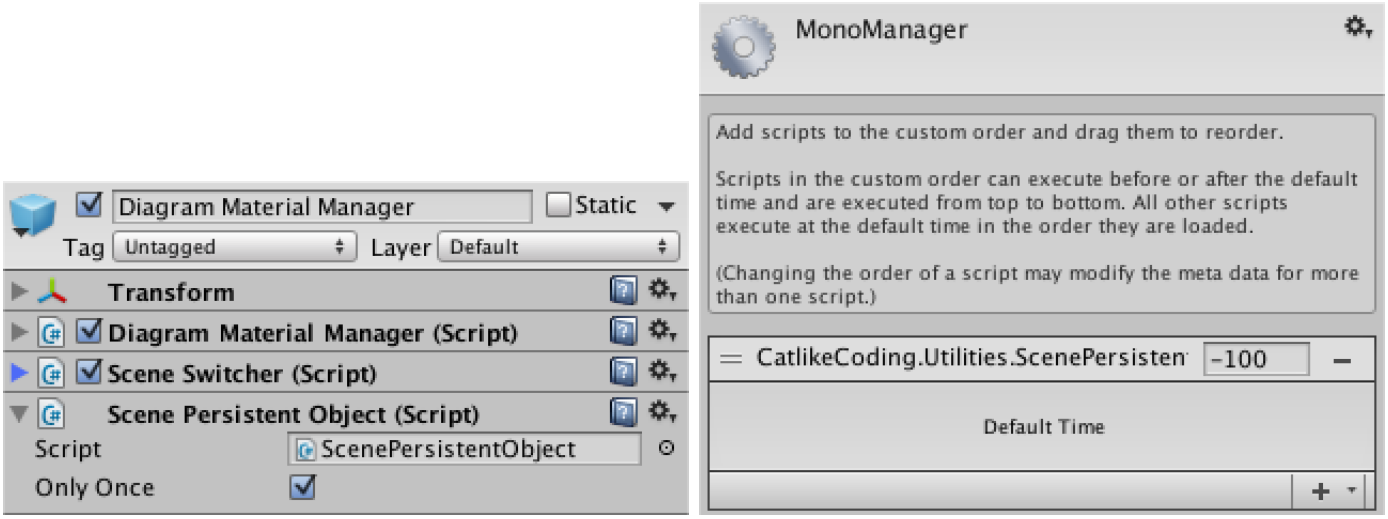
By the way, Unity screws up the global illumination when switching scenes in the editor this way, resulting in incorrect lighting. Builds work fine though.

3.3 Keeping One Manager Alive

If you can prevent the manager instance from being destroyed when another scene is loaded, then the textures won't be destroyed either. You can use the `DontDestroyOnLoad` method to accomplish this via a script. However, this won't stop the manager from the other scene to be created, so you would end up with two manager instances. To prevent double management, you have to get rid of the second manager instance before it awakens.

While you could program this yourself, there already is a script that does all this work for you. You can add it to your manager prefab via *Component / Scripts / CatlikeCoding.Utilities / Scene Persistent Object*. Make sure that *Only Once* is checked.

To guarantee that this component gets rid of any unwanted manager instances before they awaken, add it to the top of the script execution order, which you can find via *Edit / Project Settings / Script Execution Order*.



A manager that persists.

Having a manager per scene is useful when you're editing those scenes manually. If you're using multiple scenes but don't edit them by hand, you wouldn't need to include a manager instance in each of them. Including one in the first scene would suffice.

documentation written by Jasper Flick