SECTION TWO: CREATING PREFABS

A common task for a programmer in Unity is to create various prefabs. When multiple objects have been added, prefabs must be created for every new object. This can be a time consuming, cumbersome task. With many prefabs to create, there is a high possibility of mistake. To solve this problem, you will create a utility to generate a prefab from selected materials and meshes.

To use this utility, the user will:

1. Select the game objects they wish to turn into prefabs
2. Use the menu option “Create Prefab”
3. Move the newly created prefabs to the appropriate locations

PSEUDOCODE

```
New menu item Project Tools > Make Prefab
Get user selections
Get Object name
Get Object folder location
If prefab already exists in the main assets folder
    Ask the user if they want to overwrite the prefab
Else
    Create the new prefab

Create New Prefab()
    Create Empty Prefab
    Place selected object into prefab
    Refresh database

    Destroy selected object
    Replace object with prefab
```

STEP ONE: SETUP

Follow the steps outlined previously to create the Editor folder (if it does not already exist in your project). This is where you will place the script to create the prefab. Create a new script in this folder as well, following appropriate naming conventions. Make sure to include the using UnityEditor statement!

STEP TWO: CREATING THE MENU ITEM

Create a menu item that will go under Project Tools > Create Prefab.

This menu item should call a function named CreatePrefab().

STEP THREE: WORKING WITH SELECTIONS

The next thing you need to do it get the game objects the user has selected, and store some information about the selection. To start, you want to remember what the user has selected, and then store the objects name and location.

You will use an array to hold the user's selections. In order to get the users selections you can use a special function: Selection.gameObjects.

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class MakePrefabs : MonoBehaviour {

    [MenuItem("Project Tools/Create Prefab")]
    public static void CreatePrefab()
    {
        GameObject[] selectedObjects = Selection.gameObjects;
    }
}
```

STEP FOUR: COLLECTING DETAILS

Using a foreach statement, you can pull the name of each object in the array using Object.name. You will need to manually search the main assets folder for the prefab. First, construct a string with the path.

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class MakePrefabs : MonoBehaviour {

    [MenuItem("Project Tools/Create Prefab")]
    public static void CreatePrefab()
    {
        GameObject[] selectedObjects = Selection.gameObjects;
        foreach(GameObject go in selectedObjects)
        {
            string name = go.name;
            string assetPath = "Assets/" + name + ".prefab";

            Debug.Log ("Name: " + go.name + " Path: " + assetPath);
        }
    }
}
```

By using “Assets/” + name + “.prefab” we get a path similar to Assets/object.prefab. We can use this path to determine if the prefab already exists in this location. This is mainly a safeguard to prevent overwriting a pre-existing prefab. Remember that once the prefabs are created, the user will have to move them into the appropriate folder.

CHECKPOINT!

To test the functionality of this new menu item:

1. Create a 3D cube object in the scene.
   a. Make sure the new object is selected in the hierarchy
2. Use the menu command “Create Prefab”
3. If you check your console, you should have the follow debug statements:

```
Name: Cube Path: Assets/Cube.prefab
```

STEP FIVE: CHECK IF THE PREFAB EXISTS

Once you have the location of the object, you need to determine if the prefab already exists. You can do this by utilizing the AssetDatabase.LoadAssetAtPath(string path, Type type) function. You can use this function by checking if an asset was actually loaded when the function is used.

```
using UnityEngine;
using System.Collections;
using UnityEditor;

```

```
public class MakePrefabs : MonoBehaviour {

   [MenuItem("Project Tools/Create Prefab")]
   public static void CreatePrefab()
   {
      GameObject[] selectedObjects = Selection.gameObjects;

      foreach(GameObject go in selectedObjects)
      {
         string name = go.name;
         string assetPath = "Assets/" + name + ".prefab";

         if(AssetDatabase.LoadAssetAtPath(assetPath, typeof(GameObject)))
         {
            Debug.Log ("Asset exists!");
         }
         else
         {
            Debug.Log ("Asset does not exist!");
         }
         Debug.Log ("Name: " + go.name + " Path: " + assetPath);
      }
   }
}
```

CHECKPOINT!

To test the functionality of this new menu item:

4. Create a 3D cube object in the scene.
    a. Make sure the new object is selected in the hierarchy
5. Use the menu command "Create Prefab"
6. If you check your console, you should have the follow debug statements:

```
Asset does not exist!
Name: Cube Path: Assets/Cube.prefab
```

7. Create a prefab of the object you created in step 1.
    a. Make sure the prefab is in the base assets folder.

8. Select the object in the hierarchy
9. Use the menu command "Create Prefab"
10. If you check your console, you should now have the following debug statements:

```
Asset exists!
Name: Cube Path: Assets/Cube.prefab
```

STEP SIX: VERIFY

If the asset does already exist, you need to ask the user if they are sure they want to overwrite the prefab. In order to do this, you can use a special editor utility. The [EditorUtility.DisplayDialog(string title, string message, string ok, string cancel ="")]() will cause a dialog box to pop up and ask the user a yes/no question.

```csharp
using UnityEngine;
using System.Collections;
using UnityEditor;

public class MakePrefabs : MonoBehaviour {

   [MenuItem("Project Tools/Create Prefab")]
   public static void CreatePrefab()
   {
      GameObject[] selectedObjects = Selection.gameObjects;

      foreach(GameObject go in selectedObjects)
      {
         string name = go.name;
         string assetPath = "Assets/" + name + ".prefab";

         if(AssetDatabase.LoadAssetAtPath(assetPath, typeof(GameObject)))
         {
            Debug.Log ("Asset exists!");
            if(EditorUtility.DisplayDialog("Caution", "Prefab already exists. " +
                                  "Do you want to overwrite?","Yes","No"))
             {
               Debug.Log ("overwriting!");
             }
```

```
        }
        else
        {
            Debug.Log ("Asset does not exist!");
        }
        Debug.Log ("Name: " + go.name + " Path: " + assetPath);
    }
  }
}
```

CHECKPOINT!

To verify this is working properly, use the Cube created at the previous checkpoint.

1. Verify that you have a prefab of the cube in the assets folder.
2. Verify that you have a copy of the prefab in the hierarchy.
3. Click on the Cube in the hierarchy.
4. Run the “Create Prefab” menu option.
5. When the dialog window pops up, click “Yes”.
6. You should have the following message in your console:

```
Asset exists!
overwriting!
Name: Cube Path: Assets/Cube.prefab
```

7. Clear the Console.
8. Run the “Create Prefab” menu option again.
9. When the dialog window pops up, click “No”.
10. You should have the following message in your console:

```
Asset exists!
Name: Cube Path: Assets/Cube.prefab
```

STEP SEVEN: CREATENEW() FUNCTION

The infrastructure is now in place to create the prefab. Make a new static function named “createNew”. It should take a GameObject (the selected object) and a string (the path to the object).

CREATING A NEW (EMPTY) PREFAB

Utilizing the PrefabUtility this time, use PrefabUtility.CreateEmptyPrefab(string path) to create an empty prefab at the location passed into the function.

```
public static void CreateNew(GameObject obj, string location)
{
    Object prefab = PrefabUtility.CreateEmptyPrefab (location);
}
```

PLACE OBJECT INTO PREFAB

The next step is to actually place the selected object into the prefab, effectively making a shallow "copy" of the object as a prefab. To accomplish this you can use PrefabUtility.ReplacePrefab(GameObject go, Object targetPrefab).

```
public static void CreateNew(GameObject obj, string location)
{
    Object prefab = PrefabUtility.CreateEmptyPrefab (location);
    PrefabUtility.ReplacePrefab (obj, prefab);

}
```

REFRESH AND DESTROY

Since you are finished making changes to the project files, you need to refresh the asset database to sync the changes in the editor.

After refreshing the asset database, you need to tie up loose ends. Currently, there is a prefab with a copy of the game object in the scene. There is also the game object in the scene, which is separate from the prefab. You need to destroy the object in the scene, and replace it with the prefab.

Destroying a game object can be done using DestroyImmediate. This function is similar to destroy, however it will not run any deconstruction code. Destroy will never be run in edit mode, which is why you use DestroyImmediate instead.

Placing the prefab object can be done using PrefabUtility.InstantiatePrefab.

```
public static void CreateNew(GameObject obj, string location)
{
    Object prefab = PrefabUtility.CreateEmptyPrefab (location);
    PrefabUtility.ReplacePrefab (obj, prefab);
    AssetDatabase.Refresh ();

    DestroyImmediate (obj);
    GameObject clone = PrefabUtility.InstantiatePrefab (prefab) as GameObject;
}
```

CHECKPOINT!

To verify this is working:

1. Replace the two debug statements in the CreatePrefab() function with the function CreateNew
   a. Pass it the current game object in the loop, as well as the asset path we created.
2. Comment out the debug statement that is printing the name and asset path of the object
3. Delete the prefab we created in the last checkpoint
   a. Make sure to keep the cube in the scene
4. Click the cube in the scene
5. Run the menu item "Create Prefab"
6. You should have a new prefab in the base assets folder named "Cube".
7. The cube in the scene should be linked to this prefab.
8. Reselect the Cube in the scene.
9. Run the menu item "Create Prefab"
10. When the dialogue button pops up, click Yes
11. You should have a new prefab, with the same name, in your assets folder.
12. Run the menu item "Create Prefab"

SECTION THREE: 2D SPRITE AUTOMATION

In this section you will create a 2D sprite creation tool. Working on 2D games requires diligence in sprite sheet management. With this tool, you can quickly create animations from sprite sheets without having to spend large amounts of time breaking up individual animations. The tool is created as a one-shot tool. This means that the user cannot go back in and edit information already put in. If the user closes the tool, they must start over. This is to reinforce that the tool is for creation, not management.

To use this tool, the user will:

1. Select the sprite sheet in the assets folder.
   a. The sprite sheet must already be changed from Texture to sprite, with a sprite mode of multiple and then sliced.

2. Use the menu item Project Tools > 2D Animations
3. Input the name for the controller that will be created.
4. Input how many animations the sprite sheet has.
5. Input a name for each animation.
6. Specify the start frame, end frame, framerate, and spacing of the frames.
7. Specify if the animation will loop or Ping-Pong.
8. Click the create button.

PSEUDOCODE

```
Setup to store the following information:
    selected object
    name of the controller
    names for all animations
    frame rates for all animations
    time between frames for all animations
    start frame for all animations
    end frame for all animations
    if each animation should pingpong
    if each animation should loop

New menu item Project Tools > 2D Animations
    Capture and store selected object
    If the user has no object selected
        do nothing

    Create a new pop-up window to collect information
    show the window

While the window is being shown
    Create a label with the name of the object that the sprites will be pulled from

    Get the controller name and number of animations that will be created
    For every animation that will be created
        get its name
        get the start and end frame
        get the frame rate
        get the time between the frames
        ask if it should loop
        ask if it should pingpong

    If the user presses the create button
        create an empty animation controller with the appropraite name
        for every animation that will be created
          Create the animation clip
          if the animation clip should loop
            set the clip to loop
          add the clip to the controller

Creating an animation clip
    get the path to the sprite sheet
    extract all sliced sprites from the sprite sheet
    determine how many frames the animation will be
    determine the time between each frame
    create an empty animation clip
    set the frame rate for the clip

    prepare to create a curve on the animation
    set the curve to be for the spriterenderer
    give the curve a property type of sprite

    set up for the keyframes to be stored and recorded
    if there is no pingpong
        initialize the number of keyframes to be calcuated amount
    otherwise
        initialize the number of keyframes to be twice the calcuated amount

    keep track of how many frames have been added to the animation (starting at 0)
    for every frame between the specified start and end frame
        create a new keyframes
        place it at the appropriate location
```

```
        set its value to the appropriate sprite
        store the keyframe inside of the collection of keyframes created earlier
        increase the number of frames that have been added

    if the animation should pingpong
        for every frame between the specified end and start frame
            create a new keyframes
            place it at the appropriate location
            set its value to the appropriate sprite
            store the keyframe inside of the collection of keyframes created earlier
            increase the number of frames that have been added

    Create a new keyframe
    place it at the appropriate location
    set its value to the start value
    store the keyframe inside of the collection of keyframes created earlier

    assign the name of the clip
    bind the curve and keyframes to the animation
    create the animation clip
    return the clip

When the window gains focus
    If the user changes playmode (from paused/stopped to play or vise-versa)
        close the window
```

STEP ONE: SETUP

Follow the steps outlined previously to create the Editor folder (if it does not already exist in your project). This is where you will place the script to create the animations. Create a new script in this folder as well, following appropriate naming conventions. Make sure to include the using UnityEditor statement!

Also create the following variables:

```
public static Object selectedObject;
```

```
int numAnimations;
string controllerName;
string[] animationNames = new string[100];
float[] clipFrameRate = new float[100];
float[] clipTimeBetween = new float[100];
int[] startFrames = new int[100];
int[] endFrames = new int[100];
bool[] pingPong = new bool[100];
bool[] loop = new bool[100];
```

The selected object is static so it can be accessed inside of the menu function (which is also static).

The array variables are initialized to a size at the start of the script because they will be accessed inside of a function similar to the update function (meaning we don’t want to create new arrays every time we enter the function).

STEP TWO: CREATING THE MENU ITEM

Create a menu item that will go under Project Tools > 2D Animations.

This menu item should call a function named Init().

STEP THREE: POPUP WINDOWS

The Init function is very simple in this menu item. It has two goals: store the selected object, and open a popup window.

Capturing a single selected object (as opposed to multiple selected objects like you did in the previous section) can be done with Selection.activeObject.

Next, you want to make sure that the selected object actually exists (is not null). If it doesn’t exist, this function should do nothing (return).

Lastly, opening a popup window. There are many different types of preset popup windows that you can use (such as the display dialogue used in the previous section). However you are going to want full control over this popup windows so you can display whatever you need. To do this, you need to create a blank popup window that is customizable. This can only be done in the unity editor!

In order to do this, you need a class that is an EditorWindow. You can either create a new script to be your EditorWindow, or you can use the menu script you are already working with. Since you are going to be passing the selected object around, then using the existing menu script will be easiest. This means that the menu script needs to be an EditorWindow.

Popup windows are specific to the script they are created in. You can only have one type of custom popup window per script (or class). A popup window has a type that is identical to the class name it is contained in. This means that if you have a class (or script) named Testing, and you want that script to use a popup window, the popup window will also be of type Testing.

Creating a popup window is done with EditorWindow.GetWindow(type t). Again, the type will be identical to the script (or class) that the window is created in! Lastly, to show the window that was created you use EditorWindow.Show().

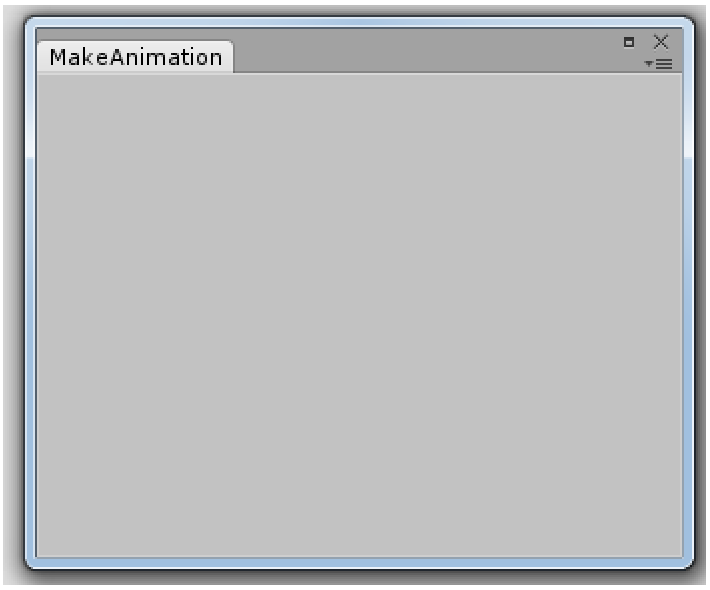
```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class MakeAnimations : EditorWindow {

   public static Object selectedObject;

   int numAnimations;
   string controllerName;
   string[] animationNames = new string[100];

   float[] clipFrameRate = new float[100];
   float[] clipTimeBetween = new float[100];
   int[] startFrames = new int[100];
   int[] endFrames = new int[100];
   bool[] pingPong = new bool[100];
   bool[] loop = new bool[100];


   [MenuItem("Project Tools/2D Animations")]
   static void Init() {

      selectedObject = Selection.activeObject;

      if (selectedObject == null)
         return;

      MakeAnimations window = (MakeAnimations)EditorWindow.GetWindow (typeof (MakeAnimations));
      window.Show();
```

```
    }
}
```

Note that if you were to create a second script to be your popup window in the future, you would just replace MakeAnimations with the name of the script that you created the popup window in.

CHECKPOINT!

To test that this is working:

1. Download the sprite sheet: download here
    a. Right click the image and save it to your computer.
2. Import the image into Unity
3. Click the image in your assets folder.
4. Change the Texture Type in the inspector to Sprite (2D or UI).
5. Change the Sprite Mode to Multiple.
6. Enter the sprite editor.
7. Slice the sprite to a 60 by 60 grid.
8. Apply the changes.
9. Make sure the sprite is selected in the Assets folder.
10. Click Project Tools > 2D Animations.
11. You should get a popup window that looks like the following:
    a. 

STEP FOUR: POPULATING THE POPUP

Armed with an empty popup window, you can now begin to display custom information to the user. The appearance and interactions with the popup window are controlled in a function called OnGUI(). This is a built-in unity function that controls what a window (or GUI) looks like when it is displayed. Since we are writing the window as a MakeAnimation window, the OnGUI will control what the MakeAnimation window will look like.

The first thing that you want to show in the window is the name of the object the animations will be created from. This way the user is aware of what they are working with. However, you only want to show the name of the object you are working with if the object actually exists. By making a check for the object existing early, you can avoid odd behavior such as trying to edit a deleted object.

There are two main ways of displaying GUI items. The first is with a class called EditorGUI, and the second with a class called EditorGUILayout. Each class contains almost identical functions. The main difference is that with EditorGUI you must specify everything about the display. This includes how large each individual display is, exactly where the text will fall, and other similar items. Using EditorGUILayout will automate that, displaying each item from top to bottom.

For this section you will be using EditorGUILayout.

Within the GUI class, there are functions that exist for different types of items. You will use the following for this section:

- void EditorGUILayout.LabelField(string label)
    - Creates an item to display text
- string EditorGUILayout.TextField(string label, string text)
    - Creates an item for text input
- int EditorGUILayout.IntField(string label, int value)
    - Creates an item for integer input
- float EditorGUILayout.FloatField(string label, float value)
    - Creates an item for float input
- bool EditorGUILayout.Toggle(string label, bool value)
    - Creates an item for Boolean input
- void EditorGUILayout.Separator()
    - Creates a space between the previous item and the next item
- void EditorGUILayout.BeginHorizontal()
    - Starts a section where the following items are displayed horizontally instead of vertically
- void EditorGUILayout.EndHorizontal()
    - Ends a section where the previous items are displayed horizontally instead of vertically

For functions that are capturing input, the value is what to display in the field. You still have to assign the value to a variable. For example, to capture the text input and store it in a string variable named “sampleText”, you would do the following:

```
void OnGUI()
{
   sampleText = EditorGUILayout.TextField("Sample text: ", sampleText);
}
```

This would display a text input item, with the value of the sampleText variable shown by default. As the user edits the item, it will save to the sampleText variable. Variables must be initialized before they can be used for GUI items.

In order to display the name of the object that the user is editing, you would use a Label Field:

```
void OnGUI()
{
   EditorGUILayout.LabelField("Animations for " + selectedObject.name);
}
```

After that, you will want a space. A space can be created using Separator:

```
void OnGUI()
{
   EditorGUILayout.LabelField("Animations for " + selectedObject.name);
   EditorGUILayout.Separator();
}
```

Next, you need to collect the name for the animation controller that will be created. This can be done with a Text Field.

```
void OnGUI()
{
   EditorGUILayout.LabelField("Animations for " + selectedObject.name);
   EditorGUILayout.Separator();
```

```
    controllerName = EditorGUILayout.TextField("Controller Name", controllerName);
}
```

Finally, you need to determine how many animations this program will be creating. This can be done with an Int Field.

```
void OnGUI()
{
    EditorGUILayout.LabelField("Animations for " + selectedObject.name);
    EditorGUILayout.Separator();

    controllerName = EditorGUILayout.TextField("Controller Name", controllerName);
    numAnimations = EditorGUILayout.IntField("How many animations?", numAnimations);
}
```

Next, you need to begin displaying options for each animation. Each animation should have a set number of options:

1. The name of the animation
2. The start frame of the animation
3. The end frame of the animation
4. The frame rate of the animation
5. The spacing of the animation
6. If the animation should loop
7. If the animation should pingpong

Setting up the GUI for the animations entails looping through each theoretical animation and placing the appropriate fields.

```
void OnGUI()
{
    //Display the objects name that the animations will be created from
    EditorGUILayout.LabelField("Animations for " + selectedObject.name);
    //Create a space
```

```
EditorGUILayout.Separator();
//Get the name for the animation controller
controllerName = EditorGUILayout.TextField("Controller Name", controllerName);
//Determine how many animations there will be
numAnimations = EditorGUILayout.IntField("How many animations?", numAnimations);
//Loop through each theoretical animation
for (int i = 0; i < numAnimations; i++)
{
   //Determine a name for the animation
   animationNames[i] = EditorGUILayout.TextField("Animation Name", animationNames[i]);

   //Start a section where the following items will be displayed horizontally instead of vertically
   EditorGUILayout.BeginHorizontal();
   //Determine the start frame for the animation
   startFrames[i] = EditorGUILayout.IntField("Start Frame", startFrames[i]);
   //Determine the end frame for the animation
   endFrames[i] = EditorGUILayout.IntField("End Frame", endFrames[i]);
   //End the section where the previous items are displayed horitontally instead of vertically
   EditorGUILayout.EndHorizontal();

   //Start a section where the following items will be displayed horizontally instead of vertically
   EditorGUILayout.BeginHorizontal();
   //Determine the frame rate for the animation
   clipFrameRate[i] = EditorGUILayout.FloatField("Frame Rate", clipFrameRate[i]);
   //Determine the space between each keyframe
   clipTimeBetween[i] = EditorGUILayout.FloatField("Frame Spacing", clipTimeBetween[i]);
   //End the section where the previous items are displayed horitontally instead of vertically
   EditorGUILayout.EndHorizontal();


   //Start a section where the following items will be displayed horizontally instead of vertically
   EditorGUILayout.BeginHorizontal();
   //Create a checkbox to determine if this animation should loop
   loop[i] = EditorGUILayout.Toggle("Loop", loop[i]);
   //Create a checkbox to determine if this animation should pingpong
   pingPong[i] = EditorGUILayout.Toggle("Ping Pong", pingPong[i]);
   //End the section where the previous items are displayed horitontally instead of vertically
   EditorGUILayout.EndHorizontal();
```

```
            //Create a space
            EditorGUILayout.Separator();
        }
    }
```

Next, the window needs a button. A special GUILayout button can be used. To use a GUILayout button, you enclose it in an if statement. The if statement returns true if the button has been pressed.

```
void OnGUI()
{
    <code omitted>

    //Loop through each theoretical animation
    for (int i = 0; i < numAnimations; i++)
    {
        <code omitted>
    }

    //Create a button with the label "Create"
    if (GUILayout.Button("Create"))
    {
        //If the button has been pressed...

    }
}
```

STEP FIVE: CONTROLLERS AND LOOPING

After the “Create” button has been pressed, you can create the animation controller that will be used to store the animations.

```
    //Create a button with the label "Create"
    if (GUILayout.Button("Create"))
    {
      //If the button has been pressed...
      UnityEditor.Animations.AnimatorController controller =
                        UnityEditor.Animations.AnimatorController.CreateAnimatorControllerAtPath(("Assets/" +
                        controllerName + ".controller"));

    }
```

UnityEditor.Animations.AnimatorController is a well-hidden class for manipulating animator controllers via script. This particular function creates an animator controller as the specified path, including the name and extension of the controller. With the controller created, you can begin to create each animation. You will create a function later for the actual animation creation, so for now you can use a dummy animation. After the animation is created, you will want to determine if you should set the looping on the animation to true. Lastly, you want to add the animation to the animator controller.

```
//Create a button with the label "Create"
    if (GUILayout.Button("Create"))
    {
      UnityEditor.Animations.AnimatorController controller =
                        UnityEditor.Animations.AnimatorController.CreateAnimatorControllerAtPath(("Assets/" +
                        controllerName + ".controller"));

      for (int i = 0; i < numAnimations; i++)
      {
        //Create animation clip
        AnimationClip tempClip = new AnimationClip();

        if (loop[i])
        {
          //set the loop on the clip to true
        }

        controller.AddMotion(tempClip);
        }
    }
```

AnimatorController.AddMotion(AnimationClip clip) will add an animation clip to the controller, creating a new state in the process. The first clip added to the controller will be the default clip. All of the others will be treated as clips with no transitions.

To set the loop on an animation clip to true, you first have to extract the animation clip's settings. Once you have them, you can edit them. After editing them, you can reassign the settings back onto the clip.

```
            AnimationClip tempClip = new AnimationClip();
            if (loop[i])
            {
                AnimationClipSettings settings = AnimationUtility.GetAnimationClipSettings(tempClip);
                settings.loopTime = true;
                settings.loopBlend = true;
                AnimationUtility.SetAnimationClipSettings(tempClip, settings);
            }
```

AnimationUtility is a utility class that has a variety of functions for editing animations and animation clips.

STEP SIX: CREATING THE ANIMATIONS

With a majority of the framework setup, you can now create the actual animation. In order to create the animation, you should create a new function. The function should return an AnimationClip. The parameters for the function should include:

- The object that contains the sprites
- The start/end frame of the animation
- The framerate of the animation
- The spacing of the animation
- The name of the animation
- If the animation should pingpong

```
    public AnimationClip CreateClip(Object obj, string clipName, int startFrame, int endFrame, float frameRate, float
timeBetween, bool pingPong)
    {
}
```

Once inside, you need to:

- Get the path to the object that contains all the sprites
- Load the sprites from the object
- Determine how many frames there are going to be, including one extra frame
- Determine how long the animation is going to be
- Create a new animation
- Set up the frame rate
- Bind the clip as a Sprite Renderer animation
- Set up the key frames, accounting for pingPong
- Apply the key frames to the clip
- Assign the clips name
- Create the clip
- Return the clip

LOADING THE SPRITES

You should be familiar with getting the path to an object using AssetDatabase.GetAssetPath. With the asset path, you can load in all of the sprites. To load all of the assets at a specific location (such as Assets/SpriteSheet.png) you use a special function: AssetDatabase.LoadAllAssetsAtPath(string path). This works because once you use the sprite manager in the inspector to split up a spritesheet, it creates all of the individual sprites. They are just contained within the original object. So when you tell unity to load all of the assets at the original objects location it will load the original object plus all of the individual sprites that were created.

```
//Get the path to the object
string path = AssetDatabase.GetAssetPath(obj);

//Extract the sprites
Object[] sprites = AssetDatabase.LoadAllAssetsAtPath(path);
```

NUMBER OF FRAMES

Once you load all of the sprites, finding out how many frames there are is a matter of subtraction. Take the specified end frame, subtract that from the start frame. However, you need to add an extra frame in. At the end of the animation (when the last sprite is shown) you will need to add in the first sprite again (for the looping). Otherwise, it will quickly flicker from the last sprite to the first one.

For example, if we had a sprite with four frames (represented by f1, f2, f3, f4), with three seconds between each frame (each second being represented by a -) the animation would look like this;

f1 - - - f2 - - - f3 - - - f4

Now, when the animation got to the fourth frame, it will go back to the first frame for the loop – immediately, since there is no time after it. Therefore, you want it to be set up like this:

f1 - - - f2 - - - f3 - - - f4 - - - f1

This will allow the fourth frame to be displayed for three seconds before it goes back to the first frame.

```
    //Get the path to the object
string path = AssetDatabase.GetAssetPath(obj);

    //Extract the sprites
Object[] sprites = AssetDatabase.LoadAllAssetsAtPath(path);

    //Determine how many frames, and the length of each frame
int frameCount = endFrame - startFrame + 1;
float frameLength = 1f / timeBetween;

    //Create a new (empty) animation clip
AnimationClip clip = new AnimationClip();

    //Set the framerate for the clip
clip.frameRate = frameRate;
```

CURVE BINDING

The next part includes the curve binding. When you manually create a new animation, you generally add a curve and then drag in keyframes. You still have to replicate this via code when creating sprite animations. The goal when creating the curve via code is to link the curve to the sprite renderer, then assign it to a sprite animation. This tells unity that you will be animating the sprite renderer by changing the sprites of the object. After the curve is created, you can start to generate the keyframes where the sprites will be changed.

```
    //Set the framerate for the clip
clip.frameRate = frameRate;

    //Create the new (empty) curve binding
EditorCurveBinding curveBinding = new EditorCurveBinding();
    //Assign it to change the sprite renderer
curveBinding.type = typeof(SpriteRenderer);
```

```
        //Assign it to alter the sprite of the sprite renderer
    curveBinding.propertyName = "m_Sprite";
```

KEY FRAMES

With the curve binding ready to accept keyframes, you need to create the keyframes. A keyframe is referred to as an ObjectReferenceKeyframe. You need an ObjectReferenceKeyframe for every time the sprite changes in the animation. You also need to assign where the keyframe will fall along the timeline, and the sprite that the keyframe represents.

The last thing to keep in mind while creating the keyframes is that if the animation should pingpong, then you need twice as many keyframes! Then, after creating the animation forwards, you will need to create the animation backwards to create the ping pong effect.

```
//Create the new (empty) curve binding
    EditorCurveBinding curveBinding = new EditorCurveBinding();
        //Assign it to change the sprite renderer
    curveBinding.type = typeof(SpriteRenderer);
        //Assign it to alter the sprite of the sprite renderer
    curveBinding.propertyName = "m_Sprite";

        //Create a container for all of the keyframes
    ObjectReferenceKeyframe[] keyFrames;

        //Determine how many frames there will be if we are or are not
pingponging
    if (!pingPong)
      keyFrames = new ObjectReferenceKeyframe[frameCount + 1];
        else
      keyFrames = new ObjectReferenceKeyframe[frameCount*2 + 1];

        //Keep track of what frame number we are on
    int frameNumber = 0;

        //Loop from start to end, incrementing frameNumber as we go
    for (int i = startFrame; i < endFrame + 1; i++, frameNumber++)
    {
            //Create an empty keyframe
      ObjectReferenceKeyframe tempKeyFrame = new ObjectReferenceKeyframe();
            //Assign it a time to appear in the animation
      tempKeyFrame.time = frameNumber * frameLength;
            //Assign it a sprite
            tempKeyFrame.value = sprites[i];
                    //Place it into the container for all the keyframes
            keyFrames[frameNumber] = tempKeyFrame;
        }

            //If we are pingponging this animation
        if (pingPong)
        {
                    //Create keyframes starting at the end and going backwards
                    //Continue to keep track of the frame number
            for(int i = endFrame; i >= startFrame; i--, frameNumber++)
            {
                ObjectReferenceKeyframe tempKeyFrame = new ObjectReferenceKeyframe();
                tempKeyFrame.time = frameNumber * frameLength;
                tempKeyFrame.value = sprites[i];
                keyFrames[frameNumber] = tempKeyFrame;
            }
        }

            //Create the last sprite to stop it from switching quickly from the last
    frame to the first one
        ObjectReferenceKeyframe lastSprite = new ObjectReferenceKeyframe();
        lastSprite.time = frameNumber * frameLength;
        lastSprite.value = sprites[startFrame];
        keyFrames[frameNumber] = lastSprite;
```

RETURNING THE CLIP

After the keyframes are created, they can be put onto the curve created earlier. The name of the clip can also be set, and the clip can be created. Lastly, you can return the clip back to the calling function.

```
            //Create the last sprite to stop it from switching quickly from the last
    frame to the first one
        ObjectReferenceKeyframe lastSprite = new ObjectReferenceKeyframe();
        lastSprite.time = frameNumber * frameLength;
        lastSprite.value = sprites[startFrame];
        keyFrames[frameNumber] = lastSprite;

            //Assign the name
        clip.name = clipName;

            //Apply the curve
```

```
AnimationUtility.SetObjectReferenceCurve(clip, curveBinding, keyFrames);

        //Creat the clip
AssetDatabase.CreateAsset(clip, ("Assets/" + clipName + ".anim"));

        //return the clip
return clip;
```

With the clip returned, you can use it in the original function.

```
for (int i = 0; i < numAnimations; i++)
{
   //Create animation clip
   AnimationClip tempClip = CreateClip(selectedObject, animationNames[i], startFrames[i], endFrames[i],
                   clipFrameRate[i], clipTimeBetween[i], pingPong[i]);
   if (loop[i])
   {
      AnimationClipSettings settings = AnimationUtility.GetAnimationClipSettings(tempClip);
      settings.loopTime = true;
      settings.loopBlend = true;
      AnimationUtility.SetAnimationClipSettings(tempClip, settings);
   }

   controller.AddMotion(tempClip);
}
```

STEP SEVEN: FINISHING TOUCHES

Lastly before this tool is finished, you need to make sure that you are only going through all this effort if the object is still around! Inside of the OnGUI function, check that the selected object is not null before doing anything!

Also, you want to make sure to close the window if unity enters or exits play mode. This is because unity will automatically drop the reference to the object upon entering/exiting playmode. By keeping the window open with no object, you could get a significant amount of erros!

```
void OnFocus()
{
        if (EditorApplication.isPlayingOrWillChangePlaymode)
                this.Close ();
}
```

WRAP-UP

You now have the tools to:

1. Manipulate Animator Controllers
2. Manipulate Animation Clips
3. Show new unity windows
4. Populate new unity windows

DEFINITIONS

Each definition is hyperlinked to the unity scripting reference if you wish to learn more.

ASSETDATABASE.GETASSETPATH(OBJECT ASSETOBJECT)

public static string GetAssetPath(int instanceID);

public static string GetAssetPath(Object assetObject);

Parameters

instanceID	The instance ID of the asset.
assetObject	A reference to the asset.

Returns

String	The asset path name, or null, or an empty string if the asset does not exist.

Description

Returns the path name relative to the project folder where the asset is stored.

All paths are relative to the project folder, for example: "Assets/MyTextures/hello.png".

ASSETDATABASE.LOADASSETATPATH(STRING ASSETPATH, TYPE TYPE)

public static Object **LoadAssetAtPath**(string **assetPath**, Type **type**);

Parameters

assetPath	Path of the asset to load.
Typetest	Data type of the asset.

Description

Returns the first asset object of type type at given path assetPath.

Some asset files may contain multiple objects. (such as a Maya file which may contain multiple Meshes and GameObjects). All paths are relative to the project folder, for example: "Assets/MyTextures/hello.png".

Note:

The assetPath parameter is not case sensitive.

ALL asset names & paths in Unity use forward slashes, even on Windows.

This returns only asset object that is visible in the Project view.

EDITORUTILITY.DISPLAYDIALOG(STRING TITLE, STRING MESSAGE, STRING OK, STRING CANCEL =””)

public static bool **DisplayDialog**(string **title**, string **message**, string **ok**, string **cancel** = "");

Parameters

title	The title of the message box.
message	The text of the message.
ok	Label displayed on the OK dialog button.
cancel	Label displayed on the Cancel dialog button.

Returns

Bool	True of the ok button is pressed.

Description

Displays a modal dialog.

Use it for displaying message boxes in the editor.

ok and cancel are labels to be displayed on the dialog buttons. If cancel is empty (the default), then only one button is displayed. DisplayDialog returns true if ok button is pressed.

DESTROYIMMEDIATE(OBJECT OBJ)

public static void DestroyImmediate(Object obj, bool allowDestroyingAssets = false);

Parameters

obj	Object to be destroyed.
allowDestroyingAssets	Set to true to allow assets to be destroyed.

Description

Destroys the object obj immediately. You are strongly recommended to use Destroy instead.

This function should only be used when writing editor code since the delayed destruction will never be invoked in edit mode. In game code you should use Object.Destroy instead. Destroy is always delayed (but executed within the same frame). Use this function with care since it can destroy assets permanently! Also note that you should never iterate through arrays and destroy the elements you are iterating over. This will cause serious problems (as a general programming practice, not just in Unity).

EDITORWINDOW

Description

Derive from this class to create an editor window.

Create your own custom editor window that can float free or be docked as a tab, just like the native windows in the Unity interface.

Editor windows are typically opened using a menu item.

EDITORWINDOW.GETWINDOW(TYPE T)

public static EditorWindow GetWindow(Type t, bool utility = false, string title = null, bool focus = true);

Parameters

t	The type of the window. Must derive from EditorWindow.
utility	Set this to true to create a floating utility window. False creates a normal window.

title	If GetWindow creates a new window, it will get this title. If this value is null, use the class name as the title.

Description

Returns the first EditorWindow of type t which is currently on the screen.

If there is none, creates and shows new window and returns the instance of it.

EDITORWINDOW.SHOW()

public void Show();

public void Show(bool immediateDisplay);

Description

Show the EditorWindow.

OBJECT.NAME

public string name;

The name of the object.

Components share the same name with the game object and all attached components.

PREFABUTILITY.CREATEEMPTYPREFAB(STRING PATH)

public static Object CreateEmptyPrefab(string path);

Creates an empty prefab at given path.

If a prefab at the path already exists it will be deleted and replaced with an empty prefab. Returns a reference to the prefab.

PREFABUTILITY.INSTANTIATEPREFAB(OBJECT TARGET)

public static Object InstantiatePrefab(Object target);

Instantiates the given prefab.

This is similar to Instantiate but creates a prefab connection to the prefab.

PREFABUTILITY.REPLACEPREFAB(GAMEOBJECT GO, OBJECT TARGETPREFAB)

public static GameObject ReplacePrefab(GameObject go, Object targetPrefab, ReplacePrefabOptions options = ReplacePrefabOptions.Default);

Replaces the targetPrefab with a copy of the game object hierarchy go.

Returns the prefab game object after it has been created. If connectToPrefab is enabled go will be made an instance of the created prefab.

SELECTION.ACTIVEOBJECT

public static Object activeObject;

Description

Returns the actual object selection. Includes prefabs, non-modifyable objects.

When working with objects that are primarily in a scene, it is strongly recommended to use Selection.activeTransform instead.

SELECTION.GAMEOBJECTS

public static GameObject[] gameObjects;

Returns the actual game object selection. Includes prefabs, non-modifyable objects.

When working with objects that are primarily in a scene, it is strongly recommended to use Selection.transforms instead.

COMPLETE CODE

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class MakeAnimations : EditorWindow
{

    //Will hold the object that the user has selected when the script is run
```

```
public static Object selectedObject;

//Will store how many animations will be created
int numAnimations;
//Name of the controller to be created
string controllerName;
//name of each of the animations to be created
string[] animationNames = new string[100];

//The frame rate for each animation
float[] clipFrameRate = new float[100];
//The time between each animation
float[] clipTimeBetween = new float[100];
//What frame each animation starts at
int[] startFrames = new int[100];
//What frame each animation ends at
int[] endFrames = new int[100];
//If each animation should pingpong
bool[] pingPong = new bool[100];
//if each animation should loop
bool[] loop = new bool[100];


[MenuItem("Project Tools/2D Animations")]
static void Init()
{
  //Grab the active object
  selectedObject = Selection.activeObject;

  //If the object doesn't exist, do nothing
  if (selectedObject == null)
    return;

  //Otherwise, create a new window
  MakeAnimations window = (MakeAnimations)EditorWindow.GetWindow(typeof(MakeAnimations));

  //Show the window
  window.Show();
```

```
}

void OnGUI()
{
    if (selectedObject != null)
    {
        //Display the objects name that the animations will be created from
        EditorGUILayout.LabelField("Animations for " + selectedObject.name);
        //Create a space
        EditorGUILayout.Separator();
        //Get the name for the animation controller
        controllerName = EditorGUILayout.TextField("Controller Name", controllerName);
        //Determine how many animations there will be
        numAnimations = EditorGUILayout.IntField("How many animations?", numAnimations);
        //Loop through each theoretical animation
        for (int i = 0; i < numAnimations; i++)
        {
            //Determine a name for the animation
            animationNames[i] = EditorGUILayout.TextField("Animation Name", animationNames[i]);

            //Start a section where the following items will be displayed horizontally instead of vertically
            EditorGUILayout.BeginHorizontal();
            //Determine the start frame for the animation
            startFrames[i] = EditorGUILayout.IntField("Start Frame", startFrames[i]);
            //Determine the end frame for the animation
            endFrames[i] = EditorGUILayout.IntField("End Frame", endFrames[i]);
            //End the section where the previous items are displayed horitontally instead of vertically
            EditorGUILayout.EndHorizontal();

            //Start a section where the following items will be displayed horizontally instead of vertically
            EditorGUILayout.BeginHorizontal();
            //Determine the frame rate for the animation
            clipFrameRate[i] = EditorGUILayout.FloatField("Frame Rate", clipFrameRate[i]);
            //Determine the space between each keyframe
            clipTimeBetween[i] = EditorGUILayout.FloatField("Frame Spacing", clipTimeBetween[i]);
            //End the section where the previous items are displayed horitontally instead of vertically
            EditorGUILayout.EndHorizontal();
    //Start a section where the following items will be displayed horizontally instead of vertically
    EditorGUILayout.BeginHorizontal();
    //Create a checkbox to determine if this animation should loop
    loop[i] = EditorGUILayout.Toggle("Loop", loop[i]);
    //Create a checkbox to determine if this animation should pingpong
    pingPong[i] = EditorGUILayout.Toggle("Ping Pong", pingPong[i]);
    //End the section where the previous items are displayed horitontally instead of vertically
    EditorGUILayout.EndHorizontal();

    //Create a space
    EditorGUILayout.Separator();

}

//Create a button with the label "Create"
if (GUILayout.Button("Create"))
{
    //Create an animator controller
    UnityEditor.Animations.AnimatorController controller =
                    UnityEditor.Animations.AnimatorController.CreateAnimatorControllerAtPath(("Assets/" +
                    controllerName + ".controller"));

    for (int i = 0; i < numAnimations; i++)
    {
        //Create animation clip
        AnimationClip tempClip = CreateClip(selectedObject, animationNames[i], startFrames[i], endFrames[i],
                        clipFrameRate[i], clipTimeBetween[i], pingPong[i]);

        //Detemine if the clip should loop
        if (loop[i])
        {
            //If so, capture the settings of the clip
            AnimationClipSettings settings = AnimationUtility.GetAnimationClipSettings(tempClip);

            //Set the looping to true
            settings.loopTime = true;
            settings.loopBlend = true;
```

```
                        //Apply the settings to the clip
                        AnimationUtility.SetAnimationClipSettings(tempClip, settings);
                    }

                    //Add the clip to the Animator Controller
                    controller.AddMotion(tempClip);
                }
            }
        }
    }

    public AnimationClip CreateClip(Object obj, string clipName, int startFrame, int endFrame, float frameRate, float
timeBetween, bool pingPong)
    {
        //Get the path to the object
        string path = AssetDatabase.GetAssetPath(obj);

        //Extract the sprites
        Object[] sprites = AssetDatabase.LoadAllAssetsAtPath(path);

        //Determine how many frames, and the length of each frame
        int frameCount = endFrame - startFrame + 1;
        float frameLength = 1f / timeBetween;

        //Create a new (empty) animation clip
        AnimationClip clip = new AnimationClip();

        //Set the framerate for the clip
        clip.frameRate = frameRate;

        //Create the new (empty) curve binding
        EditorCurveBinding curveBinding = new EditorCurveBinding();
        //Assign it to change the sprite renderer
        curveBinding.type = typeof(SpriteRenderer);
        //Assign it to alter the sprite of the sprite renderer
        curveBinding.propertyName = "m_Sprite";
```

```
//Create a container for all of the keyframes
ObjectReferenceKeyframe[] keyFrames;

//Determine how many frames there will be if we are or are not pingponging
if (!pingPong)
   keyFrames = new ObjectReferenceKeyframe[frameCount + 1];
else
   keyFrames = new ObjectReferenceKeyframe[frameCount*2 + 1];

//Keep track of what frame number we are on
int frameNumber = 0;

//Loop from start to end, incrementing frameNumber as we go
for (int i = startFrame; i < endFrame + 1; i++, frameNumber++)
{
   //Create an empty keyframe
   ObjectReferenceKeyframe tempKeyFrame = new ObjectReferenceKeyframe();
   //Assign it a time to appear in the animation
   tempKeyFrame.time = frameNumber * frameLength;
   //Assign it a sprite
   tempKeyFrame.value = sprites[i];
   //Place it into the container for all the keyframes
   keyFrames[frameNumber] = tempKeyFrame;
}

//If we are pingponging this animation
if (pingPong)
{
   //Create keyframes starting at the end and going backwards
   //Continue to keep track of the frame number
   for(int i = endFrame; i >= startFrame; i--, frameNumber++)
   {
      ObjectReferenceKeyframe tempKeyFrame = new ObjectReferenceKeyframe();
      tempKeyFrame.time = frameNumber * frameLength;
      tempKeyFrame.value = sprites[i];
      keyFrames[frameNumber] = tempKeyFrame;
   }
}
```

```
        //Create the last sprite to stop it from switching quickly from the last frame to the first one
        ObjectReferenceKeyframe lastSprite = new ObjectReferenceKeyframe();
        lastSprite.time = frameNumber * frameLength;
        lastSprite.value = sprites[startFrame];
        keyFrames[frameNumber] = lastSprite;

        //Assign the name
        clip.name = clipName;

        //Apply the curve
        AnimationUtility.SetObjectReferenceCurve(clip, curveBinding, keyFrames);

        //Creat the clip
        AssetDatabase.CreateAsset(clip, ("Assets/" + clipName + ".anim"));

        //return the clip
        return clip;

    }
}
```

ON YOUR OWN

Make sure that the following tool follows the User Centered Ideas discussed in class, including but not limited to Mental Loads, Action Cycle, and Design guidelines!

Create a utility that will allow the user to select a sprite sheet (which has been sliced into multiple sprites), and then create animations from that sprite sheet.

Each animation should have its own options for:

- Looping
- Ping-ponging
- Start Frame
- End Frame

Make sure that the end frame cannot be higher than the number of frames the sprite sheet has been broken up into! (Hint: you may to have to research the function you use to load the sprites)

Once the animations are created, create a sprite in the scene view at 0,0,0 with the newly created animation controller added to it.

Make the sprite a prefab in the main assets folder.

POST-LAB

1. What is the difference between EditorGUI and EditorGUILayout?
2. What is the OnGUI function?
3. Do variables have to be initialized before they can be used for GUI items?
4. Instead of hard-coding the prefabs to go into the asset folder, how could you allow the user to choose what folder to create the prefab?
5. What is the difference between Instantiate and InstantiatePrefab?
6. What are some things you can do with EditorUltility.DisplayDialoge?
7. How would you use the utility created in this section?
8. Why must you refresh the asset database after creating the prefabs?