Implementando la interfaz de usuario

Índice

- Introducción
- Modo inmediato
- Diseño de la interfaz
- Construyendo la interfaz
- Interacción con la interfaz
- Construyendo un Popup
- Conectando la interfaz con el juego

Introducción

- Vamos a estudiar la implementación de interfaces de usuario para nuestros juegos
- Hay varios tipos de GUI en el juego: los menús iniciales, pantallas de ayuda, interfaces sobre la ventana de juego (HUD, o heads-up display)…

Introducción

- La interfaz de usuario del juego está compuesta por los widgets típicos de cualquier aplicación gráfica (etiquetas, botones, sliders, cuadros de texto, etc)
  - Aunque típicamente la apariencia se ajusta a la estética del juego

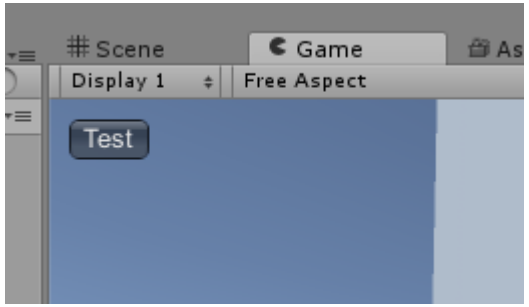
Introducción

- Vamos a añadir una GUI básica a nuestro juego de cartas
- Unity tiene dos formas de construir GUI:
  - Inmediata:
    - Era la forma original de construir interfaces.
    - Consiste en dibujar y consultar cada widget en cada frame.
    - El motor no almacena el estado, si no que hay que proporcionárselo cada vez.
    - La ventaja es que es muy rápido añadir un widget (p.e., un botón) para probar alguna funcionalidad durante el desarrollo
  - Retenida
    - Disponible a partir de Unity 4.6
    - Se definen los componentes de la interfaz y Unity lleva toda la gestión de estados
    - Tiene un potente sistema para ajustar la interfaz a los cambios de resolución y razón de aspecto
    - Permite construir interfaces más complejos

Modo inmediato

- En nuestros juegos usaremos principalmente el modo retenido, pero el modo inmediato es útil durante el desarrollo
- Consiste en llamar a algún método estático de la clase GUI en el método OnGUI:

```
public class ImmediateGUI : MonoBehaviour {
  void OnGUI() {
    if (GUI.Button(new Rect(10, 10, 40, 20), "Test"))
      Debug.Log("Test");
  }
}
```



Modo inmediato

GUI

class in UnityEngine / Implemented in:UnityEngine.IMGUIModule

Description

The GUI class is the interface for Unity's GUI with manual positioning.

See Also: GUI tutorial.

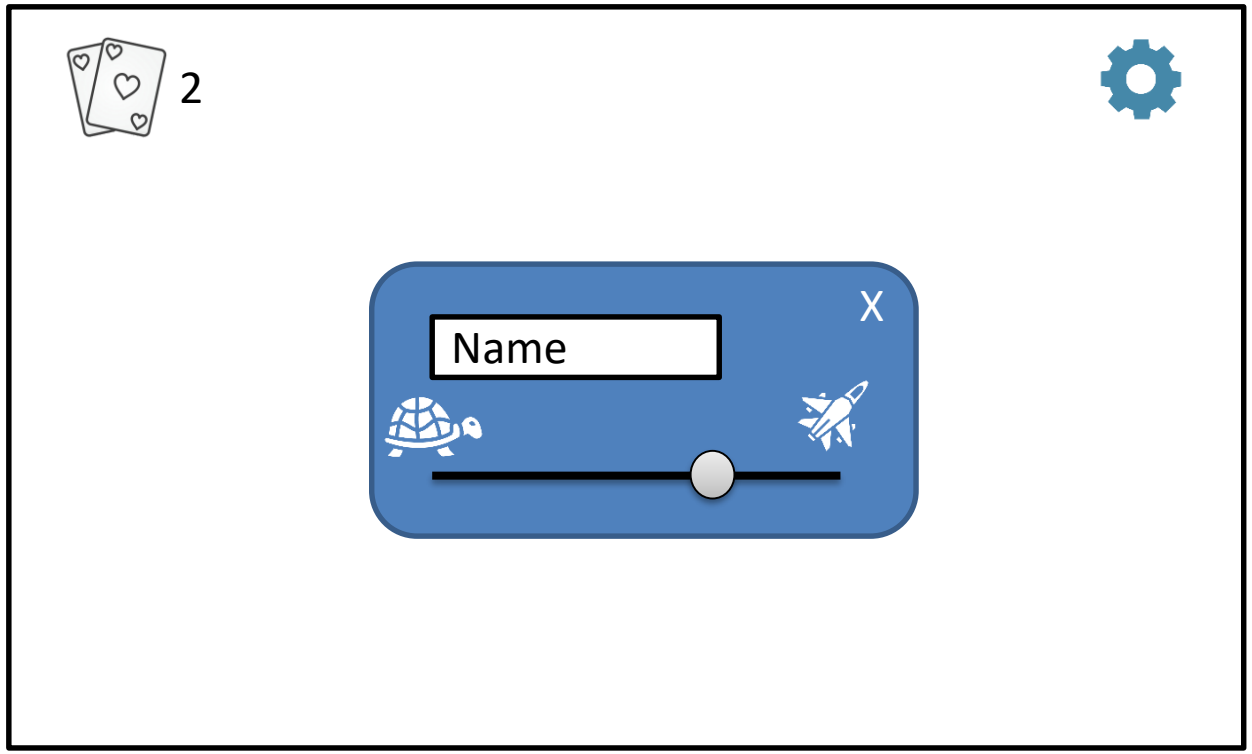
Static Methods

BeginGroup	Begin a group. Must be matched with a call to EndGroup.
BeginScrollView	Begin a scrolling view inside your GUI.
Box	Create a Box on the GUI Layer.
BringWindowToBack	Bring a specific window to back of the floating windows.
BringWindowToFront	Bring a specific window to front of the floating windows.
Button	Make a single press button. The user clicks them and something happens immediately.
DragWindow	Make a window draggable.
DrawTexture	Draw a texture within a rectangle.
DrawTextureWithTexCoords	Draw a texture within a rectangle with the given texture coordinates.
EndGroup	End a group.
EndScrollView	Ends a scrollview started with a call to BeginScrollView.
FocusControl	Move keyboard focus to a named control.
FocusWindow	Make a window become the active window.
GetNameOfFocusedControl	Get the name of named control that has focus.
HorizontalScrollbar	Make a horizontal scrollbar. Scrollbars are what you use to scroll through a document. Most likely, you want to use scrollViews instead.
HorizontalSlider	A horizontal slider the user can drag to change a value between a min and a max.
Label	Make a text or texture label on screen.

https://docs.unity3d.com/ScriptReference/GUI.html

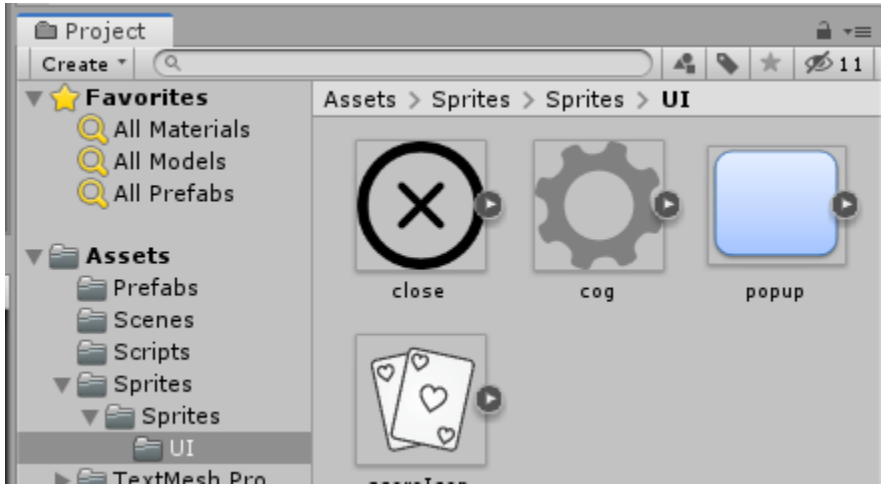
Diseñando la interfaz

- Vamos a añadir una dificultad al juego de cartas, junto a una interfaz de usuario con la puntuación y la posibilidad de configurar la dificultad



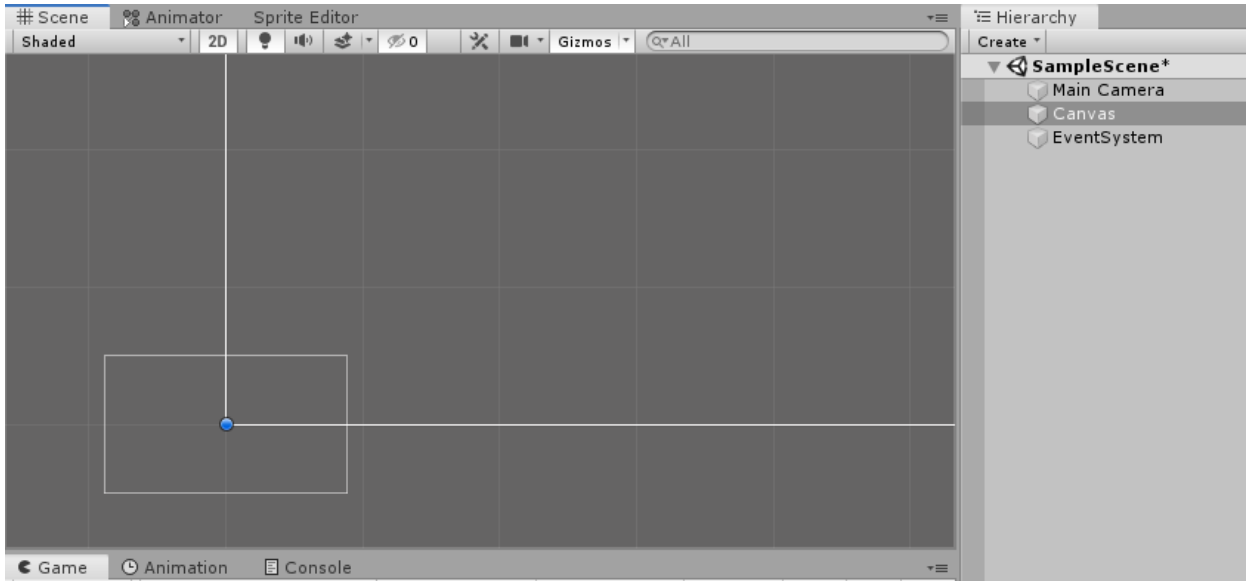
Importando las imágenes

- El primer paso es importar las imágenes de los elementos de la interfaz
  - Como estamos en un proyecto 2D, el tipo de las imágenes por defecto ya es “Sprite (2D and UI)”, por lo que no hay que cambiar nada



Construyendo la interfaz

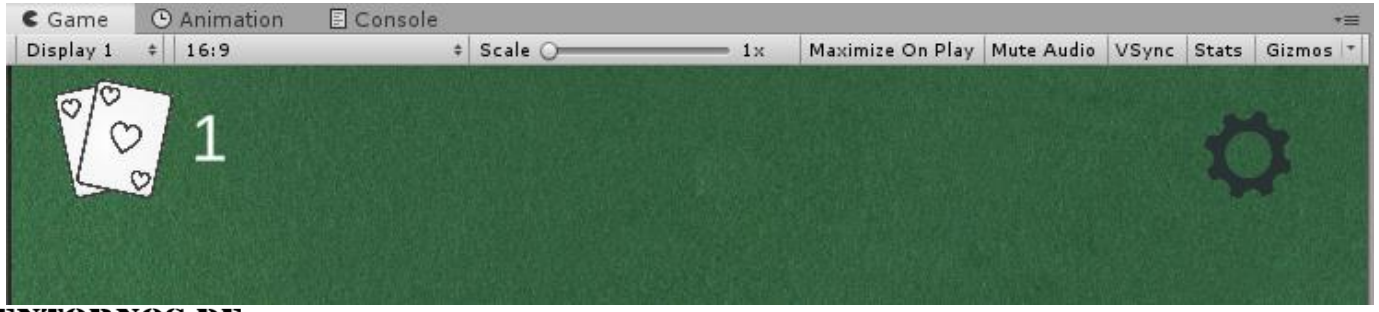
- La base de toda interfaz moderna en Unity es el Canvas, un contenedor que representa la ventana y permite calcular automáticamente la posición de los controles
- Inserta un canvas en la escena

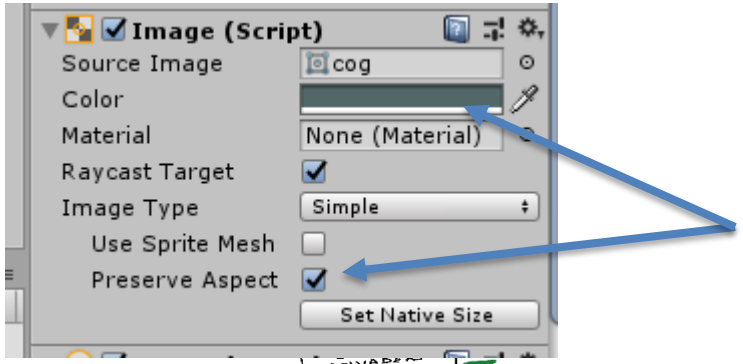


- Selecciona el canvas y pulsa F sobre la ventana de la escena para encuadrarlo
- El canvas aparece como un rectángulo enorme, con una escala de una unidad del mundo por píxel de la ventana
- Unity también crea un objeto EventSystem (que puedes ignorar)

Construyendo la interfaz

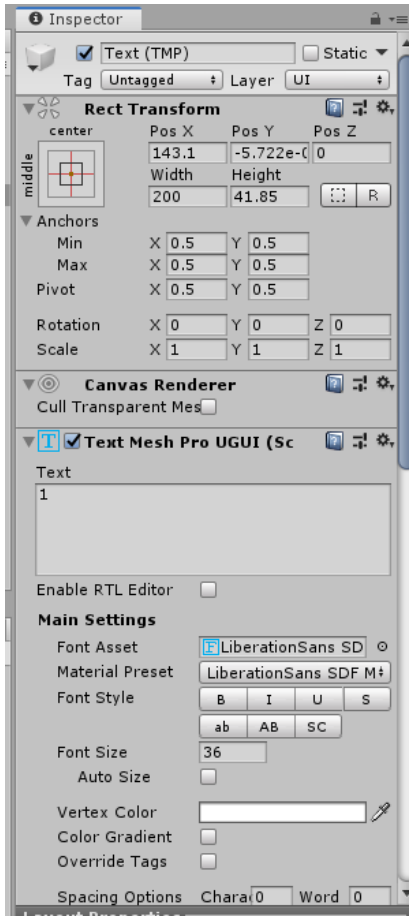
- Vamos a añadir los controles de la parte superior de la ventana. Borra los que había
- Dentro del menú UI, añade una imagen, un texto (TextMeshPro) y un botón
  - Automáticamente Unity los hace hijos del canvas
- Haz que el texto sea hijo de la imagen, para que se muevan juntos
- El botón tiene un componente texto que no hace falta, bórralo
- Sitúa los elementos más o menos en sus esquinas
- Asigna los sprites al campo `Source Image` de cada elemento





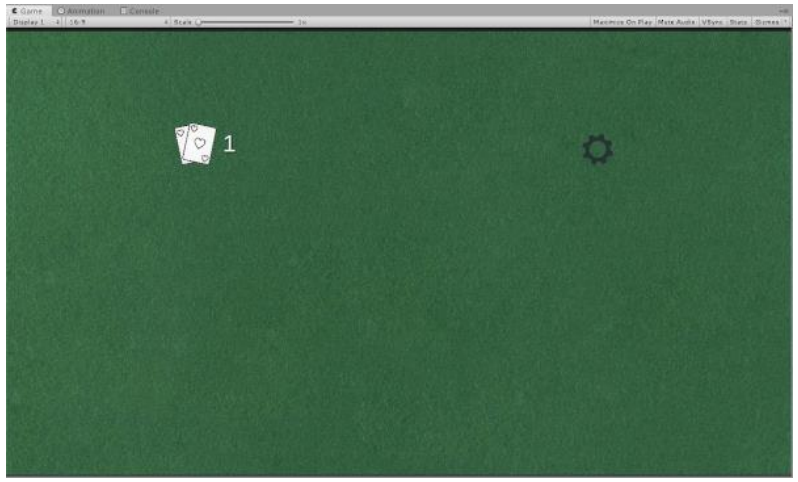
Construyendo la interfaz

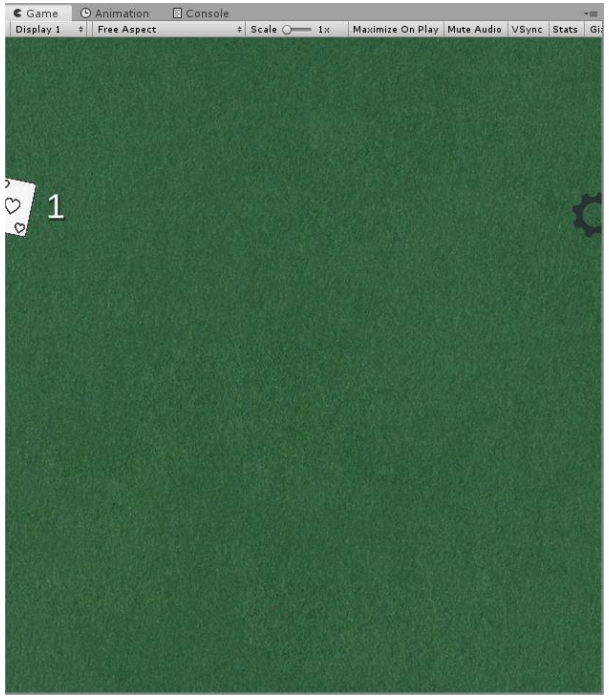
- Con respecto al texto para mostrar la puntuación, configúralo hasta obtener el resultado deseado:



Construyendo la interfaz

- Si ejecutas el juego, verás que los elementos de la interfaz aparecen encima de la escena
- Sin embargo, si cambias el tamaño de la ventana verás que los controles no se ajustan correctamente



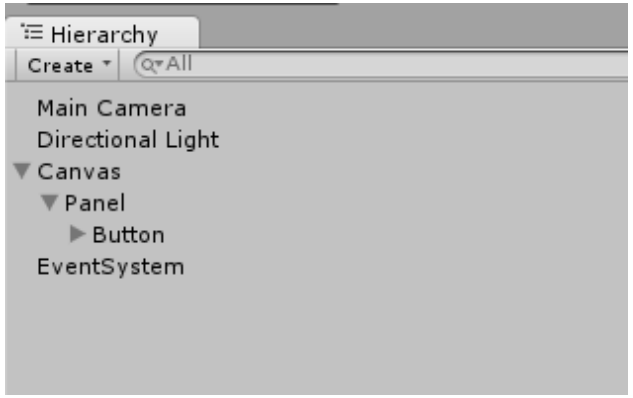


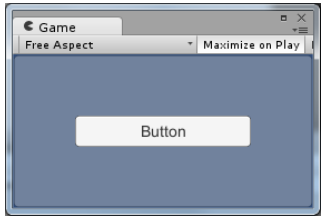
Situando los elementos de la interfaz

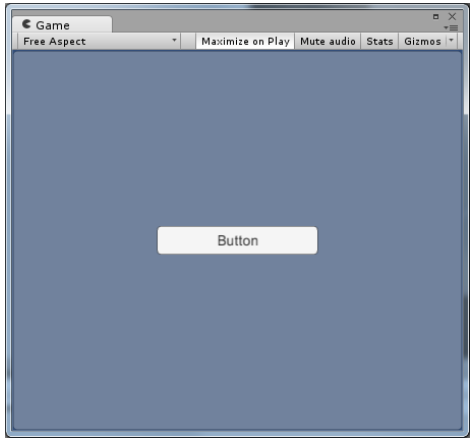
- Es muy importante entender el sistema de composición (*layout*) de Unity para construir interfaces que se adapten a diferentes razones de aspecto
  - Las interfaces de usuario fijas que no aprovechan el espacio disponible o se truncan porque no caben en la ventana dan mala imagen
- Deberíamos considerar el comportamiento de cada elemento de la IU cuando el padre cambia de tamaño
- El lugar de cada elemento de la IU depende del de su padre
  - El comportamiento por defecto de un *Panel* es extenderse para ocupar todo el espacio del padre
  - Por otro lado, el comportamiento por defecto de un botón es mantener su tamaño original y aparecer centrado en el espacio del padre

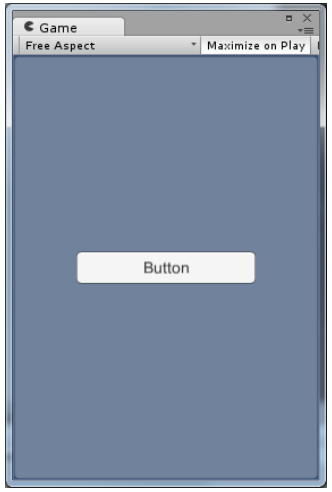
Situando los elementos de la interfaz

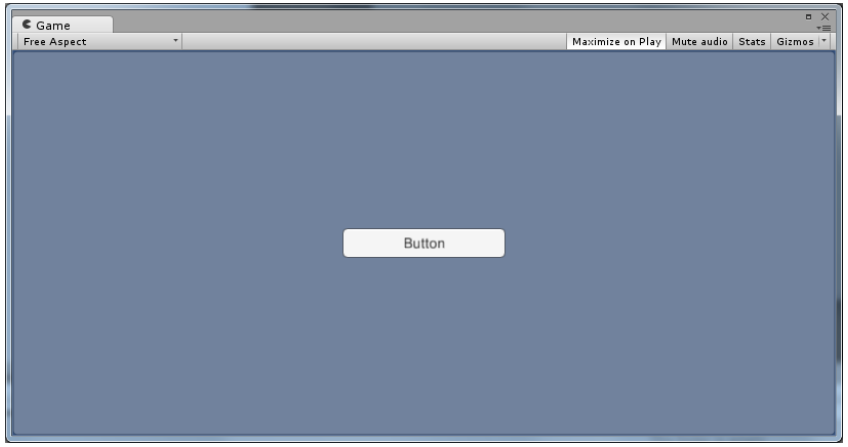
- Comportamiento por defecto del panel y del botón





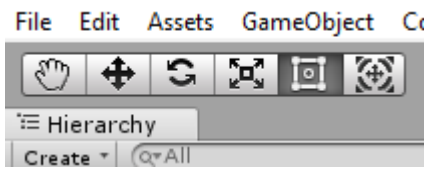


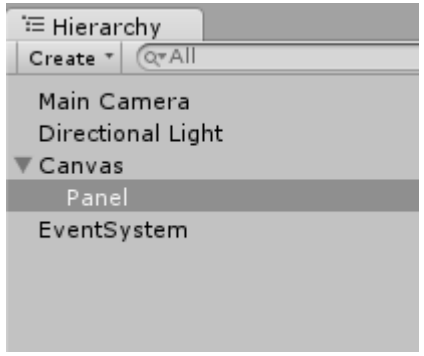


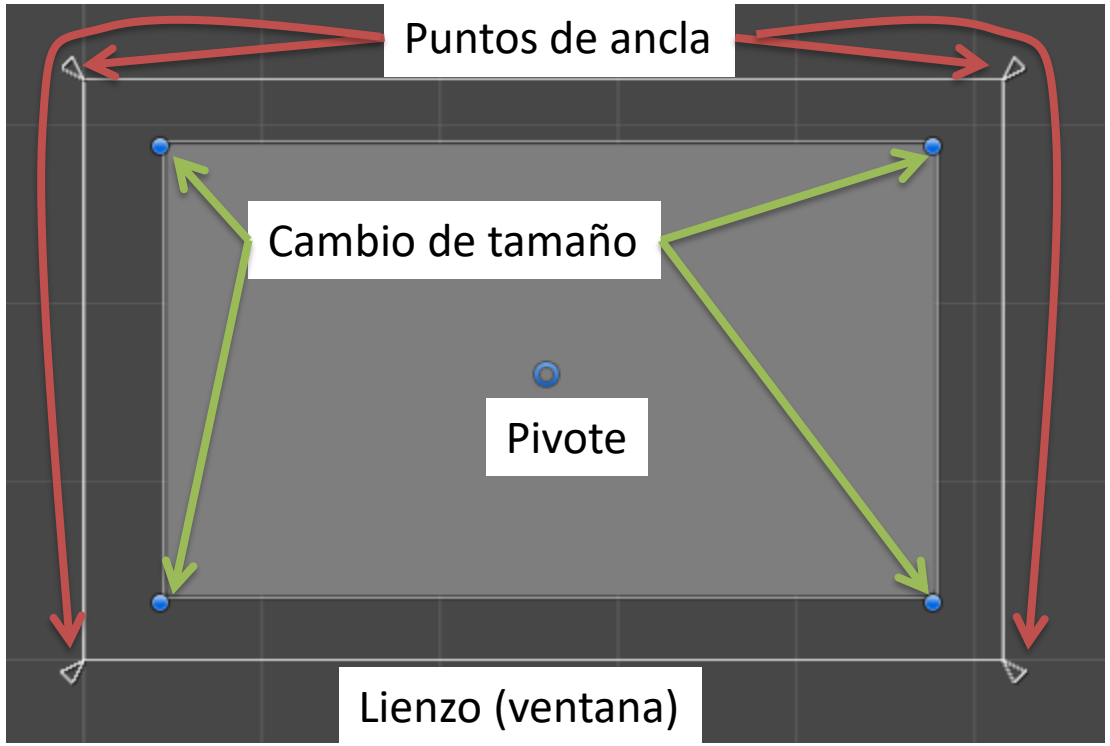


La herramienta *Rect*

- Ofrece todos los elementos para controlar el alineamiento y la posición de los componentes de la IU

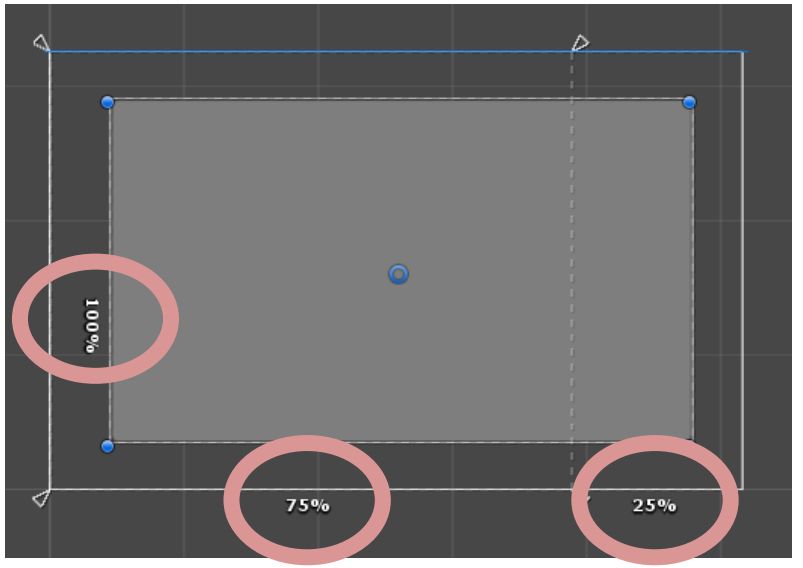


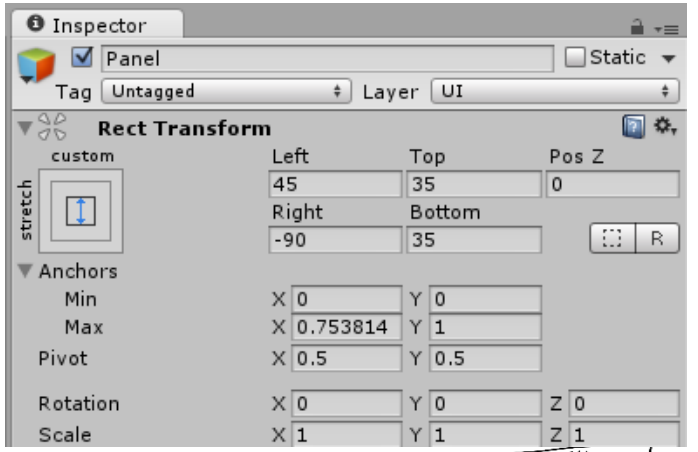




Puntos de ancla (*Anchors*)

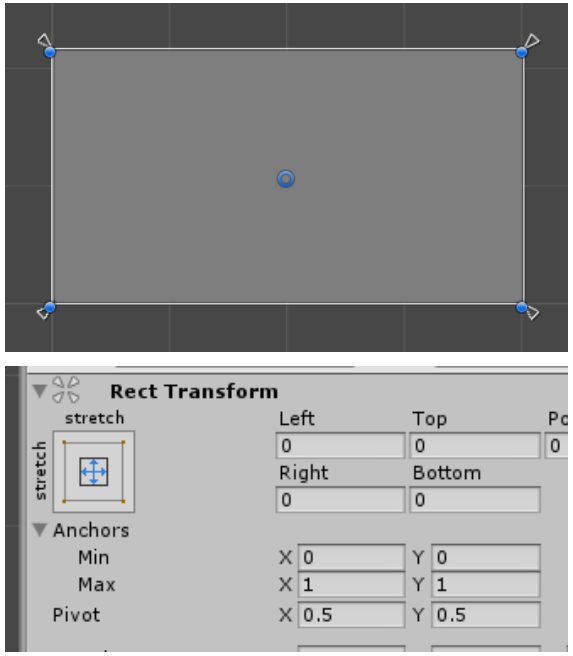
- Definen la posición del hijo con respecto al rectángulo del padre
  - Origen: esquina inferior izquierda
- Se pueden modificar usando los manejadores de la vista de la Escena o en el Inspector:
  - Sus coordenadas van de 0 a 1
  - No pueden salir del rectángulo del padre
  - Cuando el padre cambia de tamaño, los puntos de ancla se mueven proporcionalmente
- Cada esquina del widget va asociado con un punto de ancla



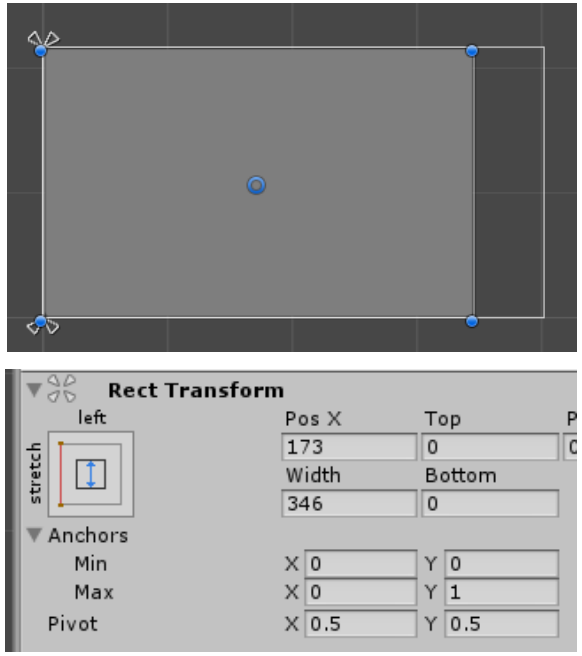


Puntos de ancla (*Anchors*)

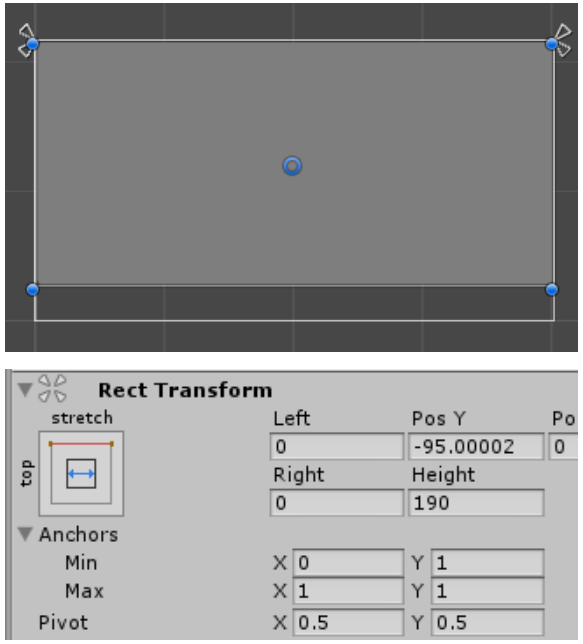
Ejemplos



El panel crece horizontal y verticalmente



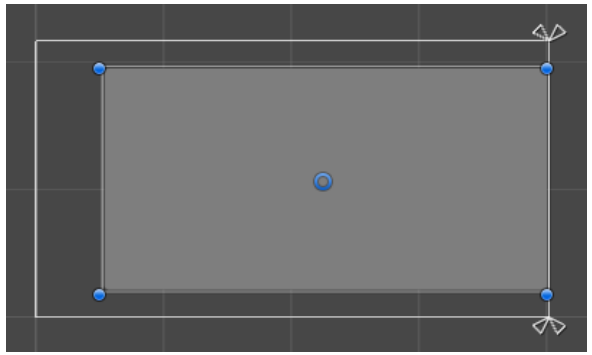
El panel crece verticalmente. Anchura fija. Acoplado al borde izquierdo del lienzo

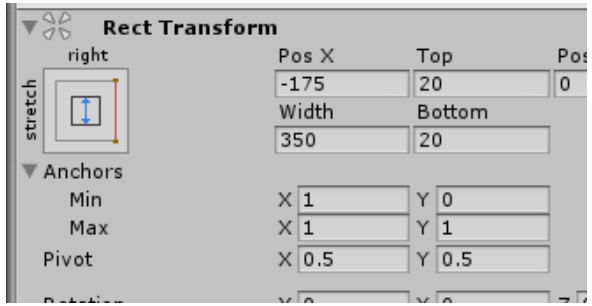


El panel crece horizontalmente. Altura fija. Acoplado al borde superior del lienzo

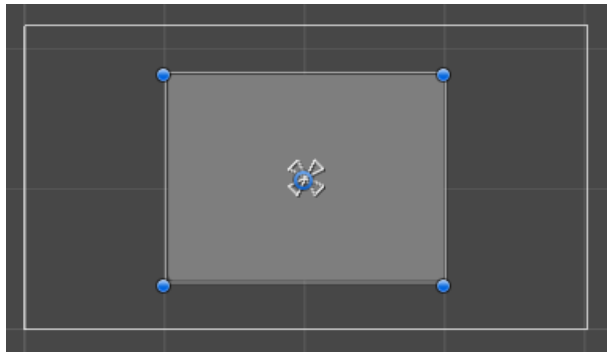
Puntos de ancla (*Anchors*)

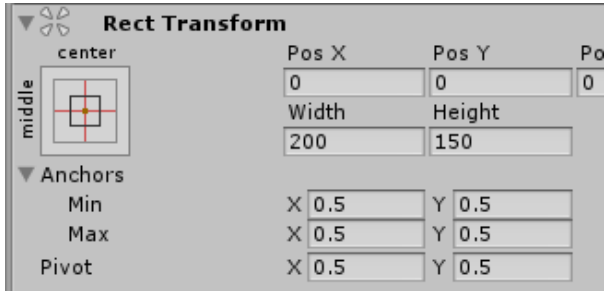
Ejemplos



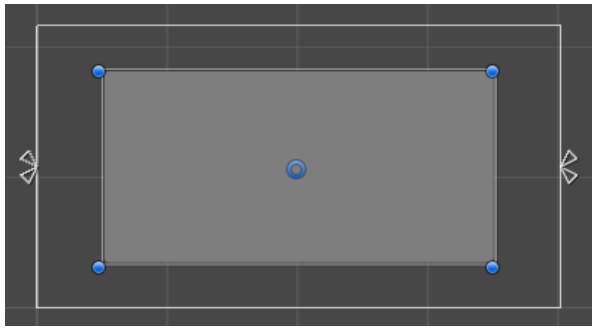


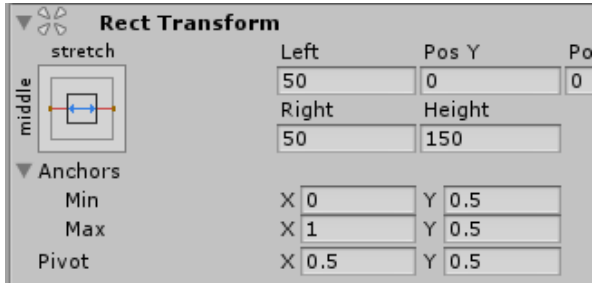
Crece verticalmente. Relleno de 20 pixeles arriba y abajo. Anchura fija. Espacio a la izquierda variable





Siempre centrado. Tamaño fijo.

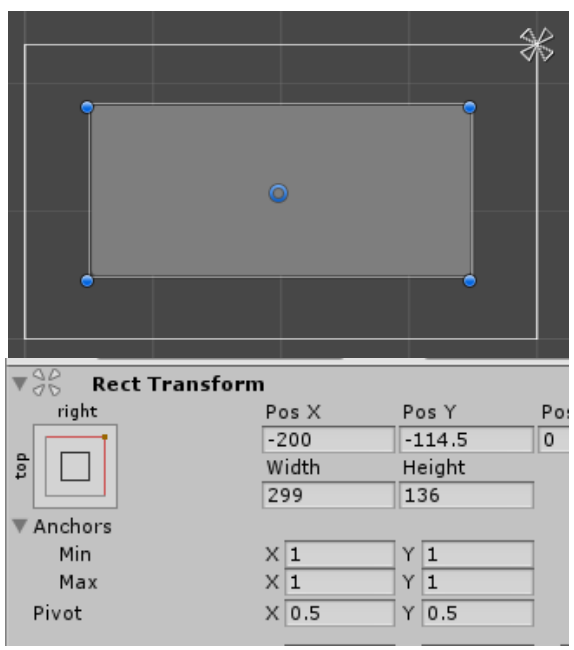




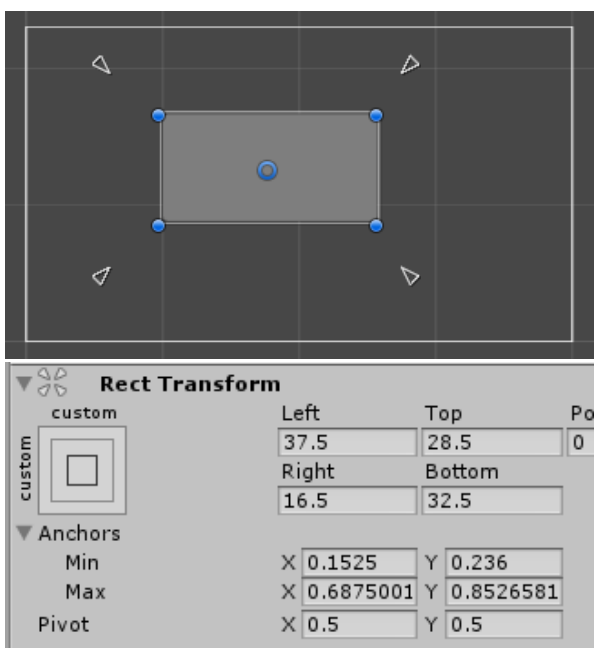
Siempre centrado. Altura fija. Anchura variable. Relleno de 50 píxeles a la izquierda y a la derecha

Puntos de ancla (*Anchors*)

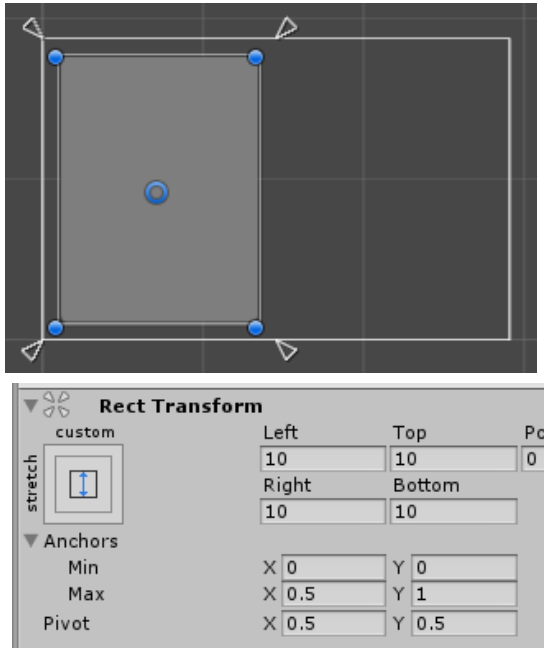
Ejemplos



Tamaño fijo. Se mantiene la distancia a los puntos de ancla



Tamaño variable. La distancia de cada esquina a su punto de ancla se mantiene



El panel siempre ocupa la mitad izquierda del lienzo, con 10 píxeles de margen alrededor

Puntos de ancla (*Anchors*)

- Resumen:
  - Un punto: tamaño fijo

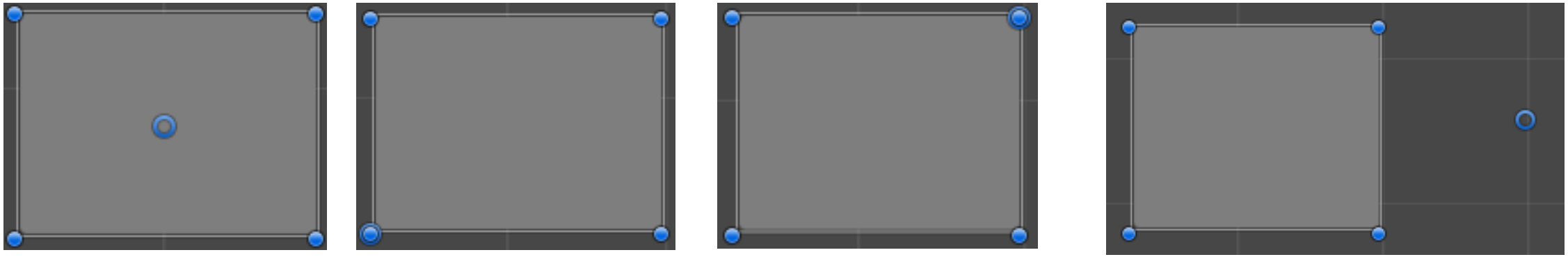
  - Dos puntos:
    - Crece sólo horizontalmente

    - Crece sólo verticalmente

  - Cuatro puntos:
    - Crece horizontal y verticalmente

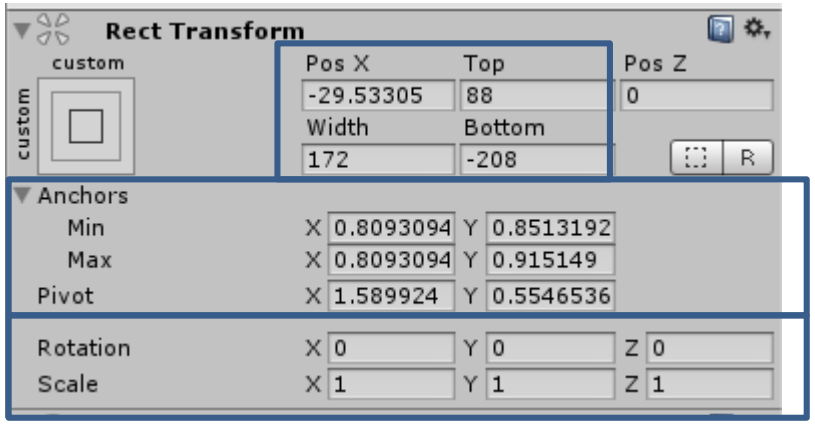
Pivote

- El pivote se muestra como un anillo azul
- Se define con respecto al rectángulo del widget
  - (0,0) es la esquina inferior izquierda, (1,1) es la esquina superior derecha
  - Puede estar fuera de su rectángulo
- Define el centro que se usa para las transformaciones (rotaciones, escalados), y también define la posición del elemento con respecto a sus puntos de ancla



El panel *Rect Transform*

- Muestra la posición actual de los puntos de ancla, el pivote, el rectángulo y sus transformaciones

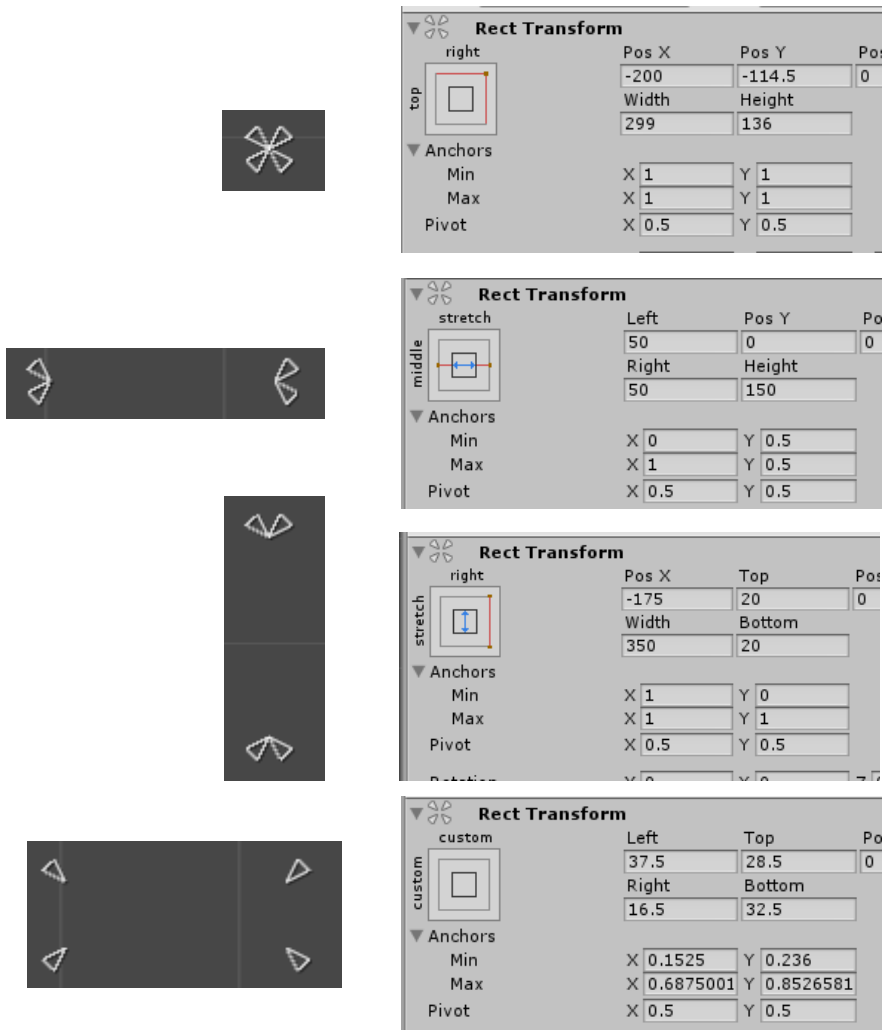


La etiqueta y significado de estos elementos cambian según el modo de los puntos de ancla

Posición de los puntos de ancla y pivote

Transformaciones

El panel *Rect Transform*



`Width`, `Height`: dimensiones fijas

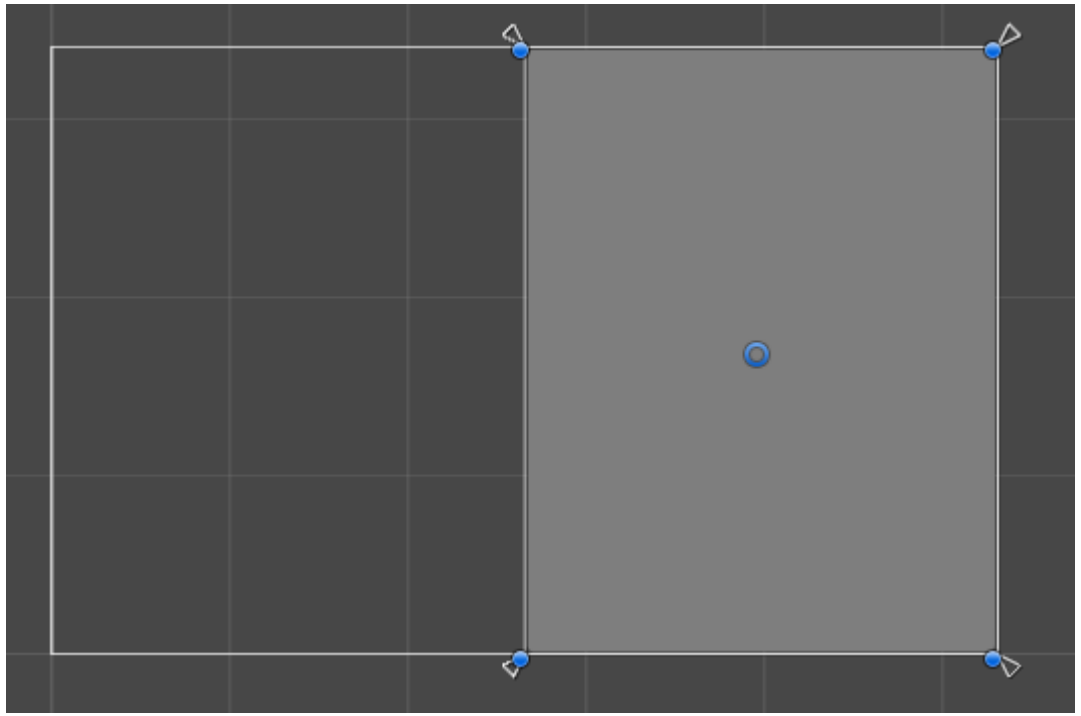
`Left`, `Right`, `Top`, `Bottom`: margen fijo

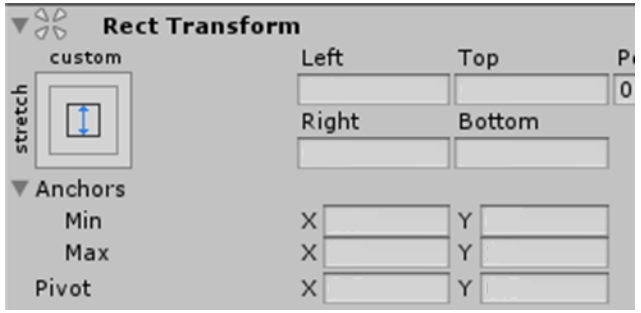
`Pos X`, `Pos Y`: desplazamiento del pivote con respecto a un lado de las anclas

Las unidades siempre son píxeles

Ejercicio

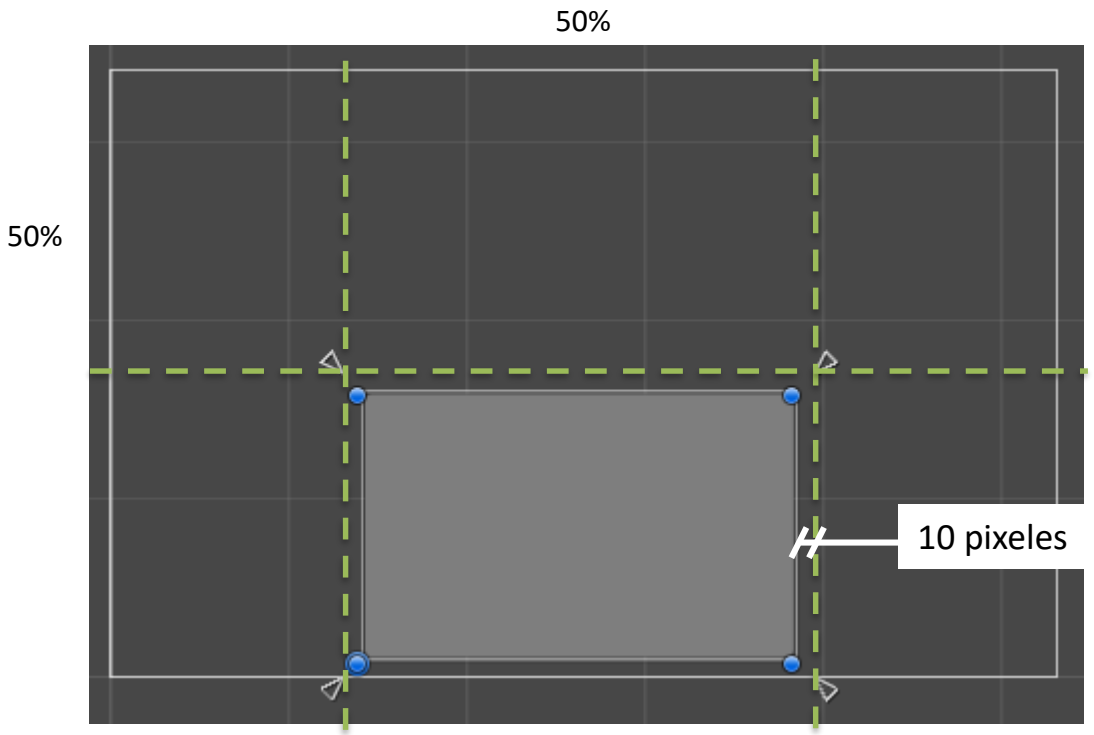
- ¿Qué valores tiene cada campo?

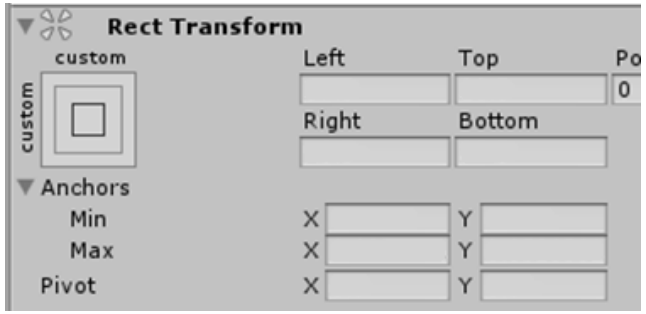




Ejercicio

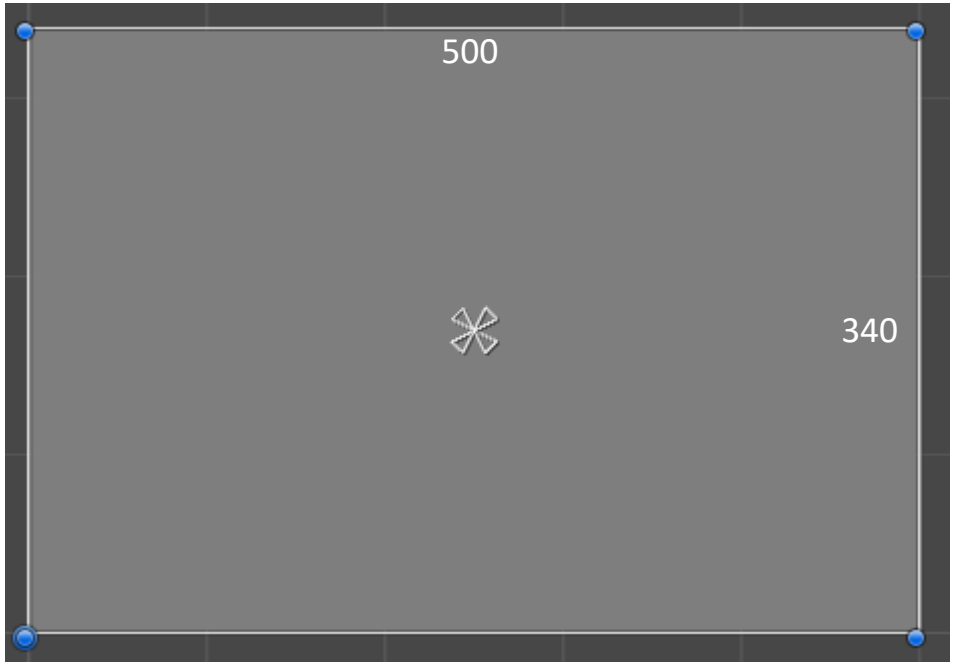
- ¿Qué valores tiene cada campo?

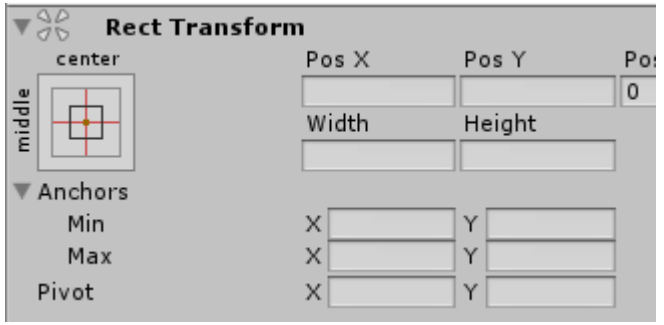




Ejercicio

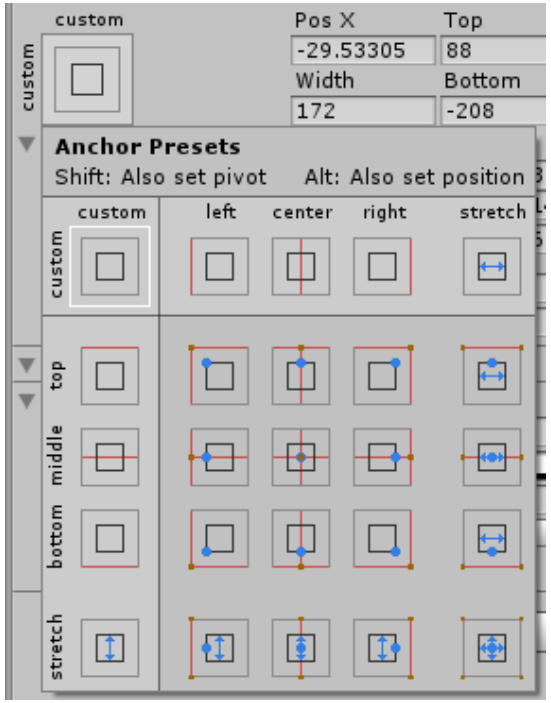
- ¿Dónde están los puntos de ancla y el pivote?



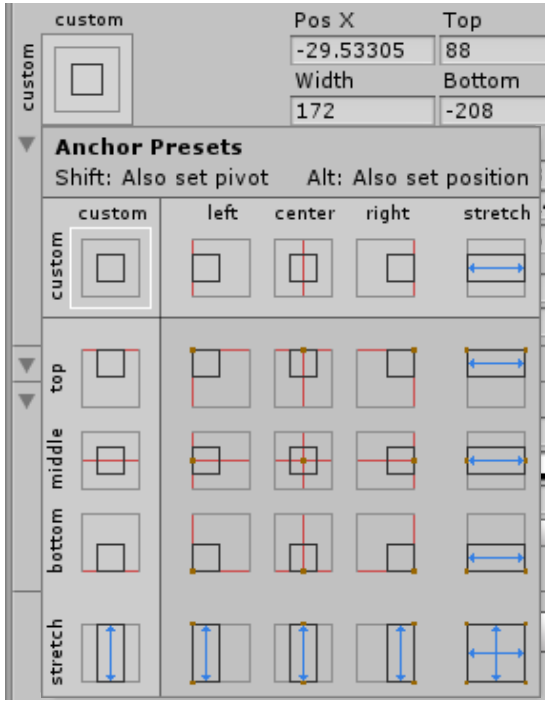


El panel *Rect Transform*

- El botón grande del panel es un atajo para establecer las anclas, el pivote y/o la posición del rectángulo del widget



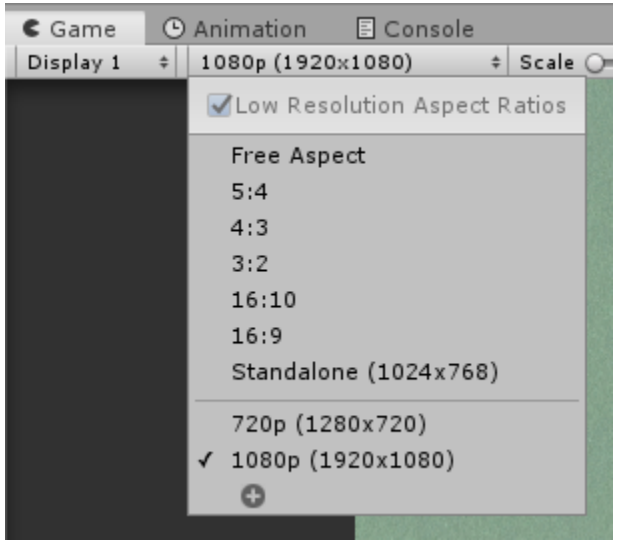
Pulsando Mayúsculas



Pulsando Alt

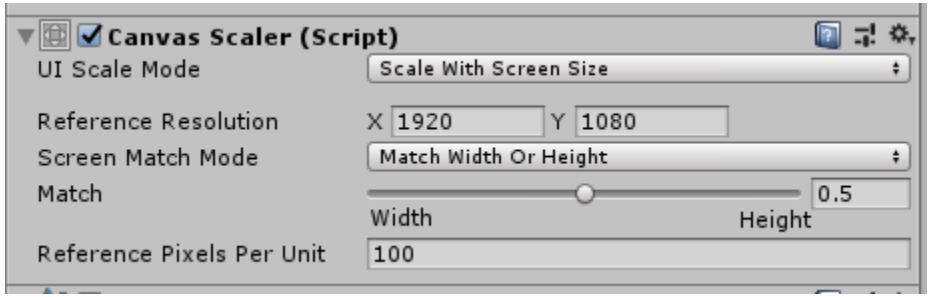
Escalado del Canvas

- Otro detalle a tener en cuenta es cómo se adapta el canvas a distintas resoluciones
- Normalmente, a la hora de diseñar un juego, hay un resolución de referencia (aunque, como ya hemos visto, el juego debería ser jugable en cualquier resolución)
- Añade al panel de la escena las resoluciones más habituales:



Escalado del Canvas

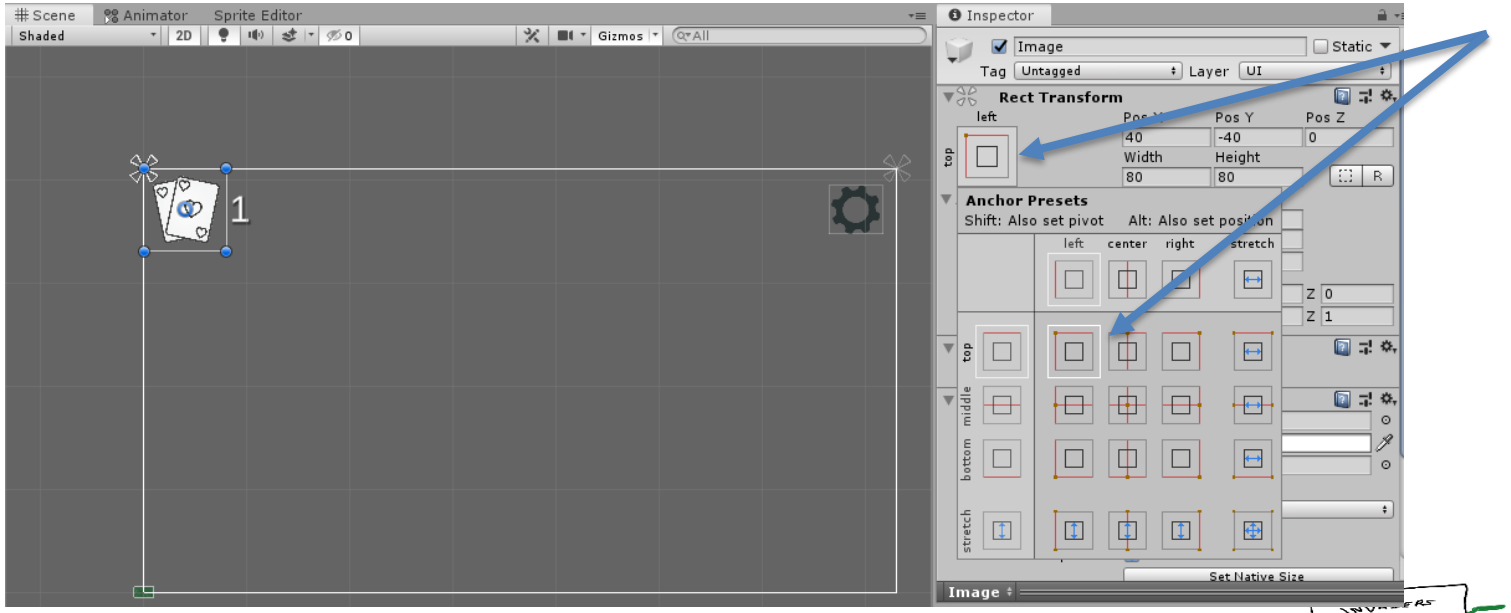
- Selecciona el canvas y selecciona el modo “Scale with Screen Size” en el campo *UI Scale Mode*
- Introduce la resolución de referencia
- Establece el campo Match a 0.5:



- Continúa diseñando la interfaz en la resolución de referencia (y probando frecuentemente la interfaz con distintas resoluciones y razones de aspecto)

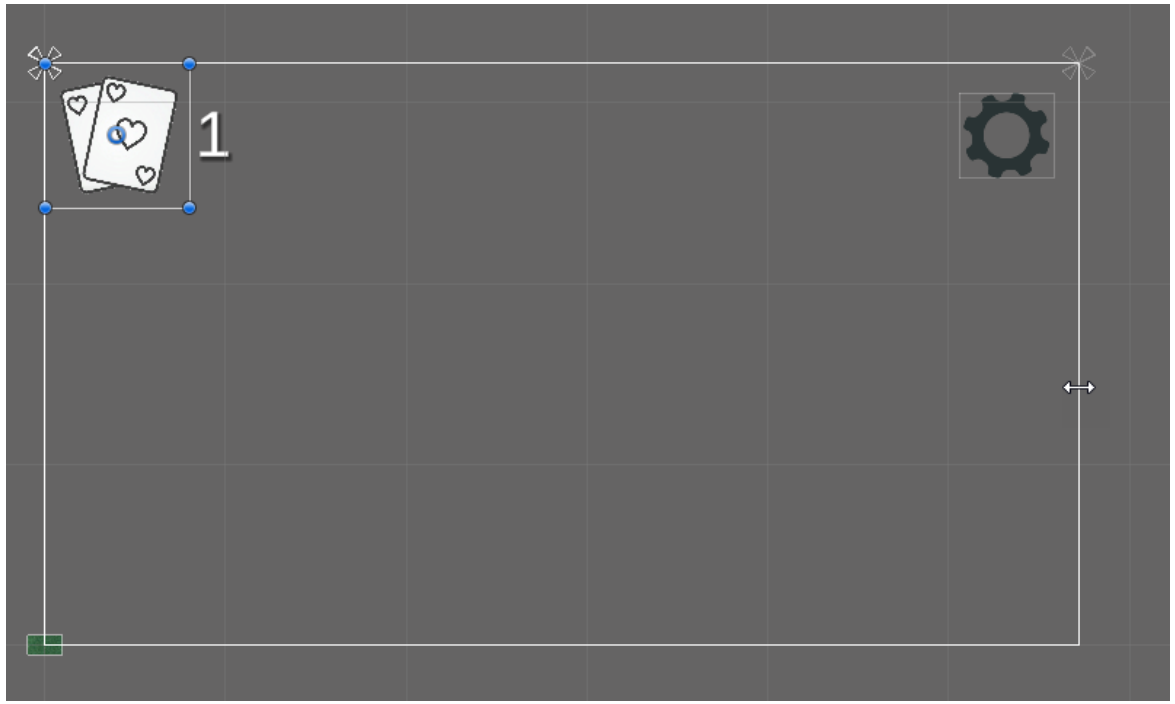
Construyendo la interfaz

- Vamos a hacer que los elementos de la interfaz se ajusten a las esquinas del canvas
- Selecciona la imagen y, en el inspector, haz que el Anchor esté en la esquina superior izquierda
- Haz lo mismo para el botón, pero para la esquina superior derecha



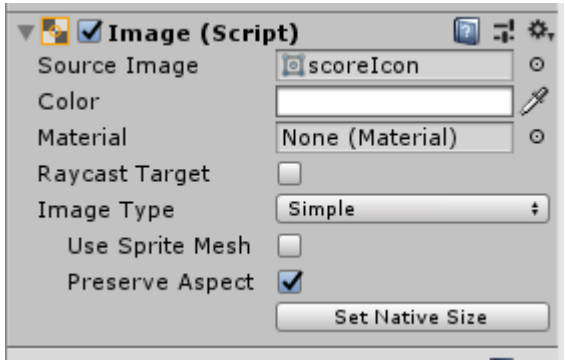
Construyendo la interfaz

- Para ver cómo se ajustan los elementos de la interfaz a los cambios de tamaño, puedes arrastrar los bordes del canvas en la ventana de la escena (al tener seleccionada la imagen):



Interactuando con la interfaz

- Como la imagen y el texto de la puntuación no son interactuables, podemos desactivar la opción Raycast Target para evitar que capturen eventos



Interactuando con la interfaz

- El proceso para añadir una UI a un juego consiste en:
  - Crear un control en la escena (p.e., un botón)
  - Crear un script que se ejecutará al operar el control
  - Incrustar el script a un objeto de la escena
  - Enlazar los objetos necesarios a las propiedades del script

Interactuando con la interfaz

- Vamos a volver a conectar la etiqueta de puntuación con el script de SceneController:

```
using TMPro;

public class SceneController : MonoBehaviour {
   […]
  [SerializeField] private TextMesh scoreLabel;
  [SerializeField] private TextMeshProUGUI scoreLabel;
   […]
  private IEnumerator CheckMatch() {
    if (_firstRevealed.id == _secondRevealed.id) {
      _score++;
      scoreLabel.text = "Score: " + _score;
      scoreLabel.text = _score.ToString();
    } else {
```


Interactuando con la interfaz

- O mejor, añadamos un UIController a GameController:

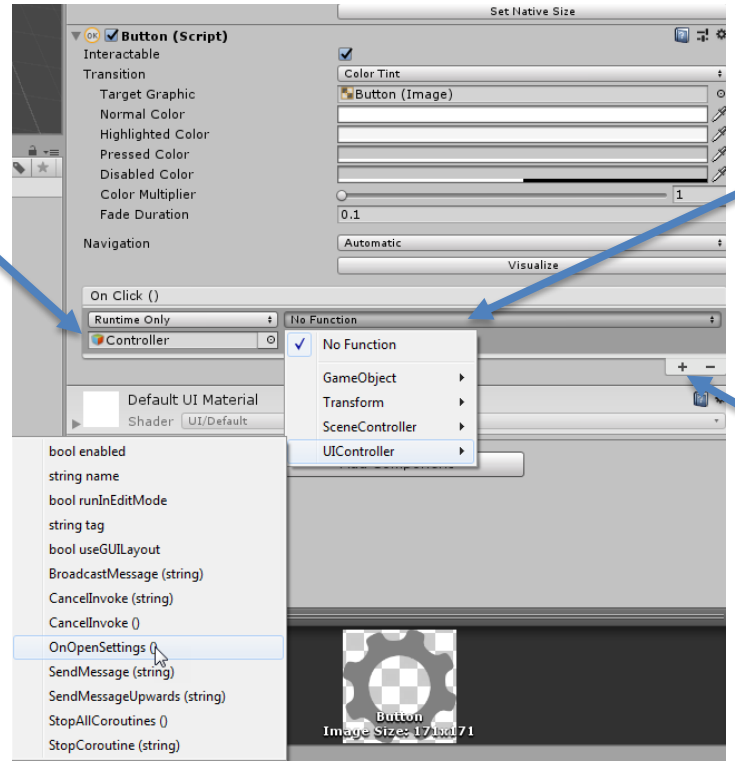
```
using TMPro;
public class UIController : MonoBehaviour {
  [SerializeField] private TextMeshProUGUI scoreLabel;

  public void SetScore(int score) {
    scoreLabel.SetText(score.ToString());
  }
  public void OnOpenSettings() {
    Debug.Log("Open popup");
  }
}
```

```
public class SceneController : MonoBehaviour {
  […]
  private int _score = 0;
  private UIController ui;
  private void Awake() {
    ui = GetComponent<UIController>();
  }
  private IEnumerator CheckMatch() {
    if (_firstRevealed.id == _secondRevealed.id) {
      _score++;
      ui.SetScore(_score);
    } else {
```


Interactuando con la interfaz

- El siguiente paso es hacer que cada vez que se pulse el botón se invoque al método `OnOpenSettings` definido en el script

2. Arrastrar aquí el objeto que contiene el script que se desea ejecutar

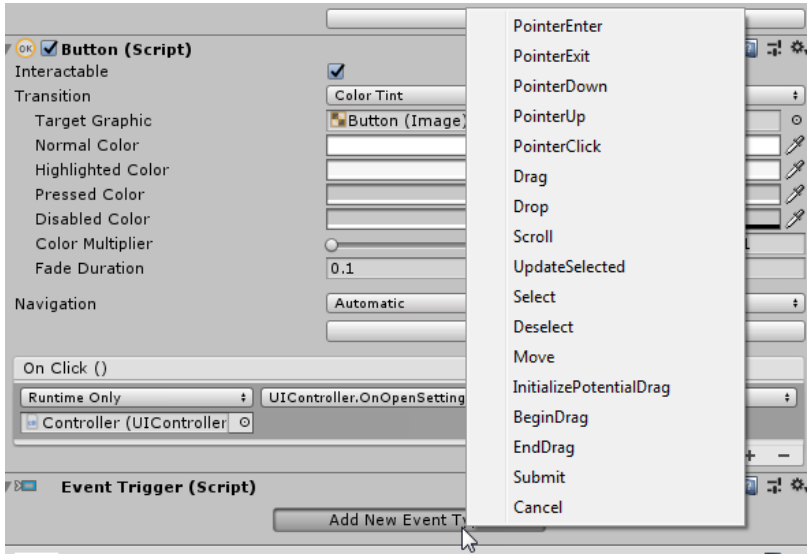


3. Seleccionar aquí la función a ejecutar

1. Pulsar en + para añadir un manejador del evento de clic

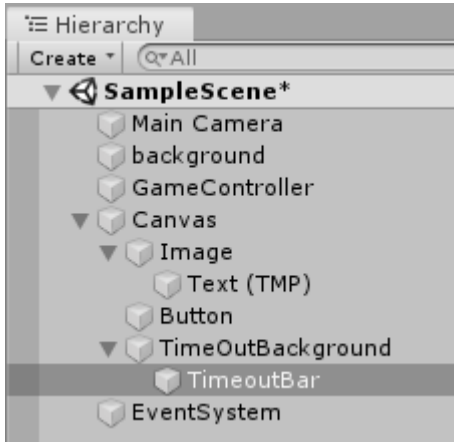
Interactuando con la interfaz

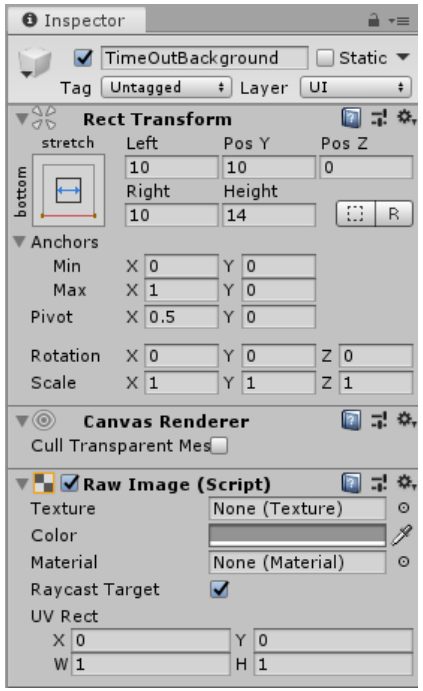
- Si quieres capturar otros eventos, por ejemplo, cuando el puntero del ratón entra en el botón, o cuando se pulsa el botón (sin soltar):
  - Añade un componente `Event Trigger` al control
  - Pulsa en `Add New Event Type` y selecciona el tipo de evento que quieres gestionar
  - A continuación, ya puedes enlazar al evento un método definido en cualquier script asociado con un GameObject de la escena, igual que antes

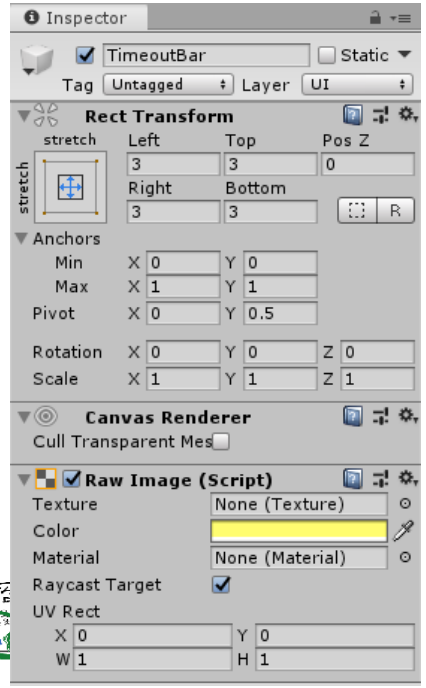


Complicando el juego

- Vamos a añadir una dificultad al juego, en forma de timeout que obliga al jugador a destapar cartas
- Cada x segundos, si el jugador no ha destapado una carta, se restará un punto a la puntuación actual
- Representaremos el tiempo restante con una barra de progreso formada por dos *raw images* superpuestas en la parte inferior de la ventana:







Complicando el juego

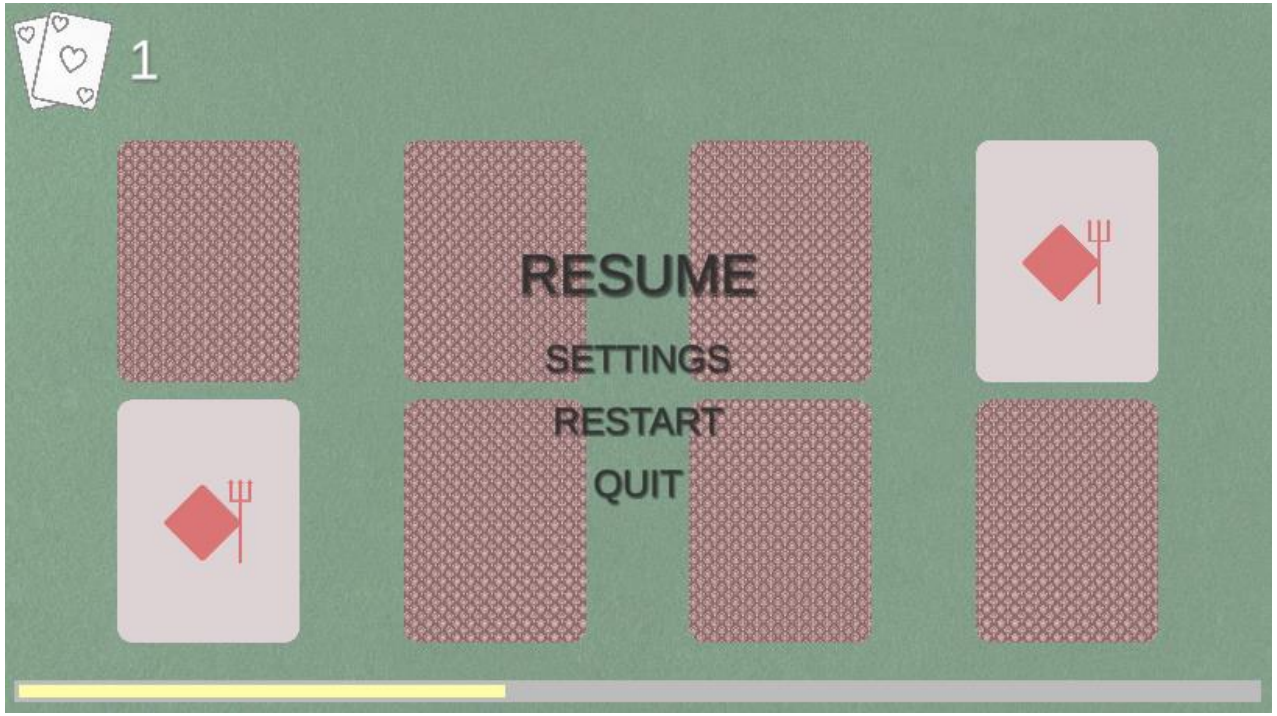
```
using UnityEngine.UI;

public class UIController : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI scoreLabel;
    [SerializeField] private RawImage barImg;

    public void SetTimeOutValue(float timeout)
    {
        barImg.transform.localScale = new Vector3(timeout, 1.0f);
    }

    public void EnableTimeOutBar(bool enable)
    {
        barImg.gameObject.SetActive(enable);
    }
}
```

```
public class SceneController : MonoBehaviour {
  private int _score = 0, _pairsFound = 0;
  public float timeOutSeconds = 3.0f;
  float currentTime;

  private bool _paused;
  public bool paused {
    get { return _paused; }
    set { _paused = value;
          Time.timeScale = value ? 0.0f : 1.0f; }
  }
  public bool gameover {
    get { return _pairsFound ==
           gridCols * gridRows / 2; }
  }
  void Update() {
    currentTime -= Time.deltaTime;
    if (currentTime < 0) {
      if (_score > 0) {
        _score--; ui.SetScore(_score);
      }
      ResetTimeOut();
    }
    ui.SetTimeOutValue(currentTime / timeOutSeconds);
  }
```

```
public bool canReveal {
  get { return _secondRevealed == null && !paused; }
}
void ResetTimeOut() {
  currentTime = timeOutSeconds;
}

public void CardRevealed(MemoryCard c) {
  ResetTimeOut();
  if (_firstRevealed == null) {
  […]
}

private IEnumerator CheckMatch() {
  if (_firstRevealed.id == _secondRevealed.id) {
    _score++;
    ui.SetScore(_score);
    _pairsFound++;
    if (gameover) {
      ui.EnableTimeOutBar(false);
      paused = true;
    }
  } else {
```

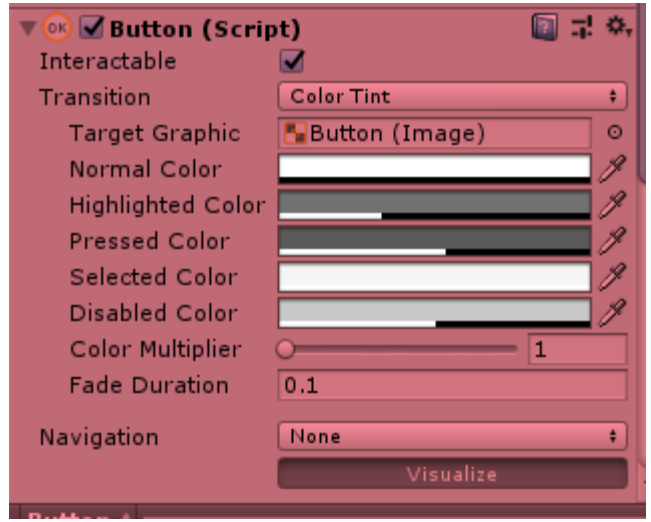

Menú de pausa

- Ya que hemos implementado la funcionalidad de pausa, vamos a mostrar al usuario que el juego está pausado, y mostrar un botón para reanudar el juego



Menú de pausa

- Añade un nuevo Canvas (puedes llamarlo PausedCanvas)
  - En el componente Canvas, establece el campo Sort Order a 2 para que aparezca por encima del resto de canvas
- Añade un panel a este canvas
  - Por defecto aparece de color blanco semitransparente
  - Deja Source Image a None
- Añade los botones
  - Ajusta sus propiedades. Por ejemplo:



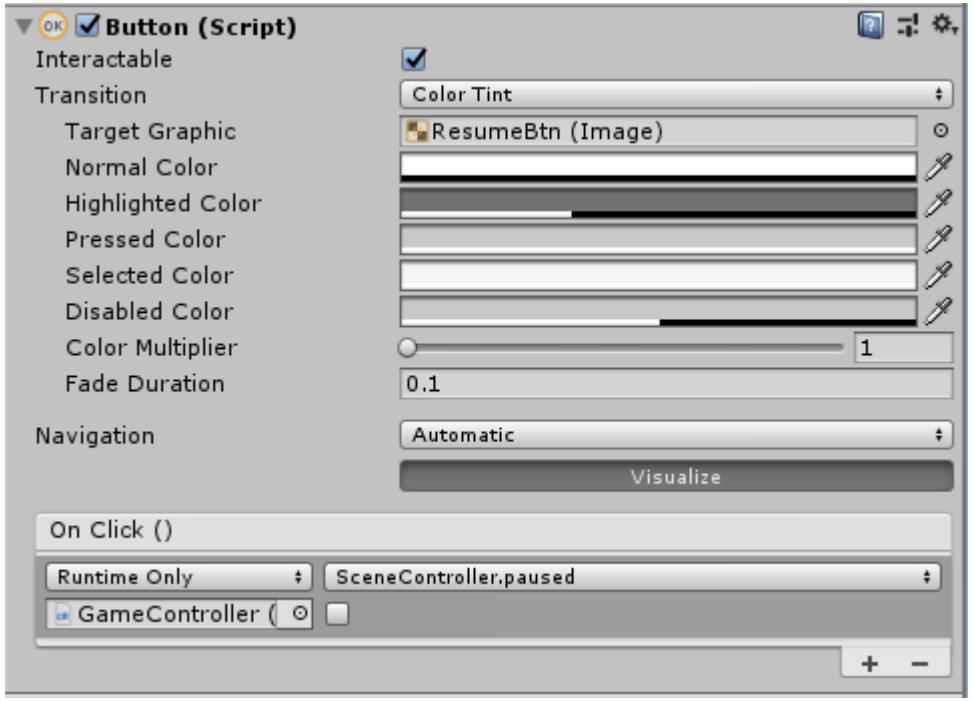
Normal

Highlighted

Pressed

Menú de pausa

- El botón de reanudar tiene que poner a false la propiedad “paused” de SceneController



Menú de pausa

- Vamos a añadir la funcionalidad de abrir y cerrar el menú de pausa

```
public class UIController : MonoBehaviour
{
  [SerializeField] private TextMeshProUGUI scoreLabel;
  [SerializeField] private RawImage barImg;
  [SerializeField] private SettingsPopup settingsPopup;
  [SerializeField] private GameObject pauseMenu;

  public void ShowPauseMenu(bool isPaused)
  {
    pauseMenu.SetActive(isPaused);
  }
}
```

Menú de pausa

- Y en SceneController:

```
public class SceneController : MonoBehaviour {

  private UIController ui;

  private bool _paused;
  public bool paused {
     get { return _paused; }
     set { _paused = value; Time.timeScale = value ? 0.0f : 1.0f; ui.ShowPauseMenu(value); }
  }
  void Update() {
    if (Input.GetKeyDown(KeyCode.Escape)) {
      paused = !paused;
    }
    if (paused) return;
    currentTime -= Time.deltaTime;
    if (currentTime < 0)
```


Ejercicio

- El siguiente código carga la escena indicada. Se puede utilizar para reiniciar el juego. Haz que el botón de reinicio funcione

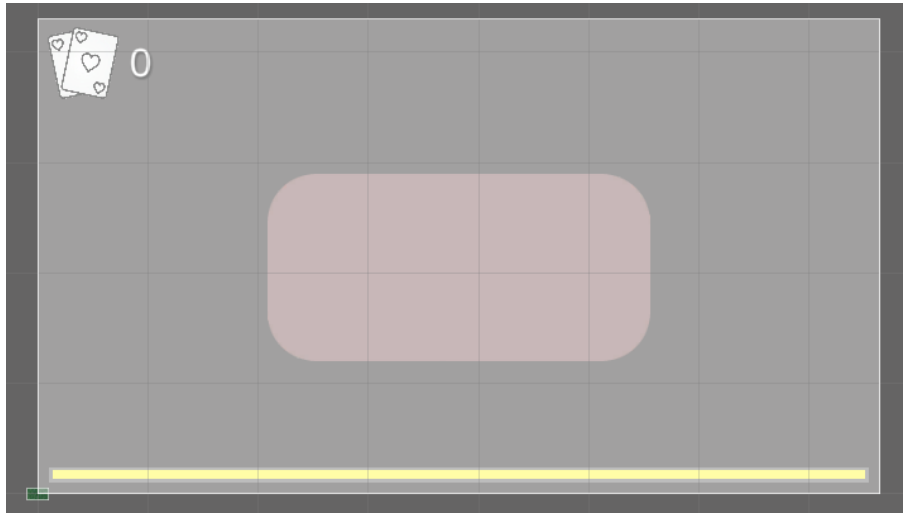
```
using UnityEngine.SceneManagement;
[…]
SceneManager.LoadScene("SampleScene");
```

- El siguiente código termina el juego (y sale del modo de juego si estamos dentro de Unity). Conéctalo al botón correspondiente del menú de pausa

```
#if UNITY_EDITOR
  UnityEditor.EditorApplication.isPlaying = false;
#else
  Application.Quit();
#endif
```

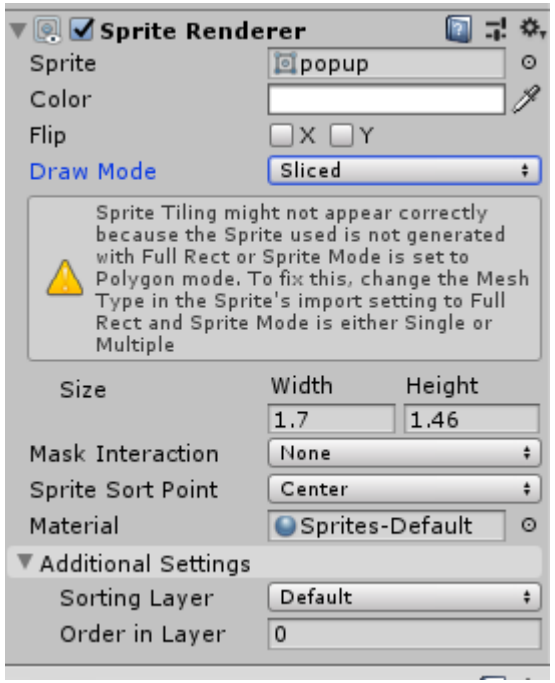
Abriendo un pop-up

- Las opciones del juego se encuentran en una ventana secundaria, que se tiene que abrir al pulsar el botón correspondiente
- El popup está formado por una imagen, que servirá de contenedor para el resto de controles
- Vamos a cubrir los botones de pausa con el popup
- Crea una imagen y asígnale el sprite correspondiente



Abriendo un pop-up

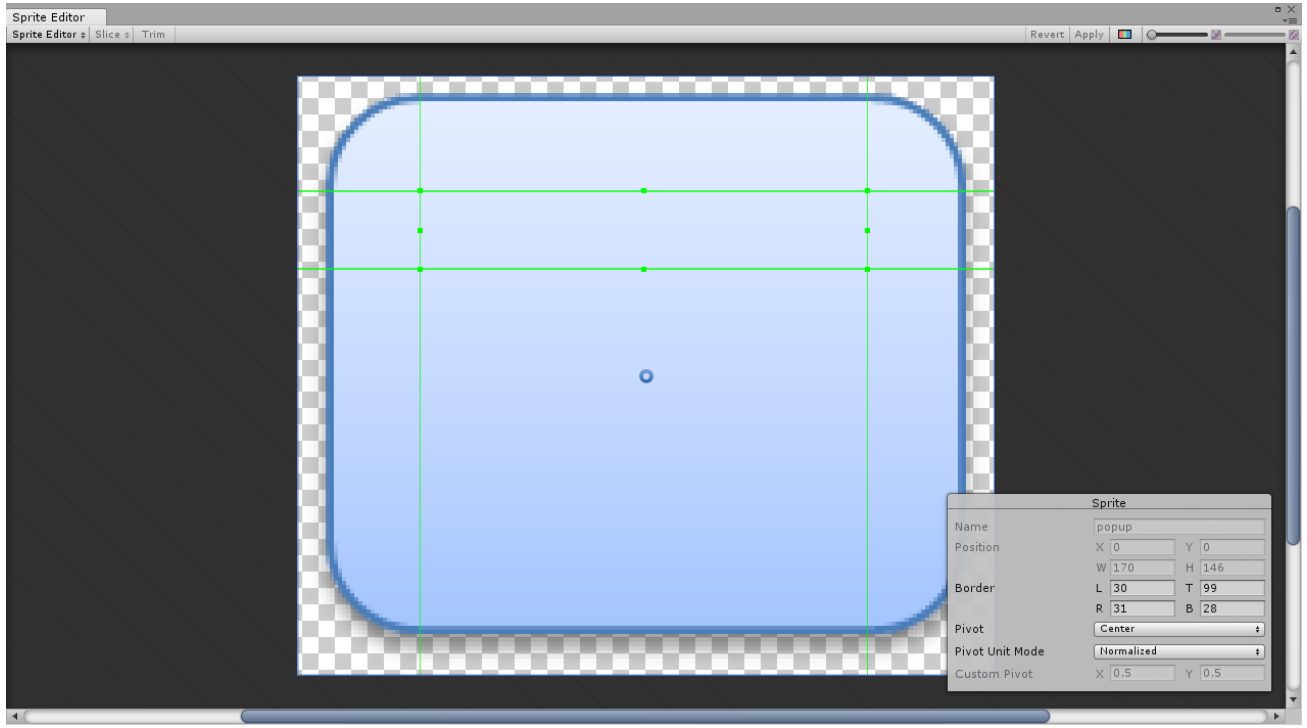
- Sprites troceados
  - El comportamiento del widget imagen cuando se cambia de tamaño es escalar el sprite asociado
  - Esto implica que si la imagen debe crecer mucho, los sprites deben ser de alta resolución para mantener la calidad
  - Sin embargo, se pude definir que el sprite asociado es de tipo sliced:



El error indica que al importar el sprite no se indicado no se ha indicado que es de tipo Full Rect

Abriendo un pop-up

- Para dividir un sprite hay que seleccionarlo en el panel Project y abrir el Sprite Editor



Sliced

Simple

Abriendo un pop-up

- El orden en el que se dibujan los objetos del canvas (y, por tanto, en el que se solapan) es el orden en el que aparecen en la jerarquía
- Por ello, asegúrate de que el popup aparece el último, para que al moverlo aparezca por encima del resto de controles
- Vamos a crear un script para el popup:

```
public class SettingsPopup : MonoBehaviour {
  public void Open() {
    gameObject.SetActive(true);
  }
  public void Close() {
    gameObject.SetActive(false);
  }
}
```


Abriendo un pop-up

- Y ahora en el controlador de la UI, vamos a cerrar y abrir el popup:

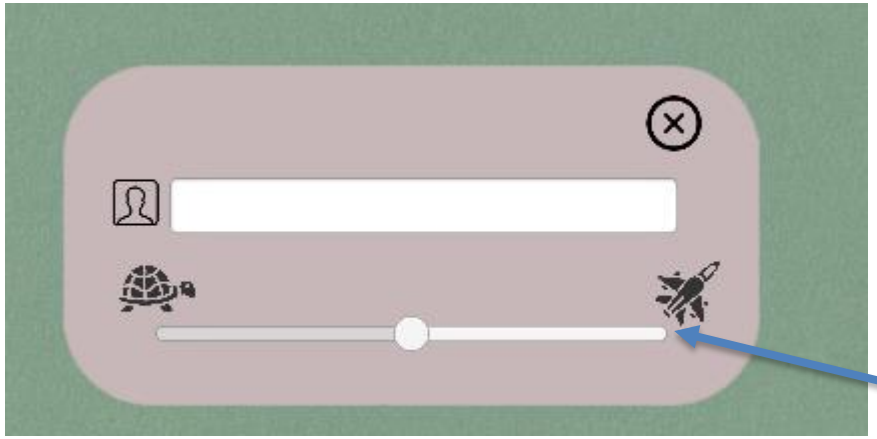
```
[SerializeField] private SettingsPopup settingsPopup;
void Start() {
  settingsPopup.Close();
}

public void OnOpenSettings() {
  settingsPopup.Open();
}
```

Ejercicio

- Conecta el botón correspondiente del menú de pausa con el código anterior para abrir el popup.
- Añade el botón y la funcionalidad para cerrar el popup

Completando el popup

- Vamos a añadir el resto de controles del popup
- Añade un `InputField TextMeshPro` y un `Slider` como hijos del popup
- Configúralos como se muestra en la imagen:



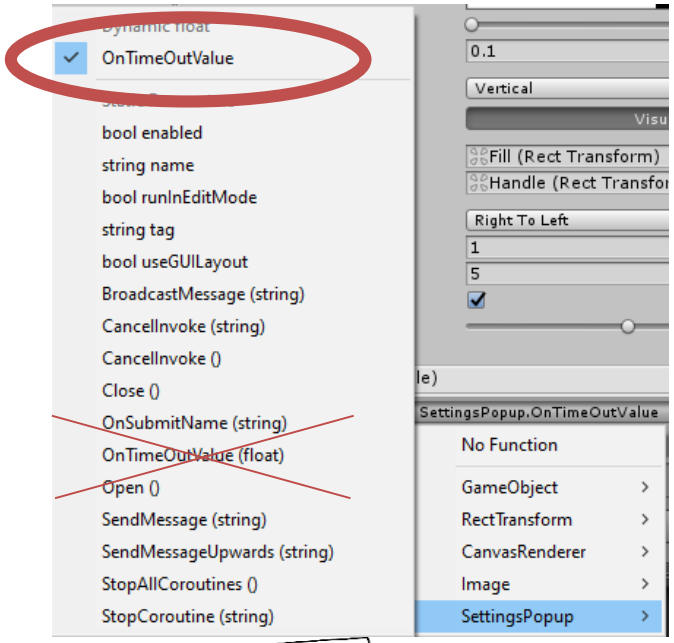
Valores enteros, mínimo 1, máximo 5, valor 3, dirección de derecha a izquierda

Completando el popup

- Añade el siguiente código a SettingsPopup:

```
public void OnSubmitName(string name) {
  Debug.Log(name);
}

public void OnSpeedValue(float speed) {
  Debug.Log("Speed: " + speed);
}
```

- Haz que el slider llame al método `OnSpeedValue` cuando cambie su valor.
- Ídem para el evento `On End Edit` del InputText para el método `OnSubmitName`



Guardando información entre ejecuciones

- Unity ofrece una forma sencilla de almacenar información entre distintas ejecuciones
  - Cómo lo hace es transparente, y depende de la plataforma
- La clase encargada es PlayerPrefs:

Static Methods

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.
HasKey	Returns true if key exists in the preferences.
Save	Writes all modified preferences to disk.
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.

https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

Guardando información entre ejecuciones

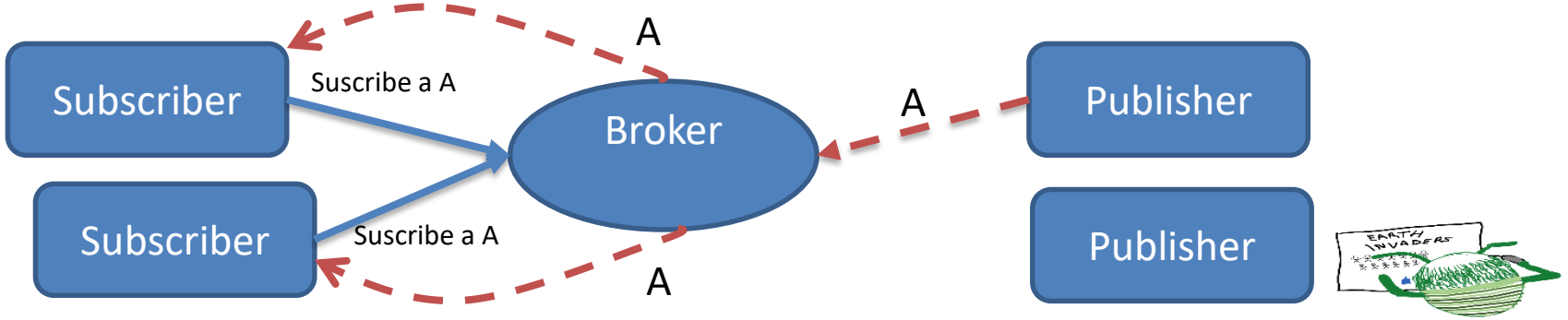
- Por ejemplo, para almacenar la configuración de la velocidad definida por el slider:

```
using UnityEngine.UI;

public class SettingsPopup : MonoBehaviour {
  [SerializeField] private Slider timeOutSlider;

  private void Start() {
    timeOutSlider.value = PlayerPrefs.GetFloat("timeout", 3f);
  }
  public void OnTimeOutValue(float timeout) {
    Debug.Log("Timeout: " + timeout);
    PlayerPrefs.SetFloat("timeout", timeout);
  }
```

Valor por defecto

Conectando el GUI con el juego

- Por el momento la UI sólo lanza mensajes de depuración. Vamos a conectarla con el resto del juego
- Una opción sería a través de scripts y referencias a GameObjects, tal y como hemos hecho hasta ahora
  - Problema: el juego debe conocer cómo funciona la UI y viceversa
  - Solución: usar el patrón de publicación-subscripción, que desacopla módulos de un sistema mediante el paso de mensajes:
    - Hay una serie de mensajes predefinidos (el jugador ha recibido 10 puntos de daño, ha explotado una bomba, etc.)
    - Los scripts interesados en cierto mensaje se pueden subscribir a través de un servidor (bróker) de mensajes
    - Otros scripts generan mensajes que mandan al bróker, y este los redistribuye a los subscriptores correspondientes



Conectando el GUI con el juego

- Vamos a utilizar una librería externa que implementa un sistema de paso de mensajes para Unity:
  - Crear un script llamado Messenger.cs
  - Copia el código correspondiente de la página http://wiki.unity3d.com/index.php/CSharpMessenger_Extended
  - Crea un script llamado GameEvent, donde se definirán los tipos de eventos que puede generar el juego:

```
public class GameEvent {
  public const string PAUSED = "PAUSED";
  public const string RESUMED = "RESUMED";
  public const string TIMEOUT_CHANGED = "TIMEOUT_CHANGED";
}
```

Conectando el GUI con el juego

- Vamos a empezar gestionando la pausa mediante mensajes:

```
public class UIController : MonoBehaviour {
  […]
  private void Awake() {
    Messenger.AddListener(GameEvent.PAUSED, OnPaused);
    Messenger.AddListener(GameEvent.RESUMED, OnResumed);
  }
  private void OnDestroy() {
    Messenger.RemoveListener(GameEvent.PAUSED, OnPaused);
    Messenger.RemoveListener(GameEvent.RESUMED, OnResumed);
  }
  void OnPaused() {
    ShowPauseMenu(true);
  }
  void OnResumed() {
    ShowPauseMenu(false);
  }
```

Conectando el GUI con el juego

- Y para generar el mensaje, dentro de SceneController:

```
public bool paused {
  get { return _paused; }
  set { _paused = value; Time.timeScale = value ? 0.0f : 1.0f;
        Messenger.Broadcast(value ? GameEvent.PAUSED : GameEvent.RESUMED);
  }
}
```

Conectando el GUI con el juego

- Ahora vamos a hacer que el UI afecte al juego, en este caso el ajuste de la velocidad del juego. En SceneController:

```
private void Awake() {
  ui = GetComponent<UIController>();
  Messenger<float>.AddListener(GameEvent.TIMEOUT_CHANGED, OnTimeOutChanged);
  timeOutSeconds = PlayerPrefs.GetFloat("timeout");
}

private void OnDestroy() {
  Messenger<float>.RemoveListener(GameEvent.TIMEOUT_CHANGED, OnTimeOutChanged);
}

void OnTimeOutChanged(float timeout) {
  timeOutSeconds = timeout;
  ResetTimeOut();
}
```

Conectando el GUI con el juego

- Y para emitir el mensaje, en el manejador del slider en SettingsPopup.cs:

```
public void OnTimeOutValue(float timeout) {
PlayerPrefs.SetFloat("timeout", timeout);
  Messenger<float>.Broadcast(GameEvent.TIMEOUT_CHANGED, timeout);
}
```

- Prueba el juego y asegúrate de que puedes cambiar el tiempo de timeout

Ejercicios

- Implementa una ventana para cuando el jugador termina una partida al destapar todas las cartas (puedes reutilizar parte de la ventana de pausa)
- Haz que se muestre en la parte superior de la pantalla el nombre del jugador con la mayor puntuación (haz que se pueda acumular la puntuación de cada partida mientras que no se salga del juego)

Bibliografía

- Joseph Hocking. Unity in Action. 2nd edition. Manning, 2018.
  - Capítulo 7