Weaver v3.0

Created by Kybernetik

Contents

1. Feature Comparison

This table is a comparison of various approaches to asset creation in Unity.

- **Manual**: use the Unity Editor interface to create objects, add components, drag and drop references, etc.
- **Procedural at Runtime**: use scripts to create objects from scratch when the game loads or during execution.
- **Procedural in Editor**: as above, but done in the Unity Editor before the game is built.
- **Weaver**: a framework for Procedural in Editor.

	Manual	**Procedural at Runtime**	**Procedural in Editor**	**Weaver**
No coding required	✓			
No magic strings for loading assets		✓		✓
No magic strings or bitwise operations for using layers				✓
Easily reference procedural objects in code		✓		✓
Easily reference non-procedural assets in code				✓
Reduced build size		✓		
Fast initialisation at runtime	✓		✓	✓
Automatic reference caching for fast reuse				✓
Create dynamic objects and meshes		✓		*
Simplified procedural mesh generation				✓
Use editor-only functionality	✓		✓	✓
View and place objects in edit mode	✓		✓	✓
Create procedural scripts			✓	✓
Register automatic Pre-Build events with an attribute				✓
Easily make several similar objects which inherit from the same base		✓	✓	✓
Easily perform complex operations like loops and calculations		✓	✓	✓
Merge differences when using version control on a team		✓	✓	✓
Easily view and manage procedural assets in the Unity Editor				✓
Automatic asset saving				✓
Automatic sub-asset management				✓
Automatic dependency management				✓

* Code written for Weaver Procedural Assets can also be called at runtime if necessary.

Weaver Feature Comparison	**Lite**	**Pro**
Asset Linker: directly reference assets via auto-generated properties instead of using magic strings with `Resources.Load`.	Max 10 Assets	Unlimited
Asset Collections: turn an asset folder into an array or dictionary instead of loading assets individually.		✓
Customise the Asset Linker Settings.		✓
The Weaver Asset Class simplifies loading and caching of assets.	✓	✓
Procedural Assets: easily create procedural assets of any type: prefab, script, mesh, texture, etc.	Max 5 Assets	Unlimited
Procedurally generate Asset Collections.		✓
Layer Linker: directly reference layers via auto-generated constants instead of using magic strings with `LayerMask.NameToLayer`.	✓	✓
Easily create custom Layer Masks without bit shifting.	✓	✓
Detailed user manual and examples.	✓	✓
Source code included.	Kybernetik Core only	✓

You can open the Weaver Pro page in the asset store via Window/Weaver/Purchase Weaver Pro.

2. Quick Start Guide

If reading an 18 page manual isn't your idea of fun, here's a quick guide on how to use Weaver.

- Weaver's menu items are located in *Window/Weaver*.
- You can use the Asset Linker to replace string based resource loading with direct references. Open it via *Window/Weaver/Asset Linker*, select the assets you want to link and click [**Apply**], then you can use the procedurally generated `Weaver.Assets` class to reference the linked assets in any script.
- You can use the Layer Linker to replace string based layer access with constants in the procedurally generated `Weaver.Layers` class. Open it via *Window/Weaver/Layer Linker*.
- The Weaver Asset Class is useful for accessing assets outside of the Asset Linker.
- You can turn any `static readonly Weaver.Asset` field into a Procedural Asset by creating a static method in the same class called `GenerateX`, where X is the name of the field.
- The `Weaver.MeshBuilder` class is very useful for procedurally generating meshes. An example of its use can be found in *Assets/Weaver/Crosshair.cs*.
- The main Weaver settings can be found in *Edit/Preferences/Weaver*.
- You can also find some other useful features in the Appendix.
- The *Assets/Weaver* folder contains an example scene which shows some of the ways Weaver can be used. You can delete that folder once you are done with the examples.

2.1. Weaver Namespace

If you frequently use classes in the `Weaver` namespace, you may want to consider adding `using Weaver;` to the top of Unity's default script template (located at C:\Program Files\Unity\Editor\Data\Resources\ScriptTemplates\81-C# Script-NewBehaviourScript.cs on Windows). Note that this change will be overwritten if you install a new version of Unity.

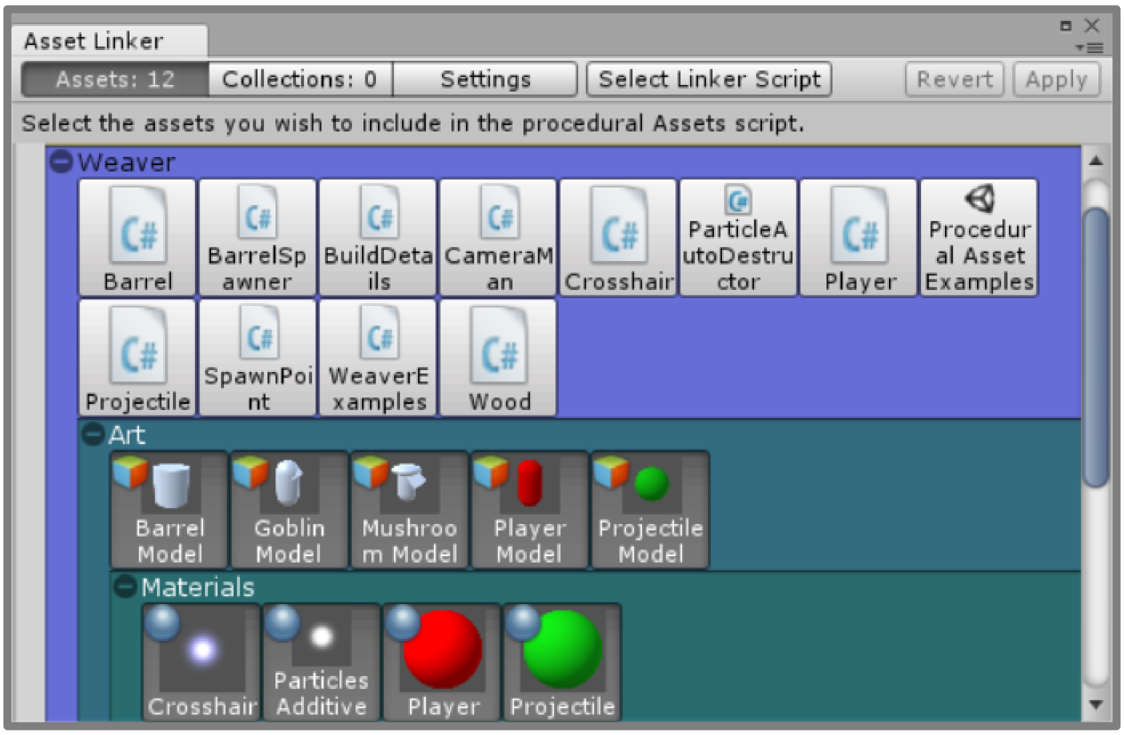
3. Asset Linker

The Asset Linker is a system that procedurally generates a script containing fields and/or properties which automatically load and cache assets for you. Where you would normally use `Resources.Load<GameObject>("Creatures/Goblins/Warrior")`, this script would let you use `Assets.Resources.Creatures.Goblins.Warrior` instead.

- For a summary of the benefits of this feature, take a look at Benefits of the Asset Linker.
- The Asset Linker can be opened via *Window/Weaver/Asset Linker*.
- The generated script is called *Assets.cs* and is located in *Assets/Weaver* unless you chose a different path for procedural scripts.
- The script will automatically regenerate itself whenever you change, add, or remove any assets it targets (after you click out of the Project Window). If the game is currently playing, it will wait until it stops before regenerating.

3.1. Assets

The Assets tab displays all the assets in your project so you can select which ones you want the generated script to contain. The right click context menu contains several other useful functions.

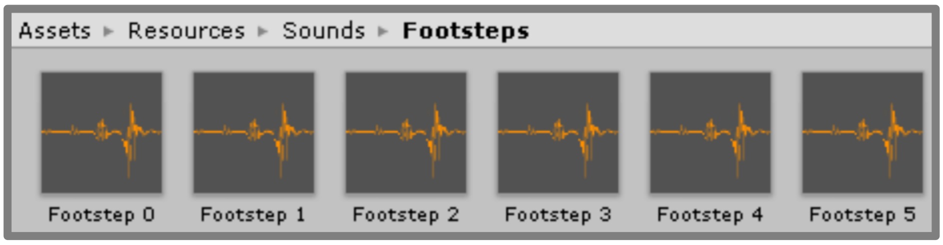


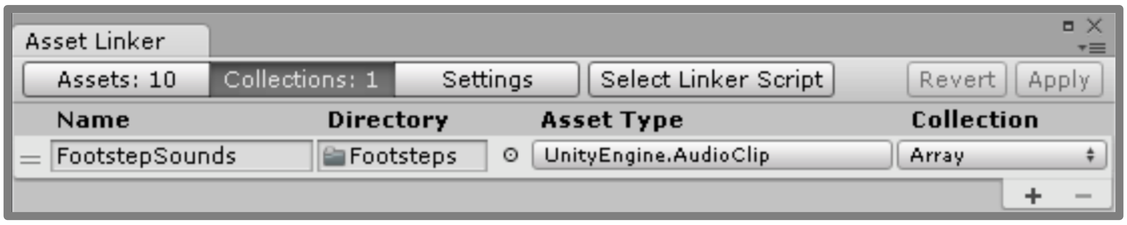
Select the assets you wish to include in the procedural Assets script.

- You can also link and un-link assets using *Assets/Add Selection to Asset Linker* and *Assets/Remove Selection from Asset Linker*. These commands are also accessible by right clicking in the Project Window.
- You can drag and drop the asset buttons out into the scene or hierarchy to instantiate objects as if you were dragging their assets out of the Project Window.
- Linked assets are shown in the Project Window with a link icon which you can middle click to open the Asset Linker.
- Note that Weaver Lite only allows you to link up to 10 assets at a time. Weaver Pro has no limit.

3.2. Collections [Pro-Only]

Sometimes you want to access assets as a group rather than referencing them individually; for example, you might want an array of weapon prefabs for the player to choose between or an array of slightly different footstep sounds so you can pick a random one each time the player takes a step. This is what the Collections tab is for; instead of selecting each of the audio clips you want in the Assets tab individually, you could add a collection called *FootstepSounds*, select the folder containing the assets you want, change the asset type to `UnityEngine.AudioClip`, and set the collection type.





Name	Directory	Asset Type	Collection
FootstepSounds	Footsteps	UnityEngine.AudioClip	Array

Note that you can drag and drop an asset onto the Asset Type field to use its type.

The above images will result in the following code being added to the *Assets* script:

```
/// <summary>Assets/Resources/Sounds/Footsteps as Array.</summary>
public static readonly Asset<UnityEngine.AudioClip>[]
    FootstepSounds =
    {
        "Sounds/Footsteps/Footstep 0",
        "Sounds/Footsteps/Footstep 1",
        "Sounds/Footsteps/Footstep 2",
        "Sounds/Footsteps/Footstep 3",
        "Sounds/Footsteps/Footstep 4",
        "Sounds/Footsteps/Footstep 5",
    };
```

3.3. Settings [Pro-Only]

The Settings tab allows you to customise the contents of the generated script. The options all have tooltips explaining what they do.

When configuring the Asset Linker, take note of the following:

- Procedural assets are hidden by default because you should generally access them via the field you used to declare them, not via the Asset Linker.
- If you disable `Weaver.Asset` fields, the properties will create their own backing fields in which to cache their assets once loaded. This will provide an extremely minor performance increase as it avoids the overhead of an extra generic object, but it prevents you from using some features like easy path serialization from The Weaver Asset Class.
- Asset collections will always use `Weaver.Assets`, regardless of whether you have them enabled for individual assets. This allows the system to load them when first used rather than loading the entire collection at once.

3.4. Benefits of the Asset Linker

Unity's system for loading assets at runtime (namely `Resources.Load`) has several major flaws which prevent it from being an ideal tool for the job. Take the following line for example:

```
Resources.Load<GameObject>("Creatures/Goblins/Warrior")
```

- First you need to go into your project and find the exact path to the asset you want, then type it into your code. If you want to change it to a different kind of goblin, you have to look in your project again to find out what they're each called.
- If you spell the path wrong, it will compile, but won't actually work when you run it.
- If you move or rename the asset, you need to remember to update the code or you'll waste time figuring out why previously working code is now broken.

The original design goal for Weaver was to solve these problems. Instead of using the above line, the Asset Linker would generate the following property for you to use:

```
Assets.Resources.Creatures.Goblins.Warrior
```

- Any good code editor will be able to display a list of all members in a class (in Visual Studio this feature is called Intellisense). Instead of swapping back and forth between your code and your project, you can simply use that list to pick the asset you want. Changing to a different kind of goblin is just as easy.
- If you spell the path wrong, the code simply won't compile. You'll get an error that tells you exactly where the problem is.
- If you move or rename the asset, Weaver will update the property accordingly and you'll get compile errors everywhere you referenced it (because it no longer exists) instead of needing to remember to find and replace each instance manually.
- This property will also cache the loaded asset to improve performance if you need to use it multiple times.

3.5. Feedback

The Asset Linker system is still very new and while an effort has been made to ensure it is as robust as possible, there will still be situations where it generates invalid code or even throws exceptions. If this happens, please email the generated *Assets.cs* file to kybernetikgames@gmail.com along with a description of the offending file/folder structure and any other relevant details or recommendations on how the situation should be handled.

Other feedback and feature requests are also welcome. Do you want it to use a different naming convention? Do you want to add assets in different folders to the same collection? Do you have a suggestion on how the GUI could be improved? Would you want to be able to assign a custom alias so that `Assets.Resources.Creatures.Goblins.Warrior` can instead be accessed as `Goblins.Warrior` or `Creatures.Goblins.Warrior`? Let us know.

4. The Weaver Asset Class

The `Weaver.Asset<T>` class can be used to simplify the way you access resources and other assets. For example, the following code:

```
private static GameObject _Warrior;
public static GameObject Warrior
{
    get
    {
        if (_Warrior == null)
            _Warrior = Resources.Load<GameObject>("Creatures/Goblins/Warrior");
        return _Warrior;
    }
}
```

Can be replaced with a single field:

```
public static readonly Asset<GameObject> Warrior = "Creatures/Goblins/Warrior";
```

- You can either use an implicit conversion from string as shown above, or you can use a regular constructor like so: `new Asset<GameObject>("Creatures/Goblins/Warrior")`.
- You can access the actual goblin warrior prefab using `Warrior.Target`. This will load the prefab the first time it is actually used and cache the value for better performance when you need the asset again in the future, just like the property in the above example.
- You can also implicitly cast the `Warrior` field to a `GameObject`. Unfortunately, you can't use the basic `Object.Instantiate(Warrior)`, but you can use any of the other overloads:
  - `Object.Instantiate(Warrior.Target)`
  - `Object.Instantiate(Warrior, position, rotation)`
  - `Object.Instantiate<GameObject>(Warrior)`
  - You can also use `Warrior.Instantiate()` or any of its overloads.
- If you specify a path that begins with `"Assets/"` and includes the file extension (such as `"Assets/Art/Creatures/Goblins/Warrior.fbx"`), it will use `AssetDatabase.LoadAssetAtPath` instead of `Resources.Load`. This allows you to target any asset in your project, though it won't be able to load it at runtime once your project is built. The Asset Linker makes use of this feature if you have `Weaver.Asset` fields enabled.
- You can get the asset path using `Warrior.Path`. This makes it easy to serialize the asset path without needing to store it in a separate variable.

5. Procedural Assets

To create a procedural asset, you first define it in a script then use the Procedural Asset Manager to generate it.

- For a summary of the benefits of this feature, see Benefits of Procedural Assets.
- Note that Weaver Lite only allows you to define up to 5 Procedural Assets. Weaver Pro has no limit.

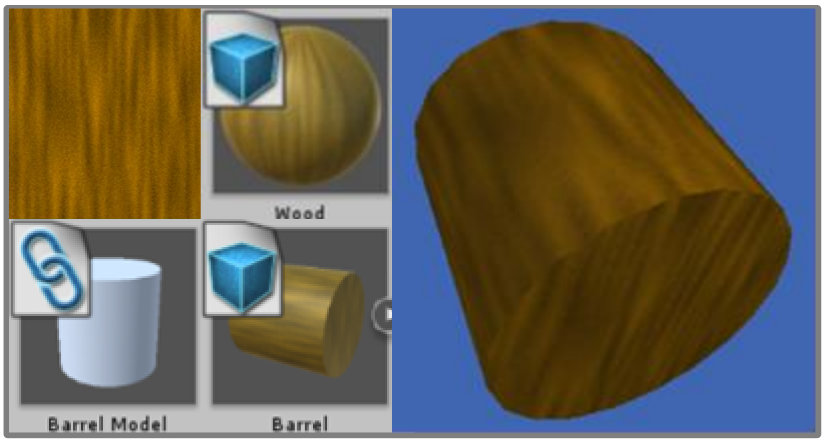
5.1. Defining a Procedural Asset

To define a procedural asset, you simply make a `static readonly Weaver.Asset<T>` field in any non-generic class along with a `static` method called `GenerateX`, where X is the name of the field. For example:

```
public static readonly Asset<GameObject> Player = "Player";

private static GameObject GeneratePlayer()
{
    var go = new GameObject();
    // Add components and scripts, instantiate a model, etc.
    return go;
}
```

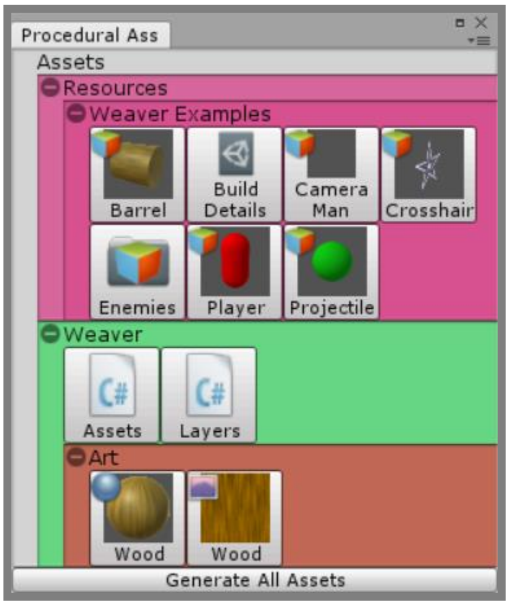
- You can use a procedural asset just like any other asset. You can drag and drop a procedural prefab from the project window into a scene and it will update all instances whenever you regenerate it. Or you can use it in code just like any other `Weaver.Asset`.
- If you define a procedural asset using a resource path (I.E. no `"Assets/"` at the start and no file extension at the end) it will be put into `"Assets/Resources/"` by default, and will use the default file extension for its asset type (see Asset Generators).
- Weaver uses reflection to find all `static readonly Asset` fields and generator methods regardless of their access modifier (`public`, `private`, etc), so you can control access to them as you see fit.
- If it finds a method, it will verify that it has the correct return type and parameters, and use a console message to let you know if it is wrong. If it doesn't find a method, that field won't be treated as a procedural asset.
- You can make a procedural asset of any type derived from `UnityEngine.Object`, such as `GameObject`, `Mesh`, `TextAsset`, `Texture`, `AudioClip`, or your own `ScriptableObject` type. Some asset types can't be saved directly by Unity, see Asset Generators for more details.



5.2. Using the Procedural Asset Manager

Once you have defined a procedural asset (or made changes to an existing one), you need to generate it using the Procedural Asset Manager.

The Procedural Asset Manager can be opened via *Window/Weaver/Procedural Asset Manager*.



The Procedural Asset Manager displays every procedural asset in your project, grouping them according to their resource path (for resources) or asset path (for other assets).

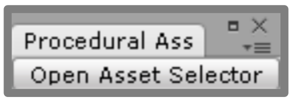
Clicking on an asset or folder:

- **Left Click** = select/deselect.
- **Double Left Click or Middle Click** = generate.
- **Right Click** = open context menu.

Clicking the [**Generate**] button:

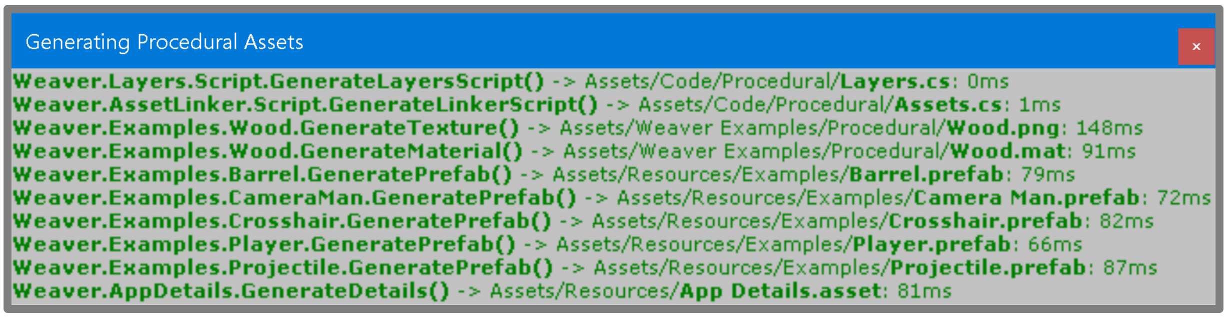
- **Left Click** = generate selected assets.
  If nothing is selected, generate all.
- **Right Click** = generate all assets.

You can drag and drop the asset buttons out into the scene or hierarchy to instantiate objects as if you were dragging their assets out of the Project Window.



If you shrink the window down to be very short, it changes the [**Generate**] button so that Left Click opens the asset selector as a popup window (the other buttons remain the same). This allows you to keep it within reach at all times by docking it into another window (such as at the bottom of the hierarchy or inspector).

The Procedural Asset Manager also has a few useful menu items (in the context menu in the top right, just under the X).



```
Weaver.Layers.Script.GenerateLayersScript() -> Assets/Code/Procedural/Layers.cs: 0ms
Weaver.AssetLinker.Script.GenerateLinkerScript() -> Assets/Code/Procedural/Assets.cs: 1ms
Weaver.Examples.Wood.GenerateTexture() -> Assets/Weaver Examples/Procedural/Wood.png: 148ms
Weaver.Examples.Wood.GenerateMaterial() -> Assets/Weaver Examples/Procedural/Wood.mat: 91ms
Weaver.Examples.Barrel.GeneratePrefab() -> Assets/Resources/Examples/Barrel.prefab: 79ms
Weaver.Examples.CameraMan.GeneratePrefab() -> Assets/Resources/Examples/Camera Man.prefab: 72ms
Weaver.Examples.Crosshair.GeneratePrefab() -> Assets/Resources/Examples/Crosshair.prefab: 82ms
Weaver.Examples.Player.GeneratePrefab() -> Assets/Resources/Examples/Player.prefab: 66ms
Weaver.Examples.Projectile.GeneratePrefab() -> Assets/Resources/Examples/Projectile.prefab: 87ms
Weaver.AppDetails.GenerateDetails() -> Assets/Resources/App Details.asset: 81ms
```

- When you generate procedural assets, a popup displays a list of all assets being generated.
- After each asset is generated, the time it took to generate and save it is shown (measured in milliseconds), followed by the time it took to generate. So an asset that shows “100ms (10)” took 10 milliseconds to execute the asset generator method itself, and a total of 100 milliseconds for the entire generation and saving process.
- The window will close automatically when done unless you hold shift as it finishes.
- If an exception is thrown by any of the assets being generated, it stops generating and shows the offending asset in red along with the exception details and buttons to Retry, Skip, Abort, or open the source file.

5.3. Asset Generation Process

Weaver executes the following process to generate a Procedural Asset:

1. Create a new temporary scene so that the current scene isn't inadvertently modified.
2. If the `AssetPath` has changed since the last time the asset was generated, move the old asset to the new path.
3. Identify the correct Asset Generator to use for the asset based on its type.
4. Invoke the generator method using that Asset Generator.
5. Save the returned asset at the specified `AssetPath`. If the asset was given a resource path (I.E. doesn't begin with "*Assets/*" and doesn't have a file extension) then it will be saved in "*Assets/Resources/…*" and given an appropriate file extension by the Asset Generator. These details can be modified by giving the generator method an `[AssetModifier]` attribute and setting the `BaseResourcesPath` property.
6. Close the temporary scene and return to the original scene setup.

5.4. Asset Generators

By default, a procedural asset is saved using `AssetDatabase.CreateAsset` once your generator method returns it. Unfortunately, this doesn't work for some asset types; a `GameObject` must be saved using the `PrefabUtility` class, a `Texture` must be encoded in a specific image format (such as png), and so on. This is what Asset Generators are for:

- You don't need to call an Asset Generator yourself; Weaver will automatically pick the one that's most appropriate for each asset type. However, you do need to be aware that certain asset types use different rules than others.
- Each Asset Generator targets a specific asset type, for which it defines the default file extension (which is used if you don't include a file extension in the file path when declaring the `Weaver.Asset`), as well as the return type and parameters you must use for the generator method. Weaver currently includes 5 Asset Generators:

Class Name	Asset Type	File Extension	Return Type	Parameters
`AssetGenerator<T>`	`Object`	`.asset`	`T`	`()`
`PrefabGenerator`	`GameObject`	`.prefab`	`GameObject`	`()`
`TextureGenerator`	`Texture2D`	`.png`	`Texture2D`	`()`
`TextGenerator`	`TextAsset`	`.txt`	`void`	`(System.Text.StringBuilder)`
`ScriptGenerator`	`MonoScript`	`.cs`	`void`	`(System.Text.StringBuilder)`

- You can define your own Asset Generator by making a class that inherits from `Weaver.AssetGenerator<T>`. Weaver will find your class via reflection and use it whenever it generates a procedural asset of type `T`.
- The Procedural Asset Manager has a menu item called *Log All Asset Generators* which can be used to find out the details of all Asset Generators currently in your project.

5.5. Auto-Generate in Play Mode

Weaver 3.0 introduces a new experimental feature in the Unity Editor which allows it to automatically regenerate each procedural asset the first time it is accessed in Play Mode. This means you can modify the generator method then immediately run the game without needing to manually regenerate the asset using the Procedural Asset Manager first.

- This feature is disabled by default. It can speed up your development, but requires additional care to use correctly. You can enable it via *Edit/Preferences/Weaver*.
- The first time a procedural asset is accessed while playing the game in the Unity Editor, this feature will call its generator method instead of loading the asset from the saved file.
- If an exception is thrown, the asset will be loaded normally.
- Assets generated using this feature will not be saved, so you will still need to use the Procedural Asset Manager before making a build.
- This feature will not work for all assets. For example, there's no point in generating procedural scripts at runtime and assets like the example `BuildDetails` script where the generator method references the asset itself would cause infinite recursion. In these cases, Weaver simply loads the asset normally.
- You can disable this feature for a specific asset by giving the generator method an `[AssetModifier]` attribute and setting the `AutoGenerateInPlayMode` property.
- This feature is disabled by default because there are some significant differences between edit mode and play mode which can cause major problems.
    - `MonoBehaviour.Reset` is called when a component is added in edit mode but not play mode while `MonoBehaviour.Awake` is called in play mode but not edit mode.
    - While normally you'd be able to add a component then configure it as you like so that it's ready for `Awake` to be called when it gets instantiated at runtime, doing so while using this feature would mean that `Awake` gets called as soon as you add the component before you get a chance to configure it.
    - If this causes problems for you, you can try disabling each `GameObject` as soon as you create it in the generator method and then enabling it once you have added and configured all the components so Unity can safely call the `Awake` methods. This is done automatically if you make the generator method take a `GameObject` parameter instead of returning a new `GameObject`.

5.6. Auto-Generate on Build

When you build your application, Weaver will automatically generate all of your procedural assets to ensure that they're up-to-date.

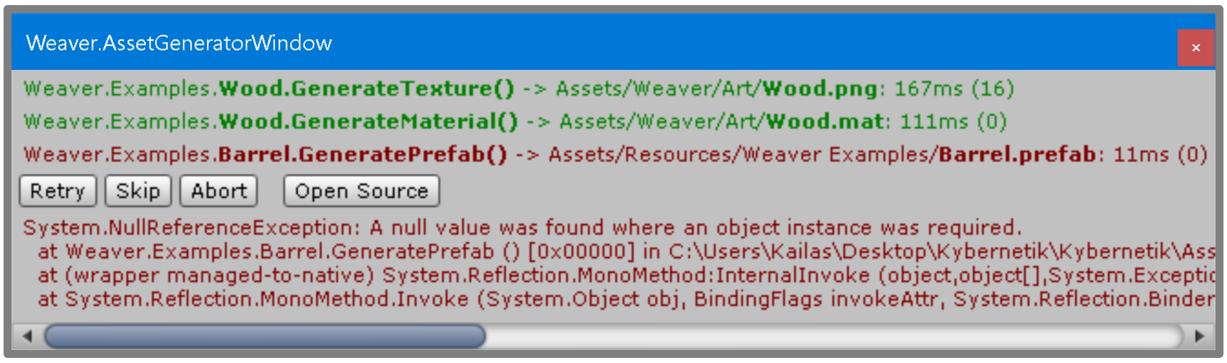
- This feature is enabled by default. You can disable it via *Edit/Preferences/Weaver*.
- You can disable this feature for a specific asset by giving the generator method an `[AssetModifier]` attribute and setting the `AutoGenerateOnBuild` property.
- This feature does not apply to procedural scripts since Unity would not include the newly generated script in the build.
- If you want to execute other functions during a build, you can use the PreBuild Attribute.

5.7. Using Editor-Only Functionality to Generate Assets

If your generator method uses any functionality only available in the Unity editor, you will need to use conditional compilation (`#if UNITY_EDITOR` and `#endif`) around such functionality to be able to build your project without errors. If you don't need to use your generator method at runtime, you can put it around the whole method.

5.8. Exceptions

If an exception is thrown while generating a procedural asset, Weaver will stop generating and display the exception along with buttons to retry, skip, abort, or open the source file which threw the exception.



```
Weaver.Examples.Wood.GenerateTexture() -> Assets/Weaver/Art/Wood.png: 167ms (16)
Weaver.Examples.Wood.GenerateMaterial() -> Assets/Weaver/Art/Wood.mat: 111ms (0)
Weaver.Examples.Barrel.GeneratePrefab() -> Assets/Resources/Weaver Examples/Barrel.prefab: 11ms (0)
System.NullReferenceException: A null value was found where an object instance was required.
  at Weaver.Examples.Barrel.GeneratePrefab () [0x00000] in C:\Users\Kailas\Desktop\Kybernetik\Kybernetik\Ass
  at (wrapper managed-to-native) System.Reflection.MonoMethod:InternalInvoke (object,object[],System.Exceptic
  at System.Reflection.MonoMethod.Invoke (System.Object obj, BindingFlags invokeAttr, System.Reflection.Binder
```

5.9. Dependencies

Referencing one asset while generating another creates a dependency. This means that whenever either one of them is generated, the other should also be generated (dependency first). Weaver will automatically detect asset dependencies and generate such assets together in the correct order.

The Weaver examples include a procedural barrel prefab (`Barrel.Prefab` in *Barrel.cs*) which instantiates a model and assigns a procedural material to it (`Wood.Material` in *Wood.cs*). The material is created using the standard shader with its main texture set as a procedural texture (`Wood.Texture` in *Wood.cs*).

If you tried to generate the barrel prefab before the wood material, it would try to load the non-existent material and end up assigning null to the barrel, which isn't what you'd want. It doesn't really apply to this example, but if you had a procedural prefab (the *parent*) which instantiates another procedural prefab (the *child*) while generating, generating just the *child* on its own would mean that the *parent* no longer has the most up to date version of the *child* because Unity doesn't support nested prefabs. This is why Weaver automatically detects and enforces asset dependencies for both parents and children.

Weaver detects procedural asset dependencies by keeping track of any `Weaver.Asset`s that are used during a generator method. Unfortunately, this means that dependencies can only be detected during generation, which has two minor drawbacks:

- The first time you generate the barrel prefab it will detect that the wood material is referenced and add it as a dependency. If the wood material wasn't already generated, Weaver will need to generate it then generate the barrel prefab again. This wastes a little bit of time but will only happen when a new dependency is added to a procedural asset.
- If the generator method doesn't always reference the dependency, it will cause the above situation to occur more often than necessary, which is simply annoying. You can avoid this by calling `AddDependency` on the asset while generating it.

5.10. Benefits of Procedural Assets

The Unity Editor is a great tool for game development; being able to manually put together prefabs and import models, textures, and sound files which you've created in other programs or downloaded from the internet is an absolute requirement for any good program. But as with any tool, sometimes there's a better tool for the job; sometimes you want to procedurally generate things using code. Likewise, sometimes it's better to procedurally generate things in the editor before building your game instead of doing it when your game starts up or while it is running. Here are the main benefits of doing it this way:

- You can use procedural assets like regular assets (drag and drop into a scene). This is particularly useful if you are working in a team where a programmer can procedurally generate something then a level designer can position it in the scene however they want without needing to touch the code.
- You don't need to worry as much about the efficiency of your generation code since it won't affect runtime performance at all. Even just instantiating a prefab with a few components on it is more efficient than creating a new game object and adding the components individually at runtime. This is most important for things like procedural meshes and textures.
- You can use editor only functionality which wouldn't be available at runtime. For example, you can load assets from anywhere in the project instead of needing to put them in a Resources folder.
- You can procedurally generate scripts (like the *Assets.cs* and *Layers.cs* included in Weaver). It is possible to generate code at runtime using reflection, but it is much more complicated, less efficient, and isn't allowed on some platforms.

6. Procedural Asset Collections [Pro-Only]

Sometimes it is useful to generate an unknown number of procedural assets. For example, you might want to set up a spreadsheet containing the stats of every enemy in your game so you can easily analyse and work with them in a program such as Microsoft Excel before turning them into prefabs in Unity.

6.1. Defining a Procedural Asset Collection

To define a Procedural Asset Collection, you simply make a `static` method which returns a collection of assets and give it a `[ProceduralAssetCollection]` attribute.

- The method can return any type that implements the `IEnumerable<T>` interface, where `T` inherits from `UnityEngine.Object` (such as an array or `List<T>`). Note that the chosen return type has no impact on the way the assets are accessed at runtime (see below).
- Once defined, a collection will show up in the Procedural Asset Manager to be generated like any other Procedural Asset and automatically added to the Asset Linker to be accessed like a manually specified Collection.
- If you want to change the runtime collection type to be something other than an array (such as a dictionary), you can specify a `builderType` in the `[ProceduralAssetCollection]` attribute's constructor. Weaver's inbuilt collection builders are located in the `Weaver.AssetLinker.Collections` namespace.
- See *Assets/Weaver/Collection Example* for an example collection.
- Note that the Asset Linker will include every asset of the same type in the directory of a Procedural Asset Collection, even assets that weren't generated as part of the collection.

6.2. Collection Generation Process

Weaver executes the following process to generate a Procedural Asset Collection:

1. Create a new temporary scene so that the current scene isn't inadvertently modified.
2. Ensure that the `AssetPath` directory exists. If the collection was given a resource path (I.E. doesn't begin with "*Assets/*") then it will be saved in "*Assets/Resources/…*". This can be modified by giving the generator method an `[AssetModifier]` attribute and setting the `BaseResourcesPath` property.
3. If the directory has changed since the last time this collection was generated, move any assets generated last time to the new directory.
4. Identify the correct Asset Generator to use for the collection based on the method's return type (specifically the `T` type parameter).
5. Invoke the generator method using that Asset Generator.
6. For each asset in the returned collection, save it in the directory using its name as the file name and the default file extension of the Asset Generator.
7. If any assets generated last time were not overwritten this time, delete them.
8. Close the temporary scene and return to the original scene setup.

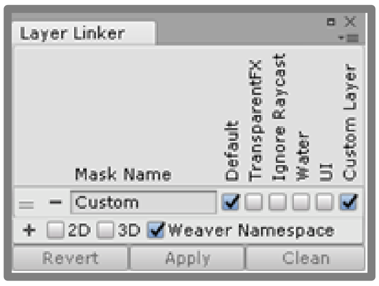
7. Layer Linker

Weaver includes a system which allows you to work with layers as `int` constants rather than via `string` based lookups as you normally would. To do this, it procedurally generates a script containing a constant for each layer in your project, allowing you to use `Weaver.Layers.MyLayer` where you would normally have used `LayerMask.NameToLayer("MyLayer")`.

The generated script is called *Layers.cs* and is located in *Assets/Weaver* unless you chose a different path for procedural scripts. It will automatically regenerate itself whenever you change, add, or remove any layers.

7.1. Layer Masks

You can customise the procedural `Layers` class using *Window/Weaver/Layer Linker*.



Each mask you create will correspond to a constant in the generated class, named using the mask name field followed by "Mask". For example, the image to the right would generate the following code:

```
/// <summary>Default | Custom Layer</summary>
public const int CustomMask = 8;
```

The toggles for *2D* and *3D* determine whether or not the generated script should also include a constant representing the layer collision matrix for each layer as specified in the physics settings.

The Layer Linker window has several menu items which are useful shortcuts to various layer related menus: *Tags and Layers*, *Physics Settings*, and *Physics 2D Settings*.

7.2. Obsolete Values

When generating the `Layers` class, the system checks whether any constants are about to be removed from the file, such as if you rename or remove a custom layer or mask. Such constants are kept in the new file and given the `Obsolete` attribute. This means that any code referring to the marked constant will continue to work, but Unity will give you a warning letting you know that you need to update your code to match the change you just made. Once you have found and resolved all such warnings, you can use the [**Clean**] button in the Layer Linker to regenerate the script with all obsolete fields removed.

8. Appendix

8.1. Troubleshooting

If you get compile errors when importing Weaver, make sure you delete any older versions of Weaver before importing the new one.

If you still get compile errors, try deleting the examples folder (*Assets/Weaver*).

If you try to delete Weaver and Unity immediately reimports it, simply close Unity and delete the files manually. Weaver itself doesn't have the ability to recreate itself like that so if this happens, please submit a bug report to Unity.

If you experience any other problems, please contact: kybernetikgames@gmail.com.

8.2. The PreBuild Attribute

You can register a method to be called when you build your project by giving it a `PreBuild` attribute.

- Static methods are run once at the start of a build.
- Instance methods are run once for every instance of the declaring script in each scene being built. Instance methods only work in Components, not on other types.
- You can optionally specify an execution time for the attribute, though static methods are kept in a separate list and will always go before instance methods.
- Note that procedural scripts cannot be generated by pre-build methods. More specifically, they can be generated, but Unity won't include the newly generated script in that build.
- You can execute all pre-build methods manually using *File/Invoke Pre-Build*.

8.3. Utilities

Weaver contains many utility classes and methods which you can make use of in your own code. This section outlines some of the most useful.

`Kybernetik.Utils` contains many miscellaneous methods with a variety of purposes:

- `DeepToString` can be used to print any collection (such as an array or list) as a string.
- `LogTemp` simply calls `Debug.Log`, but all calls to `LogTemp` will cause compile warnings. This is very useful if you want to log something for debugging purposes and want to make sure you don't forget to remove the call once you're done.
- `LogErrorIfModified` can be used for "locking" a collection which you don't want modified without allocating a new read-only collection. Note that this only logs an error, it doesn't actually prevent modifications.
- `Utils.Timer.Start` starts and returns a simple timer which inherits from `System.IDisposable` so you can write `using (Utils.Timer.Start())` to time something and automatically log the result when the using block completes.
- `Indent` is useful for generating neat procedural scripts and other multi-line text.

`Kybernetik.CSharp` contains a variety of methods relating to C# code which are particularly useful for generating procedural code:

- `GetReference` returns the full name of a type or member (such as a field or method) as it would appear in C# code. For example, the regular `typeof(List<float>).FullName` returns: "*System.Collections.Generic.List`1[[System.Single, mscorlib, Version=2.0.0.0, Culture=neutral,*

PublicKeyToken=b77a5c561934e089]]" while `typeof(List<float>).GetReference()` would instead return "*System.Collections.Generic.List<float>*" .
- `OpenScope` and `CloseScope` are simply shorthand for appending indented curly brackets and incrementing/decrementing the *indent* counter.

`Kybernetik.Reflection` contains methods to easily perform common reflection functions such as iterating through all types in the project to find everything with a particular attribute or everything which inherits from a particular base type.

`Kybernetik.ObjectPickerField` is useful for creating editor GUI fields which can be clicked to bring up a window for the user to select an item from a list. The Asset Linker uses it to allow the user to pick the asset type for each collection.

8.4. Feature Stripping

Weaver is comprised of several DLLs which are initially set up for ease of use, but you can make some minor performance optimisations by altering their settings. Instead of deleting a DLL you don't want, you can select it in the project window and deselect all platforms from its import settings. That way you can easily re-enable it if you later decide you want its features.

The Weaver DLLs are located in *Assets/Plugins/Weaver* by default.

- If you aren't using the Asset Linker, you can remove *Weaver.AssetLinker.dll*.
- If you aren't using the Layer Linker, you can remove *Weaver.LayerLinker.dll*.
- If you aren't using any `Weaver.Asset`s at runtime, you can remove *Runtime/Weaver.dll*.
- If you aren't using anything in the `Kybernetik` namespace at runtime, you can remove *Assets/Plugins/Kybernetik/Runtime/Kybernetik.Core.dll*.

8.5. Source Code Development [Pro-Only]

The Weaver Pro package contains its own source code as a Visual Studio 2015 solution located at *Assets/Plugins/Weaver/Source Code.zip*.

The solution contains several projects, each of which has a Post-Build event set up to automatically copy its DLL into the Unity project when it recompiles. For this to work, you need to extract the Source Code folder into your Unity project folder (the same folder as Assets, Library, etc).

8.6. Upgrading from the Procedural Asset Framework

Weaver is a revolutionary upgrade from the Procedural Asset Framework; the focus has been shifted from procedural assets towards the use of the new Asset Linker system and Procedural Assets themselves have undergone drastic structural changes.

This new system is better in almost every way: it's easier to learn and to use, there's less code to write which makes it faster to set up and modify, accessing an asset is as simple as using a regular field, and it should theoretically execute faster as well (though the performance impact of either system would be so small as to be very difficult to accurately test).

This section is a guide on how to deal with the major changes if you're upgrading from the old system.

While Weaver is distributed as a separate product, this is not an attempt to double charge anyone who purchased the Procedural Asset Framework. To upgrade, simply email your Invoice Number to kybernetikgames@gmail.com and you will receive a free voucher for Weaver.

The biggest change is the way Procedural Assets are declared: instead of making a new class, you just make a field and a method. Both of the following examples would generate a prefab at *Assets/Resources/Player.prefab*.

```
// Procedural Asset Framework.
public class Player : Procedural.Prefab
{
    public override string Name { get { return "Player"; } }

    protected override void Generate()
    {
        // ...
    }
}

// Weaver.
public static readonly Asset<GameObject> Player = "Player";

private static GameObject GeneratePlayer()
{
    // ...
}
```

With the change in declaration comes a change in the way you access procedural assets in code. Instead of using `Static<Player>.Instance` to access the asset, you just reference the `Player` field directly. Instead of using `Prefab.Instantiate<Player>()`, you use `Player.Instantiate()`.

Weaver also handles Procedural Asset Dependencies differently. Instead of using a `TypeDependency` attribute to manually specify a dependency, they are automatically detected and enforced by the system.

The `Weaver.Asset` class now handles everything `Procedural.ManualAsset` did as well as being used for procedural assets.

Questions, feedback, feature requests, etc: kybernetikgames@gmail.com.