Decoupled Pipeline Scripting

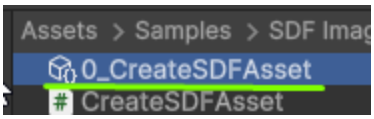
Status	Up-to date

For introduction see Starting Decoupled Pipeline

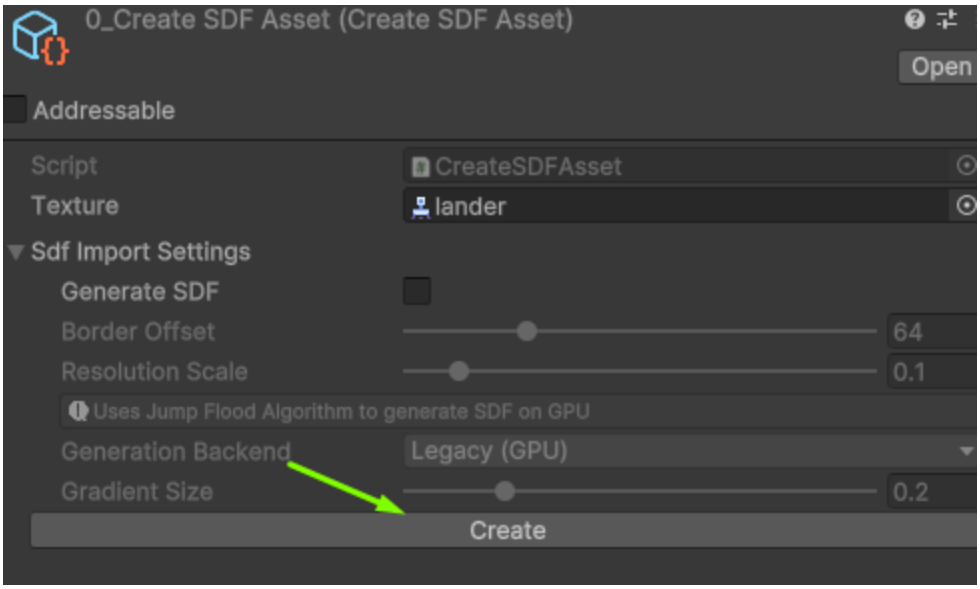
> Section contains examples of common decoupled pipeline in code usages

Creating SDFAsset

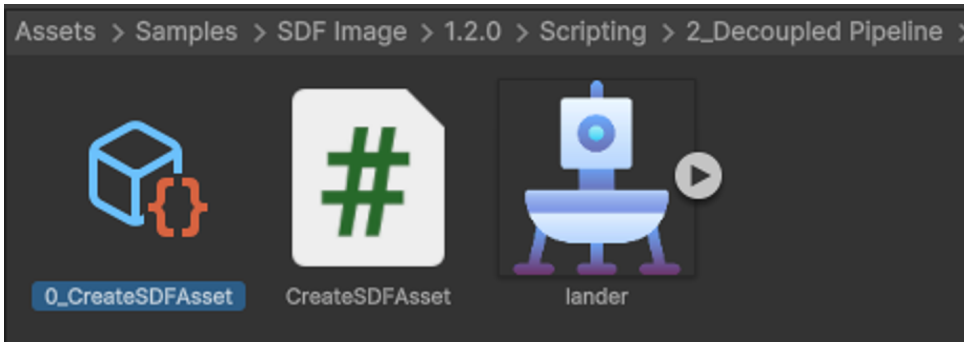
- Select CreateSDFAsset



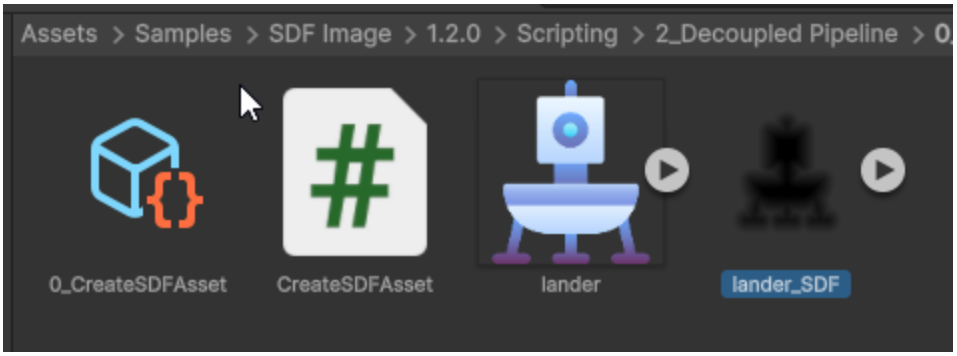
- Press Create button in its editor



- See how SDFAsset is generated for lander.png

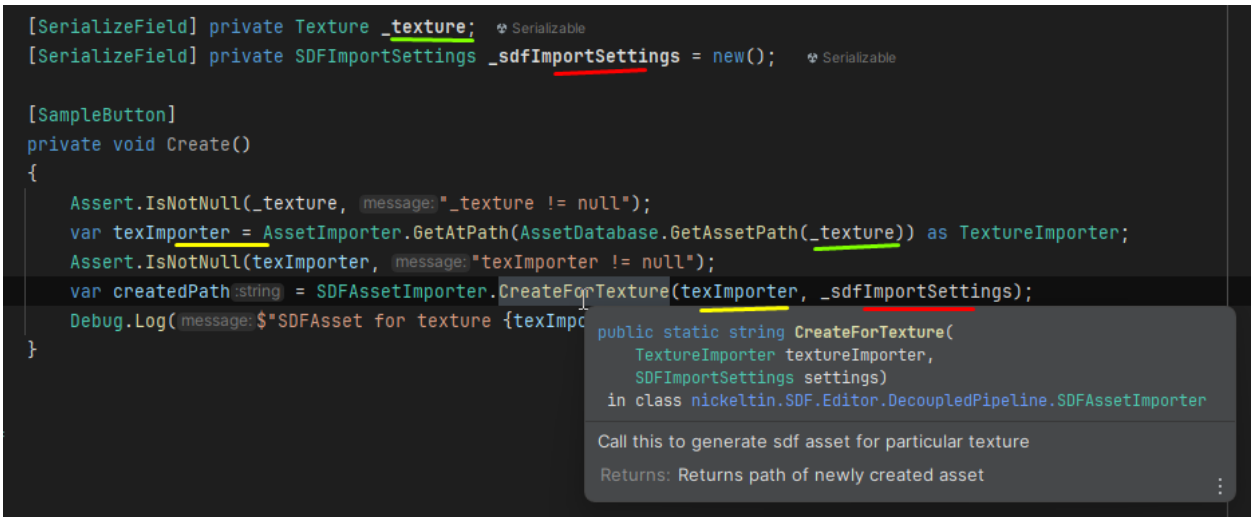


Before



After

Code



Process is pretty simple

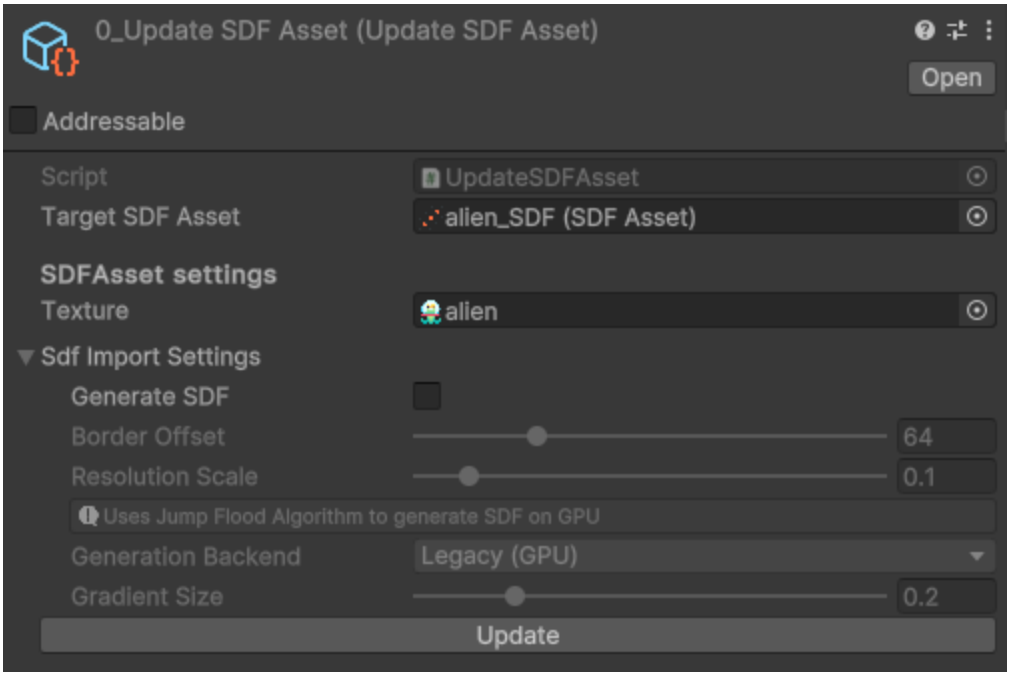
- get texture importer
  - get texture path first
- invoke `SDFAssetImporter.CreateForTexture()` with texture import and sdf import settings
  - in
    - loaded texture importer
    - desired import settings
  - return
    - path where SDFAsset was created

> 💡 At this point asset is created and imported, you can load it from its path to do further manipulations

Changing SDFAsset Settings

This sample shows how to change SDFAsset import settings.

- Try changing settings and press Update



- Texture and *Sdf Import Settings* will be pushed to *Target SDF Asset*
- Play around with settings

Code

```
[SerializeField] private SDFAsset _targetSDFAsset;  Changed in 2 assets

[Header("SDFAsset settings")]
[SerializeField] private Texture _texture;  Serializable
[SerializeField] private SDFImportSettings _sdfImportSettings = new();  Serializable

[SampleButton]
private void Update()
{
    Assert.IsNotNull(_targetSDFAsset, message:"_targetSDFAsset != null");
    var importer = SDFAssetImporter.Get(AssetDatabase.GetAssetPath(_targetSDFAsset));
    Assert.IsNotNull(importer, message:"importer != null");
    importer.Texture = _texture;
    importer.ImportSettings = _sdfImportSettings;
    importer.SaveAndReimport();
}
```

Since SDFAsset is standalone asset with custom importer, in order to change its import settings we need to load its importer.

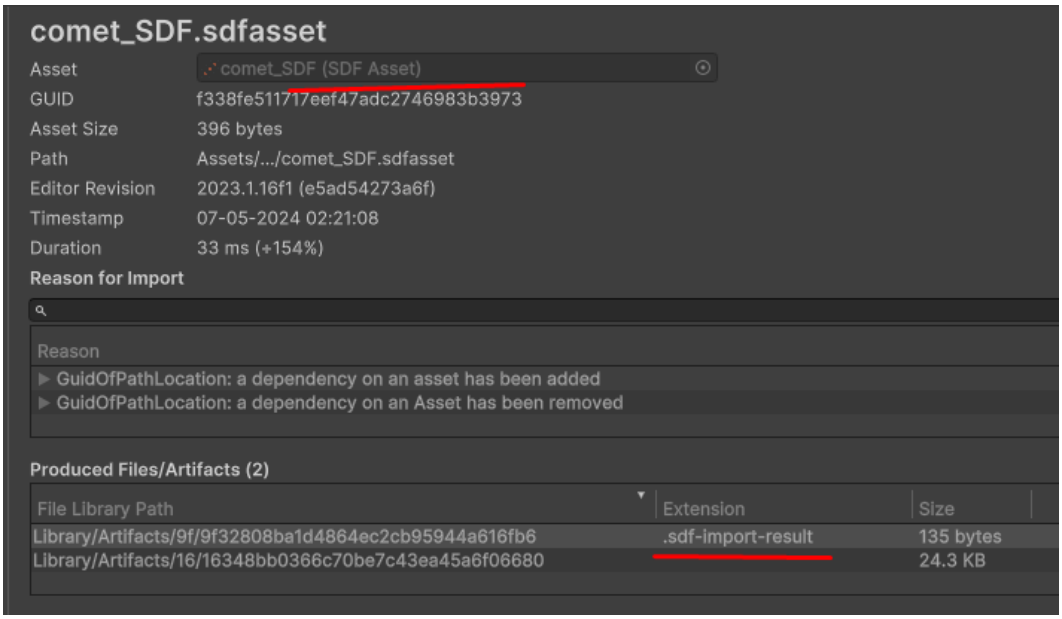
- load `SDFAssetImporter`
  - get `SDFAsset` path first
- change importer properties
  - `Texture`
  - `ImportSettings`
- save and reimport

> 👏 That's it!
> Process is the same as if you were modifying any unity asset import settings

Finding SDF Assets

Custom artifacts

There is already custom search provider that searches for `SDFSpriteMetadataAsset`. It uses custom artifacts files that can be seen in Import Activity Window.

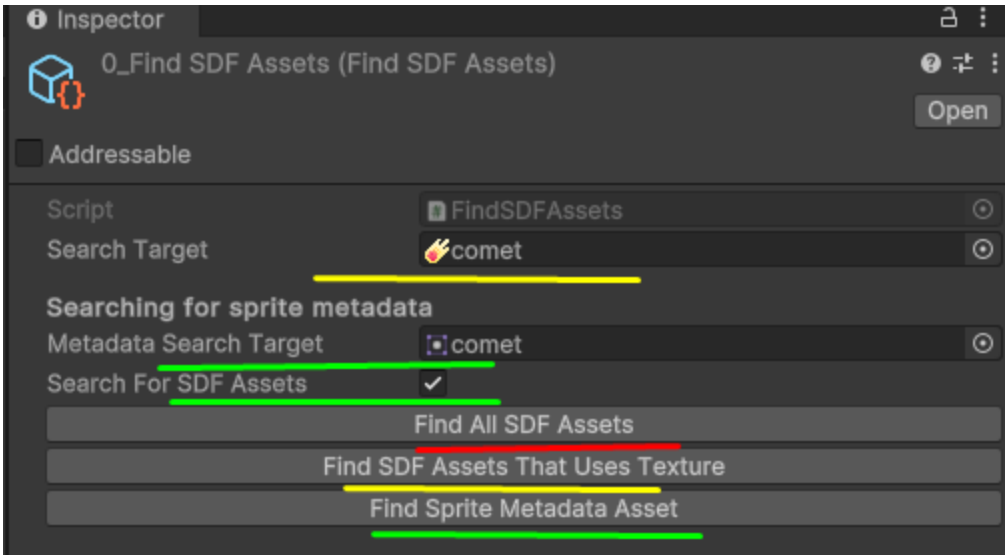


This file can be opened in text editor and will contain some post sdf generation data.

```
{
    "MetaAssetsLocalIDs":[-8445036440928941807],
    "SourceSpritesLocalIDs":[21300000],
    "SourceTextureGUID":"a214e5bb9597ff443b264d21db9bb408"
}
```

💡 Using this artifacts allows to make an efficient project-wide search for all `SDFAsset`'s that uses some particular texture.

Sample



Editor view

Find All SDF Assets

Simplest option, just a wrapper around regular `AssetDatabase.FindAssets()`, searches for all `SDFAsset`'s

```
/// Finds all <see cref="SDFAsset"/>'s
/// </summary>
/// <returns>GUID's</returns>
[SDFPipelineCompatible(SDFPipelineFlags.DecoupledAnd2023)]
2 usages  Valentyn
public static IEnumerable<string> FindAllSDFAssets()
{
    return AssetDatabase.FindAssets( filter: $"t:{typeof(SDFAsset)}");
}
```

Find SDF Assets That Uses Texture

Complex options. Will find all `SDFAsset`'s that uses provided texture.

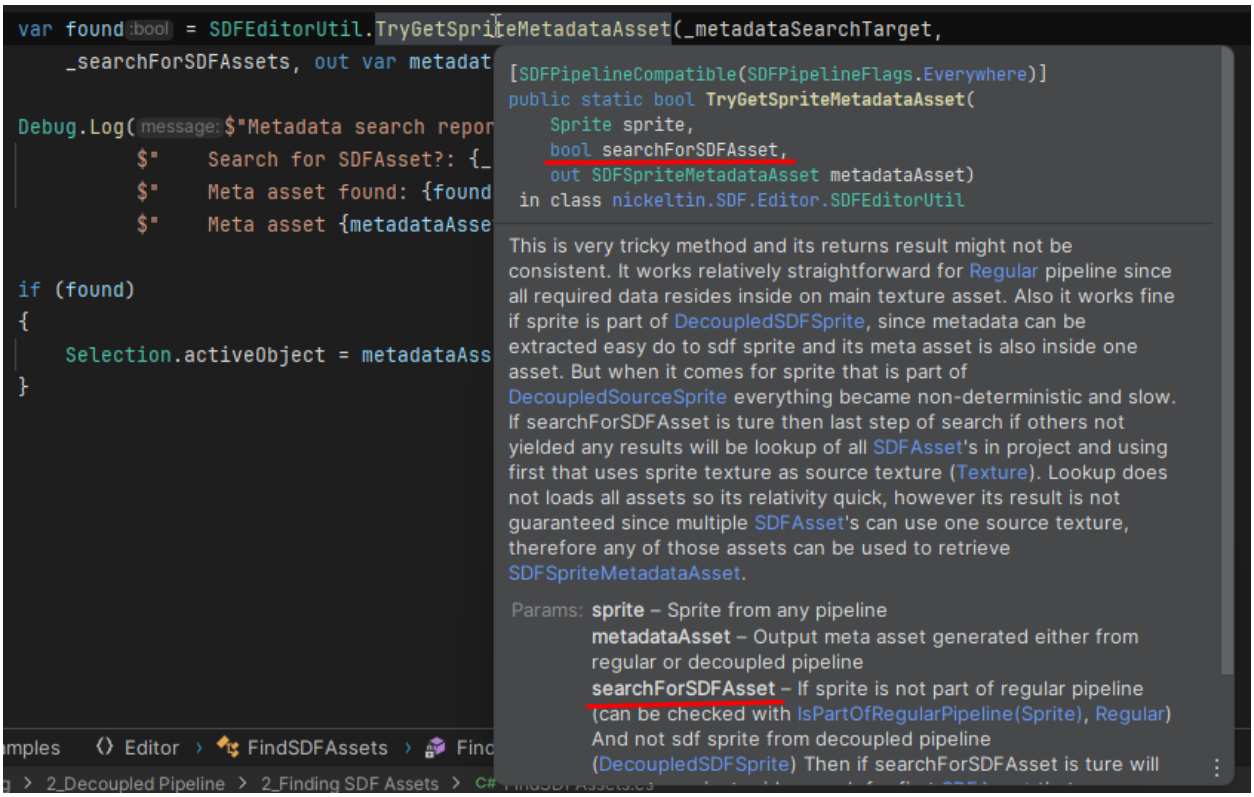
```
private void FindSDFAssetsThatUsesTexture()
{
    Assert.IsNotNull(_searchTarget, message: "_searchTarget != null");

    var texImporter = AssetImporter.GetAtPath(AssetDatabase.GetAssetPath(_searchTarget)) as TextureImporter;
    Assert.IsNotNull(texImporter, message: "texImporter != null");
    var texGUID = AssetDatabase.GUIDFromAssetPath(texImporter.assetPath).ToString();
    var sdfAssetsGUIDs :IEnumerable<string> = SDFEditorUtil.FindSDFAssetsThatUsesTexture(texGUID);
```

- get `Texture` GUID
  - get texture path
  - convert path to GUID
  - (optionally) get `TextureImporter`
- Invoke `SDFEditorUtil.FindSDFAssetsThatUsesTexture()`
  - in
    - `Texture` GUID
  - return
    - list of `SDFAsset`'s that uses texture
- Process provided GUID's
  - Load, convert to path, or just save this list somewhere

Congrats, now you know how to search for `SDFAsset`'s. This technique is quite efficient, since it's not loading any assets, just analyzes meta files and custom *.sdf-import-result artifact files.

Find Sprite Metadata Asset

Search technique above can be used to locate `SDFSpriteMetadataAsset` for any sprite in the project, if `SDFAsset`'s is true.



There is detailed explanation of how this method works in its docs, so I'll go over simplified version

`SDFEditorUtil.TryGetSpriteMetadataAsset()` signature

- in
  - Sprite
  - Search for sdf asset - set to true to ensure that at least one SDFAsset will be found

> 💡 Search for sdf assets will be the last step, since function will try to locate sdf meta asset with cheaper methods then project-wide search

- return

    - Is any `SDFSpriteMetadataAsset` was found
- out
    - found `SDFSpriteMetadataAsset`

Steps of `SDFEditorUtil.TryGetSpriteMetadataAsset()`

Function executed in several steps, working with assumptions about with what asset we are dealing. These assumptions is tested from cheap to expensive in terms of performance, if none of them was correct that means Sprite does not have sdf generated for it.

Assumptions:

- Sprite is from regular pipeline

  Correct if: Has proper import settings for regular pipeline, checked with `SDFEditorUtil.ShouldImportSDF()`
    - Finds `SDFSpriteMetadataAsset` at sprite path that matches provided sprite
- Else maybe sprite is sdf sprite generated inside of `SDFAsset` (decoupled pipeline)

  Correct if: Main asset at path is `SDFAsset`
    - Finds `SDFSpriteMetadataAsset` at sprite path that matches provided sprite
- Else we only left with option to search if it enabled

  Correct if: `SDFAsset` was found
    - Performs project wide search `SDFEditorUtil.FindSDFAssetsThatUsesTexture()`, then finds `SDFSpriteMetadataAsset` at path of found `SDFAsset` that matches provided sprite

> 👏 Congrats on figuring this function out, this probably most complex function from sdf import pipeline.
> It exist to unify sdf meta asset lookup for any sprite in project, not depending on used pipeline, or unity version.