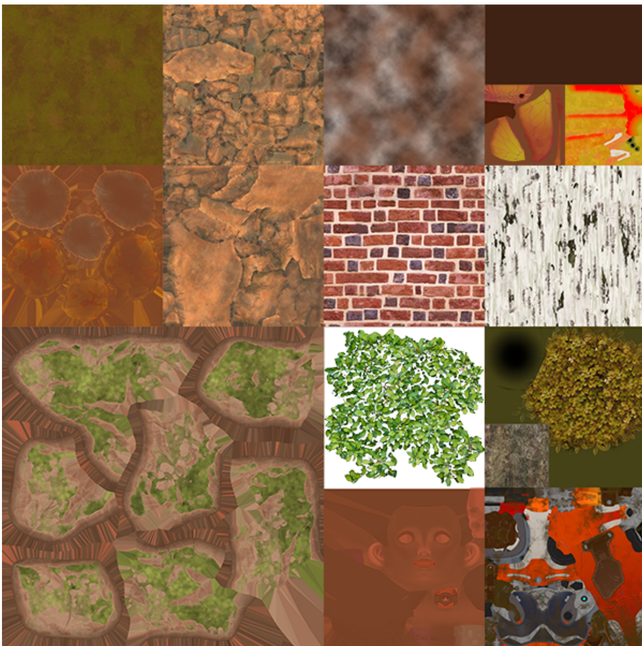
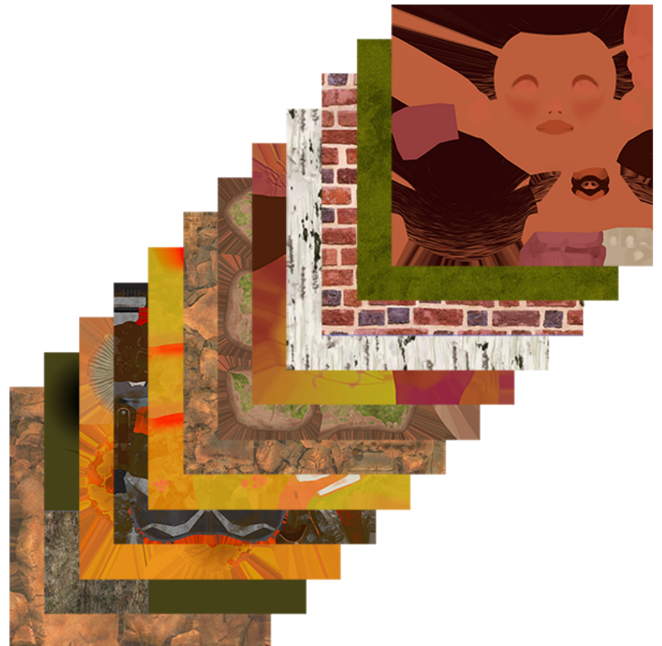
Digital Opus

MESH BAKER – TEXTURE ARRAYS

Warning: using Texture Arrays is an advanced topic than requires some programming ability, ability to modify shaders, and a basic technical understanding of rendering.

Atlases and Texture Arrays are both *"Textures"* that contain other textures. Atlases combine textures in a single, larger, texture. Texture Arrays combine textures in a stack of slices like a deck of cards (one texture per slice).





Texture Array assets can be assigned to material properties like regular Texture assets. When a shader samples a Texture Array, an extra "slice" uv coordinate is required to indicate which slice is being sampled.

```
1.  // sample a regular texture
2.  fixed2 uv = i.uv
3.  fixed4 col = tex2D(_MainTex, uv);
4.
5.
6.  // sample a texture array
7.  fixed sliceIdx = 3;
8.  fixed3 uv = fixed3(i.uv.x, i.uv.y, sliceIdx);
9.  fixed4 col = UNITY_SAMPLE_TEX2DARRAY(_MainTex, uv);
```

Advantages

- Texture Arrays can pack many more source textures than is possible to pack into Atlases.
- Texture Arrays can support unlimited UV tiling. Atlases cannot tile without baking the tiling (which uses up a lot of space).

Limitations

- All slices must have the same height and width dimension. If a Texture Array contains textures of different sizes (128, 256, 512, 1024, etc) these will all be scaled to the dimensions of the Texture Array. Smaller textures will be scaled up and larger textures will be scaled down. Note that Mesh Baker can pack small textures into a single slice. Each slice can be a small atlas. This can help reduce the size of Texture Arrays.
- When using Texture Arrays in place of Textures, Shaders must be modified. Out-of-the-box Shaders included with Unity will not work.
- Unity does not include a Texture Array importer that will automatically set the correct compression for different platforms. When building for multiple platforms, you may need to bake a separate texture array for each platform with a different compression setting and write a script to assign the correct texture array to materials when building.
- Combining Textures with different channels (eg. RGB, RGBA) may not work well. The same goes for Textures with different filter mode, wrap mode and mip settings. The generated texture array can only have one value for each of these settings.
- Combining source Textures with different compression formats (eg. ETC_RGB vs. ETC_RGBA) can result in combined meshes that look slightly different (the result texture can only have one compression setting).
- Texture Arrays may require the slice index to be inserted into a mesh coordinate. This is difficult to achieve and maintain without a tool like Mesh Baker.

How To Use Texture Arrays

Modifying Shaders To Use Texture Arrays

Using Texture Arrays usually requires two versions of each shader:

1. One that uses regular Texture2D so that artists can use regular photo manipulation programs to create and modify Texture assets.
2. Another that uses Texture2DArrays for optimizing a scene.

The Unity Manual describes how to modify shaders to use Texture Arrays in the section Texture Arrays. Mesh Baker includes an example:

MeshBaker -> Examples -> UnlitTextureArray

An example Standard Shader that is modified to use Texture Arrays can be found at https://github.com/Phong13/BuiltinShadersTextureArrays. This has been distributed as a .git project so that it is easy to see the modifications that have been made by comparing the current code with the original check-in.

Creating Texture Arrays With Mesh Baker

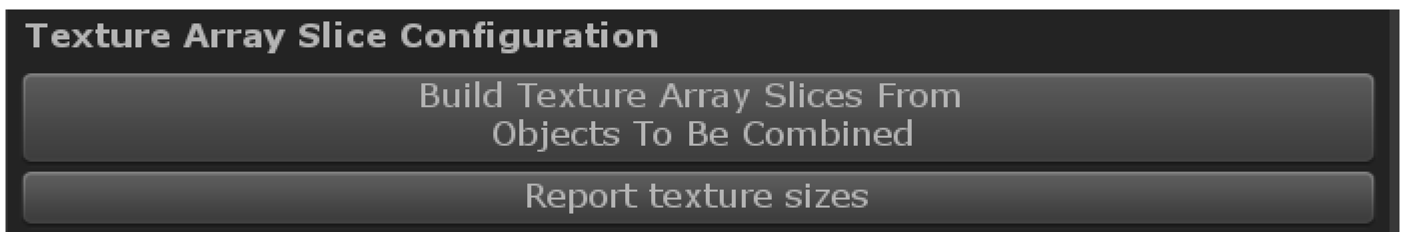
Typical workflow:

1. Add a MB3_TextureBaker to a scene
2. Add “*Objects To Be Combined*”
3. Set *“Result Type”* to *“Texture Array”*
4. *“Create Empty Assets For Combined Materials”*
5. Set the shader on the combined material to the Texture Array version of your shader.
6. Map source materials to Texture Array slices
7. Create one or more output formats
8. Adjust texture baking settings
9. Bake the textures

Only steps 5 and 6 are significantly different from the normal Texture baking process.

Map Source Materials To Texture Array Slices

The easiest way to configure slices is to use the button: *“Build Texture Array Slices From Objects To Combine”*.



For best results, set “*Max Atlas Size*” to the desired height/width of your Texture Array before using the button. All materials on the source models will be organized into slices. Textures smaller than the “*Max Atlas Size*” may be grouped into atlas-slices. You can modify the configuration after it has been created or delete the generated slices and re-create them.

The "*Report Texture Sizes*" button generates a list of all textures used by the slices, their formats, and their sizes. You may choose to remove some objects from the configuration if their textures would be downsized unacceptably, or if they are in a format that is not compatible with other textures due to the wrap mode, channels, compression format, or size.

Slices can contain more than one texture / source material. Slices containing more than one texture will be baked into atlas-slices.

Slices can "*Consider UVs*". This can be useful for extracting small parts of a large source atlas if props only use a small part. "*Consider UVs*" should NOT be used for props with large amounts of UV tiling. Props with large amounts of UV tiling should be put on a slice of their own (that does NOT use "*Consider UVs*") so that they are free to tile.

Slices that use "*Consider UVs*" need a source renderer reference as well as a source material reference.

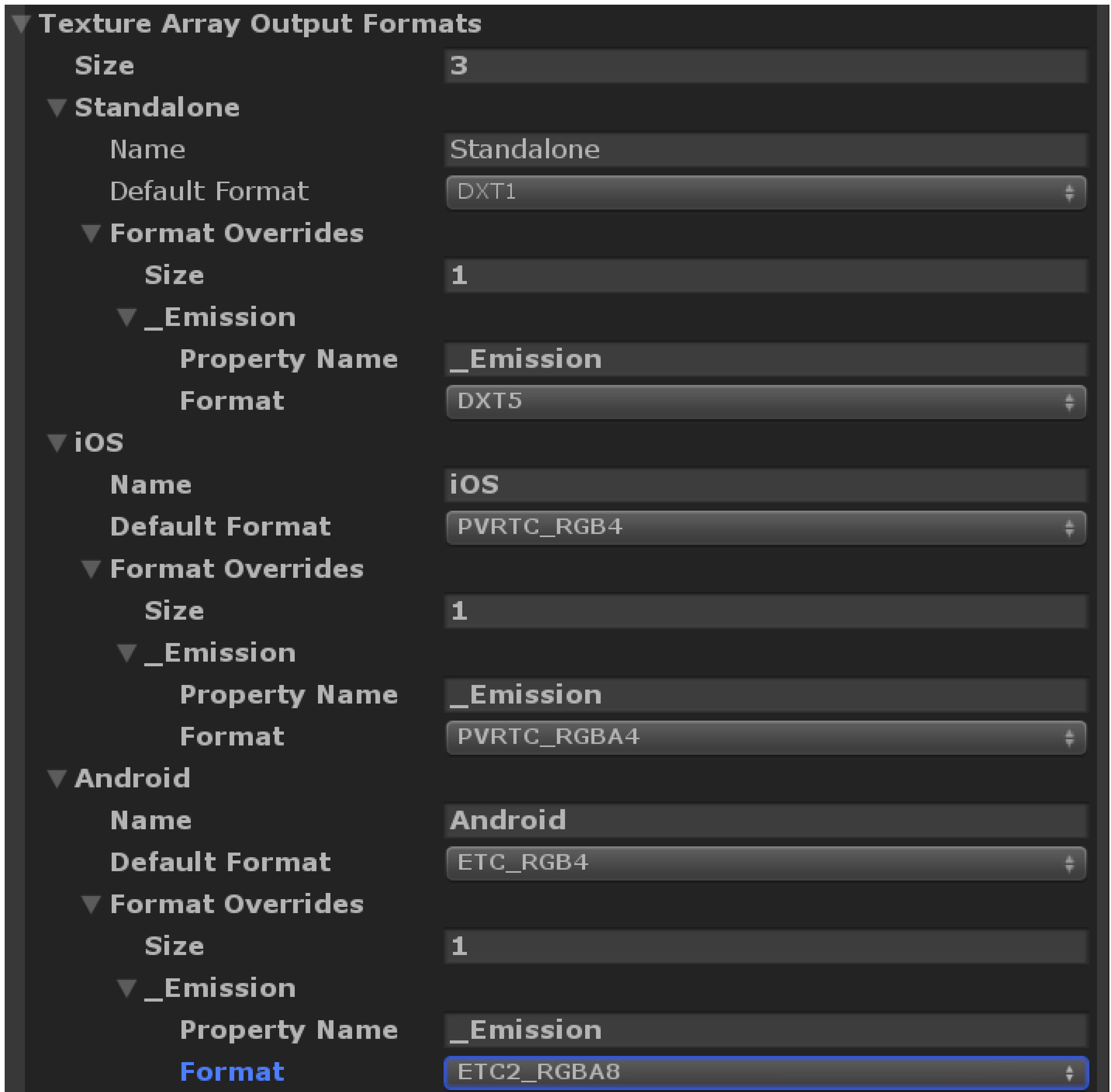
Create One Or More Output Formats

To get good performance from GPUs it is necessary to compress Texture Arrays. Unfortunately, most platforms require different texture compression formats.

Unity does not provide a way to switch the compression format of a texture array once it has been created. This means that a separate Texture Array must be created for each required compression format / build platform. Mesh Baker lets you define a set of formats. A copy of the Texture Array is generated for each format. Assign a "*Name*" (usually be the name of the build platform) for each format.

Sometimes there is a need to override the format for some material texture properties. For example, most of the texture properties of the Standard shader need three channel textures (RGB). However, the Emission texture property requires a fourth Alpha channel (RGBA). This can be handled by providing a format override for the Emission property.

An example format setup for this might look like:



Modifying Meshes To Use Texture Arrays

How will different meshes know which slice(s) to use in the Texture Array? The easiest solution is to embed the slice-index into the baked mesh. The slice-index can be added as a UV coordinate, as a vertex colour, or as another unused mesh coordinate. Mesh UV channels can have an additional (z) coordinate that can store the slice index. One advantage of this technique is that submeshes (multiple materials) can be collapsed into a single sub-mesh / material. Each triangle can sample a different slice.

Note that it is also possible to use GPU instancing and to assign slice-indexes to instanced material properties. However, this technique is too advanced to explain here, and does not handle collapsing

submeshes (multiple materials).

Encode The Slice Index Into The Baked Mesh

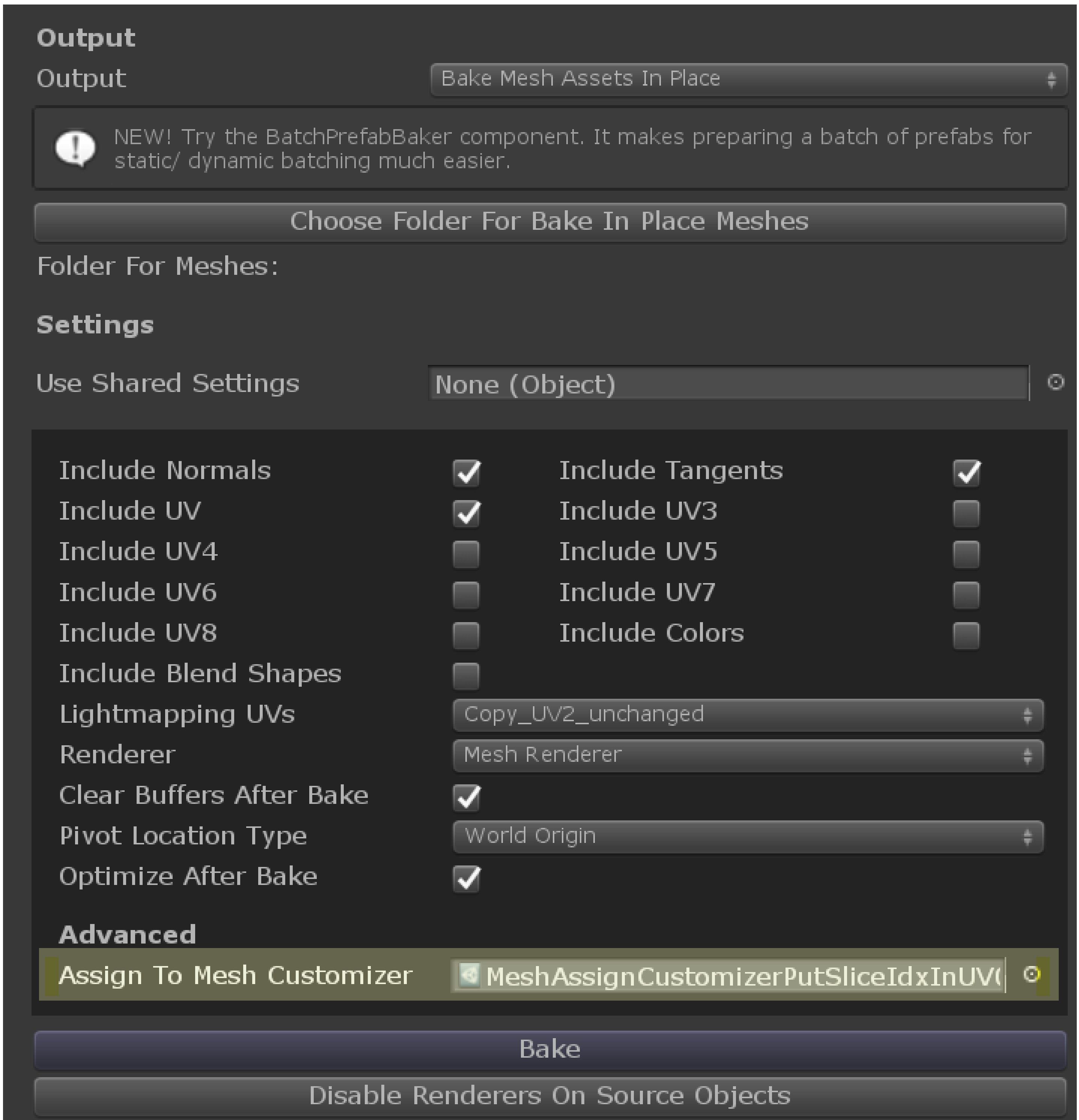
The slice-index can be assigned to a mesh's UV coordinates, Vertex Colors, or other unused channel.

Mesh Baker includes a script that will bake the slice-index into the UV.z coordinate. You can also create a custom script that will plug into Mesh Baker to assign the slice-index to a custom mesh coordinate.

To use the script included with Mesh Baker, create a *"Mesh Customizer"* asset that will assign the slice-index to UV0.z using the project view context menu:

Create -> Mesh Baker -> Assign To Mesh Customizer -> Put Slice Index In UV0.z

This will create a new asset in the project. Drag this asset to the *MB3_MeshBaker(inspector) -> Assign To Mesh Customizer* field.



To create a *customizer* script, look at the included example scripts in the AssignToMeshCustomizers folder (example: CustomizerPutSliceIndexInUV0_z.cs). Create a similar script (must inherit from *"MB_ DefaultMeshAssignCustomizer")* that puts the slice-index into the desired mesh coordinate.

Switching Texture Arrays When Building For Different Platforms

Each Platform (OSX, Windows, Android iOS etc) uses different texture compression formats. Unlike regular Textures, Texture Array assets in Unity do NOT automatically switch compression format when

the build platform is switched. If you are using Texture Arrays and the project supports multiple platforms, you will need to write a custom build script that will switch Texture Arrays when creating a build or creating asset bundles.

Here is some example code that uses the data stored in the TextureBakeResult to switch Textures Arrays on a combined material. This code must be in an Editor folder or an Editor Assembly.

```
1.  using UnityEngine;
2.  using UnityEditor;
3.  using UnityEditor.Build;
4.  using UnityEditor.Build.Reporting;
5.  using DigitalOpus.MB.Core;
6.
7.  class MyCustomBuildProcessor : IPreprocessBuildWithReport
8.  {
9.    public int callbackOrder { get { return 0; } }
10.
11.   public void OnPreprocessBuild(BuildReport report)
12.   {
13.     Debug.Log("Running custom build script");
14.
15.     // References to TextureArrays
16.     // in all the formats are stored
17.     // in the TextureBakeResult.
18.
19.     // Load a TextureBakeResult.
20.     // It would be better
21.     // to store a list of
22.     // TextureBakerResults in a
23.     // ScriptableObject asset and process
24.     // each of those instead of loading from
25.     // a hardcoded path
26.
27.     string pathToTextureBakeResult = "INSERT PATH HERE";
28.     MB2_TextureBakeResults bakeResult =
29.       AssetDatabase.LoadAssetAtPath(
30.           pathToTextureBakeResult);
31.
32.     ProcessTextureBakeResult(bakeResult);
33.   }
34.
35.     void ProcessTextureBakeResult(
36.       MB2_TextureBakeResults bakeResult)
37.     {
38.       BuildTarget buildTarget =
39.          EditorUserBuildSettings.activeBuildTarget;
40.
41.     Debug.Assert(bakeResult.resultType ==
42.         MB2_TextureBakeResults.ResultType.textureArray);
43.     // Visit each Combined Material in the Texture Bake Result
44.     foreach (MB_MultiMaterialTexArray resMat
```

```
45.            in bakeResult.resultMaterialsTexArray)
46.      {
47.        // Visit each texture property
48.        // in the combined material
49.        foreach(
50.            MB_TexArrayForProperty texProperty in
51.            resMat.textureProperties)
52.        {
53.          Texture2DArray ta = null;
54.          // Get the texture array
55.          // in the format matching the platform
56.          // For this to work we need the
57.          // "name" of the format to match the
58.          // build platform.
59.          bool foundAPlaform = false;
60.          foreach (MB_TextureArrayReference format in
61.              texProperty.formats)
62.          {
63.              if (format.texFromatSetName.Equals(
64.                  buildTarget.ToString()))
65.              {
66.                  foundAPlaform = true;
67.                  ta = format.texArray;
68.                  break;
69.              }
70.          }
71.
72.          if (foundAPlaform)
73.          {
74.            resMat.combinedMaterial.SetTexture(
75.              texProperty.texPropertyName, ta);
76.          } else
77.          {
78.            Debug.LogError(
79.                "Could not find a TextureFormat entry" +
80.                "Matching platform '" +
81.                  buildTarget.ToString() + "'" +
82.                " Names must match exactly.");
83.          }
84.        }
85.      }
86.    }
87.  }
```