Section One: Automatic folder Creation

Folder creation and organization in a project is critical to a projects success. In small projects, organization can be easily overlooked. However in a larger project with a larger team you can easily loose objects, scripts, and other important items. A simple but effective project structure will be created in this section. Keep in mind that a project structure has to be understood by all members on a team for it to be effective.

Knowing how you are going to organize your project is only half the battle. Once you know how you are going to organize it, you have to create all of the folders. Instead of doing this manually, you are going to create a utility to do this for you.

Folder Structure

The folder structure for this class is outlined below.

- Assets
  - Dynamic Assets
    - Resources
      - Animations
        - Sources
      - Animation Controllers
      - Effects
      - Models
        - Character
        - Environment
      - Prefabs
        - Common
      - Sounds
        - Music
          - Common
        - SFX
          - Common
      - Textures
        - Common
      - folderStructure.txt
  - Editor
    - folderStructure.txt
  - Extensions
    - folderStruction.txt
  - Gizmos
    - folderStructure.txt
  - Plugins
    - folderStructure.txt
  - Scripts
    - Common
    - folderStructure.txt
  - Shaders
  - Static Assets
    - Animations
      - Sources
    - Animation Controllers
    - Effects
    - Models
      - Character
      - Environment
    - Prefabs
      - Common
    - Scenes
    - Sounds
      - Music
        - Common
      - SFX
        - Common
    - Textures
      - Common
      - folderStructure.txt
  - Testing
    - folderStructure.txt
  - folderStructure.txt

Dynamic Assets

Dynamic Assets are assets that are loaded into the game via Resources.Load(…). In other words, dynamic assets are assets that are placed into the game during runtime. If you have an asset that is both dynamic and static, put it into the dynamic assets folder.

Inside of the dynamic assets folder, there is a second organizational structure. This structure is catered to animators and musicians who will need access to your files.

When an object is imported into Unity, if it is animated it creates an object with the animations nested inside of it. This is not always optimal. Instead, you want to rip the animations off of the object and have it stored somewhere that you can reference.

If you had an object named Space-Ship and you wanted to import it into the Dynamic Assets folder, you would follow these steps:

1. Create a new folder under Animations/Sources with the name of the object
   a. Animations/Sources/Space-Ship
2. Create a new folder under Animations with the name of the object
   a. Animations/Space-Ship
3. Import the object to the appropriate animations/sources folder
   a. Animations/Sources/Space-Ship
4. Copy the .anim file that is nested inside of the object from the sources folder to the animations folder
   a. From Animations/Sources/Space-Ship to Animations/Space-Ship
5. Create the animation controller for the object, naming it appropriately.
6. Place the animation controller in the AnimationControllers folder.

The Effects folder is where you will place particle effects. Create a new folder for each particle effect.

The Models folder is for raw models. These are for single looping-animation models or for non-animated models.

The Prefabs folder is for all prefabs. A new folder should be created for each level/world/scene the prefab is used in. Any prefab used in multiple levels/worlds/scenes should be placed in the common folder. Only one copy of each prefab should exist.

The Sounds folder is for all sounds. It should follow the same organization as the prefabs folder.

The Textures folder is for all textures that are applied separately to models or changed at runtime.

Editor

The Editor folder has to be created manually. This is where you place Editor scripts. The organization of this folder is project-dependent.

Extensions

This is a folder for third party assets such as asset packages. The exception to this is the standard assets folder. The standard assets folder should be left in the default location.

Gizmos

This is a folder for gizmo scripts.

Plugins

This is a folder for plugin scripts.

Scripts

This is a folder for all other scripts. It should be separated by common scripts found across multiple objects, and then scripts by level or by type.

Shaders

This is a folder for all shader scripts. There is no inherent organizational structure for this folder.

Static Assets

Static Assets are all remaining assets (including scenes) that are not loaded into the game at runtime.

Inside of the static assets folder, there is a second organizational structure. This structure is catered to animators and musicians who will need access to your files.

When an object is imported into Unity, if it is animated it creates an object with the animations nested inside of it. This is not always optimal. Instead, you want to rip the animations off of the object and have it stored somewhere that you can reference.

If you had an object named Space-Ship and you wanted to import it into the Static Assets folder structure, you would follow these steps;

1. Create a new folder under Animations/Sources with the name of the object
   a. Animations/Sources/Space-Ship
2. Create a new folder under Animations with the name of the object
   a. Animations/Space-Ship
3. Import the object to the appropriate animations/sources folder
   a. Animations/Sources/Space-Ship
4. Copy the .anim file that is nested inside of the object from the sources folder to the animations folder
   a. From Animations/Sources/Space-Ship to Animations/Space-Ship
5. Create the animation controller for the object, naming it appropriately.
6. Place the animation controller in the AnimationControllers folder.

The Effects folder is where you will place particle effects. Create a new folder for each particle effect.

The Models folder is for raw models. These are for single looping-animation models or for non-animated models.

The Prefabs folder is for all prefabs. A new folder should be created for each level/world/scene the prefab is used in. Any prefab used in multiple levels/worlds/scenes should be placed in the common folder. Only one copy of each prefab should exist.

The Sounds folder is for all sounds. It should follow the same organization as the prefabs folder.

The Textures folder is for all textures that are applied separately to models or changed at runtime. This folder is also used for 2D sprites, and UI textures.

Step One: Editor folder

Any script that edits the Unity Editor (by adding new menu items, creating custom inspectors and panels, and various other edits) needs to be placed in the Editor folder. This Editor folder is a special folder that Unity will not build into your actual game. It only exists when the Unity Editor is running.

The Editor folder is also case sensitive, so when you create the folder you must name it exactly “Editor”.

Step Two: Create Script

The next thing you want to do is create a new script that you will use to create your folders. You can name this script whatever you want, as long as you follow the typical naming conventions.

Step Three: Start Editing

Once the script is created, there are a few goals to accomplish. In this script, you want to:

- Add a menu item to the toolbar at the top of Unity to access this script.
- Create multiple folders.
- Include text documents that are prepopulated with how the structure of the project is going to be.

In this example, you are going to use a shorter version of the folder structure that you should be familiar with

- Assets
  - Materials
    - folderStructure.txt
  - Textures
    - folderStructure.txt
  - Prefabs
    - folderStructure.txt
  - Scripts
    - folderStructure.txt
  - Scenes
    - folderStructure.txt
  - Animations
    - AnimationControllers
    - folderStructure.txt
  - folderStructure.tx

Step Four: Creating a Menu Item

You will need to create a menu item in the Unity Toolbar called "Project Tools". This will allow you to click on "Project Tools" and automatically run the script. In order for you to do that, you must tell Unity that you are going to be editing the Unity Editor itself. You need to add the using UnityEditor; statement to the top of your script.

```
001 using UnityEngine;
002 using System.Collections;
003 using UnityEditor;
```

To add a menu item to your Editor, you need to include the attribute MenuItem(string args).

The “string args” part of this attribute is for the name of the nested menu item you wish to add. You cannot add a menu item without it being nested.

For example, one of the menu items that already exists is Game Object > Create Empty. If you were to recreate this menu item, you would do:

```
001 using UnityEngine;
002 using System.Collections;
003 using UnityEditor;
004
005 public class ToolCreation : MonoBehaviour {
006
007     [MenuItem("Game Object/Create Empty")]
```

The Create Empty option has a hotkey shortcut: Ctrl + Shift + N

You can define keyboard shortcuts for custom menu items as well, by adding special key-codes at the end of your menu item definition.

To create a hotkey, you can use the following special characters:

- % (ctrl on Windows, cmd on OS X)
- # (shift)
- & (alt)
- _ (no key modifiers).

For example, to create a menu with the hotkey shift-alt-g you would use "MyMenu/Do Something #&g". To create a menu with a hotkey of g and no key modifiers pressed you would use "MyMenu/Do Something _g".

Some special keyboard keys are supported as hotkeys. For example, "#LEFT" would map to shift-left. The keys supported like this are:

- LEFT
- RIGHT
- UP
- DOWN
- F1 .. F12
- HOME
- END
- PGUP
- PGDN

A hotkey text must be preceded with a space character ("MyMenu/Do_g" won't be interpreted as hotkey, while "MyMenu/Do _g" will).

Example: To recreate the Empty Game Object menu item with the shortcut Ctrl + Shift + N:

```
001 using UnityEngine;
002 using System.Collections;
003 using UnityEditor;
004
005 public class ToolCreation : MonoBehaviour {
006
007     [MenuItem("Game Object/Create Empty %#n")]
```

Recreating a menu item that already exists isn't going to help you very much.

You will need to create a new menu item, and nest it under a new section on the toolbar. You will call it "Create folder" under the section "Tool Creation".

```
001 using UnityEngine;
002 using System.Collections;
003 using UnityEditor;
004
005 public class ToolCreation : MonoBehaviour {
006
007     [MenuItem("Tool Creation/Create folder")]
```

Step Five: Attach an Action to the Menu

At this point in time, this menu item does not do anything. If you click it, nothing will happen.

In order to attach an action to a menu item, you need to create a function immediately below the menu item. All actions have to be declared as `public static void.`

Be advised: These are cascading effects! For example if I do this:

```
001 using UnityEngine;
002 using System.Collections;
003 using UnityEditor;
004
005 public class ToolCreation : MonoBehaviour {
006
007     [MenuItem("Tool Creation/Create folder")]
008     [MenuItem("Tool Creation 2/Create folder 2")]
009     public static void Createfolder()
010     {
011         Debug.Log("Creating a folder!");
012     }
013 }
```

If you click either of the menu items, they will both run the Create folder action!

Now that you have a menu item, and an action that will occur when the menu item is clicked, you can start to make your folders.

Step Six: Creating folders

In order to create a folder, you will use AssetDatabase.Createfolder(string parentfolder, string newfolderName). This will create a folder as a child to the parentfolder, with the name of newfolderName. For all folders that are not a child, you must use "Assets" as the parent folder. For all folders that are a child, you will use "Assets/Parentfolder" as the parent folder.

To create your folder hierarchy, you will use the following code snippet:

```
001 using UnityEngine;
002 using System.Collections;
003 using UnityEditor;
004
005 public class ToolCreation : MonoBehaviour {
006
007     [MenuItem("Tool Creation/Create folder")]
008     public static void Createfolder()
009     {
010         AssetDatabase.Createfolder("Assets","Materials");
011         AssetDatabase.Createfolder("Assets","Textures");
012         AssetDatabase.Createfolder("Assets","Prefabs");
013         AssetDatabase.Createfolder("Assets","Scripts");
014         AssetDatabase.Createfolder("Assets","Scenes");
015         AssetDatabase.Createfolder("Assets","Animations");
016         AssetDatabase.Createfolder("Assets/Animations","AnimationControllers");
017     }
018 }
```

Step Seven: Creating folderStructure.txt

The next step is to add in the files that explain the structure of each folder, as well as the structure of the project as a whole. You will want to create a file in the appropriate directory, write the text information, and then close the file. If you remember from the C# class, you can use System.IO to create file streams. In order to utilize items in the System.IO, you need to add the using directive using System.IO.

You need a way to access the folder location, and you also want to be able to use this action regardless of what computer you are running on. Therefore, you cannot just hardcode the path (C:/Users/Student/Tools Project/Assets/Textures). Instead, you want to generate the path to the assets folder, and then navigate from there.

In order to get the path to the assets folder (which only exists outside of a build) you can use Application.dataPath. This function will give you the path to the assets folder if you are in the Unity Editor. Using that, we can add the rest of the folder structure to locate our file.

For example:

```
string folderLocation = Application.dataPath + "/Materials";
```

Then, to create your text we are going to use a function that will create the file, write to the file, and close the file all in one. This function is System.IO.File.WriteAllText(string path, string contents). In order to use this function, you need to know where to create the file, and what to write into the file. You can get the location of the assets folder using Application.dataPath, and then tack on the rest of the location. As for what to write, that depends on the folder. Materials: This folder is for storing materials.

Textures: This folder is for storing textures.

Prefabs: This folder is for storing prefabs.

Scripts: This folder is for storing scripts.

Scenes: This folder is for storing scenes.

Animations: This folder is for storing raw animations.

AnimationControllers: This folder is for storing animations

```
controllers.System.IO.File.WriteAllText(Application.dataPath+"/Materials/folderStructure.txt", "This folder is for storing materials!");
```

Step Eight: Refreshing the Project

Now that we have created a bunch of folders, and added new text files explaining their organization, the last step is to update our project to reflect these changes using the statement AssetDatabase.Refresh()

```
AssetDatabase.Refresh();
```

Step Nine: On Your Own

Create a script that creates the editor folder structure outlined at the beginning of this section. Make sure to include text in each of the files!

Wrap-Up

You now have the tools to:

- Create a script that effects the Editor
- Add a new item to the toolbar, with keyboard shortcuts
- Create a new folder within a script
- Create a .txt with a script
- Write text to a file

Definitions

ASSETDATABASE.CREATEFOLDER(STRING PARENTFOLDER, STRING NEWFOLDERNAME)

public static string Createfolder(string parentfolder, string newfolderName);

Parameters

parentfolder	The name of the parent folder
newFolderName	The name of the new folder.

Description

Creates a new folder.

ASSETDATABASE.REFRESH()

Import any changed assets.

This will import any assets that have changed their content modification data or have been added-removed to the project folder.

This method implicitly triggers an asset garbage collection (see Resources.UnloadUnusedAssets).

APPLICATION.DATAPATH

Contains the path to the game data folder (Read Only)

MENU ITEM (STRING ARG) : UNITY EDITOR

The MenuItem attribute allows you to add menu items to the main menu and inspector context menus.

The MenuItem attirbute turns any static function into a menu command. Only static functions can use the MenuItem attribute.

To create a hotkey you can use the following special characters:

- % (ctrl on Windows, cmd on OS X)
- # (shift), & (alt)
- _ (no key modifiers)

For example to create a menu with hotkey shift-alt-g use "MyMenu/Do Something #&g". To create a menu with hotkey g and no key modifiers pressed use "MyMenu/Do Something _g".

Some special keyboard keys are supported as hotkeys, for example "#LEFT" would map to shift-left. The keys supported like this are:LEFT

- RIGHT
- UP
- DOWN
- F1 .. F12
- HOME
- END
- PGUP
- PGDN

A hotkey text must be preceded with a space character ("MyMenu/Do_g" won't be interpreted as hotkey, while "MyMenu/Do _g" will).

SYSTEM.IO.FILE.WRITEALLTEXT(STRING PATH, STRING CONTENTS)

public static void WriteAllText(string path, string contents)

Creates a new file, writes the specified string to the file, and then closes the file. If the target file already exists, it is overwritten.

USING SYSTEM.IO

The System.IO namespace contains types that allow reading and writing to files and data streams, and types directory support.

USING UNITYEDITOR

This provides the programmer with the ability to use various Unity Editor functions and classes.

Complete Code

```
001 using UnityEngine;
002 using System.Collections;
003 using UnityEditor;
004
005 public class ToolCreation : MonoBehaviour {
```

```
006   [MenuItem("Tool Creation/Create folder")]
007   public static void Createfolder()
008   {
009      //Create a folder named Materials in the Assets folder
010      AssetDatabase.Createfolder("Assets","Materials");
011      //Create a file named "folderStructure.txt" in the Assets/Materials folder
012      System.IO.File.WriteAllText (Application.dataPath + "/Materials/folderStructure.txt", "This folder is for storing materials!");
013
014      //Create a folder named Textures in the Assets folder
015      AssetDatabase.Createfolder("Assets","Textures");
016      //Create a file named "folderStructure.txt" in the Assets/Textures folder
017      System.IO.File.WriteAllText (Application.dataPath + "/Textures/folderStructure.txt", "This folder is for storing textures!");
018
019      //Create a folder named Prefabs in the Assets folder
020      AssetDatabase.Createfolder("Assets","Prefabs");
021      //Create a file named "folderStructure.txt" in the Assets/Prefabs folder
022      System.IO.File.WriteAllText (Application.dataPath + "/Prefabs/folderStructure.txt", "This folder is for storing Prefabs!");
023
024      //Create a folder named Scripts in the Assets folder
025      AssetDatabase.Createfolder("Assets","Scripts");
026      //Create a file named "folderStructure.txt" in the Assets/Scripts folder
027      System.IO.File.WriteAllText (Application.dataPath + "/Scripts/folderStructure.txt", "This folder is for storing Scripts!");
028
029      //Create a folder named Scenes in the Assets folder
030      AssetDatabase.Createfolder("Assets","Scenes");
031      //Create a file named "folderStructure.txt" in the Assets/Scenes folder
032      System.IO.File.WriteAllText (Application.dataPath + "/Scenes/folderStructure.txt", "This folder is for storing Scenes!");
033
034      //Create a folder named Animations in the Assets folder
035      AssetDatabase.Createfolder("Assets","Animations");
036      //Create a file named "folderStructure.txt" in the Assets/Animations folder
037      System.IO.File.WriteAllText (Application.dataPath + "/Animations/folderStructure.txt", "This folder is for storing raw Animations!");
038
039      //Create a folder named AnimationControllers in the Animations folder
040      AssetDatabase.Createfolder("Assets/Animations","AnimationControllers");
041      //Create a file named "folderStructure.txt" in the Assets/Animations/AnimationControllers folder
042      System.IO.File.WriteAllText (Application.dataPath + "/Animations/AnimationControllers/folderStructure.txt", "This folder is for storing Animation Co
043
044      //Refresh the project structure to commit all changes
045      AssetDatabase.Refresh();
046   }
047 }
```

Post-Lab

1. What is the purpose of the Editor folder?
2. Menu items can be added to the inspector using what statement?
3. What must you watch out for when adding menu items?
4. How do you attach an action to a menu item?
5. What is the purpose of AssetDatabase.Refresh()?
6. What is the difference between a static asset and a dynamic asset?

7. Why is it important to keep static and dynamic assets in different folders?
8. Think about your workflow, what would be your optimal folder structure/organization?