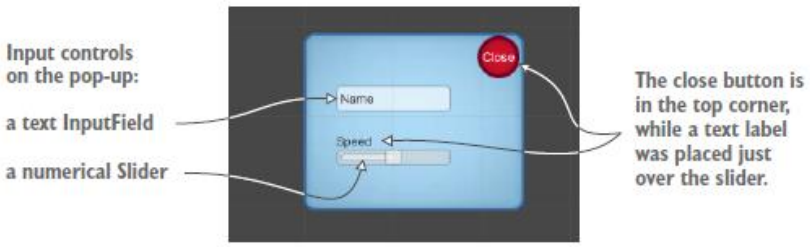
# Unity Game Engine

Introduction to Unity – 2D GUI in 3D Game (2)



# Popup Window & Items

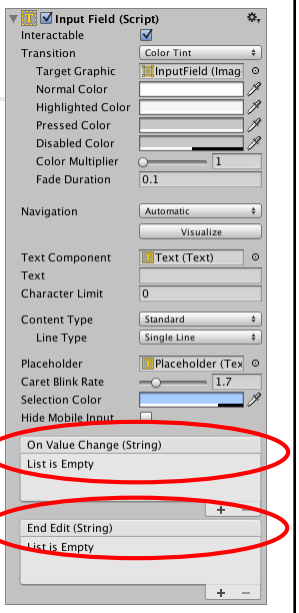



- GameObject > UI > InputField
- GameObject > UI > Text
- GameObject > UI > Slider

Make them children of the pop-up window.

# InputField

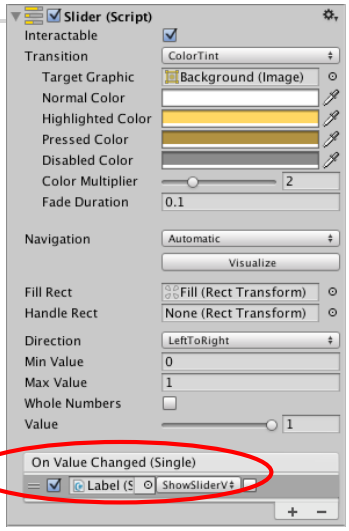
- **Input Field :** An Input Field is used to make the text of a Text Element editable by the user. It has a UnityEvent (**OnValueChanged**) to define what it will do when the text content is changed, and an another (**OnEndEdit)** to define what it will do when the user has finished editing it.






# Slider

- **Slider :** A Slider has a decimal number **Value** that the user can drag between a minimum and maximum value. It can be either horizontal or vertical. It also has a **OnValueChanged** UnityEvent to define what it will do when the value is changed.
- Slider settings in Inspector
  - Min, Max, Whole Numbers (integer values?), Value ( current numeric value of the slider.
- https://docs.unity3d.com/Manual/script-Slider.html



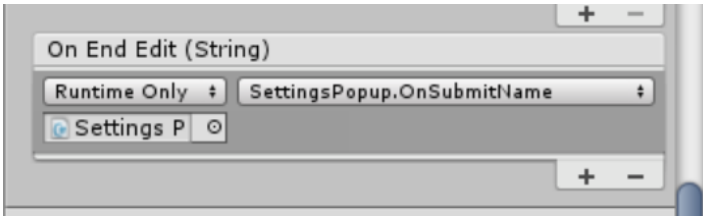
# Modify SettingsPopup.cs

- Add new methods to it.

```
using UnityEngine;
using System.Collections;
public class SettingsPopup : MonoBehaviour {
  public void Open() {
    gameObject.SetActive(true);
  }
  public void Close() {
    gameObject.SetActive(false);
  }
  public void OnSubmitName(string name) {
    Debug.Log(name);
  }
  public void OnSpeedValue(float speed) {
    Debug.Log("Speed: " + speed);
  }
}
```

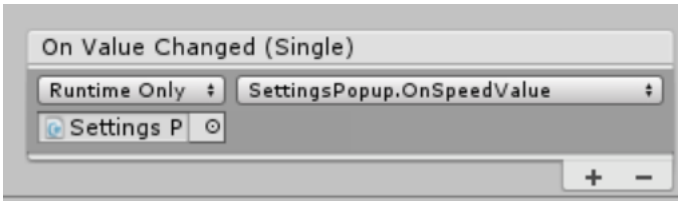
# UnityEvent Handler - InputField

- Assign **OnSubmitName()** of the pop-up to the **OnEndEdit** UnityEvent of the InputField.
- **OnEndEdit** is the Unity Event to call when editing has ended.



# UnityEvent Handler - Slider

- Assign **OnSpeedValue()** of the pop-up to the **OnValueChanged** UnityEvent of the Slider.
- **OnValueChanged** is a callback executed when the value of the slider is changed.






# Saving Game Information

- Large game data – save into files using C# File I/O
- Small game data (eg settings) – can use **PlayerPrefs**
  - https://docs.unity3d.com/ScriptReference/PlayerPrefs.html
  - See where (in which folder) the information is stored depending on the platform.

# `PlayerPrefs` Class

- Stores and accesses player preferences between game sessions.
- Static functions:

| | |
|---|---|
| **DeleteAll** | Removes all keys and values from the preferences. Use with caution. |
| **DeleteKey** | Removes key and its corresponding value from the preferences. |
| **GetFloat** | Returns the value corresponding to key in the preference file if it exists. |
| **GetInt** | Returns the value corresponding to key in the preference file if it exists. |
| **GetString** | Returns the value corresponding to key in the preference file if it exists. |
| **SetFloat** | Sets the value of the preference identified by key. |
| **SetInt** | Sets the value of the preference identified by key. |
| **SetString** | Sets the value of the preference identified by key. |



# PlayerPrefs Class

- PlayerPrefs provide simple commands to get and set named values (it works a lot like a hash table or dictionary).
- See how to use `SetFloat` and `GetFloat`.

```
public static void SetFloat(string key, float value);
public static float GetFloat(string key, float defaultValue = 0.0F);
```

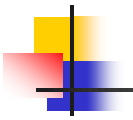
```
PlayerPrefs.SetFloat("speed", 1.5);
print(PlayerPrefs.GetFloat("speed"));
```

# Modify SettingsPopup.cs

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;
public class SettingsPopup : MonoBehaviour {
  [SerializeField] private Slider speedSlider;
  void Start() {
    speedSlider.value =
          PlayerPrefs.GetFloat("speed", 1);
  }
  ...
  public void OnSubmitName(string name) {
    Debug.Log(name);
  }
  public void OnSpeedValue(float speed) {
    PlayerPrefs.SetFloat("speed", speed);
  }
}
```

# Updating the Game

- The main game and UI should interact back and forth. (e.g update the number of enemies killed).
- 2 approaches:
  - Inter-object references : scripts refers to each other. Tightly coupled the scene and HUD.
  - Via events : alert the UI through events. Keep the scene and HUD independent.



# EventSystem

- Whenever a canvas is created in a scene, an Event System game object is also created with the canvas automatically.
- It handles inputs from the user and sends events to the affected UI element/UI elements in the scene.



# Event Handling through Message Broadcasting

- See http://wiki.unity3d.com/index.php/CSharpMessenger_Extended
- Download two utility scripts:
  - **Messenger.cs**: Implements broadcasting message system
  - **GameEvent.cs** : defines constants for event messages

# Messenger.cs

- **Registering an event listener**
  `Messenger<type>.AddListener();`
- **Unregistering an event listener**
  `Messenger<type>.RemoveListener();`
- **Broadcasting**
  `Messenger.Broadcast<type>();`



# Broadcast Messaging from the Game

- Assume that when an enemy dies, it will emit an event.
  - Modify RayShooter.cs
- The UI controller will respond to that event. Modify UIController.cs as shown below:
  - Delete the entire **`Update()`** method in the UIController.cs since UIController will respond to events only.
  - Register an event listener in the **`Awake()`.**
  - Provide a callback (event listener) function to handle the event.
  - Remove the listener when the script is destroyed.

# Broadcast Messaging from the Game

- Assume that when an enemy dies, it will emit an event.
  - Modify RayShooter.cs
- The UI controller will respond to that event. Modify UIController.cs as shown below:
  - Delete the entire `Update()` method in the UIController.cs since UIController will respond to events only.
  - Register an event listener in the `Awake()`.
  - Provide a callback (event listener) function to handle the event.
  - Remove the listener when the script is destroyed.



# Modify RayShooter.cs

```
...
if (target != null) {
  target.ReactToHit();
  Messenger.Broadcast(GameEvent.ENEMY_HIT);
} else {
...
```

# Handling Messaging from the Game

- Assume that when an enemy dies, it will emit an event.
  - Modify RayShooter.cs
- The UI controller will respond to that event. Modify UIController.cs as shown below:
  - Delete the entire **`Update()`** method in the UIController.cs since UIController will respond to events only.
  - Register an event listener in the **`Awake()`.**
  - Provide a callback (event listener) function to handle the event.
  - Remove the listener when the script is destroyed.



# Modify UIController.cs

```
...
private int _score;

void Awake() {
  Messenger.AddListener(GameEvent.ENEMY_HIT, OnEnemyHit);
}
void OnDestroy() {
  Messenger.RemoveListener(GameEvent.ENEMY_HIT, OnEnemyHit);
}
void Start() {
  _score = 0;
  scoreLabel.text = _score.ToString();
  settingsPopup.Close();
}
private void OnEnemyHit() {
  _score += 1;
  scoreLabel.text = _score.ToString();
}
```

# UIController.cs Explained

- **MonoBehaviour.Awake():**
  - `Awake` is called when the script instance is being loaded.
  - `Awake` is called once.
  - `Awake` is always called before any `Start` functions.
  - The difference between `Awake` and `Start` is that `Start` is called if the script instance is enabled.
- **MonoBehaviour.OnDestroy():** This function is called when the MonoBehaviour will be destroyed.



# UIController.cs Explained

- **GameEvent.ENEMY_HIT** : Event name
- **`OnEnemyHit()`** : This listener name. This is invoked when **`GameEvent.ENEMY_HIT`** is broadcasted by RayShooter.cs.

# Broadcast Messaging from the HUD

- The settings pop-up affects the settings of the game.
- Assume that the speed change in the pop-up window affects both the player's and the enemy's speeds.
  - Broadcast message from the pop-up.
    - Modify SettingsPopup.cs.
  - Make the enemy and the player respond to it.
    - Modify WanderingAI.cs for the enemy
    - Modify FPSInput.cs for the player.



# Modify SettingPopup.cs

```
...
public void OnSpeedValue(float speed) {
    PlayerPrefs.SetFloat("speed", 1.5)
    Messenger<float>.Broadcast(GameEvent.SPEED_CHANGED,
                speed);
  }
...
```

When the player changes the speed value through the slider, OnSpeedValue will be invoked.

The SettingPopup script broadcasts GameEvent.SPEED_CHANGED event.

## Modify FPSInput.cs

```
public const float baseSpeed = 6.0f;
...
void Awake() {
  Messenger<float>.AddListener(GameEvent.SPEED_CHANGED,
    OnSpeedChanged);
}
void OnDestroy() {
  Messenger<float>.RemoveListener(
              GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
...
private void OnSpeedChanged(float value) {
  speed = baseSpeed * value;
}
...
```

## Modify WanderingAI.cs

```
...
public const float baseSpeed = 3.0f;
...
void Awake() {
  Messenger<float>.AddListener(GameEvent.SPEED_CHANGED,
    OnSpeedChanged);
}
void OnDestroy() {
  Messenger<float>.RemoveListener(
             GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
...
private void OnSpeedChanged(float value) {
    speed = baseSpeed * value;
}
...
```

# Changing the speed of spawned enemies

- Currently the speed value is only updated for enemies already in the scene and not for newly spawned enemies; new enemies aren't created at the correct speed setting.
- Set the speed on spawned enemies.
  - Hint: add a **`GameEvent.SPEED_CHANGED`** listener to SceneController.cs, because that script is where enemies are spawned from.



# GameEvent.cs

```
public static class GameEvent {
  public const string ENEMY_HIT = "ENEMY_HIT";
  public const string SPEED_CHANGED = "SPEED_CHANGED";
}
```

Defines constants for event message names.

# Resources

- Modified scripts
  - FPSInput.cs
  - RayShooter.cs
  - SettingsPopup.cs
  - UIController.cs
  - WanderingAI.cs
- Utility scripts
  - GameEvent.cs
  - Messenger.cs