USER GENERATED CONTENT

CONTENTS

INTRODUCTION

The user generated content unit provides the student with an introduction to opening up their games for users to edit after the game has been built.

User generated content is getting a huge boost in the market, and taps into a significant amount of potential by allowing the community to contribute and feel powerful over their game. Many games have implemented different types of user generated content. In the past, this was limited to modding games. By opening up small portions of their engine and creating tools for users to edit those portions users could make small changes to the game within set parameters.

With technology advancing, and the userbase of games expanding, it becomes more important to engage the community and allow them to have creative outlets in games.

LEARNING OBJECTIVES

User generated content has the potentional to be used in any game. And it is not always certain if the person generating the content is a beginner, intermediate user, or advanced user.

Upon completion of this unit, the student will be able to extract, save, and import data that has been modified outside of the engine. This data will then be used to create a console display. The student will also observe differnet methods for how tools are constructed, and learn how to determine what parts of an engine need to be exposed for user generated content to be created.

READING

<Introduction reading>

Section One

<Section reading information>

THINK ABOUT IT!

1. This is where homework questions will be.

HANDS-ON

<snippet/introduction if needed>

Section One: Wheel of Fortune

In this section you will look at how to read and write data into a game, and have it affect gameplay.

In this game, the user is supplied with a series of blank spots where letters can go. The user then guesses the letters that belong in the sentence or phrase by clicking on buttons corresponding to the letters of the alphabet. For each letter they guess correctly, they get one point. For each letter they guess incorrectly, they loose one point. If they run out of points, they loose the game. The user is also supplied a hint that pertains to what the sentence or phrase is about.

STEP ONE: SETUP

For the setup of this lab you will need to download the supplied Wheel of Fortune game. Play it a few times, and take a look at the code so you understand what is happening in the game before moving on to the user generated content section.

STEP TWO: FILE STRUCTURE

The next step is to determine the file structure for the imported file you will use in your game. In order to determine the file structure, you need to determine what your engine needs to accomplish the goals. For this game, the goal is to allow the player to input their own sentences/phrases and clues/hints for the game. It would be nice if they could put multiple sentences and clues, so they can play the game with a friend for long periods of time (as opposed to just a single sentence and clue).

With those goals in mind, you can start to determine how you want your file to be set up. Since you are only looking for two pieces of information from the user (a sentence and a phrase) then it would make the most sense to have each on it's own line.

Knowing that each sentence and clue will be on it's own line presents a new challenege, however. How will you determine which is a sentence, and which is a clue?

The easiest way is to alterate. For example, each even line in the file is a clue, and each odd line in the file is a sentence. You can then go on to say that the line immediately preceding a clue is the sentence that goes along with that clue (or the other way around!).

For this lab, the heuristic for the file will be the following:

- Every odd line is a sentence
- Every even line is a clue
- The first line starts at index 0, and is considered an even line
- No line can be skipped (barring end of text file!)
- All sentences must have a clue
- All clues must have a sentence
- No sentence is over 51 characters long (including spaces)
- All sentences contain only alphabetical characters or spaces

STEP THREE: IMPORTING FROM EMBEDDED

If you look inside of the Assets > Resources folder you should see a file named "embedded". This will be your first step in importing a file to use in a game. Your goal is to import that embedded file and allow the engine access to the different sentences and clues.

The first thing you will need to do is create a loading script. This will handle loading in your data, and eventually allowing the References script to collect information from it. Inside of this script you will need to include using System.IO. This will allow you access to built-in C# functions for reading files. You will also be utilizing Lists (as opposed to arrays), so go ahead and include a using System.Collections.Generic

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Collections.Generic;

public class LoadScript : MonoBehaviour {

        // Use this for initialization
        void Start () {

        }

        // Update is called once per frame
        void Update () {

        }
}
```

The next step is to begin getting access to your file. Unity comes with a built-in class called a TextAsset. A TextAsset is simply a class that allows you to read **raw** text files.

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Collections.Generic;

public class LoadScript : MonoBehaviour {

    TextAsset textFile;


}
```

You can then load the text file like you do with any other resource that you want to load during runtime: Resources.Load(). Remember, when using Resources.Load() you **must** caste the loaded object to your target class. In this case, you will need to caste it to a text asset.

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Collections.Generic;

public class LoadScript : MonoBehaviour {

    TextAsset textFile;

    void Start()
    {
        textFile = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
    }

}
```

The next thing you need to prepare to do is read the lines of text inside of the file. There are various methods you can use to read a file. The most common methods include StreamReaders and TextReaders. A StreamReader is typically used if you are trying to encode/decode a text file. Since you will not be doing encoding in this lab, you can use a TextReader. A TextReader is simply a class that is capable of reading lines of text inside of an opened file.

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Collections.Generic;

public class LoadScript : MonoBehaviour {

    TextAsset textFile;
    TextReader reader;

    void Start()
    {
        textFile = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
    }

}
```

You can now load the file into the reader (so that it can be read!) by creating a new StringReader and placing the file inside of it. A StringReader is a form of TextReader that can read strings (lines of text in a text file are considered strings).

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Collections.Generic;

public class LoadScript : MonoBehaviour {

    TextAsset textFile;
    TextReader reader;

    void Start()
    {
        textFile = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
        reader = new StringReader(textFile.text);
    }

}
```

Notice that you do not pass the entire file into the string reader. The only thing a StringReader is concerned about is strings. The underlying data in a text file is useless to a StringReader. So instead of passing in the entire file (which includes off the underlying data) you are only going to pass in the text of the file. This is a shortcut that you can use because your file is currently stored as a TextAsset (which is essentially **raw** strings with each part of the file broken up into properties).

Now that you have the file loaded, you can begin to read the lines of the file. You will want to create two new variables: One to store what line number you are on, and one to store the strings that the StringReader will send to you from your file.

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Collections.Generic;

public class LoadScript : MonoBehaviour {

    TextAsset textFile;
    TextReader reader;

    void Start()
    {
        textFile = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
        reader = new StringReader(textFile.text);

        string lineOfText = 0;
        int lineNumber;
    }

}
```

The next command you will need to execute is actually stepping through the file. This is accomplished by sending the command StringReader.ReadLine() to your StringReader. This will cause your reader to read the text from the start, until it finds the escape sequence of an enter key press (\n). You are going to want to have the reader read every single line of your text file, until it finds nothing else on the line (aka it returns null – an empty string).

This is typically accomplished with a using() or with a while() statement. In this lab you will use a while statement. While the reader is returning lines of text, do something.

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Collections.Generic;

public class LoadScript : MonoBehaviour {

    TextAsset textFile;
    TextReader reader;

    void Start()
    {
        textFile = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
        reader = new StringReader(textFile.text);

        string lineOfText;
        int lineNumber = 0;

        //tell the reader to read a line of text, and store that in the lineOfTextVariable
        //continue doing this until there are no lines left
        while ((lineOfText = reader.ReadLine()) != null)
        {

        }
    }

}
```

Inside of the while loop you will have access to the line of text that the StringReader just read. This is where you will implement the heuristic for determining where phrases and clues go. Now that you have access to this information, you will want to create a place to store the phrases and clues that you read from the text file.

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Collections.Generic;

public class LoadScript : MonoBehaviour {

    TextAsset textFile;
    TextReader reader;

    public List<string> textFile = new List<string>();
    public List<string> clues = new List<string>();

    void Start()
    {
        sentences = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
        reader = new StringReader(textFile.text);

        string lineOfText;
        int lineNumber = 0;

        //tell the reader to read a line of text, and store that in the lineOfTextVariable
        //continue doing this until there are no lines left
        while ((lineOfText = reader.ReadLine()) != null)
        {

        }
    }

}
```

You're heuristic is paying attention to odd and even lines, with the 0th line being counted as even. You can easily determine if something is odd by using modulus two. (Remember: modulus will return the remainder of the division).

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Collections.Generic;

public class LoadScript : MonoBehaviour {

    TextAsset textFile;
    TextReader reader;

    public List<string> textFile = new List<string>();
    public List<string> clues = new List<string>();

    void Start()
    {
        sentences = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
        reader = new StringReader(textFile.text);

        string lineOfText;
        int lineNumber = 0;

        //tell the reader to read a line of text, and store that in the lineOfTextVariable
        //continue doing this until there are no lines left
        while ((lineOfText = reader.ReadLine()) != null)
        {
            if(lineNumber%2 == 0)
            {
                //even lines
            }
            else
            {
                //odd lines
            }

            lineNumber++;

        }
    }

}
```

This will also capture the case of the 0th line, since 0/2 = 0.

Inside of these statements you only need to add each sentence and hint to the appropriate list.

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Collections.Generic;

public class LoadScript : MonoBehaviour {

    TextAsset textFile;
    TextReader reader;

    public List<string> textFile = new List<string>();
    public List<string> clues = new List<string>();
```

```
        void Start()
        {
            textFile = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
            reader = new StringReader(sentences.text);

            string lineOfText;
            int lineNumber = 0;

            //tell the reader to read a line of text, and store that in the lineOfTextVariable
            //continue doing this until there are no lines left
            while ((lineOfText = reader.ReadLine()) != null)
            {
                if(lineNumber%2 == 0)
                {
                    sentences.Add(lineOfText);
                }
                else
                {
                    clues.Add(lineOfText);
                }

                lineNumber++;

            }
        }

    }
```

And that is the end of reading the file! You now have access to all of the text information, following a very simpe yet (mostly) effective heuristic for determining where the data is going to be stored.

STEP FOUR: ACCESSING THE DATA

Your next challenge is to access the data that is stored in your LoadScript script so your engine can use it. Since you have a References script, it would make the most sense to send the data over to that script first. This makes sure that all of your references are stored in one unified location.

However, at this point there is going to be a flow problem. If you tell the References script in it's start method to pull the data off of LoadScript, there is no gaurentee that LoadScript has completed the tast of opening and reading the file yet. It could still be in the process of assigning strings to the lists, opening the file, or various other jobs. To fix this problem, you can send a message from the LoadScript script to the References script telling it that the LoadScript script has completed its work, and References can pull information from it.

This is accomplished using a method called SendMessage(string methodName). This method will call the method methodName on **every** component that the calling script is apart of. This means if you place the LoadScript script onto the main camera with the other scripts and then use the SendMessage method, it will attempt to call methodName on every single script on the main camera. Make sure that if you are using SendMessage you are using it wisely and justly!

Since there is only two other scripts on the main camera (References and Engine) it is pretty safe to allow the LoadScript to send a message to them once at the start of the game without incurring any lag or other reoccurring issues. Since we are going to be sending a message that we want References to receive, we need to set up a method in References to receive the message. Since you don't want References to do anything until LoadScript is done doing what it needs to do, you can just change the Start method on References to something like "Gather", and then send the message from LoadScript to call the "Gather" method.

```
        void Gather () {

            //Initialize the arrays to store 52 items, which is the number of letters on the wheel of
            //fortune board
            letters = new Text[52];
            correct = new Image[52];
            unfilled = new Image[52];
```

```
    using UnityEngine;
    using System.Collections;
    using System.IO;
    using System.Collections.Generic;

    public class LoadScript : MonoBehaviour {

        TextAsset textFile;
        TextReader reader;

        public List<string> textFile = new List<string>();
        public List<string> clues = new List<string>();

        void Start()
        {
            textFile = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
            reader = new StringReader(textFile.text);

            string lineOfText;
            int lineNumber = 0;

            //tell the reader to read a line of text, and store that in the lineOfTextVariable
            //continue doing this until there are no lines left
            while ((lineOfText = reader.ReadLine()) != null)
            {
                if(lineNumber%2 == 0)
                {
                    sentences.Add(lineOfText);
                }
                else
                {
                    clues.Add(lineOfText);
                }

                lineNumber++;

            }

            SendMessage("Gather");
        }

    }
```

While you are at it, the SendMessage will only send messages to all components on the same game object that the script calling it is on. In this case, SendMessage will only send messages to the same game object that LoadScript is on. To make sure that no matter what, there will be a script to receive the message (and more importantly, the script that we want to receive the message) we can use the [RequireComponent] attribute. What this attribute will do is for any game object you add the LoadScript scrip to, it will check to see if the required component is on the object as well. If it is not, it will automatically add the script to the object.

```
    using UnityEngine;
    using System.Collections;
    using System.IO;
    using System.Collections.Generic;

    [RequireComponent(typeof(References))]
    public class LoadScript : MonoBehaviour {

        TextAsset textFile;
        TextReader reader;

        public List<string> sentences = new List<string>();
        public List<string> clues = new List<string>();

        void Start()
        {
            textFile = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
            reader = new StringReader(textFile.text);

            string lineOfText;
            int lineNumber = 0;

            //tell the reader to read a line of text, and store that in the lineOfTextVariable
            //continue doing this until there are no lines left
            while ((lineOfText = reader.ReadLine()) != null)
            {
                if(lineNumber%2 == 0)
                {
                    sentences.Add(lineOfText);
                }
                else
                {
                    clues.Add(lineOfText);
                }

                lineNumber++;
            }

            SendMessage("Gather");
        }

    }
```

CHECKPOINT!

1. Place the LoadScript script onto the Main Camera
2. Press Play on the unity engine
3. Check the variables on the LoadScript script on the Main Camera.
4. You should now have an array full of approx. 13 strings for both the sentences and clues list.

STEP FIVE: UTILIZING THE DATA

Now that you have the data, all that is left is to pull the data into the appropriate script and start using it!

There are a few places we need the data. First, in the References script, and second in the engine. You should already be familiar with pulling values off of other scripts at this point, so below is the code snippets used to accomplish the goal.

References.cs

```
    [HideInInspector]
    public Button[] buttons;
    [HideInInspector]
    public List<string> sentences = new List<string>();
    [HideInInspector]
    public List<string> clues = new List<string>();

    void Gather () {
        sentences = GetComponent<LoadScript>().sentences;
        clues = GetComponent<LoadScript>().clues;
```

Engine.cs

```
    /// <summary>
    /// Sets all player data to default values
    /// </summary>
    public void StartGame()
    {
        //Sets the goal, and how many letters the player has guessed to zero
        charGoal = 0;
        charGot = 0;

        //ADD
        int numSent = refs.sentences.Count;
        int randomSent = Random.Range(0, numSent);

        //ADD
        word = refs.sentences[randomSent];
        hint = refs.clues[randomSent];
```

CHECKPOINT!

1. Play the game, and verify that the new phrases and clues are appearing.

STEP SIX: INCORPORATING EXTERNAL FILES

Now that you have an embedded file working, you want to allow the user to add new phrases and clues after the game has been built. Luckily, this only required small modifications. You will keep the LoadScript you currently have, and use it as a fallback incase there is no user-file in the directory to pull information from.

In this step, you are going to want to read an external text file. In order to read external files, there are a few things that you need. The first is something called a FileInfo. A FileInfo is a class that gives you a lot of access to reading, writing, deleting, moving, and opening files. There are other, more complicated, ways to open a file but using a FileInfo is a simple and effective method for basic file access.

```
[RequireComponent(typeof(References))]
public class LoadScript : MonoBehaviour {

    FileInfo originalFile;
    TextAsset textFile;
    TextReader reader;
```

Your next step is load the file into your FileInfo. To do this, you will create a new FileInfo and point it to the location of the text document. You should be familiar with getting the path to the assets folder at this point, and all you need to do after that is tack on the file name and extension of the file.

```
    public List<string> sentences = new List<string>();
    public List<string> clues = new List<string>();

    void Start()
    {

        originalFile = new FileInfo(Application.dataPath + "/sentences.txt");

        textFile = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
        reader = new StringReader(textFile.text);
```

Before going any further in your code, you are going to want to verify that something was actually loaded. Creating a new FileInfo will only search for existing files at that location, and load it in. It will not create a new file if one does not exist. To determine if the file was property loaded, you can check to see if your FileInfo is null. You can also use a property inside of the FileInfo class called Exists. This will simply return if the file does or does not exist.

```
    public List<string> sentences = new List<string>();
    public List<string> clues = new List<string>();

    void Start()
    {

        originalFile = new FileInfo(Application.dataPath + "/sentences.txt");

        if(originalFile != null && originalFile.Exists)
        {

        }
```

If the file does exist, then you will want to load the file into the TextReader that you were working with earlier. The method to open the file from FileInfo is not very difficult, you call the OpenText method on the FileInfo class.

```
    void Start()
    {

        originalFile = new FileInfo(Application.dataPath + "/sentences.txt");

        if(originalFile != null && originalFile.Exists)
        {
            reader = originalFile.OpenText();
        }
```

Now you can say if the file does not exist, default to the embedded document:

```
    void Start()
    {

        originalFile = new FileInfo(Application.dataPath + "/sentences.txt");

        if(originalFile != null && originalFile.Exists)
        {
            reader = originalFile.OpenText();
        }
        else
        {
            textFile = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
            reader = new StringReader(textFile.text);
        }

        string lineOfText;
        int lineNumber = 0;
```

CHECKPOINT!

1. Run the game from the unity editor. It should not appear to function any differently than before.
2. Build the game.
3. Go to the data folder that was created with the game build.
4. Add a text file named “sentences” to the folder.
    a. The same folder that contains the Resources, Mono, and Managed files!
5. Add a few different lines of text, making sure to follow the file heuristic outlined at the beginning of the lab
6. Save the text file
7. Run the **built** game (not the version in the editor!)
8. You should see your newly added text

WRAP UP

You now have the tools to read, import, and utilize basic files from inside and outside of a game engine.

COMPLETE CODE

LoadScript.cs

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Collections.Generic;

[RequireComponent(typeof(References))]
public class LoadScript : MonoBehaviour {

    FileInfo originalFile;
    TextAsset textFile;
    TextReader reader;

    public List<string> sentences = new List<string>();
    public List<string> clues = new List<string>();

    void Start()
    {

        originalFile = new FileInfo(Application.dataPath + "/sentences.txt");

        if (originalFile != null && originalFile.Exists)
        {
            reader = originalFile.OpenText();
        }
        else
```

```
            {
                textFile = (TextAsset)Resources.Load("embedded", typeof(TextAsset));
                reader = new StringReader(textFile.text);
            }

            string lineOfText;
            int lineNumber = 0;

            //tell the reader to read a line of text, and store that in the lineOfTextVariable
            //continue doing this until there are no lines left
            while ((lineOfText = reader.ReadLine()) != null)
            {
                if (lineNumber % 2 == 0)
                {
                    sentences.Add(lineOfText);
                }
                else
                {
                    clues.Add(lineOfText);
                }

                lineNumber++;

            }

            SendMessage("Gather");
        }

    }
```

Engine.cs Edits

```
      /// <summary>
        /// Sets all player data to default values
        /// </summary>
        public void StartGame()
        {
            //Sets the goal, and how many letters the player has guessed to zero
            charGoal = 0;
            charGot = 0;

            //ADD
            int numSent = refs.sentences.Count;
            int randomSent = Random.Range(0, numSent);

            //ADD
            word = refs.sentences[randomSent];
            hint = refs.clues[randomSent];

            //If there are too many letters in the phrase, skip it
            //Probably some more robust error checking and fail-safe should happen here
            if (word.Length > 51)
            {
                Debug.Log("PHRASE TO LONG!");
                return;
            }
```

References.cs EDITS

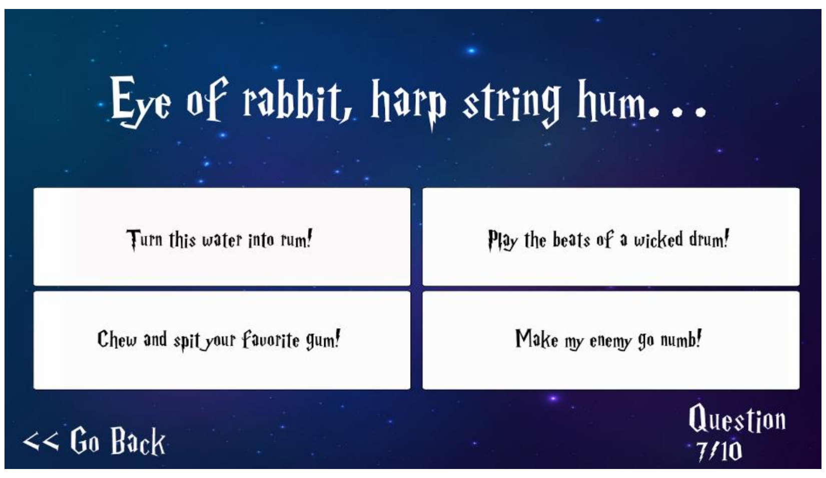
```
        [HideInInspector]
        public Button[] buttons;
        //ADDED
        [HideInInspector]
        //ADDED
        public List<string> sentences = new List<string>();
        //ADDED
        [HideInInspector]
        //ADDED
        public List<string> clues = new List<string>();

        //Altered
        void Gather () {
            //ADDED
            sentences = GetComponent<LoadScript>().sentences;
            //ADDED
            clues = GetComponent<LoadScript>().clues;

            //Initialize the arrays to store 52 items, which is the number of letters on the wheel of
            //fortune board
            letters = new Text[52];
            correct = new Image[52];
            unfilled = new Image[52];
```

THINK ABOUT IT!

1. Why are lists used instead of arrays in the LoadScript?
2. What other method(s) could you use to load the data from an external file?
3. What is the purpose of having two loading systems in your load script file?
4. Will FileInfo ever be null? Why/Why not?
5. How could you allow the player to tell the engine where their file is that contains their custom sentences/phrases?

ON YOUR OWN!

Create a simple trivia game. The trivia type can be of your choice. The UI for your trivia should be extremely simple, and can look something like this:


Eye of rabbit, harp string hum...
Turn this water into rum!
Play the beats of a wicked drum!
Chew and spit your favorite gum!
Make my enemy go numb!
<< Go Back
Question 7/10

It needs to include: A location for the question to appear, at least 2 options (up to 4) for possible answers, a display that shows how many questions the user has answered. There does not need to be a limit (like 7/10 questions) it can be an endless trivia game! You do not have to include the option for the user to go back to previous questions. There does need to be a display to show how many answers the user has gotten right, and how many the user has gotten wrong.

Allow your user to add their own questions and answers. Make sure they can notate which answer is correct! If the user does not supply questions and answers, your game should fall back onto an **embedded** file that is read with the same heuristic as the users file. Include an explanatory text document that informs the user of the heuristic used for your file.